ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI INFORMATICA — SCIENZA E INGEGNERIA

Corso di Laurea in Ingegneria e Scienze Informatiche

Progettazione e sviluppo di un Event Display 3D interattivo basato su OpenGL per il rivelatore SND@LHC

Tesi di Laurea in Computer Graphics

Relatrice:
Prof.ssa Damiana Lazzaro

Presentata da:
Enrico Bartocetti

Correlatori:
Prof. Luigi Guiducci
Dott.ssa Giulia Paggi

Sessione Unica
Anno Accademico 2025/2026

A tutta la mia famiglia,
per aver sempre creduto in me.

Abstract

SND@LHC è un esperimento di fisica delle particelle, situato al CERN, dedicato allo studio dei neutrini prodotti al Large Hadron Collider. L'event display attualmente utilizzato dalla collaborazione presenta limiti significativi in termini di interattività e usabilità, fornendo unicamente proiezioni bidimensionali della geometria del rivelatore.

Questa tesi descrive la progettazione e lo sviluppo di un nuovo event display tridimensionale e interattivo, basato su OpenGL, pensato per superare tali limiti. Dopo un'analisi dei requisiti della collaborazione, vengono illustrate le scelte progettuali e implementative del sistema, fondate sui principi della computer grafica tridimensionale e su un'architettura modulare ed estendibile.

Il software sviluppato consente di esplorare la struttura del rivelatore e gli eventi registrati da prospettive arbitrarie, garantendo un'interazione fluida in tempo reale. Il confronto su dati sperimentali reali evidenzia come questo migliori sensibilmente la comprensione visiva degli eventi rispetto al sistema preesistente.

Indice

Elenco delle figure

Elenco dei listati

Introduzione

SND@LHC è un esperimento di fisica delle particelle situato al CERN, dedicato allo studio delle interazioni dei neutrini in un intervallo di energie finora inesplorato. L'analisi di queste particelle offre infatti una nuova chiave di lettura su fenomeni che vanno oltre il Modello Standard, la teoria attualmente alla base della fisica delle particelle.

Tra gli strumenti software utilizzati dalla collaborazione di SND@LHC per la ricostruzione e lo studio degli eventi è presente un event display, un'applicazione grafica che consente di visualizzare i segnali registrati dai sensori all'interno del rivelatore. Il software attuale è però affetto da alcune limitazioni significative: mette innanzitutto a disposizione solo due viste fisse, ovvero le proiezioni bidimensionali laterali e dall'alto del rivelatore, senza consentire alcuna interazione con la geometria. Inoltre, presenta criticità dal punto di vista dell'usabilità: il suo avvio richiede di utilizzare molti argomenti mnemonici da riga di comando, e il passaggio a un evento di una run diversa richiede il riavvio del software. Infine, non prevede alcuna modularità, dato che l'intero software è contenuto in un unico file Python.

Questa tesi si propone quindi di progettare e sviluppare un nuovo event display per la collaborazione, basato su una scena tridimensionale interattiva. Il nuovo sistema dovrà inoltre offrire una buona usabilità e rispettare le pratiche di buona progettazione del software, così da garantirne la manutenibilità e l'estendibilità nel tempo.

La tesi è così organizzata:

- Capitolo 1: descrive l'esperimento SND@LHC nel suo complesso, a partire dal contesto fisico fino alla struttura dettagliata di ogni sottorivelatore.
- Capitolo 2: contiene lo studio del software attuale e l'analisi dei requisiti discussi con la collaborazione.
- Capitolo 3: presenta le basi matematiche di computer grafica necessarie alla costruzione del motore di rendering tridimensionale, così da comprendere nel dettaglio ogni scelta implementativa.
- Capitolo 4: descrive l'architettura del software, basata su un ampio utilizzo di design pattern per ricondurre le principali sfide progettuali a problemi noti con soluzioni consolidate.
- Capitolo 5: applica la teoria e le scelte progettuali nella stesura del software, descritta nel dettaglio e accompagnata da porzioni rilevanti di codice commentato.
- Capitolo 6: mostra, tramite schermate dell'interfaccia utente, le funzionalità implementate nel software, così da illustrare concretamente come il nuovo event display possa migliorare lo studio degli eventi da parte della collaborazione di SND@LHC.

Il lavoro si chiude con un capitolo conclusivo, in cui vengono riassunti i risultati raggiunti, discussi i limiti del sistema sviluppato e proposti possibili sviluppi futuri.

Capitolo 1

L'esperimento SND@LHC

Scattering and Neutrino Detector at the LHC (SND@LHC) è un esperimento il cui obiettivo è lo studio di neutrini ad alta energia prodotti nel Large Hadron Collider. Lo scattering (o diffusione / dispersione) è un fenomeno fisico in cui particelle o onde vengono deflesse e disperse in varie direzioni a causa dell'interazione con altre particelle o ostacoli. SND@LHC è stato approvato nel 2021, costruito, installato e messo in funzione in circa un anno e ha iniziato a raccogliere dati durante la Run 3 dell'LHC, iniziata nel luglio 2022.

1.1 Il CERN e LHC

Il Conseil Européen pour la Recherche Nucléaire (CERN, in italiano Organizzazione Europea per la Ricerca Nucleare) è il laboratorio di fisica delle particelle più grande al mondo, fondato il 29 settembre 1954. Si trova al confine tra la Francia e la Svizzera, più precisamente alla periferia ovest della città di Ginevra, nel comune di Meyrin.

Ospita un ampio complesso di acceleratori di particelle, illustrato in Figura 1.1. Un acceleratore di particelle accelera particelle cariche, come protoni o elettroni, ad altissime velocità prossime a quella della luce: queste particelle vengono poi fatte collidere con un bersaglio o con altre particelle che circolano nella direzione opposta. L'energia liberata si trasforma poi in materia sottoforma di altre particelle (le più massicce delle

quali dominavano l'universo primordiale), seguendo la famosa equazione di Einstein $E = mc^2$ [1].

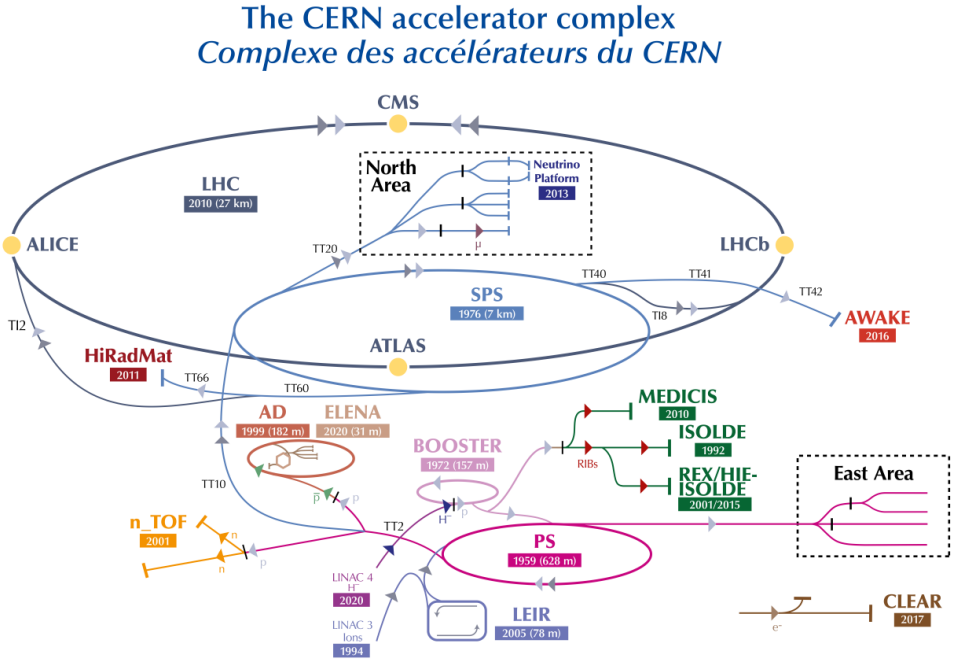


Figura 1.1: Schema del complesso di acceleratori del CERN [2]

Alla data odierna il complesso del CERN è costituito da nove acceleratori (e due deceleratori): ognuno di questi accelera le particelle a energie sempre più elevate per poi passare mano a mano ad acceleratori sempre più grandi. Attualmente il Large Hadron[1] Collider (LHC) è l'acceleratore di particelle più potente e grande del mondo: installato nel 2008, fa collidere fasci di protoni a velocità vicine a quella della luce per studiare le particelle fondamentali. Si trova in un tunnel posto a 100 metri di profondità ed è costituito da un anello lungo 27 chilometri di magneti superconduttori mantenuti a una temperatura di $-271.3\,°\mathrm{C}$ [3].

[1]Gli adroni sono particelle subatomiche composte; i più comunemente conosciuti sono i neutroni e i protoni.

1.2 I Neutrini

I leptoni sono una famiglia di particelle elementari (elencate in Figura 1.2) appartenenti alla classe dei fermioni. Si dividono in tre generazioni, ciascuna composta da una particella carica e dal rispettivo neutrino associato: l'elettrone (e), il muone (μ) e il tau (τ).

Il neutrino è una particella subatomica elementare di massa piccolissima (non ancora precisamente determinata) e carica elettrica nulla che interagisce molto raramente con la materia. L'ipotesi dell'esistenza dei neutrini fu formulata da W. Pauli nel 1930; tuttavia, a causa della loro particolare elusività, furono rilevati per la prima volta solamente nel 1956 in un esperimento condotto da C. Cowan e F. Reines in un reattore nucleare negli Stati Uniti. A questo seguirono scoperte sui neutrini che andarono ad arricchire il panorama della fisica delle particelle, portando alla scoperta di tre famiglie – dette sapori – di neutrini: elettronico, muonico e tau, ciascuno associato univocamente al rispettivo leptone carico e indicato con i simboli ν_e, ν_μ e ν_τ.

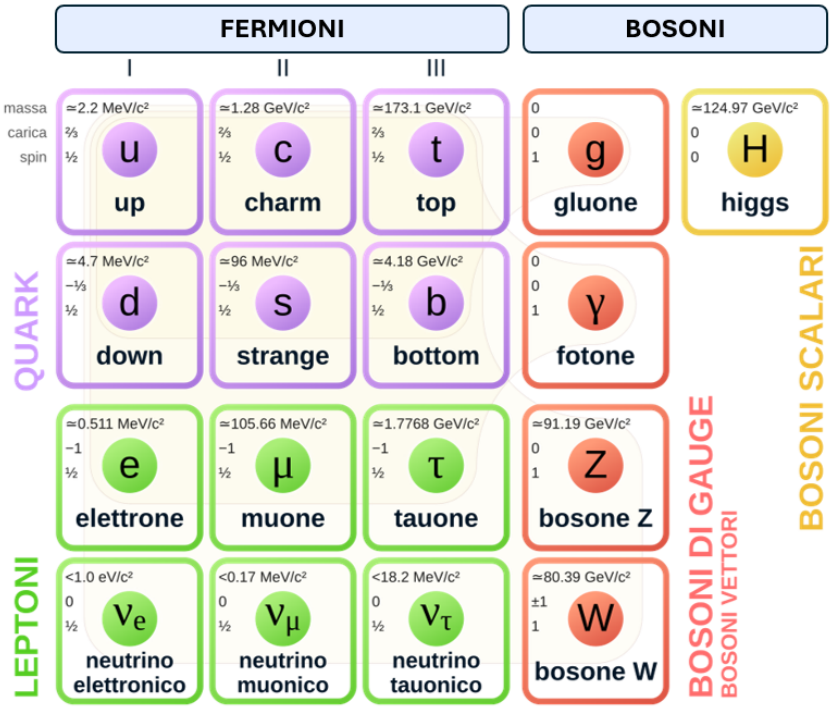


Figura 1.2: Particelle elementari del Modello Standard [4]

I tre tipi di neutrino sono soggetti ad un fenomeno chiamato oscillazioni di sapore per cui, in certe condizioni, si possono trasformare l'uno nell'altro: l'esistenza di questo fenomeno implica che i neutrini debbano avere una massa, seppur piccolissima, diversa tra loro. Questo risultato è in disaccordo con il Modello Standard che, al contrario, prevede che i neutrini abbiano massa nulla [5].

L'esistenza di queste particelle rappresenta quindi una prova dell'esistenza di una fisica che va oltre il Modello Standard [6], e analizzandone le caratteristiche potremo avere un nuovo punto di vista per studiare l'universo [7].

1.3 Obiettivi

Negli scorsi decenni le misure della sezione d'urto dei neutrini, cioè la loro probabilità di interazione, sono state per lo più effettuate a basse energie (al di sotto di 350 GeV[2]). La regione tra i 350 GeV e i 10 TeV è rimasta sconosciuta fino a qualche anno fa [8]: i neutrini prodotti ad LHC consentono di osservarne le interazioni in questo range. L'alta intensità delle collisioni protone-protone si traduce in un abbondante flusso di neutrini, e le alte energie dei neutrini prodotti implicano sezioni d'urto relativamente grandi [9].

SND@LHC è posizionato leggermente fuori asse rispetto alla linea di collisione dei fasci di protoni di LHC, facendo sì che l'intervallo di pseudorapidità[3] dei neutrini studiati cada in una regione precedentemente inesplorata [10]. Per questa caratteristica SND si differenzia dall'esperimento ForwArd Search ExpeRiment (FASER), che è invece posizionato sulla traiettoria del fascio.

[2]L'elettronvolt, indicato con il simbolo eV, è un'unità di misura del lavoro e dell'energia utilizzata principalmente in fisica atomica, nucleare e delle particelle.

[3]In fisica delle particelle, la pseudorapidità, indicata con il simbolo η, è una coordinata spaziale usata per descrivere l'angolo relativo tra una particella e l'asse del fascio.

1.4 Ubicazione dell'esperimento

In Figura 1.3 è riportata in maniera semplificata l'ubicazione dell'esperimento. TI18 è un tunnel lungo 280m che si innesta nell'anello di LHC tramite il punto di giunzione UJ18, a circa 480 m dal punto di impatto di ATLAS[4], chiamato Interaction Point 1 (IP1). Era stato inizialmente costruito per l'iniezione di positroni[5] dal Super Proton Synchrotron (SPS) all'acceleratore Large Electron-Positron Collider (LEP), quest'ultimo rimosso alla fine del 2000 per cedere il posto ad LHC, ponendo così in disuso TI18.

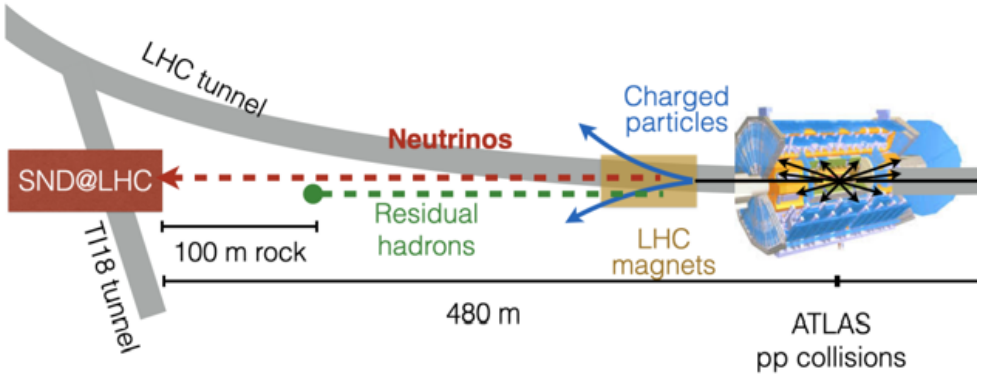


Figura 1.3: Posizione del rivelatore SND@LHC: a sinistra il tunnel TI18 in cui è collocato SND, a destra il punto di collisione IP1 di ATLAS [11]

Il tunnel TI18 è stato successivamente chiuso, ad eccezione di una piccola sezione di circa 20m prima di entrare nella giunzione UJ18 con LHC. Questa collocazione risulta particolarmente adeguata dal momento che il breve tratto scelto intercetta l'asse di collisione di IP1 (schematizzato in Figura 1.4). L'esperimento è posizionato leggermente fuori asse, consentendo così di studiare neutrini prodotti in un intervallo di pseudorapidità $7.2 < \eta < 8.4$.

[4] A Toroidal LHC ApparatuS (ATLAS) è uno dei nove rivelatori di particelle costruiti per LHC installato nel 2008 (vedi posizione in Figura 1.1).

[5] Il positrone, chiamato anche antielettrone, è l'antiparticella dell'elettrone: rispetto a quest'ultimo, ha la stessa massa ma carica elettrica uguale e opposta.

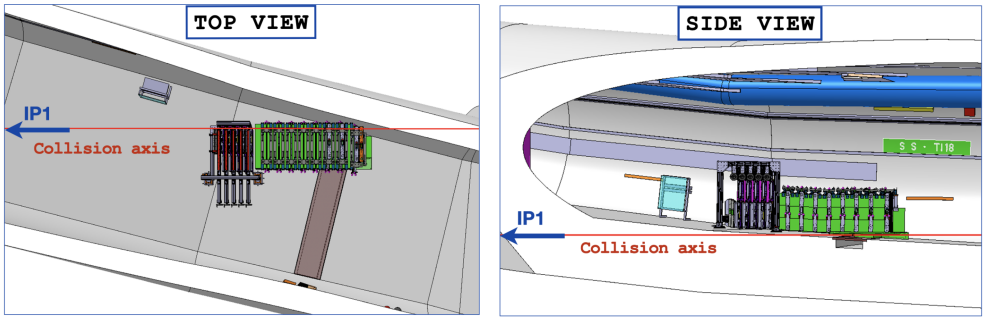


Figura 1.4: Vista laterale e dall'alto del rivelatore nel tunnel TI18 [12]

1.5 Struttura del rivelatore

La struttura di SND@LHC (schematizzata in Figura 1.5) è stata studiata per far sì che possano essere identificati tutti e tre i sapori di neutrino sfruttando lo spazio a disposizione nel tunnel TI18. Grazie all'esperimento OPERA[6] è stato dimostrato che la soluzione ottimale è data dalla combinazione di emulsioni nucleari e rivelatori elettronici [13].

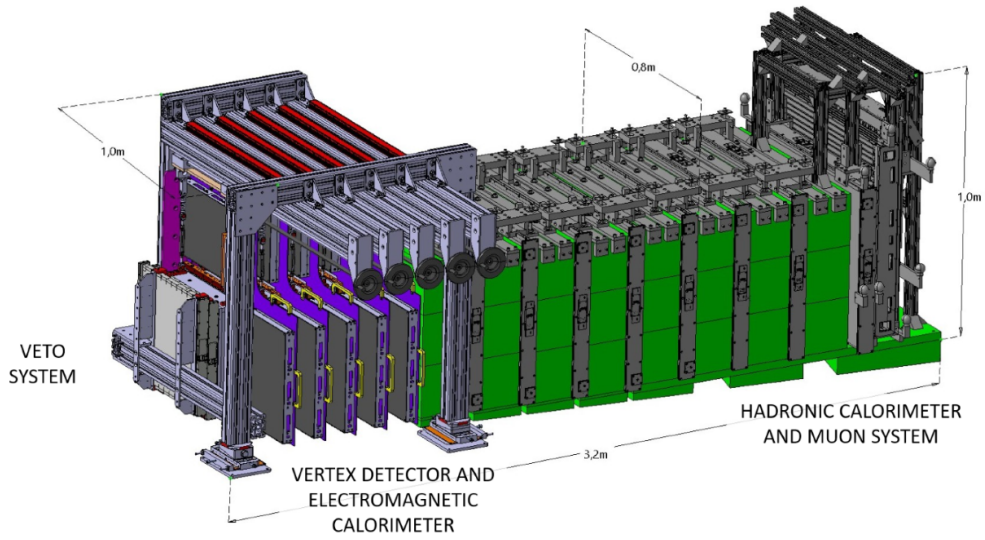


Figura 1.5: Struttura dell'esperimento SND@LHC [14]

[6]Oscillation Project with Emulsion-tRacking Apparatus (OPERA) è stato un esperimento (2008 - 2012) posizionato ai Laboratori Nazionali del Gran Sasso per lo studio dell'oscillazione di sapore dei neutrini muonici in neutrini tau.

1.5.1 Principi di progettazione

La tipica interazione di neutrino che è possibile identificare nel rivelatore avviene con un nucleo del bersaglio: quest'ultimo è costituito per la maggior parte di un materiale denso, il tungsteno, per aumentare la probabilità che l'interazione avvenga. Una delle particelle costituenti il nucleo atomico riceve una grossa quantità di impulso nella interazione con il neutrino e viene proiettata fuori dal nucleo, producendo uno sciame di particelle adroniche, che può depositare una quantità significativa di energia negli elementi attivi del rivelatore. Il neutrino invece, nel caso di interazioni cosiddette a corrente carica, si trasforma nel corrispondente leptone carico (quindi un elettrone, un muone o un leptone tau a seconda del sapore del neutrino interagente).

Discriminazione del sapore

Elettroni e muoni di alta energia hanno un comportamento molto diverso quando attraversano un materiale denso: il muone interagisce poco e riesce ad attraversare, relativamente indisturbato, grossi spessori di materiale, mentre l'elettrone interagisce immediatamente, producendo un conseguente sciame elettromagnetico costituito da elettroni e positroni. Il leptone tau ha una brevissima vita media e decade immediatamente, per esempio in un muone e due neutrini, oppure in adroni. Mediamente il tau percorre, prima del suo decadimento, una distanza misurabile (ad esempio qualche millimetro alle energie di LHC) con un rivelatore di tracce di risoluzione sufficientemente elevata.

In conclusione tutti gli eventi di interazione sono accomunati dalla presenza di un significativo sciame di particelle adroniche. Il sapore del neutrino interagente può essere determinato dalle altre particelle presenti nello stato finale (muone o sciame elettromagnetico) e dalla ricostruzione precisa delle tracce delle particelle prodotte nel vertice di interazione (nel caso del leptone tau).

Stato attuale

Attualmente il rivelatore è composto dai seguenti sistemi:

- Sistema di veto: rileva particelle cariche (principalmente muoni di LHC) che entrano frontalmente, permettendo di discriminare i prodotti delle interazioni neutre.
- Volume bersaglio: composto da pareti di emulsioni nucleari, che fungono da rivelatore di vertice, alternate a piani di tracciatori elettronici a fibre scintillanti (Scintillating Fibres, SciFi), che forniscono la marcatura temporale e la misura dell'energia agli eventi ricostruiti nelle emulsioni. La maggior parte della sua massa è costituita da fogli di tungsteno: essendo molto denso massimizza la probabilità di interazione dei neutrini. Nella vista frontale di Figura 1.6 viene messa in evidenza la regione di pseudorapidità $7.2 < \eta < 8.4$.
- Sistema di identificazione dei muoni: situato a valle del bersaglio, identifica i muoni e agisce come calorimetro adronico[7] a campionamento per la misura dell'energia degli sciami.

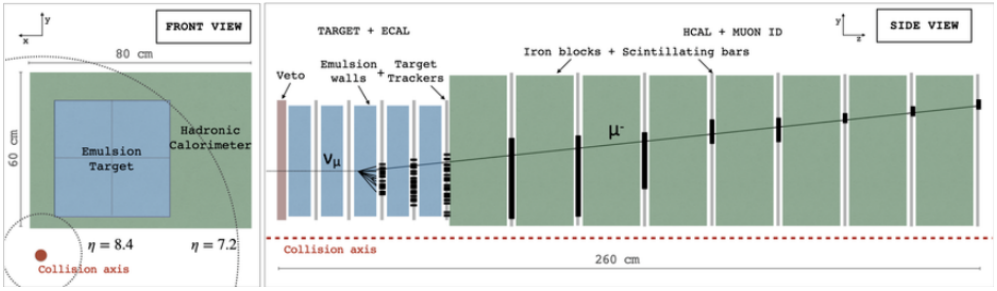


Figura 1.6: Rappresentazione schematica della vista frontale (sinistra) e laterale (destra) del rivelatore [12]

1.5.2 Veto

Il sistema di veto è necessario per riconoscere le particelle cariche rilevate da SND@LHC provenienti frontalmente dall'esterno. Questo filtro

[7] Un calorimetro adronico (Hadronic Calorimeter, HCAL) è uno strumento progettato per misurare l'energia degli adroni.

è indispensabile poiché si vogliono studiare gli eventi in cui vengono prodotte delle particelle cariche a partire dall'interazione di particelle neutre (ad esempio un neutrino muonico ν_μ che urtando un nucleo di tungsteno si trasforma in un muone μ^-): se entra una particella già carica, l'evento non è rilevante ai fini dello studio.

Alla base del suo funzionamento vi sono le barre scintillanti, ovvero delle barre di un particolare materiale plastico che emette piccole quantità di luce quando è attraversato da particelle cariche: la luce emessa è poi misurata da fotosensori a silicio (Silicon Photomultiplier, SiPM) posti alle estremità delle barre. Il sistema è composto da due piani paralleli di barre, ognuna delle quali ha dimensioni $42 \times 6 \times 1\,\mathrm{cm}^3$ ed è avvolta in un foglio di alluminio per assicurarne l'opacità e isolarla dalla luce proveniente da barre adiacenti. Ciascun piano comprende sette barre scintillanti racchiuse in una struttura di alluminio con pareti dello spessore di 4 mm. Ogni estremità di ciascuna barra è letta da otto SiPM, che convertono i segnali luminosi in impulsi elettrici. Lo scostamento verticale di 2 cm tra i due piani, consente di ottenere la massima copertura del bersaglio nonostante la presenza di spazi tra le barre indotti dalle variazioni delle loro altezze ($\sim 250\,\mu$m) e dai fogli di alluminio ($\sim 60\,\mu$m) [12]. A partire dal 2024, al fine di migliorare la capacità di rigettare eventi in cui c'è una particella carica in ingresso, è stato installato un altro piano di barre scintillanti, orientate verticalmente per incrementarne la segmentazione.

1.5.3 Bersaglio e rivelatore di vertice

Il bersaglio di SND@LHC è la sezione del rivelatore dove avvengono effettivamente le interazioni dei neutrini: è stato progettato in maniera tale da essere allo stesso tempo sia una massa in cui far avvenire le interazioni, sia un rivelatore ad altissima precisione. Il vertice è infatti il punto esatto in cui il neutrino interagisce con un nucleo (nel nostro caso tungsteno), trasformandosi poi in altre particelle: è necessaria una risoluzione spaziale estrema per identificare questo punto.

Strutturalmente, è composto da cinque muri (wall) di sezione trasversale $384 \times 384\,\mathrm{mm}^2$ a loro volta divisi in quattro celle elementari dette

mattoni (brick). Ogni mattone è costruito sulla base della tecnologia Emulsion Cloud Chambers (ECC): 60 film di emulsioni nucleari (ognuno spesso 300 μm e di sezione trasversale $192 \times 192\,\mathrm{mm}^2$) usati come dispositivi di tracciamento sono intervallati a 59 lastre di tungsteno spesse 1 mm che fungono da bersaglio per i neutrini. Il mattone risultante (rappresentato in Figura 1.7) ha uno spessore totale di circa 78 mm e una massa di 41.5 kg, facendo sì che la massa totale del bersaglio composto dai cinque muri di 2×2 mattoni sia di 830 kg. A ogni muro del bersaglio è intervallato un piano SciFi [12].

La regione del bersaglio è racchiusa all'interno di un contenitore chiamato cold box, un elemento puramente passivo che non raccoglie dati ma che ricopre una duplice funzione. Mantiene il bersaglio a temperatura ed umidità controllate, facendo sì che SciFi non raggiunga temperature pericolose e le emulsioni non assorbano umidità causandone la dilatazione. Inoltre, funge da schermo contro i neutroni poiché è composto da polietilene borato: il boro rallenta e blocca i neutroni, particelle neutre con interazioni simili a quelle dei neutrini.

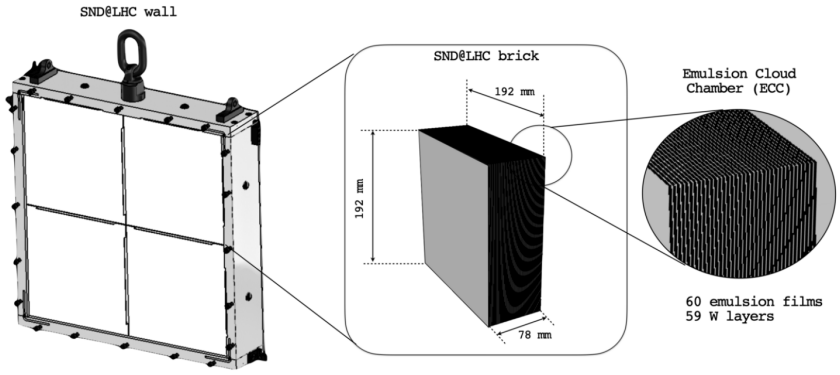


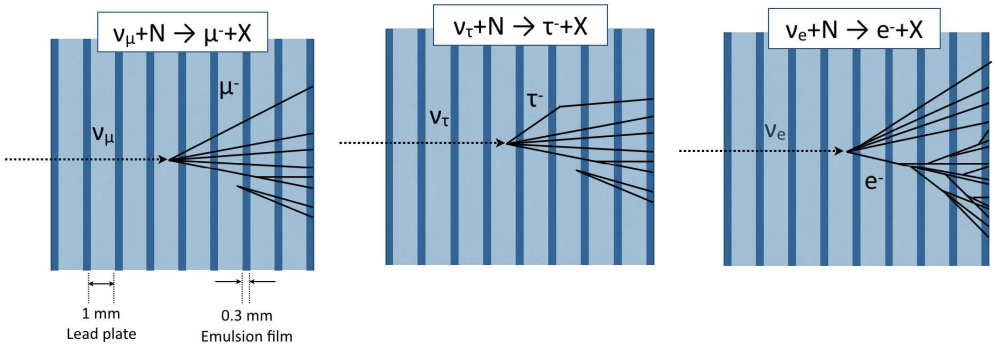
Figura 1.7: Una parete a emulsione è composta da quattro mattoni, ciascuno costituito da 60 pellicole di emulsioni intervallate da 59 lastre di tungsteno [12]

Emulsioni

I film di emulsioni nucleari sono il rivelatore tridimensionale più leggero e compatto che garantisce al contempo una risoluzione spaziale submicrometrica e angolare al milliradiante. Durante lo sviluppo dei film in camera oscura è utilizzato un processo chimico che permette di ottenere dei cluster dal diametro di $0.6\,\mu$m della traccia lasciata dalle particelle cariche che li hanno attraversati.

Un film è composto da due strati dallo spessore di $70\,\mu$m separati da una base plastica trasparente di $170\,\mu$m. Nel rivelatore sono impiegati 1200 film di emulsioni, per un totale di $44\,\mathrm{m}^2$ analizzati da microscopi ottici completamente automatizzati.

Collegando i punti rappresentanti il passaggio di una particella carica su entrambi gli strati del film può essere misurata la direzione di ogni traccia, oltre alla sua posizione, consentendo la completa ricostruzione di tutte le particelle cariche prodotte nell'interazione del neutrino. È così possibile identificare il sapore del neutrino interagente, poiché interazioni di sapore diverso producono topologie distinte, come mostrato schematicamente in Figura 1.8.



(a) Neutrino muonico (b) Neutrino tau (c) Neutrino elettronico

Figura 1.8: Schematizzazione dello scattering dei tre sapori di neutrino nelle pareti di emulsioni [15]

Piani SciFi

I cinque piani di fibre scintillanti, alternati agli altrettanti di muri del bersaglio, costituiscono il sistema di tracciamento elettronico di SND@LHC. Ogni piano è costituito da 6 strati di fibre (foto in Figura 1.9) dal diametro di 250 μm, offrendo una risoluzione spaziale di $\sim$ 50 μm e temporale di $\sim$ 250 ps.

Poiché le emulsioni nucleari rimangono a raccogliere dati per alcune settimane, tramite le rilevazioni elettroniche in SciFi è possibile assegnare un tempo preciso alle interazioni di neutrino impresse nei film. Inoltre, permettono anche di misurare l'energia degli sciami elettromagnetici e adronici prodotti in tali interazioni.

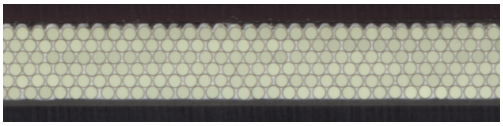


Figura 1.9: Foto degli strati di fibre che compongono un piano SciFi [12]

1.5.4 Sistema di identificazione dei muoni

A valle del bersaglio di SND@LHC è presente il sistema di rilevazione di muoni, visibile in Figura 1.10. Il suo scopo principale è quello di identificare i muoni e, insieme alle SciFi, costituisce anche un calorimetro adronico per la misura dell'energia degli sciami adronici.

È composto da 8 strati di piani scintillanti, alternati a blocchi di ferro spessi 20 cm che fungono da materiale passivo. Se una particella attraversa tutti gli otto piani è quasi sicuramente identificata come un muone: interagendo poco con la materia, essi possono penetrare quasi indisturbati gli strati del rivelatore.

Il sistema è diviso in due regioni: i primi cinque piani a monte (upstream, US) e gli ultimi tre a valle (downstream, DS) [12]. Per migliorare ulteriormente le capacità di tracciamento dei muoni, a partire dal 2025 è stato aggiunto un nuovo sottorivelatore, composto da due stazioni di Mini Drift Tubes (MiniDT).

Figura 1.10: Foto del rivelatore, in cui vengono messi in evidenza le due regioni US e DS del sistema di identificazione dei muoni e la cold box del bersaglio

Upstream – US

I 5 piani di US sono simili ai piani del sistema di veto, ad eccezione delle dimensioni. Ogni piano è formato da 10 barre scintillanti sovrapposte, ognuna di dimensioni $1 \times 6 \times 83\,\mathrm{cm}^3$, ricomprendo un'area di $81 \times 60\,\mathrm{cm}^2$. Così come accade per quelle del sistema di veto, ogni barra è avvolta in un foglio di alluminio e ognuna delle sue estremità è letta da otto SiPM.

Downstream – DS

Gli ultimi tre piani completano l'identificazione dei muoni: essendo caratterizzati da un'alta granularità, la regione DS consente di determinarne la posizione e direzione con una risoluzione migliore. Ogni piano è formato da due strati di barre scintillanti: il primo strato è composto da 60 barre orizzontali da $1 \times 1 \times 82.5\,\mathrm{cm}^3$, il secondo da 60 barre verticali da $1 \times 1 \times 63.5\,\mathrm{cm}^3$.

Ogni barra orizzontale è letta da un SiPM a ogni estremità; quelle verticali sono lette da un solo SiPM all'estremità superiore. Anche in questo

caso le barre sono avvolte strettamente da fogli di alluminio (visibili in Figura 1.11) per evitare che la luce passi da una barra all'altra e minimizzarne la perdita a causa di riflessioni multiple. Per le barre verticali, all'estremità inferiore non collegata ad alcun SiPM, vi è un altro strato piatto che ottimizza la riflessione.

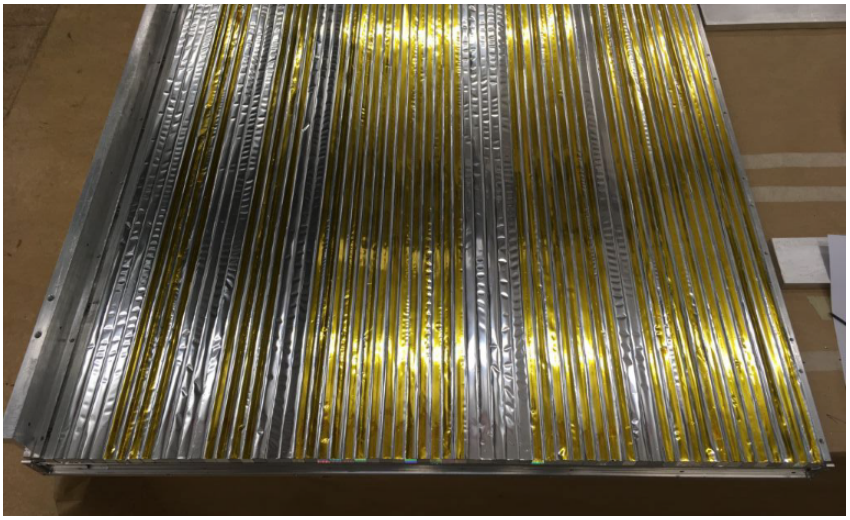


Figura 1.11: Foto scattata durante il processo di produzione di un piano di barre scintillanti di DS [12]

Mini Drift Tubes - MiniDT

I due moduli aggiunti al sistema di identificazione dei muoni sono chiamati MiniDT poiché sono repliche in scala ridotta delle camere Drift Tube (DT) dell'esperimento CMS[8] [16]. Ogni cella è percorsa da un filo anodico centrale lungo $\sim 60\,\text{cm}$ carico positivamente: le particelle cariche che le attraversano ionizzano il gas presente all'interno, con la conseguente deriva degli elettroni verso l'anodo. Misurando questo tempo e conoscendo la velocità di deriva degli elettroni nel gas utilizzato, è possibile determinare con precisione la coordinata della traccia.

Ogni MiniDT è costituita da quattro strati di tubi paralleli, con 16 celle per strato, per un totale di 64 tubi. Vengono utilizzate due stazioni

[8]Compact Muon Solenoid (CMS) è uno dei nove rivelatori di particelle costruiti per LHC installato nel 2008 (vedi posizione in Figura 1.1).

di MiniDT orientate in modo ortogonale tra loro, così da ottenere una ricostruzione spaziale completa su entrambe le componenti del piano XY. Grazie alla loro risoluzione di 250 μm consentono di raddoppiare la precisione nella ricostruzione delle tracce all'indietro delle particelle all'interno del bersaglio, il cui processo è schematizzato in Figura 1.12.

Essendo state installate di recente, le MiniDT sono ancora in fase di integrazione nel software dell'esperimento: benché presenti nei file di geometria, i dati da esse raccolti non sono ancora pronti per essere visualizzati.

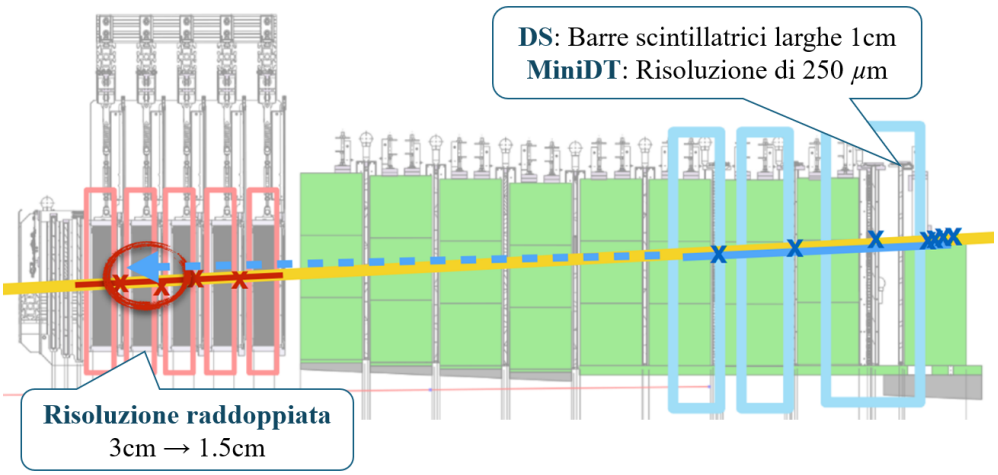


Figura 1.12: Funzionamento delle MiniDT [17]

In sintesi, la complessa architettura di SND@LHC – comprendente i sistemi di veto, il volume bersaglio a emulsioni nucleari con i piani SciFi e il sistema di identificazione dei muoni (US e DS) – è ingegnerizzata per fornire una precisione geometrica e temporale adeguata alla rivelazione dei neutrini di LHC. Al fine di sfruttare appieno le potenzialità del rivelatore, si rende necessario uno strumento che permetta di analizzare complessivamente i dati raccolti da ogni sotto-rivelatore in modo intuitivo per l'utente.

Nel prossimo capitolo verrà presentato l'event display utilizzato finora dal gruppo di ricerca di SND, così da delineare i requisiti e le necessità sui quali è stata fondata la progettazione del nuovo software.

Capitolo 2

Software esistente e requisiti del nuovo sistema

Una volta compresa la struttura del rivelatore SND@LHC, è necessario analizzare gli strumenti software attualmente a disposizione della collaborazione per la visualizzazione degli eventi. L'analisi sarà quindi focalizzata sull'ambiente software effettivamente utilizzato in SND@LHC, a partire dal framework su cui esso si basa.

ROOT è un framework software open-source per il calcolo scientifico, sviluppato originariamente dal CERN a partire dal 1995, scritto principalmente nel linguaggio C++. Standard di fatto nella fisica delle alte energie, ROOT fornisce le librerie essenziali per l'analisi dei dati, la gestione di grandi set di informazioni e la simulazione statistica di eventi. Il nome, che tradotto dall'inglese significa letteralmente radice, è una metafora utilizzata per racchiudere lo scopo del framework. ROOT deve fungere da base per qualsiasi applicazione venga sviluppata, come sottolineò René Brun – uno dei fondatori del progetto – all'interno della mailing list ufficiale nel 1996:

> *"[...] in January 1995, I came quickly to the name ROOT for our system. [...] ROOT really means the 'roots' for end-user applications. Once we have the roots well installed in the ground, Trees or other Flowers can flourish. We hope that we are reaching this stage now."*[18]

ROOT mette inoltre a disposizione strumenti grafici per il rendering di oggetti 2D e 3D. In questo capitolo verranno tuttavia approfonditi i motivi per cui tali strumenti non risultano adeguati al caso in esame, delineando al contempo i requisiti che costituiranno la base per la progettazione del nuovo software.

2.1 Il framework sndsw

Per la gestione dell'intero ciclo di vita dei dati prodotti nell'esperimento SND@LHC, la collaborazione ha sviluppato lo *Scattering and Neutrino Detector Software* (sndsw). Scritto principalmente nei linguaggi di programmazione C++ e Python e configurato tramite CMake, è basato a sua volta sui framework FairRoot (inizialmente sviluppato per l'esperimento tedesco Facility for Antiproton and Ion Research, FAIR, costruito interamente su ROOT) e FairShip (il framework dell'esperimento Search for Hidden Particles – SHiP – anch'esso basato su FairRoot).

Sndsw include un event display bidimensionale, strumento che presenta tuttavia alcune limitazioni, che hanno reso necessario lo sviluppo di un nuovo event display capace di rappresentare in modo più efficace la complessità geometrica e strutturale del rivelatore SND@LHC.

2.1.1 Organizzazione dei dati

Per interagire con i dati di sndsw, il nuovo software deve conoscerne l'organizzazione. Una *Run* indica un intervallo di tempo durante il quale il sistema di acquisizione dati è attivo; la fine della registrazione corrisponde alla chiusura della run. Ogni run ha come identificatore univoco un numero intero progressivo, il *run number*. L'*Evento* è l'unità minima di dato registrato, corrispondente a una singola interazione (o collisione) all'interno del rivelatore. Ogni evento all'interno di una run è contrassegnato da un numero progressivo, l'*event number*. Per identificare univocamente un evento in un database o in un archivio, si utilizza sempre la coppia *(run number, event number)*.

I file contenenti i dati delle collisioni sono archiviati all'interno di EOS Open Storage (EOS), una soluzione software open-source per lo storage distribuito. Sviluppato dal CERN a partire dal 2010 per gestire gli enormi volumi di dati scientifici prodotti da LHC (nell'ordine degli exabyte), EOS consente l'accesso simultaneo e scalabile ai file di analisi da parte dell'intera collaborazione.

L'organizzazione dei file è variata nel corso del tempo; in ogni caso, i dati vengono sostanzialmente divisi per anno, poi per numero di run e infine raggruppati in partizioni, ovvero file che accorpano 1 milione di eventi ciascuno così da limitarne la dimensione.

2.1.2 Event display

Un *event display* – letteralmente "visualizzatore di eventi" – è un software di visualizzazione utilizzato principalmente nell'ambito della fisica delle particelle: il suo scopo è rappresentare graficamente le complesse collisioni tra particelle (gli "eventi") registrate dai rivelatori, fornendo ai ricercatori una visualizzazione immediata di ciò che è avvenuto al loro interno durante le collisioni. In linea di massima, ogni esperimento di fisica delle particelle sviluppa il proprio software event display, adattato alla geometria e alle esigenze specifiche del rivelatore. In Figura 2.1 è riportato un esempio di output di quello sviluppato per il rivelatore CMS.

L'event display 2D di SND@LHC

L'event display sviluppato dalla collaborazione e utilizzato regolarmente nell'analisi degli eventi di SND@LHC è interamente contenuto in un singolo script Python di circa 1300 righe: l'assenza di una struttura modulare ne limita significativamente la manutenibilità e l'estensibilità. Per visualizzare un'interazione, l'utente deve fornire all'eseguibile, come argomenti, sia il percorso della partizione contenente i dati da rappresentare che il file di geometria da caricare. La selezione dell'evento avviene poi tramite la shell interattiva di Python. Ad esempio, per aprire l'evento numero `112104442` della run `5262`, è necessario utilizzare i comandi riportati nel Listato 2.1; il risultato è mostrato in Figura 2.2. Ne consegue

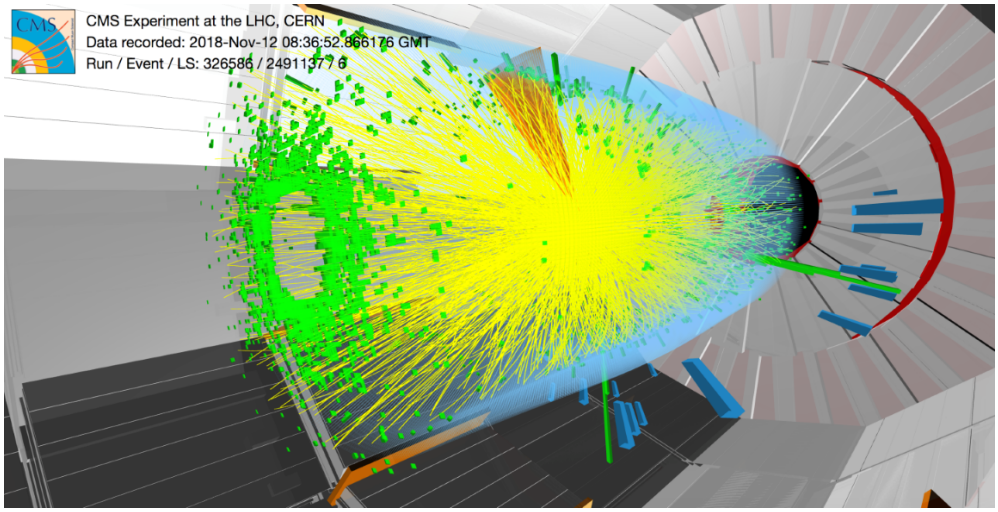


Figura 2.1: Output di un event display di CMS, in cui è raffigurato un candidato bosone Z (una delle particelle elementari in Figura 1.2) decaduto in due elettroni [19]

che l'utente debba conoscere l'intricato sistema di archiviazione dei dati di sndsw descritto nella sezione precedente, e rende necessario riavviare l'applicativo nel caso si volesse caricare un evento non contenuto nello stesso file di quello già in memoria.

Listato 2.1: Comandi da terminale per aprire il display dell'evento numero `112104442` della run `5262`

```
# Lancio dell'event display
python \
-i $SNDSW_ROOT/shipLHC/scripts/2dEventDisplay.py \
-p root://eospublic.cern.ch//eos/experiment/sndlhc/\
convertedData/physics/2022/run_005262/ \
-f sndsw_raw-0112.root \
-g /eos/experiment/sndlhc/convertedData/physics/2022/\
geofile_sndlhc_TI18_V4_2022.root

# Selezione evento
loopEvents(104442, hitColour='q')
```

Una volta avviato il programma, inoltre, l'interazione con l'applicativo risulta limitata a poche operazioni. Anche funzionalità basilari, come lo zoom, richiedono l'inserimento di ulteriori comandi nella shell interattiva di Python nei quali è necessario specificare manualmente l'area da visualizzare. Tutto ciò compromette significativamente l'usabilità del software.

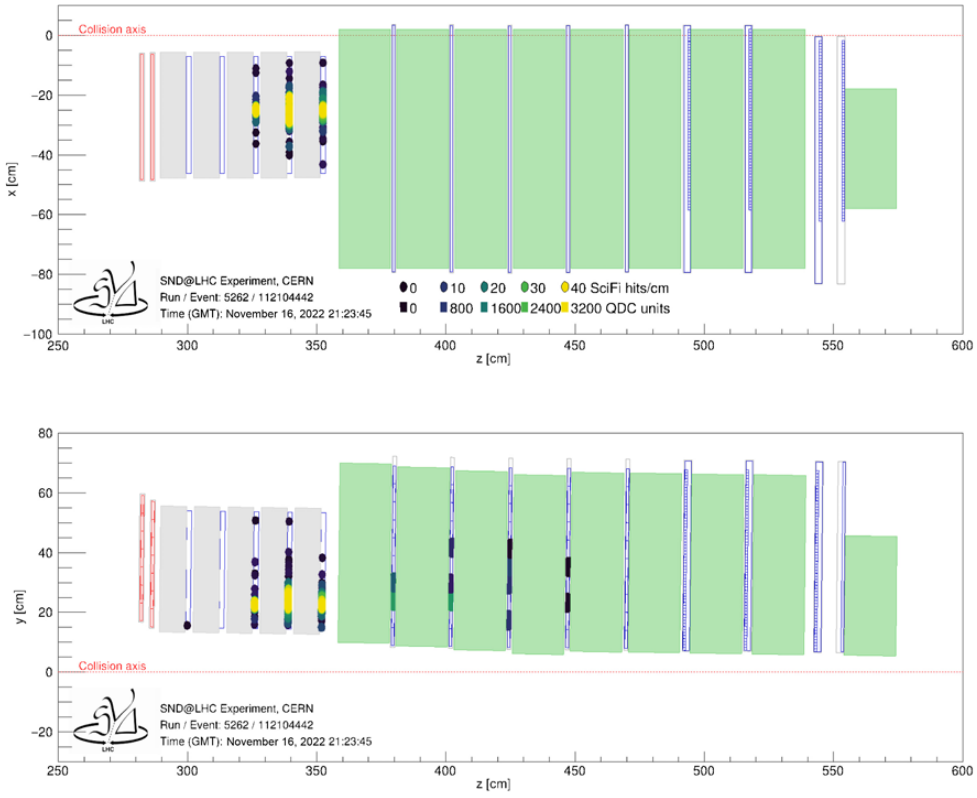


Figura 2.2: Output dell'event display 2D per l'evento `112104442` della run `5262` [20]

2.2 Analisi

Nel Capitolo 1 abbiamo affrontato la complessità della struttura del rivelatore SND@LHC: per sfruttarne al massimo le potenzialità e per agevolare lo studio dei ricercatori, è necessario che la collaborazione abbia a disposizione uno strumento che si adatti perfettamente alle proprie esigenze.

2.2.1 Requisiti

Per comprendere appieno le necessità della collaborazione, è stata affrontata una prima fase di analisi con i membri del gruppo SND@LHC dell'Istituto Nazionale di Fisica Nucleare (INFN) della sezione di Bologna. Dall'incontro è emersa una serie di requisiti, successivamente catalogati in funzionali e non funzionali.

Requisiti funzionali

Il software dovrà consentire all'utente di:

- selezionare tramite interfaccia il numero di run e di evento da visualizzare;
- interagire con la scena, controllandone traslazione, rotazione e zoom, così da osservare la geometria del rivelatore e le traiettorie delle particelle da prospettive arbitrarie;
- visualizzare l'energia depositata nel rivelatore dalle particelle, in modo coerente con i dati letti tramite il framework ROOT;
- selezionare le possibili configurazioni dei dati da caricare.

Requisiti non funzionali

Il software dovrà inoltre:

- garantire un'interazione reattiva, senza ritardi percettibili durante la manipolazione della scena;
- offrire un'interfaccia ridimensionabile;
- rendere agile il passaggio da un evento all'altro;
- fornire un'interazione intuitiva, senza richiedere l'utilizzo di comandi mnemonici.

2.2.2 Scelta delle API grafiche

ROOT mette a disposizione un framework per l'event display chiamato `TEve`, il cui modulo grafico `TGL` si basa su OpenGL 1.x. Tale tecnologia risulta però inadeguata per lo sviluppo di un software moderno: prevede una pipeline grafica fissa – ovvero non programmabile tramite shader – che non permette di agire sul calcolo del colore in modo arbitrario (ad esempio per trasparenze ed illuminazione), e fa largo uso della *immediate mode*, in cui i dati geometrici vengono inviati alla GPU ad ogni frame anziché essere memorizzati in buffer dedicati, con conseguenti forti limitazioni in termini di prestazioni. Gli sviluppatori di ROOT, riconosciute le limitazioni di TEve, hanno sviluppato REve, il suo successore moderno basato su un'architettura client-server con rendering via browser [21].

Per il progetto del nuovo event display, la scelta è ricaduta su OpenGL 3.3: oltre ad avere la pipeline programmabile per sviluppare shader su misura per la visualizzazione degli eventi, è perfettamente integrabile in un progetto in linguaggio C++, permettendo di affrontare la progettazione tramite il paradigma ad oggetti. A differenza di Vulkan[1], infatti, OpenGL 3.3 consente di sfruttare una pipeline programmabile senza introdurre una complessità progettuale eccessiva; rispetto a soluzioni come WebGPU, inoltre, si presta meglio allo sviluppo di un'applicazione locale integrata con sndsw.

Lo sviluppo di un motore di rendering che soddisfi i requisiti precedentemente discussi richiede una serie di concetti geometrici e algebrici propri della computer grafica, che verranno introdotti nel capitolo seguente.

[1]Vulkan è la nuova API per la grafica 2D e 3D sviluppata dal Khronos Group, lo stesso consorzio che gestisce OpenGL, di cui è considerato il diretto successore. Promette prestazioni migliori di OpenGL al costo di una maggior complessità di programmazione.

Capitolo 3

Fondamenti di computer grafica

Per poter comprendere appieno il funzionamento della pipeline grafica, è prima necessario ripassare una serie di concetti base di algebra lineare. Di questi verranno presentati i punti chiave più rilevanti ai fini del progetto, che saranno poi utilizzati per approfondire le basi geometriche della computer grafica tridimensionale.

3.1 Richiami di algebra lineare

Si introducono innanzitutto le tre entità fondamentali necessarie per definire i concetti e le operazioni che verranno utilizzati nel seguito: scalari, punti e vettori.

Uno *scalare* è un numero utilizzato per descrivere una grandezza.

Un *punto* P identifica una posizione nello spazio rispetto a un sistema di riferimento.

Un *vettore* $\mathbf{v}$ è un'entità caratterizzata da lunghezza (numero reale non negativo), direzione (individuata da una retta) e verso, indipendente dalla posizione nello spazio; in $\mathbb{R}^n$ può essere rappresentato tramite una

n-upla di componenti:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

3.1.1 Spazi vettoriali e affini

A partire dalle entità appena introdotte è possibile definire strutture algebriche più complesse, utili per formalizzare le operazioni geometriche che verranno impiegate nella pipeline grafica.

Definizione 3.1 (Spazio vettoriale)**.**
Si definisce *spazio vettoriale* V una struttura algebrica formata da un insieme di vettori su cui sono definite l'addizione tra vettori e la moltiplicazione per uno scalare.

Poiché verranno utilizzati i vettori formati da numeri reali, introduciamo lo spazio $\mathbb{R}^n$ e un modo di rappresentare i vettori dello spazio.

Definizione 3.2 ($\mathbb{R}^n$)**.**
Definiamo $\mathbb{R}^n$ come l'insieme dei vettori a n componenti reali. $\mathbb{R}^n$ è uno spazio vettoriale su $\mathbb{R}$ su cui sono definite le operazioni di addizione tra due vettori $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ e moltiplicazione di un vettore $\mathbf{v} \in \mathbb{R}^n$ per uno scalare $\lambda \in \mathbb{R}$.

$$\mathbf{u} + \mathbf{v} = \begin{bmatrix} u_1 + v_1 \\ \vdots \\ u_n + v_n \end{bmatrix}, \qquad \lambda\mathbf{v} = \begin{bmatrix} \lambda v_1 \\ \vdots \\ \lambda v_n \end{bmatrix}$$

Definizione 3.3 (Combinazione lineare)**.**
Dati k vettori $\mathbf{v}_1, \ldots, \mathbf{v}_k \in \mathbb{R}^n$ e k scalari $\lambda_1, \ldots, \lambda_k \in \mathbb{R}$, si definisce *combinazione lineare* la quantità:

$$\sum_{i=1}^{k} \lambda_i \mathbf{v}_i = \lambda_1 \mathbf{v}_1 + \cdots + \lambda_k \mathbf{v}_k$$

Definizione 3.4 (Base)**.**
Dato uno spazio vettoriale V, un suo sottoinsieme $\mathcal{B} \subseteq V$ di n vettori

$\mathcal{B} = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ è detto *base* di V se è linearmente indipendente e genera V, ovvero se qualsiasi $\mathbf{v} \in V$ è esprimibile come combinazione lineare:

$$\mathbf{v} = \alpha_1 \mathbf{v}_1 + \cdots + \alpha_n \mathbf{v}_n$$

dove i coefficienti $\alpha_1, \ldots, \alpha_n \in \mathbb{R}$ sono unici e sono detti *coordinate* di $\mathbf{v}$ rispetto alla base scelta. Il numero n di elementi della base è detto *dimensione* di V.

Esempio 3.1 (Base canonica di $\mathbb{R}^n$)**.**
La *base canonica* di $\mathbb{R}^n$ è l'insieme di vettori $\mathcal{B} = \{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$ dove $\mathbf{e}_i \in \mathbb{R}^n$ è il vettore con 1 in posizione i-esima e 0 altrove. Ad esempio, il vettore $\mathbf{v} = \begin{bmatrix} 4 & 5 & 6 \end{bmatrix}^T \in \mathbb{R}^3$ può essere espresso come combinazione lineare della base canonica $\mathcal{B} = \{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$:

$$\mathbf{v} = \sum_{i=1}^{3} v_i \mathbf{e}_i = 4\mathbf{e_1} + 5\mathbf{e_2} + 6\mathbf{e_3}$$

Le coordinate di $\mathbf{v}$ rispetto alla base canonica di $\mathbb{R}^3$ sono quindi date dal vettore $\begin{bmatrix} 4 & 5 & 6 \end{bmatrix}$.

Lo spazio vettoriale da solo non è sufficiente a rappresentare posizioni nello spazio: i vettori descrivono direzioni e grandezze ma non punti, rendendo necessario estendere questo concetto.

Definizione 3.5 (Spazio affine)**.**
Uno *spazio affine* è un insieme di punti sul quale è possibile applicare traslazioni mediante i vettori di uno spazio vettoriale associato, senza che sia definito un punto di origine privilegiato.

Lo spazio euclideo è un esempio di spazio affine reale. Con la definizione degli spazi affini si introducono quindi nuove operazioni:

- Differenza punto-punto: definisce un vettore $\mathbf{v} = P - Q$
- Somma punto-vettore (derivata dalla precedente): definisce un punto $P = Q + \mathbf{v}$

Osservazione 3.1.
È importante tenere a mente che:

- La somma punto-punto non è definita.
- I punti sono locazioni nello spazio e la differenza tra loro è data dal vettore che li congiunge.

La distinzione tra punti e vettori è fondamentale nella computer grafica tridimensionale: i punti rappresentano posizioni degli oggetti nella scena, mentre i vettori descrivono direzioni, spostamenti e normali alle superfici. Tale distinzione verrà formalizzata ulteriormente tramite le coordinate omogenee, introdotte nelle sezioni successive.

3.1.2 Combinazioni e simplessi

Vengono riportate le definizioni dei concetti di combinazione di punti, illustrando successivamente il ruolo fondamentale che ricoprono nell'ambito della computer grafica.

Definizione 3.6 (Combinazione affine)**.**
Una *combinazione affine* è una combinazione lineare di punti, i cui coefficienti hanno somma 1.

$$P = \alpha_0 P_0 + \cdots + \alpha_n P_n = \sum_{i=0}^{n} \alpha_i P_i \quad \text{con} \quad \sum_{i=0}^{n} \alpha_i = 1$$

I coefficienti $\alpha_0, \ldots, \alpha_n$ sono le *coordinate baricentriche* (affini) di P nello spazio affine.

Definizione 3.7 (Simplesso)**.**
Un *simplesso* di dimensione k è il più piccolo insieme convesso che contiene $k+1$ punti indipendenti. I $k+1$ punti sono detti *vertici* del simplesso.

Esempio 3.2 (Simplessi conosciuti)**.**
Un simplesso è una generalizzazione a dimensioni arbitrarie di concetti familiari: ad esempio per $k = 0$ è un singolo punto, per $k = 1$ un segmento (due vertici distinti), per $k = 2$ un triangolo (tre punti non allineati) e per $k = 3$ un tetraedro (quattro punti non complanari).

Definizione 3.8 (Combinazione convessa)**.**
Una *combinazione convessa* è una combinazione affine in cui ogni coefficiente $\alpha_i \in [0, 1]$:

$$P = \sum_{i=0}^{n} \alpha_i P_i \quad \text{con} \quad \sum_{i=0}^{n} \alpha_i = 1 \text{ e } \alpha_i \in [0, 1] \ \forall \ i = 0, \ldots, n$$

Osservazione 3.2.
Le coordinate baricentriche rappresentano un sistema di coordinate relative ai vertici di un simplesso. Se vincolate a formare una combinazione convessa, garantiscono che i punti ottenibili come combinazione dei vertici di un simplesso appartengano al simplesso stesso.

Nel contesto della pipeline grafica, queste nozioni sono particolarmente rilevanti perché i triangoli costituiscono la primitiva geometrica fondamentale elaborata dalla GPU. Le coordinate baricentriche permettono infatti di interpolare sugli elementi triangolari grandezze definite ai vertici, come colori, normali o coordinate di texture.

3.1.3 Operazioni tra vettori

Si introducono ora alcune operazioni fondamentali tra vettori, che verranno utilizzate in diversi passaggi della pipeline grafica, ad esempio nel calcolo dell'illuminazione. Verrà inoltre richiamata la norma euclidea, che consente di misurare la lunghezza di un vettore.

Definizione 3.9 (Prodotto scalare)**.**
Si definisce *prodotto scalare* un'applicazione $\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ che gode delle seguenti proprietà $\forall \ \mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ e $\lambda_1, \lambda_2 \in \mathbb{R}$:

1. *Simmetria*: $\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{u}$
2. *Linearità*: $\mathbf{v} \cdot (\lambda_1 \mathbf{u} + \lambda_2 \mathbf{w}) = \lambda_1 (\mathbf{v} \cdot \mathbf{u}) + \lambda_2 (\mathbf{v} \cdot \mathbf{w})$
3. *Definitezza positiva*: $\mathbf{v} \cdot \mathbf{v} > 0 \iff \mathbf{v} \neq 0$

Definizione 3.10 (Prodotto scalare standard)**.**
Si definisce *prodotto scalare standard* $\langle \cdot, \cdot \rangle$ tra due vettori $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$:

$$\langle \mathbf{u}, \mathbf{v} \rangle = \mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{n} u_i v_i = u_1 v_1 + \cdots + u_n v_n = \mathbf{u}^T \mathbf{v}$$

Definizione 3.11 (Prodotto vettoriale)**.**
Dati due vettori $\mathbf{u}, \mathbf{v} \in \mathbb{R}^3$ si definisce il loro *prodotto vettoriale* come il vettore $\mathbf{u} \times \mathbf{v} \in \mathbb{R}^3$ così definito:

- Modulo:
$$\|\mathbf{u} \times \mathbf{v}\| = \|\mathbf{u}\| \cdot \|\mathbf{v}\| \cdot \sin\theta$$
(ove θ è l'angolo convesso fra $\mathbf{u}$ e $\mathbf{v}$)
- Direzione: retta perpendicolare a $\mathbf{u}$ e $\mathbf{v}$
- Verso: dato dalla "regola della mano destra"

Dal momento che si lavora con vettori nello spazio, è necessario introdurre una misura della lunghezza per queste quantità. In computer grafica viene utilizzata la *norma euclidea* (o norma 2):

$$\|\mathbf{v}\|_2 = \sqrt{\langle \mathbf{v}, \mathbf{v} \rangle} = \sqrt{\mathbf{v}^T \mathbf{v}} = \sqrt{\sum_{i=1}^{n} (v_i)^2} = \sqrt{v_1^2 + \cdots + v_n^2}$$

Osservazione 3.3.
La norma 2 è invariante rispetto alle trasformazioni ortogonali – come rotazioni e riflessioni – e rispetto alle traslazioni.

3.2 Sistemi di riferimento e trasformazioni geometriche

Per descrivere oggetti e scene tridimensionali è necessario introdurre sistemi di riferimento e trasformazioni geometriche, che consentano di esprimere la posizione degli oggetti nello spazio e di passare da un sistema

di coordinate a un altro. In questa sezione viene presentato il formalismo delle coordinate omogenee, utile per rappresentare in modo uniforme punti e vettori e per descrivere le trasformazioni geometriche tramite moltiplicazioni matriciali. Per semplicità, in alcuni passaggi verrà utilizzata una notazione bidimensionale, senza perdita di generalità rispetto ai concetti illustrati.

3.2.1 Sistemi di riferimento affini e coordinate omogenee

Definizione 3.12 (Sistema di riferimento)**.**
Un *sistema di riferimento* F in uno spazio affine consente di determinare univocamente la posizione di ogni punto dello spazio; è identificato da:

- una base di vettori linearmente indipendenti $\mathcal{B} = \{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n\}$;
- un punto di origine P_0.

Osservazione 3.4.
Rappresentare punti e vettori tramite n scalari (dove n è la dimensione dello spazio) è ambiguo: la stessa n-upla potrebbe indicare sia un punto che un vettore. Estendendo convenzionalmente il prodotto scalare-punto agli spazi affini con $1 \cdot P_0 = P_0$ e $0 \cdot P_0 = 0$, possiamo scrivere:

$$P = v_1\mathbf{e}_1 + v_2\mathbf{e}_2 + v_3\mathbf{e}_3 + 1 \cdot P_0$$

$$\mathbf{v} = v_1\mathbf{e}_1 + v_2\mathbf{e}_2 + v_3\mathbf{e}_3 + 0 \cdot P_0$$

Notiamo che la coordinata finale vale 1 per i punti e 0 per i vettori.

Definizione 3.13 (Coordinate omogenee)**.**
La coordinata aggiuntiva che distingue punti e vettori è detta *coordinata omogenea* e indicata con w. Dato il sistema di riferimento $F(\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, P_0)$, punti e vettori in coordinate omogenee sono rappresentati da:

$$P = \begin{bmatrix} p_1 & p_2 & p_3 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \\ \mathbf{e}_3 \\ P_0 \end{bmatrix} \qquad \mathbf{v} = \begin{bmatrix} v_1 & v_2 & v_3 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \\ \mathbf{e}_3 \\ P_0 \end{bmatrix}$$

3.2.2 Cambiamento di sistema di riferimento

Si mostra ora come il formalismo introdotto permetta di descrivere il passaggio da un sistema di riferimento a un altro mediante opportune matrici di trasformazione, che consentono di esprimere le coordinate di uno stesso punto rispetto a sistemi differenti.

Dati due sistemi di riferimento affini $F_1 = (\mathbf{x}, \mathbf{y}, O)$ e $F_2 = (\mathbf{u}, \mathbf{v}, \mathbf{e})$, rappresentiamo i vettori e l'origine di F_2 in funzione dei vettori e dell'origine di F_1:

$$\begin{cases} \mathbf{u} = u_x\mathbf{x} + u_y\mathbf{y} + 0 \cdot O \\ \mathbf{v} = v_x\mathbf{x} + v_y\mathbf{y} + 0 \cdot O \\ \mathbf{e} = e_x\mathbf{x} + e_y\mathbf{y} + 1 \cdot O \end{cases} \quad \Longrightarrow \quad \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \\ \mathbf{e} \end{bmatrix} = \underbrace{\begin{bmatrix} u_x & u_y & 0 \\ v_x & v_y & 0 \\ e_x & e_y & 1 \end{bmatrix}}_{M} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ O \end{bmatrix}$$

Proposizione 3.1.
Sia $\mathbf{a} = \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix}^T$ il vettore delle coordinate omogenee di un punto P rispetto al sistema F_1, e sia $\mathbf{b} = \begin{bmatrix} x_2 & y_2 & 1 \end{bmatrix}^T$ il medesimo punto espresso rispetto a F_2. Indicando con M la matrice che esprime la base di F_2 rispetto a quella di F_1, valgono le seguenti relazioni di cambiamento di coordinate:

- La matrice trasposta M^T trasforma le coordinate dal sistema F_2 al sistema F_1:
$$\mathbf{a} = M^T\mathbf{b}$$
- La matrice inversa della trasposta $(M^T)^{-1}$ trasforma le coordinate dal sistema F_1 al sistema F_2:
$$\mathbf{b} = (M^T)^{-1}\mathbf{a}$$

Una volta introdotti gli strumenti per esprimere le coordinate di uno stesso punto rispetto a sistemi di riferimento differenti, è possibile passare alle trasformazioni geometriche più comuni, utilizzate nella pipeline grafica per manipolare gli oggetti della scena.

3.2.3 Trasformazioni geometriche elementari

Le trasformazioni geometriche elementari (traslazione, rotazione e scalatura) possono essere espresse mediante matrici, in modo analogo a quanto visto per i cambiamenti di sistema di riferimento. In questo caso, tuttavia, non si modifica il sistema di coordinate, ma si applicano le matrici direttamente alle coordinate dei punti, così da cambiarne la posizione nello spazio. Nel seguito si mostra come costruire le matrici associate alle singole trasformazioni e come applicarle a un punto espresso in coordinate omogenee. Per semplicità, si considera il caso bidimensionale, trasformando un punto $P = \begin{bmatrix} P_x & P_y & 1 \end{bmatrix}$ in un altro punto $P' = \begin{bmatrix} {P'}_x & {P'}_y & 1 \end{bmatrix}$. Il risultato delle diverse trasformazioni è illustrato in Figura 3.1.

Traslazione

Dato il vettore di traslazione $\mathbf{e} = \begin{bmatrix} e_x & e_y \end{bmatrix}$, otteniamo il punto P' traslando P del vettore $\mathbf{e}$ grazie alla matrice di traslazione T:

$$P' = \underbrace{\begin{bmatrix} 1 & 0 & e_x \\ 0 & 1 & e_y \\ 0 & 0 & 1 \end{bmatrix}}_{T(\mathbf{e})} P$$

Rotazione

Dato l'angolo di rotazione θ, otteniamo il punto P' ruotando P dell'angolo θ attorno all'origine del sistema di riferimento utilizzando la matrice di rotazione R:

$$P' = \underbrace{\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{R(\theta)} P$$

Per le rotazioni nello spazio tridimensionale, risulta comodo utilizzare un asse di rotazione $\hat{\mathbf{n}}$ e un angolo θ.

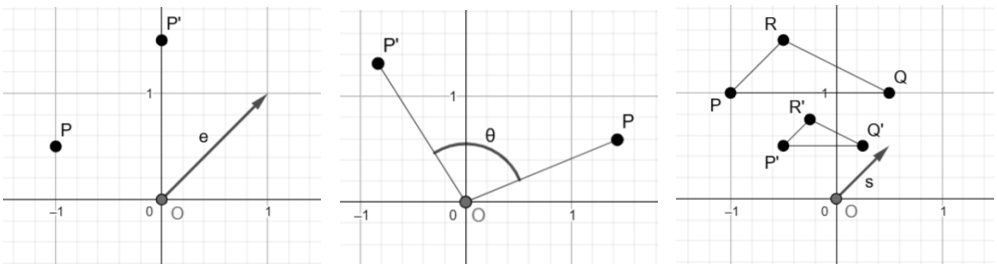
Scalatura

Dato il vettore contenente i fattori di scala $\mathbf{s} = \begin{bmatrix} s_x & s_y \end{bmatrix}$, otteniamo il punto P' scalando P con i fattori $\mathbf{s}$ grazie alla matrice di scala S:

$$P' = \underbrace{\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{S(\mathbf{s})} P$$

Tenendo conto che:

- Fattori $s < 1$: i punti si avvicinano all'origine
- Fattori $s > 1$: i punti si allontanano dall'origine
- Se $s_x \neq s_y$: le proporzioni non sono mantenute (*scalatura non uniforme*)
- Se $s_x = s_y$: le proporzioni sono mantenute (*scalatura uniforme*)



(a) Traslazione (b) Rotazione (c) Scalatura

Figura 3.1: Applicazione delle trasformazioni geometriche elementari a dei punti del piano

Dopo aver introdotto gli strumenti geometrici necessari per rappresentare e trasformare punti e vettori, si può ora mostrare come tali concetti intervengano nel processo di rendering.

3.3 La pipeline grafica

La pipeline grafica è la sequenza di operazioni atte a costruire un'immagine partendo dagli oggetti tridimensionali presenti nella scena. L'immagine prodotta è di tipo *raster*, poiché è composta da una matrice di punti – detti picture elements (pixel) – ognuno contenente le proprie informazioni sul colore.

La GPU gestisce il processo di rendering tramite alcuni buffer, che non sono altro che matrici in cui ad ogni pixel è associata una cella di memoria. I principali sono il *color buffer*, che contiene il colore di ogni pixel dell'immagine finale, e il *depth buffer* (o *z-buffer*), che memorizza la profondità del frammento più vicino all'osservatore per ogni pixel, permettendo di gestire correttamente la visibilità tra oggetti sovrapposti. L'insieme di questi buffer forma il *framebuffer*, che viene letto e scritto nelle varie fasi della pipeline.

Poiché all'interno del color buffer viene scritta l'informazione del colore, è importante capire come questo viene gestito dalla GPU. Il modello di colore RGBA – ovvero Red, Green, Blue, Alpha – è un'estensione del modello RGB in cui si include una quarta componente per il canale alpha, utilizzato per indicare l'opacità del colore. La GPU rappresenta quindi i colori tramite la quaterna di valori RGBA, con ogni componente compresa nell'intervallo $[0, 1]$: per $\alpha = 1$ il colore è completamente opaco, mentre per $\alpha = 0$ è completamente trasparente.

L'intero processo di visualizzazione può essere schematizzato in quattro sottosistemi:

- Applicazione: produce le primitive geometriche tridimensionali a partire dai dati e dalle interazioni dell'utente.
- Geometria: applica alle primitive una sequenza di trasformazioni matriciali, che consentono di passare dai sistemi di riferimento della scena 3D alle coordinate normalizzate utilizzate dalla pipeline grafica.
- Rasterizzazione: converte le primitive geometriche risultanti in frammenti da visualizzare sullo schermo tramite appositi algoritmi.

- Visualizzazione: mostra sul dispositivo di output, ad esempio un monitor, l'immagine finale generata.

Durante le fasi di geometria e rasterizzazione, a partire da OpenGL 2.0, il programmatore può intervenire all'interno della pipeline tramite dei programmi scritti in linguaggio OpenGL Shading Language (GLSL), che prendono il nome di *shader*. Esistono vari tipi di shader che agiscono in momenti distinti, ma gli obbligatori sono il *vertex shader*, eseguito per ogni vertice all'inizio del sottosistema geometrico, e il *fragment shader*, eseguito per ogni frammento prodotto dalla rasterizzazione. Nel progetto viene utilizzato anche il *geometry shader*, che viene eseguito subito dopo il vertex shader e, a differenza di quest'ultimo, riceve in input una primitiva geometrica completa, potendo così accedere a tutti i suoi vertici contemporaneamente. In Figura 3.2 è possibile visualizzare dove intervengono durante il processo di produzione delle immagini.

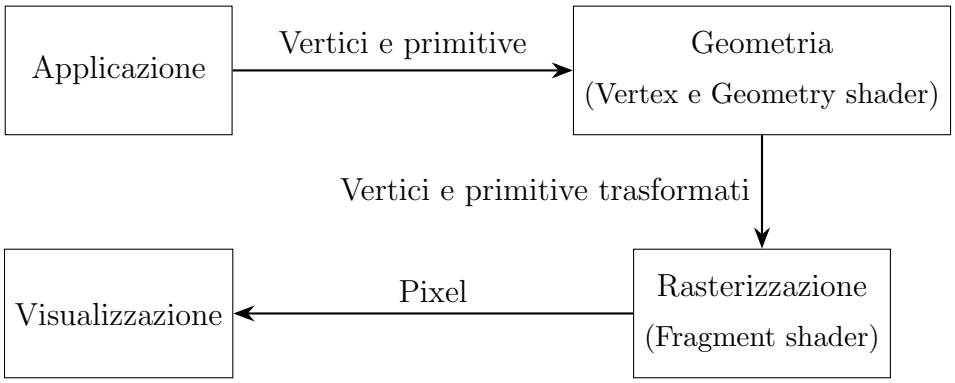


Figura 3.2: Schema del processo di produzione di un'immagine

Dopo aver introdotto gli strumenti necessari per rappresentare e trasformare punti e vettori, è possibile analizzarne l'impiego all'interno della pipeline grafica. Nel sottosistema geometrico tali strumenti permettono di descrivere le trasformazioni tra i diversi sistemi di riferimento; nel sottosistema di rasterizzazione, invece, intervengono nella conversione delle primitive geometriche in frammenti.

3.3.1 Modellazione

Inizialmente, ogni oggetto è modellato in un proprio sistema di riferimento, detto *Object Coordinate System* (OCS). Per essere inserito nella scena tridimensionale, l'oggetto viene poi trasformato nel sistema di riferimento globale, detto *World Coordinate System* (WCS). Questa fase è detta di *modellazione*, e consiste nell'applicare ai vertici dell'oggetto una sequenza di trasformazioni geometriche, tipicamente scalatura, rotazione e traslazione. Poiché le trasformazioni geometriche sono associative ma non commutative, è necessario che vengano applicate nell'ordine corretto. Dal momento che nelle moltiplicazioni matriciali l'ultima matrice è la prima ad agire, la matrice di modellazione è composta nella seguente maniera:

$$M = T(e) \cdot R(\theta) \cdot S(s)$$

3.3.2 Vista

In questa fase, viene posizionata e orientata nella scena una telecamera virtuale, che determina il punto di vista da cui la scena viene osservata. La trasformazione di vista è una matrice che trasforma tutti i vertici dal WCS al *View Coordinate System* (VCS), cioè il sistema di riferimento solidale con la camera. In tale sistema, l'origine coincide con la posizione della camera e, per convenzione, la direzione di osservazione è orientata lungo l'asse $-z$.

Uno dei modelli di camera virtuale più intuitivi e utilizzati, è quello della camera *look at*, che si basa su quattro informazioni:

- Punto C: posizione della telecamera nel WCS;
- Punto A: punto verso cui la telecamera guarda;
- *Field of View* (FOV): angolo di visibilità verticale nella scena, ovvero quanta porzione di scena viene inquadrata;
- *Depth of Field*: determina la distanza tra l'oggetto più vicino e quello più lontano che appaiono a fuoco.

Da queste informazioni possiamo ricavare i versori che definiscono il sistema di riferimento VCS.

La direzione di vista è il versore $\hat{\mathbf{w}}$ che da C punta verso A:

$$\hat{\mathbf{w}} = \frac{C - A}{\|C - A\|}$$

Il *View Up Vector* (VUP) è il vettore che punta verso l'alto della telecamera.

L'asse $\hat{\mathbf{u}}$ è il versore che punta alla destra dell'osservatore: è perpendicolare al piano individuato da VUP e $\hat{\mathbf{w}}$:

$$\hat{\mathbf{u}} = \frac{VUP \times \hat{\mathbf{w}}}{\|VUP \times \hat{\mathbf{w}}\|}$$

L'asse $\hat{\mathbf{v}}$ è a sua volta perpendicolare a $\hat{\mathbf{w}}$ e $\hat{\mathbf{u}}$, ma dato che sono già versori non è necessario rieseguire la normalizzazione:

$$\hat{\mathbf{v}} = \hat{\mathbf{w}} \times \hat{\mathbf{u}}$$

Si definisce quindi il View Coordinate System come il sistema di riferimento $VCS(\hat{\mathbf{u}}, \hat{\mathbf{v}}, \hat{\mathbf{w}}, C)$, avente origine nella posizione della camera C e assi individuati dai versori $\hat{\mathbf{u}}, \hat{\mathbf{v}}$ e $\hat{\mathbf{w}}$. Conoscendo questi vettori risulta semplice calcolare la matrice di vista. Come visto nella Sezione 3.2.2 per il cambio di sistema di riferimento, la matrice di trasformazione delle basi che esprime il $VCS(\hat{\mathbf{u}}, \hat{\mathbf{v}}, \hat{\mathbf{w}}, C)$ in termini del $WCS(\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}, O)$ è:

$$M = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ C_x & C_y & C_z & 1 \end{bmatrix}$$

Poiché la matrice di vista V dovrà trasformare i punti da WCS a VCS, per la Proposizione 3.1 è composta come:

$$V = (M^T)^{-1}$$

3.3.3 Proiezione

In generale, una proiezione è una trasformazione geometrica che ha come dominio uno spazio di dimensione n e come codominio uno spazio

di dimensione $n-1$. Nel caso della grafica tridimensionale, la proiezione consente di rappresentare un oggetto 3D su un piano bidimensionale.

Tale rappresentazione si ottiene mediante un insieme di rette di proiezione, dette proiettori, che attraversano i punti dell'oggetto e intersecano il piano di proiezione, determinando così l'immagine risultante. Quando i proiettori sono rette e la superficie di proiezione è un piano, si parla di proiezioni geometriche piane. Queste si distinguono principalmente in due categorie: proiezioni prospettiche e proiezioni parallele

Nella pipeline grafica, l'operazione di proiezione può essere distinta in due fasi: la prima consiste nella definizione del volume di vista e nella sua mappatura in un cubo canonico, trattate in questa sezione; la seconda riguarda la riduzione della rappresentazione tridimensionale a immagine bidimensionale, affrontata nella sezione successiva. Il volume di vista è la regione di spazio inquadrata dalla camera virtuale, delimitata da due piani, detti piano vicino (*near plane*) e piano lontano (*far plane*).

Nella prima fase della proiezione, mediante la matrice di proiezione P, le coordinate dei vertici vengono trasformate dal View Coordinate System (VCS) al sistema delle *Normalized Device Coordinates* (NDC), un cubo canonico dai lati di lunghezza 2 centrato nell'origine tale che $x, y, z \in [-1, 1]$, così da semplificare sensibilmente i calcoli svolti negli stadi successivi. Al termine della moltiplicazione per la matrice di proiezione, la pipeline riporta automaticamente i punti in coordinate cartesiane (quindi con la quarta coordinata $w = 1$) dividendo tutte le componenti per w, operazione detta *perspective divide*.

Proiezioni parallele

Il nome proiezioni parallele deriva dalla proprietà di parallelismo dei proiettori: non c'è un centro di proiezione, ma si parla di direzione di proiezione. Poiché vengono mantenute le distanze, linee parallele nella realtà lo sono anche nel volume proiettato.

Si dividono a loro volta in due classi, le proiezioni oblique e quelle ortografiche. Nelle proiezioni oblique i proiettori possono formare un angolo qualsiasi col piano di proiezione: ne è un esempio la cavaliera, utilizzata

nel disegno tecnico, in cui la direzione di proiezione forma un angolo di 45 gradi con il piano di proiezione. Nelle proiezioni ortografiche invece, la direzione di proiezione coincide con la normale al piano di proiezione: in questo caso il volume di vista corrisponde a un parallelepipedo (visibile in Figura 3.3). La sua trasformazione nel cubo canonico delle NDC può essere ottenuta mediante due operazioni: una traslazione, che porta il centro del volume di vista nell'origine, e una scalatura, che riporta le coordinate dei suoi vertici nell'intervallo $[-1, 1]$. Dati i vertici del parallelepipedo, il suo centroide è identificato dal punto:

$$C = \begin{bmatrix} C_x & C_y & C_z \end{bmatrix} = \begin{bmatrix} \frac{left+right}{2} & \frac{top+bottom}{2} & \frac{-near-far}{2} \end{bmatrix}$$

Per traslarlo nell'origine, il vettore di traslazione è semplicemente:

$$\mathbf{e} = O - C = \begin{bmatrix} -C_x & -C_y & -C_z \end{bmatrix}$$

Mentre i fattori di scala per ottenere lati di lunghezza 2 saranno:

$$\mathbf{s} = \begin{bmatrix} \frac{2}{right-left} & \frac{2}{top-bottom} & \frac{2}{-near+far} \end{bmatrix} = \begin{bmatrix} s_x & s_y & s_z \end{bmatrix}$$

La matrice di proiezione ortografica si ottiene quindi componendo la matrice di scalatura con quella di traslazione:

$$P_{ortho} = \underbrace{\begin{bmatrix} s_x & & & \\ & s_y & & \\ & & s_z & \\ & & & 1 \end{bmatrix}}_{S(\mathbf{s})} \cdot \underbrace{\begin{bmatrix} 1 & & & -C_x \\ & 1 & & -C_y \\ & & 1 & -C_z \\ & & & 1 \end{bmatrix}}_{T(\mathbf{e})} = \begin{bmatrix} s_x & & & -C_x s_x \\ & s_y & & -C_y s_y \\ & & s_z & -C_z s_z \\ & & & 1 \end{bmatrix}$$

Dall'ultima riga della matrice notiamo che la coordinata w viene sempre mantenuta a 1: in questo caso la perspective divide risulta quindi ininfluente.

Proiezioni prospettiche

Nelle proiezioni prospettiche il centro di proiezione si trova a distanza finita dal piano di proiezione: gli oggetti più vicini all'osservatore risulteranno più grandi e viceversa. Il volume di vista è un tronco di piramide

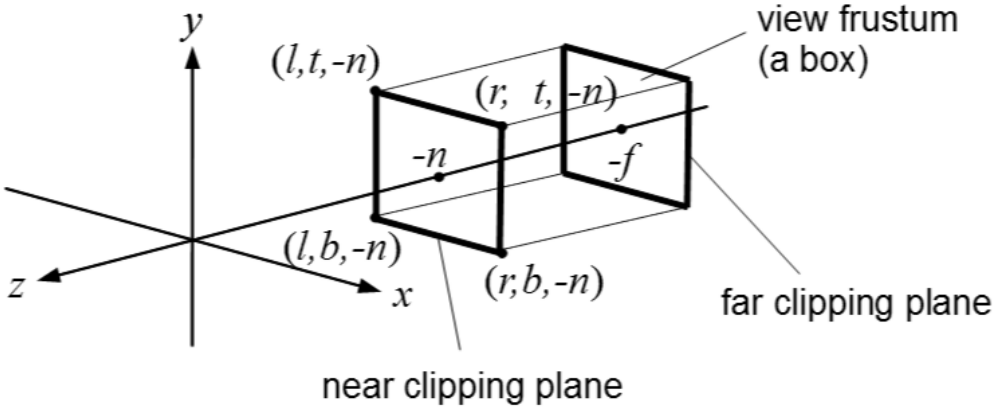


Figura 3.3: Volume di vista della proiezione ortografica [22]

(visibile in Figura 3.4); per ricondurre tale volume al cubo canonico delle NDC, si procede in due passaggi. Anzitutto il tronco di piramide viene trasformato in un parallelepipedo, comprimendo progressivamente i punti in funzione della loro distanza dalla camera. Successivamente, il parallelepipedo ottenuto viene normalizzato nel cubo canonico mediante una trasformazione analoga a quella utilizzata per la proiezione ortografica.

Detta d la distanza del piano di proiezione dal centro di proiezione, si vuole che i punti $P = \begin{bmatrix} x & y & z & 1 \end{bmatrix}$ vengano mappati in altri punti in cui la grandezza dipende dalla distanza: l'obiettivo è quindi costruire una matrice M_{persp} tale che, dopo la perspective divide, si ottenga $x' = x \cdot d/z$ e $y' = y \cdot d/z$.

Per ottenere questa divisione per $\frac{z}{d}$, è sufficiente che la quarta riga della matrice produca $w = \frac{z}{d}$: la quarta riga della matrice è quindi $\begin{bmatrix} 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$. Le prime due righe lasciano invece x e y inalterati, poiché la divisione per w penserà a scalarle correttamente.

Rimane da determinare la terza riga della matrice, che deve rimappare z nell'intervallo $[-1, 1]$ del cubo canonico. Essa non può dipendere da x e y (altrimenti la profondità di un punto cambierebbe in base alla sua posizione orizzontale o verticale), assumendo quindi necessariamente la forma $\begin{bmatrix} 0 & 0 & a & b \end{bmatrix}$, dove a e b sono due incognite. Dopo la perspective divide, la coordinata normalizzata z_{NDC} risulta:

$$z_{NDC} = \frac{az + b}{z/d} = \frac{(az + b)d}{z}$$

Imponendo che il piano vicino venga mappato in $z = -1$ e il piano lontano in $z = +1$, e assumendo che il piano di proiezione corrisponda al piano vicino ($d = near$):

$$\begin{cases} \dfrac{(a \cdot near + b) \cdot d}{near} = -1 \\ \dfrac{(a \cdot far + b) \cdot d}{far} = +1 \end{cases} \quad \overset{d \,=\, near}{\Longrightarrow} \quad \begin{cases} a \cdot near + b = -1 \\ \dfrac{(a \cdot far + b) \cdot near}{far} = +1 \end{cases}$$

La cui soluzione è:

$$a = \frac{far + near}{far - near}, \qquad b = -\frac{2 \cdot far \cdot near}{far - near}$$

La matrice M_{persp} è quindi:

$$M_{persp} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \frac{far+near}{far-near} & -\frac{2 \cdot far \cdot near}{far-near} \\ & & \frac{1}{near} & 0 \end{bmatrix}$$

Una volta trasformato il volume di vista in un parallelepipedo, è sufficiente applicare P_{ortho} per normalizzare nel cubo NDC, ottenendo la matrice di proiezione prospettica completa:

$$P_{persp} = P_{ortho} \cdot M_{persp}$$

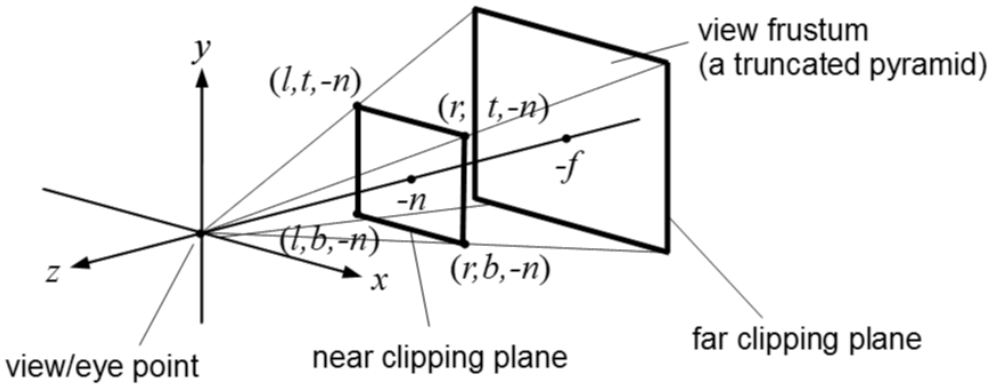


Figura 3.4: Volume di vista della proiezione prospettica [23]

3.3.4 Clipping e viewport

Dopo la trasformazione nel sistema delle Normalized Device Coordinates (NDC), la pipeline esegue il *clipping*, cioè l'eliminazione delle porzioni di primitive geometriche esterne al volume di vista. In questo modo, le fasi successive operano solo sulle parti della scena effettivamente visibili. Il risultato di questa operazione è schematizzato, per una scena bidimensionale, in Figura 3.5.

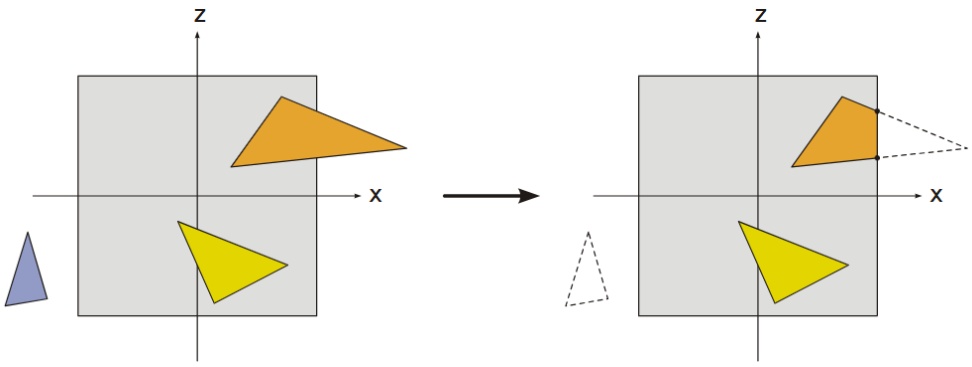


Figura 3.5: Clipping di primitive in una scena bidimensionale [24]

Infine le coordinate x e y, espresse nel sistema NDC, vengono trasformate nel sistema di *viewport*, cioè nel sistema di coordinate in pixel dell'area effettivamente visualizzata sullo schermo. La trasformazione dipende dall'origine della viewport, indicata con (x_0, y_0), e dalle sue dimensioni, indicate con *width* e *height*:

$$x_{vp} = \frac{width}{2} \cdot x_{NDC} + \left(x_0 + \frac{width}{2} \right)$$

$$y_{vp} = \frac{height}{2} \cdot y_{NDC} + \left(y_0 + \frac{height}{2} \right)$$

A questo punto termina il sottosistema geometrico della pipeline e l'elaborazione passa alla rasterizzazione.

3.3.5 Rasterizzazione

Il sottosistema raster è l'ultima fase della pipeline di rendering. Il processo di rasterizzazione si occupa di convertire le primitive geome-

triche (punti, linee, poligoni) espresse in coordinate di viewport in una rappresentazione discreta composta da frammenti, come mostrato in Figura 3.6. Nel passaggio dai vertici ai frammenti, la GPU interpola per ciascun frammento gli attributi associati ai vertici della primitiva, utilizzando le coordinate baricentriche introdotte in precedenza. Una volta generati i frammenti, interviene il *fragment shader*, il cui compito principale è determinarne il colore.

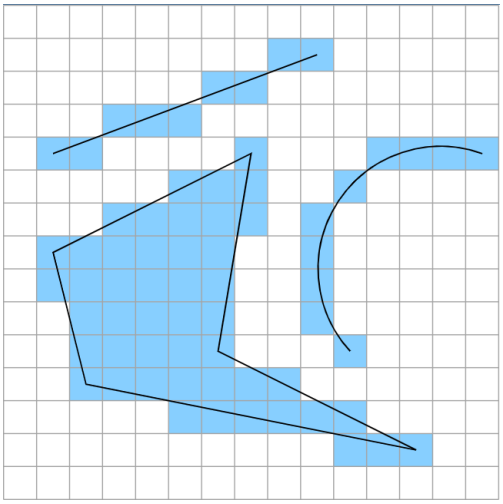


Figura 3.6: Dettaglio di rasterizzazione di alcune primitive geometriche, in cui si mostra come alcuni pixel vengano "accesi" per approssimarne le forme [25]

Una volta terminata l'elaborazione dei frammenti, tramite il *depth test* la GPU decide se scartare il frammento – ad esempio perché coperto da un oggetto più vicino alla camera – o scriverlo nel color buffer. In quest'ultimo caso, il colore viene eventualmente miscelato a quello già presente qualora il *blending* sia attivo. Il blending è una tecnica utilizzata principalmente per implementare la trasparenza: invece di sovrascrivere il color buffer con il colore del frammento candidato, i due colori vengono miscelati in base all'opacità del frammento in arrivo. Un esempio di funzione di blending può essere la seguente, denominata da OpenGL come `GL_ONE_MINUS_SRC_ALPHA`:

$$C_{out} = \alpha \cdot C_{src} + (1 - \alpha) \cdot C_{dst} \quad \text{con} \quad \alpha \in [0, 1] \tag{3.1}$$

3.4 Modelli di illuminazione e shading

I modelli di illuminazione descrivono i fattori che determinano il colore di una superficie in un determinato punto, tramite le interazioni tra luci e superfici, le proprietà delle superfici (materiali) e la natura della radiazione luminosa incidente. Sono classificati in due gruppi, i modelli di illuminazione locale e i modelli di illuminazione globale.

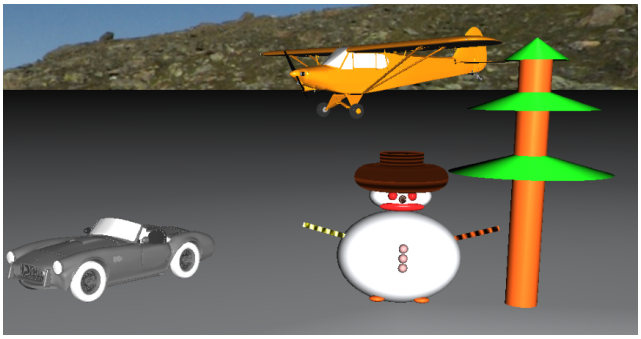
Un modello di illuminazione locale calcola l'intensità luminosa in un singolo punto della superficie considerando esclusivamente il contributo diretto delle sorgenti luminose. Non vengono quindi calcolate ombre, riflessioni all'interno dell'ambiente e/o rifrazioni. Esempi di modelli di illuminazione locale sono il modello di Phong e il modello di Blinn-Phong, che verranno approfonditi successivamente.

Un modello di illuminazione globale, invece, calcola il colore in un punto considerando sia la luce diretta che quella che raggiunge il punto dopo essere stata riflessa e/o trasmessa dalle altre superfici presenti nell'ambiente circostante. I modelli di illuminazione globale più conosciuti sono il *ray tracing*, che simula il percorso dei raggi luminosi, e il *radiosity*, che modella lo scambio di energia luminosa tra superfici tramite equazioni di conservazione dell'energia.

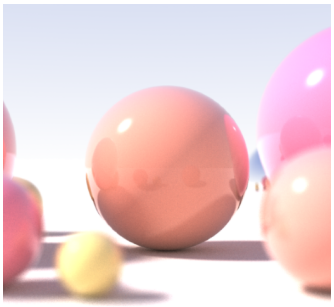
Sebbene i modelli locali siano caratterizzati da maggiore semplicità e minore costo computazionale, i modelli globali consentono di ottenere risultati più vicini al fotorealismo, poiché tengono conto anche dei contributi indiretti della luce. Le differenze tra le due categorie sono illustrate nel confronto riportato in Figura 3.7.

Mentre il calcolo dell'illuminazione si occupa di definire l'intensità luminosa di un punto in base alle proprietà della luce e del materiale, lo *shading* specifica come tale informazione viene effettivamente utilizzata per colorare i pixel dell'immagine finale.

Nei paragrafi seguenti vengono analizzati il modello di Phong e la sua variante migliorata di Blinn-Phong, per poi illustrare come questi vengono applicati attraverso i principali modelli di shading.



(a) Illuminazione locale



(b) Illuminazione globale

Figura 3.7: Immagini in cui si nota il divario tra i modelli di illuminazione locali e globali [26]: emerge chiaramente la differenza nella gestione delle ombre e delle riflessioni

3.4.1 Modello di illuminazione di Phong

Il modello di illuminazione di Phong è un modello di illuminazione locale ideato dall'informatico Bui Tuong Phong nel 1975 [27]. Phong approssima tutti i contributi della luce in 3 componenti – ambientale, diffusivo e speculare – e fa uso di sorgenti di luce puntiformi, ovvero sorgenti dotate di una posizione specifica nello spazio, prive di volume e area, che emettono luce uniformemente in ogni direzione.

Componente ambientale

Nella realtà gli oggetti che non sono direttamente colpiti dalla luce non appaiono completamente neri: questo perché vengono illuminati dalla luce riflessa dall'ambiente circostante. Si approssimano tutti i contributi di questi riflessi nella luce ambientale, una luce diffusa e non direzionale che colpisce ugualmente tutte le superfici da tutte le direzioni. La quantità di luce dell'ambiente riflessa dalla superficie dell'oggetto è determinata dal coefficiente di riflessione ambientale $k_a \in [0, 1]$, ed è una proprietà che caratterizza il materiale di cui la superficie è composta.

Sia k_a il coefficiente di riflessione ambientale del materiale, e I_{amb} l'intensità della luce ambientale (supposta costante per tutti gli oggetti

della scena), allora la componente ambientale A è determinata da:

$$A = I_{amb} \cdot k_a$$

Gli oggetti illuminati da sola luce ambientale sono ancora uniformemente illuminati su tutta la superficie.

Componente diffusiva

Questo termine modella la riflessione lambertiana, tipica di materiali opachi e rugosi come il gesso. Tali superfici appaiono ugualmente luminose da qualunque punto di vista vengano osservate poiché riflettono uniformemente la luce in tutte le direzioni. La luminosità dipende solo dall'angolo θ, ovvero l'angolo formato dalla direzione del raggio luminoso L e la normale alla superficie nel punto d'incidenza N (schematizzato in Figura 3.8): è quindi indipendente dalla posizione dell'osservatore. La quantità di luce data dalla riflessione lambertiana della superficie dell'oggetto è determinata dal coefficiente di riflessione diffusiva $k_d \in [0, 1]$.

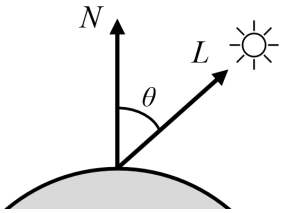


Figura 3.8: Vettori della componente diffusiva

Sia k_d il coefficiente di riflessione diffusiva del materiale e I_p l'intensità della sorgente luminosa puntiforme, allora la componente diffusiva D è data da:

$$D = I_p \cdot k_d \cdot \cos\theta$$

Nel caso in cui N e L siano normalizzati utilizziamo il prodotto scalare:

$$D = I_p \cdot k_d \cdot (\hat{N} \cdot \hat{L})$$

dove la notazione $\hat{\mathbf{v}}$ indica la normalizzazione del vettore $\mathbf{v}$.

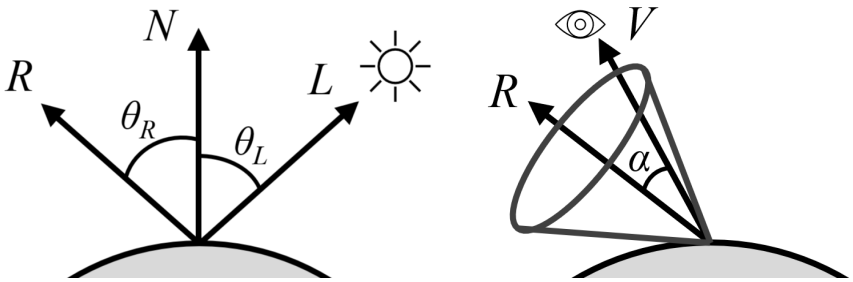
Componente speculare

Qualora la superficie di un oggetto non sia completamente opaca, la luce non viene riflessa ugualmente in tutte le direzioni, ma si concentra in un intervallo ristretto. La direzione di riflessione R è geometricamente la direzione di incidenza della luce L riflessa rispetto ad N. L, R, N sono vettori che giacciono sullo stesso piano, e per definizione l'angolo di riflessione θ_R è uguale all'angolo di incidenza della luce θ_L. Il vettore V è la direzione di vista (ovvero il vettore dato dalla posizione della camera sottratto alla posizione del suo obiettivo), e l'angolo formato col vettore di riflessione è indicato come α. Tutte queste componenti sono rappresentate in Figura 3.9.

Nei materiali perfettamente speculari (come gli specchi), l'osservatore vede il riflesso solo se $\alpha = 0$, quindi solo se il vettore di vista è allineato con il vettore riflessione. Phong trova un modo per simulare i materiali non perfettamente speculari: fa decadere rapidamente la luce riflessa all'aumentare di α, elevando $\cos\alpha$ per un coefficiente n: si genera così un cono intorno alla direzione di riflessione (visibile in Figura 3.9b), in cui il riflesso è comunque visibile. Questo coefficiente n è detto *shininess* (lucentezza), o coefficiente di riflessione speculare, e controlla la dimensione e l'intensità delle macchie di luce che appaiono sulle superfici quando vengono illuminate. Nella formula $\cos^n\alpha$, l'aumento di n corrisponde a una riduzione della dimensione del cono di direzioni intorno al vettore R in cui la luce riflessa rimane visibile: valori alti (100 - 500) corrispondono a un riflesso nitido e puntiforme, tipico delle superfici metalliche, mentre valori bassi (< 100) producono un'ampia zona di massima lucentezza più sfocata, tipica ad esempio di plastica e cera. La quantità di luce riflessa dalla superficie dell'oggetto è invece determinata dal coefficiente di riflessione speculare $k_s \in [0, 1]$.

Dati il coefficiente k_s, la shininess n, l'angolo α formato tra R e V e I_p l'intensità della sorgente luminosa, allora la componente speculare S è data da:

$$S = I_p \cdot k_s \cdot \cos^n \alpha$$



(a) Vettore di riflessione (b) Vettore di vista, cono di riflessione

Figura 3.9: Vettori che concorrono alla creazione della componente speculare

Nel caso in cui R e V siano normalizzati:

$$S = I_p \cdot k_s \cdot (\hat{R} \cdot \hat{V})^n$$

Modello completo

Il modello di illuminazione di Phong unisce le tre componenti appena studiate, ovvero ambientale A, diffusiva D e speculare S, il cui contributo è mostrato in un esempio in Figura 3.10.

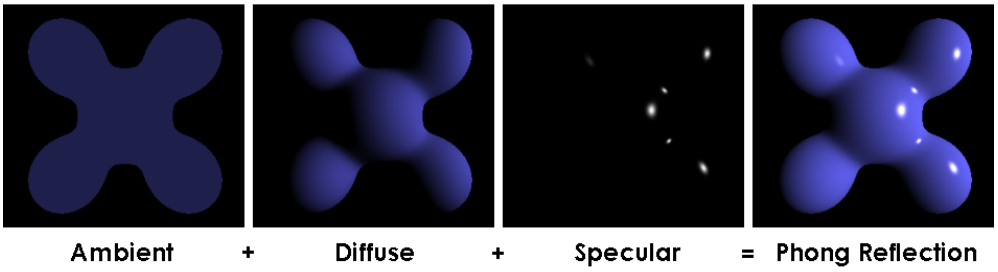


Figura 3.10: Esempio di come ogni componente (ambientale, diffusiva, speculare) dia il suo contributo per la creazione dell'immagine finale [28]

Nel calcolo dell'illuminazione, si può voler tenere conto anche dell'attenuazione dell'intensità luminosa, ad esempio all'aumentare della sua distanza dal punto di incidenza d_L: $f_{att} = 1/{d_L}^2$. Unendo tutti questi contributi si ha la formula $I = A + f_{att}(D + S)$, che espansa fornisce l'equazione del modello di illuminazione di Phong, in cui i vettori utilizzati

sono schematizzati in Figura 3.11:

$$I = I_{amb} \cdot k_a + f_{att} \cdot I_p[k_d(\hat{N} \cdot \hat{L}) + k_s(\hat{R} \cdot \hat{V})^n]$$

Nel caso siano presenti più luci è sufficiente sommare le componenti diffusive e speculari per ogni sorgente puntiforme, lasciando invariata quella ambientale:

$$I = I_{amb} \cdot k_a + \sum_{l \in luci} f_{attl} \cdot I_{pl}[k_d(\hat{N} \cdot \hat{L}_l) + k_s(\hat{R}_l \cdot \hat{V})^n]$$

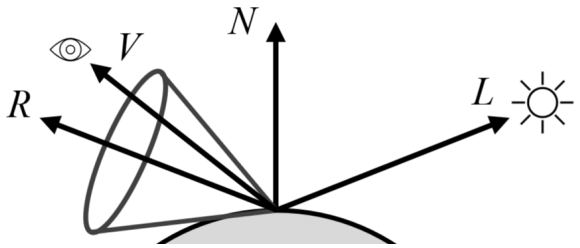


Figura 3.11: Vettori del modello di illuminazione di Phong

3.4.2 Modello di illuminazione di Blinn-Phong

James F. Blinn, un informatico statunitense, nel 1977 propone una variante del modello di Phong per quanto riguarda il calcolo della componente speculare [29]. Introduce l'*halfway vector* al posto del vettore di riflessione $\hat{R}$ che, oltre a produrre risultati visivamente più accurati per superfici viste a angoli rasenti, alleggerisce la complessità computazionale del modello.

L'halfway vector H è un vettore a metà strada tra la direzione di vista V e la luce L: è quindi la diagonale del parallelogramma formato da V ed L (visibile in Figura 3.12). Dato che sappiamo che la diagonale del parallelogramma formato da due vettori non è altro che la loro somma, il vettore H si calcola:

$$H = L + V$$

Spostando V verso R accade che H si avvicina a N. Nel punto di massima riflessione, ovvero $V = R$, si ha che H è allineato ad N: l'angolo β, formato tra N e H, approssima bene l'angolo α.

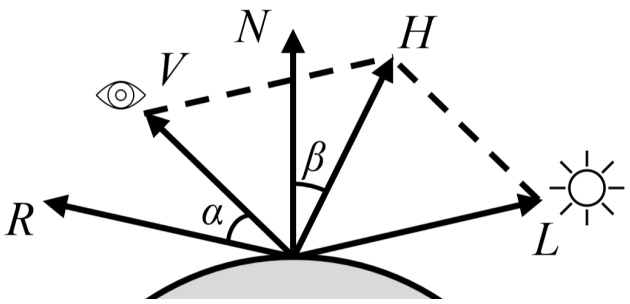
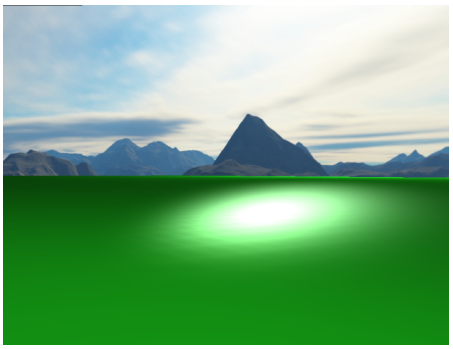


Figura 3.12: Costruzione dell'halfway vector

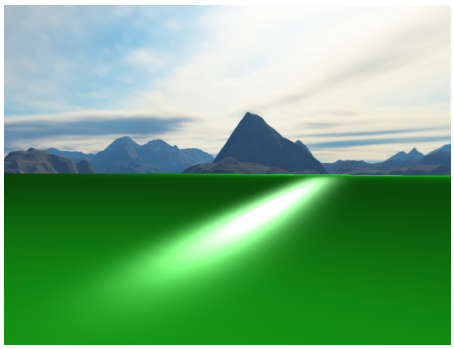
Blinn decide quindi di sostituire $(\hat{R} \cdot \hat{V})^n$ con $(\hat{N} \cdot \hat{H})^n$, creando così l'equazione del modello di illuminazione di Blinn-Phong:

$$I = I_{amb} \cdot k_a + f_{att} \cdot I_p[k_d(\hat{N} \cdot \hat{L}) + k_s(\hat{N} \cdot \hat{H})^n] \tag{3.2}$$

Bisogna però notare che spesso β è più piccolo di α usato in Phong, tranne quando la superficie è vista da un angolo molto ripido. Per avvicinarsi di più al risultato di $(R \cdot V)^n$ di Phong, in $(N \cdot H)^{n_\beta}$ sarebbe opportuno scegliere $n_\beta > n$. Inoltre, mentre Phong produce riflessi quasi circolari su una superficie piana, quelli di Blinn-Phong diventano ellittici quando la superficie viene vista da un angolo radente, ad esempio quando il sole vicino all'orizzonte viene riflesso sulla superficie del mare, rappresentato in Figura 3.13.



(a) Modello di Phong



(b) Modello di Blinn-Phong

Figura 3.13: Confronto dei riflessi speculari generati ad angolo radente: la formulazione di Phong produce una macchia quasi circolare, mentre il modello di Blinn-Phong introduce l'allungamento ellittico caratteristico delle superfici reali

3.4.3 Modelli di shading

I modelli di shading rappresentano il processo matematico e algoritmico utilizzato per determinare il colore di tutti i pixel che ricoprono una superficie, applicando un determinato modello di illuminazione. Si differenziano principalmente per la frequenza con cui viene ricalcolato il modello di illuminazione, dando diversi effetti visibili in Figura 3.14.

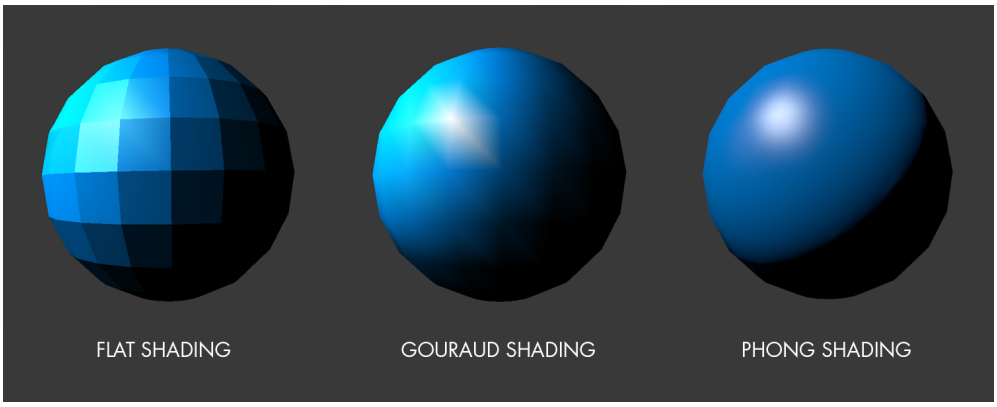


Figura 3.14: Confronto tra i principali modelli di shading; lo shading interpolativo, non rappresentato, si colloca concettualmente tra flat e Gouraud shading [30]

Flat shading

È il modello più semplice ed efficiente: il modello di illuminazione è applicato una sola volta per poligono. Questo può rendere le divisioni in facce visibili su superfici curve e crea una discontinuità netta tra facce adiacenti. Viene ad esempio calcolato il colore solo per il centroide del triangolo, per poi applicarlo a tutto il poligono.

Shading interpolativo

L'equazione di illuminazione viene calcolata per ogni vertice del poligono, per poi eseguire un'interpolazione lineare per colorarne ogni pixel. Poiché non teniamo conto che un triangolo fa parte di una geometria più complessa, abbiamo una sola normale per triangolo. Questo causa brusche variazioni delle normali ai bordi dei poligoni adiacenti, renden-

do i bordi nettamente visibili: questo fenomeno prende il nome di *Mach banding*.

Gouraud shading

Henri Gouraud, un informatico francese, propone nel 1971 un modello di shading che rappresenta la diretta evoluzione di quello interpolativo [31]. Esegue comunque il calcolo dell'illuminazione per ogni vertice del poligono, ma invece di utilizzare la normale del poligono usa quella della superficie che sta approssimando. Se la normale della superficie non è nota, la si approssima con la media delle normali ai poligoni che condividono il vertice. Si procede poi con l'interpolazione della luce calcolata nei vertici. Non è ideale per materiali molto riflettenti o speculari, poiché i punti luce possono andare persi se ricadono all'interno di un poligono senza colpirne i vertici.

Phong shading

È il modello più sofisticato e realistico ma anche più computazionalmente costoso. Proposto sempre da Phong nel 1975 [27], prevede il calcolo dell'illuminazione per ogni pixel del poligono, utilizzando il vettore normale ottenuto dall'interpolazione delle normali note. Promette così di gestire correttamente le riflessioni speculari, anche quando queste non colpiscono direttamente i vertici.

I concetti introdotti in questo capitolo costituiscono la base teorica necessaria per comprendere le scelte progettuali e implementative del nuovo event display 3D, che verranno descritte nel capitolo successivo.

Capitolo 4

Progettazione del sistema

In questo capitolo vengono illustrate l'architettura del sistema e le principali scelte progettuali adottate nello sviluppo del nuovo event display. Tali scelte derivano dall'analisi dei requisiti discussa nel Capitolo 2 e mirano a garantire modularità, manutenibilità ed estendibilità del software. Dopo una descrizione generale dell'interazione tra i moduli, verranno analizzate nel dettaglio le soluzioni progettuali adottate, con particolare attenzione ai *design pattern* utilizzati [32].

4.1 Architettura generale

Per favorire la modularità del software, è stato scelto di dividerlo in package, così da ridurre le dipendenze che aumenterebbero la complessità della manutenzione. La struttura generale è riportata in Figura 4.1, che evidenzia come la gestione di ogni dipendenza esterna principale sia delegata a un solo modulo.

Il cuore del sistema è contenuto nel package `core`, che racchiude le classi responsabili del ciclo di vita del software, dalla creazione della finestra all'interazione con l'utente. Al suo interno è incluso il modulo `state`, a cui è delegata la gestione della macchina a stati finiti che governa il ciclo di vita dell'applicativo, che verrà illustrata nelle prossime sezioni.

Il modulo `io` isola la dipendenza da ROOT: contiene infatti le strutture utilizzate per rappresentare i dati degli eventi, e un manager che si

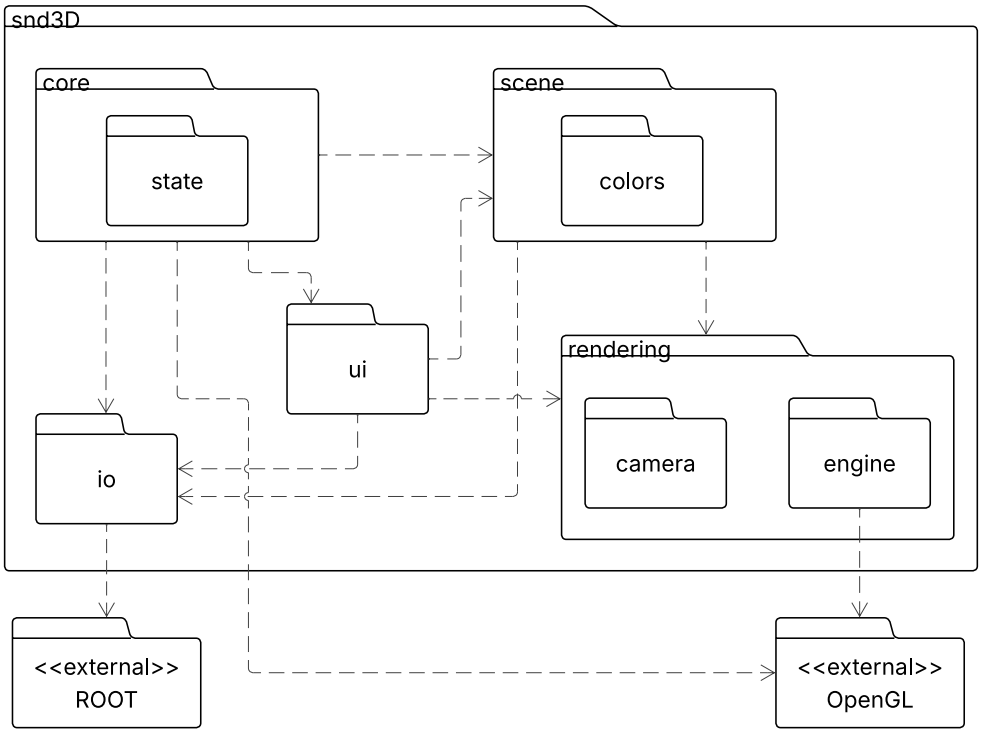


Figura 4.1: Diagramma dei package del sistema, in cui sono riportate anche le principali dipendenze esterne

interfaccia con sndsw per popolarle. Permette quindi di concentrare in un unico punto la lettura e la conversione dei dati da ROOT, rendendone semplice il riadattamento in caso di future modifiche a sndsw.

Passando al livello di visualizzazione dei dati, il modulo `core` presenta una dipendenza verso `scene`, che modella la gerarchia tridimensionale (nodi, mesh, luci) e ne gestisce la rappresentazione. Per evitare accoppiamenti rigidi, `scene` riceve i dati geometrici dal modulo `io` tramite strutture dati astratte, mentre definisce le proprietà cromatiche degli elementi attraverso il package `colors`.

La visualizzazione avviene grazie al modulo `rendering`, che è diviso in due sotto-moduli: `camera` che definisce come la scena viene inquadrata e proiettata e `engine`, che utilizza le API di OpenGL per eseguire il rendering degli elementi della scena. Nell'eventualità di un upgrade futuro delle API grafiche, sarà quindi sufficiente riadattare le classi contenute in `engine`. In realtà, anche il modulo `core` presenta una dipendenza limitata

a OpenGL, circoscritta all'inizializzazione del contesto e alla gestione del render loop; tutta la logica di rendering degli oggetti è invece delegata ai moduli `rendering` e `scene`.

Infine, il package `ui` presenta il maggior numero di dipendenze, in quanto è responsabile delle funzionalità legate all'interazione con l'utente. Per soddisfare i requisiti di usabilità, esso consente sia di visualizzare le informazioni testuali associate ai dati caricati, sia di controllare la rappresentazione della scena, ad esempio selezionando quali parti del rivelatore SND@LHC includere o escludere dalla visualizzazione.

4.2 Gestione della macchina a stati finiti

L'intero software è basato su una macchina a stati finiti (*Finite State Machine*, FSM) così da avere il pieno controllo sul suo flusso di esecuzione. In questa sezione vengono presentati gli stati di alto livello, riportati in Figura 4.2; successivamente saranno approfonditi gli stati più complessi, che verranno scomposti nei relativi sottostati.

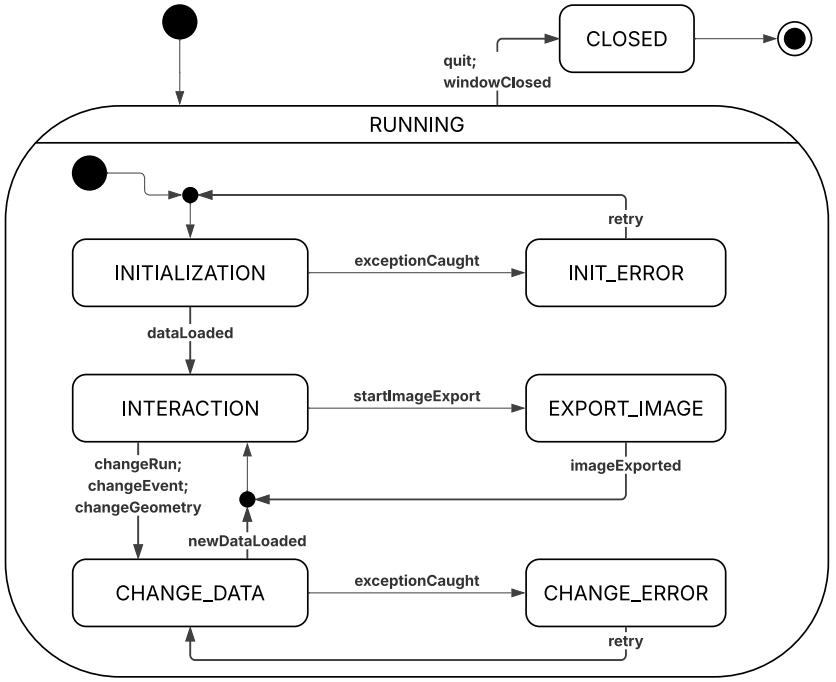


Figura 4.2: Diagramma ad alto livello della FSM

La prima divisione negli stati `RUNNING` e `CLOSED`, permette di chiudere il software in qualsiasi momento del suo ciclo di vita e centralizzarne l'arresto. È ovvero possibile catturare l'evento di chiusura in svariati modi (chiusura finestra, pulsante nel menu, ecc.), garantendo in ogni caso la corretta deallocazione delle risorse e la corretta sequenza di arresto.

Il primo stato in cui si entra automaticamente all'avvio del sistema è quello di `INITIALIZATION`, in cui è richiesto all'utente di inserire numero di run ed evento per poter avviare il caricamento dei dati. Eventuali errori vengono gestiti mediante lo stato `INIT_ERROR`, che mostra all'utente la causa del problema e gli consente di riprendere l'esecuzione dal punto in cui si era interrotta.

Una volta completato il caricamento dei dati e inizializzati i componenti necessari, il sistema passa allo stato `INTERACTION`. Qui avvengono le principali interazioni dell'utente con l'event display, tra cui il movimento della camera e il cambiamento delle opzioni di rendering. È inoltre possibile esportare un'immagine della visualizzazione corrente tramite lo stato `EXPORT_IMAGE`; al termine dell'esportazione, si ritorna automaticamente in interazione.

Durante l'interazione è possibile cambiare la run, l'evento o la geometria visualizzata nella scena: questo avviene grazie allo stato `CHANGE_DATA`, che permette all'utente di caricare nuovi dati senza riavviare il software. Come per l'inizializzazione, in caso di errori questi vengono gestiti nello stato `CHANGE_ERROR`.

4.2.1 Inizializzazione

`INITIALIZATION` è uno stato composto, il cui schema è riportato in Figura 4.3. Quando è necessario acquisire informazioni dall'utente, il processo viene suddiviso in due fasi: una fase di scelta (`CHOICE`), nella quale l'utente inserisce i dati richiesti, e una fase di caricamento (`LOAD`), nella quale il software elabora tali informazioni. Questo schema viene adottato, in particolare, per l'inserimento della run, dell'evento e del file di geometria.

In ogni momento l'utente può tornare alla schermata di inserimento precedente tramite l'evento `goBack`, attivato mediante la pressione di un pulsante.

Per una data run, il sistema tenta inizialmente di individuare e caricare il file di geometria predefinito. Qualora questa operazione dovesse fallire, ad esempio perché il file non esiste, viene mostrata all'utente una schermata di errore che consente di selezionare manualmente il file di geometria corretto.

Al termine dell'inizializzazione viene richiesto il caricamento della geometria al framework ROOT: questo avviene solo alla fine perché una volta caricata la geometria ROOT, non può più essere sostituita se non riavviando il software. Se la geometria venisse caricata quando l'utente ha ancora la possibilità di cambiare run, nel caso in cui le due geometrie non dovessero corrispondere ROOT genererebbe un errore.

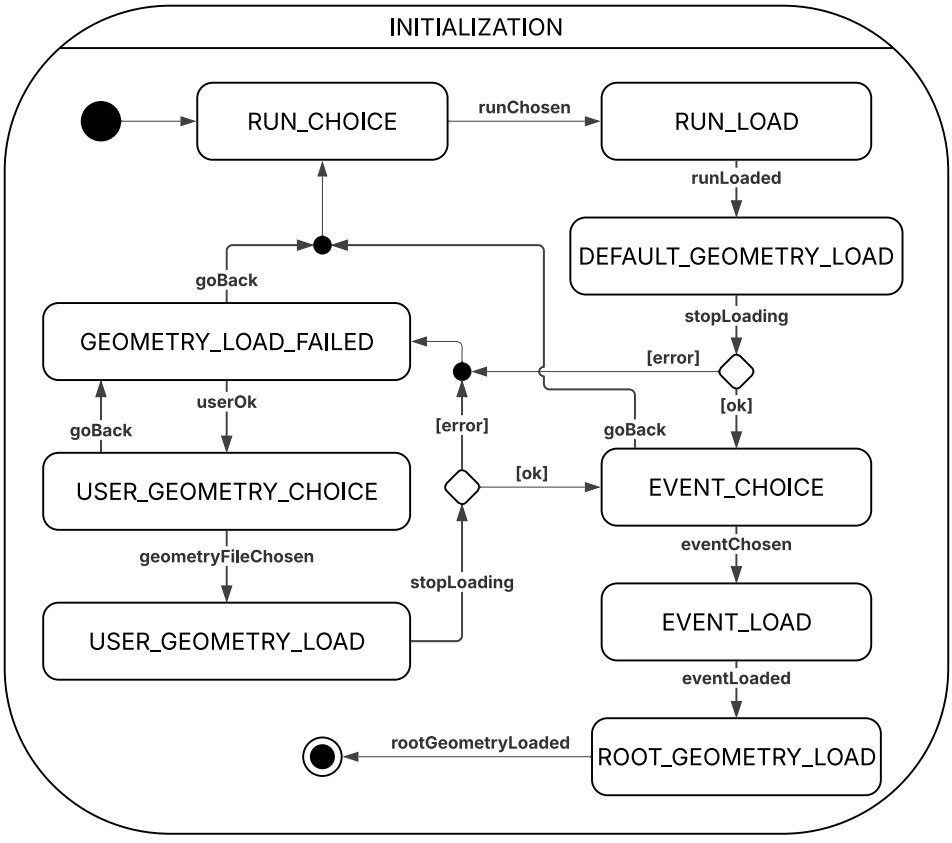


Figura 4.3: Espansione dello stato composto `INITIALIZATION`

In ogni caso, gli errori – come visto nello schema generale di Figura 4.2 – vengono gestiti tramite uno stato esterno detto `INIT_ERROR`, in cui viene mostrata all'utente la causa scatenante (ad esempio run number inesistente), così che possa tornare indietro e rieseguire l'azione richiesta.

4.2.2 Cambio dati

Per cambiare i dati visualizzati durante l'interazione si entra nello stato composto `CHANGE_DATA` (il cui diagramma è riportato in Figura 4.4), che rispecchia l'inizializzazione: anche in questo caso, gli stati sono sdoppiati in `CHOICE` e `LOAD`, e gli errori sono gestiti entrando nello stato esterno `CHANGE_ERROR`. In questa fase si può però scegliere tramite interfaccia se cambiare il file di geometria, il numero di run o l'evento. Poiché può essere necessario scorrere più eventi appartenenti alla stessa run, sarebbe poco efficiente richiedere ogni volta l'inserimento dell'intera coppia (run number, event number) e ricaricare dati già validi. Per questo motivo, il sistema consente sia di specificare nuovamente entrambi i valori, sia di modificare soltanto il numero di evento nel caso in cui la run rimanga invariata.

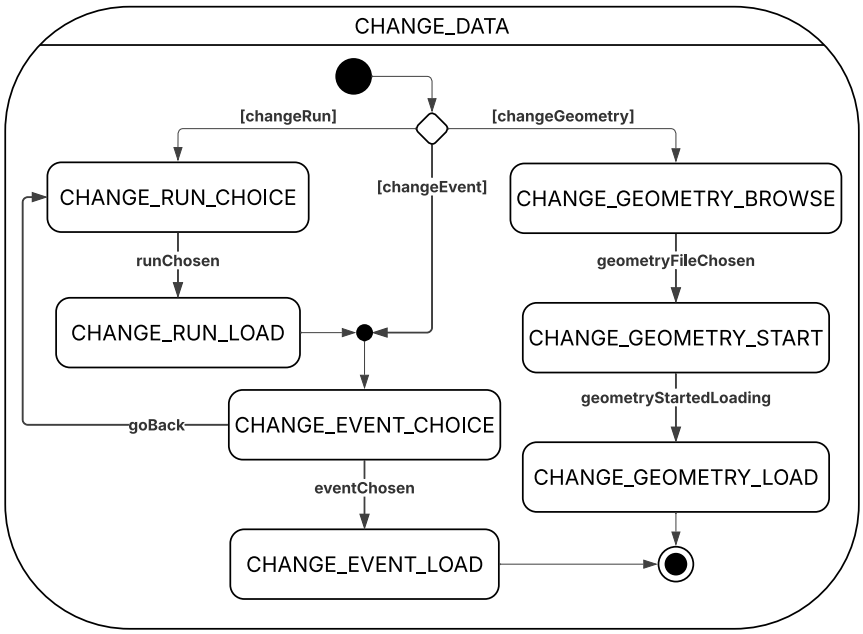


Figura 4.4: Espansione dello stato composto `CHANGE_DATA`

Il cambio della geometria è indipendente dal cambio di run ed evento: la collaborazione potrebbe infatti necessitare di caricare file di geometria 3D corrispondenti a configurazioni diverse del rivelatore, senza che questo implichi necessariamente un cambio dei dati fisici visualizzati.

4.3 Struttura dell'applicazione

Per garantire il corretto funzionamento del software è stata introdotta una classe principale, incaricata di coordinare i moduli fondamentali del sistema. La classe `App`, posta all'interno del package `core`, rappresenta il punto di ingresso dell'applicazione e ne orchestra l'intero ciclo di vita, aggregando i componenti principali descritti in Figura 4.5:

- `AppStateManager`: mantiene lo stato corrente della FSM descritta nella Sezione 4.2 e ne gestisce i passaggi di stato.
- `AppSettings`: raccoglie tutte le impostazioni configurabili dall'utente, condivise tra i vari componenti.
- `WindowManager`: gestisce il ciclo di vita della finestra.
- `Callbacks`: raccoglie le callback per la gestione degli input da tastiera e mouse, delegandoli ai componenti appropriati.
- `Gui`: gestisce l'interfaccia utente.
- `Scene`: contiene la scena tridimensionale, con la geometria del rivelatore e le hit dell'evento corrente.
- `SndswEventManager`: si interfaccia con il framework sndsw per il caricamento dei dati fisici.

Il metodo `run` contiene il render loop: cicla fino a che il software rimane in stato `RUNNING`, aggiornando tutti i componenti dell'app ed eseguendo il render di gui e scena tridimensionale.

Il metodo `update`, invocato ad ogni ciclo del render loop, funge da dispatcher della FSM: ad ogni frame legge lo stato corrente e delega l'o-

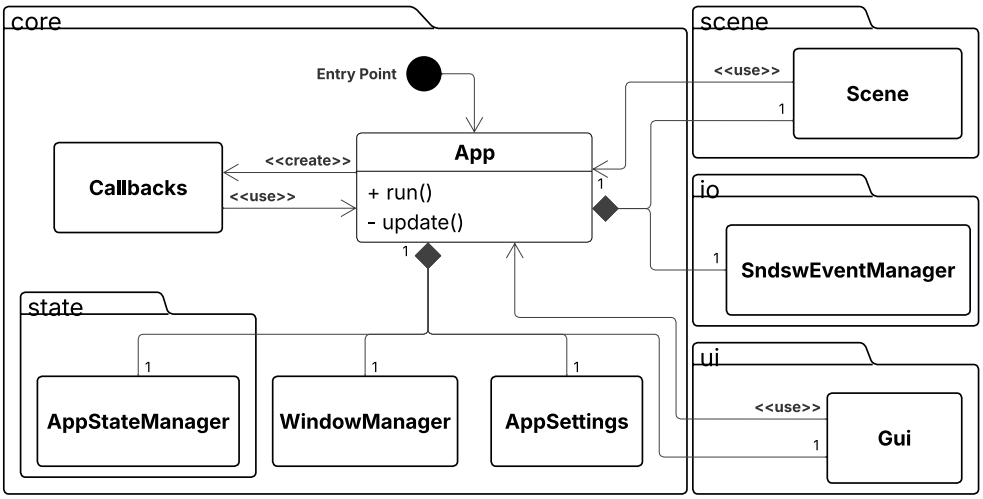


Figura 4.5: Dettaglio delle classi gestite da App

perazione corrispondente al componente appropriato – ad esempio il caricamento di una run a `SndswEventManager`. Gli errori sollevati durante il caricamento vengono catturati e notificati all'`AppStateManager`, che provvede a transitare nello stato di errore appropriato.

4.4 Design dettagliato

In questa sezione vengono approfondite le principali scelte progettuali del sistema, illustrando per ciascuna il problema affrontato e la soluzione adottata. La progettazione fa ampiamente uso dei *design pattern*: per ogni componente viene descritto il pattern adottato e il contesto specifico della sua applicazione.

4.4.1 Oggetti tridimensionali della scena

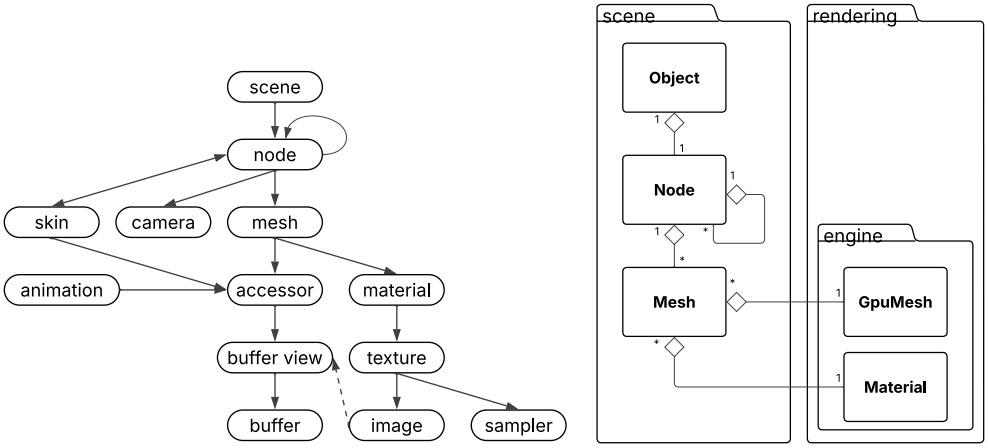
ROOT memorizza le geometrie dei rivelatori in formato proprietario. Poiché la conversione diretta di tali file avrebbe comportato un lavoro considerevole, non strettamente legato agli obiettivi della tesi, si è scelto di adottare una conversione preventiva nel formato glTF 2.0 tramite uno strumento dedicato. Questa scelta è giustificata anche dal fatto che

le geometrie disponibili sono in numero limitato e vengono aggiornate raramente, indicativamente una volta all'anno.

Lo standard glTF

Il *Graphics Library Transmission Format* (glTF) è uno standard open source per la condivisione di modelli e scene 3D, sotto forma di file di testo formattato in JSON. Organizza internamente i dati secondo una struttura ad albero (in Figura 4.6a è riportato l'albero dello standard glTF 2.0), di cui sono rilevanti i seguenti nodi ai fini di questo progetto:

- scene: contiene il puntatore al nodo radice dell'albero.
- node: generico nodo che contiene la matrice di trasformazione e opzionalmente una mesh e altri nodi figli.
- mesh: contenitore dei dati grezzi per il rendering, quali vertici, colori, normali e indici; eventualmente può anche avere un puntatore a un materiale.
- material: definisce le proprietà fisiche della superficie secondo il modello Physically Based Rendering.



(a) Specifica glTF 2.0 [33] (b) Oggetti della scena

Figura 4.6: Confronto tra la struttura dei file glTF e la rappresentazione degli oggetti della scena

Il *Physically Based Rendering* (PBR) è un approccio che simula il comportamento fisico della luce basandosi su parametri quali il colore di base, il coefficiente metallico (metallic) e quello di rugosità (roughness). Nel progetto, al contrario, il colore di base viene impiegato per creare un materiale – classe `Material` – secondo le componenti del modello di illuminazione di Blinn-Phong, discusso nella Sezione 3.4.2.

Struttura degli oggetti della scena

Seguendo il pattern *Composite*, sono state create le classi (illustrate in Figura 4.6b) che riflettono la struttura ad albero glTF per la memorizzazione dei componenti della scena:

- `Object` funge da componente astratto, ed è costituito dal solo nodo radice.
- `Node` rappresenta il componente composito, poiché può contenere altri nodi figli. Inoltre, mantiene il riferimento alle mesh associate al nodo e memorizza la matrice di modellazione corrispondente.
- `Mesh` è la foglia, in quanto rappresenta un elemento terminale della gerarchia privo di figli.

Poiché le geometrie possono essere condivise tra più nodi della scena, i dati delle mesh sono stati disaccoppiati dal concetto logico di mesh: un oggetto `Mesh` punta a una `GpuMesh` che può essere condivisa tra più istanze, evitando la duplicazione di dati geometrici in memoria GPU. Lo stesso ragionamento vale per `Material`. Questa soluzione corrisponde al pattern *Flyweight*, applicato per minimizzare il consumo di memoria nella rappresentazione di geometrie e materiali ricorrenti. Lo stato intrinseco è rappresentato dalle informazioni condivisibili (contenute in `GpuMesh` e `Material`), mentre lo stato estrinseco è dato dal contesto in cui l'oggetto viene utilizzato – ad esempio la matrice di modellazione contenuta in `Node`, che permette di posizionare e orientare la stessa `GpuMesh` in punti diversi della scena. Per garantire la condivisione dello stato intrinseco, la creazione degli oggetti `Object` è delegata alla classe `ObjectFactory`, che

svolge il ruolo di FlyweightFactory. Per le geometrie di uso più frequente – ad esempio il cubo, utilizzato per rappresentare ogni singola hit del rivelatore – la `GpuMesh` viene istanziata una sola volta alla creazione della factory e successivamente condivisa tra tutti gli `Object` richiesti, evitando di duplicare i dati geometrici in memoria GPU anche quando il numero di istanze nella scena è elevato.

4.4.2 Caricamento dei dati

Per la lettura dei dati sperimentali tramite il framework ROOT è necessario gestire un elevato numero di classi. In accordo con il pattern *Facade*, il controllo di tutte queste dipendenze è stato centralizzato nella classe `SndswEventManager`, in modo da avere un unico punto di accesso verso l'esterno del software; questo disaccoppiamento risulta evidente nella Figura 4.7.

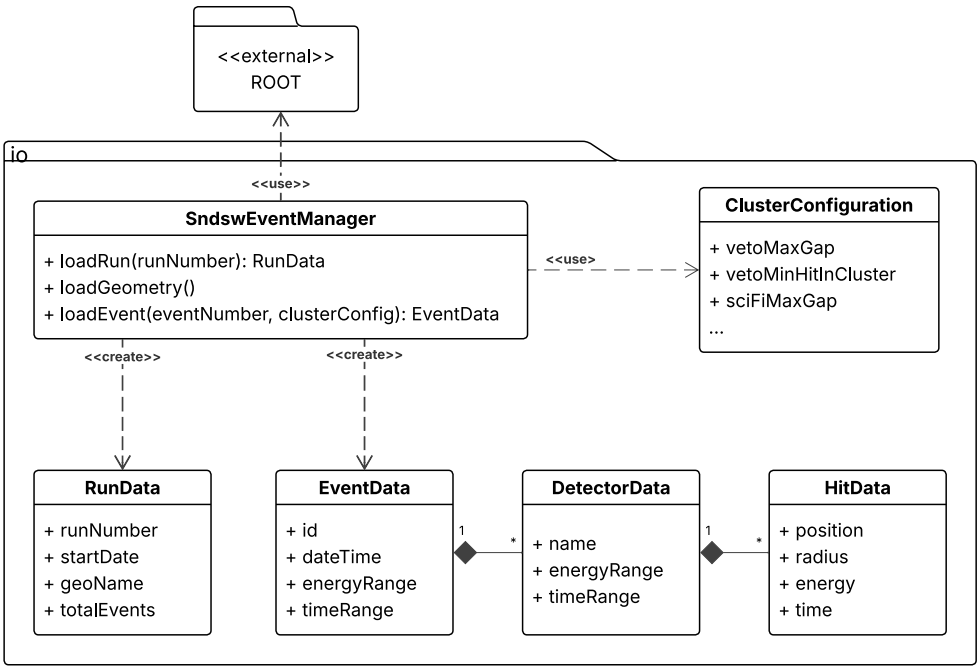


Figura 4.7: Diagramma delle classi coinvolte nella lettura dei dati dal framework ROOT

Comunicando il numero di run prescelto ne vengono lette le caratteristiche e salvate nella classe `RunData`, così che l'event display possa mostrare all'utente i dati della run attualmente in memoria.

Il caricamento di un evento avviene invece passando le configurazioni come parametro tramite la struttura `ClusterConfiguration`, restituendo il risultato in un oggetto della classe `EventData`. Come discusso nella Sezione 1.5, SND@LHC è composto da complessi sottorivelatori (classe `DetectorData`), ogni evento si compone delle hit (classe `HitData`) rilevate in ognuno di essi.

4.4.3 Scelta tipo di proiezione

Per consentire all'utente di selezionare il tipo di proiezione da utilizzare, il sistema prevede due classi distinte, dedicate rispettivamente alla proiezione ortografica e alla proiezione prospettica. Come mostrato nel capitolo precedente, entrambe seguono la stessa logica generale, ma differiscono nel modo in cui viene calcolata la matrice di proiezione. Questa struttura ha motivato l'adozione del pattern *Template Method*, che permette di definire uno schema comune delegando alle sottoclassi l'implementazione del calcolo specifico. Gli attori del pattern sono illustrati in Figura 4.8.

Per migliorare le performance dell'event display evitando il ricalcolo continuo della matrice di proiezione, ne viene tenuta una copia sovrascritta ad ogni cambio delle caratteristiche della proiezione. Poiché va ricalcolata solo al cambio dell'aspect ratio o del fov, i metodi template `setAspectRatio` e `setFov` definiti nella classe astratta `BasicProjection` richiamano il metodo astratto `computeProjectionMatrix`, ovvero l'operazione primitiva. Le classi concrete, in questo caso `OrthographicProjection` e `PerspectiveProjection`, si occupano solo di specificare l'algoritmo di calcolo della matrice nel metodo astratto sovrascritto, evitando di ripetere codice comune e semplificando l'aggiunta di nuovi tipi di proiezioni.

Infine, nella proiezione ortografica è presente un metodo aggiuntivo per impostare la distanza della camera dal target: dato che questo tipo

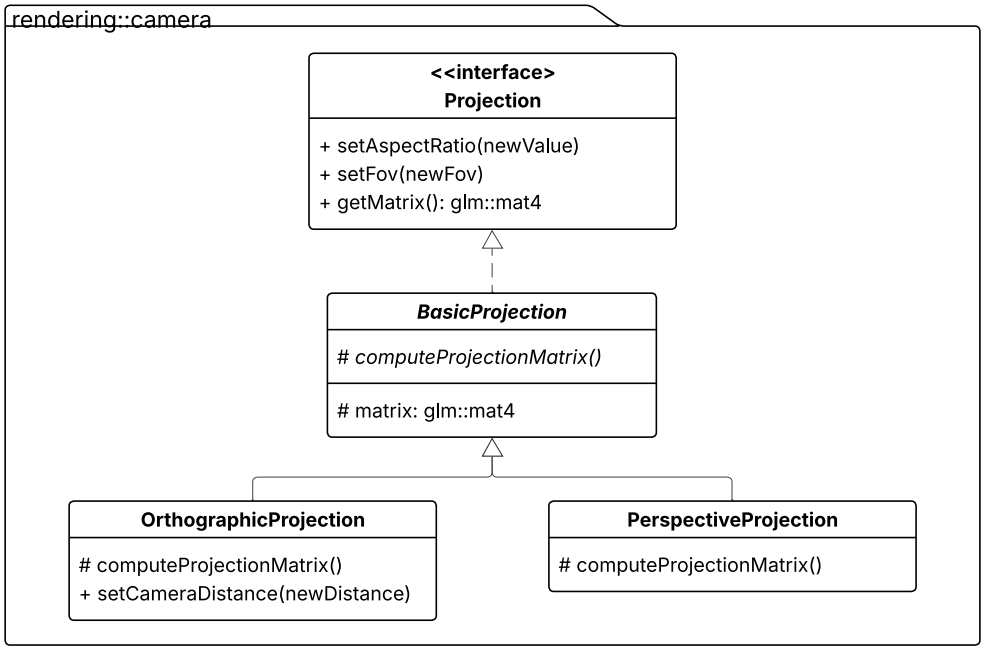


Figura 4.8: Diagramma delle classi per l'applicazione del pattern Template Method alle proiezioni

di proiezione preserva il parallelismo, il semplice avvicinamento della camera al target non produce l'effetto di zoom. Si rende quindi necessario cambiare le dimensioni dei piani vicino e lontano del volume di vista.

4.4.4 Colorazione hit

Per permettere all'utente di esplorare i dati da prospettive differenti, l'event display supporta diverse modalità di colorazione delle hit.

La determinazione del colore avviene sulla base di due criteri:

- la grandezza fisica di riferimento, come tempo ed energia.
- la scala da adottare, ad esempio monocromatica, lineare o logaritmica.

La classe `ColorPalette` utilizza infatti due istanze del pattern *Strategy* – illustrato in Figura 4.9 – in modo da disaccoppiare questa doppia variabilità senza violare il *Single Responsibility Principle* (SRP), ma centralizzando l'operazione di colorazione delle hit. Essa si occupa infatti di:

- calcolare il colore per ogni hit.
- restituire il range di valori della scala di riferimento, comprensivo dell'unità di misura.
- generare campionamenti del colore su intervalli fissati, ad esempio per visualizzare la legenda della scala cromatica associata all'intervallo di valori considerato.

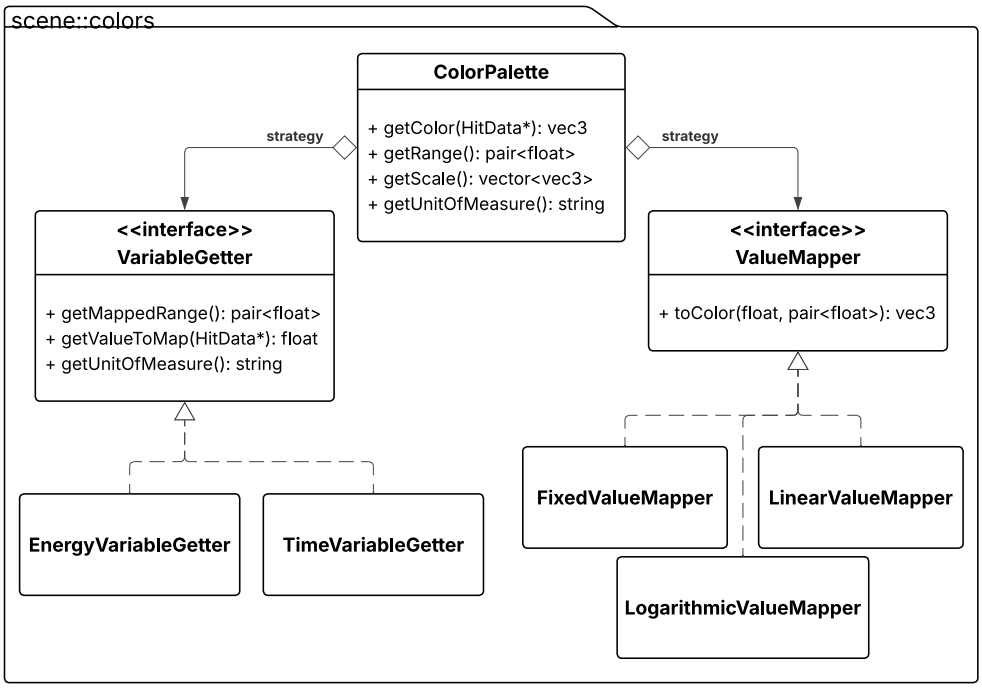


Figura 4.9: Diagramma delle classi che illustra l'uso del doppio pattern Strategy per la gestione dei colori dell'event display

`ColorPalette` svolge quindi il ruolo di classe contesto e utilizza due strategie di calcolo: `VariableGetter`, per selezionare la grandezza fisica di riferimento, e `ValueMapper`, per associare a una hit un colore in funzione del valore assunto all'interno di un determinato intervallo.

Nel sistema sono state implementate strategie concrete che consentono, tramite `VariableGetter`, di basare la colorazione sull'energia o sul tempo associati alla hit. Tramite `ValueMapper`, invece, sono disponibili

diverse modalità di costruzione della scala cromatica: monocromatica, lineare e logaritmica.

Questa struttura rende semplice l'estensione futura del sistema: nuovi criteri di colorazione potranno essere introdotti definendo ulteriori strategie concrete, senza modificare la logica interna di `ColorPalette`.

Le scelte progettuali descritte in questo capitolo costituiscono la base per l'implementazione del sistema. Nel capitolo successivo verrà quindi illustrato come tali scelte siano state tradotte in una versione completa e funzionante del nuovo event display.

Capitolo 5

Implementazione del software

Dopo aver introdotto i fondamenti teorici necessari alla comprensione della pipeline grafica e aver definito le principali scelte progettuali del sistema, è possibile passare all'implementazione dell'event display. In questo capitolo verranno illustrate le librerie utilizzate, alcuni passaggi algoritmici rilevanti accompagnati da porzioni di codice commentato, gli shader realizzati e le ottimizzazioni introdotte.

5.1 Tecnologie e strumenti

Si è scelto di implementare il nuovo event display in C++, linguaggio già ampiamente utilizzato in sndsw. Rispetto a Python – l'altro linguaggio principale del framework – C++ offre prestazioni superiori grazie alla compilazione nativa e alla gestione diretta della memoria, caratteristiche particolarmente rilevanti per un'applicazione di rendering interattivo. Inoltre, le principali librerie grafiche utilizzate nel progetto, descritte nel paragrafo successivo, espongono API native in C e C++.

5.1.1 Librerie utilizzate

Poiché il CERN promuove la filosofia del *Free and Open-Source Software* (FOSS), nella scelta delle librerie è stata posta particolare attenzione alle licenze adottate. In Tabella 5.1 sono riportate quelle utilizzate nel progetto.

Libreria	Licenza	Scopo
GLAD	MIT	Loader per le funzioni OpenGL
GLFW	zlib	Gestione finestra e input
GLM	MIT	Matematica vettoriale e matriciale
Assimp	BSD 3	Parsing file glTF
Dear ImGui	MIT	Interfaccia utente
ImGuiFileDialog	MIT	File browser
stb_image	MIT	Caricamento e salvataggio immagini

Tabella 5.1: Librerie utilizzate nel progetto

Le librerie GLAD e Dear ImGui sono state integrate direttamente nella cartella `external/`: nel caso di GLAD perché i file vengono generati tramite un apposito servizio web, nel caso di Dear ImGui perché gli stessi sviluppatori raccomandano l'inclusione diretta dei sorgenti. Le restanti dipendenze sono gestite tramite il modulo CMake `FetchContent`, che le scarica automaticamente durante la configurazione del progetto.

5.2 Caricamento dati sperimentali

Il caricamento dei dati tramite la classe `SndswEventManager` avviene in tre fasi, ognuna implementata in un apposito metodo: lettura della run, della geometria e dell'evento. Questo processo è orchestrato dalla FSM (descritta nella Sezione 4.2) tramite la classe `App`: i dati restituiti da `SndswEventManager` vengono quindi passati alla `Scene` per l'effettiva visualizzazione.

5.2.1 Caricamento run e geometria

All'interno del metodo `loadRun(runNumber)`, viene inizialmente estratto il nome del file della geometria richiesta. Questo approccio consente di verificare preventivamente la mancata corrispondenza con la geometria eventualmente già presente in memoria, sollevando, se necessario, l'eccezione personalizzata `GeometryMismatchException`. Successivamen-

te, l'interfaccia fa uso del framework sndsw per caricare la run all'interno di un oggetto `TChain`, la struttura dati nativa di ROOT preposta alla gestione dei dati distribuiti su più file. Dalla `TChain` vengono infine estratti i metadati necessari a istanziare l'oggetto `RunData`, restituito al chiamante del metodo. Nel Listato 5.1 è riportato lo scheletro del codice che esegue quanto spiegato.

```
RunData* SndswEventManager::loadRun(int64_t runNumber) {

  { ... } // Estrazione e validazione della geometria

  //Acquisizione della TChain ROOT tramite sndsw
  unique_ptr<TChain> newChain =
    snd::analysis_tools::GetTChain(runNumber);
  if (newChain->GetEntries() <= 0) throw runtime_error("Run empty.");
  this->chain = move(newChain);

  /* Setup dei branch ROOT sui vettori dell'esperimento
   * ed estrazione dei metadati da restituire */
  this->header = new SNDLHCEventHeader();
  { ... }
  this->loadedRun = runNumber;

  return new RunData(
    runNumber, date, fileName, this->chain->GetEntries()
  );
}
```

Listato 5.1: Integrazione con il framework sndsw ed estrazione dei dati della run in `SndswEventManager`

L'inizializzazione della geometria tramite il metodo `loadGeometry` avviene solo dopo il primo caricamento di una run. In questo modo vengono predisposti i sottorivelatori corrispondenti alla configurazione selezionata, che potranno poi essere popolati con i dati dei singoli eventi.

5.2.2 Caricamento e clustering degli eventi

Poiché la maggior parte dei sottorivelatori acquisisce dati in due dimensioni, mostrare graficamente ogni singola hit comporterebbe un inutile sovraccarico della scena: per evitare confusione, si è scelto di applicare algoritmi di clustering che raggruppano le attivazioni vicine. Per mantenere

il sistema flessibile, l'utente può comunque personalizzare i parametri di soglia che regolano la formazione di questi cluster.

Quando run e geometria sono stati entrambi inizializzati, è possibile procedere con il caricamento dell'evento utilizzando il metodo `loadEvent (eventNumber, clusterConfiguration)`. Dopo aver controllato la validità del numero, viene richiesto a ROOT di leggerne i dati dallo storage. Si procede poi a ciclare su tutti i piani di ogni sottorivelatore, invocando la funzione di clustering con le configurazioni richieste ed istanziando oggetti delle classi `DetectorData` e `HitData` che verranno restituiti all'interno di `EventData`.

Nel Listato 5.2 è riportato, a titolo di esempio, il processo di clustering per i piani SciFi: le hit vengono prima raggruppate in cluster tramite `ClustersPositions`, per poi essere tradotte in oggetti `HitData` e aggiunte al `DetectorData` corrispondente. Lo stesso schema si ripete per ciascun sottorivelatore.

```
/* ---- CLUSTER SU PIANI SCIFI ---- */
this->scifiPlanes = snd::analysis_tools::FillScifi(..);
vector<vector<snd::analysis_tools::Cluster>> scifiClusters(this->config->
    scifi_n_stations);
DetectorData *detector = new DetectorData("SciFi");
eventDataToReturn->addDetector(detector);

// Calcolo dei cluster
for (auto &p : this->scifiPlanes)
  if (isValid(p.GetStation() - 1)) {
    auto planeClusters =
      ClustersPositions(p.GetHits(), clustConf...);
    { ... } // Aggiunta cluster trovati al vettore
  }

// Aggiunta dei cluster al DetectorData
for (int i=0; i < this->config->scifi_n_stations; ++i)
  for (auto &c : scifiClusters[i])
    detector->addHit(new HitData(c.center.X(), ...));
```

Listato 5.2: Dettaglio di creazione e lettura dei dati dei cluster per i piani SciFi

5.3 Scena e camera

La classe `Scene` è il coordinatore degli oggetti da renderizzare: possiede il riferimento agli oggetti tridimensionali della geometria e delle hit, oltre alla camera e la proiezione.

5.3.1 Costruzione della scena

La costruzione degli oggetti di scena viene sempre avviata dal loop di rendering principale della classe `App` durante gli appositi stati di `LOAD`.

Caricamento geometria glTF

Per caricare la geometria, `App` richiede all'`AppStateManager` il percorso del file che ne contiene la mesh, indipendentemente dal fatto che si tratti della geometria predefinita o di quella selezionata dall'utente. Il percorso viene poi passato alla scena, tramite il metodo `Scene::loadGeometry (path)`: al suo interno la scena lo inoltra alla `ObjectFactory` per creare un `Object`, invocando il metodo `ObjectFactory::getFromFile(path)`.

La factory utilizza la libreria assimp per leggere dal file tutte le mesh e i materiali che contiene, creando le liste con gli oggetti `Material` e `GpuMesh` che verranno condivisi (in accordo con il pattern illustrato nella Sezione 4.4.1). Viene quindi invocato il costruttore di `Object` a cui sono passate la scena caricata da assimp e le due liste condivise. A partire da questi dati, viene costruito ricorsivamente l'albero dei nodi, associando a ciascun nodo le mesh e i materiali condivisi, come mostrato nel Listato 5.3.

Caricamento dati sperimentali

Nelle sezioni precedenti è stato illustrato il processo di acquisizione dei dati tramite il framework sndsw. Quando `App` ha ottenuto l'oggetto `EventData` da `SndswEventManager`, lo passa alla scena, affinché possa essere utilizzato per la creazione degli oggetti da visualizzare. Analogamente a quanto avviene per il caricamento della geometria, anche in questo caso viene utilizzata l'`ObjectFactory` che restituisce l'`Object` contenente tutte le hit. Tale oggetto viene costruito a partire dai dati contenuti in

```
Node::Node(const aiScene *_scene, aiNode *_node, vector<shared_ptr<
    GpuMesh>> &_meshes, vector<shared_ptr<Material>> &_materials)
{

  // Lettura dati del nodo corrente
  this->localModelMatrix = (mat4)_node->mTransformation;
  this->name = _node->mName;

  // Aggiunta mesh associate al nodo
  for (int i = 0; i < _node->mNumMeshes; i++) {
    int meshId = _node->mMeshes[i];
    aiMesh *assimpMesh = _scene->mMeshes[meshId];

    unique_ptr<Mesh> mesh = make_unique<Mesh>(
      assimpMesh->mName, _meshes[meshId]
    ); // Creazione mesh dai dati di assimp

    mesh->setMaterial(
      _materials[assimpMesh->mMaterialIndex]
    ); // Copia del materiale condiviso

    this->meshes.push_back(move(mesh));
  }

  // Creazione ricorsiva nodi figli
  for (int i = 0; i < _node->mNumChildren; i++)
    this->childrenNode.push_back(make_unique<Node>(
      _scene, _node->mChildren[i], _meshes, _materials
    ));
}
```

Listato 5.3: Creazione ricorsiva dei nodi utilizzando la libreria assimp

`EventData` e dalla `ColorPalette` selezionata, utilizzata per determinare il colore associato a ciascuna hit.

La factory costruisce un albero di nodi il cui primo livello contiene un nodo per ciascun sottorivelatore, associando ogni hit a una `Mesh`. Questa suddivisione rende agevole la navigazione dell'albero dei nodi tramite interfaccia grafica. Per rappresentare le hit viene utilizzata la `GpuMesh` del cubo – istanziata una sola volta nella factory – e condivisa tra tutte le istanze delle `Mesh`, ciascuna posizionata e scalata tramite la propria matrice di modellazione in base a posizione e raggio contenuti in `HitData`.

5.3.2 Colorazione delle hit

Durante il caricamento dell'evento, il metodo `Scene::setEvent` provvede a creare la palette che definirà i colori delle hit. Sulla base delle impostazioni scelte dall'utente tramite l'interfaccia grafica, verranno istanziati i due oggetti strategy `VariableGetter` e `ValueMapper`, utilizzati per la creazione della `ColorPalette` che viene poi passata alla `ObjectFactory`.

La factory, durante la creazione di ogni mesh, non dovrà fare altro che assegnarle il colore restituito dal metodo `getColor` della `ColorPalette` ricevuta come parametro. L'attribuzione del colore avviene tramite la creazione di un oggetto `Material`: esso consente infatti di creare un materiale per il modello di illuminazione di Blinn-Phong a partire dal colore base. Nel Listato 5.4 è riportato come viene creato un materiale a cui si assegna un'importante componente diffusiva, con dei riflessi tendenti al bianco. La bassa shininess utilizzata produce riflessi diffusi su una vasta area anziché concentrati in un unico punto, tipici di una superficie leggermente opaca.

```
1 Material::Material(vec3 color) {
2   this->baseColor = color;
3   this->ambient = vec4(color * 0.5f, 1.0f);
4   this->diffuse = vec4(color * 0.8f, 1.0f);
5   this->specular = vec4(color * 0.3f + vec3(0.3f), 1.0f);
6   this->shininess = 32.0f;
7 }
```

Listato 5.4: Creazione del materiale per il modello di illuminazione di Blinn-Phong a partire da un colore base

5.3.3 Camera e proiezioni

La gestione della camera e della proiezione da utilizzare è centralizzata nella classe `ViewPort`. Essa contiene infatti un oggetto `Camera` e un oggetto `Projection`, e fornisce quindi le matrici di vista e di proiezione necessarie al rendering della scena. Per mantenerle aggiornate espone il metodo `ViewPort::update()`, da richiamare a ogni esecuzione del ciclo di rende-

ring, in cui aggiorna la proiezione in caso di variazione delle dimensioni della finestra e sposta la camera in caso di interazione dell'utente.

Camera

La camera utilizzata nel progetto segue il modello look at illustrato nella Sezione 3.3.2: i suoi attributi sono quindi la posizione, il target osservato e il versore che definisce l'up del mondo, da cui è possibile ricavare tutti gli altri vettori necessari. Ogni tipo di utilizzo della camera si traduce in operazioni geometriche su questi tre attributi: si riporta l'esempio dello spostamento parallelo nel Listato 5.5, ovvero quello utilizzato per spostare la camera e il target lungo il piano individuato dai versori up e right, preservandone la distanza reciproca.

```
// I delta sono lo spostamento del mouse lungo i due assi
void Camera::moveParallel(float deltaX, float deltaY) {

  // Modifica lo spostamento proporzionalmente alla distanza
  float distance = glm::length(this->target - this->position);
  deltaX *= constants::factors::PAN_SPEED * distance;
  deltaY *= constants::factors::PAN_SPEED * distance;

  // Calcolo dei vettori che identificano il piano di spostamento
  vec3 direction = glm::normalize(this->target - this->position);
  vec3 right = glm::normalize(glm::cross(direction, this->worldUp));
  vec3 up = glm::normalize(glm::cross(right, direction));

  // Applica lo spostamento e ricalcola la matrice di vista
  this->position += up * deltaY + right * deltaX;
  this->target += up * deltaY + right * deltaX;
  this->recomputeMatrix();
}
```

Listato 5.5: Operazioni che permettono lo spostamento della camera lungo il piano parallelo alla vista corrente

Per evitare di ricalcolare la matrice di vista a ogni iterazione del ciclo di rendering, ne viene salvata una copia aggiornata. Tale matrice viene ricalcolata solo quando cambiano i parametri della camera: i metodi di interazione modificano infatti posizione, target e worldUp, invocando poi `Camera::recomputeMatrix()`, che ricostruisce la matrice di vista tramite la funzione `lookAt(position, target, worldUp)` della libreria glm.

Proiezioni

Come discusso nella Sezione 3.3.3, il calcolo della matrice di proiezione dipende dal tipo di proiezione adottato, ortografica o prospettica: le classi di implementazione sono rispettivamente `OrthographicProjection` e `PerspectiveProjection`. La classe `BasicProjection` utilizza un meccanismo di caching analogo a quello adottato per la camera: la matrice di proiezione viene memorizzata e ricalcolata solo quando cambiano le dimensioni della finestra. In entrambi i casi vengono utilizzate le funzioni messe a disposizione dalla libreria glm – `ortho` e `perspective` – che necessitano però di informazioni diverse. Mentre la prospettica richiede il campo di vista, il rapporto d'aspetto della finestra e la distanza dai due piani vicino e lontano, per l'ortografica sono necessarie le dimensioni dei due piani e la loro distanza dal punto di vista. In quest'ultimo caso, l'estensione di tali superfici è calcolata sulla base del campo visivo e della distanza dal target, così da simulare l'effetto di zoom anche per la proiezione ortografica. Nel Listato 5.6 è riportata l'implementazione per i due tipi di proiezione.

```
1  void PerspectiveProjection::computeProjectionMatrix() {
2    this->matrix = glm::perspective(glm::radians(this->fovY), this->
         aspectRatio, this->nearPlane, this->farPlane);
3  }
4
5  void OrthographicProjection::computeProjectionMatrix() {
6    // Calcolo dimensioni piano di vista
7    float halfHeight =
8      this->distance * std::tan(glm::radians(this->fovY * 0.5f));
9    float halfWidth = this->aspectRatio * halfHeight;
10
11   this->matrix = glm::ortho(-halfWidth, halfWidth, -halfHeight,
         halfHeight, this->nearPlane, this->farPlane);
12 }
```

Listato 5.6: Calcolo delle matrici di proiezione per entrambi i tipi previsti

5.4 Motore di rendering

I dati da renderizzare sono memorizzati in appositi buffer, i cui puntatori sono salvati nella classe `GpuMesh`. Prima di eseguire la chiamata ad OpenGL per il rendering delle primitive contenute in un buffer, è necessario selezionare gli shader da utilizzare e fornire loro i dati aggiuntivi richiesti.

Nel software sviluppato, i sorgenti degli shader vengono compilati dalla classe `ShaderMaker`: da quel momento risiederanno quindi nella GPU associati a un *program ID*, memorizzato come attributo della classe `Shader`. Il rendering avviene quindi in tre fasi:

- attivazione di uno shader.
- passaggio delle variabili allo shader.
- chiamata di render OpenGL su un buffer.

Dal momento che ogni chiamata alle API grafiche OpenGL ha un costo non indifferente, verranno illustrati nelle sezioni successive la gestione dei buffer della GPU e le ottimizzazioni introdotte per ridurre tali chiamate.

5.4.1 Gestione delle mesh sulla GPU

Per ogni vertice da visualizzare è necessario memorizzare le informazioni associate – posizione, colore e vettore normale – ciascuna salvata in un *Vertex Buffer Object* (VBO) identificato da un indice detto *layer*, così da renderle accessibili agli shader. Per accorpare un insieme di VBO in un unico oggetto, riducendo le chiamate al driver grafico, viene utilizzato il *Vertex Array Object* (VAO).

Molti vertici sono inoltre condivisi dai triangoli in cui vengono suddivise le superfici: invece di duplicarne le informazioni, a ciascuno viene associato un indice. La lista degli indici da utilizzare per il rendering viene memorizzata in un *Element Buffer Object* (EBO), anch'esso contenuto nel VAO. Per eseguire il rendering di una geometria è quindi sufficiente attivare il VAO e richiedere il disegno degli indici memorizzati nell'EBO. Il processo di creazione di questi buffer e la sequenza di rendering sono

riportati nel Listato 5.7; poiché si lavora con triangoli, la `renderMode` utilizzata è sempre `GL_TRIANGLES`.

```
GpuMesh::GpuMesh(vector<vec3>& vertices, vector<vec4>& colors, vector<
    vec3>& normals, vector<GLuint>& indices, vec3 anchorPosition) {

  { ... } // Controllo che le dimensioni dei vettori siano coerenti

  // Generazione e attivazione VAO
  glGenVertexArrays(1, &this->vao);
  glBindVertexArray(this->vao);

  // Generazione, attivazione e popolamento del VBO dei vertici
  glGenBuffers(1, &this->vboVertices);
  glBindBuffer(GL_ARRAY_BUFFER, this->vboVertices);
  glBufferData(GL_ARRAY_BUFFER, ..., vertices.data(), ...);

  // Caricamento del VBO dei vertici nel layer richiesto
  glVertexAttribPointer(VERTICES_LAYER, 3, GL_FLOAT, ...);
  glEnableVertexAttribArray(VERTICES_LAYER);

  { ... } // Stessa cosa per i VBO di colori e normali

  // Generazione, attivazione e riempimento EBO
  glGenBuffers(1, &this->eboIndices);
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->eboIndices);
  glBufferData(GL_ELEMENT_ARRAY_BUFFER, ..., indices.data(), ...);

  { ... } // Unbind dei buffer
}

void GpuMesh::render(bool showAnchor) {
  glBindVertexArray(this->vao); // Bind del VAO
  glDrawElements(this->renderMode, this->numIndices, ...); // Render
}
```

Listato 5.7: Istruzioni OpenGL per la creazione di VAO, VBO ed EBO, e per eseguirne il rendering

5.4.2 Ottimizzazioni

Per rendere l'interazione con l'utente il più fluida possibile, sono state introdotte alcune ottimizzazioni: la prima punta a diminuire il peso della navigazione dell'albero degli oggetti e dei nodi della scena, la seconda,

invece, ha l'obiettivo di minimizzare il numero di chiamate al driver grafico durante la gestione degli shader.

Precalcolo delle matrici di modellazione

La struttura del file glTF prevede che ogni nodo trasformi i suoi figli applicandogli la propria matrice di modellazione. La trasformazione finale di una mesh è quindi data dalla moltiplicazione successiva di tutte le matrici dei nodi da cui discende.

Dal momento che il software non prevede la modifica interattiva di specifici nodi della geometria, la matrice di ogni mesh rimane invariata per tutto il ciclo di vita dell'oggetto. Per questo motivo, si è scelto di precalcolare la matrice di modellazione finale al momento della creazione dell'oggetto. `Object` innesca il calcolo ricorsivo con il metodo `updateGlobalModelMatrix` del suo nodo radice, così che la trasformazione finale venga salvata come attributo degli oggetti `Mesh`. La funzione ricorsiva dei nodi è riportata nel Listato 5.8.

```
1  void Node::updateGlobalModelMatrix(const mat4 &parentModelMatrix) {
2    // Composizione matrice di modellazione globale
3    this->globalModelMatrix = parentModelMatrix * this->localModelMatrix;
4
5    // Chiamata ricorsiva sui nodi figli
6    for (auto &node : this->childrenNode)
7      node->updateGlobalModelMatrix(this->globalModelMatrix);
8
9    // Salvataggio matrice calcolata nelle mesh
10   for (auto &mesh : this->meshes)
11     mesh->updateGlobalModelMatrix(this->globalModelMatrix);
12 }
```

Listato 5.8: Sistema di aggiornamento ricorsivo della cache della matrice di modellazione per le `Mesh`

Riduzione delle chiamate per gli shader

Le variabili esterne richieste dagli shader in esecuzione sulla GPU, e fornite dall'applicazione, sono dette *uniform*. Inizialmente era previsto che, per ogni mesh, venissero impostate singolarmente tutte le uniform necessarie prima di ciascuna chiamata di rendering.

Tuttavia, OpenGL si basa su una macchina a stati: una volta selezionato uno shader e assegnati i valori delle relative uniform, questi rimangono validi per tutte le chiamate di rendering successive, fino alla loro prossima modifica o al cambio di shader attivo. Questa osservazione permette di dividere le uniform in due gruppi, in base alla frequenza con cui il loro valore cambia effettivamente nel corso di un frame:

- le globali sono quelle che restano invariate per l'intero ciclo di rendering di un frame, come le matrici di vista e proiezione, e le proprietà delle luci.
- le locali sono invece quelle che cambiano per ogni mesh renderizzata, come la matrice di modellazione e il materiale della superficie.

Ne consegue che assegnare nuovamente le uniform globali per ciascuna mesh sia un'operazione superflua, dato che il loro valore non cambia tra una chiamata di rendering e la successiva. Il metodo per passare le uniform allo shader è stato quindi diviso in due: `bindGlobalUniforms`, invocato una sola volta per ogni oggetto subito dopo l'attivazione del relativo shader, e `bindLocalUniforms`, invocato invece per ciascuna mesh prima della rispettiva chiamata di rendering. Questa separazione permette di ridurre considerevolmente il numero di chiamate al driver grafico necessarie a ogni frame.

5.5 Shading: illuminazione e trasparenza

In un *event display*, i segnali dei rivelatori devono essere visibili al loro interno: diventa quindi fondamentale rendere trasparenti le geometrie esterne, che altrimenti occulterebbero le informazioni in esse contenute.

Inoltre, l'inserimento di un sistema di illuminazione è essenziale per migliorare la percezione della profondità e della forma degli oggetti, evitando l'effetto piatto prodotto dall'applicazione di un colore uniforme alle superfici. Il modello di illuminazione adottato è il modello di Blinn-Phong, descritto in Sezione 3.4.2, che offre un buon compromesso tra realismo visivo e semplicità computazionale, adatto a un'applicazione interattiva come un event display.

Nel progetto sono stati sviluppati diversi shader per gestire i differenti elementi della scena; tuttavia, nei prossimi paragrafi verrà presentato il codice dello shader più completo, che integra al suo interno sia la logica per il calcolo dell'illuminazione sia la gestione della trasparenza.

La tecnica di shading adottata è quella di Phong: come si vedrà nelle sezioni successive, l'applicazione del modello di illuminazione avviene all'interno del fragment shader, per ogni frammento. Questa scelta consente di calcolare correttamente i riflessi speculari: adottando il Gouraud shading, ad esempio, questi andrebbero persi qualora ricadessero all'interno di un poligono senza colpirne i vertici, circostanza frequente nel caso di mesh composte da pochi vertici distanti tra loro, come i parallelepipedi utilizzati per rappresentare le hit (vedi Sezione 3.4.3).

5.5.1 Vertex shader

All'interno del vertex shader, ovvero quello eseguito per ogni vertice presente nel VBO, vengono preparati i dati che saranno utilizzati successivamente per il calcolo del colore. I vettori richiesti dal modello di illuminazione sono stati inseriti nella struttura `IlluminationData`, in modo che contenga i vettori normali (N), di vista (V) e la direzione di ogni luce (L_i). Tutti i calcoli per l'illuminazione sono eseguiti in VCS, così che il calcolo di $V = Pos_{camera} - Pos_{vertice}$ si semplifica in $V = -Pos_{vertice}$ dal momento che la camera è l'origine del VCS.

Nel Listato 5.9, è riportato il codice del vertex shader, in cui:

- `aPos` e `vertexNormal` sono le coordinate e la normale del vertice in OCS prese dai VBO.

- `Model`, `View`, `Projection`, `lights[]` e `numLights` sono le matrici di trasformazione e le informazioni delle luci passate dall'applicazione come uniform.

- `vLocalPos` e `vIlluminationData` sono le variabili di output verso il geometry shader, contenenti rispettivamente la posizione nelle coordinate OCS del vertice (usata per la trasparenza) e i vettori d'illuminazione calcolati in VCS.

È importante notare che, nel calcolo del vettore N in coordinate VCS, non viene applicata direttamente la matrice di trasformazione $V \cdot M$. Al contrario, viene calcolata la trasposta dell'inversa della sua sottomatrice 3×3 superiore sinistra, ovvero $((V \cdot M)_{3\times3})^{-T}$. Questo accorgimento si rende necessario poiché, in presenza di trasformazioni di scala non uniforme, l'applicazione diretta della matrice di modellazione e vista farebbe perdere la perpendicolarità del vettore normale rispetto alla superficie, compromettendo la correttezza dei successivi calcoli di illuminazione.

```
void main() {
  vLocalPos = aPos;

  // Posizione del vertice in Clip space
  gl_Position = Projection * View * Model * vec4(aPos, 1.0);

  // Posizione del vertice in VCS
  vec4 eyePosition = View * Model * vec4(aPos, 1.0);

  // Vettore normale e di vista del vertice in VCS
  vIlluminationData.N =
    normalize(transpose(inverse(mat3(View * Model))) * vertexNormal);
  vIlluminationData.V = normalize(-eyePosition.xyz);

  for (int i = 0; i < numLights; i++) {
    // Vettore che dal vertice va alla luce in VCS
    vec4 eyeLightPos = View * vec4(lights[i].position, 1.0);
    vIlluminationData.L[i] = normalize((eyeLightPos -eyePosition).xyz);
  }
}
```

Listato 5.9: Vertex shader: trasformazione del vertice in Clip Space e calcolo dei vettori d'illuminazione in VCS

5.5.2 Geometry shader

Il calcolo della trasparenza viene effettuato rendendo un frammento tanto più trasparente quanto maggiore è la sua distanza dai lati del triangolo. Questa operazione risulta più semplice disponendo delle coordinate baricentriche del frammento, introdotte nella Sezione 3.1.2.

È importante notare che nel passaggio dal geometry al fragment shader (o dal vertex nel caso il geometry non sia utilizzato), la GPU interpola automaticamente tutte le variabili di output dello shader, così che i valori definiti sui vertici della primitiva vengono interpolati per ciascun frammento generato dalla rasterizzazione.

Nel geometry shader è disponibile la primitiva geometrica completa, ossia tutti e tre i vertici del triangolo. Per generare le coordinate baricentriche è sufficiente impostare a 1 una componente diversa di un vettore $\in \mathbb{R}^3$ per ciascun vertice: al primo vertice viene assegnato $(1, 0, 0)$, al secondo $(0, 1, 0)$ e al terzo $(0, 0, 1)$. Grazie all'interpolazione automatica eseguita dalla GPU durante la rasterizzazione, ogni frammento riceve così le proprie coordinate baricentriche rispetto ai vertici del triangolo di appartenenza. Gli altri attributi associati ai vertici vengono inoltrati senza modifiche, in modo che il rasterizzatore possa interpolarli correttamente sui frammenti generati. Il procedimento descritto è riportato nel Listato 5.10.

5.5.3 Fragment shader

Il fragment shader ha il compito di determinare il colore finale di ciascun frammento della primitiva. Nell'implementazione proposta, il calcolo avviene in due fasi: l'applicazione del modello di illuminazione di Blinn-Phong per trovare il colore di base del frammento, e l'utilizzo delle coordinate baricentriche per la trasparenza.

Applicazione del modello di Blinn-Phong

Dal momento che i vettori necessari al modello di illuminazione sono già forniti dagli shader precedenti, i calcoli eseguiti dallo shader nel Li-

```
void main() {
  for (int i = 0; i < 3; i++) {
    // Assegna una coordinata baricentrica diversa per ogni vertice
    barycentric = vec3(0.0);
    barycentric[i] = 1.0;

    // Riporta questi valori invariati
    localPos = vLocalPos[i];
    gl_Position = gl_in[i].gl_Position;
    illuminationData = vIlluminationData[i];

    EmitVertex();
  }
  EndPrimitive();
}
```

Listato 5.10: Geometry shader: assegnazione delle coordinate baricentriche e inoltro dei dati al fragment shader

stato 5.11 corrispondono direttamente all'Eq. (3.2) del modello di Blinn-Phong. Il fattore di attenuazione non viene calcolato, in quanto non apporterebbe un contributo visivo significativo per un event display. La correzione della shininess per il fattore 4 è utilizzata per compensare il minore angolo β prodotto da Blinn-Phong, come evidenziato nella Sezione 3.4.2.

Calcolo della trasparenza

La trasparenza viene calcolata sfruttando le coordinate baricentriche interpolate: queste valgono 0 sul lato opposto al vertice corrispondente e 1 sul vertice stesso, quindi un frammento vicino a un bordo avrà almeno una coordinata baricentrica prossima a zero.

`fwidth` calcola la derivata di variabili interpolate dal rasterizzatore: il valore restituito è proporzionale alla velocità di variazione della variabile tra frammenti adiacenti. Il risultato sulle coordinate baricentriche è grande vicino ai bordi, dove le coordinate variano rapidamente, e piccolo all'interno del triangolo.

`smoothstep(edge0, edge1, x)` restituisce 0 se $x <$ `edge0`, 1 se $x >$ `edge1`, e, per valori intermedi, un'interpolazione cubica regolare tra i due estremi. Nel caso in esame, la funzione viene applicata alle coordinate

```
// Normalizzazione dei vettori interpolati
vec3 N = normalize(illuminationData.N);
vec3 V = normalize(illuminationData.V);

// ----- COMPONENTE AMBIENTALE -----
vec3 ambient = uAmbientLightIntensity * material.ambient;

// Accumulatore delle componenti del modello di illuminazione
vec3 baseColor = ambient;

for (int i = 0; i < numLights; i++) {
  vec3 L = normalize(illuminationData.L[i]);
  vec3 H = normalize(L + V);

  // ----- COMPONENTE DIFFUSIVA -----
  float cos_theta = max(dot(L, N), 0);
  vec3 diffuse = lights[i].color * cos_theta * material.diffuse;

  // ----- COMPONENTE SPECULARE -----
  float cos_alfa = pow(max(dot(H, N), 0), material.shininess * 4);
  vec3 specular = lights[i].color * cos_alfa * material.specular;

  // Accumula il contributo dato dalla luce corrente
  baseColor += lights[i].power * (diffuse + specular);
}
```

Listato 5.11: Fragment shader: applicazione del modello di illuminazione di Blinn-Phong

baricentriche del frammento, utilizzando $\mathtt{d} \cdot \mathtt{uEdgeThickness}$ come soglia superiore: `uEdgeThickness` è una variabile scelta dall'utente passata come uniform, e serve per aumentare il numero di frammenti considerati vicini al bordo. In prossimità dei lati del triangolo, almeno una coordinata baricentrica assume un valore piccolo; di conseguenza il frammento ricade sotto la soglia e il valore restituito da `smoothstep` tende a 0. All'interno del triangolo, invece, le coordinate baricentriche risultano maggiori e il valore restituito tende a 1. In questo modo è possibile distinguere gradualmente i frammenti vicini ai bordi da quelli più interni.

Poiché per ciascun frammento è disponibile un valore che ne esprime la vicinanza a ciascun lato del triangolo, il valore `edgeFactor` viene ottenuto prendendo il minimo tra le tre componenti calcolate. In questo modo,

`edgeFactor` assume valori prossimi a 0 quando il frammento si trova vicino ad almeno uno dei bordi del triangolo, e valori prossimi a 1 quando si trova nella parte interna della faccia.

Infine, `mix` interpola tra `uEdgeAlpha` e `uFaceAlpha` in base al valore di `edgeFactor`, producendo il valore finale del canale alpha del frammento: anche `uEdgeAlpha` e `uFaceAlpha` sono due uniform, che corrispondono al canale alpha richiesto ai bordi e all'interno delle facce del triangolo.

Assemblando il colore calcolato con il modello di illuminazione, salvato in `baseColor`, e la trasparenza appena calcolata in `finalAlpha`, si ottiene il colore finale prodotto come output dello shader per il frammento. Il calcolo completo è riportato nel Listato 5.12.

```
// Derivata delle coordinate baricentriche
vec3 d = fwidth(barycentric);

// Lontananza dai bordi del triangolo per ogni componente
vec3 a3 = smoothstep(vec3(0.0), d * uEdgeThickness, barycentric);

// 0 se vicino a qualsiasi bordo
float edgeFactor = min(min(a3.x, a3.y), a3.z);

// Interpola tra alpha del bordo e alpha della faccia
float finalAlpha = mix(uEdgeAlpha, uFaceAlpha, edgeFactor);

// Colore finale del frammento
FragColor = vec4(vec3(baseColor), finalAlpha);
```

Listato 5.12: Fragment shader: calcolo della trasparenza e produzione del colore finale del frammento

5.5.4 Trasparenza e algoritmo del pittore

Per ottenere una gestione corretta della trasparenza sono necessari due accorgimenti distinti, oltre al calcolo del canale alpha nello shader.

Il primo consiste nel disabilitare la scrittura nel depth buffer per le mesh trasparenti. Se una superficie semitrasparente scrivesse i propri valori di profondità, le mesh opache poste dietro di essa fallirebbero il depth test e verrebbero scartate, risultando nascoste anche quando dovrebbero essere visibili attraverso la superficie semitrasparente.

Il secondo accorgimento consiste nell'abilitare il blending, che consente alla GPU di miscelare il colore del frammento in arrivo con quello già presente nel color buffer; nel software è stata utilizzata la funzione standard `GL_ONE_MINUS_SRC_ALPHA` (già vista in Eq. (3.1)). È tuttavia importante osservare che il blending non è commutativo: l'ordine in cui i frammenti vengono miscelati influenza il colore finale, poiché il colore già nel buffer viene usato come destinazione e quello del frammento in arrivo come sorgente.

Per risolvere questo problema è stato implementato l'algoritmo del pittore (*painter's algorithm*), che consiste nel disegnare le geometrie dalla più lontana alla più vicina, così che si coprano correttamente. A ogni ciclo di rendering, se la camera ha subito uno spostamento, le mesh vengono riordinate in base alla distanza dell'osservatore dal proprio punto di ancoraggio (`anchorPosition`), calcolato come il centro della mesh.

Per poter ordinare le mesh di un oggetto, è stata aggiunta una lista di `Mesh` all'interno di `Object`, contenente tutte le foglie dell'albero dei nodi, così da renderne possibile l'ordinamento senza dover visitare ricorsivamente l'albero a ogni frame; si può quindi eseguire il render seguendo la lista tramite il metodo `Object::renderBuffered`. Nel Listato 5.13 è riportata l'implementazione del metodo `Object::sortMeshes`, che riceve la posizione dell'osservatore e riordina la lista delle mesh usando `std::sort`. Per efficienza, il confronto avviene utilizzando `glm::distance2`, che calcola il quadrato della distanza evitando la radice quadrata, non necessaria ai fini del confronto.

```
1 void Object::sortMeshes(glm::vec3 point) {
2   std::sort(this->meshes.begin(), this->meshes.end(),
3     [point](Mesh* a, Mesh* b) {
4       float distA = glm::distance2(a->getAnchor(), point);
5       float distB = glm::distance2(b->getAnchor(), point);
6       return distA > distB;
7     });
8 }
```

Listato 5.13: Ordinamento delle mesh per distanza dal punto scelto

5.6 Interfaccia e controlli utente

L'utente può interagire con l'applicazione principalmente mediante due modalità, a seconda dell'operazione richiesta: l'interfaccia grafica a finestre prodotta con la libreria Dear ImGui o l'interazione diretta con la scena attraverso scorciatoie da tastiera e trascinamento del mouse sull'area di disegno.

5.6.1 Interfaccia grafica con Dear ImGui

L'interfaccia grafica è realizzata tramite la libreria Dear ImGui, una *immediate mode GUI* in cui l'intera interfaccia viene ridisegnata a ogni frame a partire dallo stato dell'applicazione. È gestita interamente dalla classe `Ui`, che ha a disposizione il riferimento all'`AppStateManager` così da disegnare finestre coerenti con lo stato corrente del software. Tramite i riferimenti alle classi `Scene` e `AppSettings`, permette all'utente di visualizzare i dati caricati nell'event display – run, evento e metadati associati – e di modificare le principali impostazioni di visualizzazione, come la modalità di colorazione delle hit, il tipo di proiezione e la trasparenza delle geometrie.

Per la selezione dei file di geometria è stata integrata la libreria `ImGui FileDialog`, che estende Dear ImGui con un file browser nativo, consentendo all'utente di navigare il filesystem e selezionare il file .gltf desiderato direttamente dall'interfaccia grafica, senza dover specificare il percorso tramite riga di comando.

5.6.2 Controllo della scena da tastiera e mouse

L'interazione diretta è gestita dalla classe `Callback`, che si occupa degli eventi generati dall'utente (catturati tramite GLFW): pressione di tasti, clic e scroll del mouse, ridimensionamento e chiusura della finestra.

La manipolazione della scena tridimensionale – rotazione, traslazione e zoom – avviene invece tramite polling: anziché accodare ogni singolo evento del mouse, lo stato dei dispositivi di input viene letto una volta per frame all'interno del metodo `update` della classe `Viewport`. Questo

approccio è particolarmente vantaggioso in caso di frame rate ridotto, ad esempio durante l'esecuzione remota con inoltro della finestra grafica, poiché garantisce che la camera risponda sempre allo stato attuale dell'input senza accumulare eventi in coda.

Nel Listato 5.14 è riportata la parte del metodo `Viewport::update` che gestisce la manipolazione della camera. Da notare la condizione `isPointerUsedByGui()`: quando l'utente sta interagendo con le finestre dell'interfaccia grafica, l'input del mouse non viene propagato alla camera, evitando interferenze tra i due sistemi. La modalità di interazione dipende inoltre dal tasto modificatore: il semplice trascinamento ruota la scena tramite trackball, mentre tenendo premuto Shift si ottiene una traslazione parallela al piano di vista.

```
// Aggiorna la proiezione se la finestra viene ridimensionata
if (this->windowManager.isFramebufferChanged())
  this->projection->setAspectRatio(
    this->windowManager.getAspectRatio()
  );

// Manipolazione della camera con il tasto sinistro del mouse
if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) == GLFW_PRESS
    && !this->guiManager.isPointerUsedByGui()) {

    { ... } // Lettura posizione cursore

    if (isShiftPressed())
      this->moveParallel(deltaX, deltaY);         // Traslazione
    else
      this->rotateTrackball(origin, destination); // Rotazione

    { ... } // Aggiornamento ultima posizione del mouse
}
```

Listato 5.14: Polling dell'input nel metodo `Viewport::update`

Conclusa l'implementazione del sistema, il passo successivo consiste nel verificare il soddisfacimento dei requisiti individuati nel Capitolo 2. A tale scopo, nel capitolo seguente verranno presentate le principali funzionalità del nuovo event display mediante esempi di utilizzo basati su dati reali dell'esperimento SND@LHC.

Capitolo 6

Validazione e risultati ottenuti

In questo capitolo il sistema viene validato su dati sperimentali reali, verificando come le funzionalità implementate rispondano ai requisiti individuati nel Capitolo 2. Dopo una descrizione dei dati utilizzati, vengono presentate le principali funzionalità dell'event display attraverso schermate commentate, per poi concludere con una verifica dei requisiti e una discussione dei risultati ottenuti.

Gli esempi mostrati nelle sezioni successive fanno riferimento alla run `5262`, evento `112104442`, acquisita il 16 novembre 2022 durante la Run 3 di LHC. I dati di questa run appartengono al dataset utilizzato dalla collaborazione per la prima osservazione di interazioni di neutrini senza muoni nello stato finale, pubblicata su *Physical Review Letters* nel giugno 2025 [9]. La scelta di questo evento consente inoltre un confronto diretto con il software preesistente, del quale era già stato mostrato l'output in Figura 2.2.

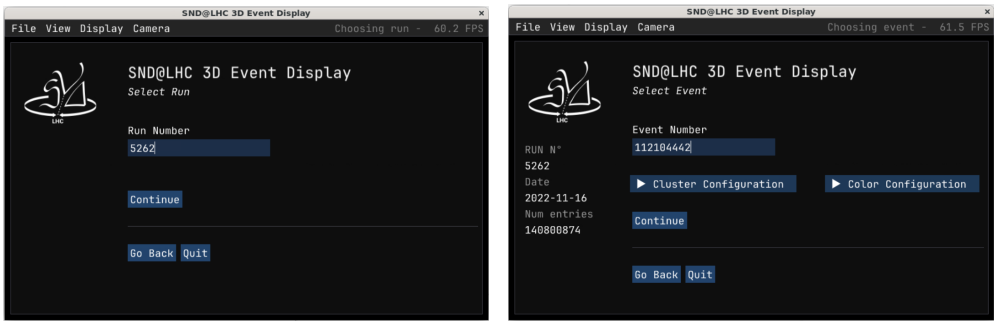
6.1 Funzionalità del sistema

Vengono ora illustrate le principali funzionalità dell'event display mediante esempi di utilizzo basati su dati reali.

6.1.1 Selezione dell'evento

La selezione dell'evento da visualizzare avviene attraverso due schermate successive. Nella prima l'utente inserisce il numero di run (Figura 6.1a); nella seconda vengono visualizzati i metadati della run caricata e viene richiesto il numero di evento, insieme alle configurazioni relative al clustering ed alla colorazione delle hit (Figura 6.1b), che verranno illustrate successivamente.

Le stesse schermate sono inoltre accessibili durante l'interazione tramite apposite voci di menu, consentendo di cambiare i dati visualizzati senza riavviare il software. Per rendere più agevole la navigazione tra eventi consecutivi, durante l'interazione l'utente può scorrere gli eventi consecutivi in avanti e indietro rispettivamente utilizzando i pulsanti `K` e `J` della tastiera.

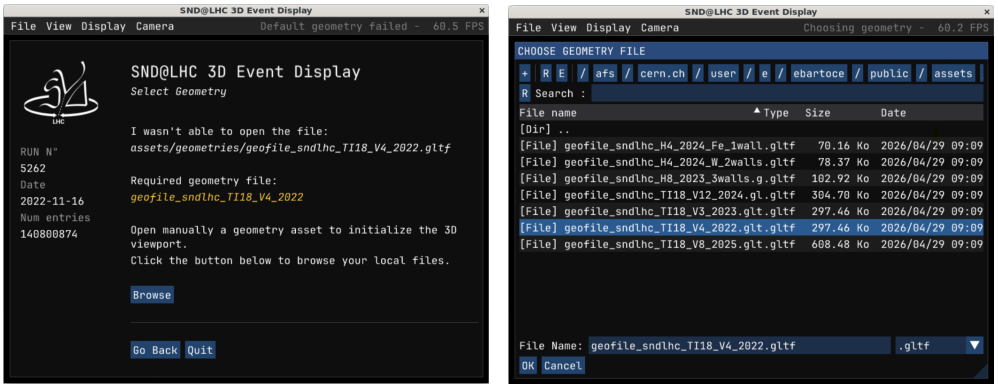


(a) Scelta run (b) Scelta evento

Figura 6.1: Selezione dei dati da visualizzare

Qualora il file di geometria predefinito non venga trovato, il sistema notifica l'errore indicando esplicitamente il file richiesto per la run selezionata (Figura 6.2a). Consente poi di selezionarne manualmente uno alternativo tramite un file browser (Figura 6.2b), riprendendo l'inizializzazione senza richiedere il riavvio del software. Il file browser rimane inoltre accessibile durante l'interazione per caricare geometrie alternative.

Questa modalità di selezione soddisfa il requisito di selezionare run ed evento tramite interfaccia, eliminando la necessità di conoscere il sistema di archiviazione EOS e i comandi da shell illustrati nel Listato 2.1.



(a) Geometria richiesta (b) Scelta manuale geometria

Figura 6.2: Gestione del file di geometria non trovato nel percorso predefinito

6.1.2 Configurazione cluster e colorazione

Le opzioni di clustering e colorazione (Figura 6.3) sono accessibili dalla schermata di selezione dell'evento e possono essere modificate in qualsiasi momento durante l'interazione, senza cambiare il numero di evento visualizzato.

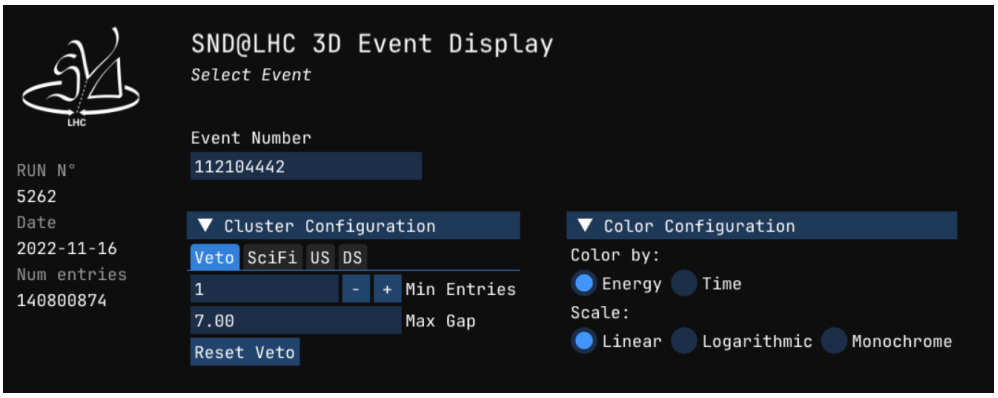
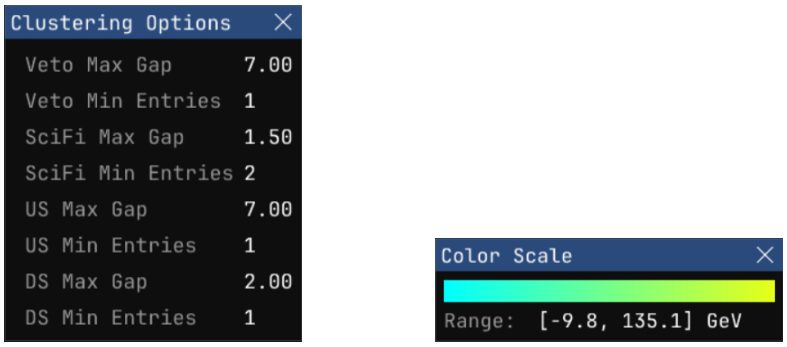


Figura 6.3: Configurazione dell'algoritmo di clustering e della scala di colore nella schermata di selezione evento

Per il clustering, i parametri configurabili sono il numero minimo di hit necessarie a formare un cluster e la distanza massima consentita tra hit adiacenti. Il raggruppamento delle hit vicine riduce l'affollamento vi-

sivo della scena, preservando le informazioni fisicamente rilevanti. Poiché i sottorivelatori hanno risoluzioni differenti, come descritto nel Capitolo 1, il sistema mette a disposizione un insieme di parametri specifico per ciascuno di essi. Durante l'interazione i parametri utilizzati sono riportati in una finestra, visualizzata in Figura 6.4a. Per evitare che l'utente debba impostarli manualmente ad ogni utilizzo del display, sono presenti dei valori predefiniti, scelti dalla collaborazione durante la fase di analisi.

Per la colorazione, è possibile scegliere la grandezza fisica di riferimento – energia o tempo – e la scala cromatica da applicare: lineare, logaritmica o monocromatica. Questa flessibilità consente di evidenziare aspetti diversi dell'evento a seconda dell'analisi in corso. È possibile abilitare la visualizzazione della scala (mostrata in Figura 6.4b), così da semplificare l'associazione di ogni colore al valore di riferimento.



(a) Configurazioni del clustering

(b) Scala colore

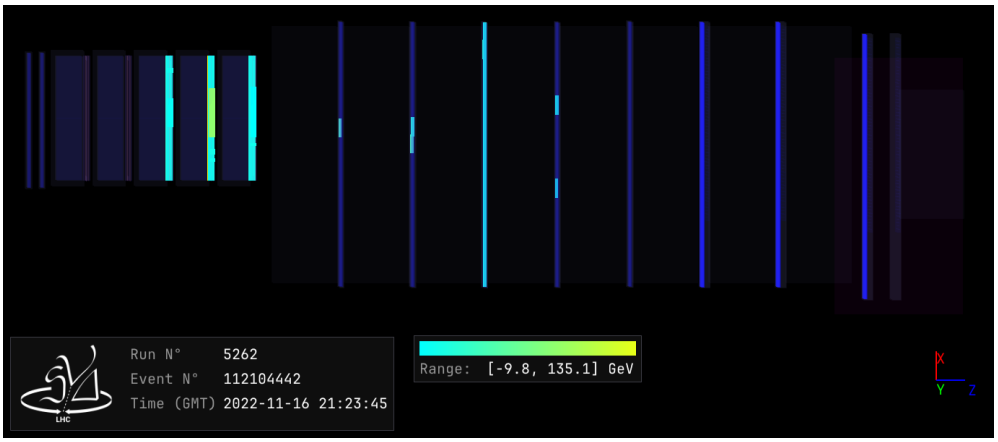
Figura 6.4: Visualizzazione configurazioni dei dati caricati

Nel complesso, le due opzioni soddisfano rispettivamente i requisiti di selezionare le possibili configurazioni dei dati da caricare e di visualizzare l'energia depositata in modo coerente con i dati letti tramite ROOT.

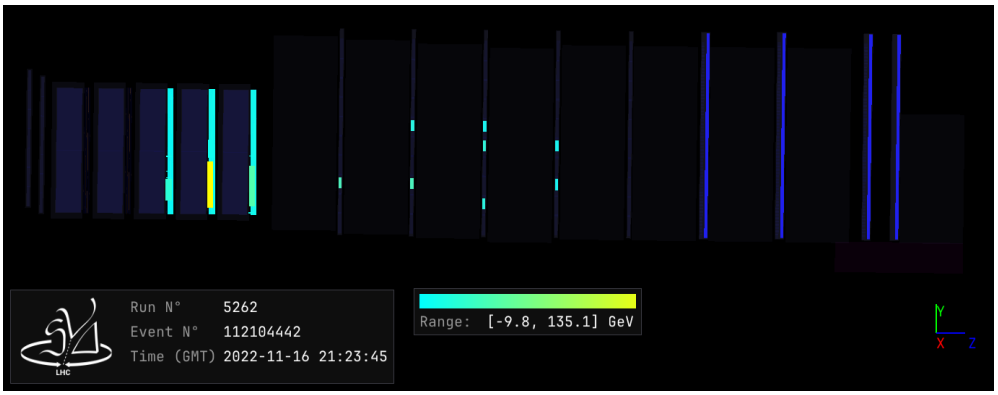
6.1.3 Interazione con la scena

Una volta caricati i dati, è possibile interagire liberamente con la scena tridimensionale mediante operazioni di rotazione, traslazione e zoom,

soddisfacendo il requisito di osservare la geometria del rivelatore da prospettive arbitrarie. I controlli sono accessibili tramite il puntatore e lo scroll del mouse, con il tasto Shift come modificatore per la traslazione. Sono inoltre disponibili quattro viste predefinite – frontale, laterale, dall'alto e isometrica – richiamabili tramite appositi tasti della tastiera, ciascuna delle quali attiva automaticamente la proiezione ortografica. In Figura 6.5 sono riportate le viste dall'alto e laterale, corrispondenti alle proiezioni già presenti nel software preesistente.



(a) Vista dall'alto

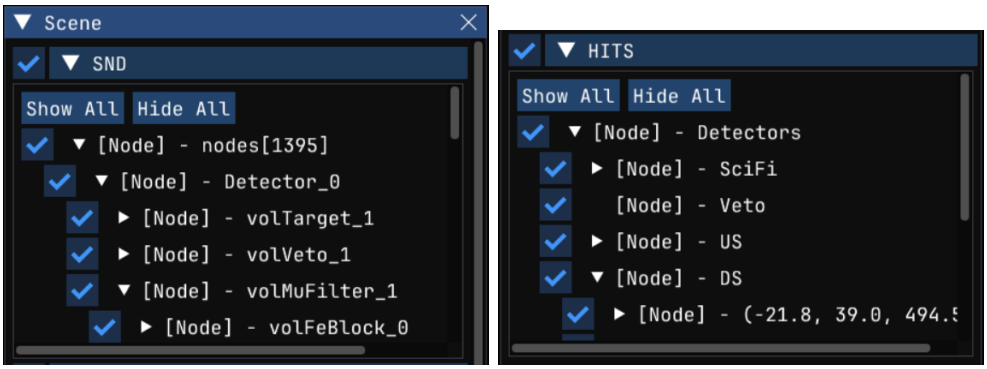


(b) Vista laterale

Figura 6.5: Attivazione delle viste predefinite

Visibilità dei nodi

Il pannello `Scene` (Figura 6.6), accessibile in qualsiasi momento durante l'interazione, espone l'albero dei nodi della scena diviso in due sezioni: la geometria del rivelatore (`SND`) e le hit dell'evento (`HITS`). Per ciascun nodo è disponibile una checkbox che ne abilita o disabilita la visualizzazione a runtime, consentendo di isolare singoli sottorivelatori senza ricaricare i dati.



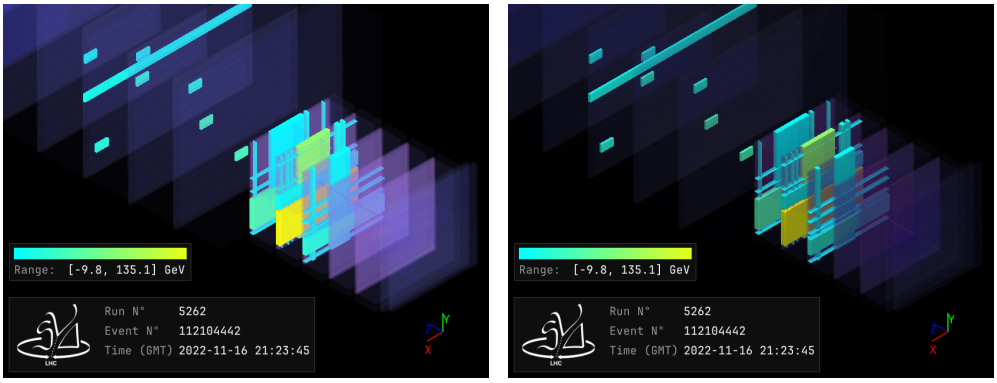
(a) Albero della geometria (b) Albero delle hit

Figura 6.6: Le due sezioni del pannello `Scene`: la geometria del rivelatore e le hit dell'evento

6.1.4 Illuminazione e trasparenza

Un menu dedicato consente all'utente di attivare o disattivare l'illuminazione e la trasparenza, oltre a configurarne i relativi parametri: per la trasparenza è possibile regolare separatamente l'opacità dei bordi e delle facce dei poligoni; per l'illuminazione è possibile modificare le proprietà delle sorgenti luminose puntiformi.

Il confronto tra la scena illuminata e non è apprezzabile nella Figura 6.7: sebbene non prevista tra i requisiti iniziali, essa contribuisce in modo significativo alla leggibilità della scena: senza illuminazione, le hit appaiono come solidi uniformemente colorati; con l'illuminazione attiva, il calcolo della luce incidente rende distinguibili le singole facce, migliorando la percezione della profondità e della forma degli oggetti nella scena.



(a) Illuminazione disattivata (b) Illuminazione attivata

Figura 6.7: Confronto di parte del rivelatore con illuminazione attivata e disattivata

6.2 Verifica dei requisiti

Sulla base delle funzionalità presentate nella sezione precedente, tutti i requisiti funzionali individuati nel Capitolo 2 risultano soddisfatti. La selezione di run ed evento avviene tramite schermate di inserimento guidato, eliminando la necessità di conoscere il sistema di archiviazione EOS e i comandi da shell del software preesistente. L'interazione con la scena è gestita tramite mouse e tastiera, con viste predefinite richiamabili istantaneamente. La visualizzazione dell'energia depositata è supportata attraverso la colorazione delle hit per energia o tempo, con scala cromatica configurabile. Infine, i parametri di clustering e la modalità di colorazione costituiscono le configurazioni dei dati selezionabili dall'utente.

Anche i requisiti non funzionali sono rispettati. La fluidità dell'interazione è garantita dalle ottimizzazioni discusse nella Sezione 5.4.2: il precalcolo delle matrici di modellazione e la riduzione delle chiamate al driver grafico consentono di raggiungere 40-60 FPS con illuminazione e trasparenza attive, un valore in linea con lo standard di fluidità delle applicazioni interattive moderne. Il ridimensionamento della finestra è gestito tramite le callback di GLFW, che aggiornano automaticamente l'area di disegno e il rapporto d'aspetto della proiezione, rendendo l'interfaccia adattabile a qualsiasi configurazione dello schermo. La navigazione tra eventi consecutivi è resa agevole dai tasti `J` e `K`, che consentono di

scorrere gli eventi in avanti e indietro senza dover reinserire i dati ogni volta. L'interazione intuitiva è infine garantita dalla barra menu, che espone tutte le funzionalità disponibili riportando accanto a ciascuna la scorciatoia da tastiera corrispondente, così da non richiedere all'utente la memorizzazione di comandi.

Nella Figura 6.8 è riportata l'intera schermata del software mostrata all'utente durante l'interazione con la scena, in cui sono visibili la barra menu con un sottomenu aperto e le indicazioni dello stato corrente e il numero di FPS, i pannelli informativi con i metadati dell'evento e la palette utilizzata, e la finestra contenente le impostazioni grafiche del software.

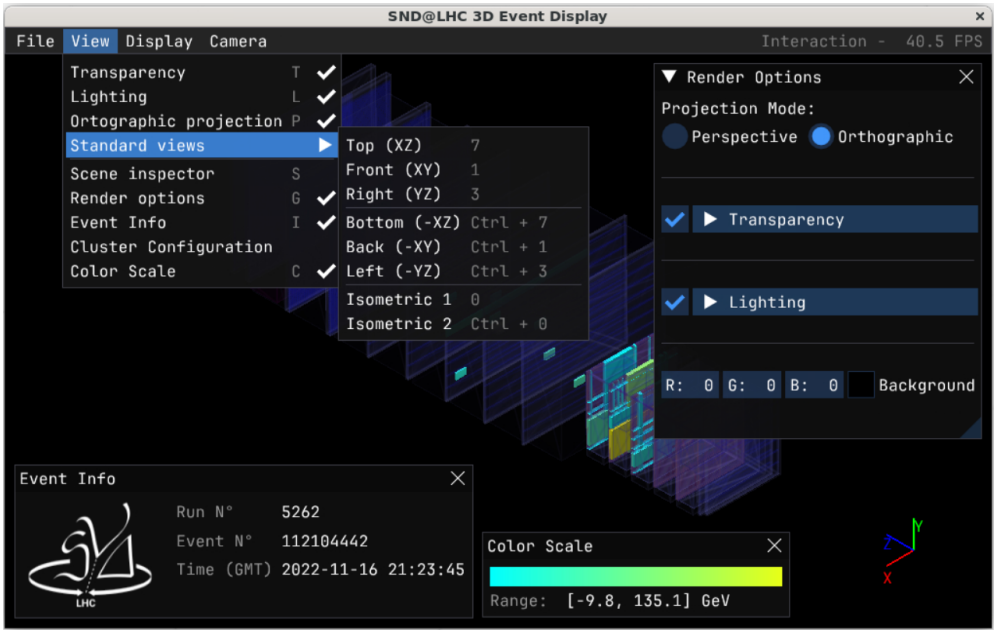
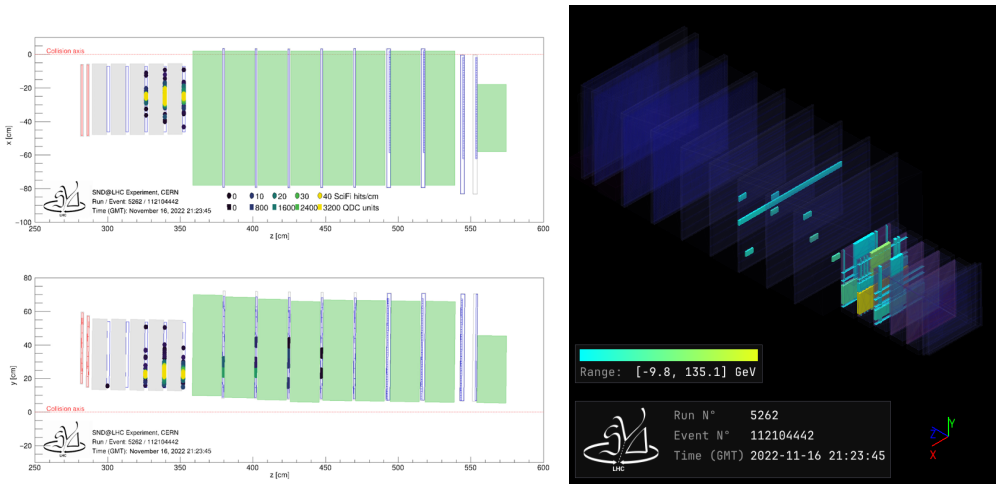


Figura 6.8: Interfaccia completa dell'event display

6.3 Confronto con il software preesistente

Viene ora mostrato un confronto diretto tra l'output ottenuto dall'event display attualmente in uso dalla collaborazione (in Figura 6.9a) e quello ottenuto tramite la funzione `Export Image` del nuovo software (in Figura 6.9b).

È evidente come la rappresentazione tridimensionale renda molto più immediata la comprensione dell'evento rilevato dal rivelatore, evitando all'utente di dover ricostruire mentalmente la posizione dei segnali a partire dalle due proiezioni bidimensionali separate. A differenza del software preesistente, vincolato alle sole viste frontale e laterale, la possibilità di scegliere liberamente l'angolazione consente di osservare la distribuzione delle hit dal punto di vista più informativo, adattandosi di volta in volta al singolo caso di studio.



(a) Event display 2D (b) Event display 3D

Figura 6.9: Confronto tra l'event display 2D del software preesistente e l'event display 3D del software sviluppato, relativi al medesimo evento (run `5262`, evento `112104442`)

I risultati ottenuti verranno ripresi nelle conclusioni, dove saranno discussi i limiti ancora presenti e i possibili sviluppi futuri del sistema.

Conclusioni

Il lavoro svolto ha riguardato l'intero processo di realizzazione del nuovo event display, a partire dall'analisi dei requisiti individuati insieme alla collaborazione fino allo sviluppo di una versione funzionante del software. Nel corso della tesi sono state descritte le principali scelte progettuali, i fondamenti matematici e grafici su cui esse si basano e, infine, le funzionalità effettivamente implementate. Come visto nel Capitolo 6, tutti i requisiti stabiliti in fase di analisi sono stati pienamente soddisfatti, creando un event display dall'interfaccia user friendly e interamente funzionante.

Nonostante i risultati raggiunti, il sistema presenta alcuni limiti, in parte già introdotti nei capitoli precedenti.

Uno dei più impattanti è la dipendenza da uno strumento esterno per la conversione dei file di geometria dal formato proprietario di ROOT allo standard glTF, come discusso nella Sezione 4.4.1.

Inoltre, data la dipendenza diretta dal framework sndsw, è possibile eseguire l'event display solo in ambienti in cui sndsw può essere compilato. Dal momento che la collaborazione ha perfezionato la compilazione solo sui server remoti del CERN, l'utilizzo del software avviene esclusivamente con forwarding del display X11 tramite ssh o tramite connessione al display remoto VNC.

L'architettura modulare adottata pone tuttavia le basi per diverse estensioni future.

Una prima possibilità riguarda l'aggiunta della selezione dei cluster con il click del puntatore. Questa funzionalità risulterebbe comoda per visualizzare nel dettaglio le informazioni dei cluster, rendendo sempre più agevole lo studio degli eventi tramite l'interazione col software.

Potrebbe anche essere utile introdurre uno slider temporale che agisce da filtro, per visualizzare l'evoluzione delle hit nel tempo, agevolando lo studio della dinamica degli eventi registrati.

Come anticipato nella Sezione 1.5.4, in sndsw non è ancora ultimata la parte di elaborazione dei dati prodotti dalle MiniDT, nonostante siano già presenti nella geometria. Non appena il framework ne includerà il supporto ufficiale, l'event display dovrà essere aggiornato per caricarli e mostrarli nella scena tridimensionale.

Il software potrebbe inoltre essere esteso per eseguire il rendering in locale anche quando i dati sono ospitati su un server remoto, tramite un'architettura client-server che disaccoppi il visualizzatore dal caricamento dei dati. La lettura dei dati sperimentali continuerebbe ad avvenire sul server remoto, dove sndsw è eseguibile, ma verrebbe esposta tramite un processo dedicato; il visualizzatore diventerebbe invece un applicativo client eseguito in locale, che riceve i dati già letti tramite rete e si occupa interamente del rendering. In questo modo, l'interazione con la scena avverrebbe interamente in locale, eliminando il lag introdotto dal forwarding del display e limitando la comunicazione di rete al solo trasferimento dei dati.

Infine, durante il *Long Shutdown 3* (2026-2030), oltre allo studio dei dati raccolti durante l'operatività del rivelatore, è prevista l'evoluzione di SND@LHC in SND@HL-LHC – ovvero High Luminosity LHC. Il sistema dovrà quindi essere mantenuto e adattato alla nuova configurazione, sfruttando la modularità dell'architettura progettata: per la geometria sarà sufficiente convertire i nuovi file ROOT, mentre andrà riadattato il modulo `io` per leggere i dati secondo le nuove strutture previste da SND@HL-LHC.

L'applicativo sviluppato è stato presentato alla collaborazione durante la *25th SND@LHC Collaboration Meeting*, tenutasi al CERN nel periodo 23-25 giugno 2026, riscuotendo un riscontro molto positivo da parte dei partecipanti. In particolare, è stato espresso un forte interesse nell'utilizzare da subito il software per scopi di divulgazione, nell'ambito del progetto *Open Science* del CERN, e nell'avere in futuro una nuova versione del sistema adattata al rivelatore SND@HL-LHC.

Questo riscontro conferma come il lavoro svolto non si limiti a un progetto fine a se stesso, ma costituisca un contributo concreto e duraturo per la collaborazione SND@LHC.

Bibliografia

[1] Conseil Européen pour la Recherche Nucléaire (CERN). «Accelerators,» visitato il 29 apr. 2026. indirizzo: https://home.cern/science/accelerators

[2] E. Lopienska, «The CERN accelerator complex, layout in 2022,» 2022, Immagine modificata. indirizzo: https://cds.cern.ch/record/2800984

[3] Conseil Européen pour la Recherche Nucléaire (CERN). «The Large Hadron Collider,» visitato il 29 apr. 2026. indirizzo: https://home.cern/science/accelerators/large-hadron-collider

[4] M. Burgess e M. R. Villarreal. «Modello Standard delle Particelle Elementari.» Immagine modificata, licenza: CC BY-SA 4.0, visitato il 28 mag. 2026. indirizzo: https://commons.wikimedia.org/wiki/File:Modello_Standard_delle_Particelle_Elementari.svg

[5] Istituto Nazionale di Fisica Nucleare (INFN) - Laboratori Nazionali del Gran Sasso (LNGS). «Neutrini,» visitato il 27 apr. 2026. indirizzo: https://www.lngs.infn.it/it/neutrini

[6] D. Marfatia, D. W. McKay e T. J. Weiler, «New physics with ultra-high-energy neutrinos,» *Physics Letters B*, vol. 748, pp. 113–116, 2015. DOI: 10.1016/j.physletb.2015.07.002

[7] M. Ackermann et al., «Astrophysics Uniquely Enabled by Observations of High-Energy Cosmic Neutrinos,» *Bulletin of the American Astronomical Society*, vol. 51, n. 3, p. 185, 2019. DOI: 10.48550/arXiv.1903.04334

[8] M. Bustamante e A. Connolly, «Extracting the Energy-Dependent Neutrino-Nucleon Cross Section Above 10 TeV Using IceCube Showers,» *Physical Review Letters*, vol. 122, n. 4, p. 041 101, gen. 2019. DOI: 10.1103/PhysRevLett.122.041101

[9] R. Albanese et al., «Observation of Collider Muon Neutrinos with the SND@LHC Experiment,» *Physical Review Letters*, vol. 131, n. 3, p. 031 802, lug. 2023. DOI: 10.1103/PhysRevLett.131.031802

[10] R. Aaij et al., «Measurements of prompt charm production cross-sections in pp collisions at $\sqrt{s} = 13$ TeV,» *Journal of High Energy Physics*, vol. 2016, n. 3, p. 159, 2016, Erratum: JHEP 09 (2016) 013, JHEP 05 (2017) 074. DOI: 10.1007/JHEP03(2016)159

[11] G. M. Dallavalle, «Recent results from SND@LHC,» *Il Nuovo Cimento C*, vol. 49, n. 1-2, p. 24, 2026, Immagine modificata. DOI: 10.1393/ncc/i2026-26024-5

[12] G. Acampora et al., «SND@LHC: the scattering and neutrino detector at the LHC,» *Journal of Instrumentation*, vol. 19, n. 05, P05067, mag. 2024. DOI: 10.1088/1748-0221/19/05/P05067

[13] R. Acquafredda et al., «The OPERA experiment in the CERN to Gran Sasso neutrino beam,» *Journal of Instrumentation*, vol. 4, n. 04, P04018, apr. 2009. DOI: 10.1088/1748-0221/4/04/P04018

[14] G. Paggi, «Muon neutrino interaction studies with SND@LHC detector,» *Proceedings of Science*, vol. EPS-HEP2025, p. 144, 2025. DOI: 10.22323/1.485.0144

[15] C. Ahdida et al., «SND@LHC - Scattering and Neutrino Detector at the LHC,» CERN, Geneva, rapp. tecn., 2021, Immagine modificata. indirizzo: https://cds.cern.ch/record/2750060

[16] S. Chatrchyan et al., «The CMS Experiment at the CERN LHC,» *Journal of Instrumentation*, vol. 3, n. 08, S08004, ago. 2008. DOI: 10.1088/1748-0221/3/08/S08004

[17] G. Paggi. «Upgrades of SND@LHC (and CMS!) and search for muon neutrinos from proton-proton collisions.» Immagine modificata, INFN Bologna, visitato il 27 mag. 2026. indirizzo: https://agenda.infn.it/event/50055/contributions/284437/attachments/144802/220264/final_paggi_16_gen.pdf

[18] R. Brun. «What does ROOT stand for?» Visitato il 3 giu. 2026. indirizzo: https://root.cern/about/what_does_root_stand_for/

[19] T. Mc Cauley e CMS Collaboration, «Event displays of PbPb collision events in the CMS detector at the end of 2018,» CMS Collection., 2018. visitato il 4 giu. 2026. indirizzo: https://cds.cern.ch/record/2648517

[20] D. Abbaneo et al., «Observation of Collider Neutrinos without Final State Muons with the SND@LHC Experiment,» *Physical Review Letters*, vol. 134, n. 23, p. 231 802, giu. 2025. DOI: 10.1103/r2qy-9hft

[21] C. Bohak et al., «RenderCore - a new WebGPU-based rendering engine for ROOT-EVE,» *EPJ Web of Conferences*, vol. 295, p. 03 035, 2024. DOI: 10.1051/epjconf/202429503035

[22] M. Kraus. «Orthographic view frustum.» Licenza: CC BY-SA 3.0, Wikimedia Commons, visitato il 10 giu. 2026. indirizzo: https://commons.wikimedia.org/wiki/File:Orthographic_view_frustum.png

[23] M. Kraus. «Oblique perspective view frustum.» Licenza: CC BY-SA 3.0, Wikimedia Commons, visitato il 10 giu. 2026. indirizzo: https://commons.wikimedia.org/wiki/File:Oblique_perspective_view_frustum.png

[24] Vierge Marie. «Cube clipping.» Licenza: Public domain, Wikimedia Commons, visitato il 11 giu. 2026. indirizzo: https://commons.wikimedia.org/wiki/Image:Cube_clipping.svg

[25] W. Muła. «Rasterization bw.» Licenza: CC BY-SA 3.0, Wikimedia Commons, visitato il 10 giu. 2026. indirizzo: https://commons.wikimedia.org/wiki/Image:Rasterization_bw.svg

[26] T. Babb. «Recursive raytrace of a sphere.» Licenza: CC BY-SA 4.0, Wikimedia Commons, visitato il 10 giu. 2026. indirizzo: https://commons.wikimedia.org/wiki/File:Recursive_raytrace_of_a_sphere.png

[27] B. T. Phong, «Illumination for computer generated pictures,» *Commun. ACM*, vol. 18, n. 6, pp. 311–317, giu. 1975, ISSN: 0001-0782. DOI: 10.1145/360825.360839

[28] B. Smith. «Phong components version 4.» Licenza: CC BY-SA 3.0, Wikimedia Commons, visitato il 10 giu. 2026. indirizzo: https://commons.wikimedia.org/wiki/File:Phong_components_version_4.png

[29] J. F. Blinn, «Models of light reflection for computer synthesized pictures,» *SIGGRAPH Comput. Graph.*, vol. 11, n. 2, pp. 192–198, lug. 1977, ISSN: 0097-8930. DOI: 10.1145/965141.563893

[30] Davi.trip. «Shading models.» Licenza: CC BY-SA 4.0, Wikimedia Commons, visitato il 10 giu. 2026. indirizzo: https://commons.wikimedia.org/wiki/File:Shading_models.png

[31] H. Gouraud, «Continuous Shading of Curved Surfaces,» *IEEE Transactions on Computers*, vol. C-20, n. 6, pp. 623–629, giu. 1971. DOI: 10.1109/T-C.1971.223313

[32] E. Gamma, R. Helm, R. Johnson e J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994, ISBN: 0201633612.

[33] The Khronos 3D Formats Working Group. «glTF™ 2.0 Specification.» ver. 2.0.1, visitato il 13 giu. 2025. indirizzo: https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html