Alma Mater Studiorum · Università di Bologna

Dipartimento di Informatica — Scienza e Ingegneria

Corso di Laurea Triennale in Informatica

ANALISI COMPARATIVA DELLE TECNICHE DI OTTIMIZZAZIONE DELLE PIPELINE DI RENDERING RASTERIZZATO NEI PRINCIPALI GAME ENGINE MODERNI

Relatore:
Prof. Damiana Lazzaro

Presentata da:
Michele Cesari

Sessione Unica

Anno Accademico 2024/2025

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Negli ultimi decenni, l'industria dell'intrattenimento digitale e della simulazione interattiva ha assistito ad un'evoluzione tecnologica senza precedenti. La computer graphics, nata inizialmente come strumento per semplici rappresentazioni bidimensionali, ha raggiunto livelli di fotorealismo tali da rendere spesso indistinguibile un'immagine generata sinteticamente da una reale. Questa corsa verso il realismo è stata fino ad ora sostenuta dalla legge di Moore e dall'evoluzione parallela delle architetture hardware, in particolare delle Graphics Processing Units o GPU, che hanno visto un incremento esponenziale della capacità di calcolo e della memoria dedicata. Tuttavia, insieme alla potenza di calcolo, sono cresciute anche le aspettative degli utenti e la complessità degli asset grafici: modelli 3D composti da milioni di poligoni, texture in altissima risoluzione (4K/8K) e sistemi di illuminazione dinamica avanzata sono ormai lo standard nelle produzioni moderne. In questo scenario, i Game Engine come Unity, Unreal Engine e Godot hanno assunto un ruolo centrale, democratizzando l'accesso a tecnologie di rendering avanzate e fornendo agli sviluppatori strumenti potenti per la gestione di mondi virtuali complessi. Nonostante l'avanzamento hardware, il real-time rendering rimane vincolato da un limite insuperabile: il tempo. A differenza del rendering offline utilizzato nell'industria cinematografica, dove un singolo fotogramma può richiedere ore per essere calcolato, un'applicazione interattiva deve generare immagini a una frequenza minima di 30 o 60 fotogrammi al secondo (FPS). Ciò impone un budget estremamente ristretto: per mantenere i 60 FPS, l'intero ciclo di elaborazione di un frame, che va dalla logica di gioco alla rasterizzazione finale, deve concludersi in circa 16,6 millisecondi. Se la complessità della scena supera la capacità di elaborazione della CPU o della GPU entro questo intervallo, si verifica un calo del frame rate, compromettendo la fluidità e l'usabilità dell'applicazione. Il problema centrale che questa tesi affronta è quindi la gestione del trade-off tra qualità visiva e prestazioni. La soluzione risiede nell'applicazione di rigorose tecniche di ottimizzazione, volte a ridurre il carico computazionale eliminando ciò che non

è visibile o riducendo la qualità di ciò che è distante, cercando di rendere questi "tagli" impercettibili all'occhio umano. Il presente elaborato si pone l'obiettivo di analizzare, classificare e confrontare le principali tecniche di ottimizzazione grafica adottate nello stato dell'arte dello sviluppo videoludico. In particolare, il lavoro si focalizza su tre macro-aree di intervento:

- Gestione della geometria: tramite tecniche di Level of Detail (LOD) e Culling (selezione degli oggetti visibili).

- Gestione delle chiamate di disegno (Draw Calls): tramite tecniche di Batching e Instancing per ridurre il collo di bottiglia sulla CPU.

- Gestione della pipeline e della memoria: analizzando differenze tra Forward e Deferred Shading e l'uso del Virtual Texturing.

Al fine di garantire la coerenza e la riproducibilità dei risultati, questa ricerca si concentra esclusivamente sulle pipeline di rendering rasterizzato. L'esclusione di tecniche basate su Hardware Ray Tracing e di algoritmi di ricostruzione dell'immagine basati su Intelligenza Artificiale (come DLSS, FSR o XeSS) risponde alla necessità di isolare le prestazioni e l'efficienza nativa del motore grafico, evitando interferenze dovute a processi ibridi o di post-processing. Questa scelta permette di condurre un confronto quantitativo omogeneo sulle performance di base, isolando i colli di bottiglia del paradigma raster.

Lo scopo non è solo teorico, ma anche pratico e comparativo: la tesi mira a valutare come queste tecniche siano implementate nei motori Unity, Unreal Engine e Godot, misurandone l'efficacia attraverso una serie di stress test controllati. Si intende fornire una guida quantitativa che evidenzi quale approccio risulti più efficiente in determinati scenari di utilizzo. Il lavoro è articolato in cinque capitoli, strutturati secondo un percorso logico che va dai fondamenti teorici all'applicazione sperimentale:

- Il Capitolo 1 fornisce i fondamenti teorici necessari alla comprensione della pipeline grafica, descrivendo matematicamente e algoritmicamente le tecniche di ottimizzazione indipendentemente dal software utilizzato.

- Il Capitolo 2 analizza nel dettaglio le architetture dei tre motori grafici presi in esame, evidenziando le specificità implementative come la Scriptable Render Pipeline di Unity o il sistema di Virtual Texturing di Unreal.

- Il Capitolo 3 descrive la metodologia di ricerca adottata, definendo gli strumenti di profiling, le metriche di riferimento e la progettazione della "Scena Stress Test" utilizzata per gli esperimenti.
- Il Capitolo 4 presenta l'analisi dei risultati sperimentali, confrontando i dati raccolti per valutare l'impatto reale delle ottimizzazioni nei diversi engine.
- Infine, il Capitolo 5 trae le conclusioni del lavoro svolto, discutendo i limiti della sperimentazione e delineando possibili sviluppi futuri legati a tecnologie emergenti.

Capitolo 1

Architettura della Pipeline Grafica e Metodologie di Ottimizzazione Real-Time

Per poter comprendere appieno le tecniche di ottimizzazione analizzate in questo elaborato, è necessario analizzare preliminarmente i principi fondamentali su cui esse si basano. In questo capitolo verranno illustrati i meccanismi della pipeline grafica moderna e le differenze tra i principali paradigmi di rendering.

1.1 La pipeline grafica

La pipeline grafica di rendering rappresenta l'insieme delle fasi necessarie per trasformare un insieme di dati geometrici rappresentanti vertici e primitive in un'immagine composta da pixel.

Originariamente concepita come una sequenza lineare di operazioni a funzione fissa, la pipeline grafica si è evoluta per includere stadi programmabili che consentono agli sviluppatori di personalizzare il suo comportamento, attraverso l'uso di shader.

Il cuore della pipeline grafica si articola in tre fasi cruciali: il **vertex processing**, la **rasterizzazione** e il **fragment processing**.

1.1.1 Vertex processing

Fase attuata dal *vertex processor* attraverso il **vertex shader**. Ogni insieme di vertici dato in input alla GPU è inizialmente espresso nel suo sistema di coordinate locali.

Il primo compito del vertex processing consiste nel collocare correttamente gli oggetti all'interno della scena, applicando trasformazioni di traslazione, rotazione e scala nello spazio di mondo tramite la **matrice di modellazione** M, che racchiude tutte le trasformazioni geometriche associate all'oggetto.

Una volta posizionati gli oggetti all'interno del mondo, è necessario esprimere la posizione di ogni vertice in relazione alla telecamera, passando dalle coordinate di mondo alle coordinate di vista. Questo avviene attraverso la **Matrice di Vista V**. Tramite questa matrice, viene aggiunta una quarta componente $w = 1$ alle coordinate tridimensionali di ogni vertice.

La fase finale della parte di vertex processing consiste nel passare dallo spazio di coordinate di vista allo spazio di coordinate di clip. Questo avviene attraverso la **matrice di proiezione P**. La moltiplicazione, inoltre, assegna alla componente omogenea w il valore della componente z.

Queste tre operazioni possono essere unificate in un unica moltiplicazione. Definiamo la matrice

$$MVP = P \cdot V \cdot M \tag{1.1}$$

Una volta fatto ciò, per applicare tutte le trasformazioni ai nostri vertici in spazio locale ed ottenere le coordinate in spazio di clip, procediamo come segue:

$$v_{\text{clip}} = MVP \cdot v_{\text{local}} \tag{1.2}$$

1.1.2 Rasterizzazione

Una volta ottenute le coordinate dei vertici in spazio di clip, i dati vengono passati a una fase della pipeline a funzione fissa, ovvero non programmabile. Questa fase prende il nome di **rasterizzazione**. Il suo compito è quello di analizzare le coordinate dei vertici, elaborarle e decidere quali pixel saranno coinvolti nella rappresentazione della nostra immagine.

La fase di rasterizzazione si articola in vari passaggi:

- **Primitive Assembly:** i vertici trasformati vengono aggregati per costituire primitive geometriche, prevalentemente triangoli. L'ordine di attraversamento dei vertici chiamato *winding order* determina l'orientamento della superficie: per convenzione standard, se i vertici appaiono ordinati in senso antiorario rispetto al punto di vista, si assume che la faccia visibile sia quella anteriore (o esterna). Solitamente questa informazione è codificata in un

vettore, chiamato *normal*, che rappresenta la direzione perpendicolare alla superficie esterna del triangolo.

- **Clipping:** prima della rasterizzazione vera e propria, viene verificato quali primitive saranno effettivamente visibili nell'immagine finale. Questo avviene confrontando la posizione di ogni vertice con un *view frustum*, ovvero un volume di vista definito da un tronco di piramide. Se tutti i vertici di un triangolo si trovano all'interno del volume di vista, allora tutto il triangolo sarà visibile. Se invece alcuni vertici del triangolo sono esterni, entrano in azione algoritmi complessi atti a "tagliare" il triangolo, rimuovendo la parte non visibile.

- **Perspective Division:** successivamente avviene la cosiddetta *divisione prospettica*: le coordinate dei vertici (x, y, z, w) nello spazio di clip vengono divise per la loro stessa componente omogenea w. Poiché $w = z$, avremo che la dimensione z sarà schiacciata a un valore di 1. A livello visivo, gli oggetti piú lontani dalla telecamera risulteranno piú piccoli, mentre quelli piú vicini saranno piú grandi. Il risultato di questa operazione sono le *Normalized Device Coordinates* (NDC), coordinate normalizzate all'interno di un cubo unitario $[-1, 1]$, pronte per essere mappate sullo schermo.

- **Viewport Transform:** le coordinate NDC vengono mappate sulle coordinate effettive della finestra a schermo.

- **Scan Conversion:** Verificando quali pixel della finestra cadono all'interno delle primitive, la GPU genera i *frammenti*, che rappresentano i potenziali pixel da colorare. Il numero totale di frammenti generati può essere significativamente superiore al numero di pixel effettivi, a causa della sovrapposizione delle primitive.

- **Interpolazione degli attributi:** attraverso l'uso delle *coordinate baricentriche*, la GPU interpola linearmente i dati associati ai vertici (come normali, coordinate UV e colori) su tutta la superficie della primitiva, fornendo a ogni frammento i propri dati specifici.

1.1.3 Fragment processing

Successivamente interviene il fragment processor attraverso il **fragment shader**. Questo stadio elabora i frammenti generati dalla rasterizzazione per determinare il colore finale di ogni pixel. In questa fase vengono calcolati i modelli

di illuminazione, campionate le texture e valutate le interazioni tra le proprietà fisiche dell'oggetto e le sorgenti di luce. Poiché il numero di frammenti può superare di ordini di grandezza il numero di vertici, è in questa fase che si concentra tipicamente il maggior carico computazionale della pipeline.

1.1.4 Colli di bottiglia nella pipeline

La pipeline grafica è quindi configurata come un sistema a dipendenze sequenziali, in cui il throughput globale è dettato dalla capacità di elaborazione della fase più restrittiva. Nel contesto del real-time rendering, il rispetto del frame budget non dipende solo dalla potenza bruta dei singoli componenti, ma anche dall'efficienza con cui esse vengono sfruttate e sul corretto bilanciamento dei carichi di lavoro

Come illustrato in Figura 1.1, il throughput effettivo di un frame può essere limitato in stadi differenti della pipeline, portando a tre configurazioni critiche:

- **Scenario CPU-Limited:** La CPU non riesce a preparare i comandi e a gestire la logica dell'applicazione entro l'intervallo temporale prefissato. In questo stato, la GPU rimane in attesa, sprecando cicli di calcolo potenzialmente disponibili.

- **Scenario Bus-Limited:** Il bottleneck si sposta sull'interfaccia di comunicazione tra CPU e GPU. La latenza dovuta al driver overhead o alla necessità di sincronizzazione impedisce un flusso costante di dati.

- **Scenario GPU-Limited:** Il bottleneck risiede nella pipeline di esecuzione della GPU. Anche se la CPU prepara i comandi istantaneamente, la densità geometrica, la complessità degli shader o la saturazione della banda VRAM impediscono alla GPU di completare la rasterizzazione e lo shading entro il frame budget.

1.2 Rendering Paths

L'applicazione dei modelli di illuminazione durante la fase di fragment processing può essere gestita attraverso architetture di rendering differenti, note come **rendering paths**. La scelta del rendering path è determinante per le prestazioni dell'applicazione, specialmente in relazione alla complessità geometrica della scena

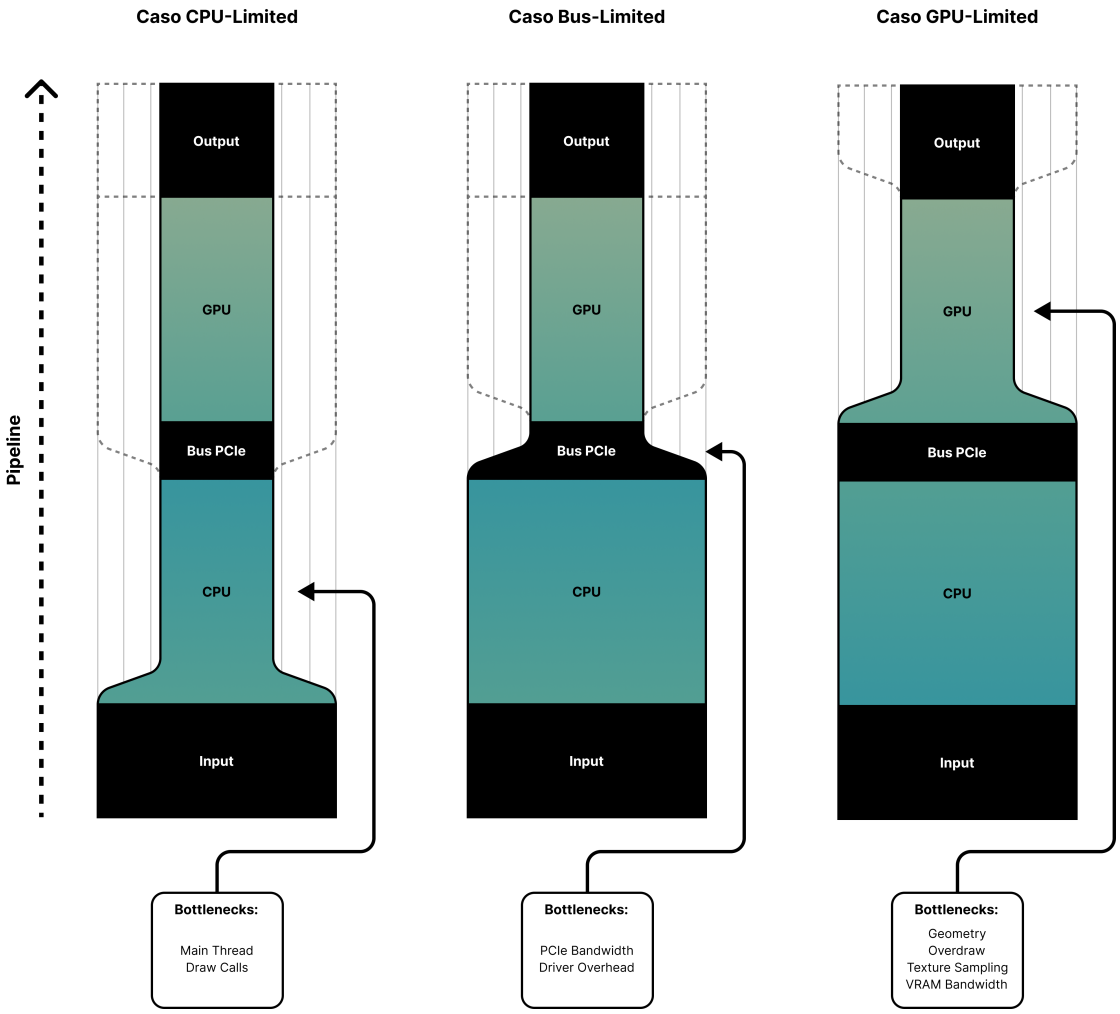

Figura 1.1: Rappresentazione schematica dei tre scenari di bottleneck nella pipeline di rendering.

e al numero di sorgenti luminose attive. Analizzeremo di seguito i due paradigmi fondamentali, ponendo le basi per la successiva discussione sulle loro varianti ottimizzate.

1.2.1 Forward Shading

Il *Forward Shading* rappresenta l'approccio classico e più immediato al rendering. In questa architettura, durante la rasterizzazione della geometria, l'illuminazione viene calcolata immediatamente per ogni frammento generato. La GPU elabora le primitive geometriche e, per ciascun frammento risultante, valuta il contributo di tutte le sorgenti luminose che influenzano l'oggetto. Matematicamente, la complessità computazionale C_F di questo approccio scala linearmente

col prodotto tra il numero di oggetti e il numero di luci considerate:

$$C_F \approx O(N_{\text{frammenti}} \cdot N_{\text{luci}}) \tag{1.3}$$

Tale dipendenza rende il Forward Shading computazionalmente oneroso in scene caratterizzate da un elevato numero di luci dinamiche, poiché ogni luce aggiuntiva richiede una nuova valutazione del modello di illuminazione per tutti i frammenti interessati.

1.2.2 Deferred Shading

Il *Deferred Shading* introduce un cambiamento di paradigma significativo rispetto al Forward Shading. Esso nasce dall'esigenza di gestire scene con un gran numero di luci dinamiche senza incorrere in un aumento lineare della complessità computazionale rispetto alla geometria. In questo approccio, la fase di rasterizzazione viene suddivisa in due passaggi distinti:

- **Geometry Pass:** Durante questo primo passaggio, la scena viene rasterizzata e i dati geometrici essenziali (come posizioni, normali, colori diffusi e speculari) vengono memorizzati in una serie di texture chiamate *G-buffer* (Geometry Buffer) la cui risoluzione coincide con quella dell'immagine finale. In questa fase non viene calcolata alcuna illuminazione.

- **Lighting Pass:** Nel secondo passaggio, l'illuminazione viene calcolata usando esclusivamente i dati precedentemente salvati nel G-Buffer. Poiché tutte le informazioni necessarie sono già disponibili, la GPU può valutare il contributo di ogni sorgente luminosa in modo indipendente, senza dover rieseguire la rasterizzazione della geometria.

La complessità computazionale C_D del Deferred Shading dipende principalmente dal numero di luci e dal numero di pixel da illuminare, piuttosto che dal numero di oggetti:

$$C_D \approx O(N_{\text{pixel}} \cdot N_{\text{luci}}) \tag{1.4}$$

Questa separazione tra geometria e illuminazione consente al Deferred Shading di gestire efficacemente un gran numero di luci dinamiche, rendendolo particolarmente adatto per scene complesse.

Tuttavia, il Deferred Shading presenta alcune limitazioni significative:

- Impossibilità di gestire correttamente le trasparenze.

- Difficoltà nell'implementazione di tecniche di anti-aliasing come MultiSample Anti-Aliasing (MSAA), a causa della natura differita del calcolo dell'illuminazione.

- Necessità di un elevato consumo di memoria per il G-buffer, che può diventare proibitivo in scenari con risoluzioni molto alte.

La scelta tra Forward e Deferred Shading modifica radicalmente la natura del bottleneck: mentre il Forward è tipicamente limitato dal numero di vertici e dalla complessità degli shader (GPU-limited), il Deferred sposta il carico sulla banda di memoria video (Bus-limited) necessaria per gestire il G-Buffer.

1.2.3 L'evoluzione dello Shading: Tiled e Clustered

Per superare i limiti di computazione del Forward Shading e gli alti consumi di memoria del Deferred Shading, la ricerca accademica e l'industria si sono mosse verso tecniche di Light Culling.

Il Light Culling è un processo di filtraggio che mira a ridurre il numero di luci da considerare per ogni frammento, eliminando quelle che non contribuiscono all'illuminazione del pixel.

Tiled Shading

Introdotto da Olsson e Assarsson nel 2011, il Tiled Shading propone di suddividere lo spazio schermo in una griglia bidimensionale di celle rettangolari, detti tile.

Prima della fase di shading, viene eseguito un passaggio di Light Culling:

1. Ogni tile viene utilizzato come una "finestra" sul mondo 3D, generando un volume di vista che si estende dal near plane al far plane[1]del volume di vista della telecamera. Questo volume di vista viene poi ristretto, portando il near plane alla distanza dell'oggetto con la profondità minima e il far plane alla distanza dell'oggetto con la profondità massima all'interno del tile.

2. Tutte le luci della scena vengono testate contro questi volumi di vista.

3. Viene generata una lista di indici di luci attive specifica per ogni tile.

[1]Il **near plane** e il **far plane** sono i piani che delimitano il volume di vista della telecamera. Il near plane è il piano più vicino alla telecamera, mentre il far plane è quello più lontano. Tutti gli oggetti al di fuori di questo intervallo non vengono renderizzati.

Durante il rendering, ogni pixel accede solamente alla lista di luci ridotta del tile a cui appartiene. Questo approccio è applicabile sia al Deferred che al Forward Shading, riducendo drasticamente il numero di calcoli inutili.

Limitazioni del Tiled Shading: La discontinuità di profondità Nonostante il Tiled Shading migliori le prestazioni, soffre di un problema critico noto come *Depth Discontinuity*. Il caso estremo di questo problema si verifica quando un tile contiene due oggetti: uno con profondità molto bassa (ad esempio $z = 1$) e uno con profondità molto alta (ad esempio $z = 1000$). Inoltre, il tile contiene un numero molto elevato di luci, ma nessuna di queste luci illumina nessuno dei due oggetti. Avremo quindi che la lista delle luci associata al tile sarà molto lunga, ma nessuna di queste luci contribuirà all'illuminazione dei pixel del tile. Di conseguenza, per ogni pixel del tile, la GPU dovrà iterare su tutte le luci della lista, ma dato che nessuno dei due oggetti è illuminato da queste luci, tutti questi calcoli saranno completamente inutili.

Questo problema rende le prestazioni del Tiled Shading inconsistenti e dipendenti dalla vista, che generalmente non è desiderabile in un motore grafico.

Clustered Shading

Per risolvere il problema della discontinuità di profondità, Olsson et al. (2012) hanno introdotto il Clustered Shading. Questa tecnica estende la griglia 2D del Tiled Shading aggiungendo la profondità.

Il volume di vista della telecamera viene suddiviso in una griglia tridimensionale. Ogni cella di questa griglia prende il nome di Cluster. La suddivisione lungo l'asse Z non è solitamente lineare ma esponenziale, in modo tale da avere cluster più piccoli e precisi vicino alla telecamera e più grandi in lontananza.

Il processo segue tre fasi principali:

1. **Light Assignment:** Le luci vengono assegnate ai cluster 3D che intersecano il loro volume di influenza[2].

2. **Shading:** Quando un pixel viene renderizzato, la sua posizione (x, y, z) viene utilizzata per calcolare un indice tridimensionale (i, j, k) che identifica univocamente il cluster di appartenenza.

3. **Light Iteration:** Per ogni frammento dell'oggetto interessato, la GPU itera solo sulle luci associate al cluster di appartenenza.

La validità teorica di questa separazione tra geometria e illuminazione del Deferred Shading verrà verificata nel Test B. In tale sperimentazione, confronteremo la scalabilità delle architetture Forward e Deferred al variare del numero di sorgenti luminose, evidenziando il punto di saturazione della banda passante del G-Buffer previsto dalla teoria.

Andiamo ora ad introdurre le tecniche di ottimizzazione fondamentali applicabili prima dell'ingresso nella pipeline grafica, quindi da parte della CPU. L'obiettivo principale di queste tecniche è quello di ridurre la complessità geometrica della scena, diminuendo il numero di vertici e triangoli da elaborare, senza compromettere eccessivamente la qualità visiva dell'immagine finale.

1.3 Level Of Detail (LOD)

Una tecnica di ottimizzazione fondamentale è il **Level of Detail** (LOD). Questa strategia mira a diminuire il numero di poligoni da renderizzare in aree dove l'alta fedeltà visiva non è necessaria, come oggetti lontani dalla camera o parti della scena che occupano una piccola porzione dello schermo.

Oltre a ridurre il carico computazionale, il LOD riveste un ruolo cruciale nella prevenzione delle inefficienze legate alla rasterizzazione di triangoli chiamati *sub-pixel*. Quando la proiezione a schermo di una primitiva geometrica risulta inferiore alla dimensione di un singolo pixel, l'hardware grafico (che normalmente opera su blocchi indivisibili di 2×2 pixel per poter mappare correttamente le texture) è costretto a processare quattro pixel quando il frammento interessato risiede in solo uno di questi pixel, sprecando cicli di calcolo in un fenomeno noto come *quad overdraw*. Mantenendo la densità geometrica proporzionale alla distanza di visualizzazione, il LOD garantisce che i triangoli coprano un'area a schermo ottimale, massimizzando l'efficienza dello stadio di rasterizzazione e riducendo gli artefatti di aliasing geometrico.

1.3.1 Discrete LOD

Il *Discrete LOD* costituisce l'approccio tradizionale per la gestione della complessità geometrica e rimane, a oggi, il metodo più diffuso nelle applicazioni 3D industriali grazie alla sua affidabilità e semplicità implementativa.

[2]Ogni luce viene descritta da un volume di influenza, che rappresenta l'area dello spazio 3D in cui la luce ha un effetto significativo.

Tale approccio consiste nella generazione offline di diverse versioni della stessa mesh, ciascuna caratterizzata da una complessità geometrica progressivamente ridotta. Il livello di dettaglio minimo può essere rappresentato in vari modi: un metodo può essere attraverso una *billboard* (o *imposter*), ovvero un piano 2D "cartellone" che utilizza un'immagine dell'oggetto come texture per ingannare l'osservatore. In alternativa, per minimizzare la quantità di dati da renderizzare, si può ricorrere alla completa rimozione del modello. A run-time, il sistema seleziona dinamicamente il modello più appropriato da renderizzare basandosi su diverse metriche, come la distanza dell'oggetto dalla camera o la sua area in pixel sullo schermo.

L'efficienza di questo metodo risiede nell'assenza di calcoli geometrici complessi a run-time: il costo computazionale per la scelta del modello è trascurabile. Tuttavia, questo approccio comporta un maggiore consumo di memoria dovuto alla necessità di memorizzare duplicati della stessa mesh. Inoltre, il passaggio istantaneo tra un livello di dettaglio e l'altro può causare artefatti visivi noti come *popping*, in cui il cambio di geometria risulta percettibile all'osservatore. Nonostante ciò, l'uso di mesh statiche pre-calcolate permette di sfruttare formati di dati ottimizzati in base all'architettura di destinazione, garantendo prestazioni elevate.

Vediamo ora in quale modo è possibile selezionare il corretto LOD a run-time.

Criteri di selezione di LOD

I criteri di selezione si suddividono principalmente in metriche geometriche e approcci basati sulla percezione visiva:

- **Distanza dalla camera:** è la metrica più immediata, basata sulla distanza euclidea tra il punto di vista e l'oggetto; una volta superata una determinata soglia di distanza, viene caricata la mesh meno complessa. Tale approccio presenta tuttavia dei limiti significativi, in quanto non considera le dimensioni reali della mesh né la configurazione del campo visivo. Di conseguenza, si possono verificare sprechi di risorse computazionali renderizzando dettagli impercettibili per oggetti piccoli in primo piano, oppure un eccessivo degradamento visivo per strutture di grandi dimensioni situate a distanze elevate.

- **Area proiettata (Screen-space coverage):** valuta l'importanza visiva calcolando quanto spazio l'oggetto occupa effettivamente sullo schermo in pixel. Evita problemi legati alla scala degli oggetti, garantendo che modelli

complessi vengano utilizzati solo quando l'oggetto è sufficientemente grande da giustificare il dettaglio aggiuntivo. Il calcolo dell'area proiettata avviene tramite la proiezione dei vertici del *bounding box*[3]3D dell'oggetto nello spazio dello schermo e la determinazione del rettangolo 2D risultante. Tuttavia, questo metodo non è invariante alla rotazione dell'oggetto: ad esempio, un oggetto sottile orientato di taglio rispetto alla camera può avere una proiezione ridotta, portando a una selezione di LOD inappropriata. Per questo motivo, spesso vengono usate altre primitive di approssimazione, come sfere o ellissoidi, per calcolare l'area proiettata in modo più robusto.

- **Criteri basati sulla priorità:** in scenari complessi, si possono assegnare priorità agli oggetti in base alla loro rilevanza all'interno della scena.

- **Criteri basati sul movimento e la visuale periferica:** gli oggetti in rapido movimento o situati ai margini del campo visivo possono essere renderizzati con LOD inferiori, poiché l'occhio umano è meno sensibile ai dettagli in queste condizioni.

L'efficacia del Discrete LOD nel mitigare il costo geometrico verrà analizzata nel Test A1. In questa sede, valuteremo come la riduzione dinamica della complessità poligonale influenzi il frame time, confrontando tale approccio con tecnologie di geometria virtualizzata come Nanite, che operano su presupposti differenti.

1.3.2 Continuous LOD

Il *Continuous LOD* rappresenta un'evoluzione rispetto all'approccio discreto. Invece di selezionare tra un numero finito di modelli pre-calcolati, il continuous LOD gestisce una struttura dati progressiva capace di generare una rappresentazione della mesh della risoluzione desiderata.

Questa tecnica permette di adattare il numero di poligoni in modo fluido ed esatto al *budget* di rendering disponibile o all'errore visivo massimo, eliminando alla radice il problema del *popping* tra livelli discreti.

[3]Un **Bounding Box** è un volume geometrico semplificato (generalmente un parallelepipedo) che racchiude interamente la mesh di un oggetto. In questo contesto, esso funge da *proxy* computazionale: proiettare gli 8 vertici del box per stimare l'occupazione a schermo è un'operazione drasticamente più rapida rispetto alla proiezione delle migliaia di vertici che compongono la geometria reale.

Algoritmi di decimazione

La generazione di livelli di dettaglio continui si basa sull'applicazione iterativa di operatori di semplificazione locale. L'obiettivo è ridurre la complessità topologica[4]della mesh minimizzando l'impatto sulla sua morfologia[5].

Edge Collapse (Collasso del lato) L'operatore fondamentale nella maggior parte degli algoritmi moderni è l'*Edge Collapse*. Dato un lato l che connette i vertici $\mathbf{u}$ e $\mathbf{v}$, l'operazione unifica i due estremi in un singolo vertice $\bar{\mathbf{v}}$.

$$(\mathbf{u}, \mathbf{v}) \rightarrow \bar{\mathbf{v}} \tag{1.5}$$

A seguito del collasso, i triangoli che condividevano il lato l degenerano in segmenti e vengono rimossi dalla lista delle primitive, riducendo il conteggio totale dei poligoni di due unità per ogni operazione.

Vertex Split (Divisione del vertice) L'operazione inversa dell'Edge Collapse è il *Vertex Split*, che consente di ripristinare la geometria originale dividendo un vertice $\bar{\mathbf{v}}$ in due vertici distinti $\mathbf{u}$ e $\mathbf{v}$, ricostruendo i triangoli precedentemente rimossi.

$$\bar{\mathbf{v}} \rightarrow (\mathbf{u}, \mathbf{v}) \tag{1.6}$$

Pair Contraction Mentre l'Edge Collapse è vincolato a operare su vertici fisicamente connessi da un lato della mesh, il *Pair Contraction* estende l'applicabilità dell'operazione a coppie di vertici arbitrari (v_i, v_j).

Tali vertici non necessitano di essere adiacenti topologicamente, ma vengono considerati connessi da un "lato virtuale" se soddisfano un criterio di prossimità spaziale, come una distanza inferiore a una soglia prefissata ϵ ($||v_i - v_j|| < \epsilon$).

Vediamo ora come queste due operazioni fondamentali vengono utilizzate negli algoritmi di decimazione.

[4]**Topologia di una mesh:** è la struttura di connessione tra i vertici. Descrive *come* gli elementi sono collegati tra di loro, indipendentemente dalla loro posizione nello spazio.

[5]**Morfologia di una mesh:** è la forma geometrica o l'aspetto visivo dell'oggetto nello spazio tridimensionale. Descrive *dove* si trovano gli elementi e definisce caratteristiche come volume, curvatura, ecc.

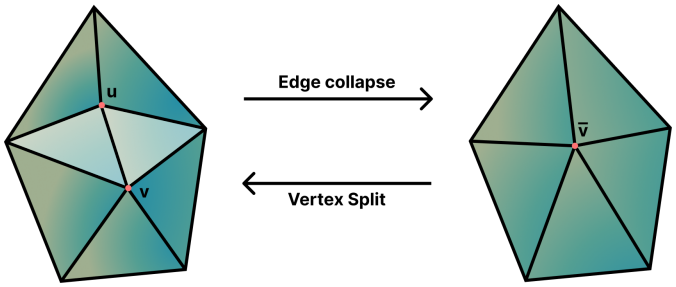

Figura 1.2: Operazioni di Edge Collapse e Vertex Split.

Strutture dati per il Continuous LOD: La Vertex Hierarchy

Per implementare un sistema di Continuous LOD è necessario organizzare la storia delle semplificazioni in una struttura dati che permetta di navigare bidirezionalmente tra la risoluzione minima e quella massima in tempo reale.

La struttura dati piú utilizzata per questo scopo è la **Vertex Hierarchy** (o *Vertex Tree*), introdotta formalmente nel concetto di *Progressive Meshes* da Hugues Hoppe nel 1996.

Progressive Mesh Una mesh arbitraria M può essere rappresentata come una mesh di base a bassissima risoluzione M^0 seguita da una sequenza ordinata di n operazioni di vertex split:

$$M = M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} \dots \xrightarrow{vsplit_{n-1}} M^n \tag{1.7}$$

Dove M^n è la mesh originale alla massima risoluzione. Ogni passo rappresenta una trasformazione atomica invertibile:

- **Edge Collapse (*ecol*):** Riduce il dettaglio ($M^i \to M^{i-1}$), fondendo due vertici figli in un vertice genitore.
- **Vertex Split (*vsplit*):** Aumenta il dettaglio ($M^{i-1} \to M^i$), dividendo un vertice genitore in due vertici figli distinti.

Andiamo ora a generare una albero binario (per ogni mesh) attraverso queste dipendenze genitori-figli. Le foglie dell'albero rappresentano i vertici della mesh originale ad alta risoluzione (M^n). I nodi interni rappresentano i vertici generati dalle operazioni di collasso. Le radici rappresentano i vertici della mesh di base (M^0).

Il Fronte Attivo A run-time, lo stato corrente della mesh visualizzata corrisponde a un "taglio" (o *fronte attivo*) che attraversa la gerarchia dei vertici.

- Se il criterio di LOD richiede più dettaglio in una zona (es. la telecamera si avvicina), il sistema esegue un *Vertex Split* sui nodi del fronte attivo, spostando il taglio verso il basso (verso le foglie).

- Se il dettaglio è superfluo, il sistema esegue un *Edge Collapse*, spostando il taglio verso l'alto (verso le radici).

Metriche di valutazione dell'errore geometrico

La *Vertex Hierarchy* deve essere in qualche modo riempita con una serie di operazioni di semplificazione invertibili. Questa serie di operazioni può essere generata in vari modi, ma negli anni lo standard de facto si è rivelato essere il metodo QEM introdotto da Garland e Heckbert nel 1997.

Quadric Error Metric (QEM) L'algoritmo di decimazione basato su QEM opera attraverso la contrazione iterativa di coppie di vertici $(\mathbf{v}_1, \mathbf{v}_2)$ in un unico vertice $\bar{\mathbf{v}}$. Una coppia è considerata valida per la semplificazione se soddisfa una delle seguenti condizioni:

1. $(\mathbf{v}_1, \mathbf{v}_2)$ costituisce un lato esistente della mesh;

2. $||\mathbf{v}_1 - \mathbf{v}_2|| < t$, dove t è una soglia di tolleranza prefissata (permettendo la chiusura di buchi o l'unione di componenti disgiunte vicine).

Per quantificare l'errore geometrico introdotto dalla semplificazione, Garland e Heckbert associano a ciascun triangolo della mesh una *quadrica dell'errore fondamentale* $\mathbf{K}_p$. Dato il piano p di equazione $ax + by + cz + d = 0$ su cui giace il triangolo, la matrice $\mathbf{K}_p$ è definita come:

$$\mathbf{K}_p = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix} \tag{1.8}$$

Questa matrice permette di calcolare la distanza quadrata di un punto qualsiasi dal piano p. Per ogni vertice $\mathbf{v}$ della mesh, viene quindi calcolata una matrice dell'errore $\mathbf{Q}$ sommando le quadriche fondamentali di tutti i piani p incidenti a tale vertice:

$$\mathbf{Q} = \sum_{p \in planes(\mathbf{v})} \mathbf{K}_p \tag{1.9}$$

Quando si valuta la contrazione di una coppia $(\mathbf{v}_1, \mathbf{v}_2) \rightarrow \bar{\mathbf{v}}$, la matrice dell'errore $\bar{\mathbf{Q}}$ associata al nuovo vertice $\bar{\mathbf{v}}$ è data dalla somma delle matrici dei vertici originali:

$$\bar{\mathbf{Q}} = \mathbf{Q}_1 + \mathbf{Q}_2 \tag{1.10}$$

L'errore geometrico $\Delta(\bar{\mathbf{v}})$ introdotto spostando i vertici originali nella nuova posizione $\bar{\mathbf{v}}$ è definito dalla forma quadratica:

$$\Delta(\bar{\mathbf{v}}) = \bar{\mathbf{v}}^T \bar{\mathbf{Q}} \bar{\mathbf{v}} \tag{1.11}$$

Il nostro obiettivo è trovare la posizione ottimale $\bar{\mathbf{v}}$ che minimizzi tale errore. Per fare ciò, calcoliamo le derivate parziali $\frac{\partial \Delta}{\partial x} = \frac{\partial \Delta}{\partial y} = \frac{\partial \Delta}{\partial z} = 0$, che equivale a risolvere il seguente sistema lineare:

$$\begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \bar{\mathbf{v}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{1.12}$$

dove l'ultima riga [0 0 0 1] sostituisce le derivate rispetto alla componente omogenea per imporre che $\bar{\mathbf{v}}$ sia un punto valido nello spazio (con $w = 1$). Assumendo che la matrice dei coefficienti sia invertibile, la posizione ottimale è data da:

$$\bar{\mathbf{v}} = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{1.13}$$

Nel caso in cui la matrice non sia invertibile (singolare), l'algoritmo adotta una strategia di fallback: si cerca la posizione ottimale lungo il segmento $\mathbf{v}_1\mathbf{v}_2$. Se anche questa ricerca fallisce, si valuta l'errore Δ nei tre punti discreti: $\mathbf{v}_1$, $\mathbf{v}_2$ e il punto medio $\mathbf{v}_m = \frac{\mathbf{v}_1+\mathbf{v}_2}{2}$, scegliendo quello che minimizza il costo.

Approcci View-Independent vs View-Dependent

Gli algoritmi di Continuous LOD possono essere classificati in base alla dipendenza dal punto di vista dell'osservatore.

- **View-Independent LOD:** La semplificazione è uniforme su tutta la superficie dell'oggetto. Questo approccio è utile per generare i livelli discreti offline. Fallisce invece nel caso di oggetti molto grandi, dove sezioni molto distanti della mesh saranno renderizzate a livelli di dettaglio molto alti.

In questo caso, il fronte attivo della Vertex Hierarchy risulterà orizzontale, mantenendo lo stesso livello di dettaglio su tutta la mesh.

- **View-Dependent LOD:** La risoluzione della mesh viene adattata localmente in tempo reale in funzione della telecamera. In questo scenario, la mesh può presentare contemporaneamente aree ad altissima densità poligonale (vicine all'osservatore) e aree fortemente semplificate (distanti), ottimizzando il vertex count in modo molto più aggressivo rispetto ai metodi uniformi. L'active cut della Vertex Hierarchy risulterà irregolare, con sezioni della mesh che si estendono verso le foglie - dove il dettaglio è molto alto - e altre che risalgono verso le radici.

1.3.3 Gestione delle transizioni di LOD

Analizziamo ora tecniche per mitigare gli artefatti visivi dovuti al cambio di LOD nel caso discreto. Le seguenti sezioni descrivono due delle tecniche piu' comuni: l'*alpha blending* e il *geomorphing*.

Alpha Blending

Questo approccio prevede che, in prossimità della soglia di cambio tra due livelli di dettaglio (LOD_i e LOD_{i+1}), venga definita una *zona di transizione* spaziale. All'interno di questo intervallo, entrambi i modelli vengono renderizzati simultaneamente nella stessa posizione, miscelando la loro visibilità tramite un valore di opacità α interpolato linearmente.

Ad esempio, con una soglia $D_{soglia} = 100m$ e un'ampiezza di transizione $\Delta d = 20m$, l'interpolazione avverrà nell'intervallo $[90m, 110m]$:

- Il modello ad alta risoluzione (LOD_i) vedrà il suo valore alpha decrescere da 1.0 a 0.0.

- Il modello a bassa risoluzione (LOD_{i+1}) vedrà il suo valore alpha crescere da 0.0 a 1.0.

Sebbene questa tecnica elimini quasi completamente le discontinuità visive, essa introduce un costo computazionale non trascurabile: durante la transizione, il motore grafico deve processare due mesh differenti per un singolo oggetto e gestire il blending tra di esse.

Geomorphing

Mentre l'Alpha Blending agisce sulla visibilità dei pixel, il **Geomorphing** opera direttamente sulla geometria.

L'idea alla base è interpolare linearmente la posizione dei vertici tra lo stato corrente e lo stato target durante il lasso di tempo della transizione.

Considerando un'operazione di *Edge Collapse* in cui il vertice $\mathbf{u}$ deve collassare nel vertice $\mathbf{v}$, definiamo un parametro di transizione $t \in [0, 1]$. La posizione renderizzata del vertice $\mathbf{p}(t)$ è calcolata come:

$$\mathbf{p}(t) = (1 - t) \cdot \mathbf{u} + t \cdot \mathbf{v} \tag{1.14}$$

Quando $t = 0$, il vertice si trova nella sua posizione originale $\mathbf{u}$. Man mano che la transizione procede ($t \to 1$), il vertice scivola visivamente verso la posizione di destinazione $\mathbf{v}$. Quando $t = 1$, la geometria è identica a quella semplificata e la topologia può essere aggiornata rimuovendo il vertice in eccesso senza alcuno scatto visivo.

Poiché viene renderizzata una sola mesh che cambia forma, non vi è una enorme perdita di prestazioni. Tuttavia l'interpolazione deve essere calcolata per ogni vertice a ogni frame, aumentando il carico sulla GPU per la fase di vertex processing.

1.4 Tecniche di visibilità: Culling

Il termine **Culling** deriva dalla lingua inglese, nello specifico dal verbo "to cull", che significa "eliminare" o "scartare" da un insieme. Nel contesto della grafica 3D, questo é esattamente ció che vanno a fare le tecniche di culling: identificare e scartare quegli oggetti (o porzioni di oggetti) che non contribuiscono all'immagine finale, evitando di sprecare risorse computazionali nel loro rendering.

La fase di culling può teoricamente avvenire in qualsiasi stadio della pipeline grafica ma, dato che il triangolo piú veloce da renderizzare è quello che non viene renderizzato, la maggior parte delle tecniche di culling vengono applicate a livello di applicazione dalla CPU, prima dell'ingresso dei dati nella pipeline grafica.

L'algoritmo di culling ideale dovrebbe essere in grado di far procedere attraverso la pipeline grafica solo l'insieme visibile esatto (EVS - Exact Visible Set) di primitive ad ogni frame. Tuttavia, a causa della complessitá geometrica delle scene 3D moderne e delle limitazioni computazionali, raggiungere questo obiettivo

in modo perfetto è praticamente impossibile. Di conseguenza, le tecniche di culling pratiche cercano di individuare l'insieme visibile potenziale (PVS - Potentially Visible Set), ovvero un'approssimazione dell'EVS.

Se il PVS include completamente l'EVS in modo tale che solo geometria invisibile venga scartata, si parla di **culling conservativo**. Al contrario, se il PVS esclude parte dell'EVS, causando la rimozione di geometria visibile, si parla di **culling approssimato**.

Le tecniche di culling che verranno descritte in questa sezione operano a diversi livelli di granularità, dalla scena globale fino ai singoli poligoni. Queste tre tecniche sono il **Backface Culling**, il **Frustum Culling** e l'**Occlusion Culling**.

1.4.1 Backface Culling

Si consideri un oggetto solido e opaco, quale una sfera, immerso in uno spazio tridimensionale. È immediato osservare che, in qualsiasi istante, approssimativamente la metà della sua superficie è orientata in direzione opposta rispetto al punto di vista dell'osservatore e risulta, di conseguenza, invisibile. Questo principio geometrico suggerisce una fondamentale ottimizzazione: le primitive che costituiscono la porzione posteriore dell'oggetto non contribuiscono alla generazione dell'immagine finale e il loro processamento rappresenta uno spreco di risorse computazionali. La tecnica che implementa questa rimozione è nota come **Backface Culling**. Nel seguito verranno esaminate due varianti di tale metodo: la prima opera con una granularità fine a livello di singola primitiva (poligono), mentre la seconda agisce su insiemi aggregati di geometria (cluster di poligoni).

Backface culling a livello di poligono

Questo metodo effettua culling un poligono alla volta. Vedremo due metodi differenti per implementare questo tipo di culling: uno che opera nello spazio tridimensionale di vista (*View Space*) nella fase di vertex processing e un altro che opera nello spazio bidimensionale dello schermo (*Screen Space*) in fase di fragment processing.

Culling in View Space (Vertex processing) Opera in view space, prima che avvenga la divisione prospettica. Il criterio di visibilità si basa sull'angolo compreso tra il vettore normale della superficie $\vec{N}$ e il vettore direzione di vista $\vec{V}$ (il vettore che congiunge il vertice della telecamera al poligono).

Dato un triangolo con normale $\vec{N}$ e un vettore di vista $\vec{V}$ (diretto dalla superficie verso la camera), la condizione di visibilità è determinata dal loro prodotto scalare:

$$\vec{N} \cdot \vec{V} > 0 \tag{1.15}$$

Se il prodotto scalare è positivo, l'angolo tra i due vettori è inferiore a 90°, implicando che la faccia è rivolta verso l'osservatore. Se negativo o nullo, la faccia è "di spalle" o di taglio, e può essere scartata.

Culling in Screen Space (Fragment processing) Questo è l'approccio comunemente implementato in hardware nello stadio di rasterizzazione, operando dopo la proiezione dei vertici in coordinate dello schermo. Il controllo non si basa più sulla normale 3D, ma sul winding order dei vertici.

Dato un triangolo proiettato con vertici $v_1 = (x_1, y_1)$, $v_2 = (x_2, y_2)$ e $v_3 = (x_3, y_3)$, è possibile calcolare la sua *Area con Segno* utilizzando il determinante della matrice dei vettori dei lati:

$$Area = \frac{1}{2} \det \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{bmatrix} = \frac{1}{2}[(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)] \tag{1.16}$$

Il segno del risultato determina l'orientamento:

- **Segno Positivo:** I vertici sono ordinati in senso antiorario.
- **Segno Negativo:** I vertici sono ordinati in senso orario.

Se l'area del triangolo è negativa, allora significa che il triangolo mostra la sua faccia interna alla telecamera e di conseguenza viene scartato prima della generazione dei frammenti. Questo metodo è robusto e computazionalmente economico.

Clustered Backface Culling

Negli anni, gli algoritmi di backface culling sono stati estesi per operare su insiemi di poligoni chiamati cluster, piuttosto che su singoli triangoli. Questo approccio, noto come **Clustered Backface Culling**, mira a migliorare l'efficienza computazionale riducendo il numero totale di test di visibilità necessari. Uno dei concetti alla base della maggior parte di questo tipo di algoritmi e' il **cono normale**.

Definizione del Cono Normale Per ogni cluster, viene pre-calcolato (spesso offline) un cono che rappresenta l'orientamento complessivo di tutte le facce contenute nel gruppo. Questo cono è definito da due parametri:

- Un asse centrale unitario $\vec{n}_c$, che rappresenta la direzione media delle normali.
- Una semi-apertura angolare α, tale per cui le normali di tutti i triangoli del cluster deviano da $\vec{n}_c$ per un angolo non superiore ad α.

Criterio di Culling A run-time, per determinare la visibilità del cluster, si confronta il cono normale con la posizione della telecamera. Affinché un singolo triangolo sia visibile, la sua normale deve formare un angolo minore di 90° con il vettore vista.

Estendendo questo concetto al cluster, possiamo affermare che l'intero gruppo è "invisibile" se l'intero cono delle normali **non** punta in direzione della telecamera. Geometricamente, ciò avviene quando l'asse del cono $\vec{n}_c$ forma con il vettore vista $\vec{v}$ (diretto dalla telecamera al cluster) un angolo maggiore di $90° + \alpha$.

La condizione matematica per il culling è data dal prodotto scalare:

$$\vec{n}_c \cdot \vec{v} > \sin(\alpha) \tag{1.17}$$

Dove $\vec{v}$ è il vettore unitario dalla telecamera al centro del cluster. Se questa disuguaglianza è vera, significa che anche il triangolo "più favorevole" all'interno del cluster è comunque orientato in senso opposto all'osservatore, e quindi l'intero gruppo può essere scartato in sicurezza.

1.4.2 Frustum Culling

É la tecnica di culling piú comune e diffusa nelle applicazioni grafiche in tempo reale, data la sua semplicità implementativa e l'efficacia nel ridurre il carico di rendering.

L'idea alla base di questa tecnica di culling è di evitare di processare tutti quegli oggetti che si trovano completamente al di fuori del volume di vista della telecamera.

Andiamo quindi a verificare per ogni oggetto l'intersezione con il volume di vista.

Bounding Volume Hierarchies (BVH)

L'applicazione del test di intersezione su ogni singola primitiva geometrica della scena risulterebbe computazionalmente proibitiva ($O(N)$ dove N è il numero di oggetti). Per ottimizzare questo processo, i motori grafici moderni organizzano la scena all'interno di strutture dati gerarchiche note come **Bounding Volume**[6]**Hierarchies** (BVH).

Gerarchia e Attraversamento Una BVH organizza questi volumi in una struttura ad albero. Il nodo radice racchiude l'intera scena; ogni nodo interno racchiude i volumi dei suoi figli, fino ad arrivare alle foglie che contengono la geometria effettiva.

Durante la fase di culling, l'algoritmo attraversa l'albero in modo ricorsivo:

1. Si testa il volume del nodo corrente contro i piani del volume di vista.

2. **Rejection:** Se il volume è completamente esterno, l'intero sottoalbero di tale nodo viene potato. Questo permette di scartare migliaia di oggetti con un singolo test geometrico.

3. **Acceptance:** Se il volume interseca il frustum o è completamente interno, si procede a testare ricorsivamente i nodi figli.

Questo approccio gerarchico riduce la complessità media del culling da lineare $O(N)$ a logaritmica $O(\log N)$, rendendo possibile la gestione di scene complesse in tempo reale.

1.4.3 Occlusion Culling

L'Occlusion Culling è una tecnica avanzata che mira a identificare e scartare quegli oggetti che, pur trovandosi all'interno del frustum, sono completamente nascosti da altri oggetti più vicini alla telecamera.

Questo tipo di problema in teoria viene già affrontato dal processo di rasterizzazione, in particolare dalla fase di depth testing utilizzando lo Z-buffer. Esso contiene il valore di profondità del frammento più vicino per ogni pixel.

[6]Un Bounding Volume è una forma geometrica semplice che funge da approssimazione per un oggetto complesso, racchiudendolo completamente. Solitamente si scelgono forme seplici come cubi o sfere.

Se un frammento viene processato e risulta essere più lontano rispetto al valore memorizzato nello Z-buffer, viene scartato (occluso) e non contribuisce all'immagine finale.

Tuttavia, prima di essere occluso, il frammento deve comunque essere processato consumando risorse computazionali preziose. L'Occlusion Culling cerca di prevenire questo spreco identificando oggetti completamente nascosti prima di inviare la geometria alla GPU, evitando così di inviare alla GPU geometria che non avrà alcun impatto visivo.

Hardware Occlusion Queries

L'approccio classico supportato nativamente dalle moderne API grafiche è rappresentato dalle *Hardware Occlusion Queries*. Il funzionamento prevede un rendering virtuale (ovvero senza visualizzazione a schermo) del bounding volume dell'oggetto. La GPU conta il numero di campioni che superano il depth test (ovvero quelli visibili) confrontandoli con lo Z-Buffer e restituisce il valore alla CPU. Se il conteggio è zero, l'oggetto è completamente occluso e la sua geometria non viene inviata.

Tuttavia, questa tecnica introduce un problema critico di sincronizzazione noto come **CPU-GPU Stall**. Poiché la CPU deve attendere il risultato dell'interrogazione dalla GPU prima di decidere se renderizzare l'oggetto, si interrompe il parallelismo tra i due processori. Per mitigare la latenza, i motori grafici utilizzano spesso i risultati del frame precedente (approccio temporale), accettando però il rischio di artefatti visivi durante movimenti rapidi della camera.

Hierarchical Z-Buffer (HZB)

Per superare i colli di bottiglia e la latenza intrinseca delle Hardware Occlusion Queries, lo stato dell'arte del rendering real-time adotta la tecnica nota come **Hierarchical Z-Buffer** (HZB). Questo approccio combina una struttura di partizione spaziale della scena con una rappresentazione gerarchica del buffer di profondità.

Analizziamo le due strutture dati fondamentali su cui si basa l'algoritmo.

Octree L'Octree è una struttura dati spaziale che partiziona ricorsivamente il volume della scena. Partendo da un Bounding Box[7] "radice" che racchiude l'intero ambiente, il volume viene suddiviso in ottanti. Il processo di suddivisione viene reiterato per ogni nodo finché non viene soddisfatto un criterio di arresto, come il raggiungimento di una dimensione minima del nodo o di un numero massimo

di primitive geometriche contenute. Questa organizzazione permette di trattare interi gruppi di oggetti come singole entità spaziali.

Z-Pyramid (Depth Pyramid) La Z-Pyramid è una catena di mipmap[8]costruita a partire dallo Z-Buffer del frame corrente.

Il livello base (Livello 0) corrisponde al buffer di profondità a piena risoluzione. Ogni livello successivo $(k + 1)$ viene generato riducendo la risoluzione del livello precedente (k) di un fattore due su entrambi gli assi.

Operativamente, ogni texel[9]del livello superiore viene calcolato esaminando il corrispondente blocco di 2×2 texel del livello inferiore e memorizzando il valore di profondità **massimo** (il più lontano dalla camera). Questo approccio conservativo garantisce che il valore nella piramide rappresenti sempre la profondità "più sicura" per il test di occlusione.

L'algoritmo di culling sfrutta queste strutture per scartare rapidamente ampie porzioni della scena. Il processo segue questi passaggi:

1. **Attraversamento Front-to-Back:** Si analizzano i nodi dell'Octree partendo dalla radice. È cruciale visitare i nodi seguendo un ordinamento *front-to-back*, ovvero dal più vicino alla camera al più lontano. Questo massimizza l'efficienza, poiché gli oggetti vicini, venendo processati per primi, hanno maggiore probabilità di occludere i nodi successivi, permettendo di interrompere presto l'attraversamento dell'octree.

2. **Proiezione:** Per ogni nodo visitato, si calcola il bounding box e lo si proietta nello spazio schermo, determinando il rettangolo minimo che lo racchiude.

3. **Selezione del Livello:** Si identifica il livello k della Z-Pyramid più appropriato per il test. Il livello ideale è quello in cui la proiezione del bounding box occupa un'area ridotta, coperta al massimo da una griglia di 2×2 texel. Questo permette di ottenere un valore di profondità rappresentativo con un numero costante di letture in memoria, scegliendo il livello di dettaglio ottimale per il test di occlusione.

[7]Un Bounding Volume di forma cubica.

[8]É una sequenza di versioni pre-calcolate della stessa immagine a risoluzione progressivamente dimezzata. Tradizionalmente sono utilizzate per il filtraggio delle texture, ma nel contesto dello Z-Buffer vengono adattate per conservare i valori di profondità massimi.

[9]L'unità discreta minima di una texture grafica. Risiede nello spazio della texture ed è indirizzato tramite coordinate (u, v) (UV).

4. **Test di Occlusione:** Si confronta la profondità minima del nodo (il punto del bounding box più vicino alla camera) con il valore di profondità massimo letto nella Z-Pyramid al livello k. Se la profondità del nodo è maggiore del valore nella piramide, significa che il nodo è interamente nascosto dalla geometria già renderizzata: esso viene quindi scartato insieme a tutto il suo sotto-albero. In caso contrario, il nodo è potenzialmente visibile e l'algoritmo procede ricorsivamente analizzando i suoi figli, fino a raggiungere le foglie contenenti la geometria da renderizzare.

I meccanismi di Frustum e Occlusion Culling, qui descritti, costituiscono il fulcro del Test A2. La sperimentazione confronterà l'efficienza dei sistemi di culling basati su CPU (più comuni nei motori tradizionali) rispetto alle controparti gestite dalla GPU, misurando l'impatto sul carico di lavoro dei thread principali.

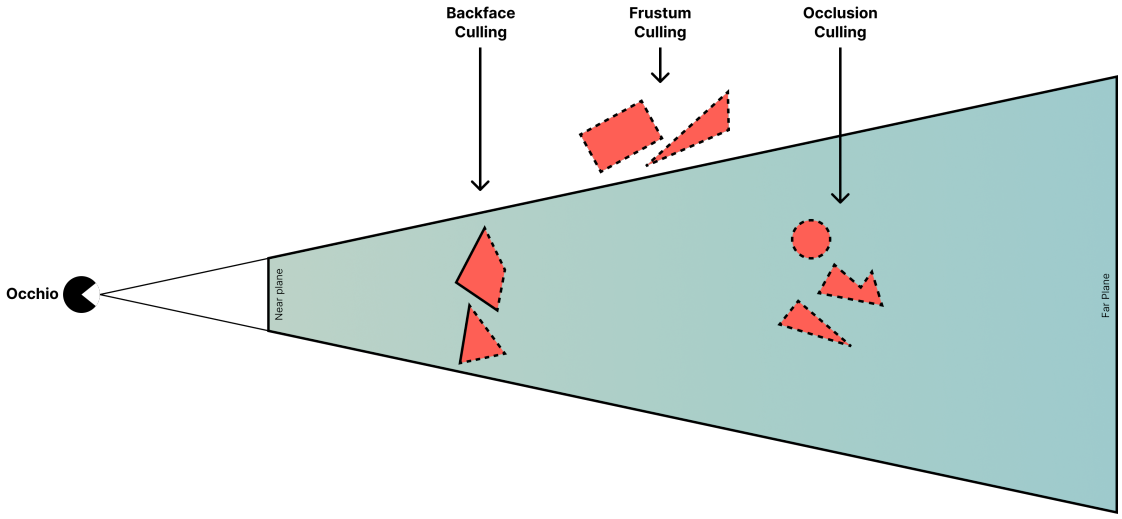

Figura 1.3: Illustrazione delle tecniche di culling: Backface Culling, Frustum Culling e Occlusion Culling. Le linee tratteggiate rappresentano le superfici escluse, mentre le linee continue indicano la geometria effettivamente processata.

1.5 Ottimizzazione memoria e riduzione delle Draw Call

Negli anni, l'aumento esponenziale della potenza di calcolo delle GPU ha permesso la computazione di scene 3D sempre piú complesse e dettagliate. Non é stato questo il caso per quanto riguarda le prestazioni delle memorie video e dei bus di comunicazione tra CPU e GPU.

Infatti, il principale collo di bottiglia delle applicazioni grafiche in tempo reale si sta gradualmente spostando dalla potenza di calcolo grezza alla larghezza di banda e alla latenza delle memorie.

In sistemi dedicati all'utilizzo grafico intensivo, come le console di gioco, la separazione tra CPU e GPU (e di conseguenza tra le loro memorie) é stata eliminata adottando architetture incentrate sulle APU (Accelerated Processing Units), in cui entrambi i processori condividono lo stesso pool di memoria ad alta velocità.

Tuttavia, nella maggior parte dei calcolatori, la CPU e la GPU rimangono entità separate con memorie distinte, rendendo cruciale l'ottimizzazione del trasferimento dei dati tra i due.

Oltre agli stadi di elaborazione del vertex processing, rasterization e fragment processing, a scopo di analisi delle prestazioni, introduciamo un ulteriore stadio chiamato Application Stage, che rappresenta lo stadio in cui la CPU esegue la logica di alto livello dell'applicazione, e dove essa prepara le istruzioni e i dati da inviare alla GPU.

1.5.1 Application Stage

I principali problemi di efficienza nell'Application Stage derivano dalla lentezza di accesso alla memoria RAM, dove sono situate le strutture dati essenziali per il rendering. Se non si è in grado di mantenere un flusso costante di dati alla GPU, si rischia di incorrere in stalli che interrompono il parallelismo tra i due processori (come nel caso delle Hardware Occlusion Queries), causando un drastico calo delle prestazioni.

Nei sistemi moderni, per ovviare a questo problema, vi è un pesante utilizzo di memorie cache. Per mantenere buone prestazioni lato applicazione, si concentrano i propri sforzi a minimizzare il numero di **cache miss** sfruttando principi come la località spaziale e temporale dei dati.

- **Località Spaziale:** I dati che si trovano vicini tra loro in memoria tendono ad essere utilizzati insieme. Ad esempio, organizzare i vertici di una mesh in modo contiguo e ordinato può migliorare significativamente le prestazioni.

- **Località Temporale:** I dati che sono stati utilizzati di recente hanno una maggiore probabilità di essere utilizzati nuovamente nel prossimo futuro. Ad esempio, mantenere in cache le texture più frequentemente utilizzate può ridurre drasticamente i tempi di accesso.

Nel caso in cui si abbiano piú attributi legati a un vertice, è importante organizzare questi dati in modo da massimizzare la coerenza di accesso. Ad esempio, utilizzare un formato di buffer interleaved (dove tutti gli attributi di un vertice sono memorizzati contigui) può migliorare le prestazioni rispetto a un formato di buffer separato (dove ogni attributo è memorizzato in un buffer distinto).

Un altra frequente causa di cache miss è rappresentata dall'uso eccessivo di puntatori, come ad esempio linked lists o grafi complessi. Questo perchè, seguire un puntatore ad un altro puntatore (pointer indirection) causa salti all'interno della memoria, salti difficili da prevedere e che spesso portano a cache miss. Per questo motivo, è preferibile utilizzare strutture dati piatte e contigue, come array o buffer, che permettono un accesso più prevedibile e ottimizzato.

Per aumentare l'efficienza della lettura dei dati in cache, è importante allineare correttamente le strutture dati. L'allineamento dei dati si riferisce alla disposizione dei dati in memoria in modo che i loro indirizzi siano multipli di una certa potenza di due. Questo è importante perché molte architetture hardware accedono alla memoria in blocchi di dimensioni fisse (cache lines) e un accesso non allineato può richiedere più cicli di clock per essere completato, causando un calo delle prestazioni.

1.5.2 Da Application Stage a Vertex Stage (Comunicazione CPU-GPU)

La comunicazione tra CPU e GPU avviene tramite chiamate alle API grafiche. Queste API forniscono un'interfaccia per la gestione delle risorse (come buffer e texture) e per l'invio di comandi di rendering.

Esse si rivelano strumenti molto efficienti e indispensabili per la gestione della comunicazione tra i due processori, ma é importante utilizzarle in modo appropriato per evitare colli di bottiglia. Questo perché ogni chiamata a un'API grafica comporta un certo overhead, dovuto alla necessità di sincronizzazione e alla gestione delle risorse da parte del driver grafico. Pertanto, é fondamentale applicare le seguenti best practice:

- **Minimizzazione delle draw calls (batching):** Vedi sezione 2.5.3.
- **Minimizzazione dei cambi di stato:** Il cambio di stato della GPU rappresenta una delle operazioni piú costose in termini di prestazioni. Ogni volta che si cambia un parametro di rendering (anche solamente la texture utilizzata, o lo shader), la GPU deve eseguire una serie di operazioni di

reconfigurazione, causando un overhead significativo. Il costo di cambio di stato varia in base all'operazione specifica, alcune delle piú comuni in ordine decrescente di costo sono:

  - Cambio di render target (ad esempio, passare da un framebuffer a un altro).
  - Cambio di shader
  - Cambio di texture
  - Aggiornamento di uniformi o costanti

  La soluzione naturale a questo problema é quella di raggruppare le operazioni di rendering in modo da minimizzare il numero di cambi di stato. Ad esempio, organizzare gli oggetti da renderizzare in base allo shader utilizzato, o alla texture applicata, permette di eseguire un numero minore di cambi di stato e quindi migliorare le prestazioni complessive.

- **Utilizzo del corretto tipo di buffer:** Le API grafiche offrono diversi tipi di buffer (ad esempio, Vertex Buffer Object, Uniform Buffer Object, Shader Storage Buffer Object) che sono ottimizzati per specifici tipi di dati e pattern di accesso. Utilizzare il tipo di buffer più appropriato per i dati che si stanno inviando può migliorare significativamente le prestazioni. Ad esempio, utilizzare un Uniform Buffer Object per inviare dati che cambiano poco frequentemente (come le trasformazioni di un oggetto statico) é più efficiente rispetto a utilizzare un Vertex Buffer Object, che é ottimizzato per dati che cambiano frequentemente (come le posizioni dei vertici).

1.5.3 Batching

Ogni draw call rappresenta un comando di rendering che viene inviato alla GPU. Se si inviano troppe draw calls, si rischia di saturare il bus di comunicazione e di incorrere in stalli. Un esempio di questo comportamento è il *"Small Batch Problem"*. Esso consiste in uno scenario dove si ha una scena contenente tanti oggetti composti da un basso numero di vertici. Se si invia una draw call per ogni oggetto, si rischia di saturare il bus con un elevato numero di comandi, causando un drastico calo delle prestazioni. La soluzione a questo problema è quella di raggruppare (*batching*) più mesh in una sola mesh composta da molti poligoni, riducendo così il numero totale di comandi inviati alla GPU e migliorando le prestazioni complessive.

Esistono due approcci principali a questa tecnica, distinti dalle modalità temporali di esecuzione.

Static Batching

Lo Static Batching è una tecnica di ottimizzazione offline, ideale per mesh statiche. Le mesh statiche che condividono lo stesso materiale vengono unite in un'unica, grande mesh combinata. I vertici vengono pre-trasformati dallo spazio locale allo spazio mondo e memorizzati in un unico Vertex Buffer.

- **Vantaggi:** L'impatto sulla CPU a run-time è quasi nullo, poiché non sono necessari calcoli di trasformazione o setup ripetuti.
- **Svantaggi:** Il limite principale è l'occupazione di memoria. Poiché la geometria viene duplicata fisicamente nel buffer combinato, renderizzare 100 copie di una mesh con Static Batching richiede 100 volte la memoria della singola mesh. Inoltre, il Frustum Culling perde di granularità: se una porzione del batch è visibile, l'intero blocco geometrico deve essere processato dalla GPU, potenzialmente sprecando risorse di vertex processing.

Dynamic Batching

Il Dynamic Batching opera a run-time. Ad ogni frame, la CPU raccoglie i vertici di oggetti (anche dinamici) che condividono lo stesso materiale, applica le trasformazioni di posizione e rotazione, e copia il risultato in un Vertex Buffer che viene poi inviato alla GPU.

- **Vantaggi:** Permette di ridurre le draw calls anche per oggetti che si muovono o ruotano.
- **Svantaggi:** Questa tecnica impone un carico di lavoro elevato sulla CPU per la trasformazione dei vertici. Per questo motivo, il Dynamic Batching è efficace solo per mesh con un numero molto ridotto di vertici. Se applicato a mesh complesse, il tempo speso dalla CPU per il batching supera il guadagno ottenuto dalla riduzione delle draw calls.

1.5.4 GPU Instancing

Il GPU Instancing rappresenta l'evoluzione moderna delle tecniche di riduzione delle draw calls. Esso risolve sia i problemi di memoria tipici del batching statico, sia il carico computazionale del batching dinamico.

Questa tecnica permette di renderizzare un numero arbitrario di copie della stessa mesh (dette "istanze") utilizzando una singola Draw Call, senza duplicare la geometria in memoria.

Principio di Funzionamento

Nell'architettura dell'Instancing, i dati vengono separati in due parti:

1. **Vertex Buffer (Condiviso):** Contiene la geometria del modello, caricata in VRAM una sola volta.

2. **Instance Buffer (Per-Istanza):** Contiene i dati che variano per ogni copia, tipicamente la matrice di trasformazione, il colore o altri parametri.

Durante il rendering, il Vertex Shader viene invocato per ogni vertice di ogni istanza. La GPU utilizza un identificativo univoco per accedere all'array dei dati di istanza e posizionare correttamente la geometria nella scena.

Il potenziale del GPU Instancing nell'abbattere il numero di Draw Call sarà quantificato nel Test A1, dove utilizzeremo una griglia di 10.000 sfere per isolare il risparmio computazionale rispetto a un approccio di rendering tradizionale.

1.6 Gestione Avanzata delle Texture

L'impatto delle texture sulle prestazioni è spesso sottovalutato: il campionamento delle texture è una delle operazioni più costose in termini di latenza e consumo di larghezza di banda della memoria. L'ottimizzazione in questo stadio non riguarda solamente la riduzione dello spazio su disco attraverso compressione, ma soprattutto la gestione dell'accesso ai dati in VRAM per massimizzare l'efficienza della cache hardware.

1.6.1 Mipmapping

Il **Mipmapping** è una tecnica standard che consiste nel pre-calcolare una catena di versioni a risoluzione decrescente della stessa texture originale. Sebbene questa pratica aumenti l'occupazione di memoria di circa il 33%, essa è fondamentale per l'ottimizzazione delle prestazioni a run-time.

In assenza di mipmapping, la rasterizzazione di oggetti distanti introduce una significativa differenza tra la risoluzione dello schermo e quella della texture. In

tale scenario, un singolo pixel dello schermo corrisponde a un'area estesa della texture originale. Questo costringe il campionatore[10]a selezionare texel che, sebbene mappati su pixel adiacenti a video, risultano memorizzati in locazioni fisiche molto distanti tra loro.

Una simile dispersione nell'accesso alla memoria annulla il principio di località spaziale su cui si basano le cache della GPU, provocando un elevato numero di cache miss. Di conseguenza, viene introdotto un pesante collo di bottiglia nella pipeline grafica dovuto alla latenza di accesso alla VRAM globale, causando frequenti stalli delle unità di calcolo.

L'adozione del mipmapping risolve interamente questa criticità. Durante la fase di campionamento, la GPU seleziona dinamicamente il livello della catena delle mipmap in cui la densità dei texel approssima maggiormente quella dei pixel a schermo. Questa corrispondenza garantisce che richieste provenienti da pixel adiacenti vengano soddisfatte accedendo a dati memorizzati in locazioni contigue all'interno della texture a bassa risoluzione, mantenendo intatto il principio di località spaziale e massimizzando l'efficienza della cache.

1.6.2 Virtual Texturing

Per scenari che richiedono una fedeltà visiva estrema o mondi vasti senza caricamenti, la memoria fisica della GPU diventa un limite invalicabile. La soluzione è l'adozione del **Virtual Texturing**, una tecnica che disaccoppia le coordinate di texture virtuali dall'allocazione fisica in memoria.

Sparse Virtual Texturing (SVT)

Il concetto alla base dello SVT è suddividere le texture molto grandi in tasselli di dimensioni fisse chiamati "pages" o "tiles". Invece di caricare l'intera texture, il sistema determina a runtime quali tile sono visibili dalla telecamera e carica solo quelle necessarie in una cache fisica pre-allocata. Per adattare l'utilizzo del campionatore classico a questo paradigma, viene introdotta una struttura dati intermedia chiamata **Page Table**, che funge da traduttore tra le coordinate UV virtuali e gli indirizzi fisici in VRAM.

1. Il Pixel Shader calcola le coordinate UV virtuali.

[10]Unità della GPU responsabile del prelievo dei dati da una texture. Esso traduce le coordinate (u, v) di un texel in indirizzi di memoria fisica.

2. la Page Table traduce l'indirizzo virtuale nell'indirizzo fisico della tile in VRAM.

3. Se il tile non è presente, viene inviata una richiesta di streaming alla CPU e, nel frattempo, viene utilizzata una tile di qualità inferiore.

L'impatto delle tecniche di Virtual Texturing e Mipmapping sulla VRAM, discusso in questa sede, sarà misurato sperimentalmente nel Test C. L'analisi si focalizzerà sul consumo di memoria video durante l'uso di texture ad alta risoluzione (8K), verificando come il sistema di streaming dei tile influenzi le performance rispetto al mipmapping tradizionale.

Capitolo 2

Paradigmi e Implementazioni a Confronto: L'Architettura del Rendering in Unity, Unreal Engine e Godot

2.1 Panoramica dei motori grafici selezionati

L'analisi comparativa oggetto di questa tesi si concentra su tre piattaforme che rappresentano, per diffusione e approccio ingegneristico, lo standard attuale nello sviluppo di applicazioni 3D in tempo reale: Unity, Unreal Engine e Godot.

Le differenze architetturali tra questi motori consentono di esplorare l'intero spettro delle principali soluzioni ingegneristiche adottate nell'industria: l'approccio modulare e fortemente scriptabile di Unity, l'architettura monolitica orientata all'alta fedeltà grafica di Unreal Engine, e l'approccio leggero e open-source di Godot.

2.1.1 Unity

Sviluppato da Unity Technologies, è storicamente basato su un'architettura a componenti che separa nettamente il motore "nativo" (scritto in C/C++ per le performance) dallo strato di scripting "gestito" (C# per la logica).

L'evoluzione più significativa nelle recenti versioni è stata la transizione dalla Built-in Render Pipeline alla Scriptable Render Pipeline. Questa architettura espone il loop di rendering al codice C#, permettendo agli sviluppatori di confi-

gurare, estendere o riscrivere interamente il flusso di rendering. Ciò permette di ottimizzare l'applicazione in base alla piattaforma di destinazione e alle esigenze specifiche del progetto.

2.1.2 Unreal Engine

Sviluppato da Epic Games, Unreal Engine rappresenta l'apice dell'approccio monolitico. Il motore fornisce una suite completa di strumenti integrati progettati per funzionare in concerto.

L'avvento di Unreal Engine 5 ha ridefinito gli standard dell'industria, introducendo sistemi rivoluzionari come Nanite, un sistema di virtualizzazione della geometria capace di gestire mesh di complessità cinematografica in tempo reale.

Sebbene l'adozione di tecnologie "bleeding edge" comporti normalmente una fase iniziale di instabilità software (riscontrata nelle prime release del ciclo vitale di UE5), il motore ha oggi raggiunto piena maturità.

2.1.3 Godot

Rappresenta il motore più giovane e meno maturo tra i tre, ma è anche quello che più si discosta dagli standard proprietari. Godot rappresenta, infatti, l'esponente principale del software libero nel panorama dei game engine. La versione 4 ha segnato una riscrittura completa del backend grafico, abbandonando l'API grafica OpenGL in favore di Vulkan.

L'architettura di Godot utilizza alcune scelte di design cruciali (che vedremo in dettaglio nel capitolo ad esso dedicato) che permettono la separazione della logica di gioco dai processi intensivi di basso livello. Ciò consente di mantenere un overhead minimo, ma richiede all'utente di gestire manualmente ottimizzazioni che in altri motori potrebbero essere automatiche.

2.1.4 Criteri di selezione delle versioni software

La scelta delle specifiche versioni dei motori grafici oggetto di analisi è stata guidata dall'obiettivo primario di questa tesi: definire lo "stato dell'arte" del rendering in tempo reale.

L'adozione delle release stabili più recenti disponibili al momento della stesura (Gennaio 2026) è necessaria per valutare le tecnologie di ottimizzazione di frontiera adottate dai singoli engine.

Tale decisione si allinea inoltre con le raccomandazioni ufficiali dei vendor, che indicano queste specifiche versioni come lo standard attuale per l'avvio di nuovi cicli produttivi.

Detto ciò, le versioni selezionate per l'analisi sono:

- **Unity:** Versione 6.3 LTS
- **Unreal Engine:** Versione 5.7.2
- **Godot:** Versione 4.6

Si procederà innanzitutto a un'analisi preliminare delle architetture di rendering dei singoli motori e delle relative tecniche di ottimizzazione, per poi sviluppare un confronto pratico fondato su benchmark e casi di studio specifici.

2.2 Unity e la Scriptable Render Pipeline

Fino alla versione 2018, il motore utilizzava esclusivamente la "Built-in Pipeline": un'architettura di rendering monolitica, interamente scritta in C++ per garantire prestazioni ottimali.

Questo approccio operava come una scatola nera: anche se altamente ottimizzato per l'hardware dell'epoca, il ciclo di rendering era rigido e opaco. Il motore C++ doveva eseguire controlli di validazione generici per ogni frame (ad esempio, verificare la presenza di ombre, nebbia o riflessi), imponendo un overhead di calcolo costante anche quando tali funzionalità non erano presenti nella scena.

Con l'evoluzione delle moderne GPU e l'aumento della complessità geometrica, questo modello ha mostrato i suoi limiti, in particolare nel collo di bottiglia della CPU. Nella pipeline integrata, infatti, ogni cambio di materiale richiedeva alla CPU di riconfigurare lo stato della GPU e inviare nuovamente i dati, sprecando un'enorme quantità di risorse sia computazionali che di memoria.

2.2.1 Architettura: il controllo del loop di rendering

L'introduzione della Scriptable Render Pipeline (SRP) risolve questi problemi attraverso una separazione architetturale tra logica di controllo e di esecuzione:

- **Logica di controllo (C#):** La logica di scheduling (decidere cosa disegnare, quando e come) viene esposta al livello di scripting in C#.

- **Logica di esecuzione (C++):** Le operazioni primitive ad alte prestazioni rimangono nel livello nativo per garantire la massima velocità di esecuzione.

Questa architettura ibrida offre due vantaggi fondamentali:

- **Trasparenza:** Esponendo il loop di rendering al livello di scripting, gli sviluppatori possono rimuovere interamente passaggi non necessari dalla sequenza di esecuzione. A differenza della vecchia pipeline che effettuava controlli inutili, la SRP esegue solo le istruzioni esplicitamente programmate, riducendo drasticamente l'overhead della CPU.

- **Ottimizzazione della Memoria e Draw Calls:** La migrazione della logica di controllo al livello di scripting permette di implementare sistemi di batching più sofisticati. É il caso dell'SRP Batcher, componente chiave della nuova architettura. Esso permette di sfruttare la VRAM per diminuire il traffico di dati tra CPU e GPU, riducendo anche il numero di draw call.

La programmabilità del loop di rendering viene risaltata dalla possibilità di scegliere tra entrambi i render path Forward e Deferred.

2.2.2 Gestione delle draw call: SRP Batcher

Per la comunicazione dei dati tra CPU e GPU, la pipeline di rendering di Unity utilizza i Constant Buffer (CBUFFER), blocchi di memoria che contengono i parametri necessari per disegnare un oggetto.

La differenza sostanziale tra la SRP e la pipeline Built-In sta nella frequenza con cui vengono trasferiti i CBUFFER. Nonostante il metodo tradizionale prevedesse il raggruppamento degli oggetti da disegnare in base al materiale, ogni cambio di materiale comportava comunque la necessità di riconfigurare e ritrasmettere i dati alla GPU:

1. La CPU rileva un cambio di materiale.
2. La CPU alloca o mappa un CBUFFER nella GPU.
3. La CPU copia i nuovi dati del materiale nel CBUFFER.
4. La CPU esegue una draw call.

Questa serie di operazioni viene chiamata "SetPass Call" e rappresenta un punto critico di inefficienza. Se una scena contiene 1.000 oggetti con materiali

diversi, la CPU deve riconfigurare e ritrasmettere i dati 1.000 volte per frame, saturando la banda passante e causando stalli nella pipeline.

L'SRP Batcher, invece di considerare i CBUFFER come contenitori temporanei da riempire ad ogni frame, li tratta come risorse residenti a lungo termine nella VRAM.

Il sistema divide i dati in due buffer distinti e ottimizzati:

- **UnityPerMaterial CBUFFER:** Un buffer persistente che contiene tutti i parametri fissi del materiale. Viene caricato in VRAM al primo utilizzo del materiale e non viene più ritrasmesso a meno che le proprietà del materiale non cambino esplicitamente.

- **UnityPerDraw CBUFFER:** Un buffer di grandi dimensioni che contiene i dati specifici delle istanze, come ad esempio le matrici di trasformazione.

Con questa suddivisione, durante il loop di rendering la CPU non deve più ritrasmettere i dati del materiale. Si limita ad inviare la GPU un buffer di tipo UnityPerDraw con i dati dinamici, mentre il buffer UnityPerMaterial rimane costantemente in VRAM.

Inoltre, se due oggetti condividono lo stesso materiale, possono essere disegnati con una singola draw call, poiché la GPU può accedere direttamente al buffer del materiale senza necessità di riconfigurazione.

2.2.3 Gestione della geometria

LOD

Unity gestisce la complessità geometrica attraverso algoritmi di LOD discreto. Nella fase di importazione di una mesh, é possibile delegare all'engine la generazione dei livelli di dettaglio minore.

Non vi sono informazioni esplicite riguardo all'implementazione degli algoritmi di generazione dei LOD, ma la documentazione fa intendere che utilizzino tecniche di decimazione simili a quelle descritte nel capitolo 2.3.2.

Per quanto riguarda la selezione del livello di LOD, Unity utilizza una variante del metodo dell'area proiettata. Non viene utilizzata l'intera area, ma solamente la componente verticale, ovvero l'altezza dell'oggetto proiettata sullo schermo. Questa scelta deriva da diversi fattori, primo tra cui é il modo in cui Unity gestisce il campo visivo.

Di default, infatti, il motore utilizza un FOV verticale fisso, perció vi é una relazione più diretta tra l'altezza proiettata e la distanza: esse scalano linearmente

$$\text{Altezza Proiettata} \propto \frac{1}{\text{Distanza}} \tag{2.1}$$

mentre l'area proiettata scala quadraticamente in relazione alla distanza, rendendo più complesso definire soglie di LOD stabili.

Vi é anche una motivazione puramente prestazionale: proiettare gli 8 vertici di un bounding box per calcolare l'area proiettata è più costoso che proiettare due punti Y_{min} e Y_{max} per poi calcolare l'altezza proiettata come

$$\text{Altezza Proiettata} = Y_{max} - Y_{min} \tag{2.2}$$

Occlusion Culling

Per anni, Unity si è affidato a soluzioni middleware di terze parti per l'occlusion culling, integrando la tecnologia Umbra. Il processo si divide in due stadi distinti: la generazione offline di una struttura dati spaziale e l'esecuzione dell'algoritmo di visibilità a run-time.

Analizziamo la creazione della struttura dati:

1. La scena viene discretizzata attraverso un processo di voxelizzazione. Lo spazio tridimensionale viene suddiviso in una griglia di cubi chiamati voxel[1], la cui dimensione è un parametro configurabile che determina il compromesso tra precisione del culling e consumo di memoria.

2. I voxel vengono classificati in base al contenuto:

   (a) Celle - View Cells: I voxel che rappresentano lo spazio vuoto e navigabile vengono raggruppati in volumi convessi di dimensioni maggiori, denominati celle. Le superfici di contatto tra celle adiacenti vengono identificate come portali.

   (b) Occlusori - Solid Voxels: I voxel che intersecano la geometria statica vengono suddivisi nuovamente in voxel di dimensioni inferiori. Tra questi, quelli che intersecano la geometria statica vengono classificati come occlusori, mentre quelli che intersecano lo spazio vuoto vengono aggregati alle celle.

[1] Volumetric pixel.

Il secondo stadio avviene a run-time sulla CPU. Il sistema Umbra utilizza la struttura generata per risolvere la visibilità per ogni frame, sfruttando una tecnica di rasterizzazione a livello software (quindi su CPU) su un buffer di profondità a bassa risoluzione.

1. Il sistema determina in quale Cella si trova la telecamera in base alla sua posizione nello spazio.

2. Il motore rasterizza i voxel degli occlusori statici visibili dalla cella corrente all'interno di un buffer di profondità software a bassa risoluzione.

3. Il sistema proietta i portali della cella corrente sullo schermo e li confronta con il buffer di profondità appena generato.

   (a) Se un portale si trova interamente dietro i valori di profondità scritti dagli occlusori, esso è considerato occluso: il traversamento si ferma e la cella oltre il portale viene ignorata.

   (b) Se il portale è visibile (anche parzialmente), il processo continua ricorsivamente nella cella adiacente, controllando a sua volta i portali e gli occlusori di quella cella, fino a quando non si raggiungono tutte le celle visibili o si raggiunge un limite di profondità.

Questo approccio tiene conto solamente degli occlusori statici, poiché la struttura dati è generata offline e non può essere aggiornata dinamicamente. Tuttavia, dato che stiamo cercando un PVS (capitolo 2.4), possiamo semplicemente trattare gli oggetti dinamici come occludibili, ma non come occlusori. Ciò significa che, se un oggetto dinamico si trova dietro un occlusore statico, esso sarà correttamente considerato invisibile, ma non potrà occludere altri oggetti.

Nonostante molto efficace e ampiamente utilizzato, questo sistema presenta alcune limitazioni intrinseche. La generazione della struttura dati può essere un processo lungo e richiede una quantità significativa di memoria, soprattutto per scene complesse con molti dettagli. Inoltre, l'algoritmo di rasterizzazione software su CPU può diventare un collo di bottiglia in scenari con molteplici occlusori o portali, limitando la scalabilità del sistema.

Per questo, e anche grazie all'avvento dei compute shader[2], dalla versione 6 di Unity è stato introdotto un nuovo sistema di occlusion culling basato su GPU, che

[2]Shader di calcolo, programmi eseguiti direttamente sulla GPU per operazioni generiche di elaborazione dati, non limitati alla grafica.

sfrutta la potenza di calcolo parallelo delle moderne schede grafiche per eseguire il culling in modo più efficiente e scalabile.

Questa implementazione si configura come una variante specializzata dell'algoritmo Hierarchical Z-Buffer, analizzato nel Capitolo 2.4.3. Sebbene l'utilizzo della Z-Pyramid rimanga fedele al modello teorico, la differenza risiede nella strategia di accesso ai dati spaziali. Il sistema non utilizza octree, che su architetture GPU risultano inefficienti a causa dell'accesso non contiguo alla memoria, ma vengono invece favoriti approcci lineari e parallelizzati. Il culling viene eseguito lanciando un thread per ogni istanza e confrontando direttamente la Bounding Sphere dell'oggetto con il livello di mipmap corrispondente della piramide di profondità.

2.2.4 Rendering Graph e Gestione Memoria

Prima dell'introduzione della SRP, lo sviluppatore era responsabile dell'allocazione e del rilascio manuale delle risorse temporanee come texture e buffer, e dell'emissione immediata dei comandi di disegno. Questo approccio, pur essendo flessibile, rendeva complessa l'ottimizzazione globale del frame, poiché il motore non aveva visibilità sulle dipendenze future.

Il **Render Graph System** è un livello di astrazione costruito a livello di logica di controllo. Invece di eseguire immediatamente i comandi, il sistema "registra" le intenzioni di rendering. Ogni frame viene rappresentato come un grafo aciclico orientato:

- I nodi rappresentano i passaggi di rendering, come il calcolo delle ombre o il post-processing.
- Gli archi rappresentano le dipendenze dei dati, ovvero quale risorsa viene prodotta da un nodo e consumata da un altro.

Il funzionamento si articola in due fasi distinte per ogni frame:

1. Registrazione: Tutti i passaggi dichiarano di quali risorse hanno bisogno in input e quali risorse forniscono in output.
2. Compilazione ed Esecuzione: Il Render Graph analizza l'intero grafo e calcola la vita utile delle risorse. Invia poi i comandi alla GPU nell'ordine ottimale.

La conoscenza del frame prima della sua esecuzione permette al Render Graph di applicare ottimizzazioni logiche impossibili nel modello imperativo:

- **Pass Culling**: Il sistema traccia le dipendenze a ritroso partendo dall'immagine finale. Se un nodo produce una risorsa che non contribuisce direttamente o indirettamente al risultato finale, quel nodo viene interamente rimosso dal grafo di esecuzione, prevenendo l'esecuzione di passaggi inutili.

- **Sincronizzazione Automatica**: Le API grafiche richiedono una gestione esplicita delle barriere di memoria per evitare *race condition* tra passaggi che scrivono e leggono la stessa risorsa. Il Render Graph, conoscendo esattamente i momenti in cui ogni risorsa viene letta o scritta, inserisce automaticamente le barriere di sincronizzazione più efficienti.

Ma il vantaggio più significativo del Render Graph risiede nella gestione della memoria video, specificamente per le "risorse transitorie", ovvero texture temporanee che esistono solo per la durata di un frame.

Poiché il sistema conosce l'intervallo temporale esatto di utilizzo di ogni risorsa, può determinare quali risorse non sono mai presenti in memoria allo stesso tempo. Di conseguenza, due texture di grandi dimensioni che non sono mai memorizzate contemporaneamente nello stesso frame possono condividere lo stesso indirizzo fisico nella memoria video.

Ad esempio, una texture `A` usata all'inizio del frame e un'altra texture `B` usata alla fine possono risiedere nello stesso spazio di memoria, in momenti diversi.

Nel modello classico, queste texture richiederebbero `Size(A) + Size(B)` bytes di VRAM. Con il Render Graph, richiedono solo `Max(Size(A), Size(B))`.

2.3 Unreal Engine 5: Virtualizzazione della Geometria

Mentre Unity adotta un approccio di ottimizzazione esplicita della pipeline tradizionale, cercando di diminuire soprattutto l'overhead CPU, Unreal Engine 5 introduce un cambio di paradigma radicale spostandosi verso una pipeline **GPU Driven**. Questo sistema mira ad escludere la CPU dal loop di rendering, evitando completamente rischi di colli di bottiglia nella comunicazione CPU-GPU.

Questo obiettivo viene raggiunto principalmente tramite la virtualizzazione della geometria, realizzata attraverso Nanite, analogamente a quanto avviene nella virtualizzazione delle texture: la complessità della risorsa viene disaccoppiata dai vincoli fisici dell'hardware.

La migrazione a una pipeline GPU driven comporta l'implementazione su GPU di tutte le ottimizzazioni che venivano gestite dalla CPU, primi tra cui il culling e il LOD.

2.3.1 Nanite: Struttura Dati e LOD Continuo

La gestione del LOD in Nanite supera il concetto classico di LOD discreto. Invece di scambiare intere mesh, Nanite opera su **cluster**, una struttura dati granulare composta da 128 triangoli.

Durante la fase di importazione, la mesh originale viene analizzata e suddivisa in cluster. A differenza dalla struttura dati teorica utilizzata da Hoppe per le *Progressive Mesh*, questi cluster vengono organizzati in una gerarchia sotto forma di grafo aciclico orientato.

Il DAG permette di fondere cluster adiacenti in cluster genitori semplificati, diminuendo il numero di cluster. A run-time, il sistema non seleziona un singolo LOD per l'intero oggetto, ma determina un taglio attraverso il grafo per ogni istanza.

Selezione del LOD View-Dependent:

La metrica di selezione risulta ibrida, utilizzando inizialmente l'errore geometrico con QEM per decidere quali lati collassare per primi, per poi proiettare tale errore in screen space e valutarlo in pixel.

- Se l'errore proiettato è inferiore a una soglia impercettibile (es. 1 pixel), il cluster viene disegnato.
- Se l'errore è maggiore, il sistema discende nel grafo caricando i cluster figli più dettagliati. Questo realizza teoricamente un **LOD Continuo**: la densità dei triangoli si adatta alla risoluzione dello schermo, eliminando in aggiunta gli artefatti di popping.

Culling Gerarchico e Two-Pass HZB Per gestire scene con complessità "infinita", Nanite implementa una pipeline di culling aggressiva lato GPU.

1. **Instance Culling:** Un primo passaggio rapido scarta istanze di oggetti fuori dal Frustum.

2. **Occlusion Culling (HZB) e Backface Culling:** Nanite implementa una pipeline basata su Hierarchical Z-Buffer a due passaggi che integra implicitamente il backface culling, scartando automaticamente i cluster il cui orientamento risulta opposto alla telecamera

   (a) *Main Pass:* I cluster vengono testati contro l'HZB del frame precedente. Se erano visibili, vengono disegnati subito.

   (b) *Post Pass:* I cluster che risultavano occlusi nel frame precedente vengono ritestati contro l'HZB corrente, aggiornato dal Main Pass. Se ora risultano visibili (es. un occlusore si è spostato), vengono disegnati in questo secondo passaggio.

Rasterizzazione Ibrida: Software vs Hardware L'ennesima innovazione introdotta da UE5 è la gestione dei triangoli sub-pixel. Nanite risolve il quad overdraw con un rasterizzatore ibrido:

- **Hardware Rasterizer:** Se i triangoli del cluster sono grandi (coprono più di 32 pixel circa), vengono inviati alla pipeline hardware standard.
- **Software Rasterizer:** Se i triangoli sono piccoli, vengono rasterizzati via software da un Compute Shader altamente ottimizzato. Questo rasterizzatore risulta fino a 3 volte più veloce dell'hardware standard per geometrie dense.

Rendering Path: Il Visibility Buffer L'integrazione di Nanite impone un cambiamento nel Rendering Path. Scrivere un G-Buffer completo per miliardi di micropoligoni richiederebbe una banda passante di memoria insostenibile.

UE5 adotta l'approccio del **Visibility Buffer**:

1. **Depth Pre-Pass:** Durante la rasterizzazione, la GPU non calcola i materiali. Scrive in un buffer solo l'ID dell'Istanza e del Triangolo visibile per quel pixel.
2. **Material Pass:** Solo in una fase successiva, per ogni pixel dello schermo, il sistema decodifica l'ID, recupera le proprietà del materiale associato al triangolo, e infine esegue il Pixel Shader. Questo disaccoppia il costo geometrico dal costo di shading: l'overdraw é inesistente, poiché il materiale pesante viene calcolato solo per il pixel finale visibile.

2.3.2 Virtual Texturing con RVT

A differenza del SVT (capitolo 2.6.2), il run-time Virtual Texturing (RVT) non preleva dati dal disco, ma genera le pagine a run-time direttamente sulla GPU. Questa tecnica è fondamentale per l'ottimizzazione di materiali complessi composti da piú layer di dettaglio sovrapposti tra di loro: invece di ricalcolare ogni frame il blending tra i vari layer (es. erba, fango, roccia, neve), il motore renderizza il risultato di questi calcoli una sola volta e lo memorizza all'interno delle pagine dell'RVT.

Durante il passaggio di rendering finale, la geometria del terreno non esegue più il pesante shader originale, ma si limita a campionare la texture RVT risultante.

Questo riduce drasticamente il costo di pixel shading, permettendo al contempo un blending ottimo tra la geometria del terreno e oggetti statici.

2.4 Godot Engine: Architettura Lightweight

L'approccio di Godot si distingue per un design modulare basato su strutture *Server* e sull'utilizzo nativo delle API grafiche Vulkan.

2.4.1 Il design a Server

La scelta architetturale più distintiva risiede nella separazione netta tra la logica di alto livello e i sottosistemi di basso livello, gestita attraverso un'architettura a **Server**.

In Godot, ogni sistema critico è incapsulato in un server indipendente. Le strutture che lo sviluppatore utilizza nell'editor 3D, fungono da semplici wrapper di alto livello che inviano comandi al server rispettivo.

Il vero punto di forza di questa architettura non è tanto la separazione in sé, quanto la possibilità per lo sviluppatore di bypassare completamente le strutture di alto livello e interagire direttamente con i server, un'operazione di ottimizzazione profonda impossibile in Unity e Unreal. Inoltre, poiché i server sono logicamente isolati, ognuno di essi può elaborare i propri comandi su un thread separato rispetto alla logica di gioco, eliminando alla radice i rischi di race conditions.

2.4.2 Il fattore Open Source come strumento di Ottimizzazione

Nel caso specifico di Godot, la natura libera del codice sorgente si trasforma da caratteristica etica a potente strumento di ottimizzazione tecnica.

A differenza dei motori proprietari che forniscono binari pre-compilati monolitici, Godot permette di ricompilare il motore escludendo interi moduli non utilizzati. Questa operazione riduce drasticamente l'utilizzo di memoria e la dimensione dell'eseguibile, minimizzando di conseguenza i cache miss.

Inoltre, tramite il sistema *GDExtension*, il motore permette di scrivere logica di gioco critica direttamente in C++ come se fosse parte integrante del motore. Questo elimina l'overhead di comunicazione tra linguaggi tipico delle soluzioni di scripting tradizionali, come il passaggio C++/C# di Unity.

2.4.3 Rendering Path e Culling

Godot adotta lo standard **Clustered Forward Rendering**, garantendo il supporto nativo a MSAA e trasparenze con un consumo di banda passante ridotto rispetto alle soluzioni Deferred.

A supporto di questa pipeline, Godot implementa nativamente il sistema **Rooms and Portals**: una variante manuale dell'approccio utilizzato nel sistema Umbra di Unity. In questo caso, la generazione della struttura dati è affidata allo sviluppatore, che deve suddividere la scena in stanze e definire i portali di collegamento.

La fase di culling a runtime opera in modo simile a Umbra, utilizzando una rasterizzazione software su CPU per risolvere la visibilità dei portali. Per gli ambienti aperti, dove i portali non sono applicabili, il motore integra questa tecnica generando un depth buffer a bassa risoluzione sulla CPU per ogni frame, mantenendo la GPU libera per il carico di shading.

Capitolo 3

Protocollo Sperimentale e Framework di Valutazione

Questo capitolo definisce la metodologia sperimentale adottata per la valutazione comparativa delle performance.

L'obiettivo principale non è stabilire un parametro generico di velocità dei motori, ma di isolare e quantificare l'impatto delle diverse tecniche di ottimizzazione in un ambiente controllato.

Nelle sezioni seguenti verranno dettagliati gli strumenti di profilazione utilizzati per monitorare le metriche critiche, la progettazione dello scenario di test e le specifiche configurazioni per i tre test: gestione della geometria (Test A), efficienza della pipeline (Test B) e saturazione della memoria (Test C).

Hardware di riferimento Poiché la maggior parte delle tecniche di ottimizzazione implementate dai tre engine oggetto di studio hanno l'obiettivo di minimizzare l'overhead CPU, è ragionevole aspettarsi che le differenze di performance risultino più evidenti su configurazioni hardware caratterizzate da CPU meno performanti, dove l'ottimizzazione della comunicazione tra CPU e GPU diventa particolarmente critica. Per questo motivo, i test sono stati eseguiti su una macchina appositamente configurata per evidenziare potenziali colli di bottiglia di questo tipo.

La configurazione hardware è la seguente:

- **CPU**: AMD Ryzen 5 2600X (6 Core, 12 Thread @ 3.6GHz base)
- **GPU**: NVIDIA GeForce RTX 4080 Super (16GB GDDR6X)
- **RAM**: 16GB DDR4 @ 3200MHz

- **Storage**: SSD NVMe M.2 da 1TB
- **Sistema Operativo**: Windows 11 Pro 64-bit (build 26200.7840)
- **Driver GPU**: NVIDIA Game Ready Driver (versione 566.36)

3.1 Strumenti di profiling e metriche di riferimento

Per garantire un'analisi dettagliata delle performance, la raccolta dei dati è stata affidata a un sistema "ibrido" che combina strumenti di profilazione hardware e script di monitoraggio interni, sviluppati ad hoc per ciascun engine.

L'acquisizione hardware è stata effettuata utilizzando NVIDIA FrameView, in modo da garantire la massima compatibilità e precisione. Esso consente di monitorare le metriche del sistema durante l'esecuzione di build compilate. Inoltre, per avere una visione più granulare delle performance e accedere ai dati del singolo *game loop*, sono stati realizzati degli script personalizzati che si interfacciano direttamente con le API di profilazione native di ciascun motore.

Per interpretare correttamente le metriche relative ai tempi di calcolo, è essenziale analizzare l'architettura di gestione dei comandi di ciascun motore. In Unity, le misurazioni si suddividono in Main Thread, responsabile dell'aggiornamento della logica di gioco, della fisica e degli script, e in Render Thread, incaricato di preparare le Draw Calls e interfacciarsi con le API grafiche. Unreal Engine adotta un'architettura più stratificata: il Game Thread gestisce la logica, mentre il Render Thread si occupa del culling e dell'ordinamento geometrico; infine, il thread RHI (Render Hardware Interface) traduce i comandi per le API grafiche prima dell'esecuzione vera e propria sulla GPU. In Godot, l'architettura isola le operazioni sul Main Thread delegando l'intera pipeline di rendering al Rendering Server, che opera su un thread separato dedicato esclusivamente all'invio dei comandi grafici.

Le metriche raccolte da ogni script si differenziano leggermente in base alle API esposte:

- **Unity:** FPS, Frame Time globale, Main Thread Time, Render Thread Time, GPU Time, Draw Calls, Vertices e VRAM allocata.
- **Unreal:** FPS, Frame Time globale, Draw Calls e Vertices.
- **Godot:** FPS, Frame Time globale, CPU Time, Draw Calls, Vertices e VRAM allocata.

Tabella 3.1: Metrica di profilazione interna ed esterna raccolte per ciascun motore grafico.

Metrica analizzata	**Unity**	**Unreal Engine**	**Godot**
FPS	Sì	Sì	Sì
Frame Time (CPU/GPU)	Frame Time globale / Main / Render / GPU	Solo Frame Time globale	Frame Time globale / CPU Time
Geometria (Draw Calls e Vertici)	Sì	Sì	Sì
VRAM	Sì	Solo tramite `Stat RHI`	Sì

3.1.1 Sfide e adattamenti nella profilazione

Durante la fase di test, sono emerse alcune limitazioni strumentali che hanno richiesto metodi alternativi per la raccolta dei dati. Nello specifico, la misurazione dell'occupazione della VRAM si è rivelata inaccessibile da parte di NVIDIA FrameView per quanto riguarda tutti e tre gli engine. Per ovviare a questa limitazione, in Unity e Godot è stato possibile accedere a questa metrica tramite le rispettive API di profilazione, ma non è stato questo il caso per Unreal Engine, che non espone un'API equivalente. Di conseguenza, per Unreal Engine le misurazioni di VRAM sono state effettuate a runtime utilizzando comandi nella console di debug integrata, nello specifico tramite il comando `Stat RHI`. I dati relativi al footprint di memoria e all'allocazione delle texture in Unreal sono stati quindi documentati tramite screenshot dei log a schermo durante l'esecuzione degli stress test.

3.2 Progettazione delle scene di test

Per cercare di fornire rilevazioni il meno rumorose possibile, l'ambiente di default di ogni engine è stato ridotto all'osso. Non è quindi presente nessun tipo di elemento che possa introdurre calcoli aggiuntivi o variabili non controllate, come ad esempio luci dinamiche, ombre o cieli. Inoltre ogni oggetto è contrassegnato come statico, in modo da escludere qualsiasi tipo di calcolo relativo alla dinamica degli oggetti.

La scena di riferimento è composta da una griglia di 100x100 entità ad alta densità poligonale, per un totale di 10.000 oggetti. Ogni entità è costituita da una sfera generata nel software di modellazione 3D Blender, composta da circa 12.000 vertici.

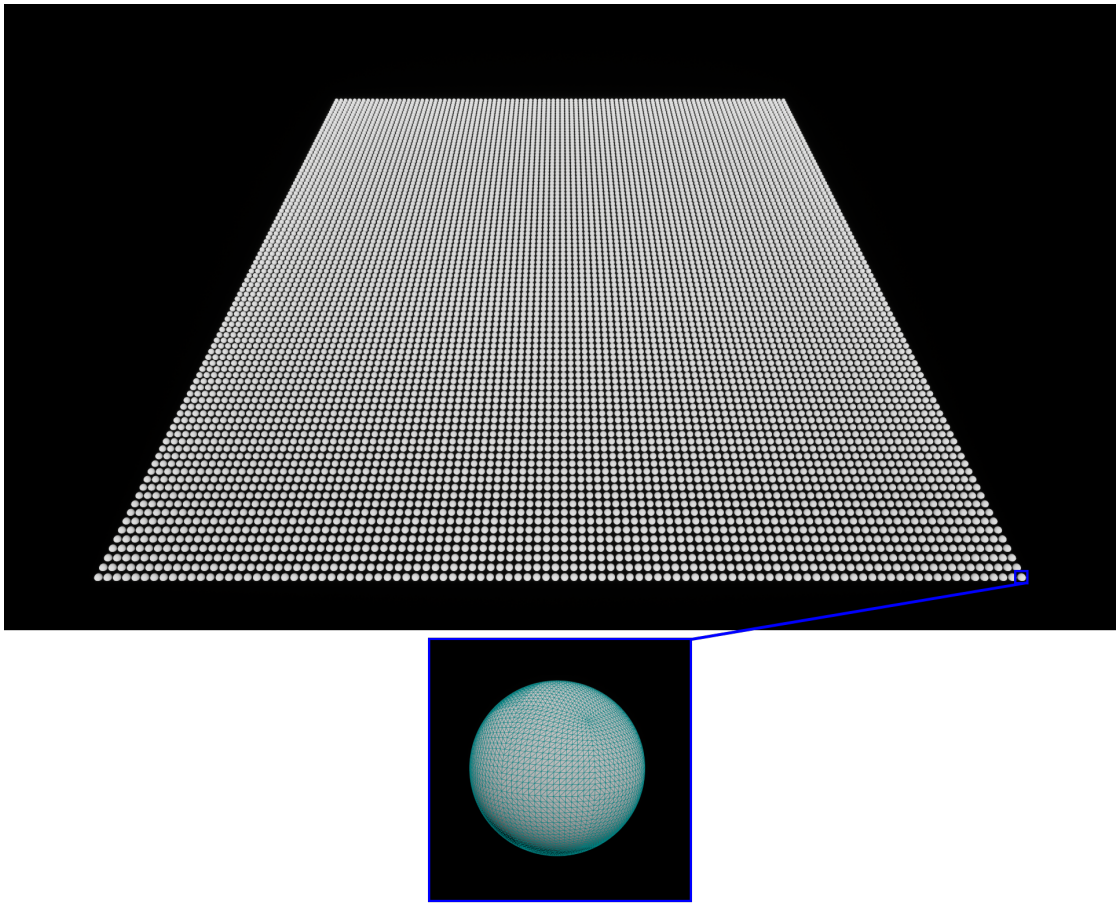

Figura 3.1: Composizione della scena di test, con dettaglio sul wireframe delle sfere.

La scelta di un numero così elevato di vertici per il singolo oggetto, in combinazione con il numero totale di entità, è stata fatta con l'obiettivo di saturare rapidamente ogni stadio della pipeline grafica. In questo modo, è possibile evidenziare le differenze di performance tra i tre engine in scenari ad alta complessità geometrica, che rappresentano un caso d'uso comune nei giochi moderni.

Questa griglia di entità andrà a costituire la base per tutti i test, con variazioni specifiche per ciascuna categoria di test che saranno descritte nelle sezioni successive.

3.3 Configurazione dei test

Ogni test cerca di spostare il collo di bottiglia su un aspetto specifico della pipeline grafica, in modo da isolare l'impatto delle tecniche di ottimizzazione implementate dai tre engine. I test sono stati suddivisi in tre macro-categorie, ciascuna con un setup specifico:

3.3.1 Test A: Geometria, Draw Calls e Culling

Il primo blocco di test mira a valutare il peso architetturale dell'invio dei comandi di disegno alla GPU e delle tecniche di riduzione del carico geometrico.

Test A1 (Saturazione comunicazione CPU-GPU)

Alla griglia di sfere vengono applicate progressivamente le tecniche di ottimizzazione native di ciascun engine per misurare l'incremento di performance rispetto a una Baseline renderizzata in modo brute force, senza nessun tipo di ottimizzazione. Le configurazioni testate includono l'Instancing, i sistemi di LOD classici e le loro combinazioni, includendo tecnologie proprietarie come l'SRP Batcher per Unity e la virtualizzazione geometrica Nanite per Unreal Engine 5.

Test A2 (Tecniche di Culling)

Mantenendo la stessa griglia, la telecamera viene spostata in modo da inquadrare solo una porzione di essa, innescando il Frustum Culling. Successivamente, è stato posizionato un volume occludente (un cubo opaco) davanti alla telecamera in modo da bloccare la visuale su un'ulteriore porzione di sfere, permettendo di misurare l'efficienza degli algoritmi di Occlusion Culling implementati sui diversi motori (soluzioni basate su CPU, GPU o rasterizzazione dinamica).

3.3.2 Test B: Pipeline di Illuminazione e Shading (Forward vs Deferred)

Questo test sposta il carico computazionale sul Fragment Shader e sulla larghezza di banda, analizzando l'efficienza delle diverse pipeline di rendering nella gestione di sorgenti luminose multiple. Utilizzando l'instancing per alleggerire il carico sulla CPU, è stato programmato lo spawn dinamico di migliaia di Point Lights con parametri randomici sovrapposte alla griglia di sfere, mantenendo la

Tabella 3.2: Configurazioni testate per la gestione della geometria e del culling.

Configurazione	**Unity**	**Unreal Engine**	**Godot**
Baseline	Brute force	Brute force	Brute force
Instancing	GPU Instancing	Instanced Static Mesh	Multi-MeshInstance3D
LOD	LOD Group	LOD Classico	LOD Automatico
Instancing + LOD	Instancing + LOD	Hierarchical ISM	MultiMesh + LOD
Tecniche Specifiche	SRP Batcher	Nanite	Nessuna tecnica specifica

generazione delle ombre disattivata. Il numero di luci testate varia a seconda delle limitazioni imposte dall'architettura di ciascun motore:

- **Unity:** Valutato sia in Forward che in Deferred Rendering con scaglioni da 100, 500 e 1000 luci, limite massimo dettato dal motore per una singola telecamera.
- **Unreal Engine 5:** Valutato sia in Forward che in Deferred Rendering con scaglioni da 100, 500, 1000, 1500, 2000, 3000 e 4000 luci, soglia oltre la quale le prestazioni risultavano eccessivamente compromesse.
- **Godot:** Valutato esclusivamente tramite la pipeline Forward, essendo l'architettura Deferred non supportata nativamente. I test si sono spinti fino a 8000 luci.

3.3.3 Test C: Gestione Memoria e Virtual Texturing

L'ultimo test è stato pensato per saturare l'allocazione hardware della VRAM e valutare le strategie di streaming degli asset. La griglia geometrica è stata ridotta a dimensioni di 51x51 elementi. Sono stati creati 5 materiali PBR[1], ciascuno composto da 4 mappe texture (Color, Normal, Roughness, Ambient Occlusion) con risoluzione a 8K. Questi materiali sono stati assegnati in modo casuale alle sfere presenti in scena.

[1]**Physically Based Rendering**: un approccio che mira a simulare il comportamento fisico della luce e dei materiali per ottenere risultati più realistici. I materiali PBR utilizzano mappe di texture specifiche per definire le proprietà visive degli oggetti, come il colore, la rugosità e l'occlusione ambientale.

Tabella 3.3: Configurazioni testate per la gestione delle luci e della pipeline di rendering.

Motore Grafico	**Pipeline Testate**	**Range di Luci Testato**	**Limitazioni note**
Unity	Forward, Deferred	100, 500, 1000	Hard-cap per telecamera
Unreal Engine	Forward, Deferred	100, 500, 1000, 1500, 2000, 3000, 4000	Test interrotto a 4000 per degrado critico
Godot	Forward	100, 500, 1000, 2000, 3000, 4000, 6000, 8000	Architettura Deferred non supportata

Le configurazioni analizzate includono un caricamento nativo (Baseline), l'utilizzo di Mipmaps tradizionali e l'attivazione dei sistemi di Virtual Texturing in Unity e Unreal Engine. Quest'ultima configurazione non è stata applicata a Godot in quanto la tecnologia non è attualmente supportata dall'engine.

Capitolo 4

Analisi dei risultati sperimentali

In questo capitolo vengono esaminati e interpretati i dati raccolti durante le sessioni di benchmarking. Al fine di garantire una valutazione lineare, la metrica principale adottata per la misurazione delle prestazioni è il *Frame Time* piuttosto che i *Frame Per Second* o FPS. Questo perchè gli FPS presentano un andamento inversamente proporzionale e non lineare, tendendo a sovrastimare le ottimizzazioni in scenari leggeri e a sottostimare il reale costo computazionale in scenari complessi[1]. L'utilizzo del tempo di rendering per singolo fotogramma permette, invece, di calcolare con esattezza il costo di ciascuna operazione e di valutare se il motore riesca a mantenere l'esecuzione al di sotto della soglia critica di 16.6 ms, corrispondente al budget ideale di 60 fotogrammi al secondo.

4.1 Analisi quantitativa

4.1.1 Test A1

I risultati ottenuti dalla configurazione baseline mostrano un livellamento prestazionale tra i tre motori analizzati. Unity, Unreal Engine e Godot registrano frame time medi molto vicini tra loro, rispettivamente pari a 14.05 ms, 13.62 ms e 13.37 ms. L'analisi della telemetria mostra che, senza nessun tipo di ottimizzazione, tutti gli engine sono costretti a emettere oltre 10.000 draw call, inviando alla GPU un volume geometrico che supera i 117 milioni di vertici per ogni fotogramma. Questa saturazione del bus di comunicazione rappresenta un collo di

[1]Ad esempio, passare da 1000 a 500 FPS rappresenta un crollo di 500 fotogrammi, ma il tempo di rendering per un singolo frame aumenta di appena 1 millisecondo (da 1 ms a 2 ms). Al contrario, passare da 60 a 30 FPS è un calo di "soli" 30 fotogrammi, ma comporta un incremento del tempo di rendering di ben 16.6 millisecondi (da 16.6 ms a 33.3 ms).

bottiglia comune che penalizza equamente i tre motori, indipendentemente dalla loro architettura software.

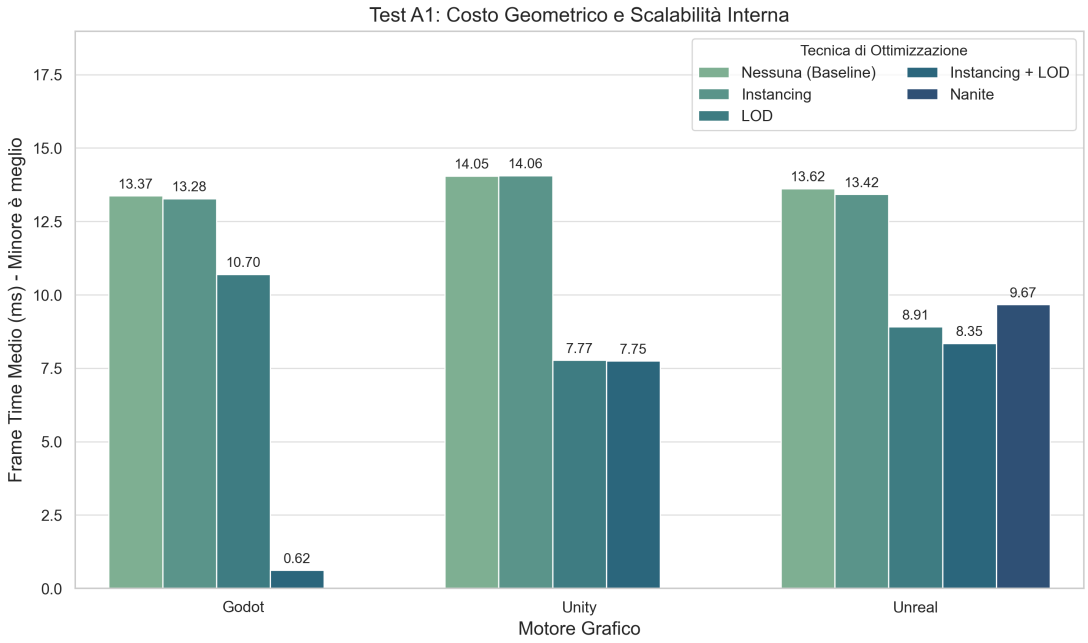

Figura 4.1: Analisi frame time Test A1.

L'attivazione dell'instancing rivela la prima vera differenza architetturale tra gli engine. L'implementazione *MultiMesh* di Godot dimostra un'eccellente efficienza nell'impacchettamento dei comandi: le draw call passano da 10.000 a 1. Nonostante ciò, il tempo di rendering varia di poco, assestandosi a 13.28 ms. L'instancing ha eliminato l'overhead sulla CPU, ma l'enorme quantità di geometria (circa 122 milioni di vertici elaborati) ha semplicemente spostato il collo di bottiglia sul Vertex Shader della GPU, la quale registra un tasso di utilizzo vicino al 99%. Unreal Engine dimostra un comportamento simile: riduce le chiamate a 37 mantenendo un frame time stabile a 13.42 ms. Al contrario, l'implementazione di GPU Instancing di Unity non è riuscita ad aggregare in maniera efficace la geometria in questo specifico stress test, mantenendo le draw call oltre le 10.500 unità (14.06 ms).

La risoluzione concreta del collo di bottiglia avviene tramite la combinazione di instancing e LOD. In questa configurazione, Godot raggiunge il massimo delle sue potenzialità come motore "lightweight": mantenendo un'unica Draw Call e riducendo i vertici ad appena 96 per frame (probabilmente scegliendo il livello LOD più basso, poi utilizzandolo per tutte le 10.000 istanze), il motore svuota completamente la pipeline grafica, registrando un frame time medio di soli 0.62 ms. In Unity, l'utilizzo del LOD abbassa il tempo di calcolo a 7.75 ms, ma mostra

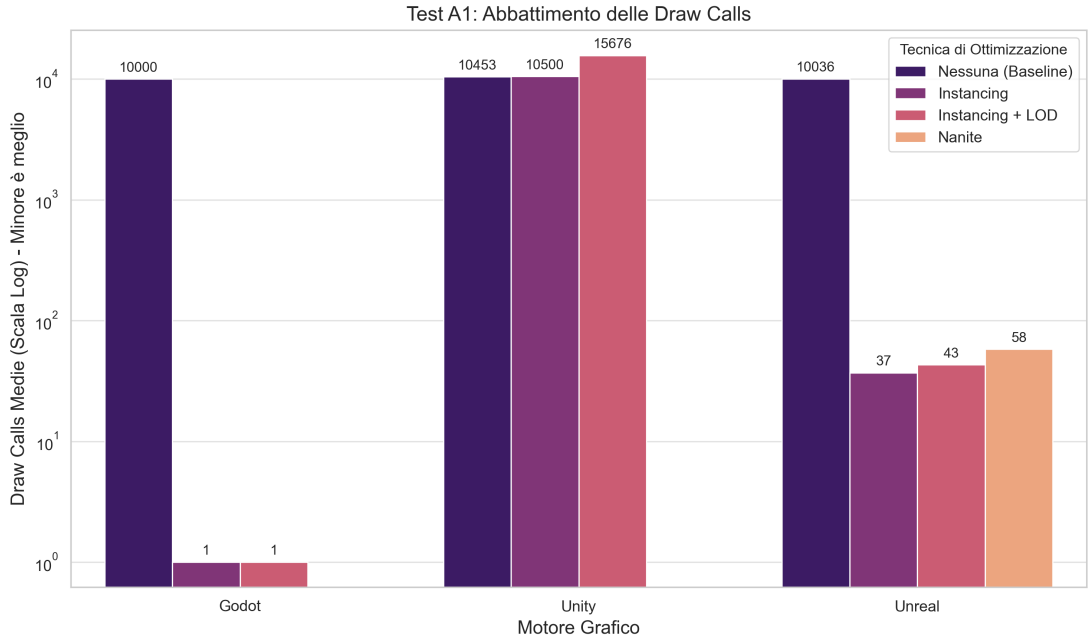

Figura 4.2: Analisi draw call Test A1.

un importante overhead strutturale: la gestione dei gruppi LOD rompe il batching, costringendo il motore a determinare quale livello LOD utilizzare per ogni istanza, e a emettere draw call separate. Di conseguenza, le draw call aumentano oltre 15.676.

L'approccio Nanite di Unreal Engine appiattisce il costo di rendering a 5.99 ms (appena 58 draw call e 93 vertici elaborati). Pur non raggiungendo i tempi di Godot in uno scenario di test "nudo", Nanite si conferma una soluzione di altissimo livello per garantire scalabilità costante. Invece, l'attivazione dell'SRP Batcher in Unity riporta valori quasi identici alla baseline (13.66 ms, 10.242 draw call). Questo risultato è coerente con la teoria: l'SRP Batcher è pensato per ottimizzare i cambi di stato della GPU tra materiali differenti, non per abbattere la quantità lorda di vertici inviati per istanze aventi la stessa mesh e materiale.

4.1.2 Test A2

Il Test A2 sposta l'attenzione sull'efficienza degli algoritmi di culling per la geometria occlusa o fuori campo visivo, un'operazione importante per evitare di renderizzare pixel invisibili all'osservatore.

I dati evidenziano differenze sostanziali nel bilanciamento del carico tra CPU e GPU. Partendo da una baseline di 7.73 ms, Godot dimostra un'efficienza altissima nella sua implementazione di culling: l'attivazione del Dynamic Occlusion Culling

(lato CPU tramite rasterizzazione software) riduce i vertici da 70 milioni a circa 12.7 milioni, facendo scendere le draw call da 5.752 a 1.035. Questo si traduce in un frame time dimezzato di 2.35 ms e in un'importante riduzione dell'utilizzo della GPU (dal 98.67% al 60.98%).

In Unity (baseline 6.92 ms), la differenza tra approccio CPU e GPU risulta particolarmente evidente. L'Occlusion Culling CPU tramite Umbra dimostra (come previsto) performance di altissimo livello grazie alla sua natura offline, effettuando tutti i calcoli pesanti di visibilità prima dell'esecuzione. Esso riduce i vertici a 12 milioni e le draw call a 1.116, portando il frame time a 3.57 ms. L'approccio GPU-driven testato su Unity si rivela controproducente in questo specifico scenario: il frame time rimane fermo a 7.09 ms, e le draw call subiscono un'aumento a 8.299. Questo comportamento è attribuibile al ritardo causato dalla comunicazione asincrona tra CPU e GPU per le occlusion queries.

Infine, Unreal Engine restituisce risultati più stabili. Con una baseline di 9.14 ms, l'occlusion culling GPU con HZB riduce il numero di vertici a 18 milioni e le draw call a 1.277. Tuttavia, il frame time registra solo un lieve miglioramento, stabilizzandosi a 8.22 ms. Ciò evidenzia come, in motori grafici con pipeline fortemente strutturate, l'overhead intrinseco di sistema tenda ad appiattire i guadagni computazionali su scenari a bassa complessità.

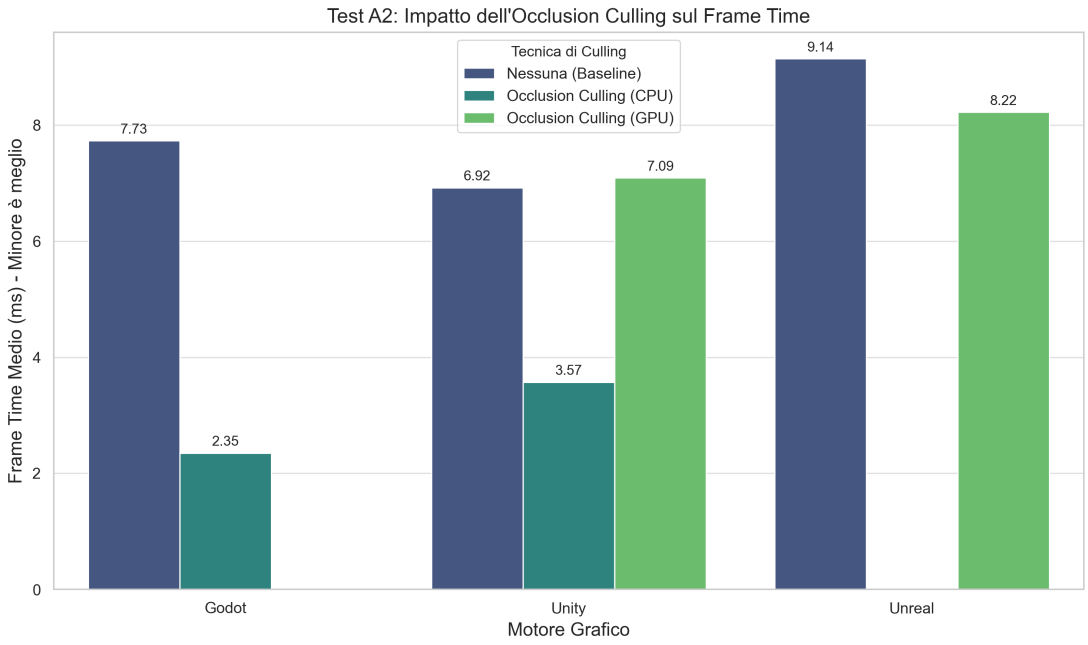

Figura 4.3: Risultati prestazionali del Test A2.

4.1.3 Test B

Con questo test spostiamo il carico computazionale dalla rasterizzazione geometrica al calcolo per singolo frammento e all'occupazione della banda di memoria. Lo scopo è analizzare il comportamento delle pipeline Forward e Deferred all'aumentare delle sorgenti luminose.

L'analisi dei dati mette ulteriormente in risalto la differenza tra approcci lightweight e architetture heavyweight. Godot, operando in modalità Forward, dimostra una forza computazionale bruta eccezionale. Con 100 Point Lights, il frame time si stabilizza a 13.32 ms. Incrementando progressivamente le luci fino al limite di 8.000 unità, il motore subisce un degrado graduale e lineare, raggiungendo i 19.80 ms. La soglia critica dei 16.6 ms viene oltrepassata una volta superate le 4.000 luci, dimostrando che in assenza di shading complesso, il partizionamento in cluster del volume di vista minimizza l'overhead senza necessitare di scritture multiple in memoria. Durante l'intero test, le draw call di Godot sono rimaste ferme a 1, permettendo alla GPU di concentrarsi esclusivamente sul calcolo luminoso.

Il comportamento di Unity evidenzia, invece, un limite architetturale alla base della pipeline di illuminazione. I test, eseguiti sia in Forward che in Deferred fino all'hard-cap del motore di 1.000 luci per telecamera, mostrano una "curva" totalmente piatta: il frame time oscilla in modo impercettibile tra i 14.03 ms e i 14.18 ms indipendentemente dal numero di luci o dal render path scelto.

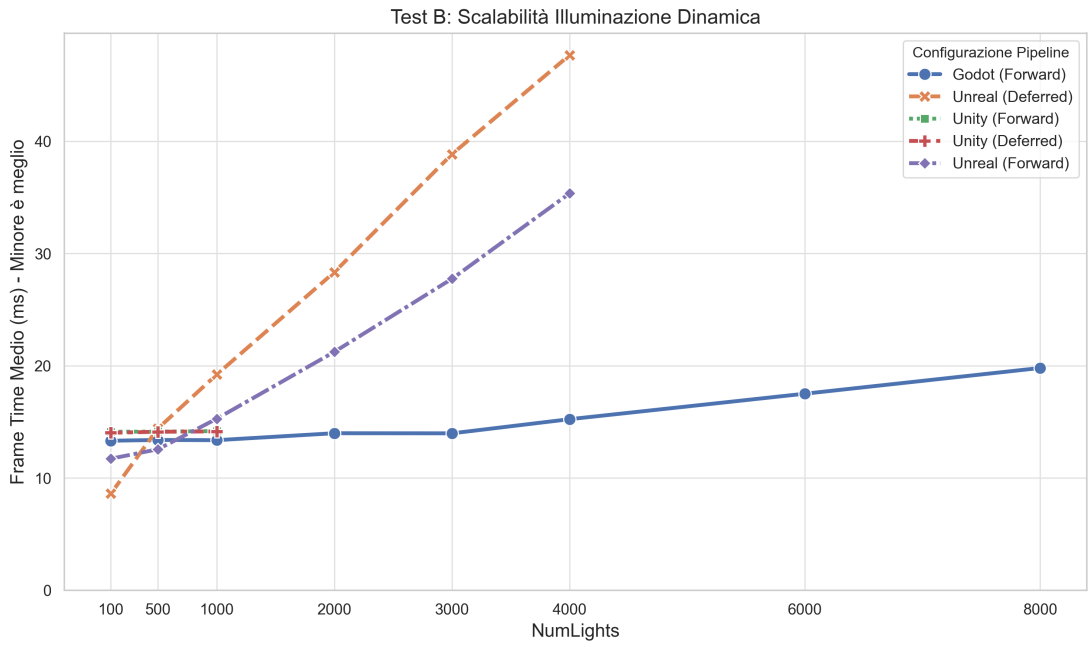

Figura 4.4: Risultati prestazionali del Test B.

Questo comportamento è sintomo di un collo di bottiglia sulla CPU: l'invio di oltre 17.000 draw call satura il Main Thread prima ancora che il fragment shader possa saturare le risorse grafiche, annullando le differenze prestazionali tra Forward e Deferred in questo specifico setup.

Unreal Engine fornisce i dati più rappresentativi rispetto ai limiti e ai vantaggi del Deferred Rendering. Con ridotte quantità di luci, la pipeline Deferred risulta superiore, registrando 8.61 ms contro gli 11.72 ms della controparte Forward. Tuttavia, le prestazioni del Deferred diminuiscono all'aumentare delle luci:

Quando il numero di luci aumenta, l'architettura Deferred comincia a soffrire l'enorme richiesta di banda passante necessaria per leggere e scrivere sul G-Buffer. Raggiunte le 500 luci, il Forward si dimostra più performante (12.56 ms contro i 14.41 ms del Deferred). Spingendo il test a 4.000 luci, il Deferred collassa a 47.68 ms, mentre il Forward si stabilizza a 35.36 ms. Questo comportamento è indicativo del principale problema dell'architettura Deferred: il G-Buffer impone un costo di accesso alla memoria che ripaga solo quando le luci sono in quantità minore e associate a calcoli fisici e ombre estremamente complessi; in scenari di pura rasterizzazione massiva di luci puntiformi, la banda di memoria diventa il fattore limitante primario.

4.1.4 Test C

L'ultimo test ha valutato l'impronta architetturale e le strategie di stoccaggio della VRAM, applicando materiali PBR con risoluzione nativa a 8K.

Godot conferma una gestione della memoria di tipo deterministico e statico. Nella baseline, l'impronta rilevata è di 1546 MB, corrispondente all'esatto peso matematico degli asset non compressi caricati nella scena. L'attivazione delle mipmaps introduce un incremento dell'occupazione di memoria pari a circa il 33%, innalzando la VRAM a 2101 MB. Tale costo si traduce in un vantaggio computazionale: il frame time scende da 3.39 ms a 2.99 ms. Questo dimostra l'efficacia delle mipmaps nel diminuire i cache miss della GPU durante il campionamento delle texture.

Totalmente opposto è l'approccio heavyweight di Unity. L'allocazione base richiede un massiccio quantitativo di memoria, pari a 4696 MB. Con l'aggiunta delle mipmaps, il consumo sale a 5859 MB (con frame time di 3.82 ms), un valore che su hardware di fascia medio-bassa esporrebbe il sistema a rischi di saturazione di memoria.

Le tecnologie di virtual texturing hanno mostrato risultati controintuitivi ma incontestabili.

In Unity, l'attivazione del VT diminuisce drasticamente il consumo, riportandolo a 1671 MB senza compromessi sul frame time (3.79 ms). Unreal Engine ha dimostrato una stabilità molto solida in ogni configurazione (frame time compreso tra 5.35 ms e 5.65 ms). Analizzando la memoria a runtime tramite il comando `Stat RHI`, è stato possibile ricostruire l'esatta configurazione fisica della memoria. Unreal Engine ha allocato una base di circa 1035 MB fisici, di cui 512 MB bloccati permanentemente per la Texture Streaming Pool[2].

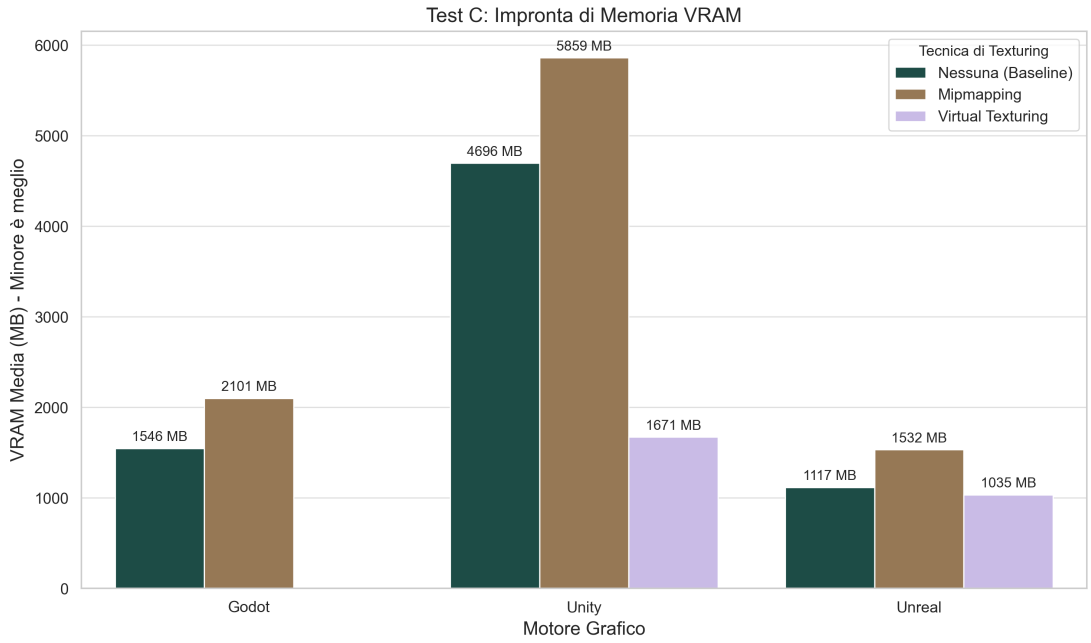

Figura 4.5: Risultati prestazionali del Test C.

Questi dati smentiscono il "falso mito" secondo cui il Virtual Texturing risparmia risorse a prescindere: in scenari relativamente scarichi di asset, esso introduce al contrario un overhead dovuto alla pre-allocazione dei buffer. Il suo reale scopo è disaccoppiare la risoluzione degli asset dalla VRAM consumata, garantendo una linea di consumo piatta.

Concludo questa sezione con una tabella che, per ogni engine, riassume le configurazioni che hanno restituito i migliori risultati in termini di frame time, draw call e consumo di memoria:

[2]La **Texture Streaming Pool** è l'area di memoria effettiva dove vengono caricati e scambiati i dati delle texture durante l'esecuzione. Essa è pre-allocata per garantire una gestione efficiente e prevedibile della memoria, evitando overhead dinamici e frammentazione che potrebbero derivare da allocazioni e deallocazioni frequenti.

Tabella 4.1: Riassunto delle migliori configurazioni di ottimizzazione per ciascun motore grafico.

Test Case	**Metrica Principale**	**Unity**	**Unreal Engine**	**Godot**
A1: Geometria	Frame Time (Instancing + LOD)	7.75 ms	5.99 ms (Nanite)	**0.62 ms**
A2: Culling	Frame Time (Occlusion Culling)	3.57 ms (Umbra)	8.22 ms (HZB)	**2.35 ms**
B: Luci (1000)	Frame Time (Path Ottimale)	14.1 ms	14.0 ms (Forward)	**13.5 ms**
C: VRAM	Occupazione Minima (Asset 8K)	1671 MB (VT)	**1035 MB (VT)**	1546 MB

4.2 Valutazione Qualitativa

Oltre all'impatto quantitativo, è fondamentale valutare l'impatto qualitativo che le diverse tecniche di ottimizzazione restituiscono a schermo. L'efficacia di un'architettura di rendering o di un'ottimizzazione non si misura solamente nella riduzione del frame time, ma anche nella sua capacità di mantenere invariata la qualità visiva percepita.

Dall'ispezione degli screenshot acquisiti durante le sessioni di test, le tecniche analizzate possono essere classificate in due macro-categorie: ottimizzazioni Lossless (visivamente conservative) e ottimizzazioni Lossy (con degrado visibile).

4.2.1 Ottimizzazioni Lossless

L'analisi qualitativa ha confermato che la maggior parte delle tecniche impiegate nei Test A1 e A2 operano in modo del tutto trasparente per l'osservatore.

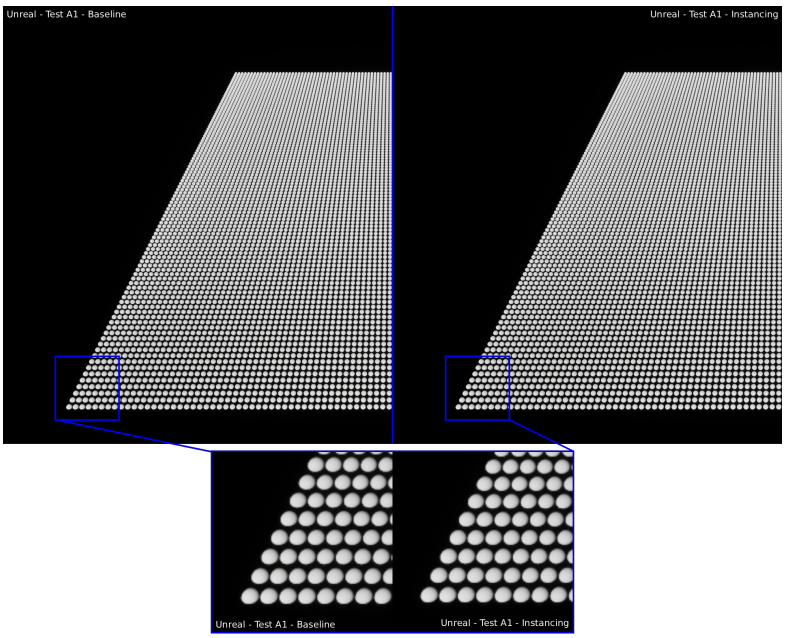

Figura 4.6: Confronto qualitativo tra la configurazione baseline e instancing di Unreal, con dettaglio.

Il GPU Instancing, il MultiMesh e l'SRP Batcher non hanno introdotto alcuna variazione nei calcoli di shading, nelle ombreggiature o nella silhouette della griglia di sfere rispetto al rendering della baseline.

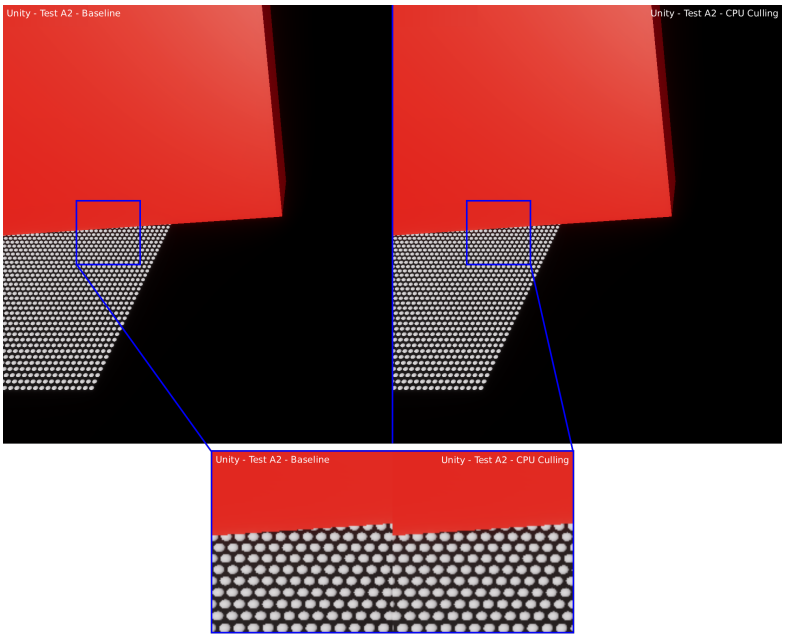

Figura 4.7: Confronto qualitativo tra la configurazione baseline e culling di Unity, con dettaglio.

Allo stesso modo, gli algoritmi di culling sia lato CPU che GPU si sono dimostrati molto precisi nel calcolo del PVS: non è stato riscontrato alcun artefatto visivo, flickering o scomparsa della geometria ai bordi dello schermo.

Nel contesto del Test B, il confronto qualitativo tra le pipeline forward e deferred non ha evidenziato alcuna differenza visiva. L'output a schermo risulta identico in ogni suo dettaglio. Entrambe le pipeline, pur adottando approcci algoritmici molto diversi per la gestione e l'accumulo dei dati, condividono gli stessi modelli matematici per il calcolo dell'illuminazione e dello shading, garantendo così una coerenza visiva totale.

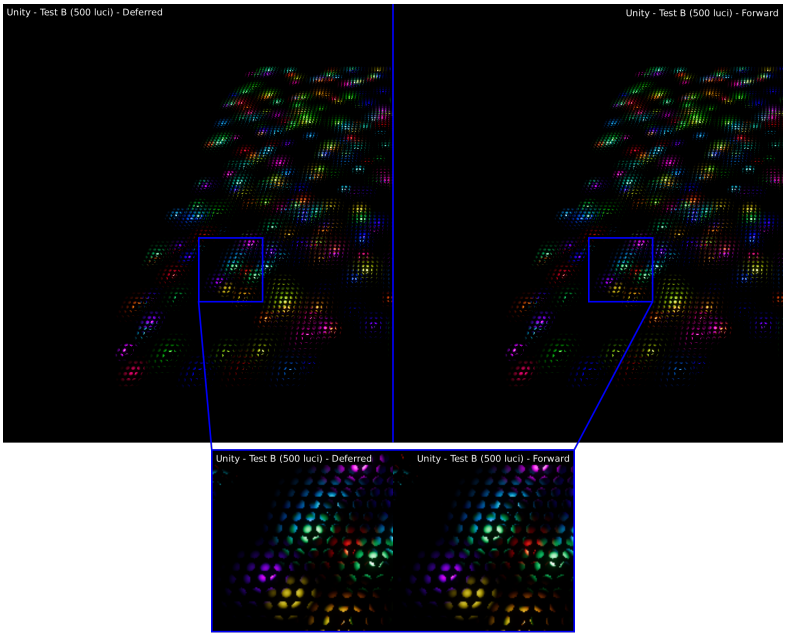

Figura 4.8: Confronto qualitativo tra pipeline forward e deferred in Unity.

L'assenza di differenze visive conferma, quindi, che la scelta tra architettura forward e deferred rappresenta un compromesso puramente computazionale e legato alla memory bandwidth, senza comportare alcuna rinuncia o differenza nella fedeltà visiva della scena.

4.2.2 Ottimizzazioni con degrado visibile (Lossy)

Le uniche differenze qualitative sono emerse con l'attivazione dei sistemi di scalabilità geometrica e di ottimizzazione sui materiali.

Per quanto riguarda la geometria, i sistemi di LOD classici hanno generato il tipico artefatto di popping visivo. Man mano che la telecamera si allontanava, la sostituzione discreta dei modelli ad alta densità con le loro varianti decimate ha

portando a una visibile alterazione delle silhouette. In alcuni casi, come Godot, questo sintomo è visibile anche con la telecamera statica, a causa di una gestione aggressiva dei livelli LOD.

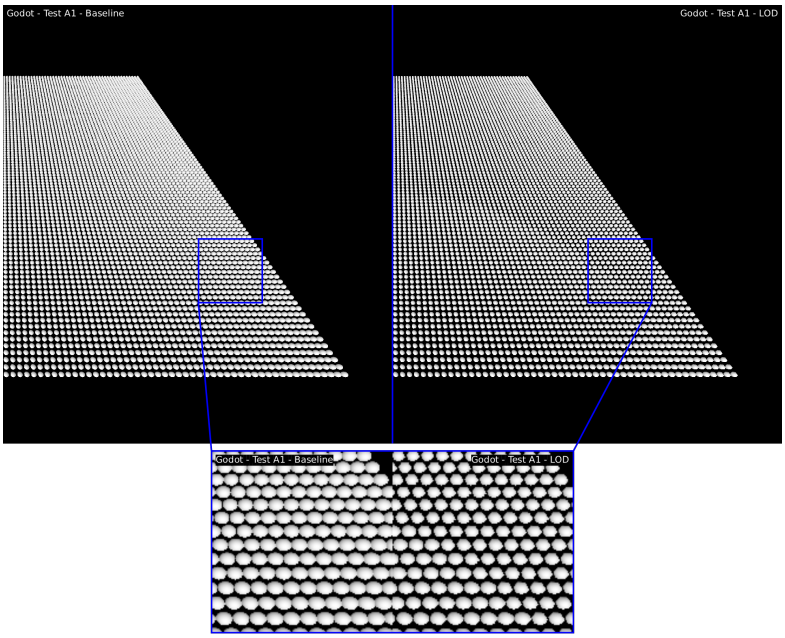

Figura 4.9: Confronto qualitativo tra la configurazione baseline e LOD di Godot, con dettaglio.

Osservando la griglia nel suo insieme, nella porzione di destra (LOD) è possibile notare un "semicerchio" che attraversa le sfere, evidenziando la linea di variazione del LOD. Sotto questo aspetto qualitativo, l'approccio Nanite di Unreal Engine si è distinto nettamente: l'utilizzo della geometria virtualizzata ha eliminato quasi totalmente l'effetto di popping, mantenendo la percezione di rotondità delle sfere anche a distanze elevate.

Nel Test C, l'analisi qualitativa ha evidenziato come il rendering nativo di texture ad altissima risoluzione senza filtri spaziali produca un risultato problematico. La geometria in lontananza mostra un severo rumore ad alta frequenza e aliasing molto marcato, causato dal sottocampionamento.

L'attivazione delle mipmaps ha risolto in modo evidente questa criticità, pur introducendo un ammorbidimento dei dettagli all'aumentare della distanza. Il ris ultato è un'immagine più pulita e visivamente stabile rispetto alla "caotica" baseline.

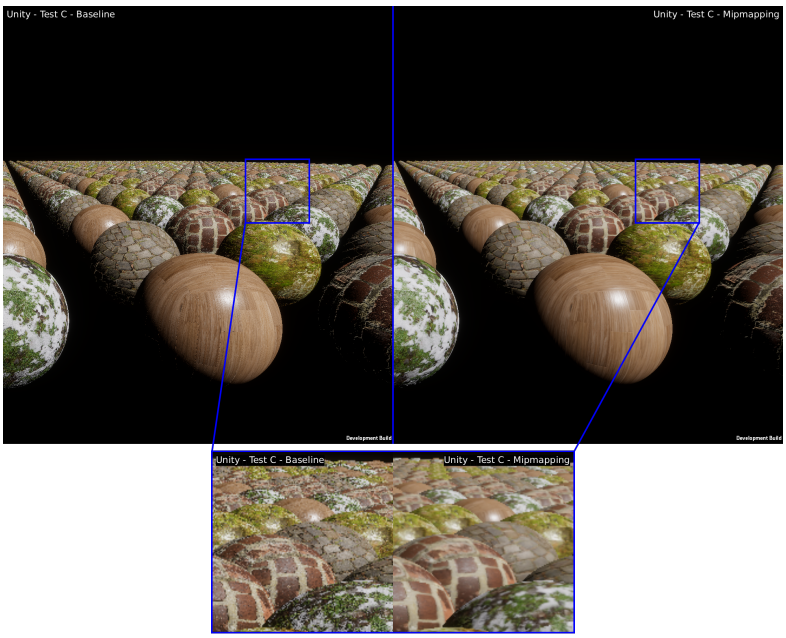

Figura 4.10: Confronto qualitativo tra la configurazione baseline e Mipmapping di Unity, con dettaglio.

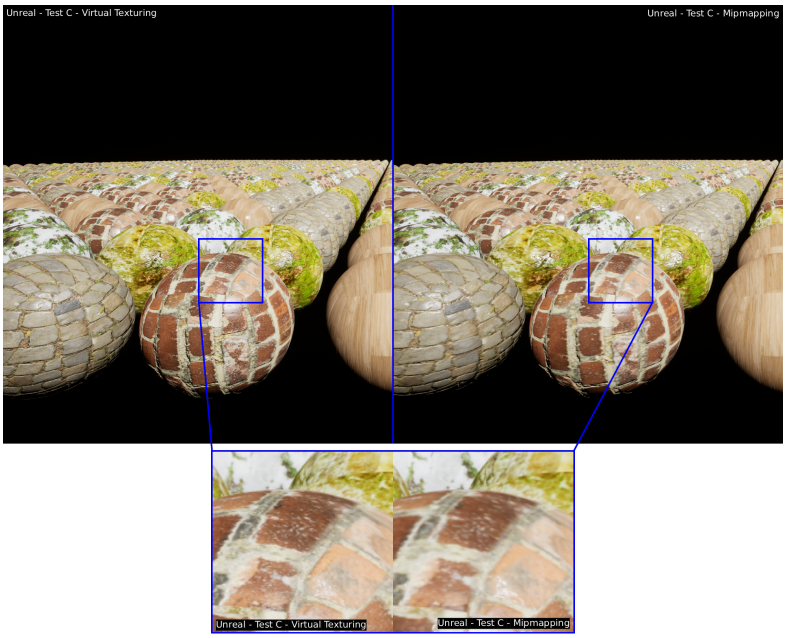

Figura 4.11: Confronto qualitativo tra la configurazione mipmaps e virtual texturing di Unreal, con dettaglio.

Tuttavia, confrontando il mipmapping tradizionale con il virtual texturing, l'ultimo dimostra un'assoluta superiorità qualitativa. In particolare in Unreal Engine, l'attivazione del VT non solo garantisce la stessa pulizia visiva e assenza di aliasing delle mipmaps, ma restituisce superfici più nitide e ricche di dettagli.

A differenza del mipmapping standard, che applica una riduzione di risoluzione globale e spesso troppo conservativa per intere texture, l'algoritmo SVT di Unreal calcola dinamicamente l'ingombro a schermo dei singoli tile. Ciò consente all'engine di caricare e campionare tile a risoluzione più elevata dove la prospettiva lo richiede, preservando un livello di dettaglio che il sistema a mipmaps sfocherebbe preventivamente.

4.3 Valutazione dell'Esperienza di Sviluppo e Workflow

Per ottenere un quadro architetturale completo è necessario valutare l'esperienza di sviluppo e l'efficienza del workflow di ciascuno dei motori testati. L'interazione diretta con i tre engine durante l'implementazione degli stress test ha evidenziato profonde differenze filosofiche e operative.

4.3.1 Unity

Unity si posiziona come un ecosistema ibrido. Se da un lato l'engine cerca di eguagliare la fedeltà visiva e la complessità strutturale dei motori AAA come Unreal, dall'altro mostra evidenti limiti nel reggere questo confronto senza richiedere importanti ottimizzazioni manuali. Tuttavia, il vero punto di forza di Unity risiede nella sua modularità: l'engine espone quasi ogni singolo parametro architetturale attraverso l'interfaccia grafica, permettendo un controllo capillare assente nei competitor. A questo si aggiunge un'infrastruttura di documentazione e una community di dimensioni tali da rendere le fasi di troubleshooting e debug molto più leggere rispetto agli altri motori.

4.3.2 Unreal Engine

Il flusso di lavoro con Unreal Engine 5 restituisce immediatamente il "feel" di un prodotto di stampo industriale. La fedeltà visiva nativa è la più alta tra i tre engine, insieme a un'interfaccia (forse troppo) semplice e un sistema di visual scripting molto maturo. A differenza di Unity ma soprattutto Godot, Unreal restituisce immediatamente la sensazione di essere un engine pensato per progetti di grande scala, con sistemi di ottimizzazione progettati per gestire quantità di dati e complessità tipiche di produzioni AAA. Un esempio è la sua pipeline di importazione degli asset: l'import e la compilazione dei materiali PBR in 8K ha richiesto

una quantità di tempo irrisoria, a dimostrazione di decenni di ottimizzazione del motore. Tuttavia, la complessità interna del motore si riflette sui tempi di compilazione: per la prima build (di una scena pressochè vuota) ha impiegato circa 30 minuti, necessari probabilmente per l'impacchettamento degli innumerevoli sottosistemi integrati. L'unica vera problematica operativa risiede nella GUI: l'engine predilige l'uso di grandi "macro-interruttori" per abilitare funzionalità complesse, come ad esempio Nanite, ma la disattivazione di singole micro-funzioni richiede spesso la navigazione in sottomenù annidati e poco intuitivi.

4.3.3 Godot

Godot può essere considerato l'equivalente di Blender nell'ambito dei motori grafici: un software estremamente leggero, open-source, e capace di essere eseguito su hardware di fascia piuttosto bassa. Questo può essere un vantaggio o uno svantaggio a seconda del contesto: se da un lato l'engine è accessibile a chiunque e permette di realizzare progetti con requisiti hardware minimi, dall'altro non è attualmente progettato per gestire produzioni di grande scala o asset di altissima complessità.

L'engine offre un controllo architetturale pressochè totale, ma impone che questo avvenga esclusivamente tramite approccio programmatico, appoggiandosi al proprio linguaggio nativo GDScript. Questo, in aggiunta alla poco intuitiva logica di strutturazione basata su alberi di scene e nodi, richiede una ripida curva di apprendimento e può risultare disorientante.

Le idee coraggiose e la natura open-source di Godot lo rendono un vero "game engine per tutti", insuperabile in progetti di piccola o media scala, dove l'assenza totale di overhead gli permette di dominare sui competitor come dimostrato nel Test A. Tuttavia, il motore risulta ancora acerbo non appena si prova ad uscire da questo scope: la gestione degli asset complessi ha causato pesanti colli di bottiglia operativi. L'importazione delle texture 8K ha richiesto tempi di attesa insostenibili (oltre 40 minuti complessivi), evidenziando che l'editor non è attualmente progettato per elaborare quantità di dati tipiche di produzioni AAA.

Capitolo 5

Conclusioni e Sviluppi Futuri

5.1 Sintesi del Lavoro

Questo elaborato ha analizzato le pipeline di rendering di Unity, Unreal Engine 5 e Godot Engine 4 per valutarne le prestazioni in condizioni di alto carico computazionale. Utilizzando il Frame Time come metrica principale e fissando l'obiettivo di fluidità a 16.6 ms per fotogramma, è stato possibile individuare chiari colli di bottiglia e limiti strutturali specifici per ciascun motore grafico.

L'analisi ha evidenziato come il real-time rendering sia una disciplina basata sul bilanciamento di tre risorse: cicli di CPU, banda di memoria e tempo di esecuzione degli shader sulla GPU. La ricerca ha dimostrato che non esiste un'architettura universale: la scelta di un motore grafico comporta un compromesso intrinseco tra la trasparenza algoritmica, la flessibilità e l'astrazione hardware avanzata.

5.2 Matrice Decisionale

Sulla base dei dati sperimentali, si propone la seguente classificazione per l'adozione dei motori grafici in relazione alle necessità progettuali:

5.3 Limiti della sperimentazione

È doveroso specificare che la natura dei risultati ottenuti è strettamente legata alla scelta di concentrarsi sul rendering rasterizzato. Questa scelta è stata funzionale al mantenimento di un controllo rigoroso sulle variabili in gioco, per i seguenti motivi:

Tabella 5.1: Matrice di decisione per la selezione del motore grafico in base alle prestazioni e al contesto d'uso.

Motore	Approccio	Contesto d'uso ideale
Godot	Lightweight	Progetti indipendenti, necessità di controllo, budget memoria limitati.
Unity	Versatile	Sviluppo cross-platform, necessità di personalizzazione tramite SRP, team con competenze di profilazione.
Unreal 5	Heavyweight	Produzioni AAA, alta complessità progettuale.

- **Purezza del dato:** Le tecniche di Hardware Ray Tracing e le tecnologie di ricostruzione dell'immagine tramite AI (DLSS, FSR, XeSS) sono state escluse. Dato che questi sistemi operano come processi "ibridi" o di post-processing che spesso alterano il costo computazionale nativo della pipeline, la loro inclusione avrebbe introdotto variabili aleatorie, compromettendo la riproducibilità dei benchmark.

- **Focus sull'architettura raster:** L'analisi si è concentrata solo sulla rasterizzazione classica per analizzare esattamente il peso di ogni singola operazione. Questo ha permesso di valutare le prestazioni base di ogni motore grafico in modo chiaro e preciso.

- **Complessità di Nanite:** Il sistema Nanite di Unreal Engine è stato esaminato esclusivamente sotto il profilo dell'efficienza della rasterizzazione tradizionale. Sebbene le sue potenzialità siano indiscusse, l'indagine si è mantenuta su scenari di complessità geometrica convenzionale. L'impiego di asset a densità poligonale estrema avrebbe infatti precluso un confronto equo con le architetture degli altri motori grafici.

5.4 Sviluppi Futuri

I limiti evidenziati suggeriscono diverse direzioni per le ricerche future:

- **Ray Tracing Integrato:** Analizzare l'impatto del Path Tracing sulle prestazioni, man mano che l'evoluzione delle GPU ridurrà la dipendenza dalla rasterizzazione classica.
- **Upscaling e IA:** Valutare come le tecnologie di ricostruzione neurale dell'immagine possano ridefinire i tradizionali obiettivi di ottimizzazione del frame time.
- **Geometria Virtualizzata:** Confrontare soluzioni avanzate per la gestione di scene ad altissima densità poligonale, come Nanite e i Mesh Shaders.

Il rendering del futuro richiederà agli sviluppatori di bilanciare sapientemente tecniche ibride, gestione dinamica della memoria e algoritmi supportati dall'Intelligenza Artificiale.

Bibliografia

[1] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, e S. Hillaire, *Real-Time Rendering*, 4a ed. Boca Raton, FL: A K Peters/CRC Press, 2018.

[2] D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, e R. Huebner, *Level of Detail for 3D Graphics*. San Francisco, CA: Morgan Kaufmann, 2003.

[3] O. Olsson e U. Assarsson, "Tiled Shading", *Journal of Graphics, GPU, and Game Tools*, vol. 15, n. 4, pp. 235–251, 2011.

[4] O. Olsson, M. Billeter, e U. Assarsson, "Clustered Deferred and Forward Shading", in *Proceedings of High Performance Graphics (HPG)*, Eurographics Association, pp. 87–96, Giugno 2012.

[5] C. Everitt, J. McDonald, e A. Rege, "Approaching Zero Driver Overhead", presentazione alla *Game Developers Conference (GDC)*, San Francisco, CA, Mar. 2014.

[6] H. Hoppe, "Progressive meshes", in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*, 1996, pp. 99–108.

[7] M. Garland e P. S. Heckbert, "Surface simplification using quadric error metrics", in *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*, 1997, pp. 209–216.

[8] F. Carucci, "Inside Geometry Instancing", in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, M. Pharr, Ed. Addison-Wesley Professional, 2005, cap. 3, pp. 47–67.

[9] Unity Technologies, "SRP Batcher: Speed up your rendering", *Unity Official Blog*, 2018. [Online]. Disponibile: `https://unity.com/blog/engine-platform/srp-batcher-speed-up-your-rendering`

[10] Unity Technologies, “Mesh LOD: Generate LODs” e “LOD Quality”, *Unity 6 Documentation*. [Online]. Disponibile: https://docs.unity3d.com/6000.3/Documentation/Manual/lod/mesh-lod-generate-lods.html

[11] Umbra3D, “Introduction to Occlusion Culling”, *Medium*, Gen. 2017. [Online]. Disponibile: https://medium.com/@Umbra3D/introduction-to-occlusion-culling-3d6cfb195c79

[12] Unity Technologies, “GPU Resident Drawer (GPU Culling)”, *Unity 6 Documentation*. [Online]. Disponibile: https://docs.unity3d.com/6000.3/Documentation/Manual/urp/gpu-culling.html

[13] Unity Technologies, “Render Graph System”, *Unity 6 Documentation*. [Online]. Disponibile: https://docs.unity3d.com/6000.1/Documentation/Manual/urp/render-graph-introduction.html

[14] B. Karis, “Journey to Nanite”, Keynote presentation alla *High Performance Graphics (HPG)*, 2022. [Online]. Disponibile: https://www.highperformancegraphics.org/slides22/Journey_to_Nanite.pdf

[15] Epic Games, “Nanite GPU Driven Materials”, presentazione alla *Game Developers Conference (GDC)*, San Francisco, CA, 2024. [Online]. Disponibile: https://media.gdcvault.com/gdc2024/Slides/GDC+slide+presentations/Nanite+GPU+Driven+Materials.pdf

[16] Godot Engine Contributors, “Godot Architecture Diagram”, *Godot 4.3 Documentation*. [Online]. Disponibile: https://docs.godotengine.org/en/stable/engine_details/architecture/godot_architecture_diagram.html

[17] Godot Engine Contributors, “Custom modules in C++” e “Build System (SCons)”, *Godot 4.3 Documentation*. [Online]. Disponibile: https://docs.godotengine.org/en/stable/engine_details/engine_api/custom_modules_in_cpp.html

[18] Godot Engine Contributors, “Internal Rendering Architecture: Forward+”, *Godot 4.3 Documentation*. [Online]. Disponibile: https://docs.godotengine.org/en/stable/engine_details/architecture/internal_rendering_architecture.html

[19] Godot Engine Contributors, "Occlusion Culling", *Godot 4.3 Documentation*. [Online]. Disponibile: `https://docs.godotengine.org/en/stable/tutorials/3d/occlusion_culling.html`