ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

Effetti Visivi Avanzati in CGI: Studio di Compute Shaders e Sistemi di Particelle con Applicazioni a Simulazioni Dinamiche

Elaborato in

Computer Graphics

Relatore

Prof.ssa Damiana Lazzaro

Presentata da

Federico Brighi

Sessione Unica
Anno Accademico 2024/2025

“ Sono un insicuro non accetto me,
senza dimostrare più a nessuno,
frate, eccetto a me”

Marracash, Body Parts – I Denti

Indice

Introduzione

Il presente elaborato nasce dalla mia passione personale per il cinema e dal profondo interesse nel comprendere le tecniche e i processi alla base della creazione degli effetti speciali, elementi ormai centrali sia per la narrazione sia per l'impatto visivo delle produzioni cinematografiche moderne. Sin dai primi anni di studio, ho sempre trovato affascinante osservare come la fusione tra arte e tecnologia consenta di dar vita a mondi e situazioni che altrimenti sarebbero irrealizzabili. Questa curiosità è stata ulteriormente alimentata dal corso di Computer Graphics che ho seguito, grazie al quale ho potuto acquisire conoscenze teoriche e competenze pratiche sui fondamenti della grafica digitale. Motivato da questo interesse, ho chiesto alla professoressa Lazzaro di assumere il ruolo di relatrice per questo progetto, il quale rappresenta per me un'occasione preziosa per approfondire e mettere in pratica quanto appreso, unendo la teoria con lo sviluppo concreto di un effetto speciale.

Il panorama cinematografico e tecnologico contemporaneo è caratterizzato da un impiego sempre più diffuso di strumenti digitali avanzati, che hanno reso gli effetti speciali una componente imprescindibile per la creazione di esperienze visive immersive e coinvolgenti.
In particolare, le tecniche di *rendering* in tempo reale e l'utilizzo di *shader* programmabili, come i *compute shader*, rivestono un ruolo centrale nell'ottimizzazione dell'efficienza e della qualità degli effetti particellari, mantenendo elevate prestazioni anche in contesti interattivi.
Tale evoluzione tecnologica rappresenta un punto di incontro tra la teoria della computer grafica e le sue applicazioni pratiche nei settori del cinema, dell'animazione e dei videogiochi. L'adozione di queste soluzioni consente di simulare fenomeni naturali e dinamici con un livello di realismo senza precedenti, aprendo nuove prospettive per la produzione di contenuti audiovisivi.

L'obiettivo principale dell'elaborato è lo sviluppo di un simulatore interattivo di distruzione architettonica basato su OpenGL, che integra modellazione 3D, animazione di personaggi e simulazione fisica avanzata. Il progetto prevede la realizzazione di una scena tridimensionale complessa, popolata da modelli architettonici in formato OBJ importati insieme ai loro materiali MTL, all'interno della quale un personaggio animato, caricato da modello FBX con supporto per animazione scheletrica, agisce come elemento scatenante per gli eventi distruttivi.

L'interazione si basa sul rilevamento delle collisioni tra il personaggio e gli elementi architettonici, meccanismo che innesca automaticamente la transizione dalla loro forma integra ad una versione frammentata, ottenuta tramite l'add-on *cell fracture* di Blender. La peculiarità tecnica del progetto risiede nell'implementazione di un sistema fisico dual-mode: una versione CPU multi-threaded tradizionale e una versione GPU che sfrutta i *compute shader* di OpenGL 4.3+ per il calcolo parallelo delle dinamiche di esplosione. Entrambe le implementazioni gestiscono

in *real-time* forze gravitazionali, dispersione esplosiva radiale, attrito aerodinamico e collisioni tra piano e frammenti, permettendo un confronto prestazionale diretto tra architetture di calcolo CPU e GPU per la simulazione fisica di sistemi particellari complessi.

Il lavoro è organizzato in cinque capitoli principali, ognuno dei quali affronta un aspetto specifico del progetto:

Il primo capitolo introduce il contesto cinematografico e i principi fondamentali degli effetti visivi e della simulazione particellare, fornendo un quadro teorico essenziale per comprendere le motivazioni e le tecniche alla base del progetto.
Il secondo capitolo si focalizza sui *compute shader* in OpenGL, presentandone i fondamenti, l'architettura e le tecniche di ottimizzazione impiegate, con un'analisi dettagliata delle problematiche hardware e delle operazioni di sincronizzazione necessarie per la gestione di sistemi complessi.
Il terzo capitolo affronta il background matematico degli effetti particellari, illustrando le formulazioni teoriche e le equazioni che guidano il comportamento delle particelle nelle simulazioni.
Nel quarto capitolo viene descritta la fase di sviluppo e implementazione del simulatore, con documentazione delle scelte progettuali, struttura del codice, integrazione degli shader, analisi dei risultati ottenuti e performance del sistema, valutando l'efficacia dell'approccio e individuando eventuali margini di miglioramento.
Infine, il quinto capitolo riporta le conclusioni tratte dal lavoro svolto, evidenziando le prospettive future e le possibili estensioni del progetto, con uno sguardo anche all'utilizzo dell'intelligenza artificiale nel cinema.

Capitolo 1

Effetti visivi nel cinema e simulazione particellare

Il seguente capitolo è dedicato all'approfondimento teorico dei **sistemi particellari**, una delle tecniche più diffuse e versatili nell'ambito della computer grafica. Dopo una descrizione generale del concetto di particella e delle sue proprietà fondamentali, verranno analizzati i modi in cui tali entità vengono impiegate per simulare fenomeni complessi della realtà, come fumo, fuoco, esplosioni, pioggia o effetti astratti di tipo visivo. I sistemi particellari, infatti, costituiscono la base di numerosi algoritmi utilizzati nei software professionali per la produzione di contenuti audiovisivi, dal cinema all'animazione digitale, fino al settore dei videogiochi.

Una sezione specifica è dedicata all'**evoluzione storica della computer grafica**, con particolare attenzione al ruolo che i sistemi particellari e le tecniche di rendering hanno avuto nello sviluppo degli effetti speciali. Verranno ripercorse le tappe principali che hanno segnato la transizione dai primi esperimenti in grafica 2D alle simulazioni tridimensionali sempre più sofisticate, capaci di integrare in modo realistico personaggi virtuali e ambienti digitali con riprese dal vivo.

All'interno del capitolo sarà inoltre fornita una panoramica delle **procedure di modellazione** adottate nella creazione di personaggi, illustrando i passaggi fondamentali dalla realizzazione di mesh e scheletri fino al rigging e all'animazione. Infine, verranno presentati alcuni esempi concreti di effetti speciali che hanno segnato la storia del cinema e dell'animazione, evidenziando come l'impiego combinato di particelle, shader e tecniche di rendering avanzate abbia permesso di raggiungere livelli sempre più elevati di realismo e spettacolarità.

1.1 Sistema particellare

Un **sistema particellare** (particle system) è una tecnica della computer grafica che utilizza un vasto numero di piccolissime sprites, modelli 3D o oggetti grafici per simulare fenomeni

complessi e ”fuzzy” difficili da riprodurre con tecniche di rendering tradizionali, come sistemi caotici, fenomeni naturali o reazioni chimiche. La prima applicazione di sistemi particellari nel cinema risale al 1984, quando William Reeves li utilizzò per simulare un muro di fuoco nel film *Star Trek II: L'ira di Khan*.

Questi sistemi, basati su una modellazione volumetrica, sono particolarmente adatti per la riproduzione di fenomeni naturali volumetrici come **fuoco**, **acqua**, **neve** e **nuvole**.
Con il tempo si sono evoluti fino a simulare effetti sempre più complessi, come **esplosioni**, **scintille**, **movimenti dell'acqua**, **bagliori** e persino **elementi spaziali**, nei quali le particelle vengono continuamente generate, animate e fatte svanire per poi essere riemesse dalla sorgente dell'effetto. Tecniche avanzate sono impiegate anche per simulare **fasci di capelli**, **fili d'erba** e altri **dettagli organici**.

Si tratta di una tecnica di **modellazione procedurale**: si parte da un'ampia collezione di particelle geometriche elementari che cambiano stocasticamente nel tempo. Per rappresentare oggetti naturali si fa uso di un ampio database di primitive geometriche; tuttavia, l'animazione, che comprende nascita, movimento e morte delle particelle, è controllata da algoritmi che agiscono attraverso un numero limitato di parametri di controllo.

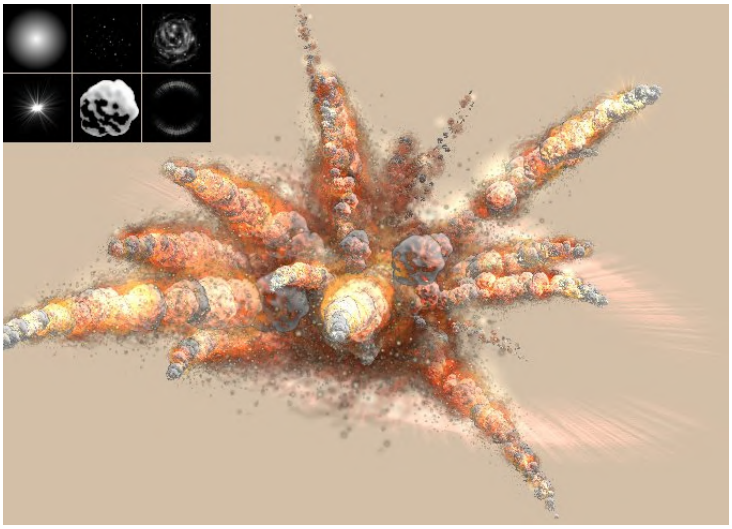

Figura 1.1: Sistema particellare complesso generato da un emitter, con particelle animate e texture visibili nel ciclo di vita.

Una **particella** è un elemento visibile solo durante il suo ciclo di vita, i cui attributi possono includere **forma** (bitmap o modello 3D), **colore** o **texture superficiale**, **dimensione**, **massa**, **durata**, **velocità**, specificati come valori fissi o intervalli di variabilità. La posizione iniziale e il movimento della particella nello spazio sono controllati da un **emitter**, una sorgente invisibile la cui posizione determina il punto di generazione e la direzione del moto (ad esempio una mesh

3D). L'emitter possiede un set di parametri relativi al comportamento delle particelle, tra cui il **tasso di generazione (spawning rate)**, la **velocità iniziale**, la **durata della vita** o il **colore**.[8]

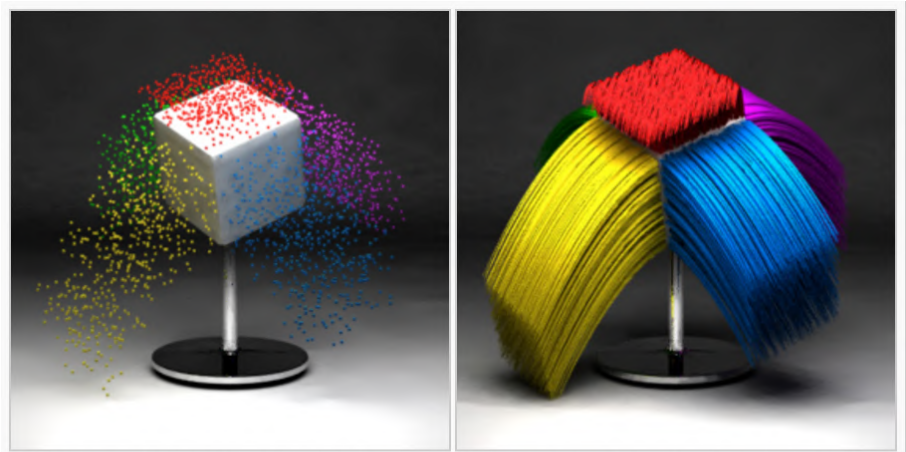

Figura 1.2: Confronto tra particelle dinamiche (sinistra) influenzate dalla gravità e particelle statiche a filamento (destra) emesse dallo stesso cubo.

Tra i principali software impiegati nella modellazione e simulazione di sistemi particellari spiccano strumenti altamente specializzati come:

- **Houdini** [1], sviluppato da SideFX, considerato uno standard industriale per la creazione di effetti visivi basati su sistemi particellari. Grazie al suo approccio procedurale e alla capacità di gestire simulazioni dinamiche di fluidi, fuoco, fumo e altro, offre un controllo granulare sugli aspetti fisici e grafici. La sua flessibilità lo rende lo strumento preferito nei grandi studi di VFX per cinema e televisione.

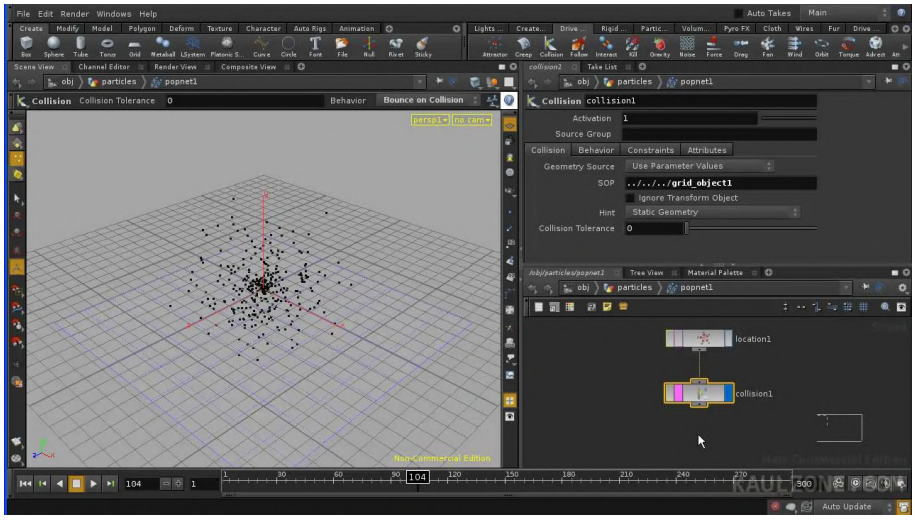

Figura 1.3: Schermata di esempio del software Houdini.

- **Maya** [2], prodotto da Autodesk, noto soprattutto per le sue capacità di modellazione e animazione 3D, integra potenti plugin e moduli dedicati alla simulazione particellare, permettendo così di generare effetti complessi integrandoli agevolmente con pipeline di produzione digitali ampie e articolate.

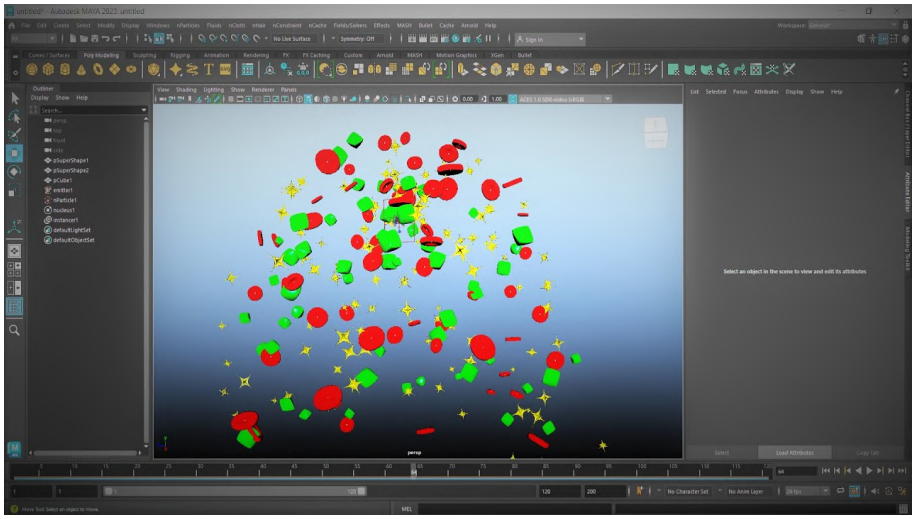

Figura 1.4: Schermata di esempio del software Maya.

- **Blender** [3], software open source in rapida crescita, che ha consolidato una suite di strumenti dedicati alla simulazione particellare, inclusi sistemi di particelle tradizionali, simulazioni di fluidi e fumo. Consente a un'ampia comunità di artisti di realizzare effetti visivi di qualità professionale senza costi di licenza, facilitando l'accesso alla tecnologia anche a chi si avvicina per la prima volta alla computer grafica (Blender è stato utilizzato per la realizzazione del progetto di questa tesi, in particolare il suo add-on "cell-fracture").

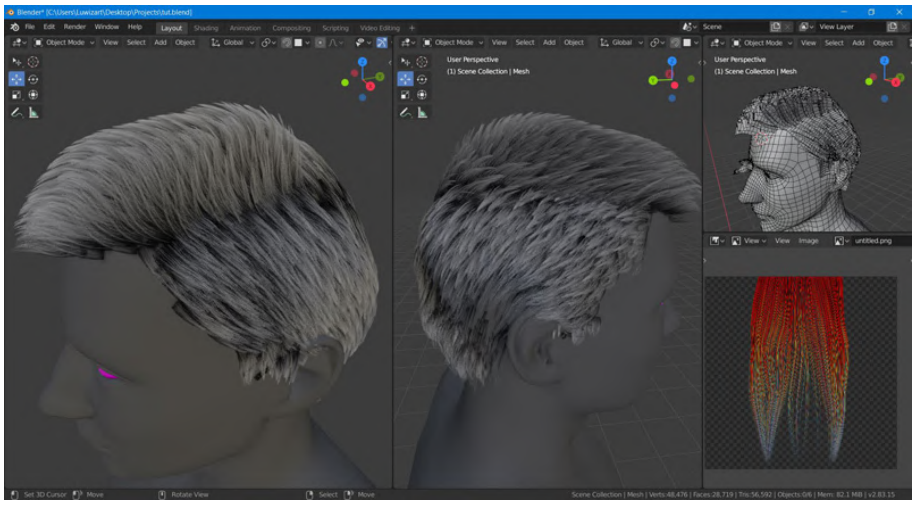

Figura 1.5: Schermata di esempio del software Blender.

- **Unreal Engine** [4], inizialmente conosciuto come motore di gioco, ha esteso le sue funzionalità includendo sistemi di particelle avanzati come *Niagara*, che permettono di gene-

rare simulazioni particellari in tempo reale. Questo aspetto è fondamentale per la produzione di contenuti interattivi, realtà virtuale e visualizzazioni immersive, aprendo nuove possibilità anche per il cinema e la produzione audiovisiva.

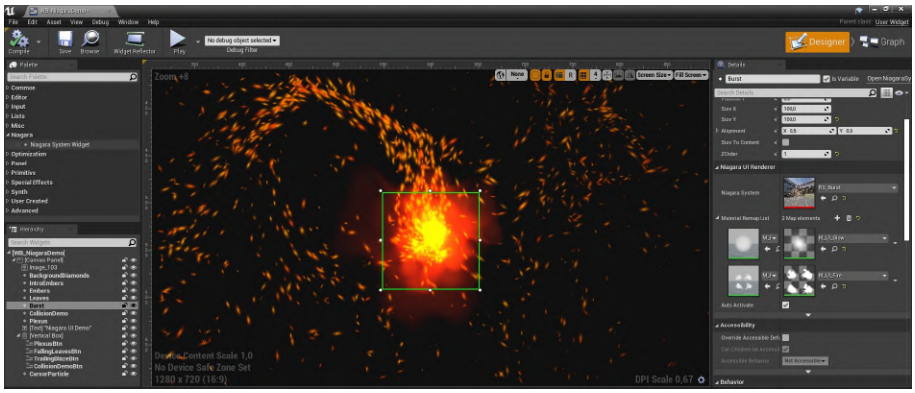

Figura 1.6: Schermata di esempio del software Unreal Engine.

La scelta del software più adatto varia in base al tipo di progetto, alla complessità dell'effetto desiderato e alle esigenze di integrazione con le altre fasi della pipeline digitale, ma tutti questi strumenti rappresentano oggi la punta di diamante nella simulazione particellare e negli effetti visivi.

1.2 CGI e VFX nel cinema

Negli ultimi decenni, l'informatica ha assunto un ruolo sempre più centrale nella produzione cinematografica, diventando uno strumento indispensabile soprattutto per la realizzazione di film e serie TV che richiedono effetti visivi e speciali di elevata complessità, spesso impossibili o troppo costosi da ottenere con le tradizionali tecniche di ripresa dal vivo.

In questo contesto, la **CGI** (*Computer-Generated Imagery*), ovvero la creazione di immagini, scene e animazioni generate al computer , rappresenta una tecnologia estremamente versatile il cui impiego va ben oltre la semplice animazione o la creazione di effetti visivi, trovando applicazione in multipli ambiti diversi come l'arte digitale, lo sviluppo di videogiochi, la modellazione tridimensionale e simulazioni di vario tipo.

Questa disciplina ha fatto un ulteriore passo avanti grazie ai progressi nel campo dell'intelligenza artificiale. Strumenti come **DALL·E 3 e VEO-3** utilizzano tecniche avanzate di deep learning per generare contenuti visivi, come immagini statiche o video, a partire da descrizioni testuali, chiamate **prompt**, aprendo nuove opportunità creative e produttive agli utenti.

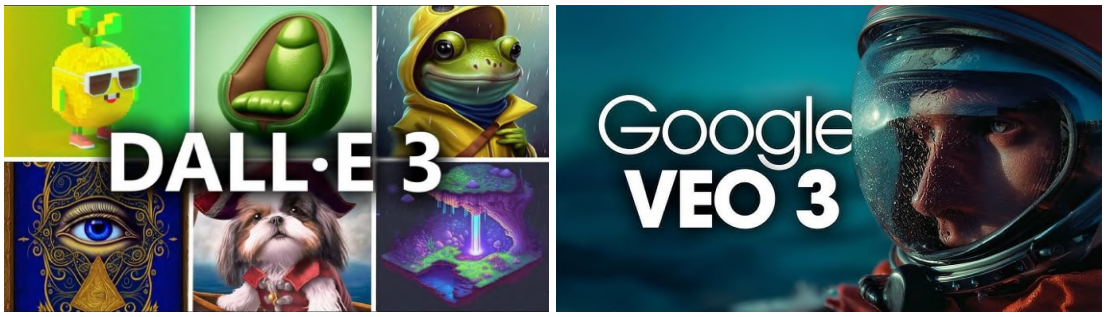

Figura 1.7: DALL·E 3 e VEO 3, software di generazione multimediale basati su intelligenza artificiale, attualmente tra i più utilizzati nel campo della creazione visiva automatizzata.

La CGI si basa sull'utilizzo della computer grafica per creare contenuti sia bidimensionali che tridimensionali: nel cinema essa può essere utilizzata autonomamente per la realizzazione di intere scene o sequenze animate, oppure integrata sapientemente con riprese live action attraverso tecniche come la **motion capture**, che permette di trasferire i movimenti di attori reali a personaggi digitali, garantendo così un realismo senza precedenti.
Per ottenere risultati di alta qualità, tuttavia, è necessario disporre di un complesso ecosistema software altamente specializzato, supportato da potenti infrastrutture di calcolo, in particolare per la fase di **rendering**, che consiste nel processo di elaborazione e trasformazione dei modelli digitali, delle animazioni e delle texture in sequenze di fotogrammi finali. La CGI è dunque utilizzata non solo per creare personaggi e ambienti digitali complessi, ma anche per la realizzazione di elementi grafici bidimensionali, che contribuiscono a caratterizzare l'estetica del prodotto audiovisivo.

L'impiego della computer grafica nel cinema risale ai primi anni '60, ma è stato negli anni '80 e '90 che questa tecnologia ha conosciuto una vera e propria esplosione, grazie a film iconici come *Tron* (1982), che ha introdotto l'uso pionieristico della CGI, *Jurassic Park* (1993), celebre per la sua rivoluzionaria integrazione tra effetti digitali e èartici, e *Terminator 2: Il giorno del giudizio* (1991), che ha portato il morphing e altri effetti digitali a nuovi livelli di complessità e realismo.

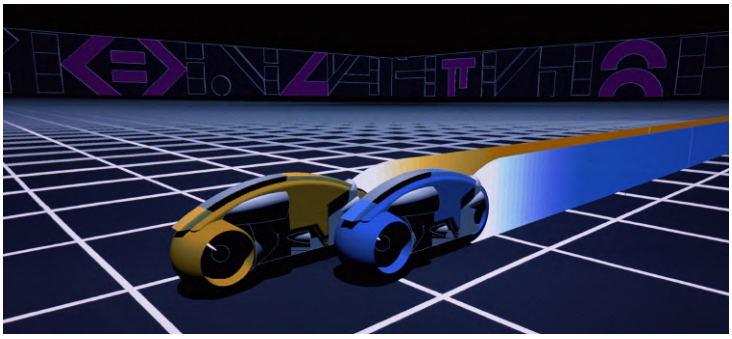

Figura 1.8: Tron (1982).

Da allora, la CGI è diventata sempre più sofisticata e pervasiva, come si può osservare nelle produzioni contemporanee di grande successo quali *Avatar*, *Dune*, *Jurassic World* e il *Pianeta delle Scimmie*, dove la perfetta integrazione tra elementi digitali e riprese live action, accompagnata da un'attenta gestione dell'illuminazione e delle ombre, contribuisce a creare mondi immersivi e visivamente spettacolari, capaci di coinvolgere lo spettatore in maniera profonda.

Figura 1.9: Avatar 2 : la via dell'acqua (2022).

Per quanto riguarda il cinema d'animazione, *Toy Story* rappresenta una pietra miliare essendo il primo lungometraggio completamente realizzato in CGI, aprendo la strada a una nuova era che ha visto la produzione di capolavori come *Inside Out* e i due film dello *Spider-Verse*, opere che hanno rivoluzionato il linguaggio visivo e narrativo del settore.

Figura 1.10: Toy Story (1995) e Spiderman: Across the Spider-Verse (2023).

L'adozione diffusa della CGI ha avuto un impatto significativo sull'industria cinematografica, offrendo ai registi e ai creativi uno strumento straordinario per dar vita a visioni che, solo pochi decenni fa, sarebbero state impensabili. Questo ha permesso di superare i limiti della realtà, spingendo sempre più lontano i confini della fantasia. Tuttavia, un uso eccessivo o sbilanciato della CGI potrebbe ridurre il valore delle tecniche tradizionali, come il trucco o gli effetti pratici sul set, che spesso contribuiscono a dare autenticità e tattilità alle scene. Nonostante ciò, nel

complesso, la computer grafica ha ampliato enormemente le possibilità creative nel cinema contemporaneo.[5]

Figura 1.11: Confronto dei VFX sui Velociraptor in Jurassic Park (1993) e in Jurassic World (2015).

1.3 Effetti visivi (VFX)

Gli **effetti visivi** (**VFX**) rappresentano oggi una componente fondamentale nella produzione cinematografica e televisiva, poiché permettono di **aggiungere**, **modificare** o **migliorare** elementi visivi che non sono presenti o realizzabili sul set durante le riprese dal vivo. Tali effetti possono spaziare dalla creazione di **ambienti digitali** altamente dettagliati, a **personaggi fantastici**, fino ad includere **esplosioni**, **catastrofi naturali** e simulazioni di **fenomeni atmosferici**, ovvero qualsiasi elemento visivo che contribuisca a costruire in modo credibile e coinvolgente l'atmosfera e la narrazione della storia.

A differenza degli **effetti pratici** tradizionali, realizzati fisicamente sul set durante le riprese, i VFX si basano sull'utilizzo di **tecnologie digitali avanzate** che consentono di integrare immagini generate al computer con le riprese live action, ottenendo così una fusione quasi **impercettibile** tra il reale e il virtuale, e ampliando notevolmente le possibilità creative dei filmmaker.
Questo processo complesso richiede una stretta collaborazione tra **artisti digitali, registi e tecnici specializzati in compositing, animazione e simulazione**, ognuno dei quali apporta competenze specifiche al risultato finale.

Figura 1.12: Dietro le quinte della realizzazione di una scena in Avengers: Infinity War.

Nel corso degli anni, i progressi tecnologici hanno portato ad un aumento significativo sia della complessità che della qualità degli effetti visivi, che oggi sono in grado di riprodurre con una precisione straordinaria dettagli un tempo impensabili, come il comportamento realistico di elementi naturali come fuoco, acqua, folle o condizioni atmosferiche. Questo livello di realismo contribuisce in modo decisivo a immergere lo spettatore nel mondo narrativo, rafforzando la credibilità visiva del film

È importante sottolineare che l'utilizzo intensivo dei VFX comporta anche un **incremento considerevole** sia nei **costi di produzione** sia nei **tempi di realizzazione** delle sequenze più complesse, richiedendo infrastrutture tecnologiche all'avanguardia e team altamente specializzati. Tuttavia, nelle produzioni di punta a livello globale, questi strumenti sono ormai **irrinunciabili**: alcuni casi emblematici includono blockbuster come *Jurassic World*, *Star Wars: Il risveglio della forza* e la saga degli *Avengers*, film caratterizzati da budget che spesso superano i 350 milioni di dollari, arrivando in alcuni casi a sfiorare i 600 milioni, in larga parte a causa dell'uso massiccio di effetti visivi sofisticati.

Figura 1.13: La troupe durante la realizzazione di una scena con green screen, sul set di un film.

Oltre all'aspetto puramente estetico, i VFX consentono di realizzare scene che sarebbero altrimenti **pericolose**, **costose** o addirittura **impossibili** da girare nella realtà, come **distruzioni catastrofiche**, **battaglie epiche** o **ambientazioni futuristiche**, ampliando in modo sostanziale le possibilità narrative e creative degli autori.

L'impatto degli effetti visivi si estende anche alla **post-produzione**, fase in cui la possibilità di intervenire sulle immagini con **precisione millimetrica** consente di **correggere**, **migliorare** o **modificare** ogni dettaglio, contribuendo così a perfezionare il prodotto finale prima della distribuzione nelle sale o sulle piattaforme digitali.
Gli effetti visivi sono diventati un elemento imprescindibile per il successo commerciale e artistico di un film, poiché consentono di superare le limitazioni tecniche e pratiche proprie delle produzioni tradizionali, offrendo al pubblico esperienze visive sempre più spettacolari, immersive e coinvolgenti.[6]

1.4 Processo di creazione dei modelli 3D

I modelli 3D di personaggi e oggetti, che vediamo nelle produzioni cinematografiche o videoludiche, nascono da un processo creativo e complesso che combina capacità artistiche e tecniche avanzate di computer grafica.
Questo percorso coinvolge diverse fasi, tra cui la modellazione geometrica, l'applicazione delle texture, la preparazione per l'animazione e l'ottimizzazione delle risorse computazionali, tutte integrate accuratamente per garantire un risultato di qualità.

- **Concept design**: la fase iniziale riguarda lo sviluppo dell'idea di base del personaggio o dell'oggetto digitale. Questo processo include la raccolta di riferimenti visivi, la creazione di mood board e schizzi, sia a mano libera che digitali, che rappresentano il design da diverse angolazioni e con vari livelli di dettaglio. Questa fase è fondamentale per definire le caratteristiche estetiche e funzionali del modello, costituendo la base per tutte le fasi successive.

Figura 1.14: Concept Art di diverse parti del corpo di un personaggio.

- **Modellazione 3D**: si passa alla costruzione digitale del modello utilizzando software specializzati come Blender, Autodesk Maya o ZBrush. Partendo da primitive geometriche semplici (ad esempio cubi, sfere o cilindri), l'artista modella la superficie del modello tramite tecniche di *sculpting* digitale o *polygonal modeling*, aggiungendo dettagli e definendo la forma finale. Nei modelli *high poly* si possono raggiungere milioni di poligoni, necessari per catturare anche i minimi dettagli. Dal punto di vista informatico, questa fase comporta la gestione e manipolazione efficiente di mesh complesse, ottimizzando le strutture dati per consentire modifiche interattive in tempo reale.

Figura 1.15: Modello 3D del personaggio visto da diverse angolazioni.

- **Retopologia**: poiché i modelli *high poly* sono troppo pesanti per essere animati o utilizzati in tempo reale, si procede con la retopologia, ovvero la creazione di una mesh *low poly* che riproduce fedelmente la forma del modello originale, ma con un numero ridotto di poligoni. Questa fase è fondamentale per migliorare le prestazioni computazionali e garantire una deformazione corretta durante l'animazione. Algoritmi di *remeshing* e *decimation* vengono impiegati per bilanciare la qualità visiva e la complessità geometrica.

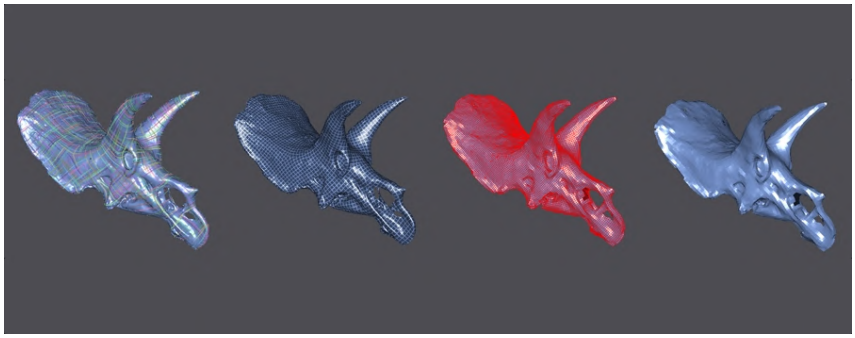

Figura 1.16: Stadi della reptologia del modello 3D.

- **UV unwrapping**: per applicare texture bidimensionali sulla superficie tridimensionale, il modello viene "srotolato" in uno spazio 2D attraverso un processo chiamato UV unwrapping. Questa operazione implica la minimizzazione di distorsioni e sovrapposizioni

delle coordinate UV, e spesso utilizza algoritmi di *parameterization* della mesh basati su metodi di ottimizzazione e minimizzazione dell'energia per preservare proporzioni e continuità. Il risultato consente di mappare immagini di texture, mappe di normalità, specularità e altri attributi visivi sul modello in modo realistico.

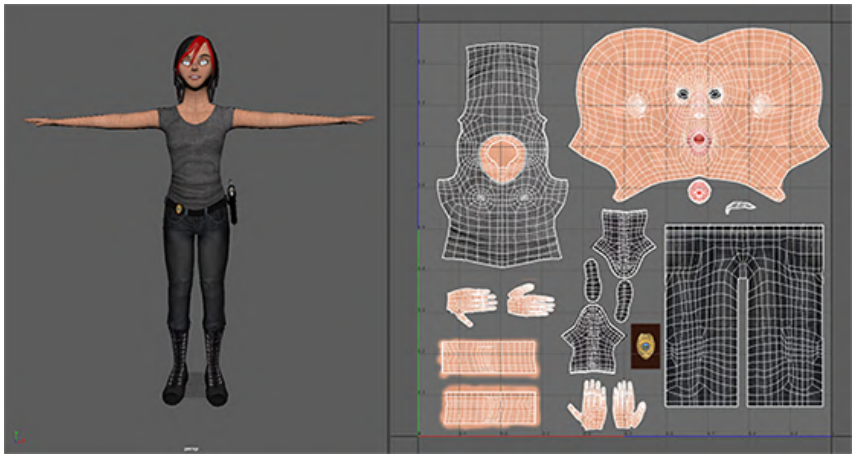

Figura 1.17: UV Unwrapping del personaggio 3D in 2D.

- **Rigging**: passaggio fondamentale per rendere il modello animabile: si crea uno scheletro digitale composto da ossa e articolazioni, definendo una struttura gerarchica che controlla il movimento. Il rigging comprende la definizione di joint, vincoli e sistemi di controllo, spesso supportati da tecniche di *inverse kinematics* (IK) per semplificare la manipolazione delle pose. Dal punto di vista algoritmico, ciò comporta la gestione di trasformazioni rigide e calcoli matriciali complessi per propagare correttamente i movimenti alle diverse parti del modello.

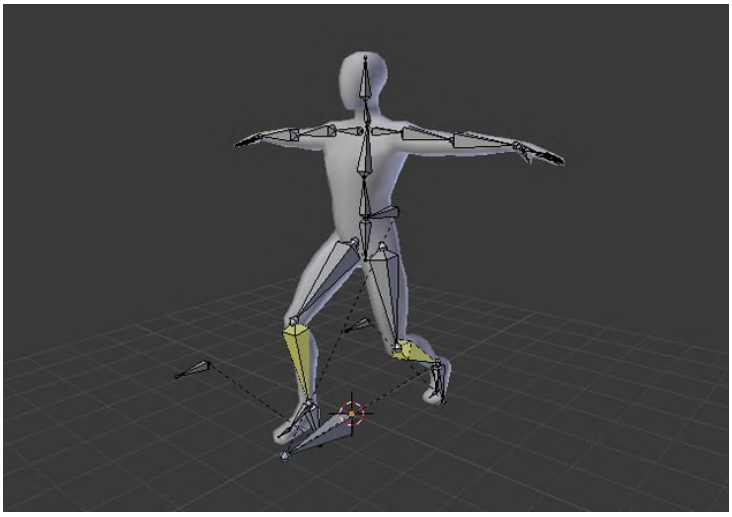

Figura 1.18: Creazione dello scheletro del personaggio per i movimenti.

- **Skinning**: questa fase consiste nell'assegnare pesi di influenza alle diverse parti della mesh in relazione alle ossa del rig, definendo come la superficie si deforma in seguito al movimento dello scheletro. Le tecniche più comuni sono il *linear blend skinning* e il *dual quaternion skinning*, che affrontano il problema della deformazione fluida della mesh, riducendo al minimo distorsioni visive e artefatti computazionali. La gestione efficiente di questi pesi e la loro interpolazione rappresentano sfide cruciali per la qualità finale dell'animazione.

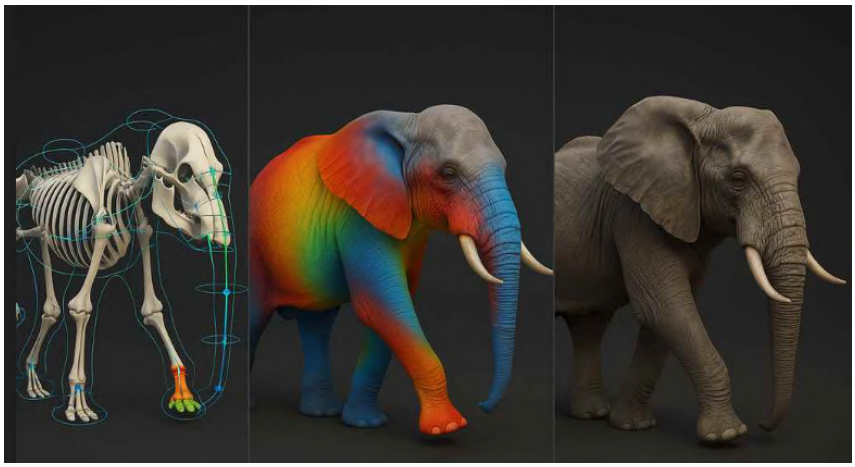

Figura 1.19: Assegnazione di materiali e colori al personaggio per renderlo più realistico.

A questo punto, il modello tridimensionale è pronto per essere animato, e le due principali tecniche utilizzate sono:

1. **Animazione keyframe**: è la tecnica tradizionale in cui l'animatore definisce manualmente le posizioni chiave (keyframes) del modello in punti specifici nel tempo. Successivamente, un algoritmo di interpolazione calcola automaticamente i fotogrammi intermedi per creare un movimento fluido. Questo metodo richiede una profonda conoscenza artistica e una buona padronanza del software di animazione, ma permette un controllo preciso sull'espressività e sul timing.

2. **Motion capture**: tecnica che sfrutta sensori e telecamere per acquisire i movimenti reali di attori o oggetti, convertendoli in dati digitali che vengono poi applicati ai modelli 3D. Dal punto di vista tecnico, implica la gestione di grandi quantità di dati di movimento, la calibrazione dei sistemi di acquisizione e algoritmi di post-processing per filtrare rumori e correggere errori. La motion capture permette di ottenere animazioni estremamente realistiche, soprattutto per personaggi umanoidi, riducendo il tempo necessario rispetto all'animazione manuale.

La creazione di modelli 3D animabili è un processo complesso e altamente specializzato che combina competenze artistiche, solide conoscenze geometriche e tecniche informatiche avan-

zate. Questo lavoro articolato ha come obiettivo non solo il raggiungimento del realismo visivo, ma anche l'ottimizzazione delle prestazioni, aspetti oggi indispensabili nelle produzioni cinematografiche, televisive e multimediali di alto livello. La sinergia tra arte e tecnologia ha profondamente rivoluzionato il settore audiovisivo, rendendo possibile la realizzazione di effetti visivi innovativi e personaggi digitali credibili, che un tempo sarebbero stati irrealizzabili.
Il risultato è una narrazione visiva sempre più coinvolgente, immersiva e spettacolare, capace di trasportare lo spettatore in mondi completamente nuovi e affascinanti.[7]

1.5 Tecniche di computer grafica con esempi cinematografici

In questa sezione vengono presentate alcune delle tecniche più importanti utilizzate nella computer grafica, corredate da esempi di film che hanno segnato la storia del cinema moderno.

1.5.1 Motion Capture: Andy Serkis, il caso di Cesare

La **motion capture** (mocap) è una tecnica che sfrutta la registrazione dei movimenti reali di attori o oggetti per animare personaggi digitali in modo estremamente realistico. Per acquisire i dati del movimento corporeo e facciale, si utilizzano telecamere e sensori speciali applicati sul corpo e sul volto dell'attore, tra cui i cosiddetti *facial markers*. Questi ultimi sono piccoli punti riflettenti o sensori posizionati in punti strategici del viso, che permettono di tracciare con precisione i minimi spostamenti e le espressioni facciali dell'attore durante la recitazione, catturando dettagli come movimenti delle sopracciglia, sorrisi, corrugamenti della fronte e persino microespressioni. Questi dati vengono poi tradotti in modelli digitali che riproducono fedelmente le emozioni e i movimenti del volto, rendendo possibile una rappresentazione estremamente realistica e sfumata dei personaggi animati.

Figura 1.20: Andy Serkis, professionista nel settore del Motion Capture.

Tra i pionieri e le figure più importanti nel settore della motion capture vi è senza dubbio **Andy Serkis**, attore e regista britannico che ha rivoluzionato il modo di interpretare personaggi digitali. Serkis ha portato la motion capture a un livello artistico superiore, trasformando la recitazione digitale in una vera e propria forma di espressione teatrale, e dimostrando come il talento umano possa fondersi con la tecnologia per creare personaggi credibili sul grande schermo. La sua capacità di infondere vita e personalità ai personaggi digitali ha contribuito a consolidare la motion capture come tecnica essenziale nel cinema contemporaneo.

Un esempio emblematico del suo lavoro è la creazione di **Cesare**, protagonista della saga de *Il Pianeta delle Scimmie*. Per realizzare questo personaggio, Andy Serkis ha indossato una tuta dotata di sensori per la cattura dei movimenti corporei e una telecamera speciale montata sul volto, per registrare in tempo reale ogni espressione facciale con altissima precisione. I dati raccolti sono stati elaborati dallo studio Weta Digital, che ha sovrapposto un modello digitale dell'anatomia di una scimmia al corpo e al volto dell'attore, integrando i movimenti e le espressioni registrati per creare un personaggio digitale animato ma dotato di una complessità emotiva e di una naturalezza mai viste prima.

Figura 1.21: Fasi del motion capture per il personaggio di Cesare: da Andy Serkis in tuta, al modello digitale, fino al risultato finale fotorealistico.

La realizzazione di Cesare rappresenta un punto di svolta nella motion capture, poiché il personaggio mostra movimenti articolati e sfumature emotive complesse, come rabbia, tristezza, esitazione e autorità, capaci di suscitare empatia nel pubblico. Questa fusione tra recitazione reale e animazione digitale ha ridefinito i confini del possibile nel cinema, aprendo nuove strade nella rappresentazione di creature e personaggi virtuali.

Tuttavia, questa tecnologia comporta sfide tecniche significative, tra cui la gestione e l'elaborazione di grandi quantità di dati complessi, la sincronizzazione precisa tra movimenti del corpo

e del volto, e l'enorme potenza di calcolo necessaria per garantire risultati realistici entro tempi produttivi compatibili con le esigenze dell'industria cinematografica.

1.5.2 Morphing: il T-1000 in Terminator 2

Il *morphing* è una tecnica di computer grafica che consente la trasformazione graduale e continua di un oggetto o personaggio in un altro, mediante l'interpolazione progressiva di forme e immagini. Questa metodologia risulta particolarmente efficace per rappresentare mutazioni, metamorfosi o transizioni surreali tra stati visivi distinti, generando effetti di forte impatto visivo.

Un esempio emblematico e rivoluzionario dell'impiego del *morphing* si trova in *Terminator 2: Judgment Day* (1991), film diretto da James Cameron. Il personaggio del T-1000, un androide composto da metallo liquido, è in grado di assumere differenti configurazioni fisiche, trasformandosi in oggetti o individui con movimenti estremamente fluidi e naturali.
Grazie al lavoro pionieristico della *Industrial Light & Magic* (ILM), vennero realizzate sequenze visive straordinarie, come il passaggio del T-1000 attraverso le sbarre di una cella o la rigenerazione di parti del corpo danneggiate, che rappresentarono un livello di sofisticazione tecnica senza precedenti per l'epoca.

L'introduzione del *morphing* digitale in *Terminator 2* costituì un punto di svolta fondamentale nel cinema degli effetti speciali, segnando il passaggio dagli effetti pratici e meccanici alle soluzioni generate interamente al computer. Tale innovazione aprì la strada a un utilizzo sempre più diffuso e avanzato della CGI nelle produzioni cinematografiche successive.[10]

Figura 1.22: Realizzazione pratica dell'effetto di Morphing del T-1000, a cui vengono aggiunti gli effetti speciali in Post-Produzione.

1.5.3 Bullet Time: Matrix, Neo schiva i proiettili

Una delle scene più iconiche e rivoluzionarie del cinema degli inizi degli anni 2000 è quella del film *Matrix*, in cui il protagonista Neo, interpretato da Keanu Reeves, schiva una raffica di proiettili piegandosi all'indietro in un movimento rallentato ed estremamente fluido, creando un effetto visivo mai visto prima sul grande schermo.

Questa spettacolare sequenza è stata realizzata utilizzando una tecnica denominata **Bullet Time**, ideata da John Gaeta e dal team di Manex Visual Effects. La tecnica si basa sull'impiego di un sistema complesso di decine di telecamere ad alta risoluzione, disposte in un arco o spirale attorno all'attore, che catturano simultaneamente o in rapida successione molteplici angolazioni della scena.

Durante la ripresa, mentre Reeves esegue il movimento rallentato, sorretto da un insieme di cavi, ciascuna telecamera scatta una fotografia in un preciso istante, creando così una sequenza di immagini che, una volta assemblate, permettono di ottenere un effetto di tempo quasi congelato ma con la possibilità di spostare virtualmente la "camera" attorno al soggetto in movimento. Questo sistema consente di esplorare la scena da prospettive dinamiche e non convenzionali, combinando un rallentamento estremo con movimenti fluidi di inquadratura che intensificano la drammaticità dell'azione.

Successivamente, gli effetti visivi digitali sono stati utilizzati per inserire proiettili, esplosioni e lo sfondo, integrandoli perfettamente con la ripresa live action per amplificare l'impatto visivo della scena. L'uso innovativo del Bullet Time ha trasformato *Matrix* in un vero e proprio cult degli effetti speciali, influenzando profondamente l'estetica del cinema d'azione e aprendo la strada a molteplici sperimentazioni nel campo della computer grafica e delle tecniche di ripresa avanzate.

Figura 1.23: Dietro le quinte della realizzazione dell'effetto del Bullet Time dove Neo schiva i proiettili, una delle più famose scene di azione del cinema.

1.5.4 Interstellar: Gargantua, il buco nero più realistico del cinema

Christopher Nolan è un regista universalmente riconosciuto per il suo approccio cinematografico che privilegia l'uso di effetti pratici rispetto alla CGI, basando questa scelta sulla convinzione che ciò che è reale risulti sempre più convincente e coinvolgente per il pubblico rispetto agli elementi generati digitalmente. La sua filosofia si traduce in un'attenta valutazione di ogni singola scena, in cui la CGI viene impiegata soltanto quando gli effetti pratici si rivelano tecnicamente impossibili o estremamente rischiosi da realizzare sul set. Un esempio emblematico di questo equilibrio è rappresentato dalle ustioni di Harvey Dent nel film *Il Cavaliere Oscuro*, dove la computer grafica è stata utilizzata esclusivamente per riprodurre ferite che non potevano essere simulate in maniera sicura e realistica con metodi tradizionali.

Questa visione si riflette nel successo dei suoi film *Inception* e *Interstellar*, entrambi vincitori dell'Oscar per i migliori effetti visivi, dimostrando come l'equilibrio sapiente tra effetti digitali e pratici possa portare a risultati di altissimo livello, capaci di sorprendere e coinvolgere lo spettatore senza sacrificare l'autenticità visiva.
Uno degli esempi più straordinari dell'approccio di Nolan alla computer grafica è rappresentato dalla realizzazione del buco nero **Gargantua** nel film *Interstellar*, un elemento visivo che ha richiesto uno sforzo combinato tra scienza, arte e tecnologia.

Il team di effetti visivi DNEG, guidato da Oliver James, ha collaborato strettamente con il fisico teorico Kip Thorne per sviluppare un software di rendering in grado di rappresentare con estrema accuratezza un buco nero rotante, tenendo conto delle complesse leggi della relatività generale.

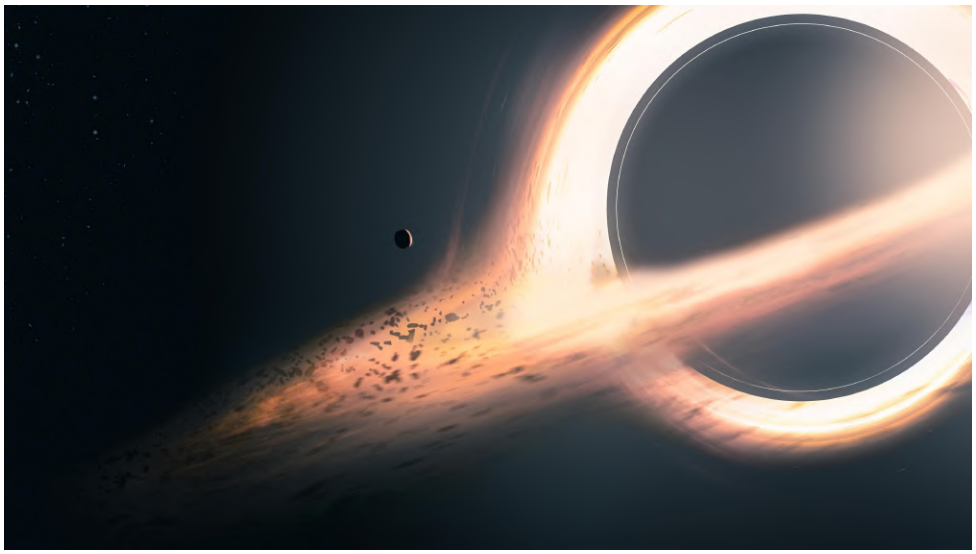

Figura 1.24: Il buco nero "Gargantua" in Interstellar (2014).

Questo software è stato progettato per simulare in modo realistico il percorso dei raggi luminosi all'interno di uno spazio-tempo fortemente curvato dalla gravità, modellare l'effetto di lente gravitazionale e riprodurre fenomeni relativistici come l'abberrazione della luce. Inoltre, il sistema permetteva di regolare vari parametri per rappresentare non solo buchi neri ma anche wormhole, garantendo così una flessibilità visiva senza precedenti.

La complessità del rendering è tale che ogni singolo fotogramma richiedeva fino a 100 ore di elaborazione, generando un'enorme mole di dati che ha superato i 700 terabyte, e facendo uso di cataloghi stellari ufficiali dell'ESA come Tper ricostruire fedelmente lo sfondo stellato.

Il risultato scientifico e artistico di questo lavoro ha portato alla pubblicazione di importanti studi accademici e libri divulgativi, come *The Science of Interstellar* scritto da Kip Thorne, e numerosi articoli apparsi su riviste come *Classical and Quantum Gravity*. Tra le sfide tecniche più significative affrontate vi è stata la simulazione di una cinepresa che si muove a velocità relativistiche, con la necessità di integrare effetti quali l'abberrazione relativistica, l'effetto Doppler, il redshift gravitazionale e la distorsione dello spazio-tempo, tutte componenti fondamentali per garantire una resa fedele della fisica.[9]

1.6 Evoluzione e prospettive future dell'industria VFX

L'industria degli effetti visivi è attualmente in una fase di crescita e trasformazione senza precedenti. Un esempio lampante è rappresentato dalla crescita esponenziale di DNEG, una delle più importanti società di VFX al mondo, che nel 2004 contava circa 80 dipendenti ed è arrivata oggi a oltre 5.000, testimonianza della crescente domanda e complessità dei progetti.

Con l'aumento delle dimensioni e della complessità delle produzioni, i software e le infrastrutture impiegate devono necessariamente essere scalabili, modulari e in grado di adattarsi rapidamente a esigenze in continuo mutamento, supportando flussi di lavoro collaborativi e integrati. Tra le innovazioni che stanno plasmando il futuro degli effetti visivi va segnalato innanzitutto il **rendering in tempo reale**, che permetterà agli artisti di effettuare iterazioni rapide e di visualizzare immediatamente le modifiche senza la necessità di lunghi rendering batch notturni, aumentando così l'efficienza e la creatività.

Inoltre, le applicazioni di **machine learning** e intelligenza artificiale stanno entrando sempre più nel processo creativo, automatizzando la generazione di espressioni facciali più realistiche, movimenti naturali dei personaggi e persino la creazione automatica di effetti complessi, riducendo i tempi e i costi di produzione.

Infine, la crescente attenzione verso l'**open data** e l'**open access** favorisce la condivisione di risorse, algoritmi e tecnologie, con realtà come DNEG che si impegnano attivamente per promuovere la collaborazione tra industria, mondo accademico e comunità scientifica, aprendo nuove opportunità di innovazione e crescita.

Figura 1.25: Sessione collaborativa di lavoro di gruppo all'interno di una sede di DNEG.

1.7 Integrazione dell'intelligenza artificiale nel cinema

L'integrazione dell'intelligenza artificiale si sta affermando in modo sempre più significativo nel mondo cinematografico, colmando il divario tra ambizioni creative elevate e limitazioni di budget. In particolare, in film di grande successo come *Avengers: Endgame*, sono stati impiegati algoritmi avanzati per cogliere ed elaborare i movimenti facciali degli attori, mettendo in luce un dominio discreto ma influente dell'AI nella fase di pre-produzione. Studi come quello di *Industrial Light & Magic e Digital Domain* hanno reso possibile, attraverso sistemi come **Masquerade**, una resa fedele delle espressioni reali di Josh Brolin trasferite sul volto virtuale del personaggio di Thanos, accelerando significativamente il processo di animazione visiva. Inoltre, per il personaggio di Smart Hulk sempre in Avengers: Endgame, *ILM e Disney Research* hanno implementato il sistema markerless **Anyma**, capace di catturare dettagli anatomici fini, come lo scorrimento della pelle, preservando la libertà dell'attore Mark Ruffalo durante le riprese.[36]

Figura 1.26: Prima e Dopo della realizzazione del personaggio di Thanos.

Case di produzione come *20th Century Fox e Warner Bros* stanno già sfruttando l'intelligenza artificiale per analizzare le sceneggiature dei loro prodotti: strumenti come **ScriptBook e Merlin** valutano elementi quali trama, sviluppo dei personaggi, tono e potenzialità commerciali, offrendo previsioni sulla riuscita al botteghino, tempi di revisione significativamente ridotti e un'accuratezza nelle selezioni del 25–35% superiore rispetto ai metodi tradizionali. Questi sistemi non solo ottimizzano la scelta dei progetti da produrre ma contribuiscono anche alla personalizzazione dei contenuti audiovisivi, generando sottotitoli in tempo reale in più lingue e ottimizzando il doppiaggio sincronizzato alle labbra degli attori.

Anche il casting è al centro della rivoluzione AI: piattaforme come **Cinelytic** sono impiegate per valutare il valore di mercato degli attori, prevedere il successo di un film e selezionare per-

former esteticamente affini al ruolo desiderato, innescando una selezione più rapida e accurata. L'AI risulta inoltre decisiva nella creazione di massa di comparse digitali: film epici come Il Gladiatore o Il Signore degli Anelli hanno sfruttato la generazione tramite AI di folle virtuali, riducendo drasticamente i costi legati all'impiego di comparse reali e offrendo al contempo straordinaria flessibilità creativa.

Un ambito particolarmente innovativo è quello del cosiddetto **"de-aging"**, ovvero la tecnica che consente di ringiovanire digitalmente un attore per rappresentarlo in diverse fasi della sua vita all'interno dello stesso film. Un esempio emblematico è il caso di *Gemini Man* (2019), in cui la tecnologia di intelligenza artificiale e grafica CGI è stata utilizzata per creare una versione completamente digitale e ringiovanita dell'attore Will Smith. Nel film, Smith interpreta sia un sicario di mezza età sia una sua controparte ventenne, generata interamente al computer.
Il processo non si è limitato all'applicazione di filtri cosmetici o correzioni superficiali: la versione giovane del personaggio è stata realizzata partendo da zero, sfruttando tecniche di motion capture, machine learning e modellazione 3D avanzata, combinando dati reali delle sue performance con un volto virtuale costruito ad hoc. La produzione ha utilizzato un mix di AI per l'analisi delle espressioni facciali, l'apprendimento automatico su reference video storici dell'attore, e rendering fotorealistici generati con tecniche di path tracing ad alta risoluzione.
L'adozione dell'AI per il de-aging si sta ormai diffondendo nell'industria cinematografica, aprendo la strada a nuove forme di storytelling in cui il tempo può essere manipolato digitalmente con estrema precisione, e in cui la performance di un attore può essere proiettata oltre i limiti anagrafici e fisici del corpo umano.

Figura 1.27: La versione ringiovanita (SX) e originale (DX) di Will Smith.

Non meno rilevante è il contributo dell'AI nei doppiaggi: Start-up come *Flawless* permettono di sincronizzare movimento delle labbra e mimica facciale con dialoghi localizzati (**"vubbing"**), oltre a modificare espressioni e parole in post-produzione (come nel film Fall), evitando rifilmati e garantendo una perfetta coerenza audio-visiva allo spettatore.
Dal punto di vista sonoro, l'AI affianca anche i compositori suggerendo melodie, orchestrazioni e arrangiamenti ispirati a stili e pattern su larga scala, ampliando la palette creativa a disposizione per quanto riguarda la realizzazione di colonne sonore.

Sul piano promozionale, servizi come Netflix usano l'intelligenza artificiale per personalizzare le anteprime (**thumbnails**) e strategie di marketing su misura, analizzando comportamenti di visione e interazioni social, mentre studi come Warner Bros adottano l'AI per prevedere gli incassi e definire strategie promozionali mirate, modellando campagne su target demografici specifici.

Tuttavia, questa rapida espansione dell'AI nel cinema solleva importanti riflessioni etiche: la possibilità di manipolare volti, voci o persino riportare in vita attori deceduti, come avviene nei deepfake, solleva interrogativi centrali riguardo al consenso, all'autenticità e all'integrità artistica. Le tecnologie di questo tipo, se non regolamentate, rischiano di erodere la fiducia del pubblico e compromettere l'identità stessa della produzione cinematografica.

Un esempio concreto è nel film Rogue One: A Star Wars Story (2016), in cui è stato ricreato digitalmente l'attore Peter Cushing per interpretare nuovamente il personaggio Grand Moff Tarkin, pur essendo morto nel 1994. Il volto di Cushing è stato applicato sul corpo dell'attore Guy Henry tramite CGI, motion capture e materiale d'archivio, con il permesso dell'erede legale.

Figura 1.28: Il processo di creazione del personaggio del Grand Moff Tarkin tramite Deepfake.

A livello normativo, il quadro comincia a prendere forma:

- In Italia, un disegno di legge sull'IA introduce l'art. 612-quater nel Codice penale, che punisce, con pene da uno a cinque anni, la diffusione illecita di contenuti audiovisivi generati o manipolati con IA, ingannevoli e dannosi per la reputazione altrui, prevedendo anche l'obbligo di segnalare visivamente i contenuti deepfake [38].
- L'Unione Europea, con l' **AI Act**, definisce i deepfake e li classifica come a rischio "limitato", imponendo trasparenza: obbligo di etichettatura, marcatura tecnica e indicazione dell'origine artificiale [37].
- La Danimarca ha proposto una normativa innovativa che attribuisce ai cittadini diritti legali sulla propria immagine, voce e somiglianza, consentendo di opporsi e ottenere risarcimento per deepfake non autorizzati, mentre le piattaforme non conformi rischiano pesanti sanzioni [39].
- Negli Stati Uniti, il **TAKE IT DOWN Act** obbliga le piattaforme a rimuovere contenuti intimi e deepfake non consensuali e in Corea del Sud la produzione o la distribuzione di deepfake pornografici non consensuali è un reato punibile severamente [40].
- Inoltre, il **GDPR** considera l'immagine e la voce come dati personali, soggetti a protezione: manipolarli senza consenso implica responsabilità legali su trasparenza, diritto alla privacy e trattamento dei dati [41].

Capitolo 2

Compute Shader in OpenGL: Fondamenti e Applicazioni

In questo capitolo vengono introdotti i **Compute Shader** in OpenGL, analizzandone i concetti fondamentali e le modalità di utilizzo. L'obiettivo è comprendere come questa tecnologia consenta di sfruttare in maniera diretta e flessibile la potenza di calcolo parallelo delle moderne GPU, andando oltre la tradizionale pipeline grafica.

Si partirà da una panoramica generale su cosa sono i compute shader e perché rappresentano un punto di svolta nel campo della grafica e della simulazione. Successivamente verranno descritti i principi di architettura e invocazione, i meccanismi di accesso ai dati tramite **Shader Storage Buffer Object (SSBO)**, e le principali tecniche di sincronizzazione tra thread.

Man mano che verranno trattate, queste sezioni conterranno esempi di codice per illustrare chiaramente come integrare questi strumenti nello sviluppo di applicazioni grafiche e fisiche moderne.

2.1 Cosa Sono i Compute Shader

Con l'evoluzione delle GPU moderne, la potenza di calcolo disponibile è cresciuta esponenzialmente, aprendo nuove possibilità oltre al tradizionale rendering grafico. In particolare, è diventato sempre più importante sfruttare questa potenza per eseguire calcoli generici e paralleli ad alte prestazioni, non limitati alla sola elaborazione di geometrie e pixel.

I **Compute Shader**, introdotti a partire da **OpenGL 4.3**, rappresentano una svolta fondamentale in questo ambito. Permettono di eseguire operazioni di calcolo generiche direttamente sulla GPU, in modo indipendente dalla normale **pipeline grafica** composta da **vertex, tessellation,**

geometry e fragment shader.[11]

Essi sono scritti in **GLSL (OpenGL Shading Language)**, lo stesso linguaggio utilizzato dagli altri shader della pipeline grafica. La loro peculiarità è di non essere legati al processo di rasterizzazione: invece di occuparsi della trasformazione di vertici o della colorazione dei pixel, i compute shader operano direttamente su buffer, immagini e strutture dati, adottando un paradigma di calcolo parallelo altamente scalabile. In questo modo la GPU può essere sfruttata come un vero e proprio motore di calcolo massivamente parallelo, ideale per applicazioni complesse come sistemi particellari, simulazioni fluidodinamiche e altre elaborazioni ad alte prestazioni.

Negli ultimi anni, i compute shader hanno assunto un ruolo centrale sia nella ricerca accademica sia nello sviluppo industriale, grazie alla loro capacità di accelerare algoritmi complessi in ambiti quali grafica avanzata, intelligenza artificiale e simulazioni scientifiche. La crescente diffusione di questa tecnologia riflette l'importanza strategica delle GPU non solo come dispositivi grafici, ma come piattaforme di calcolo altamente performanti.[12]
In questo capitolo si approfondiranno i concetti base dei compute shader, la loro architettura di esecuzione e le modalità di interazione con la memoria GPU, ponendo le basi per la progettazione di un sistema di simulazione particellare efficiente e moderno basato su queste tecnologie.

2.2 Architettura e concetto di invocazione

La GPU si basa su un'architettura di tipo **SIMD (Single Instruction, Multiple Data)**, in cui un singolo flusso di istruzioni viene eseguito in parallelo su un grande insieme di dati. Questo modello è particolarmente adatto per i compute shader, poiché consente di suddividere un problema complesso in molteplici sotto-problemi elementari che possono essere risolti contemporaneamente da centinaia o migliaia di thread GPU.

L'esecuzione di un compute shader avviene tramite il seguente comando:

```
glDispatchCompute(x, y, z);
```

Il termine **dispatch** indica il lancio di un insieme di invocazioni parallele del compute shader. I tre parametri (x, y, z) definiscono le dimensioni della griglia tridimensionale di calcolo, composta da **Work Groups**, blocchi di thread che cooperano tra loro e che possono condividere memoria locale e sincronizzarsi.

All'interno dello shader, la dimensione di ciascun work group si specifica con la direttiva:

```
layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;
```

In questo esempio, ogni work group contiene 128 thread organizzati lungo l'asse x, mentre non sono previsti thread negli assi y e z. Questo significa che il calcolo verrà suddiviso in gruppi unidimensionali di 128 invocazioni ciascuno.
Ogni thread GPU (o **invocazione locale**) può individuare la propria posizione nello spazio computazionale grazie a variabili integrate fornite dal linguaggio GLSL:

- `gl_GlobalInvocationID`: identificatore globale univoco del thread rispetto all'intera griglia lanciata con `glDispatchCompute`;
- `gl_WorkGroupID`: indice del work group corrente;
- `gl_LocalInvocationID`: posizione del thread all'interno del proprio work group;
- `gl_LocalInvocationIndex`: indice unidimensionale locale al gruppo, utile per scorrere array condivisi.

Questa organizzazione gerarchica dispatch → work groups → invocazioni locali permette di scalare il calcolo in modo efficiente, sfruttando al massimo la natura SIMD delle GPU.

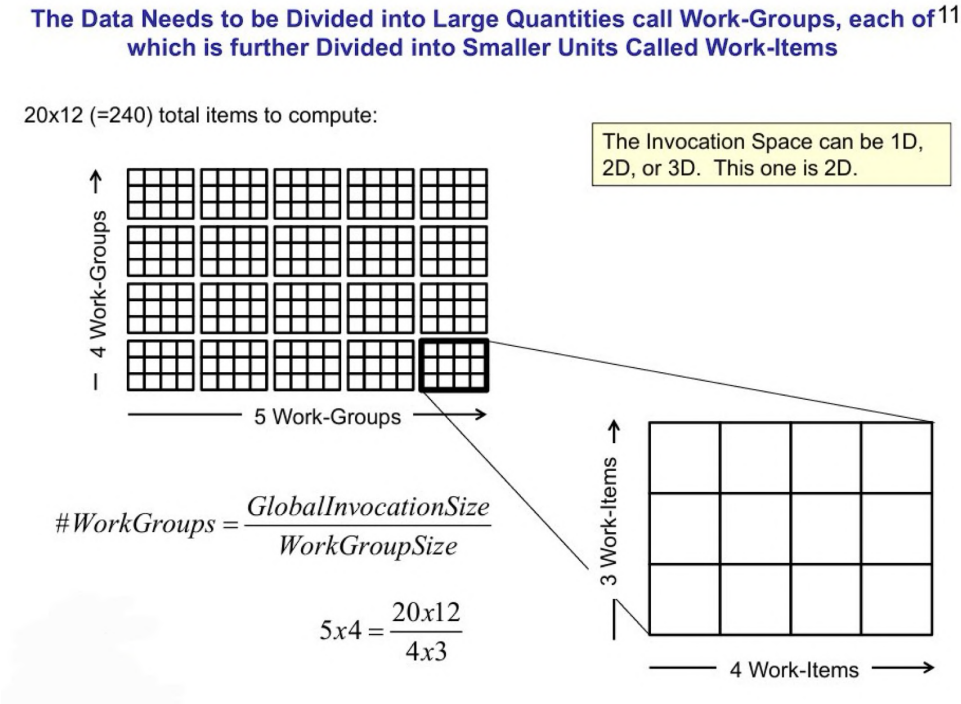

Figura 2.1: Suddivisione dello spazio computazionale in work groups e work items

2.3 Accesso ai dati: Shader Storage Buffer Object (SSBO)

Per consentire ai compute shader di leggere e scrivere grandi quantità di dati, OpenGL mette a disposizione gli **Shader Storage Buffer Object (SSBO)**. A differenza delle variabili uniform tradizionali, limitate in dimensione e solo in lettura, pensate principalmente per passare costanti agli shader, gli SSBO permettono di accedere sia in lettura che in scrittura, a strutture di dati complesse e array di dimensioni molto elevate. Questa caratteristica li rende fondamentali per applicazioni di calcolo parallelo, dove migliaia di thread GPU devono manipolare informazioni condivise in modo efficiente.[17]

Gli SSBO sono concettualmente simili agli **Uniform Buffer Object (UBO)**, poiché anch'essi vengono dichiarati tramite *interface block* in GLSL e associati a un punto di binding. Le differenze principali sono però sostanziali: mentre gli UBO sono limitati a circa 16KB, lo standard OpenGL garantisce per gli SSBO dimensioni fino a 128MB, e in molte implementazioni il limite effettivo coincide con la quantità di memoria disponibile sulla GPU. Inoltre, gli SSBO possono essere letti e scritti, anche con operazioni atomiche, rendendoli molto più flessibili degli UBO che sono invece in sola lettura. Un ulteriore vantaggio è la possibilità di dichiarare array a lunghezza variabile, sfruttando lo spazio di memoria effettivamente allocato e interrogando la dimensione a runtime mediante la funzione `.length()`.

Nel linguaggio GLSL, un SSBO si dichiara specificando il layout e il punto di binding.
Ad esempio:

```
layout(std430, binding = 4) buffer Pos {
    vec4 Positions[];
};
```

In questo caso viene dichiarato un buffer chiamato `Pos`, organizzato secondo il layout `std430`, e associato all'unità di binding numero 4. Rispetto a `std140`, il layout `std430` permette una disposizione dei dati più compatta, riducendo lo spreco di memoria grazie a regole di allineamento meno restrittive. Al suo interno è presente un array di `vec4`, che può essere utilizzato dal compute shader per leggere e scrivere posizioni in modo diretto.

Dal lato C++, si crea il buffer corrispondente e lo si associa allo stesso punto di binding, gestendo così la condivisione dei dati tra CPU e GPU in modo efficiente per le operazioni parallele di calcolo. Il frammento di codice seguente mostra come configurarlo:

```
glGenBuffers(1, &posSSbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, posSSbo);
glBufferData(GL_SHADER_STORAGE_BUFFER, size, NULL, GL_STATIC_DRAW);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 4, posSSbo);
```

La prima chiamata, `glGenBuffers`, alloca un identificatore per il buffer. Con `glBindBuffer` si associa questo identificatore al target `GL_SHADER_STORAGE_BUFFER`, indicando che il buffer verrà utilizzato come SSBO. La funzione `glBufferData` riserva effettivamente la memoria sulla GPU: in questo caso lo spazio è pari a `size`, mentre il puntatore ai dati è `NULL`, a indicare che il buffer viene inizializzato vuoto. Infine, `glBindBufferBase` lega il buffer alla binding unit numero 4, che deve corrispondere esattamente al valore specificato nello shader con la direttiva `binding = 4`.

Dal punto di vista prestazionale, è bene osservare che l'accesso agli SSBO può risultare più lento rispetto agli UBO, in quanto avviene in modo simile alle *buffer textures*, cioè tramite operazioni di memoria meno ottimizzate. Tuttavia, la loro maggiore capacità e flessibilità li rende insostituibili in scenari complessi. L'uso in scrittura richiede inoltre particolare attenzione: trattandosi di accessi di memoria incoerenti, è necessario inserire le opportune barriere di memoria, ad esempio `memoryBarrier()` nei compute shader o `glMemoryBarrier` lato CPU, per garantire la visibilità e la coerenza dei dati tra diverse invocazioni.

Un aspetto avanzato riguarda l'uso dei qualificatori di memoria. È possibile specificare attributi come `coherent`, `volatile`, `restrict`, `readonly` e `writeonly` per controllare il comportamento delle operazioni di lettura e scrittura. Ad esempio, il qualificatore `coherent` assicura che le modifiche siano visibili a tutte le invocazioni, ma obbliga anche all'uso di barriere di memoria, mentre `restrict` informa il compilatore che quella variabile è l'unico riferimento a quella porzione di memoria, consentendo ottimizzazioni più aggressive.

Grazie a queste proprietà, i compute shader possono manipolare array di milioni di elementi in un singolo dispatch, realizzando simulazioni fisiche, algoritmi di collisione, sistemi particellari o tecniche avanzate di riduzione e parallelizzazione, sfruttando la GPU come un vero e proprio motore di calcolo general-purpose.

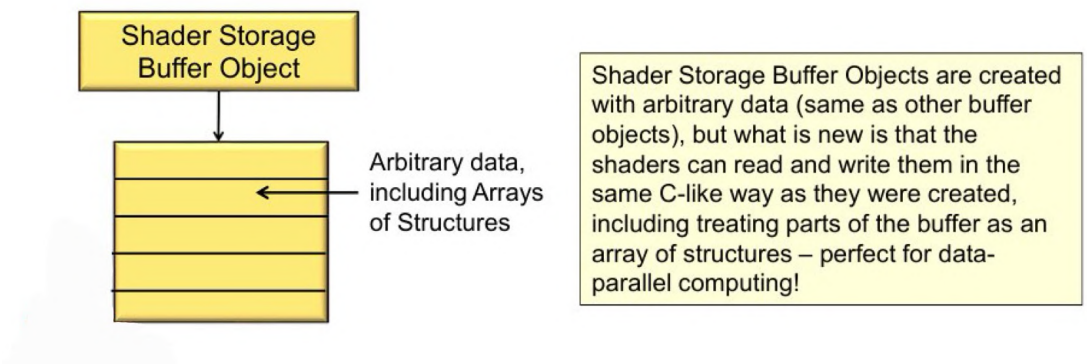

Figura 2.2: Collegamento degli SSBO in OpenGL

2.4 Confronto tra tecnologie di calcolo parallelo: CUDA, OpenCL e SYCL

Confrontare i compute shader con altre tecnologie di calcolo parallelo su GPU permette di evidenziare le peculiarità e i contesti d'uso ottimali per ciascuna soluzione.

CUDA[14]

CUDA (Compute Unified Device Architecture) è una piattaforma proprietaria sviluppata da NVIDIA, basata su un'estensione di C/C++, e concepita per l'uso esclusivo su GPU NVIDIA. Offre ampie librerie, strumenti di profiling e debugging, e un ecosistema molto maturo per l'HPC e la ricerca scientifica.

OpenCL[15]

OpenCL (Open Computing Language) è uno standard aperto gestito dal Khronos Group, pensato per supportare calcolo parallelo su piattaforme eterogenee (GPU, CPU, FPGA, ecc.). La sua natura multipiattaforma lo rende estremamente versatile, benché richieda spesso ottimizzazioni specifiche per raggiungere piena efficienza su hardware diversi.

SYCL[16]

SYCL è un modello di programmazione emergente, single-source basato su C++17, che consente di scrivere codice host e device in un unico file, aumentando la produttività. Nasce come estensione di OpenCL, ma si sta affermando come framework autonomo e più moderno per il calcolo eterogeneo.

Compute Shader (OpenGL)
I compute shader di OpenGL offrono una forma di calcolo parallelo integrata nella pipeline grafica stessa, senza necessità di contesto esterno. Pur non essendo multipiattaforma come OpenCL, risultano particolarmente efficaci e convenienti per aggiungere capacità di calcolo general-purpose all'interno di applicazioni OpenGL.

Differenze chiave

- **Portabilità**: OpenCL e SYCL sono multipiattaforma, CUDA è limitato alle GPU NVIDIA. I Compute Shader operano su GPU compatibili con OpenGL 4.3+.
- **Integrazione con la grafica**: I Compute shader sono immediatamente utilizzabili nella pipeline grafica, senza overhead esterni.

- **Ecosistema e tooling**: CUDA dispone di strumenti avanzati, OpenCL e SYCL sono più accessibili in contesti eterogenei. I Compute Shader offrono semplicità laddove la grafica è centrale.
- **Performance empiriche**: Alcune analisi mostrano che, in scenari come il volume rendering, i Compute Shader possono risultare più veloci rispetto a implementazioni in OpenCL o CUDA.

2.5 Gestione delle immagini nei C.Shader: Image Load/Store

Oltre agli **SSBO (Shader Storage Buffer Objects)**, i compute shader possono leggere e scrivere direttamente su **immagini** (texture) utilizzando le funzioni di *image load/store*, aprendo possibilità avanzate di calcolo parallelo che superano i limiti dei tradizionali accessi tramite campionamento.

Concetto ed utilizzo

Le funzionalità di **image load/store** permettono di associare le texture a binding point indipendenti, rendendole accessibili come vere e proprie aree di memoria. In questo modo è possibile effettuare letture e scritture **arbitrarie** sui dati immagine all'interno di uno shader, cosa che sarebbe impossibile con i normali **texture samplers**, i quali si limitano a fornire accessi in sola lettura con interpolazione. Grazie a questa caratteristica, le immagini diventano una risorsa flessibile non solo per il rendering, ma anche per il calcolo parallelo di tipo generale, rendendo i compute shader un potente strumento di **GPGPU (General Purpose GPU Computing)**.[13]

Applicazioni tipiche

Le operazioni di image load/store trovano impiego in numerosi scenari pratici. Un caso frequente è la gestione della **trasparenza order-independent**, dove è necessario accumulare e combinare contributi multipli senza rispettare un ordine di disegno predefinito. Altri esempi riguardano l'**accesso in lettura e scrittura** a immagini utilizzate come buffer intermedi per algoritmi di post-processing, come filtri di blur, edge detection o correzioni di colore. Inoltre, questa funzionalità permette di implementare **algoritmi non lineari** basati su texture, come simulazioni fisiche su griglie 2D/3D o tecniche avanzate di **image-based rendering**.

Coerenza e sincronizzazione

A differenza delle operazioni tradizionali su texture o framebuffer, le scritture effettuate tramite **imageStore** non garantiscono automaticamente la coerenza tra le varie invocazioni dello shader.

Questo può portare a letture di dati non aggiornati o a condizioni di race se più thread accedono simultaneamente alla stessa locazione di memoria. Per evitare tali problemi, è necessario inserire manualmente barriere di memoria, come `glMemoryBarrier`, che impongono un ordine nelle operazioni e assicurano la visibilità corretta dei dati tra invocazioni. Questo rende la gestione più complessa ma permette un controllo fine sul **parallelismo** e sull'accesso concorrente.

Strumenti in GLSL

Il linguaggio **GLSL** fornisce funzioni dedicate per la manipolazione diretta delle immagini:

- `imageLoad(...)` per leggere un **texel** da un'immagine in una determinata coordinata;
- `imageStore(...)` per scrivere un nuovo valore elaborato in un texel specifico.

Questi strumenti non dipendono dal filtraggio o dal mipmapping tipico dei samplers, ma operano a livello di memoria grezza. Sono quindi fondamentali all'interno dei compute shader per realizzare operazioni complesse su immagini 2D o volumi 3D, come simulazioni di fluidi, generazione di mappe di illuminazione o calcolo di effetti di occlusione ambientale.

Vantaggi e limiti

L'impiego di **image load/store** consente un controllo diretto e dettagliato sulla memoria immagine, svincolato dal flusso grafico tradizionale e dalle restrizioni della pipeline di rasterizzazione. Questo apre la strada a tecniche ibride che combinano calcolo parallelo e rendering in maniera molto efficiente.

Tuttavia, tale flessibilità comporta anche alcuni limiti: la gestione esplicita della **sincronizzazione** introduce complessità nello sviluppo e può ridurre le prestazioni se non progettata correttamente. Inoltre, non tutte le operazioni sono supportate: in particolare, non è possibile utilizzare direttamente le funzioni **atomiche** sui texel tramite imageStore, limitando quindi la possibilità di implementare aggiornamenti concorrenti senza strategie aggiuntive.

L'uso di image load/store è estremamente potente e rappresenta una delle funzionalità più avanzate dei compute shader, ma richiede attenzione nella programmazione per bilanciare **flessibilità**, **correttezza** e **prestazioni**.

2.6 Implementazione base di un Compute Shader

L'implementazione di un **compute shader in OpenGL** consiste in una sequenza di operazioni che permettono di far comunicare i dati preparati dalla CPU con la logica parallela eseguita sulla GPU. In pratica, la CPU prepara e trasferisce i dati, mentre la GPU si occupa di elaborarli in maniera massicciamente parallela grazie al compute shader scritto in GLSL.

Di seguito vengono descritti i principali passaggi da seguire per implementare un compute shader di base in OpenGL:

Preparazione dei dati su CPU

Il punto di partenza è sempre la CPU, che deve predisporre i dati da elaborare. In una simulazione di particelle, ad esempio, si definiscono gli array contenenti le posizioni e le velocità iniziali di tutte le particelle. Oltre ai dati, è utile memorizzare anche la dimensione complessiva del problema, indicata con N. Questo valore servirà successivamente per determinare quanti gruppi di lavoro (work group) dovranno essere lanciati, così da coprire tutti gli elementi da aggiornare.

Creazione e configurazione degli SSBO

Per rendere disponibili i dati alla GPU, occorre memorizzarli in un buffer OpenGL, in particolare in un **Shader Storage Buffer Object (SSBO)**.
Il seguente frammento di codice mostra i passaggi fondamentali:

```
GLuint posSSBO;
glGenBuffers(1, &posSSBO);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, posSSBO);
glBufferData(GL_SHADER_STORAGE_BUFFER, bufferSize, data, GL_STATIC_DRAW);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, bindingIndex, posSSBO);
```

Con `glGenBuffers` si crea un nuovo buffer, il cui identificativo viene salvato nella variabile `posSSBO`.
`glBindBuffer` associa il buffer appena creato al target `GL_SHADER_STORAGE_BUFFER`, rendendolo il buffer attivo su cui verranno eseguite le operazioni successive.
La chiamata `glBufferData` alloca la memoria per il buffer e, opzionalmente, carica al suo interno i dati iniziali (contenuti nel puntatore `data`). La dimensione riservata è pari a `bufferSize`, mentre il flag `GL_STATIC_DRAW` suggerisce a OpenGL che i dati non cambieranno frequentemente.
Infine, con `glBindBufferBase` il buffer viene legato a un determinato indice di binding

(`bindingIndex`). Questo indice deve coincidere con quello dichiarato nello shader GLSL, in modo che il compute shader sappia a quale buffer accedere.

Scrittura del compute shader (GLSL)

Il compute shader stabilisce la logica di elaborazione: ogni invocazione dello shader lavora su un sottoinsieme dei dati. Ecco un esempio:

```
#version 430
layout(local_size_x = 128) in;

layout(std430, binding = 0) buffer PosBuffer {
    vec4 Positions[];
};

void main() {
    uint id = gl_GlobalInvocationID.x;
    if (id >= Positions.length()) return;
    Positions[id] += vec4(0.0, -0.01, 0.0, 0.0);
}
```

La direttiva `#version 430` indica la versione minima di GLSL necessaria per i compute shader. L'istruzione `layout(local_size_x = 128) in` specifica che ogni gruppo di lavoro locale contiene 128 invocazioni lungo l'asse X. In pratica, ciascun work group calcolerà 128 elementi alla volta.

Il blocco `layout(std430, binding = 0) buffer PosBuffer {vec4 Positions[]}` definisce un SSBO accessibile dallo shader. Il qualificatore `binding = 0` deve corrispondere a quello impostato dalla CPU, e l'array `Positions[]` contiene i dati effettivi (posizioni delle particelle).

All'interno della funzione `main`, la variabile `gl_GlobalInvocationID.x` fornisce l'indice globale dell'invocazione dello shader. Questo indice viene usato per accedere all'elemento corretto dell'array.

La condizione `if (id >= Positions.length()) return;` evita accessi fuori dai limiti qualora ci fossero più invocazioni del numero effettivo di elementi.
Infine, l'operazione `Positions[id] += vec4(0.0, -0.01, 0.0, 0.0)` modifica

la posizione della particella, applicando uno spostamento lungo l'asse Y (ad esempio una semplice forza di gravità).

Compilazione e caricamento dello shader

Il codice GLSL deve essere trasformato in un oggetto eseguibile da OpenGL :

```
GLuint shader = glCreateShader(GL_COMPUTE_SHADER);
glShaderSource(shader, 1, &source, NULL);
glCompileShader(shader);

GLuint program = glCreateProgram();
glAttachShader(program, shader);
glLinkProgram(program);
```

Con `glCreateShader(GL_COMPUTE_SHADER)` si crea un oggetto shader specifico per il calcolo.
`glShaderSource` carica all'interno di tale oggetto il sorgente GLSL (contenuto in `source`).
`glCompileShader` compila il codice, traducendolo in una forma comprensibile dalla GPU.
Si crea poi un oggetto programma (`glCreateProgram`) e vi si collega lo shader compilato tramite `glAttachShader`.
Infine, `glLinkProgram` unisce lo shader al programma, rendendolo pronto per l'esecuzione.

Esecuzione dello shader

Per eseguire il compute shader, occorre attivare il programma e lanciare un dispatch:

```
GLuint numGroupsX = (N + 128 - 1) / 128;
glUseProgram(program);
glDispatchCompute(numGroupsX, 1, 1);
```

La variabile `numGroupsX` calcola il numero di gruppi necessari per coprire tutti gli N elementi, dividendo N per 128 (dimensione del gruppo locale) e arrotondando per eccesso.
Con `glUseProgram(program)` si attiva il programma contenente il compute shader e la chiamata `glDispatchCompute(numGroupsX, 1, 1)` avvia l'esecuzione dello shader, organizzando le invocazioni in una griglia tridimensionale di gruppi (in questo specifico caso di esempio solo lungo X).

Sincronizzazione

Una volta terminata l'elaborazione, bisogna garantire che le scritture sui buffer siano effettivamente completate prima che vengano lette da altre parti del programma (sia da GPU che da CPU).

```
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

La funzione `glMemoryBarrier` forza una barriera di memoria: in questo caso specifico, con il flag `GL_SHADER_STORAGE_BARRIER_BIT`, si assicura che le scritture sugli SSBO siano visibili e correttamente ordinate prima di qualsiasi accesso successivo.

Accesso ai risultati

Dopo la barriera di memoria, i dati aggiornati sono disponibili. È possibile leggerli dalla CPU (ad esempio mappando il buffer con `glMapBuffer`) oppure passarli direttamente ad altri stadi della pipeline grafica. In un'applicazione di rendering particellare, lo stesso SSBO delle posizioni può essere collegato a un vertex shader per disegnare le particelle, evitando così copie ridondanti tra CPU e GPU.

2.7 Sincronizzazione e Operazioni avanzate

In un contesto di calcolo parallelo, dove centinaia di thread operano simultaneamente sugli stessi dati, la sincronizzazione diventa un aspetto cruciale per garantire la correttezza dei risultati. Nei compute shader, le invocazioni appartenenti a uno stesso **work group** possono comunicare tra loro utilizzando variabili dichiarate con il qualificatore `shared`.
Queste variabili risiedono in una memoria locale al gruppo, accessibile da tutte le sue invocazioni, e permettono di condividere informazioni intermedie senza dover ricorrere alla memoria globale della GPU, che sarebbe più lenta da utilizzare.
È importante notare che le variabili `shared` non sono inizializzate automaticamente: il loro contenuto all'avvio dell'esecuzione è indefinito. Per questo motivo, è buona pratica che una sola invocazione del gruppo (tipicamente quella con `gl_LocalInvocationID == 0`) si occupi di assegnare un valore iniziale, prima che gli altri thread le utilizzino.

La sincronizzazione tra invocazioni è garantita da primitive specifiche: una delle più comuni è la funzione `barrier()`, che forza tutte le invocazioni del work group a fermarsi e attendere che le altre abbiano raggiunto lo stesso punto di esecuzione. In questo modo si assicura che tutte le operazioni precedenti siano completate prima che il programma possa proseguire in parallelo. Questa barriera è fondamentale quando più thread devono collaborare su una stessa

struttura dati, evitando condizioni di race.

Un'altra funzione essenziale è `memoryBarrierShared()`, che non ferma l'esecuzione dei thread, ma garantisce la coerenza degli accessi alla memoria condivisa. In pratica, assicura che tutte le scritture effettuate fino a quel punto siano visibili a tutte le invocazioni del gruppo prima che vengano eseguite nuove letture o scritture. Senza questa barriera, potrebbero verificarsi situazioni in cui un thread legge ancora un valore obsoleto, mentre un altro lo ha già aggiornato.

Grazie alla combinazione di **variabili shared, barriere di sincronizzazione e operazioni atomiche**, i compute shader possono implementare algoritmi paralleli sofisticati, come riduzioni, ordinamenti o accumuli distribuiti, mantenendo un controllo fine sul flusso dei dati e sulla loro coerenza.

2.7.1 Operazioni atomiche

Un aspetto particolarmente potente degli SSBO è la possibilità di eseguire **operazioni atomiche**, cioè operazioni che vengono garantite come indivisibili dal punto di vista dell'accesso concorrente alla memoria. In un contesto altamente parallelo, dove centinaia di thread possono tentare di leggere e scrivere la stessa variabile nello stesso istante, le operazioni atomiche evitano condizioni di race e garantiscono la correttezza del risultato.

Le operazioni atomiche sono supportate per tipi interi (`int`, `uint`) e possono essere applicate anche ad elementi di array o a singoli componenti di vettori. Il loro impiego è cruciale in scenari come l'implementazione di contatori globali, la costruzione di istogrammi, la gestione di strutture dati parallele o la sincronizzazione tra thread.
Alcuni esempi tipici sono:

```
atomicAdd(mem[i], value);
atomicMax(mem[i], value);
atomicCompSwap(mem[i], expected, new);
```

Nel primo caso, `atomicAdd` permette a più thread di incrementare una stessa variabile senza perdere aggiornamenti, cosa che accadrebbe con una semplice somma non atomica.
Con `atomicMax`, invece, si può mantenere il massimo globale fra valori calcolati in parallelo.
Infine, `atomicCompSwap` (compare-and-swap) è uno strumento fondamentale per costruire primitive di sincronizzazione più complesse, poiché consente di modificare una variabile solo se contiene un valore atteso.

Tutte queste operazioni restituiscono il valore precedente della variabile su cui agiscono, permettendo così di implementare logiche di controllo sofisticate. Tuttavia, è importante notare che

le operazioni atomiche, pur garantendo la correttezza, hanno un costo in termini di prestazioni: un loro utilizzo eccessivo può ridurre il parallelismo effettivo della GPU. Per questo motivo, vengono impiegate solo nei punti critici dell'algoritmo, laddove non è possibile ricorrere a soluzioni completamente parallele.

2.8 Limitazioni hardware e ottimizzazioni

L'implementazione di compute shader in OpenGL deve necessariamente tener conto di alcune limitazioni imposte dall'hardware e dalle specifiche del driver, le quali influenzano direttamente la progettazione e le prestazioni degli shader.

- **GL_MAX_COMPUTE_WORK_GROUP_COUNT**: definisce il numero massimo di work group che possono essere lanciati lungo ciascuna dimensione (X, Y, Z) della griglia di dispatch. Il valore minimo garantito dalla specifica è 65535 per ogni asse, ma può variare a seconda della GPU. Superare questo limite causa errori di runtime.

- **GL_MAX_COMPUTE_WORK_GROUP_SIZE**: indica la dimensione massima consentita per ciascuna dimensione (X, Y, Z) del singolo work group, ovvero il numero massimo di invocazioni locali (thread) per asse. Valori tipici minimi garantiti sono 1024 per X e Y e 64 per Z. Il prodotto delle dimensioni locali determina il numero totale di thread eseguibili in parallelo all'interno di un work group.

- **GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS**: specifica il numero massimo di invocazioni complessive (thread) consentite in un singolo work group, risultante dal prodotto delle dimensioni locali. Il valore minimo garantito è 1024. Questo parametro vincola la granularità del parallelismo e influisce sull'organizzazione interna del calcolo.

- **GL_MAX_COMPUTE_SHARED_MEMORY_SIZE**: indica la quantità massima di memoria condivisa (shared memory) disponibile per ciascun work group, espressa in byte. Il minimo previsto è 32 KB, ma alcune GPU possono offrire valori superiori. La memoria condivisa permette la comunicazione e la sincronizzazione efficiente tra le invocazioni all'interno del work group, ma è una risorsa limitata che deve essere gestita con attenzione per evitare colli di bottiglia.

Questi vincoli sono determinanti nella definizione dell'architettura dello shader: dimensioni troppo grandi dei work group o un uso eccessivo della memoria condivisa possono portare a malfunzionamenti, cali di prestazioni o impossibilità di compilare correttamente lo shader.
Per questo motivo, è buona pratica interrogare i valori supportati dalla GPU in fase di inizializzazione tramite chiamate come `glGetIntegeri_v()` e adattare dinamicamente la configurazione dello shader alle risorse hardware disponibili.

Inoltre, ottimizzazioni come il bilanciamento della dimensione dei work group, la minimizzazione dell'uso della memoria condivisa e la riduzione delle dipendenze tra thread sono fondamentali per ottenere prestazioni elevate e scalabilità nei sistemi di calcolo parallelo basati su compute shader.

2.9 Conclusione

I compute shader costituiscono uno strumento centrale per la grafica contemporanea e per il calcolo parallelo su GPU, poiché consentono di sfruttare appieno la capacità di elaborazione massiva delle schede video. La loro introduzione ha reso possibile lo sviluppo di simulazioni complesse e ad alte prestazioni, che si estendono ben oltre l'ambito della semplice generazione di immagini e includono settori come la fisica computazionale, la dinamica dei fluidi, l'intelligenza artificiale e molte altre forme di elaborazione dati.

Questa tecnologia risulta particolarmente adatta ad applicazioni di interesse cinematografico, scientifico e ingegneristico, dove la possibilità di gestire un elevato numero di elementi con precisione e rapidità rappresenta un requisito fondamentale. Inoltre, l'uso dei compute shader introduce un livello di modularità e flessibilità che agevola l'integrazione di nuovi algoritmi e metodologie, favorendo lo sviluppo di sistemi scalabili e facilmente estendibili.

In questo senso, i compute shader non sono soltanto un passo in avanti per l'evoluzione della computer grafica, ma anche una piattaforma versatile per la computazione parallela, destinata ad avere un impatto significativo in molteplici ambiti della tecnologia moderna.

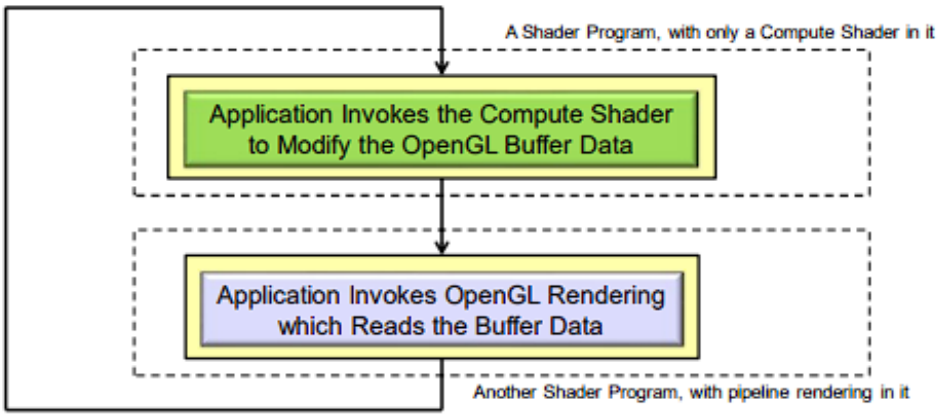

Figura 2.3: Schema del flusso di lavoro: un programma contenente il compute shader aggiorna i dati nei buffer OpenGL, mentre un programma grafico successivo legge gli stessi buffer per eseguire il rendering.

Capitolo 3

Background matematico degli effetti particellari

La capacità di simulare fenomeni complessi e volumetrici come fuoco, fumo, esplosioni, o agenti atmosferici quali pioggia e neve, in ambito cinematografico e videoludico, è resa possibile grazie ai **sistemi particellari**. Questa tecnica, fondamentale per superare i limiti della modellazione geometrica tradizionale, rappresenta entità complesse tramite un insieme dinamico di **particelle discrete** . Mentre il Capitolo 1 ha introdotto la natura dei sistemi particellari e il loro impiego, questo capitolo si addentra nei **principi matematici e fisici** che ne governano il comportamento dinamico e l'interazione con l'ambiente, aspetti cruciali per ottenere il realismo visivo richiesto nelle produzioni cinematografiche ad alta fedeltà.

3.1 Modello computazionale della singola particella

Ogni singola particella, pur essendo un elemento visivamente semplice (spesso rappresentata come una piccola sprite o un billboard), nasconde una complessità notevole a livello computazionale e fisico. Il comportamento di ogni particella viene definito attraverso un insieme articolato di **attributi fisici e visivi** che ne governano la traiettoria, l'interazione con l'ambiente circostante e l'aspetto durante tutto il suo ciclo di vita. La gestione simultanea di milioni di queste entità in parallelo rappresenta la vera sfida computazionale alla base delle simulazioni particellari realistiche.[20]

Gli attributi principali di ogni particella includono:

- **Posizione** ($\vec{x} \in \mathbb{R}^3$): vettore tridimensionale che definisce la posizione esatta della particella nello spazio virtuale, fondamentale per il calcolo delle interazioni e delle collisioni.
- **Velocità** ($\vec{v} \in \mathbb{R}^3$): vettore che indica la direzione e la velocità con cui la particella si muove, influenzato dalle forze esterne e interne al sistema.

- **Accelerazione** ($\vec{a} \in \mathbb{R}^3$): tasso di variazione della velocità, risultante dall'applicazione delle forze che agiscono sulla particella in ogni istante temporale.
- **Massa** ($m \in \mathbb{R}^+$): grandezza scalare che rappresenta l'inerzia della particella, determinando come essa risponde alle forze applicate.
- **Vita residua** (lifetime): durata temporale residua della particella, che determina quanto a lungo essa sarà visibile e attiva nella simulazione prima di essere rimossa.
- **Parametri estetici**: comprendono il colore (con canale alfa per la trasparenza), la dimensione, la forma (tipicamente sprite o mesh 3D) e l'orientamento (rotazione), tutti elementi che influenzano l'aspetto finale e l'impatto visivo.

L'evoluzione temporale degli attributi delle particelle è controllata tramite *lifetime curves*, funzioni che modulano gradualmente proprietà visive come opacità, scala e colore per ottenere effetti dinamici realistici.
Dal punto di vista fisico, il movimento delle particelle è regolato dalla seconda legge di Newton:

$$\vec{F} = m\vec{a} = m\frac{d\vec{v}}{dt}$$

che collega la forza agente all'accelerazione, mentre la posizione si aggiorna secondo:

$$\frac{d\vec{x}}{dt} = \vec{v}$$

Poiché risolvere queste equazioni in forma analitica per calcolare la posizione è complesso, si ricorre a metodi di discretizzazione numerica, adatti anche all'esecuzione parallela su GPU.
La sfida computazionale consiste nell'aggiornare in tempo reale milioni di particelle, ognuna con caratteristiche e vincoli propri, bilanciando realismo visivo e prestazioni per applicazioni real-time tipiche di cinema e videogiochi.

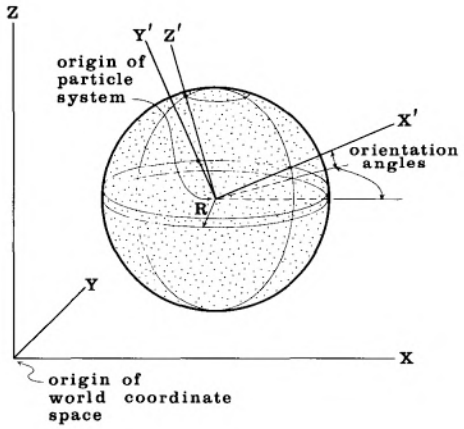

Figura 3.1: Rappresentazione schematica degli attributi spaziali e di orientamento fondamentali di una particella all'interno del sistema.

3.2 Dinamica e generazione: il ruolo dell'emitter

L'**emitter** è l'origine funzionale del sistema particellare, la sorgente da cui vengono generate dinamicamente le particelle secondo parametri configurabili. Non si limita a definire un punto di emissione, ma rappresenta una vera e propria interfaccia tra il modello fisico della simulazione e la distribuzione statistica delle proprietà iniziali delle particelle.

Tra le caratteristiche principali di un emitter sono da menzionare:

- **Posizione e forma**: l'emissione può avvenire da un punto, una linea, una superficie o un volume tridimensionale, e spesso coincide con una geometria della scena.
- **Frequenza di emissione**: definisce il numero di particelle generate per unità di tempo e può variare in funzione di eventi o condizioni ambientali.
- **Velocità e direzione iniziale**: vettori che indicano la quantità e la direzione di moto delle particelle al momento della loro nascita; sono spesso soggette a variazioni pseudo-casuali per evitare movimenti troppo uniformi e creare così un effetto più naturale.
- **Lifetime e attributi iniziali**: come durata, scala, opacità, assegnati in fase di emissione con una componente stocastica, per garantire varietà e realismo.

Particolarmente utile nella simulazione cinematografica è l'uso di **funzioni di noise** (come Perlin o Simplex), che permettono di generare fluttuazioni continue e coerenti nello spazio-tempo. Queste funzioni vengono impiegate per modulare direzione, intensità o densità dell'emissione, riproducendo effetti naturali come vento, turbolenza, o l'irregolarità del fumo che si dirama da una sorgente.

3.3 Forze fisiche e interazioni

Le particelle sono soggette a diverse forze che ne influenzano la traiettoria nel tempo.
Alcune sono forze fisiche reali (come la gravità), mentre altre sono modellate per simulare fenomeni ambientali complessi (come vento, turbolenza, attrazione locale, ecc.).

Forze base

- **Forza di Gravità**:

  $$\vec{F_g} = m\vec{g}$$

  dove m rappresenta la massa della particella e $\vec{g}$ è l'accelerazione gravitazionale, generalmente costante e orientata verso il basso. È essenziale per simulare effetti come caduta di pioggia, polvere o detriti.

- **Forza d'Attrito** (resistenza del mezzo):

$$\vec{F}_d = -k_d \vec{v}$$

Una forza di smorzamento proporzionale alla velocità, dove k_d è il coefficiente di resistenza del mezzo e $\vec{v}$ rappresenta la velocità della particella. Il segno negativo indica che la forza agisce in direzione opposta al moto, simulando così l'effetto dell'aria o di un fluido che frena progressivamente il movimento della particella nel tempo.

- **Forza Elastica** (Legge di Hooke):

$$\vec{F}_{spring} = -k_s(|\vec{l}| - r_0)\hat{l} - k_d \left(\frac{(\vec{v}_a - \vec{v}_b) \cdot \hat{l}}{|\vec{l}|} \right) \hat{l}$$

Questa forza viene utilizzata per connettere due particelle con un comportamento elastico, come nel caso di tessuti, catene o peli. Nella formula, k_s rappresenta la costante elastica che regola la rigidità della connessione, $\vec{l}$ è il vettore che unisce le due particelle, r_0 indica la lunghezza a riposo della molla e $\hat{l}$ è il versore del vettore $\vec{l}$, cioè la sua direzione normalizzata. Le quantità $\vec{v}_a$ e $\vec{v}_b$ corrispondono alle velocità delle due particelle collegate, mentre k_d è il coefficiente di smorzamento viscoelastico che introduce una resistenza proporzionale alla velocità relativa lungo la direzione della molla. Il primo termine descrive la componente elastica secondo la legge di Hooke, mentre il secondo rappresenta l'azione dello smorzamento, che riduce progressivamente le oscillazioni rendendo il sistema più stabile.

Forze avanzate

- **Turbulenza / Vento**:

$$\vec{F}_{wind} = C_{\text{wind}} \cdot \text{noise}(\vec{x}, t)$$

Viene utilizzata per simulare disturbi atmosferici complessi, come correnti d'aria irregolari, vortici o turbolenze. Nella formula, C_{wind} rappresenta il coefficiente che regola l'intensità complessiva dell'effetto, mentre la funzione $\text{noise}(\vec{x}, t)$ genera variazioni coerenti nello spazio e nel tempo in base alla posizione $\vec{x}$ della particella e all'istante temporale t. In questo modo si ottengono movimenti irregolari ma continui, che riproducono il comportamento caotico di fenomeni naturali quali vento e fumo.

- **Forza di Attrazione / Repulsione**:

$$\vec{F}_{\text{attraction}} = k_{\text{attr}} \frac{\vec{P} - \vec{x}}{|\vec{P} - \vec{x}|^2}$$

Agisce spingendo le particelle verso o lontano da un punto di riferimento P. Nella formula, k_{attr} è la costante che determina l'intensità dell'attrazione o della repulsione, $\vec{x}$ rappresenta la posizione della particella e $\vec{P}$ è la posizione del punto di riferimento. Il denominatore $|\vec{P} - \vec{x}|^2$ indica che l'intensità della forza decresce con il quadrato della distanza, in modo analogo a fenomeni fisici reali come la gravità o l'elettrostatica. Questa formulazione è utile per realizzare effetti come raccolte di particelle, esplosioni inverse o attrattori artificiali all'interno della simulazione.

- **Interazioni N-body**: Modelli complessi che considerano l'influenza reciproca tra tutte le particelle. Sono fondamentali per simulazioni fisicamente accurate (come sistemi gravitazionali), ma computazionalmente costose. L'uso di strutture di accelerazione spaziale come griglie uniformi, alberi k-d o octree riduce la complessità da $O(n^2)$ a $O(n \log n)$.

3.4 Collisioni e reazioni

Il trattamento delle collisioni è essenziale per rendere credibili le simulazioni particellari in ambienti fisici complessi. Le particelle possono interagire con geometrie statiche (es. pavimenti, pareti, mesh) o con altre particelle. Una gestione corretta delle collisioni contribuisce sia al realismo visivo che alla stabilità numerica del sistema.

Collisione con superfici

Quando una particella entra in contatto con una superficie, il sistema esegue i seguenti passaggi:

1. **Rilevamento del contatto:** verifica se la posizione della particella ha superato un limite definito dalla geometria (es. piano $y = 0$).
2. **Calcolo della normale e punto d'impatto:** si determina la normale alla superficie nel punto di collisione, indicata con $\hat{n}$, fondamentale per il calcolo della riflessione.
3. **Calcolo della velocità riflessa:**

$$\vec{v}_r = \vec{v} - (1 + e)(\vec{v} \cdot \hat{n})\hat{n}$$

   dove e è il coefficiente di elasticità (con $e = 1$ per urti elastici, $e < 1$ per urti dissipativi).
4. **Correzione della posizione:** per evitare che la particella rimanga all'interno della superficie, viene traslata lungo la normale fino a uscire dalla zona di penetrazione.

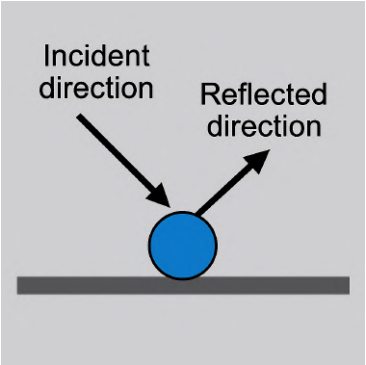

Figura 3.2: Collisione tra una particella e una superficie piana: la direzione incidente viene riflessa secondo la normale $\hat{n}$ al punto di contatto, con un coefficiente di restituzione e.

Collisioni tra particelle

Nei sistemi ad alta densità, le particelle possono entrare in contatto tra loro. Queste interazioni possono essere gestite in vari modi:

- **Forze repulsive:** quando due particelle si avvicinano oltre una certa soglia, si applica una forza inversamente proporzionale alla distanza per simulare un rimbalzo.
- **Risoluzione tramite impulsi:** si calcola la variazione di velocità causata dall'urto secondo le leggi della conservazione dell'energia e della quantità di moto.
- **Accelerazione spaziale:** per evitare controlli $O(n^2)$, si utilizzano strutture come griglie uniformi, alberi k-d o octree per individuare le coppie di particelle realmente vicine.

3.5 Integrazione numerica delle equazioni del moto

L'integrazione numerica consente di aggiornare posizione e velocità delle particelle nel tempo, risolvendo le equazioni del moto in forma discretizzata. La scelta del metodo di integrazione dipende dal compromesso tra accuratezza, stabilità e costo computazionale.

Metodo di Eulero (esplicito) [21]

Uno dei metodi più semplici e diretti:

$$\vec{x}_{t+\Delta t} = \vec{x}_t + \frac{\vec{v}_t}{\Delta t}$$

$$\vec{v}_{t+\Delta t} = \vec{v}_t + \frac{\vec{F}_t}{m} \cdot \Delta t$$

Sebbene computazionalmente leggero, può risultare instabile con Δt troppo grandi o forze intense. Per questo motivo è spesso utilizzato solo in contesti controllati, come simulazioni con forte dissipazione numerica o effetti a breve durata visiva.

Metodo di Eulero semi-esplicito [22]

Una variante molto diffusa e leggermente più stabile del metodo esplicito è il cosiddetto **Eulero semi-esplicito**. In questo approccio, la velocità viene aggiornata per prima utilizzando le forze note al tempo t, e successivamente la nuova posizione viene calcolata impiegando direttamente la velocità aggiornata $\vec{v}_{t+\Delta t}$:

$$\vec{v}_{t+\Delta t} = \vec{v}_t + \frac{\vec{F}_t}{m} \cdot \Delta t$$
$$\vec{x}_{t+\Delta t} = \vec{x}_t + \vec{v}_{t+\Delta t} \cdot \Delta t$$

Rispetto all'Eulero esplicito, in cui la posizione viene aggiornata con la velocità precedente, questo metodo offre una maggiore stabilità numerica, soprattutto nelle simulazioni che coinvolgono forze conservative come gravità o elasticità.
Nel progetto è stato adottato proprio questo schema, in quanto consente di mantenere le traiettorie dei frammenti stabili anche in presenza di tempi di integrazione relativamente grandi. Inoltre, l'algoritmo è stato arricchito con un termine di smorzamento esponenziale:

$$\vec{v}_{t+\Delta t} = (\vec{v}_t + \vec{g}\,\Delta t) \cdot e^{-k\Delta t}$$

dove k rappresenta il coefficiente di resistenza dell'aria. Questo consente di ridurre progressivamente l'energia del sistema, simulando in maniera realistica l'effetto dissipativo del mezzo e impedendo che i frammenti accelerino indefinitamente.
Il metodo di Eulero semi-esplicito costituisce quindi un buon compromesso tra semplicità computazionale e stabilità, risultando particolarmente adatto alle simulazioni in tempo reale basate su compute shader, come quella implementata in questo progetto.

Metodo di Verlet [23]

Molto usato nei motori fisici e nei videogiochi grazie alla sua stabilità numerica e al basso costo computazionale:

$$\vec{x}_{t+\Delta t} = 2\vec{x}_t - \vec{x}_{t-\Delta t} + \vec{a}_t \cdot \Delta t^2$$

Non richiede il calcolo esplicito della velocità, ma quest'ultima può essere approssimata. È ideale per simulazioni basate sulla posizione (position-based dynamics), come tessuti, corde o sistemi vincolati.

Metodo di Runge–Kutta del 4° ordine (RK4) [24]

Dato il problema ai valori iniziali

$$\dot{\vec{x}} = \vec{f}(t, \vec{x}), \qquad \vec{x}(t_0) = \vec{x}_0, \qquad h \equiv \Delta t > 0,$$

si definisce la ricorrenza

$$\vec{x}_{n+1} = \vec{x}_n + \frac{h}{6}\big(k_1 + 2k_2 + 2k_3 + k_4\big), \qquad t_{n+1} = t_n + h,$$

dove gli *incrementi* k_1, k_2, k_3, k_4 sono le stime della derivata $\vec{f}$ ai punti intermedi dell'intervallo $[t_n, t_{n+1}]$:

$$\begin{aligned} k_1 &= \vec{f}\big(t_n,\ \vec{x}_n\big), \\ k_2 &= \vec{f}\big(t_n + \tfrac{h}{2},\ \vec{x}_n + \tfrac{h}{2}\,k_1\big), \\ k_3 &= \vec{f}\big(t_n + \tfrac{h}{2},\ \vec{x}_n + \tfrac{h}{2}\,k_2\big), \\ k_4 &= \vec{f}\big(t_n + h,\ \vec{x}_n + h\,k_3\big). \end{aligned}$$

Nel calcolo della media pesata, i contributi valutati a metà intervallo hanno peso doppio.
In particolare:

- k_1 è la pendenza all'inizio dell'intervallo ($t = t_n$);
- k_2 è la pendenza a metà intervallo usando un primo passo di Eulero con k_1;
- k_3 è un'ulteriore pendenza a metà intervallo usando k_2;
- k_4 è la pendenza alla fine dell'intervallo ($t = t_n + h$) usando k_3.

Il metodo RK4 offre una migliore conservazione dell'energia e minori errori locali, ma è più oneroso computazionalmente. Per questo è utilizzato principalmente in simulazioni offline o rendering precomputati in ambito cinematografico, dove la qualità visiva è prioritaria rispetto al frame rate.

3.6 Riflessioni finali

La comprensione approfondita dei principi fisici e numerici alla base dei sistemi particellari è essenziale per ottenere simulazioni coerenti, controllabili e visivamente credibili. Ogni forza, modello di interazione e metodo di integrazione ha un impatto diretto sul comportamento delle particelle e sulla qualità finale dell'effetto visivo.

Questi fondamenti teorici non sono solo strumenti di modellazione, ma anche leve creative. La capacità di decidere quando semplificare o approfondire un modello fisico è una competenza

essenziale, soprattutto nel campo degli effetti visivi, dove è sempre necessario bilanciare realismo e prestazioni.

Un aspetto cruciale per la realizzazione pratica di tali simulazioni, soprattutto in ambito cinematografico e videoludico, è l'implementazione efficiente su GPU. Le architetture parallele offerte dai *compute shader* permettono di gestire milioni di particelle simultaneamente, sfruttando al meglio la potenza computazionale disponibile. Questo consente di superare i limiti delle simulazioni tradizionali basate su CPU, garantendo al contempo prestazioni in tempo reale o quasi real-time necessarie nelle pipeline di produzione VFX moderne.

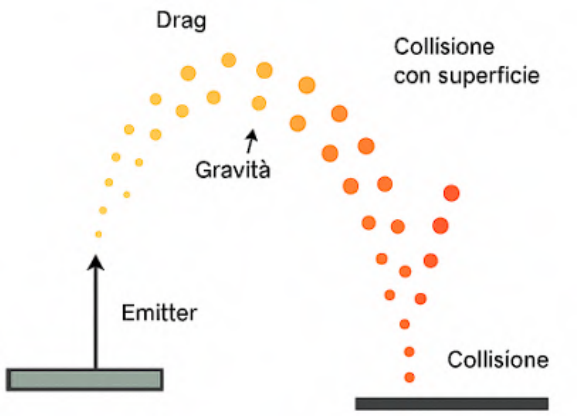

Figura 3.3: Evoluzione di un sistema particellare: emissione dall'emitter, traiettorie influenzate da forze fisiche e collisione con una superficie.

Capitolo 4

Sviluppo e implementazione del simulatore

In questo capitolo vengono illustrate nel dettaglio le fasi di realizzazione del simulatore, a partire dall'idea iniziale fino alle soluzioni implementative finali. Dopo aver presentato l'evoluzione del progetto e le tecnologie adottate, si descrivono i processi di caricamento e gestione della scena, il controllo del personaggio e della telecamera, e le varie tecniche utilizzate per il rilevamento delle collisioni. Una sezione significativa è dedicata alla fisica delle esplosioni e all'uso dei compute shader, che consentono di sfruttare la potenza della GPU per ottenere simulazioni più complesse ed efficienti. Infine, viene discusso un confronto tra le prestazioni della GPU e della CPU, utile a valutare i benefici delle diverse strategie adottate.

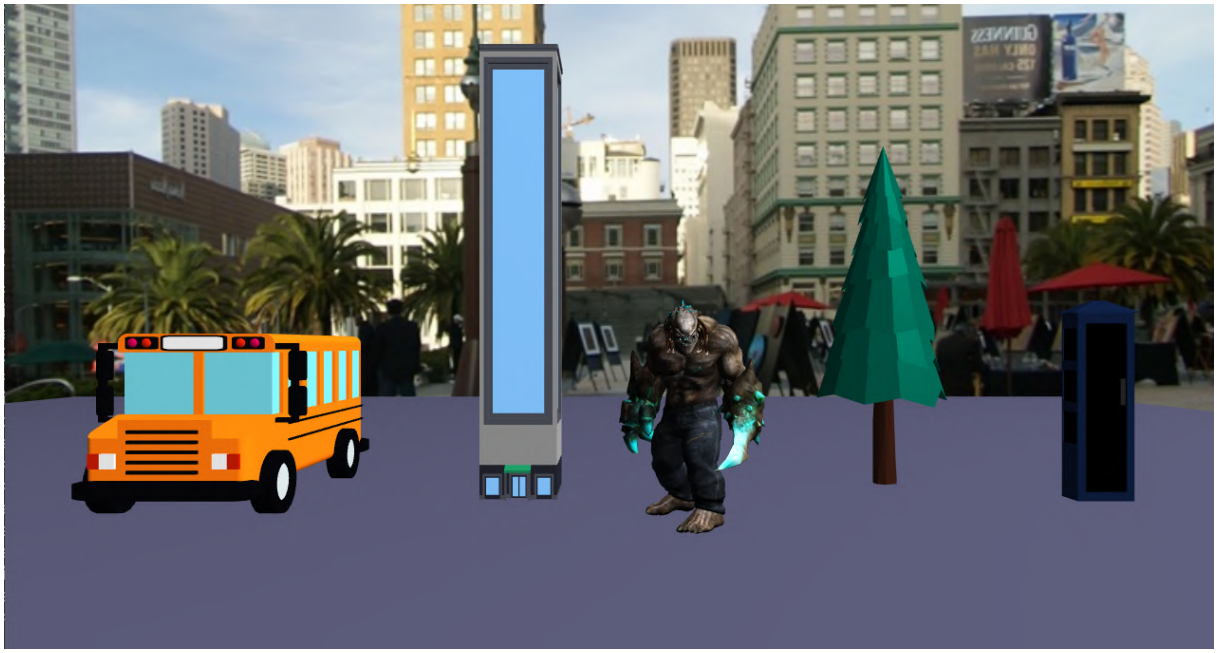

Figura 4.1: Schermata principale del progetto di tesi.

4.1 Idea iniziale e sviluppo progressivo del progetto

Il progetto si propone di sviluppare un simulatore interattivo 3D capace di riprodurre in tempo reale esplosioni e crolli architettonici. L'obiettivo è creare un ambiente realistico ma computazionalmente efficiente, nel quale un edificio possa frammentarsi e i detriti si distribuiscano nello spazio seguendo leggi fisiche semplificate. Oltre all'aspetto visivo, il progetto punta a sperimentare e applicare tecniche di calcolo parallelo per gestire dinamicamente la simulazione fisica, sfruttando la potenza della GPU per ottimizzare prestazioni e realismo.

Lo sviluppo ha seguito una progressione graduale: in una prima fase è stato costruito un prototipo di base, comprendente la finestra di rendering, i controlli della telecamera e primi tentativi di simulatore particellare.
Successivamente si è passati al caricamento di modelli più complessi, come il palazzo e l'albero, preparati con la tecnica del *cell fracture* in Blender per poterli suddividere in frammenti. Questo ha permesso di introdurre la transizione dal modello integro alla sua versione distrutta attraverso un input da tastiera, rendendo la simulazione più realistica.

Un punto centrale del progetto è stata l'introduzione della fisica: per descrivere il movimento dei frammenti, dopo vari tentativi, si è scelto un metodo di integrazione numerica semplice ma efficace, cioè l'Eulero semi-esplicito con smorzamento, in grado di tenere conto di forze come la gravità, la spinta radiale dell'esplosione e l'attrito dell'aria. Questa scelta ha garantito una buona stabilità della simulazione pur mantenendo la leggerezza dei calcoli, aspetto essenziale per il rendering in tempo reale.

Con il procedere delle fasi sono stati aggiunti altri elementi che hanno contribuito a dare completezza al simulatore: un sistema di materiali per migliorare la resa visiva, uno skybox per contestualizzare la scena,e strumenti di monitoraggio delle prestazioni per analizzare l'impatto delle varie soluzioni implementative. Infine, è stato pensato anche un livello di interattività, con la possibilità di controllare da tastiera un personaggio, il quale funge da innesco per le esplosioni e i crolli degli oggetti in scena.

Dall'insieme di queste riflessioni nasce l'idea di progetto finale: una scena 3D interattiva dove un personaggio, controllato dall'utente, si muove all'interno di un ambiente urbano popolato da vari oggetti come autobus, palazzi, alberi e cabine telefoniche. Quando il personaggio entra in contatto con uno di questi oggetti, si attiva un'animazione "punching" in cui egli sferra un pugno all'oggetto, causando la sua frantumazione in molteplici pezzi che poi cadono sul piano di base e realizzando un'esplosione realistica.[27].

4.2 Tecnologie utilizzate

Lo sviluppo del simulatore ha richiesto l'integrazione di diverse tecnologie, sia per la parte grafica che per la gestione dei modelli e delle dipendenze esterne. La selezione delle librerie non è stata casuale, ma il risultato di un processo di valutazione basato su criteri di compatibilità, stabilità e diffusione nel settore della grafica in tempo reale. L'obiettivo principale è stato creare un ambiente modulare e scalabile, capace di garantire elevate prestazioni computazionali mantenendo al contempo una struttura del codice chiara, organizzata e facilmente estendibile per futuri sviluppi o aggiunte funzionali.

OpenGL 4.3[29]

OpenGL rappresenta il nucleo grafico del progetto, ovvero l'insieme di strumenti che consente di comunicare direttamente con la scheda video per eseguire operazioni di rendering e calcolo. La versione 4.3 è stata scelta poiché introduce una delle innovazioni più rilevanti per il progetto: i *compute shader*. Questi permettono di sfruttare la GPU non solo per la grafica tradizionale, ma anche per il calcolo parallelo, rendendo possibile la simulazione della fisica dei frammenti in tempo reale.
Oltre a questa caratteristica, OpenGL fornisce i meccanismi fondamentali per la gestione della pipeline grafica: caricamento dei modelli, applicazione dei materiali, illuminazione dinamica e gestione dei buffer. La sua diffusione e la disponibilità di ampia documentazione hanno reso questa tecnologia la scelta più solida per il progetto.

GLFW [30]

GLFW è la libreria che si occupa della creazione della finestra, della gestione del contesto OpenGL e dell'elaborazione degli input da tastiera e mouse. La sua adozione è stata dettata dalla semplicità di integrazione e dalla leggerezza, che la rendono ideale per progetti che richiedono un controllo diretto sull'API grafica senza la complessità di framework più pesanti.
In questo simulatore, GLFW ha consentito di implementare un sistema di input modulare e intuitivo, permettendo all'utente di interagire con la scena attraverso comandi da tastiera e movimenti della telecamera.

GLAD[31]

Per poter utilizzare le funzioni più recenti di OpenGL, è necessario un meccanismo che ne gestisca il caricamento dinamico, in quanto non tutte le versioni delle librerie grafiche fornite dal sistema operativo espongono le stesse funzionalità. A questo scopo è stata impiegata la

libreria GLAD, che si occupa di caricare in modo trasparente i puntatori alle funzioni OpenGL necessarie al progetto.
Grazie a GLAD, è stato possibile garantire la portabilità del simulatore su diverse configurazioni hardware e software, riducendo i problemi di compatibilità legati ai driver grafici.

GL e KHR headers[32]

Un ulteriore supporto tecnico è stato fornito dai file di intestazione standard GL e KHR, che definiscono costanti, macro e prototipi delle funzioni OpenGL. Questi header costituiscono il ponte tra le specifiche ufficiali e il codice sorgente, garantendo coerenza e chiarezza nell'utilizzo delle API grafiche. Pur essendo meno visibili rispetto ad altre librerie, la loro presenza è imprescindibile per la corretta compilazione e manutenzione del progetto.

Assimp (Open Asset Import Library)[33]

La gestione dei modelli tridimensionali è stata affidata ad Assimp, libreria che supporta un ampio numero di formati tra cui `.obj`, scelto per la compatibilità con Blender. Grazie ad Assimp, il simulatore è in grado di caricare sia i modelli integri che quelli frantumati tramite la tecnica del *cell fracture*, senza necessità di sviluppare un parser proprietario.
Un vantaggio significativo di Assimp è la capacità di gestire non solo le geometrie, ma anche i materiali associati ai modelli. Questo ha permesso di mantenere un legame diretto tra i file scaricati o esportati da Blender e la scena caricata dal simulatore, riducendo tempi e complessità di sviluppo.

GLM (OpenGL Mathematics)[34]

Per quanto riguarda la parte matematica, è stata adottata GLM, libreria header-only progettata per replicare la sintassi e la logica di GLSL. Essa fornisce strumenti per la gestione di vettori, matrici e quaternioni, semplificando notevolmente le operazioni di trasformazione geometrica e calcolo delle proiezioni prospettiche.
L'utilizzo di GLM si è rivelato particolarmente efficace per mantenere coerenza tra il codice C++ e gli shader scritti in GLSL, riducendo le possibilità di errore e rendendo più intuitiva l'integrazione delle due parti. Grazie a questa libreria, è stato possibile implementare con facilità funzioni di scaling, rotazione e traslazione dei modelli caricati nella scena.

La maggior parte dei pacchetti è stata scaricata dai rispettivi siti ufficiali e inserita in una cartella delle dipendenze, suddivisa in due sottocartelle: `include` e `lib`, contenenti i file di supporto necessari.
Una volta predisposte le directory, è stato necessario collegare i file di supporto in Visual Studio eseguendo i seguenti passaggi (dal file di progetto → *Proprietà*):

1. **Include:** `Proprietà di configurazione` → `C/C++` → `Generale` → `Directory di inclusione aggiuntive:`
   Aggiunto il percorso alla cartella `include`.

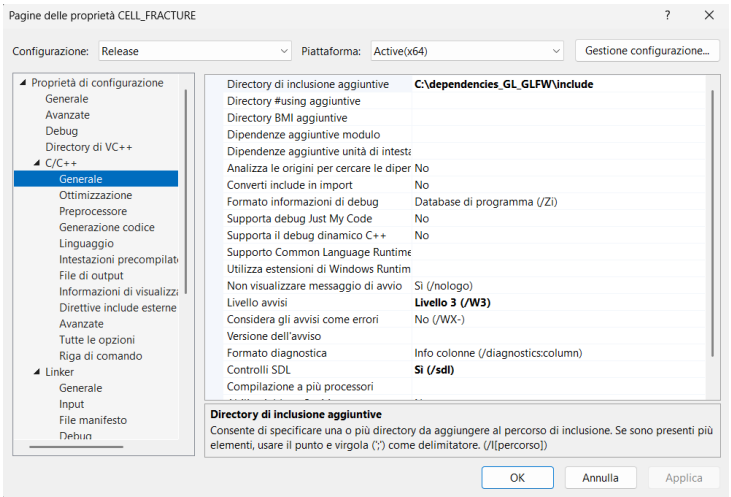

Figura 4.2: Percorso di dipendenze per gli Include

2. **Lib:** `Proprietà di configurazione` → `Linker` → `Generale` → `Directory di libreria aggiuntive`
   Aggiunto il percorso alla cartella `lib`.

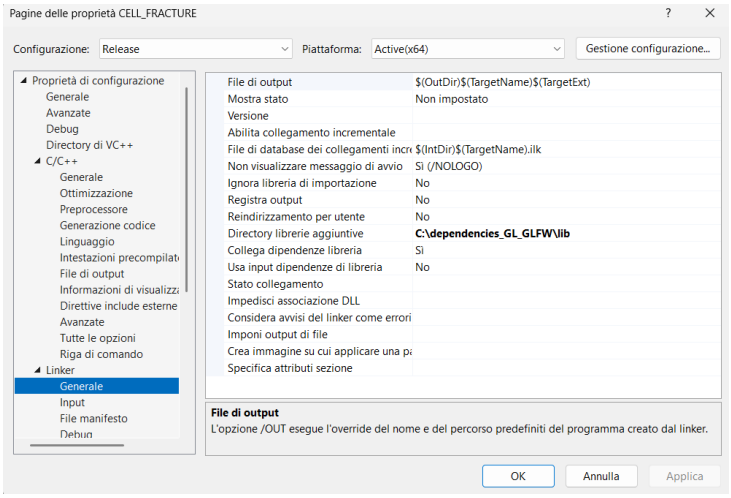

Figura 4.3: Percorso per le dipendenze per i Lib

3. **Dipendenze Aggiuntive:** `Proprietà di configurazione` → `Linker` → `Input` → `Dipendenze aggiuntive`
   Elenco dei file `.lib` aggiuntivi richiesti.

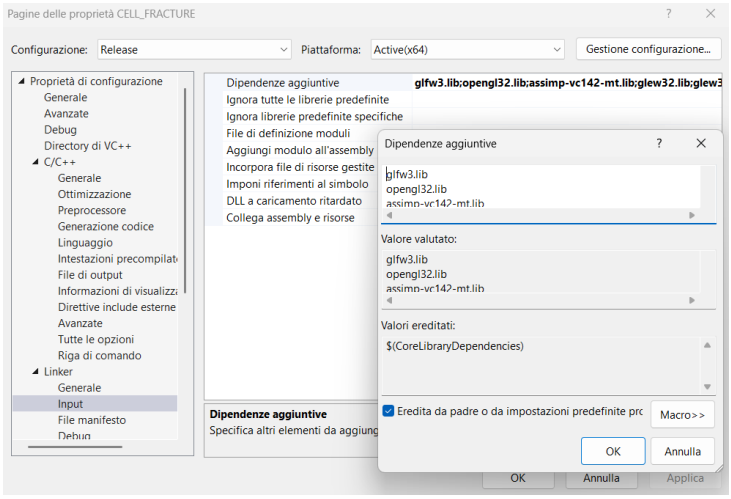

Figura 4.4: Percorso per le dipendenze aggiuntive

Grazie a questa combinazione di strumenti è stato possibile costruire un ambiente di sviluppo modulare e flessibile, dove ciascuna libreria svolge un ruolo ben definito all'interno del progetto. Questa architettura ha reso possibile integrare progressivamente nuove funzionalità, come la simulazione fisica parallela e la gestione interattiva della scena.

4.3 Reperimento dei modelli obj e Cell Fracture su Blender

Per reperire gli oggetti e il personaggio da inserire nella scena, è stato necessario esplorare diverse piattaforme che offrono modelli 3D gratuiti. Dopo una fase preliminare di ricerca e test, la mia scelta è ricaduta sul sito *PolyPizza*, che mette a disposizione un ampio catalogo di modelli low-poly scaricabili in formato `.obj`. Questa tipologia di modelli si è rivelata particolarmente adatta al progetto, poiché la semplicità geometrica delle mesh riduce il carico computazionale e agevola il processo di frantumazione necessario per simulare l'esplosione e il crollo degli oggetti.

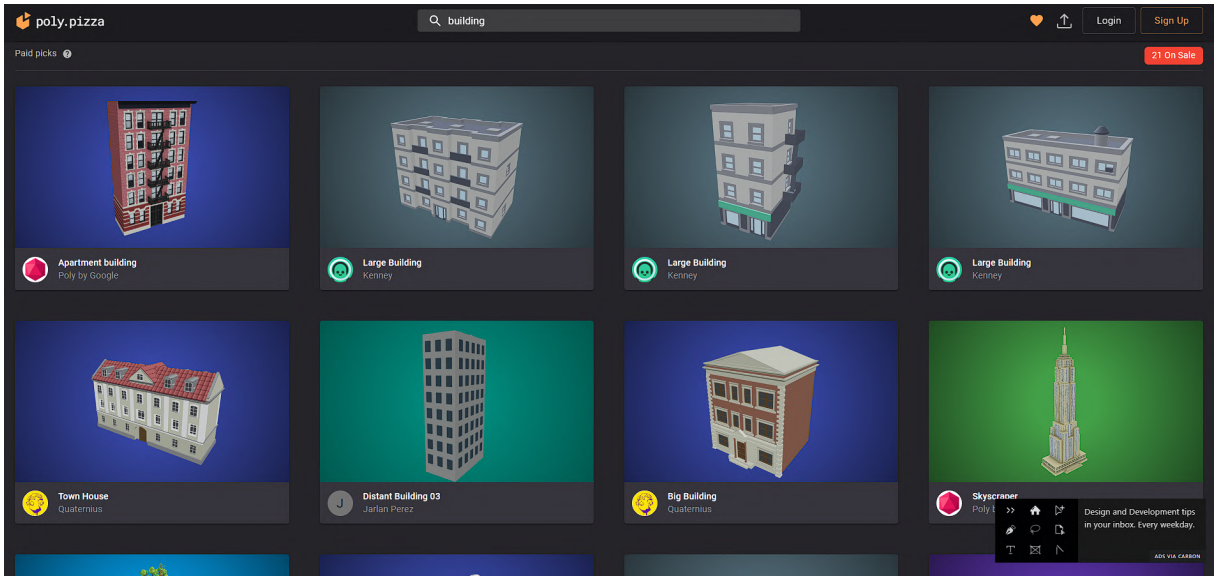

Figura 4.5: Il sito Polypizza, dove sono stati reperiti gli oggetti.

Il file .mtl (Material Template Library) associato a ogni modello 3D contiene informazioni sui materiali e sui colori di base degli oggetti, come proprietà di colore, lucentezza, trasparenza e mappature delle texture. In questo modo, l'importazione degli oggetti nel simulatore non ha richiesto interventi manuali complessi, garantendo fin da subito una resa visiva coerente con lo stile scelto. La disponibilità di file già pronti ha permesso di concentrarsi maggiormente sulla parte fisica e interattiva del progetto, senza dover investire tempo nella modellazione da zero.

Per preparare i modelli alla simulazione di distruzione è stato necessario ricorrere a *Blender*, software open-source per la modellazione e l'animazione 3D. In particolare, si è fatto uso dell'estensione *Cell Fracture*, che non è sempre inclusa di default e deve quindi essere installata manualmente. Questa estensione consente di suddividere un oggetto in un numero variabile di frammenti, generando automaticamente una mesh composta da parti indipendenti.

La procedura adottata per ciascun oggetto è stata sistematica:

1. Importazione del modello in formato `.obj` in Blender.

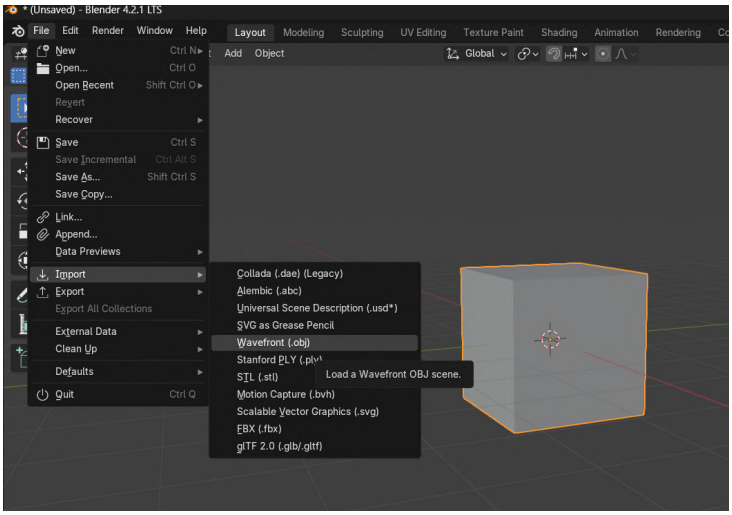

Figura 4.6: Importazione del modello in formato `.obj` in Blender

2. Selezione dell'oggetto e applicazione dell'effetto *Cell Fracture* dal menu `Object` → `Quick Effects` → `Cell Fracture`.

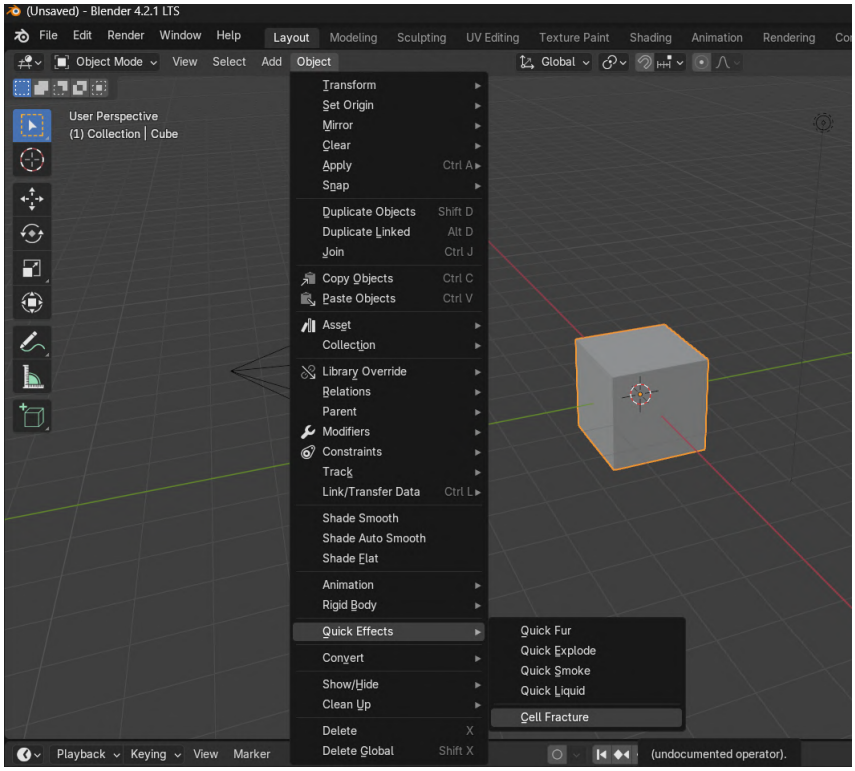

Figura 4.7: Selezione dell'oggetto e applicazione dell'effetto *Cell Fracture*

3. Utilizzo delle impostazioni di default per la maggior parte degli oggetti (circa un centinaio di frammenti), con l'unica eccezione del palazzo, per il quale è stato necessario aumentare il numero di frammenti generati al fine di ottenere una distruzione più realistica, adeguata alla sua maggiore complessità geometrica.

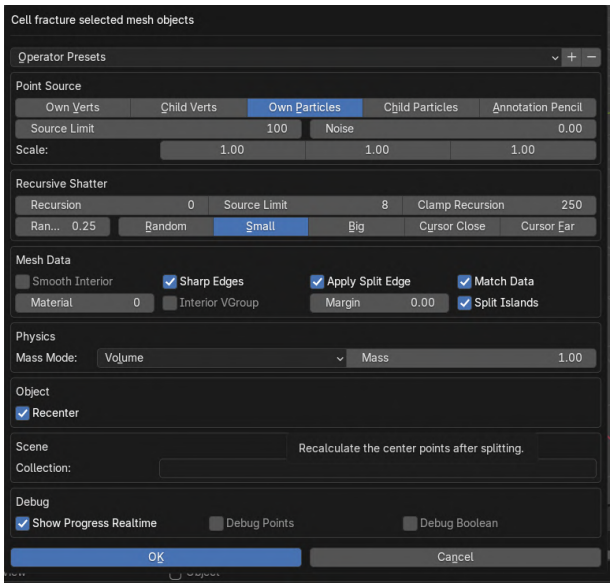

Figura 4.8: Applicazione dell'effetto di *Cell Fracture*

4. Eliminazione dell'oggetto padre integro dalla scena, mantenendo unicamente i frammenti generati dall'estensione.

5. Esportazione dell'oggetto frantumato in formato `.obj` insieme al file `.mtl` aggiornato, così da poter essere integrato nella cartella di progetto.

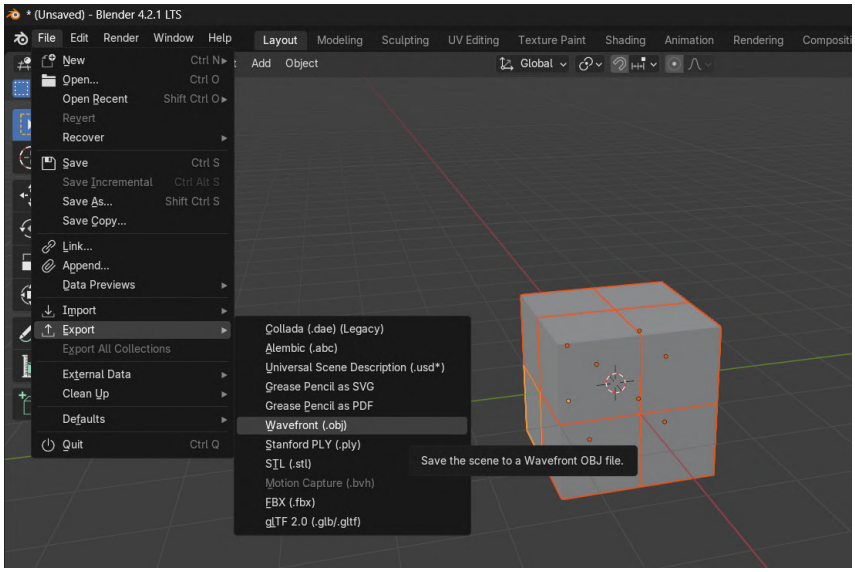

Figura 4.9: Esportazione dell'oggetto frantumato in formato `.obj`

Una volta caricati nel simulatore, i modelli sono stati gestiti come *models*, completi dei loro materiali. Per adattarli alla scena, è stato poi necessario applicare operazioni di trasformazione quali scaling, rotazioni e traslazioni (descritte in 4.5),rese possibili grazie alle funzioni della libreria `GLM` (OpenGL Mathematics).

In questo modo è stato possibile popolare la scena con oggetti coerenti, leggeri da gestire a livello computazionale e pronti per interagire con il sistema fisico di esplosione e crollo. La combinazione tra la semplicità dei modelli low-poly e la flessibilità della fratturazione tramite Blender ha reso il processo di preparazione efficiente e funzionale, consentendo di ottenere risultati visivi soddisfacenti senza appesantire l'implementazione tecnica del simulatore.

4.4 Personaggio e gestione della sua animazione.

Per quanto riguarda il personaggio è stato scelto il sito **Mixamo**[26], una piattaforma che consente di scaricare modelli tridimensionali completi di scheletro e animazioni già pronte all'uso. L'adozione di questo strumento ha permesso di integrare rapidamente un personaggio animato all'interno del simulatore, evitando la necessità di realizzare manualmente il rigging e la fase di animazione.

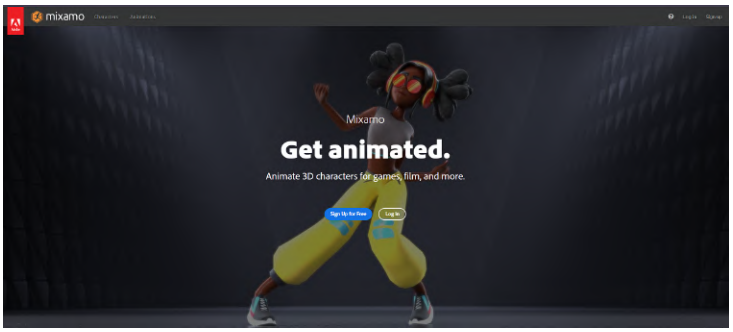

Figura 4.10: Il sito Mixamo, dove ho reperito il personaggio e le sue animazioni.

Il processo è iniziato con la selezione del modello: tra i numerosi personaggi disponibili, è stato scelto un mostro, ritenuto adatto a rappresentare un'entità in grado di colpire un edificio con sufficiente forza da innescarne l'esplosione in frammenti. Una volta individuato il modello, Mixamo lo ha impostato come personaggio principale, pronto per il test delle diverse animazioni.

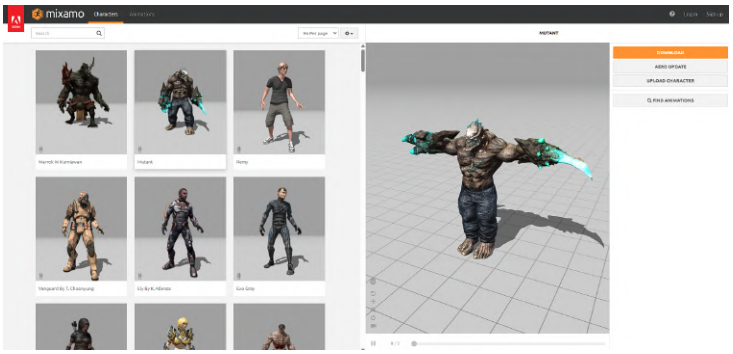

Figura 4.11: Il modello scelto per il progetto.

Il modello viene inizialmente presentato in *T-pose* (posa a T), configurazione standard utile per la successiva applicazione delle animazioni. Per renderlo più naturale all'interno della scena, sono state selezionate tre animazioni principali:

- **Breathing**, utilizzata per evitare che il personaggio rimanesse statico in T-pose, conferendo invece un movimento respiratorio di base;
- **Walking**, impostata in modalità *in place*, così da garantire una camminata continua sul posto, perfetta per simulare gli spostamenti controllati dall'utente;
- **Punching**, l'animazione principale, che mostra il personaggio mentre carica e sferra un pugno, azione impiegata per colpire gli oggetti e generare l'esplosione.

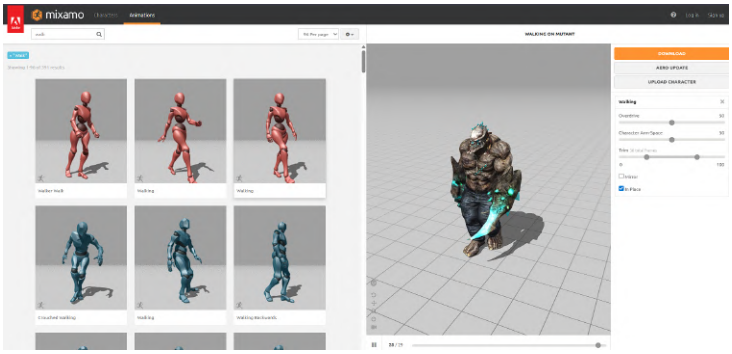

Figura 4.12: Animazione di camminata "In Place"

Dopo il download delle animazioni, è stato necessario un passaggio di correzione in **Blender**. Infatti, Mixamo esporta i modelli utilizzando l'asse verticale Z, mentre in OpenGL l'asse verticale di riferimento è Y. Questa differenza avrebbe causato un'errata interpretazione delle pose; perciò, è stato effettuato un semplice riallineamento degli assi prima dell'esportazione definitiva.

Completata la correzione, il personaggio animato è stato caricato correttamente all'interno del simulatore OpenGL, risultando pienamente integrato con la scena e pronto a interagire con gli elementi architettonici.[28]

Listing 4.1: Costruttore e setup delle animazioni

```
Character::Character(const std::string& modelPath): position(0.0f),
    rotation(0.0f), velocity(0.0f),currentState(CharacterState::IDLE),
    previousState(CharacterState::IDLE),m_punchCooldown(0.0f),
    m_isVisible(true), m_renderScale(2.0f) {

    // Caricamento del modello scheletrato (FBX)
    model = std::make_unique<AnimatedModel>("mostro.fbx");

    // Caricamento delle clip FBX (stesso rig)
    idleAnimation  = std::make_unique<Animation>("Idle_mostro.fbx",
          model.get());
    walkAnimation  = std::make_unique<Animation>("Walking_mostro.fbx"
       , model.get());
    punchAnimation = std::make_unique<Animation>("Punch_mostro.fbx",
         model.get());

    // Looping delle clip
    idleAnimation->SetLooping(true);
    walkAnimation->SetLooping(true);
    punchAnimation->SetLooping(false);

    // Animazione iniziale
    currentAnimation = idleAnimation.get();
}
```

Questo costruttore inizializza lo stato del personaggio (posizione, rotazione, visibilità, scala di rendering) e carica il modello scheletrato in formato `.fbx`, associando le tre clip principali (*Idle*, *Walking*, *Punch*). L'animazione corrente parte da *Idle*, così il personaggio è inizialmente in posa di riposo.

Come interviene Assimp

Il caricamento dei file `.fbx` avviene tramite la libreria `Assimp`, utilizzata dalle classi `AnimatedModel` e `Animation`. In particolare:

- **Clip di animazione**. Il costruttore di `Animation` usa `Assimp::Importer` per leggere la scena FBX, imposta la *Global Inverse Transform* della root e costruisce i canali (`aiNodeAnim`) di posizione/rotazione/scala per ciascun osso del rig; inoltre alloca l'array delle matrici finali per lo skinning.

Listing 4.2: Lettura di una clip FBX con Assimp (da ‘Animation.cpp‘)

```
Animation::Animation(const std::string& animationPath, Model* model)
    {
    Assimp::Importer importer;
    const aiScene* scene = importer.ReadFile(
        animationPath,
        aiProcess_Triangulate | aiProcess_GenSmoothNormals |
        aiProcess_FlipUVs    | aiProcess_CalcTangentSpace);
    if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene
       ->mRootNode) { /* ... */ }

    // Trasformazione inversa globale della root
    m_GlobalInverseTransform = glm::inverse(AssimpGLMHelpers::
       ConvertMatrixToGLMFormat(scene->mRootNode->mTransformation));

    ReadHierarchyData(m_RootNode, scene->mRootNode);
    ReadAnimationChannels(scene->mAnimations[0], model);

    m_FinalBoneMatrices.reserve(100);
    for (int i = 0; i < 100; ++i) m_FinalBoneMatrices.push_back(glm::
       mat4(1.0f));
}
```

Questo mostra chiaramente dove la clip viene analizzata e associata alle ossa del modello.

- **Modello e pesi di skinning**. `AnimatedModel` estrae per ogni mesh le ossa (`aiBone`), salva la *offset matrix* (bind pose) nella mappa delle ossa e scrive fino a quattro pesi per vertice negli array `BoneIDs`/`Weights`, normalizzandoli. L’override di `processMesh` garantisce che l’estrazione dei pesi avvenga durante il caricamento del mesh.

Listing 4.3: Estrazione di ossa e pesi (da ‘AnimatedModel.cpp‘)

```
std::unique_ptr<Mesh> AnimatedModel::processMesh(aiMesh* mesh, const
   aiScene* scene) {
    auto processedMesh = Model::processMesh(mesh, scene);
    auto& vertices = processedMesh->vertices;

    for (auto& v : vertices) SetVertexBoneDataToDefault(v);
    ExtractBoneWeightForVertices(vertices, mesh, scene); // legge
       aiBone e pesi
    processedMesh->updateVertexData();
    return processedMesh;
```

```
}

void AnimatedModel::ExtractBoneWeightForVertices(std::vector<Vertex>&
    vertices, aiMesh* mesh, const aiScene* scene) {
    for (unsigned int b = 0; b < mesh->mNumBones; ++b) {
        std::string boneName = mesh->mBones[b]->mName.C_Str();
        int boneID = GetOrCreateBoneID(boneName);

        // Offset matrix della bind-pose
        m_BoneInfoMap[boneName].offset =
            AssimpGLMHelpers::ConvertMatrixToGLMFormat(mesh->mBones[b
               ]->mOffsetMatrix);

        // Pesi per-vertice (max 4)
        for (unsigned int w = 0; w < mesh->mBones[b]->mNumWeights; ++
           w) {
            int vId = mesh->mBones[b]->mWeights[w].mVertexId;
            float weight = mesh->mBones[b]->mWeights[w].mWeight;
            SetVertexBoneData(vertices[vId], boneID, weight);
        }
    }
    // Normalizzazione dei pesi
    for (auto& v : vertices) {
        float sum = v.Weights.x + v.Weights.y + v.Weights.z + v.
           Weights.w;
        if (sum > 0.0f) v.Weights /= sum;
    }
}
```

Queste funzioni sono quelle effettivamente chiamate al caricamento del modello e dimostrano l'utilizzo di Assimp per popolare la mappa delle ossa e i pesi di skinning.

Durante l'aggiornamento dell'animazione, per ogni osso viene calcolata la trasformazione finale secondo

$$M_{\text{final}} = M_{\text{GlobalInverse}} \cdot M_{\text{NodeTransform}} \cdot M_{\text{Offset}},$$

e l'intero array `boneMatrices[0..N)` è inviato allo shader per lo skinning sulla GPU. L'utilizzo di Assimp è fondamentale per il parsing dei file FBX, l'estrazione di ossa e pesi (bind pose), la lettura dei canali di animazione con interpolazione (lineare per traslazione/scala, sferica per rotazioni), e consente di riutilizzare lo stesso rig per più clip e di sincronizzare il personaggio con la logica interattiva della simulazione.

Listing 4.4: Upload delle bone matrices allo shader

```
void Character::Render(Shader& shader) {
if (!model || !m_isVisible) return;

shader.use();
shader.setMat4("model", GetModelMatrix());
shader.setBool("hasTexture", true);

AnimatedModel* animModel = dynamic_cast<AnimatedModel*>(model.get());
if (currentAnimation && animModel) {
    std::vector<glm::mat4> boneTransforms = currentAnimation->
       GetFinalBoneMatrices();
    int maxBones = std::min(static_cast<int>(boneTransforms.size()),
       100);
    if (maxBones > 0) {
        for (int i = 0; i < maxBones; ++i) {
            std::string name = "boneMatrices[" + std::to_string(i) +
               "]";
            shader.setMat4(name, boneTransforms[i]);
        }
        shader.setBool("hasAnimation", true);
        shader.setInt("boneCount", maxBones);
    } else {
        shader.setBool("hasAnimation", false);
        shader.setInt("boneCount", 0);
    }
} else {
    shader.setBool("hasAnimation", false);
    shader.setInt("boneCount", 0);
}
model->render(shader);
}
```

Questa funzione effettua il rendering del personaggio: imposta la *model matrix* e, se è attiva un'animazione su un modello scheletrato, invia allo shader l'array di matrici ossee `boneMatrices[]` (fino a 100). Imposta anche le uniform `hasAnimation` e `boneCount` per abilitare lo *skinning* sul vertex shader. Se non ci sono ossa da animare, disattiva lo skinning.
Alla fine invoca il `render` del modello.

Listing 4.5: Movimento, stati e gestione del pugno

```
void Character::Move(const glm::vec3& direction, float speed, float
    deltaTime) {
     if (std::isnan(direction.x) || std::isnan(direction.y) || std::
        isnan(direction.z)) return;
     if (std::isinf(direction.x) || std::isinf(direction.y) || std::
        isinf(direction.z)) return;
     if (currentState == CharacterState::PUNCHING) return;
     float moveLength = glm::length(direction);
     if (moveLength > 0.1f) {
         glm::vec3 normalizedDir = glm::normalize(direction);
         float moveDistance = speed * deltaTime;
         position += normalizedDir * moveDistance;
         if (currentState != CharacterState::WALKING) {
             StartWalking();
         }
         float targetRotation = atan2(normalizedDir.x, normalizedDir.z
            );
         if (!std::isnan(targetRotation) && !std::isinf(targetRotation
            )) {
             float rotDiff = targetRotation - rotation;
             while (rotDiff > glm::pi<float>()) rotDiff -= 2 * glm::pi
                <float>();
             while (rotDiff < -glm::pi<float>()) rotDiff += 2 * glm::
                pi<float>();
             float rotSpeed = 10.0f;
             if (abs(rotDiff) > 0.1f) {
                 rotation += rotDiff * rotSpeed * deltaTime;
             }
             else {
                 rotation = targetRotation;
             }
         }
     }
     else {
         if (currentState == CharacterState::WALKING) {
             StopWalking();
         }
     }
}

```

```
34 void Character::StartPunching() {
35     if (currentState != CharacterState::PUNCHING) {
36         previousState = currentState;
37         currentState = CharacterState::PUNCHING;
38         SwitchAnimation(punchAnimation.get());
39         punchAnimation->Reset();
40         m_punchCooldown = 3.0f; // cooldown
41     }
42 }
43 bool Character::HasFinishedPunching() const {
44     return currentState == CharacterState::PUNCHING && punchAnimation
       ->IsFinished();
45 }
46
47 void Character::ResetToIdle() {
48     currentState = CharacterState::IDLE;
49     SwitchAnimation(idleAnimation.get());
50     m_punchCooldown = 0.0f;
51     m_isVisible = true;
52     m_renderScale = 2.0f;
53 }
```

Il metodo `Move` nella classe Character.cpp è responsabile della gestione dello spostamento e dell'orientamento del personaggio in base all'input ricevuto: All'inizio vengono eseguiti controlli di sicurezza per evitare che valori non validi (come `NaN` o infiniti) possano compromettere la logica. Se il personaggio si trova nello stato di `PUNCHING`, lo spostamento viene disabilitato per impedire movimenti durante l'animazione di attacco.
Se la direzione fornita ha una lunghezza significativa, il vettore viene normalizzato e utilizzato per calcolare il nuovo spostamento in funzione della velocità e del `deltaTime`. La posizione viene aggiornata di conseguenza e, se lo stato corrente non è già `WALKING`, viene invocato `StartWalking()` per avviare l'animazione di camminata.
La rotazione del personaggio viene calcolata a partire dalla direzione di marcia tramite la funzione `atan2`. Per evitare rotazioni improvvise, la differenza rispetto alla rotazione attuale viene normalizzata nell'intervallo $[-\pi, \pi]$ e applicata gradualmente con un fattore di interpolazione (`rotSpeed`), ottenendo un effetto di transizione fluida verso la nuova direzione. Se la differenza angolare è molto piccola, la rotazione viene aggiornata direttamente al valore target.
Al contrario, se la lunghezza del vettore direzionale è trascurabile, il personaggio viene considerato fermo: in tal caso, se lo stato era `WALKING`, viene richiamato `StopWalking()` per riportarlo allo stato di inattività (`IDLE`) e interrompere l'animazione di camminata.

Listing 4.6: Capsula della mano destra nel world-space

```
Capsule Character::GetRightFistCapsuleWorld() const {
    Capsule cap{ glm::vec3(0), glm::vec3(0), 0.18f * m_renderScale };
    auto* anim = currentAnimation;
    auto* animModel = dynamic_cast<AnimatedModel*>(model.get());
    if (!anim || !animModel) return cap;
    auto finals = currentAnimation->GetFinalBoneMatrices();
    const auto& map = animModel->GetBoneInfoMap();
    auto find = [&](const char* n)->const auto* {
        auto it = map.find(n); return (it == map.end() ? nullptr : &
           it->second);
    };
    auto* h = find("mixamorig:RightHand"), * f = find("mixamorig:
       RightForeArm");
    if (!h || !f) return cap;

    glm::mat4 G  = glm::inverse(anim->GetGlobalInverseTransform());
    glm::mat4 Mw = GetModelMatrix();

    glm::mat4 Hw = Mw * (G * finals[h->id] * glm::inverse(h->offset))
       ;
    glm::mat4 Fw = Mw * (G * finals[f->id] * glm::inverse(f->offset))
       ;

    cap.a = glm::vec3(Fw * glm::vec4(0, 0, 0, 1));
    cap.b = glm::vec3(Hw * glm::vec4(0, 0, 0, 1));
    return cap;
}
```

Qui si costruisce una **capsula** attorno al polso del personaggio che segue l'animazione del pugno: si recuperano le matrici finali di `RightForeArm` e `RightHand`, le si portano in world-space usando la *model matrix* e la *global inverse transform*, e si definiscono le estremità della capsula nei punti ossei trasformati. Il raggio scala con `m_renderScale`. In questo modo la hitbox creata sulla mano del personaggio è usata per il test di impatto con gli oggetti (esplosione), rendendo più realistica l'animazione.

Listing 4.7: Creazione del personaggio

```
std::vectorstd::string
characterFiles = {
"mostro.fbx", "Walking_mostro.fbx", "Idle_mostro.fbx", "Punch_mostro.
```

```
    fbx"
};

for (const std::string& fbxFile : characterFiles) {
    if (FileUtils::fileExists(fbxFile)) {
        m_character = std::make_unique<Character>(fbxFile);
        m_character->SetPosition(glm::vec3(3.0f, -0.8f, 3.0f));
        break;
    }
}
```

Durante l'inizializzazione dell'applicazione si crea il `Character` selezionando il primo file disponibile tra quelli FBX previsti e lo si posiziona nella scena con `SetPosition`.
In questo modo il personaggio è pronto per essere aggiornato e renderizzato nel *game loop*.

Listing 4.8: Input di movimento, update e trigger esplosione

```
if (m_character) {
glm::vec3 moveDirection(0.0f);
bool isMoving = false;

if (m_window->isKeyPressed(GLFW_KEY_UP))    { moveDirection.z -= 1.0f
    ; isMoving = true; }
if (m_window->isKeyPressed(GLFW_KEY_DOWN))  { moveDirection.z += 1.0f
    ; isMoving = true; }
if (m_window->isKeyPressed(GLFW_KEY_LEFT))  { moveDirection.x -= 1.0f
    ; isMoving = true; }
if (m_window->isKeyPressed(GLFW_KEY_RIGHT)) { moveDirection.x += 1.0f
    ; isMoving = true; }

if (isMoving) {
    float length = glm::length(moveDirection);
    if (length > 0.0001f) {
        moveDirection = glm::normalize(moveDirection);
        m_character->Move(moveDirection, 2.5f, deltaTime);
        glm::vec3 p = m_character->GetPosition();
        p.y = -0.85f;
        p.x = glm::clamp(p.x, -20.0f, 20.0f);
        p.z = glm::clamp(p.z, -20.0f, 20.0f);
        m_character->SetPosition(p);
    }
} else {
```

```
22      m_character->Move(glm::vec3(0.0f), 0.0f, deltaTime);
23  }
24
25  m_character->Update(deltaTime);
26
27  bool nearBuilding = m_character->IsNearBuilding(glm::vec3(0.0f, 0.0f,
        0.0f), 3.0f);
28  if (m_character->GetState() == CharacterState::WALKING &&
29      nearBuilding && m_stateManager && m_stateManager->isIntact()) {
30      m_character->InteractWithBuilding(); // avvia il punch
31  }
32
33  if (m_character->GetState() == CharacterState::PUNCHING) {
34      Capsule fist = m_character->GetRightFistCapsuleWorld();
35      AABB buildingAABB = /* AABB world del palazzo */;
36      glm::vec3 hitPoint;
37      if (CapsuleVsAABB(fist, buildingAABB, hitPoint)) {
38          m_stateManager->triggerExplosion(); // esplosione immediata
39      }
40  }
41
42  if (m_character->HasFinishedPunching()) {
43      m_character->ResetToIdle();
44  }
45  }
```

Nel ciclo di aggiornamento il programma cattura lo stato delle frecce direzionali della tastiera e costruisce un vettore di movimento che rappresenta la direzione desiderata. Se almeno una freccia è premuta, questo vettore viene normalizzato in modo da garantire una velocità uniforme in qualunque direzione, evitando che spostarsi in diagonale risulti più rapido che muoversi in linea retta. Successivamente viene invocato il metodo `Move`, che aggiorna la posizione del personaggio in funzione della direzione normalizzata, della velocità prefissata e del `deltaTime`, assicurando uno spostamento regolare e indipendente dal framerate. Dopo lo spostamento, la posizione viene vincolata all'interno dei limiti della scena tramite un'operazione di *clamping* sugli assi orizzontali, mentre l'altezza viene mantenuta costante sul piano di base, così da garantire che il personaggio rimanga correttamente ancorato al terreno virtuale. Nel caso in cui non vi siano input da tastiera, `Move` viene comunque richiamato con un vettore nullo, provocando la transizione automatica allo stato di inattività.

Una volta aggiornati movimento e posizione, il metodo `Update` si occupa di far avanzare l'animazione corrente e di gestire i cambiamenti di stato, come eventuali cooldown o il pas-

saggio tra camminata e idle. A questo punto viene verificata la distanza dal palazzo: se il personaggio si trova abbastanza vicino ed è nello stato di camminata, viene richiamata la funzione `InteractWithBuilding()`, che avvia l'animazione di pugno e imposta lo stato a `PUNCHING`. Durante questa fase viene calcolata la capsula corrispondente al braccio destro in world space, comprendente mano e avambraccio, e viene eseguito un test di collisione con la bounding box dell'edificio. Nel momento in cui la capsula del pugno interseca l'AABB, viene invocata la funzione `triggerExplosion()`, che scatena la simulazione fisica della frantumazione del palazzo. Al termine dell'animazione di attacco, grazie al controllo fornito da `HasFinishedPunching()`, il personaggio torna automaticamente nello stato di idle, pronto a ricevere nuovi input dall'utente e a ripetere l'interazione.

4.5 Creazione della scena: skybox e posizionamento degli elementi

La costruzione della scena rappresenta il momento in cui vengono definiti il contesto grafico e gli elementi interattivi che compongono l'ambiente di simulazione.
In questa fase si creano la finestra di rendering, lo skybox che funge da sfondo, gli oggetti statici della scena (come il palazzo e altri modelli in formato `.obj`) e il personaggio animato che interagisce con essi.

Caricamento della scena (Window e Application)

La prima fase riguarda la creazione della finestra e l'inizializzazione del contesto OpenGL: questa operazione è gestita dalla classe `Window`, che utilizza la libreria GLFW per la creazione della finestra e GLAD per il caricamento dinamico delle funzioni OpenGL.
All'interno del costruttore vengono impostate le versioni del contesto, la possibilità di ridimensionamento e il multisampling per migliorare la qualità grafica.

Listing 4.9: Inizializzazione della finestra con GLFW e GLAD

```
bool Window::initialize() {
    if (!glfwInit()) return false;
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_TRUE);
    glfwWindowHint(GLFW_SAMPLES, 4);

    m_window = glfwCreateWindow(m_width, m_height, m_title.c_str(),
        nullptr, nullptr);
    if (!m_window) { glfwTerminate(); return false; }

    glfwMakeContextCurrent(m_window);
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) return
        false;

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_MULTISAMPLE);
    glViewport(0, 0, m_width, m_height);
    glfwSwapInterval(1); // VSync
    return true;
}
```

Questo metodo si occupa di configurare l'ambiente grafico: vengono inizializzate le librerie, la finestra viene creata secondo parametri precisi con il contesto OpenGL 4.3, abilitate le opzioni di depth test e multisampling, e attivato il VSync per limitare gli FPS alla frequenza di aggiornamento del monitor. In questo modo l'applicazione dispone di una base solida per ospitare il rendering della scena.

Skybox

Per contestualizzare visivamente la scena è stato utilizzato uno skybox, ovvero un cubo che circonda l'intero ambiente e al quale vengono applicate sei texture (una per faccia) che simulano il cielo e l'orizzonte. Lo skybox utilizzato per questo progetto, rappresentante un paesaggio urbano coerente all'idea di sviluppo, è stato reperito dal sito **OpenGameArt.org** [25] : Dopo il download del file .zip, il contenuto viene estratto e collocato in una nuova sottocartella chiamata "skybox" all'interno della directory di progetto; in questa cartella sono presenti i sei file .jpeg, ciascuno corrispondente a un'angolazione specifica dello skybox. La classe `Skybox` gestisce sia la costruzione della geometria sia il caricamento delle immagini che costituiscono il cubemap.

Listing 4.10: Inizializzazione dello skybox e caricamento delle texture

```
bool Skybox::initialize(const std::vector<std::string>& faces) {
    setupSkyboxGeometry();
    createSkyboxShader();
    m_cubemapTexture = loadCubemap(faces);
    return m_cubemapTexture != 0;
}

void Skybox::render(const glm::mat4& view, const glm::mat4&
   projection) {
    glDepthFunc(GL_LEQUAL);
    m_skyboxShader->use();
    glm::mat4 viewNoTranslation = glm::mat4(glm::mat3(view));
    m_skyboxShader->setMat4("view", viewNoTranslation);
    m_skyboxShader->setMat4("projection", projection);
    glBindVertexArray(m_VAO);
    glBindTexture(GL_TEXTURE_CUBE_MAP, m_cubemapTexture);
    glDrawArrays(GL_TRIANGLES, 0, 36);
    glBindVertexArray(0);
    glDepthFunc(GL_LESS);
}
```

Il metodo `initialize` della classe `Skybox` ha il compito di predisporre tutti gli elementi necessari per la creazione dello sfondo tridimensionale: in primo luogo viene invocata la funzione `setupSkyboxGeometry()`, che definisce la geometria del cubo attraverso un array di vertici: le sei facce dello skybox sono infatti costruite come due triangoli ciascuna, per un totale di 36 vertici. Questo approccio consente di rappresentare un cubo unitario senza coordinate texture tradizionali, poiché le immagini verranno applicate direttamente come cubemap. Successivamente viene creato e compilato lo shader dedicato allo skybox, composto da un *vertex shader* e da un *fragment shader*, il cui compito è quello di gestire correttamente la proiezione del cubo e la visualizzazione delle texture.

Il passo successivo riguarda il caricamento delle sei immagini che compongono la cubemap: la funzione `loadCubemap` si occupa di associare ogni immagine ad una faccia del cubo (positive/negative X, Y e Z). Le immagini vengono lette con la libreria `stb_image` [35][35] e caricate in OpenGL tramite chiamate a `glTexImage2D`, specificando il formato corretto in base al numero di canali (RGB o RGBA). Per migliorare la resa grafica vengono poi impostati i parametri di filtraggio (`GL_LINEAR`) e di wrapping (`GL_CLAMP_TO_EDGE`), così da evitare artefatti visivi sui bordi delle facce. Infine, viene generata una mipmap del cubemap per garantire una resa ottimale anche quando la scena è visualizzata a distanze diverse.

Per la fase di rendering invece, la funzione `render` riceve in ingresso le matrici di vista e proiezione della camera. Prima del disegno viene modificata la funzione di profondità (nel codice presente è `glDepthFunc(GL_LEQUAL)`) per garantire che lo skybox sia sempre disegnato correttamente anche quando gli oggetti della scena si trovano nello stesso piano di profondità. La matrice di vista viene poi trasformata eliminando la componente di traslazione: questo accorgimento è fondamentale, perché consente allo skybox di ruotare insieme alla telecamera ma di non seguire mai i suoi spostamenti.

In questo modo il cubo rimane sempre centrato sull'osservatore e appare quindi infinito, creando l'illusione di un ambiente che si estende all'infinito attorno alla scena. Dopo aver impostato le variabili uniform necessarie nello shader, viene eseguita la chiamata `glDrawArrays` che disegna i 36 vertici dello skybox, applicando la texture cubemap caricata in precedenza.
Infine, la funzione di profondità viene ripristinata al valore predefinito (`GL_LESS`) per consentire il rendering corretto degli oggetti successivi nella scena.

Posizionamento di oggetti e personaggio

Gli oggetti che compongono la scena (palazzo, albero, cabina telefonica e bus) vengono caricati da file `.obj` tramite la classe `Model`. Questa utilizza Assimp per importare le geometrie e associa i materiali definiti nei file `.mtl`. Una volta importati, i modelli vengono scalati, traslati e ruotati per essere posizionati correttamente rispetto al piano di gioco.

Listing 4.11: Caricamento e posizionamento di un modello

```
glm::mat4 buildingModelMatrix = glm::mat4(1.0f);
buildingModelMatrix = glm::translate(buildingModelMatrix, glm::vec3
    (0.0f, -0.9f, 0.0f));
buildingModelMatrix = glm::rotate(buildingModelMatrix, glm::radians
    (180.0f), glm::vec3(0.0f, 1.0f, 0.0f));
buildingModelMatrix = glm::scale(buildingModelMatrix, glm::vec3(1.2f,
     1.6f, 1.2f));
```

In questo esempio il modello del palazzo, precedentemente caricato da file, viene posizionato all'interno della scena e scalato secondo valori specifici per renderlo coerente con il resto degli oggetti. Procedure analoghe vengono applicate agli altri oggetti, garantendo coerenza nelle proporzioni e nella disposizione spaziale.
Il personaggio animato, importato da Mixamo, viene invece gestito dalla classe `Character`, che integra modello e animazioni. Il posizionamento iniziale avviene al centro della scena, da cui il personaggio può muoversi in direzione degli oggetti con i comandi da tastiera, restando sempre all'interno del piano di gioco.

Listing 4.12: Inizializzazione del personaggio nella scena

```
std::cout << "Loading character..." << std::endl;
std::vector<std::string> characterFiles = { "mostro.fbx", "
   Walking_mostro.fbx", "Idle_mostro.fbx", "Punch_mostro.fbx" };

for (const std::string& fbxFile : characterFiles) {
    if (FileUtils::fileExists(fbxFile)) {
        try {
            m_character = std::make_unique<Character>(fbxFile);
            m_character->SetPosition(glm::vec3(3.0f, -0.8f, 3.0f));
            std::cout << "Character loaded successfully: " << fbxFile
                << std::endl;
            break;
        }
        catch (const std::exception& e) {
            std::cerr << "Failed to load " << fbxFile << ": " << e.
               what() << std::endl;
```

```
            }
        }
}
```

In questo frammento di codice viene gestito il caricamento del personaggio animato: viene definito un vettore contenente i percorsi ai file `.fbx` necessari: il modello principale (`mostro.fbx`) e le animazioni fondamentali per il suo funzionamento, cioè `Walking`, `Idle` e `Punch`. Questi file rappresentano rispettivamente la mesh del personaggio e le tre animazioni che ne descrivono i comportamenti base.

Il ciclo `for` scorre la lista dei file e, per ciascuno, verifica innanzitutto la presenza del file sul disco tramite la funzione `FileUtils::fileExists`. In caso positivo, prova a creare una nuova istanza di `Character`, passando come parametro il file corrente. L'uso di un blocco `try-catch` consente di gestire eventuali eccezioni generate durante la fase di caricamento, ad esempio dovute a file corrotti o incompatibili: se il caricamento ha successo, viene stampato un messaggio di conferma con il nome del file e il ciclo viene interrotto; al contrario, se si verifica un errore, l'eccezione viene catturata e viene stampato un messaggio che specifica quale file non è stato caricato e la causa del fallimento.

Una volta creato con successo, il personaggio viene posizionato nella scena con la chiamata `SetPosition`, che ne definisce le coordinate iniziali nella scena (x=3.0, y=-0.8, z=3.0). Questo posizionamento leggermente decentrato rispetto al palazzo e agli oggetti premette di predisporre il personaggio all'interazione: muovendosi nella scena, potrà avvicinarsi agli oggetti e, grazie alle animazioni collegate, eseguire i movimenti di camminata e il colpo di pugno che scatena l'esplosione.

4.6 Funzionalità varie: comandi da tastiera e gestione della camera

L'interattività del simulatore si basa sulla possibilità per l'utente di muoversi liberamente nella scena e di attivare eventi particolari, come le esplosioni. Questa funzionalità è stata ottenuta grazie all'integrazione tra la classe `InputHandler`, che gestisce la pressione dei tasti e dei relativi callback, e la classe `Camera`, che si occupa del movimento e dell'orientamento della telecamera. In questo modo l'utente può navigare nello spazio tridimensionale e interagire con gli oggetti in maniera naturale e reattiva.

Gestione degli input da tastiera

La classe `InputHandler` inizializza i binding di default e associa ogni tasto ad un'azione. Alcuni tasti gestiscono movimenti continui (come W, A, S, D), altri invece attivano eventi singoli (come la stampa di debug o il reset della scena).

Listing 4.13: Inizializzazione dei key bindings

```
void InputHandler::setupDefaultKeyBindings() {
    // Movement keys (continuous)
    addKeyBinding(GLFW_KEY_W, InputAction::MOVE_FORWARD, false);
    addKeyBinding(GLFW_KEY_S, InputAction::MOVE_BACKWARD, false);
    addKeyBinding(GLFW_KEY_A, InputAction::MOVE_LEFT, false);
    addKeyBinding(GLFW_KEY_D, InputAction::MOVE_RIGHT, false);
    addKeyBinding(GLFW_KEY_SPACE, InputAction::MOVE_UP, false);
    addKeyBinding(GLFW_KEY_LEFT_SHIFT, InputAction::MOVE_DOWN, false)
        ;

    // Toggle keys (single press)
    addKeyBinding(GLFW_KEY_C, InputAction::TOGGLE_CAMERA, true);
    addKeyBinding(GLFW_KEY_F, InputAction::TOGGLE_FULLSCREEN, true);

    // Action keys (single press)
    addKeyBinding(GLFW_KEY_R, InputAction::RESET_EXPLOSION, true);
    addKeyBinding(GLFW_KEY_ESCAPE, InputAction::EXIT_APPLICATION,
        true);

    // Debug keys
    addKeyBinding(GLFW_KEY_I, InputAction::DEBUG_INFO, true);
}
```

In questo frammento vengono definiti i tasti di movimento e di interazione: i primi (W, A, S, D, `Space`, `Shift`) sono gestiti come input continui, ovvero rimangono attivi finché il tasto è premuto (per avere un movimento fluido e non a scatti).
Al contrario, i tasti per le azioni (`R`, `ESC`, `I`) sono interpretati come input singoli e scatenano immediatamente l'evento collegato.

La gestione degli eventi viene completata tramite callback, che collegano ogni azione a una funzione.

Listing 4.14: Esempio di callback per i tasti azione

```
setActionCallback(InputAction::TOGGLE_FULLSCREEN, [this]() {
    if (m_window) {
        std::cout << "Toggling fullscreen..." << std::endl;
        m_window->toggleFullscreen();
    }
});

setActionCallback(InputAction::TOGGLE_CAMERA, [this]() {
    toggleCameraControl();
});
```

Il primo callback collega il tasto `F` alla funzione `toogleFullscreen()` passando così dalla modalità finestra alla modalità schermo intero, mentre il secondo esempio mostra invece come il tasto `C` abiliti o disabiliti il controllo della telecamera, alternando tra modalità libera e modalità bloccata.

Funzionamento della Camera

La classe `Camera` gestisce la posizione e l'orientamento nello spazio tridimensionale. Essa calcola due matrici fondamentali: la **matrice di vista**, che definisce l'orientamento e la posizione della telecamera, e la **matrice di proiezione**, che determina la distorsione prospettica.

Listing 4.15: Calcolo delle matrici di vista e proiezione

```
glm::mat4 Camera::getViewMatrix() const {
    return glm::lookAt(m_position, m_position + m_front, m_up);
}

glm::mat4 Camera::getProjectionMatrix(float aspectRatio, float
   nearPlane, float farPlane) const {
    return glm::perspective(glm::radians(m_zoom), aspectRatio,
       nearPlane, farPlane);}
```

La matrice di vista viene costruita con la funzione `glm::lookAt`, che utilizza la posizione della camera, la direzione di osservazione (`m_front`) e il vettore `up`. La matrice di proiezione sfrutta invece la funzione `glm::perspective`, che applica una proiezione prospettica parametrizzata dal campo visivo (zoom), dal rapporto d'aspetto della finestra e dai piani di clipping vicino/lontano.

Per rendere la telecamera reattiva, sono implementati metodi che gestiscono i movimenti tramite tastiera e mouse.

Listing 4.16: Movimento della telecamera da tastiera

```
void Camera::processKeyboard(Camera_Movement direction, float
    deltaTime) {
    float velocity = m_movementSpeed * deltaTime;

    switch (direction) {
    case FORWARD:  m_position += m_front * velocity; break;
    case BACKWARD: m_position -= m_front * velocity; break;
    case LEFT:     m_position -= m_right * velocity; break;
    case RIGHT:    m_position += m_right * velocity; break;
    case UP:       m_position += m_worldUp * velocity; break;
    case DOWN:     m_position -= m_worldUp * velocity; break;
    }
}
```

Questo metodo sposta la telecamera nelle direzioni fondamentali (avanti, indietro, destra, sinistra, alto, basso) in base ai tasti premuti (W, S, D, A, SPACE, SHIFT) e al tempo trascorso, garantendo un movimento fluido indipendente dal framerate.

I movimenti del mouse invece influenzano gli angoli di rotazione (yaw e pitch), permettendo di orientare la visuale.

Listing 4.17: Gestione del movimento del mouse

```
void Camera::processMouseMovement(float xpos, float ypos, bool
    constrainPitch) {
    if (m_firstMouse) {
        m_lastX = xpos; m_lastY = ypos;
        m_firstMouse = false;
    }

    float xoffset = xpos - m_lastX;
    float yoffset = m_lastY - ypos; // y invertito
```

```
    m_lastX = xpos; m_lastY = ypos;

    xoffset *= m_mouseSensitivity;
    yoffset *= m_mouseSensitivity;

    m_yaw += xoffset;
    m_pitch += yoffset;

    if (constrainPitch) {
        m_pitch = clamp(m_pitch, -89.0f, 89.0f);
    }

    updateCameraVectors();
}
```

In questo metodo viene gestito l'aggiornamento dell'orientamento della telecamera in base ai movimenti del mouse: alla prima chiamata viene inizializzato lo stato del mouse, salvando le coordinate iniziali per evitare scatti improvvisi della visuale. Successivamente si calcolano gli offset orizzontali e verticali, ossia la differenza tra la posizione attuale e quella precedente del cursore, tenendo conto che in ambiente grafico l'asse verticale ha un orientamento invertito.

Gli offset vengono scalati con un coefficiente di sensibilità, in modo da regolare la velocità di rotazione della telecamera. A questo punto i valori vengono sommati agli angoli yaw e pitch, che rappresentano rispettivamente la rotazione orizzontale e quella verticale. Per prevenire ribaltamenti indesiderati della visuale, il pitch viene limitato all'intervallo $[-89°, 89°]$, impedendo alla telecamera di ruotare completamente verso l'alto o il basso. Infine, viene invocata la funzione `updateCameraVectors()`, che ricalcola i vettori direzionali della telecamera (front, right, up) sulla base dei nuovi angoli, assicurando che la scena venga renderizzata correttamente secondo il nuovo orientamento.

Abilitazione e disabilitazione della telecamera

Il controllo della telecamera può essere attivato o disattivato tramite il tasto `C`, che attiva o disattiva il controllo della telecamera, nascondendo il cursore e abilitando il movimento con il mouse quando attivo. Questo comportamento è implementato nella funzione seguente:

Listing 4.18: Attivazione e disattivazione del controllo camera

```
void InputHandler::setCameraEnabled(bool enabled) {
    m_cameraEnabled = enabled;

    if (m_window) {
```

```
        if (enabled) {
            m_window->setCursorMode(GLFW_CURSOR_DISABLED);
            if (m_camera) {
                m_camera->resetMouseState();
            }
            std::cout << "Camera control enabled" << std::endl;
        }
        else {
            m_window->setCursorMode(GLFW_CURSOR_NORMAL);
            std::cout << "Camera control disabled" << std::endl;
        }
    }
}
```

Questa funzione permette di abilitare o disabilitare il controllo della telecamera tramite mouse: quando viene attivata, il cursore del mouse viene nascosto e bloccato al centro della finestra: in questo stato i movimenti del mouse vengono catturati interamente per aggiornare l'orientamento della telecamera, garantendo all'utente la possibilità di ruotare liberamente lo sguardo nello spazio 3D. Inoltre, lo stato interno della camera viene azzerato per evitare salti improvvisi all'attivazione. Quando invece il controllo viene disabilitato, il cursore torna visibile e può muoversi liberamente all'interno della finestra senza modificare la visuale. Questo comportamento è utile, ad esempio, quando si desidera interagire con l'interfaccia o fermare temporaneamente la simulazione.

Tasto	**Azione**
W	Muovi la telecamera in avanti
S	Muovi la telecamera indietro
A	Muovi la telecamera a sinistra
D	Muovi la telecamera a destra
SPACE	Muovi la telecamera verso l'alto
SHIFT	Muovi la telecamera verso il basso
R	Resetta la scena (palazzo integro)
I	Mostra informazioni di debug
ESC	Esci dall'applicazione
C	Abilita/disabilita il controllo della telecamera (mouse)
F	Attiva/disattiva la modalità fullscreen
P	Commutazione tra fisica su CPU e GPU

Tabella 4.1: Mappatura dei comandi da tastiera nel simulatore

Grazie all'integrazione tra la gestione degli input e la telecamera, il simulatore permette un'interazione fluida e intuitiva, consentendo all'utente di muoversi liberamente nello spazio 3D.

4.7 Gestione delle collisioni e Bounding Box

In questa sezione si descrivono le procedure per la getione delle collisioni e le strutture geometriche utilizzate nel simulatore. Per gli oggetti statici viene adottata una **Axis-Aligned Bounding Box** (AABB) definita da centro ed estensioni lungo i semiassi, mentre per il pugno del personaggio viene utilizzata una **capsula** (segmento con raggio) allineata all'animazione. La libreria include utility per trasformare AABB locali in world space tenendo conto di rotazioni e scale non uniformi, e per visualizzare le bounding box in wireframe per il debug e test discreti di collisione *capsule–AABB* e test continui "*swept*" per gestire colpi veloci (time-of-impact). Infine, sono presenti anche utility per una risoluzione minima di penetrazione tra AABB (*push-out*).

L'integrazione avviene nella classe `Application` con il calcolo degli AABB locali dai modelli, il toggle runtime della visualizzazione e l'uso dei test durante l'interazione pugno → esplosione.

AABB: struttura dati e trasformazione in spazio mondo

Listing 4.19: Struttura AABB e trasformazione locale→mondo

```
struct AABB {                   // center + half-extents
    glm::vec3 c;                // center
    glm::vec3 e;                // half-extents
};

// Converte un AABB locale in world-space (gestisce rotazione e scala
    non-uniforme)
inline AABB ToWorldAABB(const AABB& local, const glm::mat4& M) {
    glm::vec3 C = glm::vec3(M * glm::vec4(local.c, 1.0f));
    glm::mat3 R = glm::mat3(M);
    glm::mat3 AR = glm::mat3(glm::abs(R[0]), glm::abs(R[1]), glm::abs
        (R[2]));
    glm::vec3 E = AR * local.e;
    return { C, E };
}
```

L'AABB è rappresentata dal centro **c** e dalle semi-estensioni **e**. La trasformazione in world-space calcola il nuovo centro come $C = M\,[c, 1]^T$ mentre le nuove estensioni sono ottenute come $E = |R|\,e$, dove R è la parte 3×3 derivata dalla matrice di modello M e $|R|$ è la matrice con i valori assoluti delle colonne. Questo metodo, una tecnica standard, permette di incorporare rotazioni e scale non uniformi mantenendo l'AABB allineata agli assi, ottenendo così un

bounding box “abbastanza stretto” attorno al modello trasformato.

Debug: Disegno Wireframe delle AABB

Listing 4.20: Model matrix per AABB e disegno wireframe

```
inline glm::mat4 AABBModelMatrix(const AABB& w) {
    return glm::translate(glm::mat4(1.0f), w.c) * glm::scale(glm::
       mat4(1.0f), w.e * 2.0f);
}

// Disegna un AABB come cubo wireframe, usando renderCube
inline void DrawAABB_Wire(
    const AABB& wbox, const glm::vec3& color,
    const glm::mat4& view, const glm::mat4& proj,
    const std::function<void(const glm::mat4&, const glm::mat4&,
       const glm::mat4&, const glm::vec3&)>& renderCube)
{
    glDisable(GL_CULL_FACE);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    renderCube(AABBModelMatrix(wbox), view, proj, color);
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    glEnable(GL_CULL_FACE);
}
```

Per il debug, ogni AABB in world-space viene rappresentata come cubo unitario scalato alle dimensioni $2e$ (dove e è la semi-estensione) e traslato in c. La funzione `DrawAABB_Wire` imposta la modalità wireframe (poligoni contornati da linee) , e affida la draw call a un callback, renderCube, che utilizza le matrici M, V, P e un colore a scelta . Ciò consente di ispezionare rapidamente ingombri, offset e correttezza delle trasformazioni.

Test Capsula–AABB (pugno del personaggio sugli oggetti)

Listing 4.21: Capsula, punto più vicino su AABB e test discreto

```
struct Capsule { glm::vec3 a, b; float r; }; // segmento AB + raggio

inline glm::vec3 ClosestPointOnAABB(const glm::vec3& p, const AABB& b
   ){
    glm::vec3 minB = b.c - b.e, maxB = b.c + b.e;
    return glm::clamp(p, minB, maxB);
}
```

```
inline bool CapsuleVsAABB(const Capsule& cap, const AABB& box, glm::
    vec3& contact){
     const int N = 8; float best = 1e9f;
     glm::vec3 bestP(0);
     for(int i=0;i<=N;i++){
         float t = i*(1.0f/N);
         glm::vec3 p = glm::mix(cap.a, cap.b, t);
         glm::vec3 q = ClosestPointOnAABB(p, box);
         float d2 = glm::dot(p - q, p - q);
         if(d2<best){ best=d2; bestP=q; }
     }
     contact = bestP;
     return best <= cap.r*cap.r;
}
```

Il pugno è modellato come una *capsula*, cioè un segmento tra mano e avambraccio con un raggio, allineato all'animazione. Il test discreto campiona $N + 1$ punti lungo il segmento $\overline{ab}$ e, per ciascuno di essi, calcola il punto più vicino sull'AABB. Se la minima distanza al quadrato è minore o uguale a r^2, c'è contatto e viene restituito anche un punto di contatto approssimato. Questa procedura è robusta, semplice da calcolare in parallelo e sufficiente per attivare l'evento di esplosione.

Collisioni continue (swept) per colpi veloci

Listing 4.22: Time-of-impact con capsula swept

```
inline bool SweptCapsuleVsAABB(const Capsule& prev, const Capsule&
    curr, const AABB& box, float& toi, glm::vec3& contact) {
     // Approccio conservativo a passi temporali: robusto per colpi
        veloci
     const int STEPS = 12;
     Capsule c; c.r = curr.r; // raggio costante tra i due frame

     // Controllo iniziale (t = 0)
     glm::vec3 hit0;
     if (CapsuleVsAABB(prev, box, hit0)) { toi = 0.0f; contact = hit0;
         return true; }

     for (int i = 1; i <= STEPS; ++i) {
         float t = float(i) / float(STEPS);
```

```
        c.a = glm::mix(prev.a, curr.a, t);
        c.b = glm::mix(prev.b, curr.b, t);
        glm::vec3 h;
        if (CapsuleVsAABB(c, box, h)) {
            toi = t;
            contact = h;
            return true;
        }
    }
    return false;
}
```

Per evitare il *tunneling* con movimenti rapidi tra due frame (ad es. un pugno veloce che "salta" l'oggetto), viene impiegato un test *swept* che interpola linearmente la capsula dallo stato precedente a quello corrente in "STEPS" sotto-campionamenti. Appena si verifica un contatto si ritorna il *time-of-impact* $t \in [0, 1]$ e un punto di contatto. Aumentare `STEPS` migliora la robustezza (a scapito del costo).

AABB–AABB: rilevazione e vettore minimo di separazione

Listing 4.23: Overlap AABB–AABB con push-out su asse minimo

```
inline bool AABBvsAABB_Resolve(const AABB& A, const AABB& B, glm::
   vec3& push){
    glm::vec3 d = A.c - B.c;
    glm::vec3 o = (A.e + B.e) - glm::abs(d); // overlap su assi
    if (o.x<=0 || o.y<=0 || o.z<=0){ push=glm::vec3(0); return false;
        }
    // scelgo l'asse con overlap minore
    if (o.x < o.y && o.x < o.z) {
        push = glm::vec3((d.x<0?-o.x:o.x), 0, 0);
    } else if (o.y < o.z) {
        push = glm::vec3(0, (d.y<0?-o.y:o.y), 0);
    } else{
        push = glm::vec3(0, 0, (d.z<0?-o.z:o.z));
    } return true;
}
```

L'overlap tra due AABB in world-space viene calcolato asse per asse: se su almeno un asse non c'è intersezione, non vi è collisione. In caso contrario si sceglie l'asse con penetrazione minima e si restituisce un vettore **push** da applicare come *push-out* che separa gli oggetti con la minima correzione possibile. Questo metodo è utile per stabilizzare gli appoggi o per evitare che il personaggio attraversi un oggetto.

Integrazione nell'applicazione: AABB locali, piano di base e toggle debug

Listing 4.24: Calcolo degli AABB locali dai modelli

```
auto MakeLocalAABB = [](const Model* m) -> AABB {
    glm::vec3 minB, maxB;
    m->getModelMinMax(minB, maxB);    // <-- nuovo
    AABB a;
    a.c = (minB + maxB) * 0.5f;       // centro = (min+max)/2
    a.e = (maxB - minB) * 0.5f;       // half-extents = (max-min)/2
    return a;
};

if (m_buildingModel)    m_buildingLocalAABB = MakeLocalAABB(
   m_buildingModel.get());
if (m_treeModel)        m_treeLocalAABB     = MakeLocalAABB(
   m_treeModel.get());
if (m_busModel)         m_busLocalAABB      = MakeLocalAABB(
   m_busModel.get());
if (m_phoneBoothModel)  m_phoneLocalAABB    = MakeLocalAABB(
   m_phoneBoothModel.get());
if (m_character && m_character->GetModelPtr())
    m_characterLocalAABB = MakeLocalAABB(m_character->GetModelPtr());
```

Durante l'inizializzazione, per ogni modello viene estratto l'ingombro assegnando centro ed estensioni direttamente dal $\min$ / $\max$ dei vertici in spazio locale. Questi AABB locali vengono poi trasformati frame-by-frame in world space tramite `ToWorldAABB` usando la rispettiva *model matrix*, così da eseguire i test di collisione e (opzionalmente) disegnarli in overlay tramite la pressione del tasto B della tastiera.

Listing 4.25: AABB del piano di base e toggle visuale AABB (tasto B)

```
m_groundLocalAABB.c = glm::vec3(0.0f, 0.0f, 0.0f); // Centro
m_groundLocalAABB.e = glm::vec3(1.0f, 1.0f, 1.0f); // Half-extents:
    da -1 a +1

// === DEBUG BOUNDS TOGGLE (tasto B) ===
{
    bool bPressed = m_window->isKeyPressed(GLFW_KEY_B);
    if (bPressed && !m_prevB) {
        m_drawBounds = !m_drawBounds;
        std::cout << "[DEBUG] Bounding boxes: " << (m_drawBounds ? "
            ON" : "OFF") << std::endl;
    }
    m_prevB = bPressed;
}
```

Il piano di base è modellato come AABB unitario locale (poi scalato con la propria *model matrix*); ciò consente collisioni/poggi con gli altri elementi. A runtime, la pressione del tasto `B` abilita/disabilita la visualizzazione delle AABB di scena, utile per tarare scale, offset e validare i test di collisione in fase di debug.

Aggiunta sull'integrazione con gli eventi di simulazione

Durante l'interazione, quando la capsula del pugno interseca l'AABB di un oggetto (edificio, albero, bus, cabina), viene attivato l'evento corrispondente, cioè l'esplosione dell'oggetto selezionato. Successivamente, si passa allo stato fisico appropriato (con fisica attiva su GPU o CPU) per gestire l'animazione della frammentazione. La scelta tra l'utilizzo della CPU o della GPU per questa funzionalità, con impostazione di default su GPU, avviene tramite il tasto P.

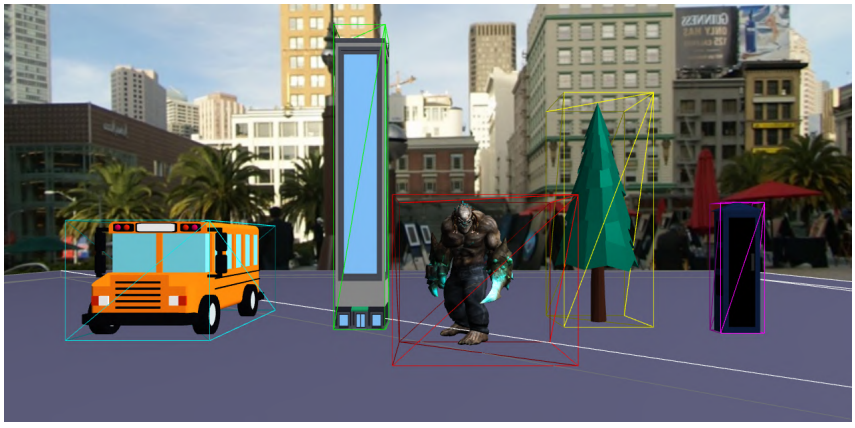

Figura 4.13: Visualizzazione dei BBox degli elementi presenti in scena.

4.8 Compute Shader e fisica dell'esplosione

Questa sezione descrive com'è stata realizzata la fisica in tempo reale dei frammenti tramite **compute shader** OpenGL 4.3 (GLSL). L'idea alla base è spostare il calcolo delle forze (gravità, spinta esplosiva, smorzamento) e degli aggiornamenti di stato (posizione, velocità, integrazione nel tempo, collisione col suolo) in un kernel GPU che lavora in parallelo sui frammenti. Il lato C++ si occupa di compilare e collegare il compute shader, allocare e popolare i buffer (parametri fisici in *UBO*, dati dei frammenti e trasformazioni istanziate in *SSBO*), aggiornare i parametri a ogni frame, lanciare il kernel (`glDispatchCompute`) e sincronizzare con barriere di memoria. *Trigger* e preset fisici sono stati gestiti all'interno della classe `PhysicsSystem`.

Architettura generale: pipeline compute e gestione shader

Il compute shader viene caricato da file (`compute_physics.glsl`), compilato e linkato nel programma OpenGL dedicato. L'inizializzazione crea inoltre i buffer per parametri e dati.
Il calcolo viene poi lanciato ogni frame con un numero di *work groups* proporzionale al numero di frammenti. Il lato C++ calcola i gruppi come $\lceil N/32 \rceil$, dato che il `local_size_x` nello shader è 32; dopo il `dispatch` viene inserita una `glMemoryBarrier` per garantire la visibilità delle scritture su SSBO/UBO al resto della pipeline.

Listing 4.26: Caricamento, compilazione, creazione UBO e dispatch del compute shader

```
bool ComputeShader::initialize() {
    if (loadFromFile("compute_physics.glsl")) {
        createBuffers();                // UBO dei parametri fisici
        m_initialized = true;
        return true;
    }
    return false;
}

void ComputeShader::createBuffers() {
    glGenBuffers(1, &m_paramsUBO);
    glBindBuffer(GL_UNIFORM_BUFFER, m_paramsUBO);
    glBufferData(GL_UNIFORM_BUFFER, sizeof(PhysicsParams), &
       m_physicsParams, GL_DYNAMIC_DRAW);
    glBindBufferBase(GL_UNIFORM_BUFFER, 0, m_paramsUBO);   // UBO
    glBindBuffer(GL_UNIFORM_BUFFER, 0);
}


void ComputeShader::dispatch() {
```

```
    if (!m_initialized || m_numFragments == 0) return;

    glUseProgram(m_computeProgram);
    glBindBufferBase(GL_UNIFORM_BUFFER, 0, m_paramsUBO);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, m_fragmentsSSBO);
       // SSBO frammenti
    if (m_instanceSSBO) glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 2,
        m_instanceSSBO);

    GLuint numGroups = (m_numFragments + 31) / 32;   //  ceil(N/32)
    glDispatchCompute(numGroups, 1, 1);

    glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT |
       GL_BUFFER_UPDATE_BARRIER_BIT);
}
```

Questa parte di codice mostra le tre fasi principali: (1) inizializzazione e compilazione del compute shader; (2) creazione dell' **UBO** dei parametri fisici (binding 0); (3) lancio del kernel con i buffer necessari a bordo (**SSBO** per i frammenti a *binding 1* e, se presente, l'SSBO per l'*instanced rendering* a *binding 2*), seguito da una barriera di memoria per rendere visibili le scritture del kernel.

Struttura del compute shader e fisica implementata

Il cuore della simulazione è racchiuso nel compute shader `compute_physics.glsl`, scritto in linguaggio GLSL (OpenGL Shading Language) con estensioni per il calcolo parallelo e per l'uso dei buffer di memoria (SSBO).

A differenza dei classici vertex o fragment shader, il compute shader non si occupa di grafica ma esegue in parallelo operazioni generiche sui dati. Ogni *work item* (invocazione) gestisce un frammento della scena, permettendo di aggiornare posizione, velocità e orientamento di centinaia di elementi contemporaneamente.

Il frammento di codice riportato di seguito mostra la funzione `main`, dove avvengono i calcoli principali:

Listing 4.27: Funzione main del compute shader

```
void main() {
    uint index = gl_GlobalInvocationID.x;
    if (index >= uint(numFragments)) return;

```

```
    if (resetSimulation != 0) {
        resetFragment(fragments[index]);
        return;
    }

    FragmentData fragment = fragments[index];

    // Impulso iniziale di esplosione
    if (currentTime < 1.0) {
        vec3 toFragment = fragment.initialPosition - explosionCenter;
        float distance = length(toFragment);

        if (explosionRadius > 0.0 && distance < explosionRadius) {
            vec3 explosionDirection = (distance > 1e-3) ? normalize(
               toFragment): vec3(0.0, 1.0, 0.0);

            float forceFalloff = 1.0 - (distance / explosionRadius);
            forceFalloff *= forceFalloff;

            // Rumore per rendere meno uniforme l'esplosione
            vec3 rnd = vec3(
                noise(fragment.initialPosition + vec3(1.0,0.0,0.0)) -
                    0.5,
                noise(fragment.initialPosition + vec3(0.0,1.0,0.0)) -
                    0.5,
                noise(fragment.initialPosition + vec3(0.0,0.0,1.0)) -
                    0.5
            ) * 0.5;

            vec3 explosionVelocity = (explosionDirection + rnd)*
               explosionForce * forceFalloff;

            explosionVelocity.y = abs(explosionVelocity.y) +
               explosionForce * 0.3;

            fragment.velocity = explosionVelocity;
            fragment.structuralIntegrity = 0.0;
            fragment.angularVelocity = rnd * 5.0;
        }
    }
```

```
    // Aggiornamento fisico standard
    if (fragment.structuralIntegrity <= 0.0) {
        int substeps = int(ceil(deltaTime / TARGET_DT));
        substeps = clamp(substeps, 1, MAX_SUBSTEPS);
        float dt = deltaTime / float(substeps);

        for (int s = 0; s < substeps; ++s) {
            float linDamp = exp(-airDamping * dt);
            fragment.velocity = (fragment.velocity + gravity * dt) *
               linDamp;
            fragment.position += fragment.velocity * dt;

            handleGroundCollision(fragment);

            fragment.angularVelocity *= exp(-airDamping * dt);
            fragment.orientation = integrateOrientation(fragment.
               orientation, fragment.angularVelocity, dt);
        }
    }
    fragments[index] = fragment;
}
```

Il linguaggio GLSL utilizza una sintassi simile al C, ma con estensioni per la grafica e il calcolo parallelo. Alcuni aspetti chiave di questo codice sono:

- **Identificazione del work item**: la variabile `gl_GlobalInvocationID.x` identifica l'indice del frammento da elaborare. In questo modo ogni invocazione del compute shader si occupa di un frammento distinto.
- **Reset della simulazione**: se il flag `resetSimulation` è attivo, i frammenti vengono riportati alle condizioni iniziali tramite la funzione `resetFragment`.
- **Impulso iniziale**: all'avvio (`currentTime < 1.0`) viene calcolata una velocità iniziale che spinge i frammenti lontano dal centro dell'esplosione. La forza decresce quadraticamente con la distanza (*force falloff*) ed è resa irregolare da un fattore di rumore pseudo-casuale.
- **Integrazione numerica**: se il frammento è "rotto" (`structuralIntegrity <= 0.0`), vengono calcolati sottopassi temporali (*substeps*) per migliorare la stabilità. L'integrazione delle equazioni del moto avviene con uno schema di Eulero semi-implicito, applicando gravità, smorzamento esponenziale e aggiornando posizione e velocità.

- **Collisione con il terreno**: la funzione `handleGroundCollision` corregge la posizione verticale, applica la restituzione e un attrito semplificato, impedendo ai frammenti di attraversare il piano di base.
- **Rotazioni**: l'orientamento di ciascun frammento è rappresentato da un quaternione; la funzione `integrateOrientation` aggiorna la rotazione in base alla velocità angolare e normalizza il risultato.
- **Scrittura dei risultati**: al termine del calcolo, i dati aggiornati del frammento vengono riscritti nello SSBO, rendendoli disponibili per il rendering istanziato.

Grazie a questa struttura, il compute shader è in grado di gestire in parallelo centinaia di frammenti, garantendo un'evoluzione fisica realistica dell'esplosione con un carico computazionale sostenuto interamente dalla GPU.

Modello dati: parametri fisici (UBO) e frammenti (SSBO)

I parametri fisici globali (centro e intensità dell'esplosione, gravità, smorzamento dell'aria, piano di rimbalzo, Δt, ecc.) risiedono in un **Uniform Buffer Object** e sono aggiornati a ogni frame. Ogni frammento ha invece un record dedicato in un **Shader Storage Buffer Object** con posizione, velocità, massa, raggio di bounding, coefficiente di restituzione, orientamento, forze accumulate, ecc. Questa mappatura 1:1 tra C++ e GLSL riduce gli *overhead* di copia e consente al kernel di leggere/scrivere direttamente i campi necessari.

Listing 4.28: Inizializzazione dei frammenti e creazione SSBO/Instance-SSBO

```
void ComputeShader::initializeFragments(const std::vector<glm::vec3>&
    fragmentCenters,const std::vector<float>& fragmentRadii) {
    m_numFragments = (int)fragmentCenters.size();
    m_fragmentsData.clear(); m_fragmentsData.reserve(m_numFragments);

    for (int i = 0; i < m_numFragments; ++i) {
        FragmentData f{};
        f.position         = fragmentCenters[i];
        f.initialPosition  = fragmentCenters[i];
        f.velocity         = glm::vec3(0.0f);
        f.angularVelocity  = glm::vec3(0.0f);
        f.mass             = 1.0f;
        f.boundingRadius   = fragmentRadii[i];
        f.restitution      = 0.3f;
        f.orientation      = glm::vec4(0,0,0,1);
        f.force            = glm::vec3(0.0f);
        f.structuralIntegrity = 1.0f;
```

```
17          m_fragmentsData.push_back(f);
18      }
19
20      // SSBO dei frammenti (binding 1)
21      glGenBuffers(1, &m_fragmentsSSBO);
22      glBindBuffer(GL_SHADER_STORAGE_BUFFER, m_fragmentsSSBO);
23      glBufferData(GL_SHADER_STORAGE_BUFFER, m_numFragments * sizeof(
            FragmentData), m_fragmentsData.data(), GL_DYNAMIC_DRAW);
24      glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, m_fragmentsSSBO);
25
26      // SSBO per instanced rendering (binding 2)
27      struct InstanceTransform { glm::mat4 modelMatrix; glm::vec3
            velocity; float explosionTime; };
28      glGenBuffers(1, &m_instanceSSBO);
29      glBindBuffer(GL_SHADER_STORAGE_BUFFER, m_instanceSSBO);
30      glBufferData(GL_SHADER_STORAGE_BUFFER, m_numFragments * sizeof(
            InstanceTransform), nullptr, GL_DYNAMIC_DRAW);
31
32      std::vector<InstanceTransform> initial(m_numFragments);
33      for (int i = 0; i < m_numFragments; ++i) {
34          initial[i].modelMatrix = glm::translate(glm::mat4(1.0f),
                fragmentCenters[i]);
35          initial[i].velocity    = glm::vec3(0.0f);
36          initial[i].explosionTime = -1.0f;
37      }
38      glBufferSubData(GL_SHADER_STORAGE_BUFFER, 0, m_numFragments*
            sizeof(InstanceTransform), initial.data());
39      glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 2, m_instanceSSBO);
40      glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
41
42      m_physicsParams.numFragments = m_numFragments;
43      updateBuffers(); }
```

In questa parte di codice invece si vede la costruzione dei record `FragmentData` (posizione iniziale, velocità, massa, raggio, restituzione, ecc.) e l'allocazione dei due SSBO: uno per i *dati fisici* (binding 1), uno per le *trasformazioni per l'instanced rendering* (binding 2), inizializzato con una `modelMatrix` di sola traslazione.
Il numero di frammenti viene sincronizzato nell'UBO per permettere al kernel di conoscere la taglia del problema.+

Innesco dell'esplosione e preset dei parametri

L'innesco dell'esplosione avviene dal `PhysicsSystem`, che inoltra al compute shader il centro dell'esplosione e i parametri correnti (forza, raggio, durata), impostati a seconda del preset. `startExplosion` azzera il tempo interno della simulazione e abilita la dinamica.

Listing 4.29: Trigger dell'esplosione: lato sistema fisico e shader

```
void PhysicsSystem::startExplosion(const glm::vec3& center) {
    m_computeShader->startExplosion(center, m_forceParams.
       explosionForce, m_forceParams.explosionRadius);
    auto& params = const_cast<PhysicsParams&>(m_computeShader->
       getPhysicsParams());
    params.maxTime = m_forceParams.explosionDuration; // durata
}

void ComputeShader::startExplosion(const glm::vec3& center, float
   force, float radius) {
    setExplosionParameters(center, force, radius);
    m_physicsParams.currentTime   = 0.0f;
    m_physicsParams.resetSimulation = 0;
    m_simulationActive = true;
    updateBuffers();
}
```

Qui invece si nota come l'innesco scriva *centro, forza e raggio* nell'UBO e resetti il tempo di simulazione. L'utilizzo dei **preset** (ad es. *realistic* vs *disintegration*) regola gravità, raggio, smorzamento e durata per caratterizzare scenari differenti, mantenendo invariata la struttura della logica alla base dell'esplosione.

Aggiornamento per frame: tempo, dispatch e sincronizzazione

Ad ogni frame, il `PhysicsSystem` valida il Δt, applica un eventuale *time scale*, aggiorna l'UBO e lancia il kernel; al termine, sincronizza e (se la simulazione è attiva) scarica un sottoinsieme dei dati aggiornati per statistiche e diagnostica dei dati (*average/max velocity, energia cinetica*, ...).

Listing 4.30: Loop di aggiornamento della fisica GPU

```
void PhysicsSystem::update(float deltaTime) {
    if (deltaTime <= 0.0f || deltaTime > 0.1f) deltaTime = 1.0f/60.0f
       ;
```

```
    float scaledDeltaTime = deltaTime * m_forceParams.timeScale;

    m_computeShader->update(scaledDeltaTime);  // aggiorna UBO
    m_computeShader->dispatch();               // lancia il compute
        shader
    m_computeShader->synchronize();            // barriera per
        visibilita' SSBO

    if (m_computeShader->isSimulationActive()) {
        m_computeShader->downloadFragmentDataFromGPU(); // debug
    }

    updateDebugInfo();
}
```

La funzione `update` della classe `PhysicsSystem` evidenzia il ciclo standard: aggiornamento dei parametri, `dispatch` del compute, barriera di memoria e (se richiesto) lettura parziale dei risultati per il monitoring di dati e prestazioni, senza interrompere la pipeline di rendering.

Forze e integrazione numerica nel compute shader

Il compute shader applica ai frammenti: **gravità** $\vec{g}$, **smorzamento dell'aria** (forza proporzionale alla velocità, $-k_d\vec{v}$), **spinta esplosiva** radiale centrata in $\vec{c}_{\text{exp}}$ con decadimento rispetto alla distanza e azzerata oltre il raggio impostato.
L'integrazione nel tempo viene effettuata mediante il metodo di Eulero semi-esplicito, usando il Δt fornito via UBO: prima si aggiornano le velocità $\vec{v}^{t+\Delta t} = \vec{v}^{t} + \frac{\vec{F}}{m}\Delta t$, poi le posizioni $\vec{x}^{t+\Delta t} = \vec{x}^{t} + \vec{v}^{t+\Delta t}\Delta t$.
La presenza di gravity, airDamping, explosionCenter, Force, Radius e deltaTime nell'UBO, unita ai campi per massa/velocità nei frammenti, riflette questa pipeline fisica, mentre la collisione col suolo usa il livello `groundLevel` e il coefficiente di `restitution` per il rimbalzo controllato.

Collisione con il suolo e parametri di contatto

Il piano di contatto è orizzontale ($y = \texttt{groundLevel}$) e ogni frammento è trattato come sfera di raggio `boundingRadius`. Quando $y - r \leq \texttt{groundLevel}$ si applica una correzione di posizione e si modifica la velocità verticale con il coefficiente di restituzione ($v_y \leftarrow -e\, v_y$), mentre l' `airDamping` agisce come smorzamento globale. Tali parametri sono impostati dal `PhysicsSystem` tramite `applyForceParameters` e propagati all'UBO.

Reset della simulazione e coerenza dei dati

Per tornare allo stato integro, il lato C++ imposta un flag di *reset* nell'UBO, lancia un `dispatch` per avviare l'operazione sulla GPU e sincronizza il processo; al termine i dati aggiornati vengono scaricati dalla GPU per riallineare lo stato sulla CPU, operazione utile per riportare le posizioni iniziali dei frammenti e ripartire da zero, ad esempio quando viene premuto il tasto `R`.

Listing 4.31: Reset GPU-side con riallineamento dei dati

```
void ComputeShader::resetSimulation() {
    m_physicsParams.resetSimulation = 1;
    m_physicsParams.currentTime = 0.0f;
    m_simulationActive = false;

    updateBuffers();   // scrive l'UBO
    dispatch();        // esegue il reset nel kernel
    synchronize();     // attende la fine

    downloadFragmentDataFromGPU(); // riallinea lo stato lato CPU

    m_physicsParams.resetSimulation = 0;
    updateBuffers();
}
```

Qui viene mostrato l'intero *reset path*: flag su UBO, dispatch, barriera, *readback* selettivo e pulizia del flag, così da garantire coerenza dei dati per l'eventuale re-render o controllo delle statistiche.

Controlli runtime e limiti hardware

Durante l'`update` l'applicazione può ispezionare i limiti hardware del dispositivo (taglia massima dei work-group, invocazioni per gruppo) per diagnosticare problemi di configurazione e tarare al bisogno le dimensioni del dispatch. Questo controllo è eseguito una sola volta e riportato in console insieme alle informazioni di debug.

Listing 4.32: Ispezione dei limiti compute della GPU

```
GLint maxWorkGroupSize[3];
glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_SIZE, 0, &maxWorkGroupSize
    [0]);
glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_SIZE, 1, &maxWorkGroupSize
    [1]);
```

```
glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_SIZE, 2, &maxWorkGroupSize
    [2]);

GLint maxWorkGroupInvocations;
glGetIntegerv(GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS, &
    maxWorkGroupInvocations);

std::cout << "Max work group size: "
          << maxWorkGroupSize[0] << "x" << maxWorkGroupSize[1] << "x"
              << maxWorkGroupSize[2] << "\n"
          << "Max work group invocations: " <<
             maxWorkGroupInvocations << std::endl;
```

Questo frammento di codice serve a interrogare la scheda grafica per conoscere i limiti del compute shader, in particolare la dimensione massima dei gruppi di lavoro e il numero totale di invocazioni consentite per gruppo. Queste informazioni sono utili per configurare correttamente il `dispatch` ed evitare errori di esecuzione. Nel progetto, il compute shader lavora in modalità *data-parallel*, cioè ogni frammento è aggiornato da un work item indipendente.

I parametri globali della simulazione (forze, raggio dell'esplosione, gravità, tempo, ecc.) vengono passati tramite un UBO, mentre lo stato dei singoli frammenti (posizione, velocità, massa, orientamento) è gestito negli SSBO insieme alle trasformazioni necessarie per il rendering istanziato. Grazie a questa architettura semplice ma efficace, la simulazione delle esplosioni resta stabile e reattiva, mantenendo un framerate elevato e permettendo eventuali estensioni future come metodi di integrazione più accurati o nuove tipologie di collisione.

4.9 Confronto sistema GPU e CPU con analisi delle prestazioni

Questo paragrafo mette a confronto le due pipeline fisiche alternative implementate all'interno del simulatore: un *metodo GPU* basato su *compute shader* che integra le equazioni del moto direttamente sulla scheda grafica e un *metodo CPU* che esegue gli stessi aggiornamenti per-frammento sul processore. Entrambe le pipeline utilizzano gli stessi dati d'ingresso (centri e raggi dei frammenti) e possono essere attivate a runtime tramite un toggle dedicato (tasto P della tastiera), garantendo così un confronto omogeneo fra i due metodi..

Implementazione su GPU (Compute Shader)

L'elaborato utilizza i Compute Shader per simulare la fisica dei frammenti in parallelo sulla GPU. Il sistema è incapsulato nella classe `PhysicsSystem`, che funge da interfaccia di alto livello verso lo shader di calcolo. Durante l'inizializzazione viene creato lo shader, configurati i parametri fisici e caricati i dati relativi ai frammenti (posizione, raggio, massa implicita). L'esecuzione ad ogni frame segue una pipeline definita: `update` → `dispatch` → `synchronize`. In questa fase, il sistema applica le forze correnti (gravità, esplosione, attrito) e calcola le nuove posizioni e velocità dei frammenti in modo parallelo su GPU.
L'aggiornamento della simulazione avviene in tempo reale, e opzionalmente si può effettuare un download dei dati aggiornati dal buffer GPU verso CPU per finalità di debug o rendering ibrido. Il seguente listato mostra l'inizializzazione, il caricamento dei frammenti, la gestione dell'esplosione e l'aggiornamento della simulazione al variare del tempo di gioco.

Listing 4.33: GPU: setup, update e parametri

```
bool PhysicsSystem::initialize(StateManager* stateManager) {
    m_stateManager = stateManager;
    m_computeShader = std::make_unique<ComputeShader>();
    if (!m_computeShader->initialize()) return false;
    applyForceParameters();
    return true;
}

void PhysicsSystem::setupFragments(const std::vector<glm::vec3>&
   centers, const std::vector<float>& radii) {
    m_computeShader->initializeFragments(centers, radii);
    applyForceParameters();
}

void PhysicsSystem::startExplosion(const glm::vec3& center) {
```

```
    m_computeShader->startExplosion(center, m_forceParams.
        explosionForce, m_forceParams.explosionRadius);
    auto& params = const_cast<PhysicsParams&>(m_computeShader->
        getPhysicsParams());
    params.maxTime = m_forceParams.explosionDuration;
}

void PhysicsSystem::update(float deltaTime) {
    if (deltaTime <= 0.0f || deltaTime > 0.1f) deltaTime = 1.0f/60.0f
        ;
    float scaled = deltaTime * m_forceParams.timeScale;
    m_computeShader->update(scaled);
    m_computeShader->dispatch();
    m_computeShader->synchronize();
    if (m_computeShader->isSimulationActive()) {
        m_computeShader->downloadFragmentDataFromGPU();
    }
}

void PhysicsSystem::applyForceParameters() {
    glm::vec3 g = m_forceParams.gravity * m_forceParams.gravityScale;
    m_computeShader->setGravity(g);
    m_computeShader->setAirDamping(m_forceParams.airDamping);
    m_computeShader->setExplosionParameters(glm::vec3(0.0f),
        m_forceParams.explosionForce, m_forceParams.explosionRadius);
    m_computeShader->setGroundLevel(-0.60f);
}
```

Implementazione su CPU

L'alternativa CPU, implementata nella classe `PhysicsSystemCPU`, riproduce la stessa semantica fisica utilizzata nel sistema basato su Compute Shader, ma eseguita interamente su architettura seriale o multithreaded a seconda del contesto. L'obiettivo è offrire un sistema fisico equivalente in termini di comportamento e parametri, ma eseguibile anche su dispositivi o configurazioni che non supportano nativamente gli shader di calcolo.
La simulazione è divisa in più fasi: inizializzazione dei frammenti, generazione dell'impulso d'esplosione con una distribuzione spaziale basata su distanza dal centro (falloff) e rumore pseudo-casuale controllato, aggiornamento delle velocità e posizioni tramite integrazione temporale esplicita, e gestione dei contatti con il piano di appoggio (rimbalzo e frizione). Al termine di ogni ciclo di aggiornamento, vengono calcolate le matrici di trasformazione che servono per il rendering di ciascun frammento.

L'impulso iniziale di esplosione assegna a ciascun frammento una velocità proporzionale alla sua distanza dal centro, arricchita da una perturbazione casuale tramite funzione `noise` per simulare l'effetto caotico e non uniforme dell'esplosione. Il sistema gestisce inoltre la gravità, lo smorzamento dell'aria e il contatto col suolo con restituzione elastica e attrito semplificato. Infine, vengono applicate condizioni di sicurezza come il clamping della posizione entro i limiti della scena per evitare errori numerici o artefatti visivi.
Il codice che segue mostra l'inizializzazione dei dati, la logica di generazione dell'impulso esplosivo e l'aggiornamento fisico ad ogni frame, comprensivo di gestione del contatto e restituzione. Viene inoltre registrato il tempo di aggiornamento (`m_lastUpdateTimeMs`) per eventuali analisi prestazionali.

Listing 4.34: CPU: init, impulso d'esplosione e update

```
void PhysicsSystemCPU::initialize(const std::vector<glm::vec3>&
    centers, const std::vector<float>& radii) {
     m_fragments.clear();
     for (size_t i = 0; i < centers.size(); ++i) {
         CPU_Fragment f; f.position = centers[i]; f.initialPosition =
             centers[i];
         f.velocity = glm::vec3(0.0f); f.radius = radii[i]; f.mass =
             1.0f;
         f.structuralIntegrity = 1.0f; m_fragments.push_back(f);
     }
     m_active = false; m_currentTime = 0.0f;
}

void PhysicsSystemCPU::startExplosion(const glm::vec3& center, float
    force, float radius) {
     m_currentTime = 0.0f; m_explosionCenter = center;
     m_explosionForce = force; m_explosionRadius = radius;
     for (size_t i = 0; i < m_fragments.size(); ++i) {
         auto& f = m_fragments[i];
         glm::vec3 toFrag = f.initialPosition - center;
         float distance = glm::length(toFrag);
         if (distance < radius) {
             glm::vec3 dir = distance > 0.001f ? glm::normalize(toFrag
                 ) : glm::vec3(0,1,0);
             float falloff = 1.0f - (distance / radius);
             falloff = glm::pow(falloff, 1.5f);                //
                 dispersione piu' graduale
             glm::vec3 randDir = glm::vec3(
                 noise(f.initialPosition+glm::vec3(1,0,0)) - 0.5f,
```

```
                noise(f.initialPosition+glm::vec3(0,1,0)) * 0.3f,
                noise(f.initialPosition+glm::vec3(0,0,1)) - 0.5f) *
                    0.6f;
            glm::vec3 finalDir = glm::normalize(dir + randDir);
            glm::vec3 v = finalDir * force * falloff;
            v.y = v.y * 0.7f + force * 0.2f;      // componente
                verticale bilanciata
            v.x *= 1.3f; v.z *= 1.3f;             // boost
                orizzontale
            f.velocity = v; f.structuralIntegrity = 0.0f;
        }
    }
    m_active = true;
}

void PhysicsSystemCPU::update(float dt) {
    if (!m_active) { m_lastUpdateTimeMs = 0.0f; return; }
    auto start = std::chrono::high_resolution_clock::now();
    m_currentTime += dt;
    for (auto& f : m_fragments) if (f.structuralIntegrity <= 0.0f) {
        f.velocity += m_gravity * dt;             // gravita'
        float speed = glm::length(f.velocity);    // smorzamento
            aria
        if (speed > 0.001f) f.velocity *= (1.0f - m_airDamping * 0.5f
            * dt);
        f.position += f.velocity * dt;            // integrazione
        float ground = m_groundLevel + f.radius + 0.05f;
        if (f.position.y <= ground) {             // contatto piano
            + restituzione
            f.position.y = ground;
            if (f.velocity.y < -0.1f) f.velocity.y = -f.velocity.y *
                m_restitution;
            else f.velocity.y = 0.0f;
            f.velocity.x *= 0.9f; f.velocity.z *= 0.9f;      //
                frizione semplice
            if (glm::length(f.velocity) < 0.01f) f.velocity = glm::
                vec3(0);
        }
        f.position = glm::clamp(f.position, glm::vec3(-100), glm::
            vec3(100));
        if (f.position.y < m_groundLevel - 1.0f) { // safety clamp
```

```
            f.position.y = ground; f.velocity = glm::vec3(0);
        }
    }
    auto end = std::chrono::high_resolution_clock::now();
    m_lastUpdateTimeMs = std::chrono::duration<float,std::milli>(end
        - start).count();
}

std::vector<glm::mat4> PhysicsSystemCPU::getFragmentTransforms()
    const {
    std::vector<glm::mat4> T;
    for (const auto& f : m_fragments)
        T.push_back(glm::translate(glm::mat4(1.0f), f.position)); //
            solo traslazione
    return T;
}
```

Toggle runtime e misura del tempo di fisica

Il passaggio CPU/GPU è gestito da `Application`: con il tasto `P` si inverte `m_useCPUPhysics` e si aggiorna subito il `PerformanceMonitor`. Nel loop di update, il tempo fisico è misurato con orologi ad alta risoluzione e inviato al monitor insieme al numero di frammenti attivi.

Listing 4.35: Toggle CPU/GPU e tempi di fisica (in 'Application.cpp')

```
m_window->setKeyCallback([this](int key,int scancode,int action,int
    mods){
    if (key == GLFW_KEY_P && action == GLFW_PRESS) {
        m_useCPUPhysics = !m_useCPUPhysics;
        std::cout << "[TOGGLE] Physics method: "
                  << (m_useCPUPhysics ? "CPU" : "GPU") << std::endl;
        if (m_performanceMonitor) {
            m_performanceMonitor->setPhysicsMode(m_useCPUPhysics ? "
                CPU" : "GPU");
        }
    }
});

// ... poi nel main update():
if (shouldRunBuildingPhysics && m_physicsSystem && m_physicsSystem->
    isInitialized()) {
    if (m_performanceMonitor) {
```

```
15          m_performanceMonitor->setPhysicsMode(m_useCPUPhysics ? "CPU"
                : "GPU");
16      }
17
18      if (m_useCPUPhysics) {
19          auto cpuStart = std::chrono::high_resolution_clock::now();
20          m_physicsCPU.update(deltaTime);
21          auto cpuEnd = std::chrono::high_resolution_clock::now();
22          float cpuMs = std::chrono::duration<float,std::milli>(cpuEnd
                - cpuStart).count();
23          int active = m_physicsCPU.getFragmentCount();
24          if (m_performanceMonitor) m_performanceMonitor->
                setPhysicsMetrics(active, 0, cpuMs);
25      } else {
26          auto gpuStart = std::chrono::high_resolution_clock::now();
27          m_physicsSystem->update(deltaTime);
28          auto gpuEnd = std::chrono::high_resolution_clock::now();
29          float gpuMs = std::chrono::duration<float,std::milli>(gpuEnd
                - gpuStart).count();
30          int active = m_physicsSystem->getFragmentCount();
31          if (m_performanceMonitor) m_performanceMonitor->
                setPhysicsMetrics(active, 0, gpuMs);
32      }
33  } else {
34      if (m_performanceMonitor) {
35          m_performanceMonitor->setPhysicsMode("Idle");
36          m_performanceMonitor->setPhysicsMetrics(0, 0, 0.0f);
37      }
38  }
```

Raccolta delle prestazioni ed export

La telemetria è invece centralizzata nel `PerformanceMonitor`: ogni frame aggiorna FPS e frame time, mantiene uno storico dei dati raccolti, salva la modalità fisica corrente e permette di esportare questi dati in un file `CSV` dedicato per l'analisi delle prestazioni a programma spento.

Listing 4.36: PerformanceMonitor: aggiornamento e export CSV

```
1  void PerformanceMonitor::updateFPSMetrics(float deltaTime) {
2      if (deltaTime < 0.016f) deltaTime = 0.016f;
3      if (deltaTime > 0.1f)   deltaTime = 0.1f;
4      m_currentMetrics.currentFPS = 1.0f / deltaTime;
```

```
5       if (m_currentMetrics.currentFPS > 60.0f) m_currentMetrics.
            currentFPS = 60.0f;
6       if (m_currentMetrics.currentFPS < 10.0f) m_currentMetrics.
            currentFPS = 10.0f;
7       m_currentMetrics.frameTime = 1000.0f / m_currentMetrics.
            currentFPS;
8   }
9
10  void PerformanceMonitor::setPhysicsMode(const std::string& mode) {
11      m_currentMetrics.physicsMode = mode;
12  }
13
14  void PerformanceMonitor::setPhysicsMetrics(int activeFragments,int
        sleeping,float physicsTime) {
15      m_currentMetrics.activeFragments = activeFragments;
16      m_currentMetrics.sleepingFragments = sleeping;
17      m_currentMetrics.physicsTime = physicsTime;
18      m_currentMetrics.physicsActive = (physicsTime > 0.0f);
19  }
20
21  void PerformanceMonitor::exportMetricsToCSV(const std::string&
        filename) const {
22      std::ofstream file(filename);
23      file << "Timestamp;FPS;FrameTime_ms;GPUUsage_%;"
24           << "RenderTime_ms;CPUUsage_%;"
25           << "MemoryUsage_%;"
26           << "PhysicsTime_ms;PhysicsMode" << std::endl;
27      size_t maxSize = std::max({ m_fpsHistory.size(),
            m_frameTimeHistory.size(), m_gpuUsageHistory.size(),
            m_renderTimeHistory.size(), m_cpuUsageHistory.size(),
            m_memoryUsageHistory.size(), m_physicsTimeHistory.size(),
            m_physicsModeHistory.size() });
28  }
```

Protocollo di confronto

Per ogni scena si procede così: prima avviene l'inizializzazione parallela dei due sistemi con gli stessi frammenti, poi avviene l'innesco dell'esplosione con i parametri fisici. Segue il campionamento di FPS, frame time e tempo fisico (da `Application`) e la memorizzazione della modalità attiva (da `PerformanceMonitor`). A questo punto si esportano i dati raccolti nel file CSV per un'analisi statistica (medie, percentili, varianza).

Alla fine dell'applicazione i dati vengono salvati automaticamente:

Listing 4.37: Export automatico a fine run

```
if (m_performanceMonitor) {
    m_performanceMonitor->exportMetricsToCSV("performance_data.csv");
}
```

Analisi delle prestazioni

Per valutare l'efficienza dei due approcci fisici, è stata condotta una serie di test su un PC fisso con **GPU NVIDIA GeForce GTX 750 Ti** dotata di 2 GB di memoria dedicata, fino a 8 GB di memoria condivisa (per un totale di 10 GB di memoria grafica disponibile) e 16 GB di RAM di sistema.

Il simulatore è stato eseguito in due modalità: con la fisica gestita da *compute shader* (GPU) e con la fisica gestita dalla CPU tramite integrazione per-frammento.

I modelli .obj dell' autobus, dell'albero e della cabina telefonica sono stati suddivisi tramite l'add-on Cell-Fracture di Blender in circa 100 frammenti, mentre il palazzo, il modello più grande è stato invece diviso in più frammenti (circa 300).
L'immagine sottostante mostra come apparirebbero i modelli frantumati all'interno della scena. Ovviamente durante la simulazione l'utente non visualizza mai questa disposizione in quanto prima vede i modelli interi con le texture caricate e poi, quando il personaggio colpisce l'oggetto selezionato, vede il modello esplodere nei vari frammenti di color arancione.

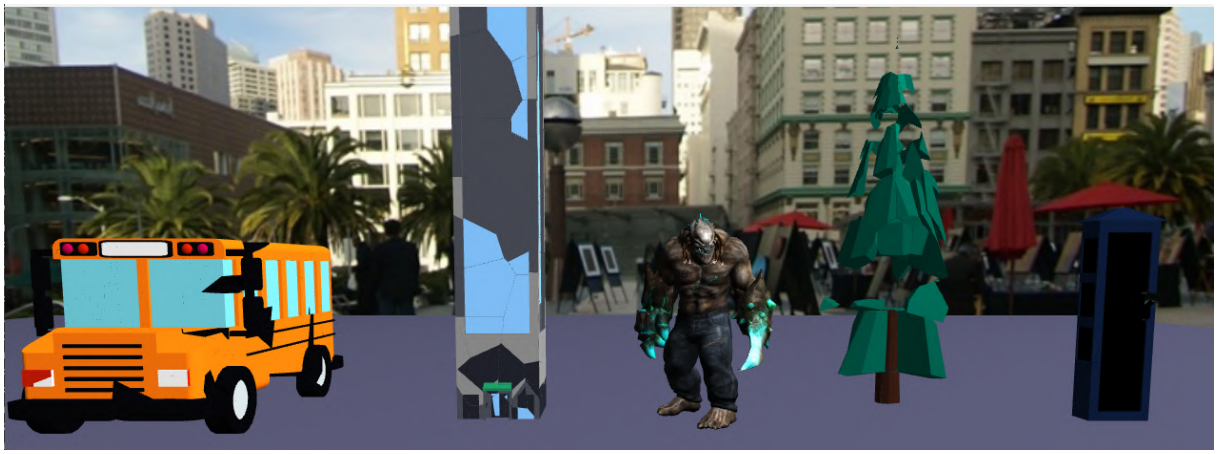

Figura 4.14: Visualizzazione della scena con i modelli frammentati.

Confronto delle Statistiche aggregate

Nelle seguenti 4 tabelle vengono riportati diversi valori, divisi a coppie: Figura 4.15 e Figura 4.16 mostrano le prestazioni di GPU e CPU con circa 600 frammenti utilizzati, mentre Figura 4.17 e Figura 4.18 presentano i risultati ottenuti su scene di complessità superiore,contenenti oltre 1000 frammenti.

FPS	FrameTim	GPUUsage	RenderTir	CPUUsage	MemoryU	PhysicsTir	PhysicsMode
832,016	1,202	69,554	0,394	10,39	34	0,62	GPU
828,363	1,207	69,443	0,395	11,642	34	0,633	GPU
810,11	1,234	69,339	0,402	10,924	34	0,632	GPU
770,475	1,298	69,386	0,483	10,396	34	0,621	GPU
792,77	1,261	69,307	0,442	12,875	34	0,626	GPU
459,517	2,176	69,207	1,123	11,593	34	0,493	GPU
712,301	1,404	69,154	0,655	9,859	34	0,609	GPU
824,606	1,213	69,067	0,395	11,571	34	0,627	GPU
429,682	2,327	68,911	1,223	11,745	34	0,496	GPU
827,061	1,209	69,699	0,397	12,286	34	0,62	GPU
765,111	1,307	68,981	0,484	9,703	34	0,625	GPU
763,242	1,31	69,029	0,497	11,303	34	0,616	GPU
822,301	1,216	69,117	0,404	10,141	34	0,623	GPU
814,664	1,228	69,061	0,399	12,812	34	0,649	GPU
820,143	1,219	69,015	0,401	9,773	34	0,613	GPU

Figura 4.15: Prestazioni simulazione su GPU - Caso 600 Frammenti.

FPS	FrameTim	GPUUsage	RenderTir	CPUUsage	MemoryU	PhysicsTir	PhysicsMode
1441,13	0,694	69,015	0,486	11,691	34	0,004	CPU
1331,203	0,751	69,111	0,522	13,637	34	0,003	CPU
1440,715	0,694	69,314	0,437	11,628	34	0,003	CPU
1474,491	0,678	69,427	0,447	11,749	34	0,004	CPU
1393,534	0,718	69,395	0,507	11,576	34	0,004	CPU
1213,003	0,824	69,422	0,421	9,742	34	0,003	CPU
1424,704	0,702	69,745	0,488	11,749	34	0,004	CPU
1403,706	0,712	69,727	0,514	9,743	34	0,003	CPU
1432,049	0,698	69,722	0,486	11,691	34	0,004	CPU
1642,845	0,609	69,732	0,41	13,568	34	0,003	CPU
1619,433	0,618	69,812	0,409	9,742	34	0,003	CPU
1451,379	0,689	69,886	0,476	13,639	34	0,003	CPU
1594,388	0,627	68,53	0,43	11,69	34	0,003	CPU
1440,922	0,694	68,578	0,485	11,634	34	0,004	CPU
1512,859	0,661	68,768	0,401	13,569	34	0,003	CPU

Figura 4.16: Prestazioni simulazione su CPU - Caso 600 Frammenti.

FPS	FrameTim	GPUUsage	RenderTir	CPUUsage	MemoryU	PhysicsTir	PhysicsMode
417,659	2,394	69,314	1,36	9,736	31	0,843	GPU
354,12	2,824	69,395	1,514	12,875	30	0,834	GPU
400,786	2,495	69,422	1,43	13,391	30	0,83	GPU
410,105	2,438	69,559	1,413	11,648	30	0,839	GPU
368,067	2,717	69,699	1,679	12,15	30	0,83	GPU
328,181	3,047	69,722	2,013	13,454	30	0,791	GPU
324,623	3,081	69,734	2,008	10,909	30	0,786	GPU
335,57	2,98	69,745	1,954	9,554	30	0,813	GPU
445,891	2,243	69,727	1,225	13,401	30	0,834	GPU
440,956	2,268	69,732	1,252	9,696	30	0,831	GPU
380,749	2,626	69,812	1,587	13,62	30	0,831	GPU
395,132	2,531	69,874	1,495	9,789	30	0,82	GPU
391,007	2,558	69,886	1,537	12,799	30	0,823	GPU
365,03	2,74	69,757	1,694	12,01	30	0,806	GPU
366,892	2,726	69,696	1,732	11,124	30	0,804	GPU

Figura 4.17: Prestazioni simulazione su GPU - Caso 1000 Frammenti.

FPS	FrameTim	GPUUsage	RenderTir	CPUUsage	MemoryU	PhysicsTir	PhysicsMode
509,944	1,961	68,704	1,717	11,634	30	0,015	CPU
503,677	1,985	68,595	1,731	11,634	30	0,015	CPU
509,684	1,962	68,39	1,732	13,44	30	0,015	CPU
500,175	1,999	68,288	1,765	11,75	30	0,015	CPU
455,602	2,195	69,029	1,955	13,573	30	0,015	CPU
589,345	1,697	69,117	1,48	11,52	30	0,014	CPU
583,567	1,714	69,061	1,492	11,52	30	0,015	CPU
555,37	1,801	69,015	1,555	7,833	30	0,015	CPU
543,006	1,842	69,111	1,592	11,634	30	0,015	CPU
443,577	2,254	69,314	1,973	13,639	30	0,011	CPU
394,089	2,538	69,427	2,271	11,634	30	0,016	CPU
508,828	1,965	69,395	1,741	13,573	30	0,014	CPU
424,268	2,357	69,045	2,033	13,373	30	0,011	CPU
526,067	1,901	68,831	1,672	11,576	30	0,015	CPU
527,482	1,896	68,67	1,665	13,64	30	0,015	CPU

Figura 4.18: Prestazioni simulazione su CPU - Caso 1000 Frammenti.

Dall'analisi dei dati riportati, emerge che aumentando il numero di frammenti inizializzati nella scena, entrambi gli approcci tendono naturalmente a diminuire le proprie prestazioni, ma in modo diverso: l'approccio GPU con Compute Shader tende ad abbassare gradualmente i Frame Per Secondo (FPS), mentre l'approccio CPU mostra un crollo nettamente più drastico.

Discussione dei risultati

PRESTAZIONI	GPU	CPU
FPS medio	774.5	1452.8
FPS min	459.5	1213.0
FPS max	832.0	1642.8
Tempo fisica medio (ms)	0.6	0.003
Uso medio GPU (%)	69.2	69.3
Uso medio CPU (%)	11.1	11.8

Tabella 4.2: Confronto tra le due modalità di simulazione - Caso 600 Frammenti.

PRESTAZIONI	GPU	CPU
FPS medio	376.0	502.5
FPS min	324.6	394.1
FPS max	441.0	589.3
Tempo fisica medio (ms)	0.7	0.015
Uso medio GPU (%)	69.6	68.9
Uso medio CPU (%)	11.6	12.2

Tabella 4.3: Confronto tra le due modalità di simulazione - Caso 1000 Frammenti.

Dall'analisi dei log emergono le seguenti evidenze sullo scenario dei circa 600 frammenti:

- **Frame rate**: nel caso dei 600 frammenti, la modalità *CPU* raggiunge in media $\sim$1452 FPS (picchi a $\sim$1642 FPS), mentre la *GPU* si assesta intorno a $\sim$774 FPS (max $\sim$832 FPS). Il minimo in CPU resta sui $\sim$1213 FPS mentre in GPU il minimo è più basso, $\sim$459 FPS.
- **Tempo di fisica**: la *GPU* oscilla mediamente sui 0.6 ms per frame con rari picchi (spike) in corrispondenza dell'innesco dell'esplosione, mentre la *CPU* resta nell'ordine dei microsecondi (0.003ms).
- **Carico risorse**: la percentuale di uso della GPU è simile (68–70%) in entrambe le modalità, segno che il rendering domina comunque il frame time. L'uso della CPU invece è leggermente più alto in *CPU mode* ($\sim$12%) rispetto alla *GPU mode* ($\sim$11%).

Secondo Scenario: oltre 1000 frammenti

Aumentando a 1000 il numero di frammenti, le prestazioni tendono a calare sia per la CPU che per la GPU, ma con dinamiche molto diverse. La GPU continua a beneficiare del parallelismo massivo offerto dai suoi *Streaming Multiprocessors* (SM), mentre la CPU mostra un degrado molto più marcato dovuto al numero limitato di core fisici disponibili.
Analizzando i dati delle tabelle si osserva che:

- **FPS medio e minimo**: passando da 600 a 1000 frammenti, gli FPS medi calano sia in GPU (da $\sim$774 a $\sim$376) che in CPU (da $\sim$1453 a $\sim$503). Tuttavia, il crollo relativo è molto più pronunciato per la CPU, che perde oltre il 65% delle prestazioni, mentre la GPU si mantiene più stabile. Anche i valori minimi seguono lo stesso andamento: la GPU resta sopra i 324 FPS, mentre la CPU scende a poco più di 394 FPS.

- **Tempo fisica medio**: entrambi i sistemi mostrano un incremento del tempo di calcolo, ma ancora una volta la GPU riesce a contenere l'aumento (da $\sim$0.6 ms a $\sim$0.7 ms), mentre la CPU cresce in maniera molto più drastica (da $\sim$0.003 ms a $\sim$0.015 ms), segnalando una perdita significativa di efficienza al crescere della complessità della scena.

- **Uso risorse**: le percentuali di utilizzo medio rimangono simili tra i due scenari, indicando che il limite non risiede tanto nella saturazione delle risorse quanto nell'architettura intrinseca: la GPU scala meglio al crescere del carico, mentre la CPU tende rapidamente al collo di bottiglia.

Accorgimenti al codice che potrebbero portare al miglioramento delle prestazioni GPU sono:

1. **Parallelismo massivo**: più frammenti presenti $\Rightarrow$ più thread attivi in contemporanea.
   Si arriverà al punto in cui gli FPS per la CPU raggiungeranno quelli della GPU, addirittura scendendo sotto di essi.

2. **Occupazione e latency hiding**: con più lavoro si saturano gli SM e si mascherano le latenze di memoria.

3. **Banda/coalescenza memoria**: accessi vettoriali e aggregati riducono i costi per frammento.

4. **Ammortamento overhead fisso**: i costi di lancio kernel/sincronizzazione pesano sempre meno all'aumentare di N.

In pratica, al raddoppiare del numero di frammenti, il tempo di calcolo della CPU tende ad aumentare in maniera pressoché lineare, mentre quello della GPU cresce più lentamente. Di conseguenza, il rapporto $T_{\mathrm{CPU}}/T_{\mathrm{GPU}}$ aumenta con l'incremento di N, fino a raggiungere la saturazione della GPU, oltre la quale il vantaggio si stabilizza.

Sintesi

Dall'analisi dei dati raccolti da entrambe le simulazioni e con diverso numero di frammenti, emergono le seguenti considerazioni principali:

1. In assenza di VSync, entrambe le modalità garantiscono l'esecuzione del programma in *real-time*, ossia un aggiornamento sufficientemente veloce da mantenere la simulazione fluida e interattiva (buoni FPS anche durante l'animazione delle esplosioni).
   Tuttavia, la *CPU* mostra FPS medi più elevati sull'attuale dataset.

2. La *GPU* introduce un costo di simulazione fisica medio dell'ordine di pochi millisecondi, con occasionali picchi in corrispondenza di eventi impulsivi (esplosioni).
   Tali oscillazioni risultano tuttavia mitigabili mediante ottimizzazioni a livello di kernel e gestione della memoria.

3. Con l'aumentare del numero di frammenti, la *GPU* si conferma la scelta preferibile per via del parallelismo massivo e della migliore scalabilità, mentre le prestazioni della *CPU* degradano in maniera pressoché lineare con la numerosità degli elementi simulati.

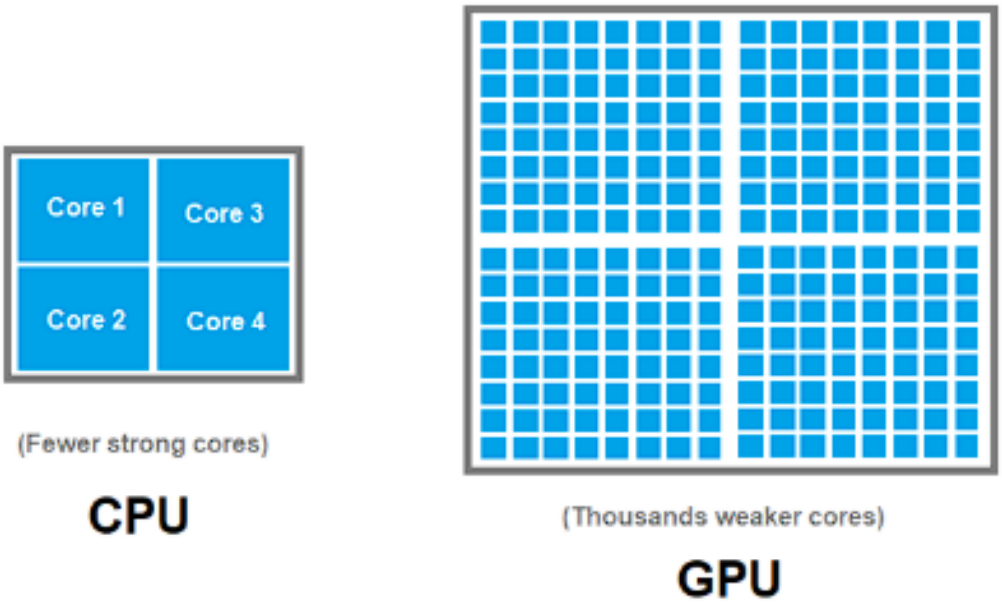

Figura 4.19: Differenza del numero di Core tra CPU e GPU.

Capitolo 5

Considerazioni finali sul progetto e prospettive future

5.1 Considerazioni finali sul progetto

Il progetto realizzato ha dimostrato come l'impiego dei *Compute Shader* su GPU, all'interno di un'architettura modulare OpenGL, permetta di ottenere simulazioni di esplosioni architettoniche con elevato livello di realismo e buone prestazioni, in cui la gestione simultanea di centinaia di frammenti, in presenza di forze fisiche quali gravità, attrito e collisioni, è stata implementata tramite un motore fisico basato sul metodo di Eulero semi-esplicito.

I test effettuati mostrano che, in condizioni di simulazione con un numero contenuto di frammenti, le performance attuali tra versione multithread su CPU e versione su GPU/Compute Shader risultano comparabili, ma con FPS più alti durante le simulazioni in CPU.
Tuttavia, l'approccio GPU risulta essere particolarmente vantaggioso quando il carico computazionale aumenta, ad esempio in scenari caratterizzati da un numero elevato di frammenti o entità fisiche da gestire simultaneamente. Questo vantaggio è dovuto al parallelismo massivo disponibile sulle GPU, che consistono di migliaia di core in grado di elaborare dati in parallelo, una caratteristica che consente una maggiore scalabilità rispetto alla CPU anche se quest'ultima dispone di molti core.

In contesti di simulazioni fisiche su larga scala (ad esempio fluidi o materiali granulari), vari studi mostrano come l'utilizzo della GPU offra significativi incrementi di throughput, arrivando a migliorare le prestazioni anche di decine di volte rispetto a implementazioni sequenziali su CPU.

Inoltre, l'architettura dei *Compute Shader* consente l'utilizzo di tecniche come Shared Local Data o gruppi di lavoro ottimizzati, migliorando ulteriormente l'efficienza in scenari ad alto carico.

Questi elementi suggeriscono che, sebbene le prestazioni attuali tra CPU e GPU siano simili in scenari leggeri, l'adozione della GPU si giustifica pienamente al crescere della complessità e del carico di simulazione. L'approccio basato su Compute Shader risulta quindi strategico e robusto, fornendo un solido punto di partenza per estensioni future finalizzate a gestire ambienti altamente dinamici e densi.

5.2 Possibili estensioni future del progetto

Sulla base delle considerazioni precedenti, emergono diverse direzioni di approfondimento per estendere il progetto:

- **Aumento della complessità della scena:** l'utilizzo della GPU diventa ancora più cruciale per gestire simulazioni con un elevato numero di frammenti o entità distruttibili, dove il parallelismo può mantenere un frame rate stabile. Un'idea potrebbe essere quella quindi di andare ad aggiungere ancora più oggetti alla scena, aumentando anche significativamente il numero dei frammenti che li compongono.
- **Ottimizzazione via strutture dati e meccanismi avanzati:** sfruttare tecniche come data buffering, dispatch indiretti e memoria condivisa (Shared Local Data) su GPU per migliorare latenza e larghezza di banda in simulazioni elaborate.
- **Metodi numerici avanzati: passaggio a RK4 per stabilità e precisione**: implementare una versione del motore fisico basata sul metodo di integrazione esplicita Runge–Kutta di quarto ordine (RK4), noto per offrire un bilanciamento ottimale tra accuratezza e costo computazionale rispetto al metodo di Eulero. RK4 riduce l'errore globale da ordine $O(h)$ a $O(h^4)$, permettendo l'utilizzo di passi temporali maggiori senza comprometterne la stabilità.
- **Integrazione di tecniche di intelligenza artificiale**: guardando ancora più avanti, un'area di ricerca promettente è l'integrazione di tecniche di intelligenza artificiale all'interno del sistema. L'uso di reti neurali o modelli predittivi potrebbe servire, ad esempio, per ottimizzare il comportamento dei frammenti, prevedere traiettorie realistiche sulla base di dati precedenti, o generare distribuzioni fisiche credibili senza dover calcolare ogni interazione in tempo reale.

Il progetto attuale costituisce una solida base che unisce efficienza, realismo e modularità, ideale per evolversi verso simulazioni sempre più dense e complesse in ambito cinematografico o videoludico.

Capitolo 6

Conclusione e Ringraziamenti

Giunti alla conclusione di questo progetto, posso affermare con soddisfazione che si è trattato di un lavoro che ha rappresentato, per me, la perfetta chiusura di un cerchio all'interno del mio percorso universitario. Questo elaborato ha racchiuso molte delle competenze acqusite nel corso dei 3 anni di studio, intrecciando in modo concreto e coerente discipline diverse: dalla Computer Graphics, base del progetto e della sua realizzazione, alla Fisica, che ha fornito gli strumenti necessari per modellare le forze coinvolte nella simulazione dell'esplosione, fino ad arrivare ai concetti di programmazione concorrente e multithreading appresi in Sistemi Operativi. Sono riuscito inoltre ad utilizzare collegamenti con aspetti normativi, grazie agli approfondimenti affrontati nel corso di Diritto riguardanti l'intelligenza artificiale e le sue implicazioni legali, lasciando anche spazio ad una parte più personale, legata alla mia passione per il cinema. È stato un lavoro impegnativo, spesso faticoso e stressante, ma che mi ha reso orgoglioso non solo per il risultato ottenuto, ma soprattutto per la crescita personale e professionale che mi ha permesso di raggiungere.

Ci tengo quindi a ringraziare in primis la professoressa Damiana Lazzaro, la quale si è sempre posta con gentilezza e disponibilità sia durante i corsi affrontati durante i 3 anni, sia durante l'intero percorso di lavoro sulla tesi. Dalla richiesta di essere la mia relatrice, alla conclusione dell'elaborato e alla stesura di questa tesi, il suo supporto è stato fondamentale.
Ringrazio poi i miei compagni, i quali hanno condiviso con me questi 3 anni, facendo sembrare più leggeri di quello che erano i momenti più impegnativi: in particolare durante il periodo di tesi il confronto, gli aggiornamenti costanti e il sostegno reciproco sono stati preziosi e determinanti, sempre vissuti con spirito di collaborazione, serietà e un sorriso.
Un ringraziamento va anche a chi mi ha "sopportato" più che "supportato" durante questo periodo stressante.

Ultimo ma non per importanza, ringrazio Federico, che non si è mai arreso nemmeno quando tutto sembrava andare male e la testa gli diceva di mollare. Complimenti, ce l'hai fatta.

Bibliografia

[1] Houdini. Disponibile su: https://www.sidefx.com/products/houdini/

[2] Maya. Disponibile su: https://www.autodesk.com/ch-it/products/maya/overview

[3] Blender, Utilizzato come base teorica per realizzare l'effetto di "Cell-Fracture". Disponibile su: https://www.blender.org/

[4] Unreal Engine. Disponibile su: https://www.unrealengine.com/en-US

[5] Boords. *Filmmaking 101: What is CGI in Movies and Animation*. Disponibile su: https://boords.com/blog/filmmaking-101-what-is-cgi-in-movies-and-animation

[6] Riaz. *Exploring the Uses of Computer Graphics in Film and VFX*. Disponibile su: https://medium.com/@riaz.cse260/exploring-the-uses-of-computer-graphics-in-film-and-vfx-6d7fa1313184

[7] N-iX Game & VR Studio. *3D Character Development Pipeline*. Disponibile su: https://gamestudio.n-ix.com/3d-character-development-pipeline/

[8] Wikipedia. *Sistema particellare*. Disponibile su: https://it.wikipedia.org/wiki/Sistema_particellare

[9] CERN Courier. *Building Gargantua*. Disponibile su: https://cerncourier.com/a/building-gargantua/

[10] Adobe. *Morphing in Animation*. Disponibile su: https://www.adobe.com/it/creativecloud/animation/discover/morphing-in-animation.html

[11] Khronos Group. *Compute Shader – OpenGL Wiki*. Disponibile su: https://www.khronos.org/opengl/wiki/Compute_Shader.

[12] Mike Bailey. *OpenGL Compute Shaders – Particle System Example*. Dispense del corso di Computer Graphics, Oregon State University, dicembre 2023. Disponibile su: https://web.engr.oregonstate.edu/~mjb/cs557/Handouts/compute.shader.6pp.pdf

[13] Khronos Group *Image Load Store*. Disponibile su: https://www.khronos.org/opengl/wiki/Image_Load_Store

[14] CUDA. Disponibile su: https://forums.developer.nvidia.com/t/opengl-computeshader-vs-cuda-can-computeshader-replace-cuda/258830

[15] What is the Difference between OpenCl and OpenGL's compute shader? Disponibile su: https://stackoverflow.com/questions/15868498/what-is-the-difference-between-opencl-and-opengls-compute-shader

[16] SYCL. Disponibile su: https://www.khronos.org/sycl/

[17] Shader Storage Buffer Object. Disponibili su https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object

[18] William T. Reeves. *Particle Systems – A Technique for Modeling a Class of Fuzzy Objects*. Note didattiche del corso di Informatique Graphique, Université Paris-Saclay. Disponibile su: https://www.lri.fr/~mbl/ENS/IG2/devoir2/files/docs/fuzzyParticles.pdf

[19] Ciara Belle. *Particle Systems: Theory and Practice*. Progetto indipendente, University of Maryland, 2012. Disponibile su: https://www.cs.umd.edu/~mount/Indep/Ciara_Belle/ciara-partic-system-final-2012.pdf

[20] Andrew Witkin, David Baraff. *Physically Based Modeling: Principles and Practice*. SIGGRAPH Course Notes, 2001. Disponibile su: https://graphics.pixar.com/pbm2001/pdf/notesc.pdf

[21] Metodo di Eulero Disponibile su: https://it.wikipedia.org/wiki/Metodo_di_Eulero

[22] Metodo di Eulero Semi-Esplicito Disponibile su: https://it.wikipedia.org/wiki/Metodo_di_Eulero_semi-implicito

[23] Metodo di Verlet. Disponibile su: https://en.wikipedia.org/wiki/Verlet_integration

[24] Metodo di Runge-Kutta. Disponibile su: https://it.wikipedia.org/wiki/Metodi_di_Runge-Kutta

[25] OpenGameArt.org, Fonte dove ho reperito lo skybox per il mio progetto. Disponibile su: https://opengameart.org/content/urban-skyboxes?page=1

[26] Mixamo.com, Fonte dove ho reperito il modellino 3d del personaggio e le relative animazioni associate. Disponibile su: https://www.mixamo.com/#/

[27] Canale Youtube di Geopop, ho preso questo video come riferimento per avere una ispirazione realistica e scientifica sul crollo del palazzo. Disponibile su: https://www.youtube.com/watch?v=xjj3oYdBJsU

[28] Olgdev.org, utilizzato come base teorica per quanto riguarda l'utilizzo di un personaggio e delle sue animazioni nella scena. Disponibile su: https://www.ogldev.org/www/tutorial38/tutorial38.html

[29] Sito ufficiale di OpenGL. Disponibile su: https://www.opengl.org/

[30] Sito ufficiale di GLFW. Disponibile su: https://www.glfw.org/

[31] Riferimenti a GLAD all'interno della wiki opengl. Disponibile su: https://www.khronos.org/opengl/wiki/OpenGL_Loading_Library

[32] Sito ufficiale di khronos con riferimenti ad opengl. Disponibile su: https://registry.khronos.org/OpenGL/index_gl.php

[33] Riferimenti ad Assimp all'interno del sito di learnOpenGL. Disponibile su: https://learnopengl.com/Model-Loading/Assimp

[34] Riferimenti alla libreria GLM all'interno del sito ufficiale di OpenGL. Disponibile su: https://www.opengl.org/sdk/libs/GLM/

[35] Sean T. Barrett, *stb_image.h – Header-only public-domain image loading library*. Disponibile su: https://github.com/nothings/stb

[36] Agenda Digitale, *L'IA nel cinema: una nuova era per produzione e creatività*. Disponibile su: https://www.agendadigitale.eu/cultura-digitale/lia-nel-cinema-una-nuova-era-per-produzione-e-creativita/

[37] Parlamento Europeo. *Regolamento sull'intelligenza artificiale (AI Act)*. Versione approvata in prima lettura il 13 marzo 2024. Disponibile su: https://eur-lex.europa.eu/legal-content/IT/TXT/?uri=CELEX:52021PC0206

[38] Senato della Repubblica Italiana. *Proposta di legge sull'intelligenza artificiale — Introduzione dell'art. 612-quater c.p.*. 2024. Disponibile su: https://futurodigitale.infocert.it/pillole-normative/reato-di-deepfake/

[39] The Guardian. *Denmark drafts law to protect citizens from AI-generated deepfakes*. 2025. Disponibile su: https://www.theguardian.com/technology/2025/jun/27/deepfakes-denmark-copyright-law-artificial-intelligence

[40] U.S. Congress. *TAKE IT DOWN Act – Bill to combat non-consensual deepfake content*. Approvato nel 2025. Disponibile su: https://en.wikipedia.org/wiki/TAKE_IT_DOWN_Act

[41] Parlamento Europeo e Consiglio. *Regolamento (UE) 2016/679 - Regolamento generale sulla protezione dei dati (GDPR)*. Disponibile su: https://eur-lex.europa.eu/legal-content/IT/TXT/?uri=CELEX:32016R0679