ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

Progettazione e sviluppo di un Sistema di LOD Adattivo basato su Tessellazione Hardware e Geometry Shader

Elaborato in
Computer Graphics

Relatore
Prof.ssa Damiana Lazzaro

Presentata da
Marco Paggetti

Sessione Unica
Anno Accademico 2024/2025

Indice

Introduzione

In applicazioni moderne come i videogiochi, le simulazioni virtuali o strumenti di visualizzazione interattiva, è importante gestire efficacemente i dati relativi a ciò che viene mostrato. Le scene tridimensionali possono contenere un elevato numero di oggetti, ciascun composto da molti punti da elaborare in tempo reale, e per questo è necessario definire delle tecniche capaci di garantire un livello di qualità visiva soddisfacente senza compromettere le prestazioni dell'applicazione. In particolare, considerando che queste applicazioni sono ormai disponibili anche su calcolatori con potenza limitata, diventa utile adottare strategie di progettazione e sviluppo che permettano comunque di ottenere una buona qualità, ottimizzando il carico di lavoro a cui viene sottoposto l'hardware.

Proprio in questo contesto si colloca l'obiettivo principale di questa tesi, ovvero lo sviluppo di un sistema avanzato di visualizzazione di dettagli grafici dinamici (*Level of Detail dinamico*, *LOD*) nel rendering 3D in tempo reale. Il sistema sfrutta le potenzialità offerte dai Tessellation Shaders e dal Geometry Shader (componenti programmabili che consentono l'esecuzione di algoritmi personalizzati sulle moderne GPU), con l'intento di raggiungere un equilibrio ottimale tra qualità visiva e prestazioni.

Accanto a questo nucleo centrale, un secondo obiettivo è l'analisi approfondita degli aspetti tecnici dei due stadi della pipeline grafica appena menzionati. La tesi mira infatti a comprenderne a fondo la struttura, il funzionamento, le variabili built-in e le possibilità applicative, così da valutarne limiti e punti di forza in scenari real-time.

Per approfondire lo studio del LOD dinamico, in questa tesi è stata realizzata un'applicazione software in C/C++, basata sulle librerie di OpenGL 4.6 e sul linguaggio GLSL per la programmazione degli *shader*. Questo sistema genera dinamicamente diversi livelli di dettaglio (LOD) partendo da una geometria base, sfruttando tecniche avanzate come i **Tessellation Shaders** e il **Geometry Shader**. In particolare, l'applicazione, a partire da una geometria semplice, elabora automaticamente versioni progressivamente più complesse in funzione della posizione della telecamera, ottimizzando così il rendering in tempo reale e raggiungendo un elevato compromesso tra qualità grafica e performance computazionali. La presenza di un modello animato con *rig* permette inoltre di valutare l'integrazione di personaggi interattivi all'interno di scenari complessi, mentre la generazione di scene diversificate, composte sia da paesaggi naturali che da ambienti urbani, offre un valido terreno di test per l'applicazione e verifica delle tecniche proposte. Un ulteriore punto di interesse

riguarda la generazione procedurale dei dettagli (ad esempio vegetazione e oggetti di scena), che arricchiscono visivamente l'ambiente senza incidere negativamente sull'efficienza e sulla fluidità del rendering.

Le soluzioni implementate vengono inoltre analizzate sia dal punto di vista delle prestazioni sia da quello della qualità visiva percepita, considerando differenti prospettive (telecamera e punto di vista del giocatore) al fine di evidenziare l'impatto reale delle tecniche di LOD sulla resa complessiva della scena.

La tesi risulta così organizzata:

- **Capitolo 1**: descrizione della pipeline di rendering, intesa come la sequenza di operazioni eseguite dalla CPU e dalla GPU necessarie per passare dalle geometrie degli oggetti alla loro visualizzazione su schermo.
- **Capitolo 2 e 3**: analisi dei Tessellation Shaders e Geometry Shader, approfondendone la struttura, il funzionamento e le possibilità applicative, con l'obiettivo di comprendere il ruolo dei due stadi nella realizzazione del LOD dinamico.
- **Capitolo 4**: richiamo di alcuni concetti di analisi e algebra, in modo da predisporre una base concettuale per i capitoli successivi.
- **Capitolo 5**: descrizione dell'applicazione software realizzata, evidenziando le principali sfide incontrate e le soluzioni adottate.
- **Capitolo 6**: presentazione dei risultati ottenuti e confronto con tecniche alternative, per valutare in maniera chiara l'efficacia del sistema implementato.

Capitolo 1

Classic Rendering Graphic Pipeline

In generale una *pipeline* è una sequenza di unità di elaborazione dati, organizzate in maniera simile ad una catena di montaggio, dove l'output di ogni unità diventa l'input di quella successiva. Nella *Pipeline Grafica*, anche chiamata *Pipeline di Rendering*, queste unità hanno il compito di processare una scena costituita da oggetti, sorgenti luminose e una telecamera virtuale, convertendo il tutto in un'immagine bidimensionale finale, visualizzabile sullo schermo.

Il compito principale di una Pipeline di Rendering classica è quindi di trasformare una scena tridimensionale in una rappresentazione 2D, tenendo conto della geometria degli oggetti, della posizione e orientamento della telecamera, delle fonti luminose (e delle loro proprietà), delle caratteristiche dei materiali che compongono gli oggetti, delle texture e delle equazioni di *shading* che definiscono l'illuminazione e l'aspetto finale delle superfici [1].

1.1 Struttura generale

Una pipeline consiste in una sequenza ordinata di *stage* (o *stadi*), ciascuno dei quali dipende dal risultato dello stage precedente. Sebbene le GPU moderne eseguano molti stage in parallelo, le prestazioni complessive sono influenzate dallo stage più lento e questo determina i cosiddetti colli di bottiglia della pipeline.

Una pipeline di rendering in tempo reale (*real-time graphics pipeline*) si compone dei seguenti quattro stadi principali, ognuno dei quali, come si chiarirà in seguito, può essere ulteriormente suddiviso in sotto-unità di elaborazione:

1. **Application Stage**
2. **Geometry Processing Stage**
3. **Rasterization Stage**
4. **Pixel Processing Stage**

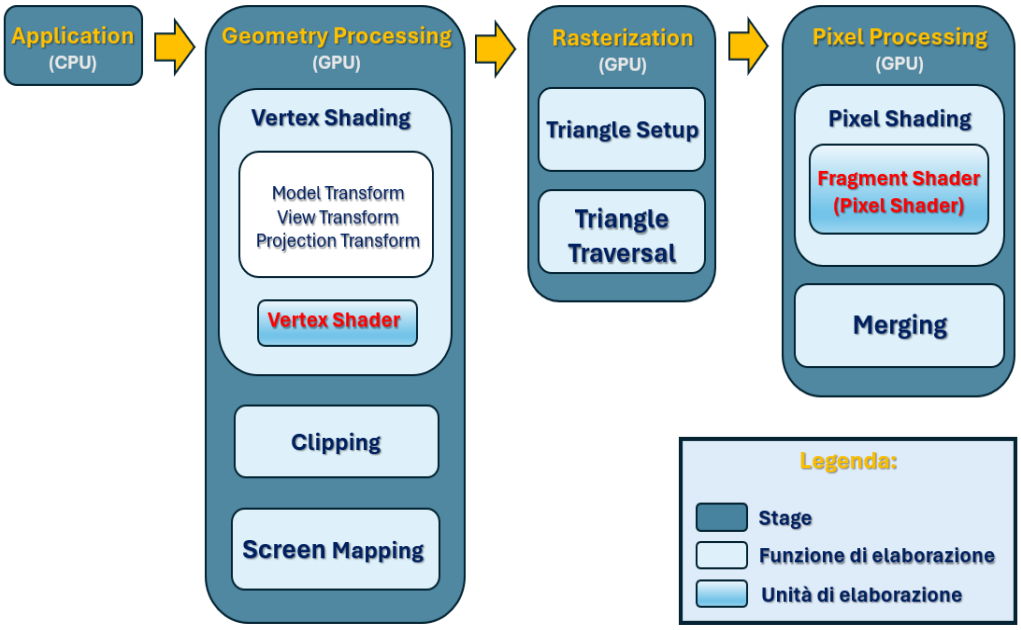

Figura 1.1: Schema della pipeline di rendering classica.

1.1.1 Application Stage

L'*Application Stage* rappresenta la fase iniziale della pipeline grafica ed è gestito interamente dal software, tipicamente in esecuzione sulla CPU. A differenza degli stadi successivi, non è strutturato come una sotto-pipeline, ma coincide con il codice dell'applicazione sviluppata dal programmatore. In alcuni casi specifici, parte delle sue operazioni possono essere demandate alla GPU tramite i *Compute Shader*, ma l'approccio tradizionale è quello che prevede l'utilizzo primario del processore centrale.

In questa fase il programmatore ha il pieno controllo del flusso, poiché qui risiedono le logiche fondamentali dell'applicazione. Oltre alla definizione degli oggetti e dei modelli che compongono la scena (tramite importazione da file esterni o mediante definizione algoritmica) e l'organizzazione dei loro vertici secondo quanto richiesto dai successivi stadi della pipeline, tra i compiti più comuni di questo stage si trovano: la gestione degli input (tastiera, mouse, controller o altri dispositivi), la gestione della logica applicativa e delle interazioni, il controllo degli eventi, il calcolo di *collision detection* e altre operazioni legate alla fisica della scena, la definizione di animazioni e trasformazioni globali degli oggetti. Inoltre, rientrano in questa fase tecniche di *culling* (per determinare gli oggetti non visibili che quindi possono essere scartati) e la gestione delle istanze (per ottimizzare il rendering di oggetti ripetuti nella scena).

L'output dell'Application Stage consiste in insiemi di vertici organizzati in modo da essere interpretati come primitive geometriche (punti, linee, triangoli o *patch* (vedi sezione 2.2.2)) dagli stadi successivi della pipeline. Queste primitive costituiscono la base su cui si innesterà l'intero processo di elaborazione grafica che porterà al risultato finale visibile sullo schermo.

1.1.2 Geometry Processing Stage

Il *Geometry Processing Stage* rappresenta il cuore di una pipeline grafica classica, in quanto gestisce la trasformazione e la preparazione della geometria della scena per il successivo processo di rasterizzazione, convertendo le primitive in *frammenti* pronti per la visualizzazione sullo schermo. Questo insieme di operazioni viene eseguito principalmente sulla GPU, che mette a disposizione sia unità hardware programmabili (*Shader Units*), sia unità non programmabili (*fixed-function*), le quali forniscono funzionalità standardizzate per alcune operazioni fondamentali. Lo scopo principale di questo stadio è quello di ricevere in ingresso un insieme di primitive grafiche (punti, linee, triangoli) provenienti dall'Application Stage e restituire in uscita primitive trasformate e proiettate nello spazio dello schermo, pronte per essere discretizzate in pixel.

Il Geometry Processing Stage è composto da una serie di sotto-stadi concatenati, ognuno con dei compiti specifici: **Vertex Shading**, **Clipping** e **Screen Mapping**. Ciascuno di questi passaggi è fondamentale per portare gli oggetti da una rappresentazione tridimensionale e locale, rispetto al sistema di riferimento del modello, a una rappresentazione bidimensionale coerente con la vista della telecamera.

1.1.2.1 Vertex Shading

Il *Vertex Shading* rappresenta il primo vero e proprio stadio programmabile della pipeline. Qui ogni vertice della geometria viene processato in maniera indipendente dagli altri e può essere arricchito di informazioni utili per le fasi successive. Dal punto di vista architetturale, la GPU esegue molteplici invocazioni del Vertex Shader in parallelo, ma ciascuna di esse ha visibilità esclusivamente sui propri vertici. Il ***Vertex Shader*** calcola la posizione finale del vertice rendendola utilizzabile dai successivi stadi della pipeline, e può inoltre fornire attributi aggiuntivi come normali, coordinate di texture o colori, utili alle fasi di illuminazione e shading successive.

Per il calcolo della posizione finale del vertice, il Vertex Shader utilizza tre trasformazioni fondamentali: *Model Transformation*, *View Transformation* e *Projection Transformation*.

Model Transformation

Le coordinate dei vertici di un modello sono inizialmente espresse nel ***Model Space*** (o *Object Space*), il sistema di riferimento locale in cui il modello è stato definito durante la fase di creazione. Questo non tiene conto della posizione e dell'orientamento dell'oggetto all'interno della scena. Per posizionare correttamente l'oggetto nella scena, le sue coordinate devono essere trasformate nel ***World Space*** tramite la cosiddetta *Model Transformation*. Questa trasformazione permette di collocare l'oggetto nello spazio globale e di orientarlo correttamente rispetto agli altri modelli presenti nella stessa scena.

View Transformation

Una volta che l'oggetto si trova posizionato nel *World Space*, entra in gioco la telecamera virtuale, poiché solamente ciò che è visibile da essa può essere mostrato a schermo. La telecamera è definita da una posizione e da una direzione all'interno del mondo (World Space). Per semplificare le operazioni successive, conviene riformulare la scena rispetto al sistema di riferimento della telecamera. Questa trasformazione viene fatta tramite la *View Transformation*, che consiste in una traslazione e una rotazione capaci di portare la telecamera all'origine del sistema di riferimento e di orientarla in modo che guardi verso la direzione negativa dell'asse z, con l'asse y positivo verso l'alto e l'asse x positivo verso destra. In questo modo, tutte le coordinate della scena 3D vengono espresse nel cosiddetto ***View Space*** (anche chiamato *Camera Space* o *Eye Space*), che è il sistema di riferimento centrato e orientato sulla telecamera.

Projection Transformation

La *Projection Transformation* è una trasformazione omogenea (ottenuta tramite moltiplicazione di una matrice), seguita da una divisione prospettica non lineare, che inizialmente mappa il solo volume di vista della telecamera (detto *View Volume*) in un cubo unitario chiamato ***Canonical View Volume*** (CVV), delimitato dai punti (-1,-1,-1) e (1,1,1). Le stesse operazioni vengono applicate anche ai vertici dei modelli dell'intera scena, che risultano così mappati nello stesso sistema di riferimento del CVV. Tale normalizzazione ha l'obiettivo di semplificare notevolmente le operazioni successive, in particolare il *clipping* e lo *screen mapping*, che mirano a rendere visibile su schermo solo la parte di scena che ricade nel View Volume.

L'obiettivo principale della Projection Transformation è quello di uniformare i due più comuni tipi di proiezione utilizzati nelle pipeline grafiche:

- **Ortografica**: le linee parallele rimangono invariate e la dimensione degli oggetti non dipende dalla loro distanza dalla telecamera. Questa proprietà la rende utile in contesti tecnici, come i sistemi CAD o le visualizzazioni ingegneristiche, dove è fondamentale preservare le proporzioni reali senza deformazioni prospettiche. In questo caso, il volume di vista assume la forma di un parallelepipedo.
- **Prospettica**: riproduce il modo in cui l'occhio umano percepisce lo spazio. Le linee parallele tendono a convergere verso un punto di fuga e gli oggetti lontani appaiono più piccoli rispetto a quelli vicini. In questo caso, il View Volume assume la forma di un *frustum*, cioè un tronco di piramide a base rettangolare.

La Projection Transformation mira a rettificare la forma del View Volume, in modo da ottenere un risultato omogeneo e indipendente dal tipo di prospettiva utilizzata per semplificare i calcoli successivi.

All'inizio della fase di Projection Transformation, al View Volume viene applicata una matrice di proiezione che la trasforma in un parallelepipedo centrato nell'origine di un nuovo sistema di riferimento (il ***Clip Space***). La stessa matrice di trasfor-

mazione viene poi applicata anche a tutte le coordinate degli oggetti della scena, le cui coordinate trasformate risulteranno riferite al nuovo sistema di riferimento (Clip Space). Queste nuove coordinate sono tutte espresse in *coordinate omogenee*, cioè con quattro componenti (x, y, z, w). La quarta componente (w) rappresenta il parametro di prospettiva della scena.

Infine, per normalizzare ciascuna componente (x, y, z) nell'intervallo $[-1, 1]$, si applica il cosiddetto *Perspective Divide*: ciascuna componente (x, y, z) viene divisa per w, producendo le *Normalized Device Coordinates* (NDC). Con queste coordinate risultano quindi identificati i punti del CVV e i vari vertici della scena. È importante notare che, nel caso di proiezione ortografica, w rimane costante e pari a 1 per tutti i vertici; di conseguenza, la divisione non altera le coordinate e non introduce alcuna scalatura prospettica.

Questa operazione ha due effetti fondamentali:

1. Nel caso di proiezione prospettica, gli oggetti lontani dalla telecamera vengono scalati, apparendo più piccoli rispetto a quelli vicini.
2. Le nuove coordinate dei vertici che si trovavano all'interno del *View Volume* risultano comprese nell'intervallo $[-1, 1]$ in tutte e tre le dimensioni, mentre quelle dei vertici esterni cadono fuori da tale intervallo.

In questo modo i vertici sono standardizzati e pronti per le fasi successive della pipeline: clipping semplificato e mappatura delle NDC sui pixel dello schermo.

1.1.2.2 Clipping

Una volta portate le primitive (gli oggetti della scena) nello spazio di clip ed aver definito il volume di vista canonico (il CVV), è necessario verificare quali primitive siano effettivamente visibili. Questo compito viene svolto dal *Clipping*, che ha il ruolo di eliminare tutte le primitive che si trovano completamente al di fuori del *Canonical View Volume* e di tagliare (*clip*) quelle che lo attraversano solo parzialmente. Solo le primitive intere o parzialmente interne vengono quindi trasmesse agli stadi successivi. Il Clipping risulta fondamentale per garantire che le primitive passate alla rasterizzazione siano limitate alla regione visibile, evitando sprechi di calcolo e artefatti grafici.

1.1.2.3 Screen Mapping

Dopo la fase di Clipping, i vertici sopravvissuti si trovano nello spazio delle Normalized Device Coordinates, cioè all'intero del Canonical View Volume. A questo punto, la pipeline deve trasformare queste coordinate nello spazio dello schermo, cioè nelle cosiddette ***Window Coordinates***.

Questa trasformazione, chiamata *Screen Mapping* o *Window-Viewport Transformation*, consiste nel riportare il cubo normalizzato all'interno del rettangolo della *viewport*, cioè la regione della finestra di rendering destinata alla visualizzazione. La viewport è definita da una posizione all'interno dello schermo, da una larghezza e da un'altezza.

In pratica, le coordinate x e y vengono scalate in base alle dimensioni della viewport e traslate per adattarsi alla sua posizione sullo schermo. Anche la coordinata di profondità z viene adattata, in modo che i valori dei vertici siano compatibili con l'intervallo utilizzato dal *Depth Buffer* durante la rasterizzazione ($[0, 1]$).

Il risultato finale è un insieme di vertici espressi nello spazio della finestra, pronti per essere convertiti in frammenti dalla fase di Rasterization.

1.1.3 Rasterization Stage

Il *Rasterization Stage*, noto anche come *Scan Conversion*, è uno stadio eseguito su GPU e rappresenta il momento in cui la pipeline grafica trasforma le primitive geometriche in frammenti discreti destinati a diventare pixel. Fino a questo punto, infatti, le informazioni gestite sono state di natura continua e astratta. La rasterizzazione ha ora il compito di determinare quali pixel dello schermo saranno coinvolti da ogni framento e quali attributi interpolati ciascun frammento dovrà possedere.

È importante chiarire che un ***fragment*** (o frammento) non corrisponde ancora a un pixel definitivo, ma una struttura temporanea che contiene tutte le informazioni necessarie per calcolare il colore e la visibilità finale, come: coordinate nello spazio della finestra (*Window Space*), *depth value* (la distanza dalla telecamera), normali interpolate, colori e coordinate di texture. Solo dopo essere stato processato negli stadi successivi, il frammento potrà eventualmente diventare un pixel effettivo sullo schermo, in base alle interdipendenze con altri fragment che insistono sullo stesso pixel.

Il processo di rasterizzazione può essere suddiviso in due fasi principali eseguite in sequenza: **Triangle Setup** e **Triangle Traversal**.

1.1.3.1 Triangle Setup

In questa fase vengono preparati tutti i dati necessari per la rasterizzazione delle primitive presenti in scena. Per ciascuna primitiva si calcolano le equazioni dei bordi nello spazio dello schermo, in modo da poter determinare in seguito se un pixel appartiene alla primitiva. Vengono inoltre predisposti i parametri necessari per interpolare correttamente attributi come colore, coordinate di texture e altre variabili, applicando la correzione prospettica per mantenere la coerenza geometrica nello spazio 3D.

1.1.3.2 Triangle Traversal

Dopo il setup iniziale, la primitiva viene effettivamente rasterizzata e i suoi confini vengono utilizzati per verificare se ciascun pixel ricade all'interno. Per ogni pixel interno viene generato un frammento, al quale vengono assegnati gli attributi interpolati a partire dai vertici originali. Contemporaneamente, viene calcolato il valore di profondità (*depth value*) per il *depth testing*, così da determinare se il frammento sarà visibile nella scena.

1.1.4 Pixel Processing Stage

Il *Pixel Processing Stage*, stadio finale eseguito sulla GPU, si occupa di trasformare i frammenti generati dallo stage precedente in valori finali da scrivere nel ***Frame Buffer***, ossia la memoria che conterrà l'immagine finale. Questo stadio può essere concettualmente suddiviso in due fasi principali: **Pixel Shading** e **Merging**.

1.1.4.1 Pixel Shading

Il *Pixel Shading* è la fase in cui viene calcolato il colore effettivo di ciascun frammento. Questo compito è eseguito da unità programmabili della GPU chiamate ***Fragment Shader*** (o *Pixel Shader*). Il programmatore può scrivere un programma che definisce come il colore di ciascun frammento deve essere determinato, utilizzando le informazioni interpolate dai vertici.

Il risultato di questa fase è un insieme di valori per ciascun frammento, tipicamente rappresentati dai componenti RGB (rosso, verde, blu) e dal *canale alpha*, che indica il grado di trasparenza del frammento. Queste informazioni sono pronte per essere scritte nel *Frame Buffer*.

1.1.4.2 Merging

Le strutture principali coinvolte in questa fase sono:

- **Color Buffer**: array bidimensionale in cui ogni posizione di memoria indica il colore di un pixel dello schermo. Poiché il colore dei pixel di un monitor si forma per sintesi additiva dai tre valori primari RGB, ogni posizione contiene tre componenti, corrispondenti ai canali rosso, verde e blu.
- **Depth Buffer** (o *Z-Buffer*): memorizza per ciascun pixel la distanza dalla telecamera della primitiva più vicina.

La fase di *Merging* si occupa di confrontare tutti i frammenti calcolati e di inserire nei buffer della GPU i soli valori che garantiscono la corretta visibilità e la coerenza dell'immagine finale.

Quando un frammento viene generato, il suo *depth value* viene confrontato con il valore memorizzato fino a quel momento nel Depth Buffer del pixel coinvolto. Poiché i valori di profondità seguono la convenzione in cui $z = 0$ corrisponde alla massima vicinanza alla telecamera e $z = 1$ alla massima lontananza, viene scelto il valore minore tra quello corrente e quello già presente. Il Color Buffer viene aggiornato con il colore del frammento che ha superato il test di profondità. Questo meccanismo permette alla pipeline di risolvere correttamente la visibilità, indipendentemente dall'ordine con cui le primitive vengono inviate alla GPU, garantendo che l'immagine finale rappresenti sempre la scena corretta.

Il Depth Buffer può memorizzare un solo valore per pixel, quindi non è in grado di gestire la trasparenza parziale in autonomia. Per gestire frammenti trasparenti si utilizza il canale alpha (associato al Color Buffer) e algoritmi di *alpha blending*

o ordinamento dei frammenti, che combinano i colori dei frammenti trasparenti secondo il loro grado di opacità.

Tutti questi valori vengono infine combinati nel Frame Buffer, cioè il contenitore finale che rappresenta l'immagine completa pronta per essere visualizzata sullo schermo. Il Frame Buffer può includere più buffer (Color, Depth, Stencil) e viene inviato al display alla fine della pipeline grafica. Altri buffer specializzati possono essere utilizzati per effetti avanzati come rifrazioni, trasparenze multiple o accumulazioni di valori intermedi per post-processing.

1.2 Stadi Programmabili della GPU

Come già accennato nelle sezioni dedicate ai singoli stadi di una pipeline, una delle innovazioni più significative nell'evoluzione della grafica in tempo reale è stata l'introduzione degli shader programmabili. In origine, infatti, la pipeline grafica era quasi interamente di tipo *fixed-function*: ogni fase eseguiva operazioni predeterminate dall'hardware, con pochi parametri configurabili dal programmatore. Questa rigidità era sufficiente per i primi sistemi di rendering, ma diventava un limite quando si richiedeva maggiore qualità visiva o effetti personalizzati.

La svolta avvenne nei primi anni Duemila, con l'introduzione del concetto di ***Shader Unit***: piccoli programmi scritti dal programmatore, compilati ed eseguiti direttamente dalla GPU su migliaia di dati in parallelo. Grazie a questi, il programmatore è in grado di definire algoritmi personalizzati per calcolare trasformazioni geometriche, illuminazione, gestione dei materiali e effetti visivi avanzati.

Come già discusso nelle sezioni precedenti, i due stadi programmabili di base sono il **Vertex Shader**, che elabora i vertici della geometria, e il **Fragment Shader**, che determina il colore dei frammenti dopo la rasterizzazione.

1.2.1 Vertex Shader

Il *Vertex Shader* rappresenta il primo stadio programmabile della pipeline grafica. Ogni vertice di un oggetto viene elaborato in maniera indipendente dagli altri, consentendo di applicare trasformazioni geometriche e di calcolare attributi utili per le fasi successive, come normali trasformate, coordinate di texture o colori.

La sua funzione principale è quella di convertire le coordinate dei vertici dai diversi spazi di rappresentazione (*Model Space*, *World Space*, *View Space*) fino al *Clip Space*, in modo che possano essere correttamente proiettate sul piano dello schermo.

Il Vertex Shader può generare informazioni aggiuntive come normali trasformate, coordinate di texture, colori, tangenti o valori definiti dal programmatore. Questi attributi vengono poi interpolati tra i vertici durante la rasterizzazione, fornendo dati continui al *Fragment Shader*.

Storicamente, il Vertex Shader veniva anche usato per calcolare l'illuminazione a livello di vertici (*per-vertex lighting*). La luce veniva determinata sulla base della

posizione e della normale di ciascun vertice, e il colore risultante veniva poi interpolato lungo le superfici. Sebbene molto efficiente, questo approccio produceva risultati approssimativi in presenza di geometrie complesse o variazioni luminose significative. Con l'introduzione dei *Fragment Shader*, i calcoli di illuminazione sono stati spostati a livello di frammento, ottenendo un realismo maggiore.

Oggi, il Vertex Shader non è più vincolato al solo calcolo dell'illuminazione, ma è diventato una vera e propria unità di calcolo flessibile. Può essere utilizzato, ad esempio, per la gestione delle animazioni scheletriche (*skeletal animation*), per la manipolazione procedurale della geometria (come la deformazione di superfici o l'applicazione di onde), o per la generazione di attributi personalizzati definiti dal programmatore.

1.2.2 Fragment Shader

Il *Fragment Shader*, chiamato anche *Pixel Shader*, è lo stadio programmabile che opera dopo la rasterizzazione, quando la geometria è stata convertita in frammenti (potenziali pixel). Per ogni frammento, il Fragment Shader riceve in input i dati interpolati dai vertici e calcola il colore finale e altri valori utili.

Le operazioni più comuni includono:

- **Texturing**: applicazione di immagini sulle superfici dei modelli per aumentarne il dettaglio visivo.
- **Illuminazione avanzata**: ad esempio tramite *shading Phong*, *Blinn-Phong* o il più moderno *Physically Based Rendering* (PBR), che simula in maniera realistica riflessioni, rifrazioni e interazioni con i materiali.
- **Effetti visivi**: trasparenza, occlusione ambientale, *bump mapping* (simula rilievi sulla superficie senza modificare la geometria), *normal mapping* (alterazione precisa delle normali per dettagli realistici) e la generazione di valori intermedi per il successivo post-processing.

Il risultato del Fragment Shader è un insieme di valori di colore (tipicamente componenti RGBA, composto dai tre colori principali e dal canale alpha) che vengono inviati al Frame Buffer. Grazie alla flessibilità del Fragment Shader, oggi è possibile ottenere effetti grafici estremamente realistici, che spaziano dalla resa fisica accurata dei materiali fino a stili visivi complessi e artistici.

1.2.3 OpenGL

OpenGL (*Open Graphics Library*) è una libreria standard open source e multipiattaforma utilizzata per sviluppare applicazioni grafiche interattive in tempo reale. Questa libreria mette a disposizione funzioni che permettono di sfruttare direttamente la potenza di calcolo delle GPU e consentono quindi di eseguire operazioni complesse come la gestione delle geometrie, la trasformazione degli oggetti e la resa dei materiali, con tempi compatibili con il real-time.

La pipeline grafica di OpenGL è organizzata in stadi, ciascuno dei quali elabora un tipo specifico di dati in sequenza. Con l'introduzione degli shader programmabili, alcune fasi predefinite possono essere sostituite da programmi personalizzati, permettendo di ottenere effetti visivi più avanzati e un controllo più preciso sul processo di rendering.

1.2.4 GLSL

Gli shader in OpenGL sono scritti in GLSL (*OpenGL Shading Language*), un linguaggio pensato per operazioni parallele su grandi quantità di dati grafici. La sua sintassi, simile al C, include estensioni per gestire vettori, matrici e texture, rendendolo particolarmente adatto alla grafica in tempo reale. Gli shader in GLSL vengono compilati ed eseguiti direttamente sulla GPU, garantendo un collegamento quasi diretto tra il codice e le unità di calcolo hardware.

1.2.5 GPU

Le capacità degli shader dipendono strettamente dall'architettura delle GPU, progettate per un'esecuzione altamente parallela. A differenza delle CPU, ottimizzate per gestire poche istruzioni complesse in sequenza, le GPU possono applicare lo stesso insieme di istruzioni a migliaia di dati contemporaneamente, secondo il modello SIMD (*Single Instruction, Multiple Data*). Questo tipo di architettura risulta particolarmente efficace nella grafica 3D, dove operazioni identiche devono essere applicate simultaneamente a migliaia di vertici.

Capitolo 2

Tessellation Shaders

In questo capitolo vengono analizzati i ***Tessellation Shaders***, un blocco opzionale delle Pipeline moderne, composto da più unità, che consente di suddividere dinamicamente le primitive e ottenere superfici più dettagliate rispetto alla Pipeline standard descritta nel capitolo 1.

2.1 Introduzione

I *Tessellation Shaders* costituiscono uno stadio opzionale, ma fondamentale, della pipeline grafica programmabile, introdotto a partire da OpenGL 4.0 e ampiamente utilizzati nelle versioni successive. Il loro scopo principale è quello di permettere la suddivisione dinamica di una primitiva in tasselli più piccoli e dettagliati, evitando così la necessità di definire manualmente tali dettagli a livello di CPU. Questa capacità di suddivisione adattiva consente di ottenere superfici più dettagliate e di controllare in modo dinamico il livello di dettaglio (LOD), ad esempio in funzione della distanza dalla telecamera virtuale o di altri criteri, migliorando l'efficienza computazionale e la qualità visiva.

I Tessellation Shaders si collocano nella pipeline grafica tra il Vertex Shader e il Geometry Shader (se presente) e si articolano in tre componenti distinti che operano in sequenza, ciascuno con un ruolo complementare nella trasformazione delle primitive. Il ***Tessellation Control Shader*** (TCS) gestisce i parametri di suddivisione delle geometrie originali, definendo come queste dovranno essere raffinate. A seguire, il ***Tessellation Primitive Generator*** (componente fisso, non modificabile dal programmatore), noto anche come *Tessellator*, realizza effettivamente la suddivisione delle geometrie originali in nuovi vertici secondo le indicazioni ricevute dal TCS. Infine, il ***Tessellation Evaluation Shader*** (TES) calcola le posizioni finali dei vertici generati e i relativi attributi, come normali e coordinate di texture, preparando così la geometria per gli stadi successivi della pipeline.

Questa struttura modulare consente di separare chiaramente la fase di controllo, quella di generazione e quella di valutazione, offrendo al programmatore la flessibilità

di intervenire solo dove necessario e lasciando alla GPU il compito di produrre una geometria densa e coerente in maniera efficiente [2], [6], [8], [9], [19].

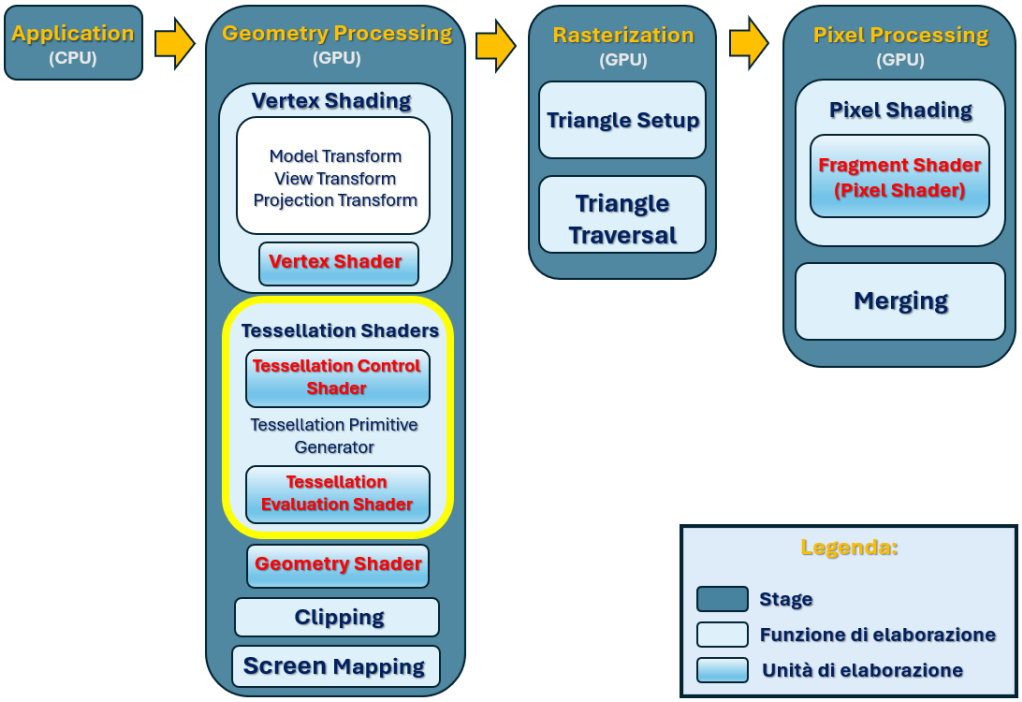

Figura 2.1: Schema della pipeline di rendering con i Tessellation Shaders.

2.2 Concetti Fondamentali

2.2.1 Tessellation

Con il termine *tessellation* (*tessellazione*) si intende il meccanismo che consente alla GPU di suddividere automaticamente una primitiva di partenza in elementi più piccoli e poi valutarne la forma con precisione, mantenendo la coerenza geometrica con quella originale. In pratica, la tessellazione serve ad aggiungere vertici ad un oggetto in modo controllato e coerente con la forma originale, consentendo di ottenere superfici più definite, curvature più fedeli e rilievi geometrici più accurati, senza dover cambiare il modello sorgente.

Il processo avviene in due fasi principali:

1. **Suddivisione della geometria originale**: la primitiva di input viene scomposta in sotto-primitive (triangoli, quadrilateri o *isolinee*) in base ai fattori di tessellazione specificati.

2. **Calcolo dei nuovi vertici**: per ogni nuovo vertice generato, la pipeline valuta posizione, normali, coordinate di texture e altri attributi interpolati. Questo permette di ottenere superfici molto più dettagliate, mantenendo però l'ingombro del modello originale.

L'aspetto essenziale della tessellation è la possibilità di generare dinamicamente, direttamente in GPU, un numero elevato di vertici interni alla primitiva originale, evitando di definirli manualmente in fase di modellazione. Questo permette di arricchire la geometria in maniera adattiva e controllata, fornendo una base più densa e precisa su cui applicare calcoli successivi. In altre parole, la tessellation consente di partire da primitive relativamente semplici e trasformarle, solo quando necessario, in rappresentazioni più dettagliate, mantenendo un controllo diretto sul livello di complessità geometrica.

Ad esempio, un terreno può essere raffinato dinamicamente nelle aree vicine alla telecamera per mostrare maggiori dettagli, riducendo invece la densità geometrica nelle zone lontane e meno visibili per risparmiare risorse computazionali. Analogamente, una superficie curva può essere approssimata in maniera sempre più fedele aumentando il livello di tessellazione, migliorando così la resa visiva senza dover aumentare a priori il numero di vertici del modello originale.

2.2.2 Patch

La tessellazione opera su primitive generalizzate chiamate *patch* e definite a livello di Application Stage. Una patch è un sottoinsieme arbitrario dei vertici che compongono la primitiva di partenza da suddividere durante lo stadio di tessellazione. A differenza di triangoli o linee tradizionali, il numero di vertici che costituiscono una patch non è fisso, ma deve essere esplicitamente dichiarato dall'applicazione tramite la funzione:

```
1 glPatchParameteri(GL_PATCH_VERTICES, n);
```

dove n indica quanti vertici compongono ciascuna patch.

L'input alla pipeline non è quindi più una semplice sequenza di primitive standard (punti, linee o triangoli), ma una sequenza di gruppi di n vertici, ciascuno considerato come una patch indipendente. Questi vertici verranno elaborati in modo coerente dallo stadio di tessellazione.

In questo modo, le patch forniscono una struttura flessibile e controllabile su cui la tessellazione può operare, permettendo di aumentare dinamicamente il numero di vertici interni e, di conseguenza, il dettaglio della superficie senza modificare il modello sorgente.

2.3 Tessellation Control Shader (TCS)

Il *Tessellation Control Shader* (TCS) è il primo stadio programmabile all'interno del blocco dei Tessellation Shaders e rappresenta il punto in cui il programmatore può intervenire direttamente sul livello di dettaglio della geometria. Questo shader riceve in ingresso le patch elaborate dagli stadi precedenti e ha il compito di definire come e quanto ciascuna patch dovrà essere suddivisa, determinando quindi i cosiddetti *livelli di tessellazione* (*Tessellation Levels*).

Il TCS viene eseguito una volta per ogni vertice della patch, ma ogni invocazione ha accesso alle informazioni di tutti i vertici che compongono l'intera patch. Questo significa che, pur lavorando in modalità *per-vertex* (un vertice alla volta), il TCS può prendere decisioni basate sull'intera patch, rendendo possibile ottenere effetti di dettaglio variabile all'interno della stessa.

Tra le principali operazioni svolte dal TCS vi è innanzitutto la gestione dei vertici della patch, applicando eventuali trasformazioni e preparando i dati per la fase di tessellazione vera e propria. In secondo luogo, lo shader calcola i parametri che definiranno i livelli di suddivisione da applicare, ad esempio in funzione della distanza dalla telecamera o della complessità della superficie. Infine, il TCS passa al *Tessellator* i vertici trasformati, insieme ai livelli di tessellazione calcolati, fornendo tutto il necessario affinché il componente fisso della pipeline possa generare la geometria aggiuntiva in modo coerente ed efficiente.

In sintesi, il Tessellation Control Shader funge da regista della suddivisione: riceve le primitive, decide come saranno suddivise e prepara i dati necessari affinché il Tessellator possa produrre nuovi vertici senza richiedere interventi manuali da parte del programmatore [4], [16].

2.3.1 Creazione e distruzione

La creazione di un Tessellation Control Shader in OpenGL segue un processo analogo a quello previsto per gli altri tipi di shader (come il Vertex Shader o il Fragment Shader). La differenza principale consiste nel tipo da specificare nella funzione `glCreateShader`, che in questo caso è `GL_TESS_CONTROL_SHADER`.

In generale, una volta creato l'oggetto shader, è necessario associargli il codice sorgente tramite `glShaderSource`, compilarlo con `glCompileShader`, e infine collegarlo a uno *shader program*, un programma eseguibile sulla GPU composto da più stadi della pipeline grafica. Per farlo, si utilizza la funzione `glAttachShader`, seguita da `glLinkProgram`, che completa il collegamento tra tutti gli shader presenti.

Al termine della fase di collegamento, è possibile eliminare lo shader con la funzione `glDeleteShader`, poiché il suo contenuto è stato già incorporato nel programma OpenGL. Questa operazione consente di liberare memoria senza compromettere il funzionamento del rendering.

È importante sottolineare che, per essere considerato valido, uno shader program deve includere almeno gli stadi minimi richiesti dal contesto di utilizzo. Nella maggior parte dei casi, ciò implica la presenza di un *Vertex Shader* e di un *Fragment Shader*. Tuttavia, in particolari casi d'uso, come nel *Transform Feedback*, lo shader program può omettere il *Fragment Shader*, poiché i dati vengono intercettati prima della rasterizzazione, direttamente all'uscita del *Vertex Shader* o del *Geometry Shader*.

Di seguito è riportato un esempio di codice C++ che mostra i passaggi fondamentali per caricare, compilare e collegare un Geometry Shader. Si noti che la funzione

`loadShaderSource`, utilizzata per leggere il contenuto del file sorgente, non è parte della libreria OpenGL, ma è stata definita dall'autore.

```
// Carica il contenuto del file shader
string source = loadShaderSource(path);

// Converte la stringa in formato C-style
const char* src = source.c_str();

// Crea lo shader specificato (es. GL_TESS_CONTROL_SHADER)
unsigned int shader = glCreateShader(type);

// Associa il codice sorgente allo shader
glShaderSource(shader, 1, &src, nullptr);

// Compila il codice sorgente nello shader
glCompileShader(shader);

// Crea un nuovo shader program
unsigned int program = glCreateProgram();

// Aggiunge lo shader compilato al programma
glAttachShader(program, geometry);

// Collega tutti gli shader nel programma finale
glLinkProgram(program);

// Elimina l'oggetto shader per liberare memoria
glDeleteShader(geometry);
```

2.3.2 Input

Il Tessellation Control Shader riceve in ingresso le patch definite dalla chiamata a `glPatchParameteri(GL_PATCH_VERTICES, n)` (come descritto nella sezione 2.2.2), la quale stabilisce il numero di vertici che compongono ciascuna patch.

È importante sottolineare che il TCS non interpreta la natura geometrica della patch (che sia linea, triangolo o quadrilatero), ma lavora in modo astratto su insiemi di vertici. Il suo compito è elaborare tali vertici e stabilire i livelli di tessellazione da fornire al Tessellator, senza la necessità di dover conoscere a priori la tipologia della primitiva.

L'esecuzione del TCS avviene una volta per ciascun vertice della patch. Tuttavia, ogni invocazione può accedere ai dati di tutti i vertici della patch, grazie a variabili built-in messe a disposizione dal linguaggio OpenGL. La prima di queste è `gl_InvocationID`, che rappresenta l'indice dell'invocazione corrente, ovvero il vertice della patch a cui fa riferimento la specifica esecuzione dello shader. Essa permette di distinguere i diversi vertici e gestirne i dati in maniera ordinata.

Un'altra variabile fondamentale è `gl_PatchVerticesIn`, che contiene il numero totale dei vertici della patch in ingresso. Questa informazione è indispensabile per gestire correttamente i dati, ad esempio quando è necessario iterare su tutti i vertici per calcolare valori globali (come i livelli di tessellazione basati sulla posizione media o sulla distanza minima dei vertici dalla telecamera).

Di particolare importanza è anche l'array `gl_in[]`, che contiene, per ciascun vertice della primitiva in ingresso, un insieme di informazioni geometriche di base. Ogni elemento dell'array è di tipo `gl_PerVertex`, una struttura definita dal linguaggio GLSL che contiene al suo interno variabili built-in comuni a più stadi della pipeline. Il numero totale di elementi dell'array dipende direttamente dal layout di input dichiarato nello shader.

La struttura `gl_PerVertex` è organizzata come segue:

```
in gl_PerVertex {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
    float gl_CullDistance[];
} gl_in[];
```

I campi principali sono:

- `gl_Position`: contiene la posizione del vertice nello spazio clip (coordinate omogenee), così come calcolata dal Vertex Shader. È la variabile principale utilizzata per calcolare medie, distanze o altri parametri necessari alla definizione dei livelli di tessellazione.
- `gl_PointSize`: rappresenta la dimensione del punto, nel caso in cui la primitiva in ingresso fosse un singolo punto. Pur essendo raramente utilizzata in un contesto di tessellazione, è comunque parte della struttura e può essere propagata per scopi specifici.
- `gl_ClipDistance[]`: array che memorizza le distanze dai piani di clipping definiti dall'applicazione. Può essere sfruttato nel TCS per prendere decisioni in base alla visibilità parziale della patch, ad esempio evitando di tessellare ulteriormente porzioni che verrebbero scartate.
- `gl_CullDistance[]`: concettualmente simile a `gl_ClipDistance[]`, ma pensata per operazioni di culling personalizzato, cioè l'eliminazione condizionale di primitive non necessarie. Questa funzionalità permette di scartare anticipatamente geometrie irrilevanti, migliorando l'efficienza della pipeline di rendering.

L'accesso a `gl_in[]` rende possibile basare i calcoli non sul singolo vertice, ma sull'intera patch. In questo modo il TCS può stabilire livelli di tessellazione coerenti con proprietà globali, come la posizione media dei vertici o la distanza minima dalla camera. L'indicizzazione avviene tramite l'indice del vertice all'interno della patch, consentendo accesso diretto a qualsiasi dato dei vertici.

Infine, va ricordato che, oltre alle variabili built-in, il TCS riceve come input anche tutte le variabili di output del Vertex Shader, organizzate in array e indicizzate rispetto al numero di vertici della patch.

2.3.3 Output

Se l'input del TCS è principalmente costituito dai dati dei vertici e dalle variabili built-in che descrivono lo stato corrente della patch, l'output è invece ciò che definisce il comportamento della tessellazione vera e propria.

Ogni invocazione del TCS può produrre nuovi dati associati ai vertici, che saranno poi accessibili al *Tessellation Evaluation Shader*. Questi dati vengono dichiarati come variabili di output dello shader e, come accade per il Vertex Shader, possono essere strutturate in array in modo che ogni invocazione scriva i valori relativi al proprio vertice. In questo modo, ad esempio, si possono passare coordinate di texture, normali o qualsiasi altro attributo necessario per la successiva elaborazione.

A tale scopo il linguaggio GLSL mette a disposizione l'array `gl_out[]`, il cui numero di elementi corrisponde al numero di vertici specificato dalla direttiva di layout. Ogni elemento dell'array è di tipo `gl_PerVertex`, la stessa struttura introdotta nella sezione 2.3.2 a proposito dell'array `gl_in[]`, e contiene quindi le stesse variabili built-in comuni a più stadi della pipeline. Lo schema è il seguente:

```
out gl_PerVertex {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
    float gl_CullDistance[];
} gl_out[];
```

In questo modo ogni invocazione del TCS può scrivere, all'interno della propria cella di `gl_out[gl_InvocationID]`, i valori che saranno letti successivamente dal TES attraverso il corrispondente array `gl_in[]`.

Ma l'aspetto più caratteristico e rilevante dell'output del TCS riguarda la definizione dei livelli di tessellazione. OpenGL mette infatti a disposizione delle variabili built-in speciali che devono essere scritte dal TCS:

- `gl_TessLevelOuter[4]`: un array di quattro valori in virgola mobile che definiscono i **livelli di tessellazione** per i lati **esterni** della patch. Il numero di elementi effettivamente utilizzati dipende dal tipo di primitiva (vedi la sezione 2.4).
- `gl_TessLevelInner[2]`: un array che definisce i **livelli di tessellazione interna**, ovvero quanti suddivisori aggiuntivi verranno inseriti all'interno della patch. Anche in questo caso, il numero di componenti effettivamente utilizzate dipende dalla tipologia di primitiva.

Queste variabili rappresentano l'elemento chiave con cui il programmatore stabilisce la densità della geometria che verrà generata dal Tessellator. Valori più alti produ-

cono un maggior numero di sotto-primitive geometriche e quindi una superficie più dettagliata, mentre valori più bassi portano a una geometria più semplice e meno costosa da elaborare. È proprio attraverso queste variabili che il TCS abilita il cosiddetto controllo adattivo del livello di dettaglio, rendendo possibile, ad esempio, incrementare la suddivisione per le porzioni di superficie vicine alla telecamera e ridurla per quelle più lontane.

L'output del TCS è inoltre governato dalla direttiva

```
1 {layout (vertices = n) out;
```

dove il valore n specifica il numero di vertici che la patch dovrà avere in uscita. Tale valore non deve necessariamente coincidere con il numero di vertici in ingresso: è infatti possibile generare un numero differente di vertici, permettendo così al programmatore di rimodellare la patch e prepararla in modo mirato per la fase successiva di tessellazione e valutazione.

2.4 Tessellation Primitive Generator (Tessellator)

Il *Tessellation Primitive Generator*, comunemente chiamato *Tessellator*, rappresenta l'elemento del blocco dei Tessellation Shaders che si colloca tra il Tessellation Control Shader e il Tessellation Evaluation Shader. A differenza dei due shader, non si tratta di uno stadio programmabile, ma di una fixed-function: il suo comportamento è determinato unicamente dai parametri calcolati e scritti dal TCS e dalle direttive di ingresso dichiarate dal TES.

Il compito del Tessellator è quello di suddividere le patch ricevute dal TCS in un insieme di primitive geometriche elementari. La suddivisione generata dipende sia dai valori dei livelli di tessellazione impostati dal TCS, che stabiliscono il grado di suddivisione della patch, sia dalle informazioni contenute nel layout di input del TES, il quale definisce il tipo di primitiva, le modalità di distribuzione dei punti (*spacing*) e il tipo di orientamento. La combinazione di questi elementi determina non solo la densità della griglia generata, ma anche la sua regolarità e coerenza, garantendo che la superficie tessellata risulti continua e priva di disallineamenti.

Un aspetto fondamentale da sottolineare è che il Tessellator non interpreta la geometria originaria della patch: la sua funzione è esclusivamente quella di produrre una griglia di campionamento. L'output è costituito da un insieme di *coordinate parametriche normalizzate*, generate nello spazio del dominio della primitiva. Queste non sono coordinate spaziali, ma valori normalizzati che identificano la posizione relativa di un punto all'interno della patch tessellata. Tali coordinate vengono restituite come output per il Tessellation Evaluation Shader tramite la variabile built-in `gl_TessCoord` (vedi sezione 2.5). Sarà poi il TES, utilizzando tali coordinate insieme ai dati della patch provenienti dal TCS, a determinare la posizione geometrica effettiva di ciascun punto nello spazio tridimensionale.

2.4.1 Tipi di primitive

Il Tessellation Evaluation Shader supporta diverse tipologie di primitive, le quali influenzano direttamente il comportamento del Tessellator nella generazione dei vertici tessellati. Le principali tipologie sono:

- **Triangles** (`triangles`): la patch originale elaborata dal Tessellator è triangolare e viene suddivisa in triangoli più piccoli. La suddivisione segue le regole definite dai livelli di tessellazione esterni e interni, determinando sia il numero di segmenti lungo i lati del triangolo sia la densità dei triangoli interni concentrici.
- **Quads** (`quads`): la patch originale elaborata dal Tessellator è quadrangolare e viene suddivisa in un reticolo regolare di triangoli. I valori dei livelli di tessellazione esterni controllano la densità dei segmenti lungo le due direzioni principali della patch, mentre eventuali livelli interni possono definire ulteriori suddivisioni interne.
- **Isolines** (`isolines`): la patch originale elaborata dal Tessellator può avere un numero variabile di vertici, anche se più comunemente lavora con patch da 4 vertici. In questo caso, però, la tessellazione genera linee parallele (*isolinee*) all'interno della patch. I livelli di tessellazione controllano il numero di linee create e la loro densità all'interno della patch.

2.4.2 Modalità di spaziatura

La distribuzione dei punti lungo i lati della patch (*modalità di spaziatura*) può essere regolata tramite il parametro di spacing, definito nel layout del TES. Questo parametro influenza però anche la posizione dei vertici generati dal Tessellator lungo ciascun lato della patch, influenzando quindi sia la densità sia la regolarità della griglia tessellata. Le principali modalità di spaziatura sono:

- **Equal spacing** (`equal_spacing`): in questa configurazione, il Tessellator genera una suddivisione uniforme dei segmenti. Il valore in virgola mobile del livello di tessellazione viene prima di tutto normalizzato all'intervallo $[1, max]$ (dove max è il valore massimo supportato dall'implementazione). Il valore risultante è successivamente arrotondato per eccesso all' intero n più vicino. Il lato corrispondente della patch viene quindi suddiviso in n segmenti di uguale lunghezza, assicurando una distribuzione regolare dei punti lungo tutto il bordo.
- **Fractional even spacing** (`fractional_even_spacing`): questa modalità genera sempre un **numero pari di segmenti**, consentendo una suddivisione frazionaria. Il valore del livello di tessellazione viene normalizzato nell'intervallo $[2, max]$ e arrotondato per eccesso all'intero pari più vicino, indicato con n. Il lato della patch viene suddiviso in n segmenti, di cui gli $n-2$ segmenti centrali hanno lunghezza identica, mentre i due segmenti agli estremi possono avere lunghezza diversa, regolata dalla parte frazionaria del livello di tessellazione originale. Quando il valore frazionario è vicino a 0, i segmenti agli estremi

hanno lunghezza simile a quella dei segmenti centrali; all'aumentare della parte frazionaria, la lunghezza dei segmenti estremi diminuisce progressivamente, mantenendo sempre una disposizione simmetrica rispetto ai due estremi del lato.

- **Fractional odd spacing** (`fractional_odd_spacing`): in questa modalità il Tessellator genera sempre un **numero dispari di segmenti**, mantenendo un comportamento frazionario. Il livello di tessellazione viene limitato all'intervallo $[1, max - 1]$ e poi arrotondato per eccesso all'intero dispari più vicino, indicato con n. Se $n = 1$, non viene effettuata alcuna suddivisione, in caso contrario, il lato viene suddiviso in n segmenti, di cui gli $n - 2$ centrali hanno lunghezza uguale, mentre i due segmenti agli estremi hanno lunghezza leggermente differente, regolata dalla parte frazionaria del livello di tessellazione originale, allo stesso modo del fractional even spacing.

2.4.3 Orientamento

Tra i parametri definiti nel TES ma che influenzano i calcoli del Tessellator, è importante menzionare anche l'*orientamento* dei triangoli generati, ovvero l'ordine in cui i vertici vengono interpretati. Questo parametro è fondamentale per garantire la corretta determinazione delle normali e la coerenza geometrica della patch tessellata. Le opzioni disponibili nel TES sono:

- `cw`: i vertici di tutti i triangoli generati sono analizzati in senso orario (*clockwise*).
- `ccw`: i vertici di tutti i triangoli generati sono analizzati in senso antiorario (*counter-clockwise*).

2.4.4 Primitive Tessellation

Prima di analizzare come vengono tessellate le diverse tipologie di patch all'interno del Tessellator, è importante soffermarsi sul significato dei valori dei livelli di tessellazione definiti dal TCS. Questi valori sono specificati in virgola mobile, ma il Tessellator non li utilizza come numeri reali continui. Essi vengono approssimati a numeri interi, in modo da determinare il numero effettivo di segmenti da generare. La parte decimale, tuttavia, non è del tutto ignorata, ma viene trattata diversamente in base alla modalità di spacing definita (vedi sezione 2.4.2) [15].

Triangle Tessellation

Quando il tipo di primitiva specificato dal TES è impostato su `triangles`, una patch triangolare viene suddivisa in un insieme di triangoli più piccoli che coprono interamente l'area originale.

In questa modalità, il processo di suddivisione dipende unicamente dai primi tre valori della variabile built-in `gl_TessLevelOuter` (cioè gli elementi con indici nell'intervallo $[0, 2]$) e dal primo valore di `gl_TessLevelInner` (cioè l'elemento con

indice 0). Il primo valore interno determina il numero di triangoli concentrici generati all'interno della patch: più alto è il valore, maggiore sarà il numero di triangoli interni e più fine sarà la tessellazione. I tre valori esterni definiscono invece il numero di segmenti lungo ciascun lato del triangolo originale, creando vertici intermedi che saranno collegati ai punti generati dai livelli interni per formare triangoli più piccoli. I valori di tessellazione vengono applicati ai lati della patch nell'ordine stabilito dalla figura seguente.

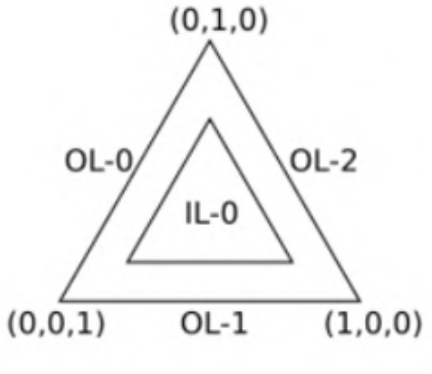

Figura 2.2: Schema di corrispondenza dei valori di tessellazione ai lati della patch triangolare.

In base al valore di questi livelli, possono presentarsi alcuni casi limite: se il livello interno e tutti e tre i livelli esterni sono impostati a 1, il triangolo originale rimane invariato. Se il livello interno è 1, ma almeno uno dei livelli esterni è maggiore di 1, il valore interno viene considerato leggermente superiore ad 1 $(1 + \epsilon)$, in modo da garantire la generazione di almeno un vertice interno e prevenire geometrie degenerate.

Quando almeno un livello è maggiore di 1, il Tessellator passa alla generazione vera e propria dei triangoli interni, calcolando progressivamente le posizioni dei vertici concentrici a partire dai lati della patch originale.

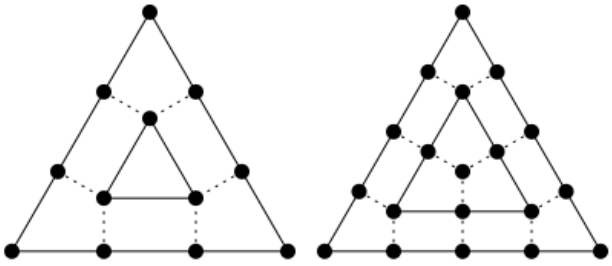

Figura 2.3: Schema di tessellazione di una primitiva `triangles`, con lati esterni suddivisi in base al livello di tessellazione interna: (3) nel primo caso e (4) nel secondo.

In questa fase, il triangolo esterno funge da riferimento per determinare le suddivisioni ricorsive successive: i suoi lati vengono **temporaneamente** suddivisi in segmenti in base al livello di tessellazione interna (arrotondato) e allo spacing selezionato, generando n segmenti. A seconda del valore di n, la ricorsione termina con

un triangolo interno degenere, cioè ridotto a un singolo punto, se $n = 2$, oppure il triangolo rappresentante l'ultimo della serie se $n = 3$.

In caso di $n > 3$, ciascun vertice del triangolo esterno determina la posizione del corrispondente vertice del triangolo interno successivo della ricorsione. La posizione di questo vertice interno non è arbitraria, ma calcolata considerando i segmenti già generati lungo i lati del triangolo esterno preso a riferimento. Per ciascun vertice esterno si tracciano due linee immaginarie, ciascuna perpendicolare a uno dei due estremi dei segmenti adiacenti al vertice stesso e l'intersezione di queste due linee definisce il vertice interno, garantendo proporzioni regolari e simmetria rispetto al triangolo esterno.

Una volta stabiliti i tre vertici iniziali del triangolo interno, ciascun lato del triangolo viene a sua volta suddiviso in $n - 2$ segmenti. I vertici generati lungo i lati interni si calcolano come intersezione tra linee perpendicolari ai segmenti del triangolo esterno (passanti per i vertici ottenuti dalla sua suddivisione iniziale) e i lati del triangolo interno appena formato. Questo assicura che ogni nuovo vertice interno mantenga connessioni coerenti con i vertici già generati nei livelli precedenti.

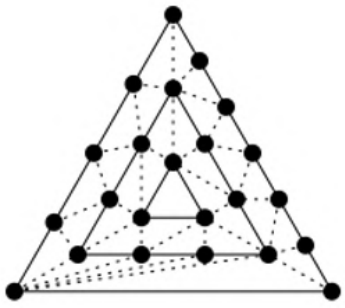

Figura 2.4: Esempio di geometria finale ottenuta dalla tessellazione di una primitiva `triangles`, con valori $(4, 1, 6)$ per la tessellazione esterna e (5) per quella interna.

A questo punto, si sostituiscono le suddivisioni temporanee dei lati esterni della patch con quelle definite dai parametri *outer* e si comincia un secondo processo ricorsivo, che parte dal collegare tra loro tutti i punti dei lati della patch con quelli del primo triangolo interno. In questo modo si formano tanti triangoli più piccoli, ciascuno dei quali condivide due vertici con un triangolo adiacente, mentre il terzo vertice coincide con uno dei vertici più vicini della patch esterna, garantendo così una griglia continua di triangoli non sovrapposti.

Dopo aver completato questa prima suddivisione, il triangolo interno diventa il nuovo triangolo esterno di riferimento per i calcoli successivi. L'intero processo viene ripetuto ricorsivamente, generando triangoli concentrici che si avvicinano progressivamente al centro della patch, fino a riempire completamente l'area originale. Nel caso in cui il triangolo più interno risulti degenere, cioè ridotto a un punto centrale, il triangolo che lo racchiude viene suddiviso in triangoli che convergono tutti verso tale punto, mantenendo una connessione coerente con i triangoli precedenti.

La specifica garantisce la presenza di connessioni tra vertici corrispondenti dei triangoli concentrici e tra i vertici sui bordi; tutte le altre connessioni sono lasciate alla discrezione dell'implementazione, che può ad esempio preferire di collegare i vertici adiacenti più vicini per ottenere una tessellazione regolare e continua su tutta la patch.

Di seguito viene mostrato uno pseudocodice che riassume le fasi appena descritte, evidenziando il flusso delle operazioni e semplificando la comprensione del processo di tessellazione dei triangoli.

```
1) Calcola il numero di segmenti esterni per ciascun lato della patch
   usando gl_TessLevelOuter

2) SE livello interno = 1 e tutti i livelli esterni = 1:
        Restituisci la patch senza modifiche
   ALTRIMENTI se livello interno = 1 e almeno un livello esterno > 1:
        Imposta livello interno = 1 + epsilon

3) Suddividi i lati esterni della patch in base al livello interno e allo
   spacing selezionato

4) PER OGNI nuovo triangolo esterno di riferimento:
        a. Determina i vertici del triangolo interno come intersezione di
           linee perpendicolari dai segmenti dei lati esterni
        b. Suddividi i lati del triangolo interno in n-2 segmenti
        c. SE numero di vertici del triangolo interno = 2:
                Triangolo interno degenere in un punto
                Passa a operazione 6)
           ALTRIMENTI se numero di vertici del triangolo interno = 3:
                Ultimo triangolo interno
                Passa a operazione 6)
           ALTRIMENTI:
                Imposta triangolo interno come nuovo triangolo esterno di
                riferimento

5) Suddividi i lati esterni della patch in base ai livelli esterni e allo
   spacing selezionato

6) PER OGNI triangolo esterno di riferimento:
        a. SE triangolo interno degenere in un punto:
                connettere tutti i vertici del triangolo esterno al punto
                centrale
           ALTRIMENTI:
                connetti ogni vertice del triangolo esterno con almeno un
                vertice del triangolo interno, senza generare triangoli
                sovrapposti
```

Quad Tessellation

Quando il tipo di primitiva specificato dal TES è impostato su `quads`, una patch quadrilaterale viene suddivisa in una griglia regolare di triangoli più piccoli che copre interamente l'area originale.

In questa modalità, il processo di suddivisione dipende dai due valori della variabile built-in `gl_TessLevelInner` (uno per ciascuna coppia di lati opposti) e dai quattro valori di `gl_TessLevelOuter` (uno per ciascun lato del quadrilatero). I valori interni determinano il numero di divisioni lungo le direzioni principali della patch, definendo così la griglia interna di vertici, mentre i valori esterni definiscono il numero di segmenti lungo ciascun lato del quadrilatero originale, creando vertici intermedi che saranno poi collegati ai punti generati dalla griglia interna per formare triangoli più piccoli. I valori di tessellazione vengono applicati ai lati della patch nell'ordine stabilito dalla figura seguente.

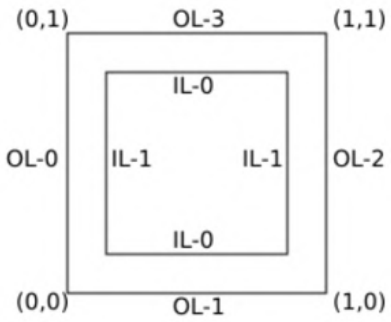

Figura 2.5: Schema di corrispondenza dei valori di tessellazione ai lati della patch quadrilaterale.

In base ai valori di questi livelli, possono presentarsi casi limite: se tutti i livelli interni ed esterni sono impostati a 1, il quadrilatero viene suddiviso in una sola coppia di triangoli che ricopre l'intera area della patch. Se uno dei due livelli interni è esattamente 1, viene considerato leggermente superiore a 1 $(1 + \epsilon)$ per evitare geometrie degenerate e garantire la presenza di almeno un vertice interno; a seconda dello spacing selezionato, questo può generare vertici molto vicini ai bordi, con possibile distorsione della griglia interna.

Quando almeno un livello di tessellazione è maggiore di 1, il Tessellator genera effettivamente la griglia interna dei vertici.

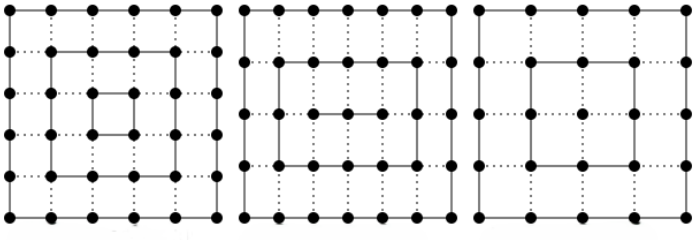

Figura 2.6: Schema di tessellazione di una primitiva `quads`, con lati esterni suddivisi in base al livello di tessellazione interna: $(5, 5)$ nel primo caso, $(6, 4)$ nel secondo e $(4, 4)$ nel terzo.

I lati del quadrilatero vengono suddivisi in segmenti in base ai livelli interni, producendo m suddivisioni lungo una direzione e n lungo l'altra. I punti di ciascun lato vengono connessi ai punti corrispondenti sul lato opposto, formando così una griglia regolare di celle quadrilaterali. Se $m = 2$ o $n = 2$, le celle più interne risultano degeneri, riducendosi a linee o punti.

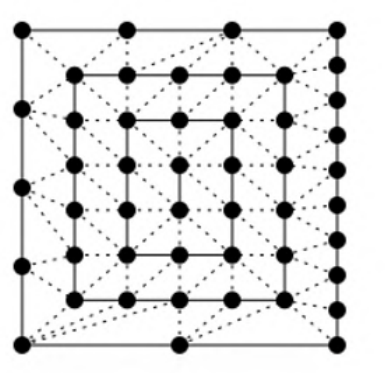

Figura 2.7: Esempio di geometria finale ottenuta dalla tessellazione di una primitiva `quads`, con valori $(4, 2, 9, 3)$ per la tessellazione esterna e $(6, 7)$ per quella interna.

Nella seconda fase si realizzano le suddivisioni dei lati della patch esterna secondo i valori *outer* e si procede quindi a suddividere ciascuna cella interna della griglia in coppie di triangoli. I vertici generati vengono poi collegati tra loro per creare triangoli non sovrapposti, garantendo che ogni triangolo condivida due vertici con i triangoli adiacenti e il terzo vertice con un vertice della griglia interna o della patch esterna, ottenendo una connessione coerente su tutta la superficie.

La specifica garantisce la presenza di connessioni tra vertici corrispondenti sulla griglia interna e lungo i lati esterni; tutte le altre connessioni sono legate all'implementazione e possono, ad esempio, essere scelte collegando i vertici adiacenti più vicini per ridurre distorsioni, ottenendo così una tessellazione regolare e continua su tutta la patch.

Nel caso in cui una delle dimensioni della griglia interna fosse degenere (ad esempio se $m = 2$ o $n = 2$), i triangoli generati si adattano alla forma ridotta della cella interna, riducendosi a linee o triangoli convergenti verso un punto centrale, senza interrompere la continuità della mesh complessiva. In questo modo, il Tessellator garantisce una tessellazione regolare, continua e coerente con i livelli di tessellazione specificati su tutta la patch quadrilaterale.

Di seguito viene mostrato uno pseudocodice che riassume le fasi appena descritte, evidenziando il flusso delle operazioni e semplificando la comprensione del processo di tessellazione dei quadrilateri.

```
1) Calcola il numero di segmenti esterni per ciascun lato della patch
   usando gl_TessLevelOuter

2) SE tutti i livelli interni = 1 e tutti i livelli esterni = 1:
       Restituisci patch suddivisa in due triangoli non sovrapposti
```

```
   ALTRIMENTI se uno dei livelli interni = 1:
        Imposta livello interno = 1 + epsilon

3) Suddividi i lati esterni della patch in base ai livelli interni e allo
    spacing selezionato

4) Genera griglia ottenuta congiungendo i punti di suddivisione del lato
    opposto

5) Definisci quadrilateri concentrici interni sulla griglia

6) SE in entrambe le dimensioni il numero di celle interne ad un
    quadrilatero interno = 2:
        Imposta quadrilatero interno come degenere in un punto
   ALTRIMENTI se numero di celle = 2 in una sola dimensione:
        Imposta quadrilatero interno come degenere in una linea

6) Partendo dal quadrilatero relativo alla patch originale, PER OGNI
    quadrilatero esterno:
        a. SE quadrilatero interno degenere in un punto:
                Connetti tutti i vertici del quadrilatero esterno col
                vertice interno
           ALTRIMENTI:
                Connetti ogni vertice del quadrilatero esterno con almeno
                un vertice del quadrilatero concentrico interno, definendo
                triangoli non sovrapposti
```

Isoline Tessellation

Quando il tipo di primitiva specificato dal TES è impostato su `isolines`, una patch quadrilaterale viene suddivisa in una serie di linee indipendenti, chiamate *isolines*, ciascuna formata da segmenti lineari.

In questa modalità, il processo di suddivisione dipende unicamente dai primi due valori della variabile built-in `gl_TessLevelOuter`. Il primo valore determina il numero totale di isolines generate, mentre il secondo specifica il numero di segmenti in cui ciascuna isoline viene suddivisa.

Il processo inizia considerando i due lati verticali del quadrilatero originale determinati in base all'ordine sequenziale dei suoi vertici, che vengono suddivisi in punti equispaziati in base al primo livello di tassellazione esterna. Ciascun punto su un lato verticale, ad esclusione del vertice più alto, viene quindi collegato al punto corrispondente sul lato opposto, generando le linee orizzontali che costituiscono le isolines. Questo posizionamento garantisce che le linee siano uniformemente distribuite, mantenendo regolarità e proporzioni coerenti con la patch originale.

Successivamente, ogni isoline viene suddivisa in segmenti lineari in base al secondo valore di tessellazione esterna. In pratica, i punti intermedi lungo la linea orizzontale sono generati in modo uniforme tra i due vertici estremi, creando segmenti lineari

connessi fra loro senza sovrapposizioni. La suddivisione garantisce che ogni isoline abbia lo stesso numero di segmenti, creando una griglia ordinata di linee parallele che coprono interamente la patch.

In base ai valori di tessellazione, possono presentarsi alcuni casi limite: se entrambi i livelli esterni sono impostati a 1, viene generata una singola isoline composta da un solo segmento. Se il primo livello esterno è 1 e il secondo è maggiore di 1, la singola isoline viene suddivisa in segmenti multipli, con punti equispaziati lungo la linea. In tutti i casi, i segmenti sono collegati in modo da garantire un insieme continuo di linee, senza gap né sovrapposizioni.

È importante notare che nonostante il dominio iniziale sia quasi sempre rappresentato da una patch quadrilaterale, la modalità isolines può essere utilizzata anche con patch costituite da un numero diverso di vertici. Un caso frequente è quello con due soli vertici, che definiscono un semplice segmento di riferimento. In questo scenario, il Tessellator genera più isolines del segmento (il cui numero è determinato dal primo livello di tessellazione esterna) utilizzando gli stessi metodi di calcolo di un generico quadrilatero. In base all'orientamento di valutazione dei vertici (`cw` o `ccw`) definito nel TES, i due vertici del segmento sono mappati come lato verticale o orizzontale di un quadrilatero degenere e tassellati di conseguenza. A livello di TES, si otterrà una serie di isoline segmentate, parallele e coincidenti col segmento (in caso di segmento considerato lato orizzontale), oppure una serie di isoline collassate nel loro punto di origine lungo il segmento e distanziate secondo il parallelismo previsto (in caso di segmento considerato lato verticale).

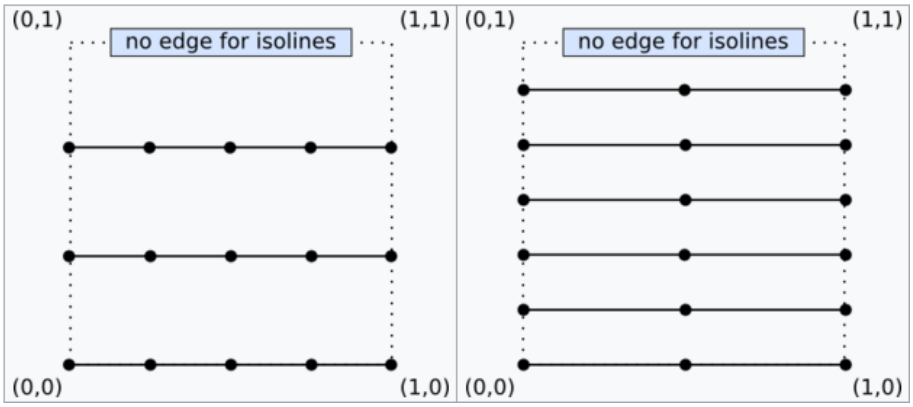

Figura 2.8: Esempio di geometria finale ottenuta dalla tessellazione di due primitive `isolines`, con valori di tessellazione: $(3, 4)$ per la prima e $(6, 2)$ per la seconda.

Di seguito viene mostrato uno pseudocodice che riassume le fasi appena descritte, evidenziando il flusso delle operazioni e semplificando la comprensione del processo di tessellazione delle isoline.

```
1) Determina i vertici corrispondenti alle suddivisioni esterne dei lati
   verticali della patch usando gl_TessLevelOuter corrispondente e lo
   spacing selezionato
```

```
2) Partendo dal vertice successivo a quello in alto, PER OGNI punto di
    suddivisione:
        Connetti il punto con quello corrispondente sul lato opposto

3) PER OGNI connessione:
        Suddividi la linea in segmenti usando gl_TessLevelOuter
        corrispondente

4) Gestisci i casi limite:
        a. SE entrambi i livelli esterni = 1:
                Genera una singola isoline con un solo segmento
        b. SE primo livello esterno = 1 e secondo > 1:
                Genera una singola isoline suddivisa in piu segmenti
        c. SE la patch ha solo due vertici:
                Tratta il segmento come patch degenere e riapplica il
                processo da capo
```

2.5 Tessellation Evaluation Shader (TES)

Il *Tessellation Evaluation Shader* rappresenta l'ultimo stadio programmabile all'interno del blocco dei Tessellation Shaders, situato subito dopo il Tessellation Primitive Generator. Il TES viene eseguito una volta per ciascun vertice generato dalla tessellazione e ha il compito di determinare le coordinate spaziali finali dei punti intermedi della superficie.

Il TES utilizza le informazioni calcolate dal TCS e le coordinate parametriche fornite dal Tessellator per interpolare in modo accurato la geometria della patch. In questa fase è possibile applicare trasformazioni geometriche di vario tipo, tra cui la semplice interpolazione dei vertici originali o il *displacement mapping*. Quest'ultima è una tecnica che sfrutta una mappa di valori (ad esempio una scala di grigi), per stabilire di quanto devono essere spostati i vertici lungo la direzione della normale, introducendo variazioni realistiche nella geometria.

Mentre il TCS definisce quanto e come suddividere una patch e il Tessellator genera i punti nel dominio parametrico, il TES determina dove, all'interno dello spazio 3D, questi punti devono essere collocati, preparandoli per gli stadi successivi della pipeline di rendering [4].

2.5.1 Creazione e distruzione

La procedura di creazione e distruzione di un Tessellation Evaluation Shader segue gli stessi passaggi descritti nella sezione 2.3.1. L'unica differenza rispetto agli altri tipi di shader riguarda il parametro passato alla funzione `glCreateShader`, che in questo caso deve essere `GL_TESS_EVALUATION_SHADER`. Questo valore identifica lo shader come Tessellation Evaluation, consentendo così a OpenGL di compilarlo e gestirlo correttamente.

2.5.2 Input

Il Tessellation Evaluation Shader riceve le patch generate dal Tessellator secondo un layout di input che ne definisce il tipo di primitiva, la modalità di spaziatura dei vertici e l'orientamento dei triangoli. Tale layout viene specificato direttamente all'inizio del codice dello shader tramite la sintassi:

```
layout(primitive, spacing, orientation) in;
```

In questa funzione, al posto di `primitive` si può inserire un valore fra `triangles`, `quads` e `isolines`. Al posto di `spacing`, un valore fra `equal_spacing`, `fractiona l_even_spacing` o `fractional_odd_spacing` e infine, al posto di `orientation`, un valore fra `cw` e `ccw`. I dettagli sul significato e l'effetto di questi valori sono spiegati nella sezione 2.4).

Le informazioni specifiche definite dal layout vengono poi rese disponibili nel TES tramite un insieme di variabili built-in, che contengono tutti i dati necessari per calcolare la posizione finale dei vertici tessellati. Tra queste:

- `gl_in[]`: se il TCS è attivo, contiene i valori restituiti da esso, altrimenti quelli restituiti dal Vertex Shader. Come già visto nella sezione 2.3.2, ogni elemento dell'array rappresenta un singolo vertice della patch di input ed è una struttura che presenta i seguenti campi: `gl_Position`, `gl_PointSize`, `gl_ClipDistance` e `gl_CullDistance`.
- `gl_PatchVerticesIn`: indica il numero di vertici appartenenti alla patch corrente.
- `gl_PrimitiveID`: contiene l'identificatore univoco della patch corrente, utile per distinguere le primitive in operazioni successive.
- `gl_TessCoord`: vettore a tre componenti che descrive la posizione del vertice generato nel dominio parametrico della patch. Per le patch triangolari, le coordinate assumono la forma baricentrica (u, v, w), con $u + v + w = 1$, in modo che ciascuna componente rappresenti l'influenza di un vertice originale sulla posizione finale. Per i quadrilateri e per le isolines, sono significative solo le componenti (u, v), che indicano la posizione relativa lungo le direzioni orizzontale e verticale della patch, mentre la terza componente viene impostata a 0. Tutti i valori sono normalizzati nell'intervallo $[0, 1]$.
- `gl_TessLevelOuter`: array contenente i livelli di tessellazione esterni della patch, che determinano la densità dei vertici lungo i bordi.
- `gl_TessLevelInner`: array contenente i livelli di tessellazione interni della patch, che regolano la distribuzione dei vertici all'interno della patch stessa.

2.5.3 Output

Il Tessellation Evaluation Shader, oltre a calcolare la posizione dei vertici generati dalla tessellazione, può produrre variabili di output da inviare allo stadio successivo

della pipeline, proprio come avviene negli altri shader programmabili.

Oltre alle variabili personalizzate definite dallo sviluppatore, il TES può gestire alcune variabili built-in, tra cui `gl_Position`, che rappresenta la posizione del vertice nello spazio clip, `gl_PointSize`, che definisce la dimensione di eventuali primitive puntiformi, e gli array `gl_ClipDistance[]` e `gl_CullDistance[]`, utilizzati rispettivamente per operazioni di clipping e di culling personalizzato (per maggiori dettagli consultare la sezione 2.3.2).

Queste variabili possono essere modificate dal TES e, insieme a eventuali variabili definite dall'utente, costituiscono l'output dello shader che verrà letto dagli stadi successivi della pipeline. Va sottolineato che, a differenza del Tessellation Control Shader, il TES non possiede un array `gl_out[]`. Ogni invocazione produce direttamente il vertice finale e le variabili di output corrispondenti.

2.6 Query e misurazioni

OpenGL fornisce un meccanismo avanzato di query statistiche che permette di raccogliere informazioni dettagliate sull'esecuzione della pipeline grafica. Questi strumenti sono estremamente utili per operazioni di profiling, debugging e ottimizzazione delle prestazioni.

Le query si basano su oggetti OpenGL appositi, inizializzabili e gestibili con le seguenti funzioni:

```
GLuint query;
glGenQueries(1, &query);
glBeginQuery(target, query);
// ... codice di rendering ...
glEndQuery(target);
GLuint response;
glGetQueryObjectuiv(query, GL_QUERY_RESULT, &response);
```

Il parametro `target` definisce il tipo di statistica che si intende misurare. Per i Tessellation Shaders, OpenGL supporta due tipi principali di query:

- `GL_TESS_CONTROL_SHADER_PATCHES`: misura il numero totale di invocazioni del Tessellation Control Shader. Ogni patch in ingresso al TCS genera un'invocazione separata, fornendo informazioni sul carico computazionale dello stadio di controllo della tessellazione.
- `GL_TESS_EVALUATION_SHADER_PATCHES`: rileva il numero totale di invocazioni del Tessellation Evaluation Shader. Ogni patch elaborata dal TES corrisponde a un'invocazione, permettendo di valutare quante patch vengono effettivamente processate dal Tessellator e dallo shader di valutazione.

È possibile effettuare query multiple in parallelo, a patto che ciascuna utilizzi un target diverso oppure oggetti di query distinti. Tuttavia, nonostante i vantaggi offerti, le query possono introdurre un certo overhead prestazionale, specialmente

se impiegate in modo sincrono o con eccessiva frequenza. In questi casi, per evitare stalli della CPU e mantenere elevata la fluidità del rendering, è consigliabile utilizzare l'opzione `GL_QUERY_RESULT_NO_WAIT`, che consente di recuperare i risultati in modo asincrono, senza bloccare l'esecuzione del programma.

In generale, l'utilizzo delle query statistiche in OpenGL rappresenta uno strumento fondamentale per comprendere a fondo l'effettivo impatto degli shaders sulla pipeline. Monitorando il numero di invocazioni del TCS e del TES, è possibile effettuare scelte più consapevoli in fase di sviluppo, migliorare le prestazioni e ridurre i colli di bottiglia legati alla tessellazione.

Capitolo 3

Geometry Shader

In questo capitolo viene analizzato in dettaglio un secondo shader che può essere aggiunto nella Pipeline standard, descritta nel capitolo 1, al fine di arricchire le funzioni offerte e migliorare i risultati finali.

3.1 Introduzione

Il Geometry Shader rappresenta uno stadio opzionale della pipeline grafica di OpenGL, introdotto a partire dalla versione 3.2 e ampiamente utilizzato nelle specifiche Core 4.6. Esso si colloca immediatamente dopo il Vertex Shader (oppure, se attivi, dopo i Tessellation Shaders) e prima del Rasterizer. La sua funzione principale è quella di elaborare primitive geometriche complete, generando nuove primitive da inviare agli stadi successivi della pipeline [2], [6], [7], [14], [18].

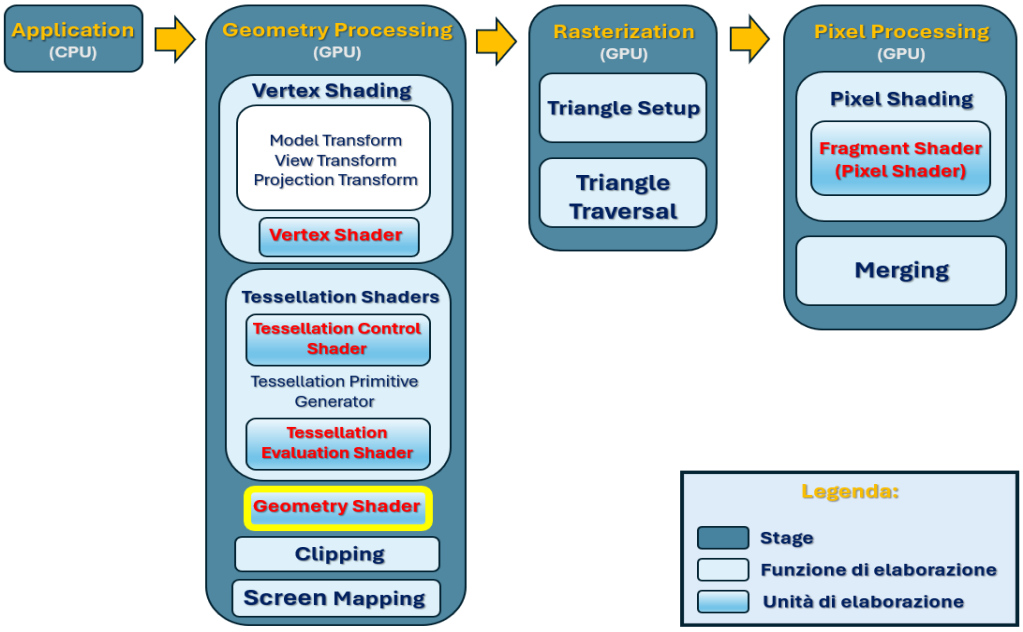

Figura 3.1: Schema della pipeline di rendering con il Geometry Shader.

3.2 Caratteristiche generali

Una delle principali peculiarità del Geometry Shader è la possibilità di accedere simultaneamente a tutti i vertici che compongono una primitiva, a differenza del Vertex Shader che opera su singoli vertici in modo indipendente. Questo lo rende adatto a operazioni che richiedono la conoscenza della struttura geometrica complessiva della primitiva, come la generazione di nuove primitive o l'eliminazione condizionata di quelle in ingresso.

Le due operazioni principali che il Geometry Shader consente di eseguire sono il *culling* e l'*amplifying*. **La prima** consiste nello scartare primitive in base a criteri specifici, evitando di emettere vertici verso gli stadi successivi della pipeline. Questo permette di ridurre il carico computazionale complessivo e ottimizzare il rendering. Se il processo di scarto coinvolge solo un sottoinsieme limitato e selezionato delle primitive, si parla di *selective culling*. **La seconda** operazione invece, indica la generazione dinamica di una o più primitive di output a partire da ciascuna primitiva in ingresso. Le primitive emesse devono appartenere a un solo tipo, che può anche essere differente rispetto a quello della primitiva originale. Questa tecnica è utile per implementare algoritmi che richiedono l'aggiunta di dettaglio geometrico in modo dinamico, come l'espansione di mesh, la generazione di silhouette o l'estrusione di superfici.

3.3 Creazione e distruzione

Per quanto riguarda il Geometry Shader, la procedura di creazione e distruzione non si discosta da quella generale illustrata nella sezione 2.3.1. La differenza risiede esclusivamente nel parametro passato alla funzione `glCreateShader`, che in questo caso è `GL_GEOMETRY_SHADER`. In questo modo lo shader viene riconosciuto da OpenGL come Geometry Shader, rendendone possibile la compilazione e l'utilizzo all'interno della pipeline grafica.

3.4 Primitive di adiacenza

Con l'introduzione del Geometry Shader in OpenGL è stata ampliata la tipologia di primitive grafiche disponibili, includendo anche primitive arricchite da vertici di adiacenza (*adjacency vertices*). Questi vertici aggiuntivi non fanno parte della primitiva principale da renderizzare, ma forniscono informazioni sui poligoni o segmenti adiacenti. Ciò risulta particolarmente utile in algoritmi che richiedono consapevolezza della topologia locale della mesh, come nel caso di rilevamento dei bordi (*edge detection*), *shadow volumes* (tecnica per generare ombre basata sulle silhouette degli oggetti) o smussatura geometrica (*smooth shading*).

Di seguito si riportano le principali primitive con adiacenze supportate da OpenGL:

- **GL_LINES_ADJACENCY** (Figura 3.2): questa modalità definisce segmenti con adiacenza laterale. Ogni geometria è caratterizzata da 4 vertici. I

vertici v_1 e v_2 sono utilizzati per costruire il segmento, mentre i vertici v_0 e v_3 forniscono informazioni topologiche sulle connessioni precedenti e successive.

Figura 3.2: Rappresentazione di una primitiva `GL_LINES_ADJACENCY`.

- **GL_LINE_STRIP_ADJACENCY** (Figura 3.3): ogni primitiva è costituita da $N+3$ vertici, collegati in modo da formare una sequenza di segmenti adiacenti (dove N indica il numero di segmenti che si andranno a generare), comprensiva delle informazioni sui vertici adiacenti alla sequenza stessa. I segmenti sono ottenuti collegando ogni coppia di vertici successivi, da v_1 a v_{N+1}. I vertici v_0 e v_{N+2} invece, fanno riferimento ai vertici adiacenti rspettivamente all'inizio e alla fine della sequenza.

Figura 3.3: Rappresentazione di una primitiva `GL_LINE_STRIP_ADJACENCY`.

- **GL_TRIANGLES_ADJACENCY** (Figura 3.4): consente di specificare un triangolo insieme ai vertici adiacenti ai suoi lati. Ogni primitiva è caratterizzata da 6 vertici. Il triangolo vero e proprio viene costruito utilizzando i tre vertici v_0, v_2 e v_4. Gli altri vertici si riferiscono ai punti adiacenti relativi ai vari lati del triangolo: il vertice v_1 è adiacente al lato (v_0, v_2), v_3 è adiacente a (v_2, v_4) e v_5 è adiacente a (v_4, v_0).

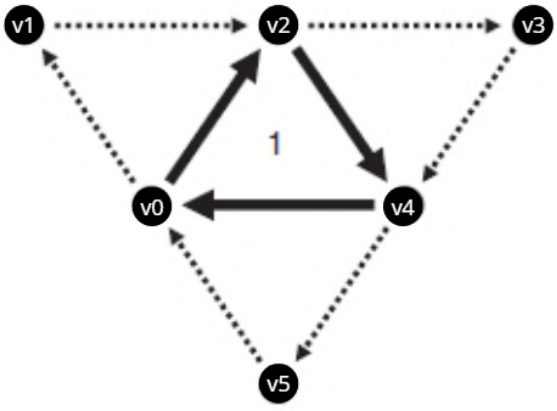

Figura 3.4: Rappresentazione di una primitiva `GL_TRIANGLES_ADJACENCY`.

- **GL_TRIANGLE_STRIP_ADJACENCY** (Figura 3.5): ogni primitiva utilizza $4 + 2N$ vertici, definendo una sequenza di triangoli con adiacenze (dove N è il numero di triangoli che si andranno a generare). Partendo dal vertice v_0, tutti i vertici con indice pari (v_0, v_2, v_4, ...) costituiscono i triangoli principali, mentre quelli con indice dispari (v_1, v_3, v_5, ...) rappresentano i

vertici adiacenti ai rispettivi lati del triangolo. Ogni triangolo viene costruito facendo scorrere una finestra di tre vertici pari consecutivi (es. v_0, v_2, v_4 poi v_2, v_4, v_6, e così via), associati ai rispettivi vertici di adiacenza. Per ciascun lato del triangolo è fornito un vertice adiacente, che permette di accedere alla geometria confinante.

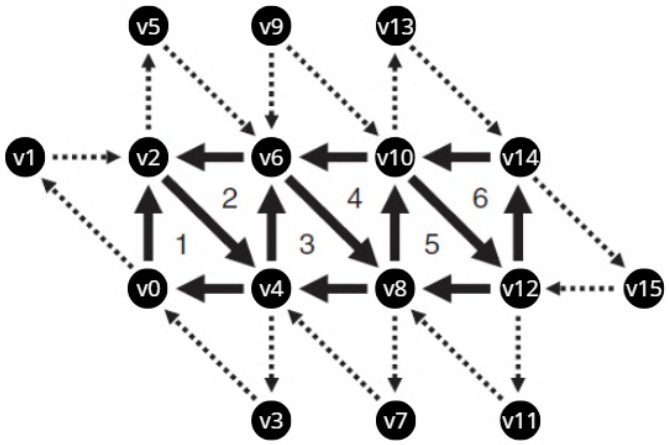

Figura 3.5: Rappresentazione di una primitiva `GL_TRIANGLE_STRIP_ADJACENCY`.

3.5 Input

Nonostante OpenGL supporti una vasta gamma di primitive geometriche nei vari stadi della pipeline di rendering, il Geometry Shader può accettare in ingresso solo un sottoinsieme ristretto di queste. Ciascuna primitiva ammessa rappresenta una singola entità geometrica composta da un numero fisso di vertici, che il Geometry Shader elabora come unità.

All'ingresso del Geometry Shader è necessario specificare il tipo di primitiva che esso riceverà in input, utilizzando la seguente sintassi:

```
1 layout(type) in;
```

dove `type` rappresenta una delle primitive geometriche supportate, elencate di seguito. Questa dichiarazione è obbligatoria, poichè informa il compilatore su come interpretare i dati dei vertici provenienti dallo stadio precedente della pipeline, ed è fondamentale affinchè lo shader possa accedere ai vertici in modo coerente.

Il tipo di primitiva specificato deve essere compatibile con l'output del Vertex Shader oppure, se presente, con quello del Tessellation Evaluation Shader. In particolare, se il Tessellation Shader è attivo, il tipo di primitiva deve corrispondere al dominio di output dichiarato nel Tessellation Evaluation Shader; altrimenti, deve essere coerente con la primitiva utilizzata a livello applicativo (Application Stage) nella chiamata a `glDrawArrays` o `glDrawElements`, così come è coerente a queste il Vertex Shader.

Di seguito vengono elencate le primitive geometriche utilizzabili come input per un Geometry Shader, con il valore da sostituire a `type` indicato tra parentesi:

- **points** (`points`): ogni invocazione dello shader riceve un singolo vertice. Utilizzato per operazioni puntuali come la generazione di particelle. È valido in combinazione con le primitive `GL_POINTS` (in caso di Vertex Shader) o in modalità `point_mode` (in caso di Tessellation Shaders).
- **lines** (`lines`): ogni invocazione riceve due vertici, che rappresentano l'inizio e la fine di un segmento. Utilizzato per primitive di tipo `GL_LINES` (in caso di Vertex Shader) o per il dominio `isolines` (in caso di Tessellation Shaders).
- **lines with adjacency** (`lines_adjacency`): ogni invocazione riceve quattro vertici. I vertici v_1 e v_2 formano la linea principale, mentre v_0 e v_3 sono vertici adiacenti alle estremità della linea. Questa configurazione fornisce informazioni topologiche locali ed è usata con le primitive `GL_LINES_ADJACENCY` e `GL_LINE_STRIP_ADJACENCY` (in caso di Vertex Shader). Questa primitiva non è compatibile con i domini dei Tessellation Shaders, risultando quindi disponibile solamente per i parametri della funzione `glDrawArrays` specificati in precedenza.
- **triangles** (`triangles`): ogni invocazione riceve tre vertici, che definiscono un triangolo. È la primitiva più comune per rappresentare superfici e viene utilizzata in presenza di `GL_TRIANGLES` (in caso di Vertex Shader) oppure con i domini `triangles` e `quads` del Tessellation Shaders (in quest'ultimo caso, i quadrilateri vengono suddivisi in triangoli).
- **triangles with adjacency** (`triangles_adjacency`): ogni invocazione riceve sei vertici. I vertici v_0, v_2 e v_4 definiscono il triangolo principale, mentre v_1, v_3 e v_5 sono vertici adiacenti ai lati opposti di ciascun vertice del triangolo. Utilizzato con `GL_TRIANGLES_ADJACENCY` e `GL_TRIANGLE_STRIP_ADJACENCY` (in caso di Vertex Shader). Questa primitiva non è compatibile con i domini dei Tessellation Shaders, risultando quindi disponibile solamente per i parametri della funzione `glDrawArrays` specificati in precedenza.

3.6 Output

Analogamente all'input, anche per l'output i tipi di primitiva che un Geometry Shader può generare costituiscono un sottoinsieme limitato rispetto all'intero set disponibile in OpenGL. In questo caso, il tipo di primitiva emessa deve essere dichiarato esplicitamente tramite l'istruzione:

```
1 layout(type, max_vertices = N) out;
```

dove il parametro `type` specifica la tipologia di primitiva prodotta dallo shader, mentre `max_vertices` indica il numero massimo di vertici che possono essere generati per ciascuna invocazione dello shader. Eventuali vertici generati oltre questo limite verranno ignorati. La specifica di `max_vertices` è obbligatoria e permette al compilatore di ottimizzare l'allocazione delle risorse durante il rendering.

I vertici emessi dal Geometry Shader vengono assemblati nella primitiva indicata dalla dichiarazione. Le tipologie di output supportate sono:

- **points** (`points`): ogni vertice emesso viene interpretato come una primitiva puntiforme indipendente.
- **line strip** (`line_strip`): viene generata una sequenza di segmenti collegati. È necessario emettere almeno due vertici per costruire la prima linea; ogni vertice successivo estende la striscia aggiungendo un nuovo segmento connesso all'ultimo vertice emesso.
- **triangle strip** (`triangle_strip`): è necessario emettere almeno tre vertici per iniziare la costruzione. Ogni nuovo vertice, a partire dal terzo, genera un nuovo triangolo insieme ai due vertici precedenti. Con N vertici emessi, è possibile generare fino a $N - 2$ triangoli. Ad esempio, con $N = 6$, i triangoli generati saranno: (v_1, v_2, v_3), (v_2, v_3, v_4), (v_3, v_4, v_5) e (v_4, v_5, v_6).

Questi tre tipi di output permettono di creare sia forme geometriche complesse sia quelle più semplici. Ad esempio, per generare un singolo triangolo è sufficiente specificare `triangle_strip` e restituire esattamente tre vertici.

È importante sottolineare che il valore assegnato a `max_vertices` non è puramente formale: questo parametro influenza direttamente l'allocazione delle risorse interne durante l'esecuzione del Geometry Shader. Se impostato troppo alto, senza una reale necessità basata sul numero di vertici generati, può causare uno spreco di memoria e ridure l'efficienza del rendering. Al contrario, se è troppo basso, si rischia di non riuscire a generare tutte le primitive desiderate. Per questo motivo, è consigliabile stimare con cura il numero massimo di vertici realmente necessari in ciascuna invocazione dello shader, e fornire un valore di `max_vertices` il più preciso possibile, lasciando eventualmente un piccolo margine di sicurezza. In questo modo si ottimizza l'uso delle risorse e si migliora la performance complessiva.

3.7 Variabili built-in e funzionamento interno

3.7.1 Gestione dei dati in ingresso

Il Geometry Shader viene invocato una volta per ciascuna primitiva in ingresso, emessa dallo stadio precedente della pipeline (il Tessellation Evaluation Shader o il Vertex Shader). Durante ogni invocazione, il Geometry Shader ha accesso ai dati relativi ai vertici della primitiva in ingresso attraverso l'array built-in `gl_in[]` (vedi sezione 2.3.2).

Oltre a questo, esistono anche altre variabili built-in globali di input a cui il Geometry Shader può accedere. Fra queste, alcune delle più importanti sono:

- `gl_PrimitiveIDIn` (`int`): identificatore della primitiva in ingresso. Viene assegnato automaticamente dal sistema, incrementato per ogni primitiva elaborata. È utile in contesti dove è necessario distinguere tra primitive individuali per applicare effetti specifici, oppure per indicizzare buffer o array uniform.
- `gl_InvocationID` (`int`): se è stata abilitata l'esecuzione multipla per primitiva tramite la direttiva `layout(invocations = N)`, questa variabile contiene

l'indice dell'invocazione corrente (da 0 a $N-1$). Ogni invocazione riceve gli stessi dati in input, ma può produrre output indipendenti, rendendo possibile il rendering parallelo di più varianti geometriche della stessa primitiva.

3.7.2 Emissione di vertici

Una delle peculiarità più potenti del Geometry Shader rispetto agli stadi precedenti della pipeline grafica è la capacità di generare dinamicamente nuove primitive partendo da quelle in ingresso. Questa funzionalità è abilitata attraverso due funzioni fondamentali fornite da GLSL:

- `EmitVertex()`: emette un singolo vertice verso lo stream di output. I dati associati al vertice (posizione, attributi, layer, ecc.) sono determinati dal valore corrente delle variabili di output, siano esse built-in o definite dall'utente. La funzione memorizza i valori attuali delle variabili e li copia nello stream. Il vertice emesso contribuisce alla costruzione della primitiva definita nel layout di output dello shader. Dopo ogni chiamata a `EmitVertex()`, le variabili di output assumono valore `undefined`, a meno che siano dichiarate con il qualificatore `flat`. In tal caso, il loro valore viene mantenuto invariato anche nei vertici successivi, evitando la necessità di riassegnarle esplicitamente. Questo comportamento è utile per trasmettere informazioni discrete o costanti, senza rischio di inconsistenza.

- `EndPrimitive()`: termina l'assemblaggio della primitiva corrente, combinando i vertici presenti nel buffer per formare la primitiva specificata dal tipo di output specificato. Dopo aver terminato la primitiva, avvia implicitamente l'assemblaggio di una nuova primitiva a partire dai successivi vertici emessi. E' indispensabile quando si stanno emettendo più primitive distinte nella stessa invocazione. Se non viene chiamata esplicitamente, OpenGL la invoca automaticamente alla fine dell'invocazione dello shader, concludendo la primitiva corrente con tutti i vertici emessi fino a quel momento.

Il numero massimo di vertici che si possono emettere è limitato dal parametro `max_vertices` dichiarato nel layout dello shader. È fondamentale non superare questo limite, pena comportamenti indefiniti o errori di validazione.

Se si utilizza il Transform Feedback, ogni vertice emesso tramite `EmitVertex()` può essere catturato in un buffer sul lato CPU/GPU. In questo contesto, la distinzione tra `EmitVertex()` e `EndPrimitive()` diventa rilevante anche per la segmentazione logica dei dati.

Il comportamento delle variabili `flat` fa riferimento al *provoking vertex*, ossia il primo vertice della primitiva per default (modificabile con `glProvokingVertex`). Questo meccanismo è utile per codificare ID di istanza, etichette di segmentazione o altre informazioni discrete che non devono essere interpolate.

3.7.3 Gestione dei dati in uscita

Nel Geometry Shader, l'emissione di vertici verso lo stadio successivo della pipeline avviene attraverso un insieme di variabili di output, sia definite dall'utente sia fornite come built-in dal linguaggio GLSL. Le variabili built-in di output sono fondamentali per controllare il comportamento della rasterizzazione, l'identità delle primitive emesse e la destinazione del rendering nel framebuffer.

Queste variabili devono essere valorizzate esplicitamente prima di ogni chiamata a EmitVertex(), poiché il loro stato viene invalidato dopo ciascuna emissione. In caso contrario, i vertici successivi potrebbero contenere dati incoerenti o non definiti.

Le principali variabili built-in di output sono:

- `gl_Position` (`vec4`): rappresenta la posizione nello spazio clip (spazio omogeneo) del vertice emesso. E' la variabile più importante, in quanto determina dove la geometria verrà proiettata all'interno dello spazio clip e, successivamente, visualizzato a schermo.

- `gl_PointSize` (`float`): specifica la dimensione in pixel del punto da rasterizzare. È usata solo se il tipo di primitiva di output è `points`. Valori troppo piccoli possono rendere il punto invisibile, mentre valori troppo grandi possono causare artefatti o clipping.

- `gl_ClipDistance[]` e `gl_CullDistance[]` (`float[]`): permettono di definire piani di clipping e culling personalizzati. Ogni elemento rappresenta la distanza del vertice da un piano implicito. Se tutti i vertici di una primitiva hanno valori negativi per un determinato piano, la primitiva viene completamente scartata.

- `gl_PrimitiveID` (`int`): identifica la primitiva emessa dal Geometry Shader. Può essere usato per distinguere e marcare le primitive nell'output finale. Mentre `gl_PrimitiveIDIn` è assegnato automaticamente alla primitiva in ingresso, `gl_PrimitiveID` può essere impostato manualmente e indipendentemente per ciascuna primitiva emessa. Questa flessibilità consente, ad esempio, di duplicare primitive assegnando loro ID distinti.

- `gl_Layer` (`int`): specifica il layer del framebuffer verso cui la primitiva sarà rasterizzata. È utilizzato nei contesti di layered rendering (generazione dinamica di ogni faccia di una cubemap, shadow map array, rendering simultaneo su più texture 2D-array, ...). Se omesso, la primitiva viene scritta nel layer 0 di default.

- `gl_ViewportIndex` (`int`): determina l'indice della viewport attiva su cui rasterizzare la primitiva. Usato in combinazione con più viewport definite tramite `glViewportArrayv` (utile per scenari di rendering stereoscopico, split-screen o visualizzazione simultanea da angolazioni diverse).

Tutte queste variabili, se utilizzate, devono essere coerenti con il tipo di output dichiarato nel layout dello shader e con il contesto di rendering.

3.7.4 Gestione di più stream di output

Il Geometry Shader offre una funzionalità avanzata poco utilizzata ma molto potente: la gestione simultanea di stream multipli di output. Questa capacità è stata introdotta con l'obiettivo di rendere più flessibile e modulare il processo di Transform Feedback, permettendo a uno stesso shader di scrivere simultaneamente su più flussi distinti di dati.

In questo contesto, uno stream rappresenta un canale logico distinto attraverso cui lo shader può emettere vertici. Ogni stream è identificato da un intero $(0, 1, 2, ...)$, e può essere utilizzato per generare e differenziare più insiemi di geometrie simultaneamente.

Per specificare a quale stream indirizzare un blocco di output o una singola emissione, si usa il qualificatore: `layout(stream = i)`.

Quando si lavora con stream multipli, non è possibile utilizzare `EmitVertex()` e `EndPrimitive()`, poiché queste funzioni operano implicitamente sullo stream 0. In alternativa, si ricorre alle versioni esplicite:

- `EmitStreamVertex(i)`, per emettere un vertice verso lo stream `i`, utilizzando i valori correnti delle variabili di output dichiarate con `stream = i`.
- `EndStreamPrimitive(i)`, per terminare la primitiva sullo stream `i`, analogamente a `EndPrimitive()` ma per lo stream specificato.

Tuttavia, solo il flusso con `stream = 0` può essere effettivamente rasterizzato. Gli stream con ID maggiore di zero non vengono rasterizzati, ma possono essere catturati tramite Transform Feedback, rendendoli utili per operazioni come la generazione dinamica di geometrie, simulazioni, oppure deferred rendering multi-pass.

Inoltre, l'utilizzo del multi-stream è limitato al tipo di primitiva `points`. Non è consentito emettere linee o triangoli su stream diversi da zero.

3.7.5 Uso e sincronizzazione delle variabili in e out

Nel contesto della pipeline programmabile di OpenGL, lo scambio di dati tra gli stadi avviene attraverso l'utilizzo di variabili qualificabili come `in` (input) e `out` (output), i cui valori vengono interpolati, propagati o semplicemente trasmessi da uno stadio all'altro.

Nel caso specifico del Geometry Shader, tale comunicazione presenta delle particolarità uniche, dovute al fatto che questo stadio opera sull'intera primitiva, e non su singoli vertici come accade nel Vertex Shader.

Quando un Vertex Shader emette una variabile di output, il Geometry Shader riceve i valori per ciascun vertice della primitiva in ingresso. Per questo motivo, ogni variabile `in` nello shader di geometria deve essere dichiarata come array, contenente tanti elementi quanti sono i vertici della primitiva.

Le variabili di output del Geometry Shader vengono dichiarate con il qualificatore `out`, analogamente agli altri stadi. Tuttavia, a differenza del Vertex Shader,

dove ciascun `out` è automaticamente associato al vertice corrente, nel Geometry Shader è necessario assegnare esplicitamente tali valori prima di ogni chiamata a `EmitVertex()`.

È fondamentale che le variabili `out` di uno stadio corrispondano esattamente, per nome e tipo, alle variabili `in` dello stadio successivo. In caso contrario, OpenGL potrebbe generare errori di link durante la compilazione del programma shader, oppure comportamenti indefiniti in fase di esecuzione.

3.8 Layered Rendering

3.8.1 Render to screen e render to texture

Nel contesto della grafica in tempo reale, è fondamentale distinguere tra due modalità principali di rendering:

- **Render to screen**: è la modalità predefinita, in cui l'output finale della pipeline grafica viene visualizzato direttamente sullo schermo, tramite il *default framebuffer*, fornito dal sistema operativo o dalla libreria grafica (come GLFW). Questa modalità consente di scrivere su una sola superficie alla volta e non è adatta a tecniche avanzate che richiedono più destinazioni di output.
- **Render to Texture**: conosciuto anche come *off-screen rendering*, consiste nel disegnare il risultato della pipeline su una texture invece che sullo schermo. Questa texture può poi essere riutilizzata in passaggi successivi della pipeline grafica, per effetti come *shadow mapping*, riflessi e rifrazioni, *deferred shading*, *post-processing*, o rendering multi-vista.

3.8.2 Caratteristiche generali

Il *layered rendering* è una tecnica di *render to texture* migliorata per permettere di scrivere simultaneamente su più layer di una texture complessa all'interno di un'unica draw call.

In OpenGL, questa tecnica è abilitata tramite la variabile built-in `gl_Layer` nel Geometry Shader (o in altri stadi avanzati tramite estensioni). Ogni primitiva generata dallo shader può essere indirizzata verso un layer specifico della texture, senza dover effettuare più passaggi di rendering separati.

Il layered rendering può essere applicato ai seguenti tipi di texture:

- **Texture 3D**: si tratta di texture organizzate come un insieme di *slice* bidimensionali lungo l'asse della profondità. Ogni slice rappresenta uno strato della struttura tridimensionale. Queste texture sono particolarmente utili per effetti volumetrici (come nebbia, fumo o materiali traslucidi) e simulazioni tridimensionali, dove è necessario campionare dati in tutto il volume.
- **Texture Array 2D**: collezioni di texture 2D indipendenti ma tutte della stessa dimensione. Ogni layer dell'array rappresenta una singola immagine.

Questa struttura consente di accedere a più texture nello stesso shader senza dover cambiare binding, risultando efficiente per scenari come il rendering simultaneo di ombre generate da più luci (*cascaded shadow maps*, ossia mappe d'ombra generate a più distanze per luci direzionali) o per il *multiview rendering*, dove la stessa scena viene renderizzata da più angolazioni in un unico passaggio.

- **Cube Map**: composte da sei facce quadrate che rappresentano le sei direzioni dello spazio tridimensionale ($\pm X, \pm Y, \pm Z$). Queste texture vengono spesso utilizzate per creare riflessioni ambientali e per lo *shadow mapping omnidirezionale*, tipico di luci puntiformi, dove è necessario calcolare ombre in tutte le direzioni a partire da una singola sorgente luminosa.

3.8.3 Funzionamento

Il *layered rendering* sfrutta la variabile built-in `gl_Layer` all'interno del Geometry Shader, per indirizzare ciascuna primitiva generata verso un layer specifico di una texture complessa, come una 3D, una 2D array o una cube map. Questo consente di aggiornare più layer contemporaneamente all'interno di una singola *draw call*, evitando la necessità di effettuare più passaggi di rendering separati.

Dal punto di vista della CPU, il processo richiede alcuni passaggi preparatori fondamentali. Innanzitutto, è necessario creare la texture di destinazione, allocando tutti i layer desiderati direttamente in memoria GPU. Questa texture viene poi collegata a un *Framebuffer Object* tramite gli attachment appropriati. Spesso viene allocato anche un depth buffer condiviso, in modo che tutte le primitive scritte nei vari layer possano essere testate correttamente. Parallelamente, si preparano gli shader che utilizzeranno `gl_Layer`, assicurandosi che siano compilati e pronti per la pipeline.

Durante il rendering, le primitive generate in un'unica draw call possono essere distribuite tra i diversi layer in base a logiche definite nello shader, come l'indice della luce, la slice di un volume o la faccia di una cube map. Lato CPU, è importante avere una corrispondenza chiara tra layer e significato applicativo. Questo consente di controllare e prevedere correttamente il contenuto di ciascun layer.

Al termine del rendering, la texture ottenuta può essere riutilizzata in passaggi successivi della pipeline grafica quali: *shadow mapping* (calcolo delle ombre proiettate dagli oggetti rispetto a una sorgente luminosa), *deferred shading* (tecnica in cui le informazioni geometriche e di materiale vengono prima memorizzate in buffer separati e poi usate per calcolare l'illuminazione), *post-processing* (applicazione di effetti visivi sull'immagine finale) e *multiview rendering* (rendering simultaneo della stessa scena da più angolazioni).

Infine, i risultati di questi passaggi vengono memorizzati in appositi buffer o texture, che possono essere poi combinati, filtrati o campionati negli stadi successivi della pipeline per generare l'immagine finale da visualizzare sullo schermo o da riutilizzare in effetti grafici più complessi. In questo modo, ogni passaggio contribuisce alla costruzione dell'immagine finale senza dover riscrivere nuovamente tutta la scena.

3.8.4 Considerazione sulle prestazioni

Il layered rendering permette di ridurre drasticamente il numero di passaggi di rendering richiesti per aggiornare più superfici, con notevoli benefici in termini di performance, soprattutto su GPU moderne. Tuttavia, deve essere usato con attenzione poichè può aumentare la complessità degli shader e incrementare l'utilizzo di risorse di memoria.

3.9 Query e misurazioni

Come già visto nel capitolo precedente (vedi sezione 2.6), OpenGL mette a disposizione meccanismi di query statistiche per valutare le prestazioni e l'esecuzione dei Tessellation Shaders. Analogamente, questi strumenti possono essere utilizzati anche per monitorare vari aspetti del Geometry Shader, fornendo informazioni precise sul suo comportamento all'interno della pipeline grafica.

Per il Geometry Shader, OpenGL supporta due tipi principali di query:

- `GEOMETRY_SHADER_INVOCATIONS`: conta il numero totale di invocazioni del Geometry Shader effettuate durante il rendering. Ogni primitiva in ingresso allo shader (es. un triangolo o una linea) genera un'invocazione distinta. Questo valore è utile per verificare il corretto comportamento della pipeline e per valutare il carico computazionale effettivo sul Geometry Shader.
- `GEOMETRY_SHADER_PRIMITIVES_EMITTED`: rileva il numero complessivo di primitive effettivamente emesse dallo shader. Questo dato tiene conto delle chiamate a `EmitVertex()` e `EndPrimitive()`, e può risultare diverso dal numero di primitive in ingresso, specialmente nei casi di generazione o filtraggio procedurale. E' uno strumento chiave per comprendere l'efficienza e l'utilizzo dello stadio, soprattutto in applicazioni che sfruttano il Geometry Shader per amplificare o ridurre dinamicamente la geometria.

Anche per il Geometry Shader, come per i Tessellation Shaders, è possibile eseguire query multiple in parallelo e leggere i risultati in modalità asincrona tramite `GL_QUERY_RESULT_NO_WAIT`, riducendo così l'eventuale overhead prestazionale.

In generale, l'utilizzo delle query statistiche permette di ottenere una visione chiara del comportamento del Geometry Shader, aiutando a ottimizzare le prestazioni e a individuare eventuali colli di bottiglia nella pipeline grafica.

3.10 Geometry Shader Instancing

Il Geometry Shader Instancing è una funzionalità avanzata del linguaggio GLSL che consente l'esecuzione parallela e indipendente di più istanze dello shader per ciascuna primitiva in ingresso. In altre parole, per ogni primitiva ricevuta dallo stadio precedente (il Tessellation Evaluation Shader o il Vertex Shader), è possibile avviare più invocazioni dello stesso Geometry Shader, ognuna delle quali lavora in modo autonomo ma condividendo gli stessi dati di input.

L'instancing si abilita dichiarando nel layout di ingresso del Geometry Shader il numero di invocazioni desiderato tramite il qualificatore: `layout(invocations = N) in`, dove N è un intero positivo che specifica quante istanze dello shader verranno eseguite per ogni primitiva. Il valore di N non può superare il limite imposto dall'hardware, accessibile tramite la costante `MAX_GEOMETRY_SHADER_INVOCATIONS`, ovvero il valore massimo stabilito dal linguaggio per il Geometry Shader.

All'interno del codice GLSL dello shader, ogni invocazione può distinguersi mediante la variabile built-in: `in int gl_InvocationID`. Questa variabile assume un valore intero compreso nell'intervallo $[0, N-1]$ e identifica univocamente ciascuna istanza. In questo modo, è possibile differenziare il comportamento delle invocazioni sulla base del loro ID, pur partendo dallo stesso input.

Ogni invocazione è libera di emettere vertici e primitive separatamente dalle altre, utilizzando le normali funzioni `EmitVertex()` ed `EndPrimitive()`. Tuttavia, tutte le invocazioni devono rispettare il limite massimo dichiarato tramite: `layout(max_vertices = M) out`.

Sebbene l'instancing offra una grande flessibilità, è importante tenere conto del costo computazionale: ogni invocazione introduce un overhead aggiuntivo. È consigliabile: limitare il numero di invocazioni, evitare duplicazioni di lavoro tra invocazioni e sfruttare il parallelismo solo quando necessario, per evitare sprechi di risorse GPU.

Capitolo 4

Fondamenti Matematici

La matematica costituisce la base teorica indispensabile per lo studio e lo sviluppo delle tecniche di computer grafica affrontate in questa tesi, sia da un punto di vista teorico, sia da un punto di vista pratico attraverso il progetto. In particolare, la descrizione di superfici, la manipolazione di dati geometrici e la definizione di alcuni algoritmi di rendering richiedono la conoscenza di alcuni concetti fondamentali di analisi matematica e algebra vettoriale [17].

Questo capitolo ha lo scopo di presentare una panoramica degli strumenti matematici di riferimento, in modo da costruire una base solida che permetta di comprendere e giustificare i passaggi che, nei capitoli successivi, verranno assunti come già acquisiti.

4.1 Spazi vettoriali e operazioni sui vettori

Gli spazi vettoriali forniscono la struttura matematica fondamentale per rappresentare punti, vettori e direzioni nello spazio. La comprensione di concetti come combinazioni lineari, basi, norme e prodotti scalari o vettoriali è indispensabile per descrivere concetti geometrici alla base di molte operazioni di computer grafica.

4.1.1 Concetti fondamentali

Spazio vettoriale

Sia dato un generico campo $\mathbb{K}$ (ad esempio $\mathbb{R}$, $\mathbb{C}$ o $\mathbb{Q}$). Gli elementi di $\mathbb{K}$ sono detti *scalari*. Uno *spazio vettoriale* su $\mathbb{K}$ è un insieme V di elementi, detti *vettori*, che possono essere rappresentati, nel caso particolare di $\mathbb{K}^n$, tramite una tupla di n componenti scalari. Geometricamente, in questo caso, ciascun vettore può essere interpretato come un segmento orientato che parte dall'origine e termina nel punto definito dalle sue componenti, definendone modulo, direzione e verso. Uno spazio vettoriale è poi dotato di due operazioni:

- Una operazione detta *somma* ($V + V \to V$), che associa a due vettori $v, w \in V$ un terzo vettore $v + w \in V$.

- Una operazione detta *prodotto per scalare* ($\mathbb{K} \times V \to V$), che associa ad un vettore $v \in V$ e ad uno scalare $\lambda \in \mathbb{K}$ un vettore $\lambda v \in V$.

Queste due operazioni soddisfano le seguenti proprietà:

- $v + w = w + v \quad \forall\, v, w \in V$
- $(v + w) + u = v + (w + u) \quad \forall\, v, w, u \in V$
- $a(bv) = (ab)v \quad \forall\, v \in V \quad \forall a, b \in \mathbb{K}$
- $(a + b)v = av + bv \quad \forall\, v \in V \quad \forall a, b \in \mathbb{K}$
- $a(v + w) = av + aw \quad \forall\, v, w \in V \quad \forall a \in \mathbb{K}$
- $\exists\, (-v) \mid v + (-v) = 0 \quad \forall\, v \in V$
- $0 + v = v \quad \forall\, v \in V$
- $1v = v \quad \forall\, v \in V$

Combinazione lineare

Siano $v_1, v_2, \ldots, v_n$ vettori appartenenti a uno spazio vettoriale V. Un vettore $v \in V$ è una *combinazione lineare* di questi vettori, se esistono degli scalari $\lambda_1, \lambda_2, \ldots, \lambda_n \in \mathbb{K}$ tali che:

$$v = \lambda_1 v_1 + \lambda_2 v_2 + \cdots + \lambda_n v_n$$

In altre parole, un vettore è combinazione lineare di altri vettori di partenza se può essere scritto come somma pesata dei vettori di partenza con coefficienti scalari.

Inoltre, i vettori $v_1, \ldots, v_n$ sono detti *linearmente indipendenti* se nessuno di questi vettori è combinazione lineare degli altri, cioè se l'unica combinazione lineare che produce il vettore nullo è quella in cui tutti i coefficienti sono nulli:

$$\lambda_1 v_1 + \lambda_2 v_2 + \cdots + \lambda_n v_n = 0 \quad \Longrightarrow \quad \lambda_1 = \lambda_2 = \cdots = \lambda_n = 0$$

Se esiste una combinazione non banale (con almeno un coefficiente diverso da zero) che dà il vettore nullo, i vettori sono invece detti *linearmente dipendenti.*

Base

Sia V uno spazio vettoriale definito su un campo $\mathbb{K}$. Una *base* di V è un insieme di vettori linearmente indipendenti $\{v_1, \ldots, v_n\}$, tale che ogni vettore $v \in V$ possa essere scritto come combinazione lineare di essi:

$$v = \lambda_1 v_1 + \cdots + \lambda_n v_n, \quad \lambda_i \in \mathbb{K}$$

Base canonica in $\mathbb{R}^n$

In uno spazio vettoriale $\mathbb{R}^n$, la *base canonica* è l'insieme di vettori $\{e_1, e_2, \ldots, e_n\}$ definiti come:

$$e_1 = (1, 0, 0, \ldots, 0), \quad e_2 = (0, 1, 0, \ldots, 0), \quad \ldots, \quad e_n = (0, 0, \ldots, 1)$$

Ogni vettore della base canonica ha infatti tutte le componenti nulle eccetto una componente pari a 1, corrispondente alla posizione del vettore nella base.

Questo particolare tipo di base permette di esprimere in modo diretto ogni vettore $x \in \mathbb{R}^n$, con componenti $x = (x_1, x_2, \ldots, x_n)$, come combinazione lineare dei suoi vettori:

$$x = x_1 e_1 + x_2 e_2 + \cdots + x_n e_n$$

4.1.2 Operazioni fondamentali

Somma di vettori

Data una coppia di vettori $a, b \in \mathbb{R}^n$, la loro somma è definita componente per componente:

$$a + b = (a_1 + b_1,\ a_2 + b_2,\ \ldots,\ a_n + b_n)$$

Geometricamente, questa operazione viene eseguita secondo il *metodo del parallelogramma*: si pongono a e b con origine comune, costruendo, a partire da questi, il parallelogramma relativo. La diagonale del parallelogramma formato rappresenta il risultato della somma $a + b$.

Prodotto per scalare

Dato un vettore $a \in \mathbb{R}^n$ e uno scalare $\lambda \in \mathbb{K}$, il loro prodotto è definito come:

$$\lambda a = (\lambda a_1,\ \lambda a_2,\ \ldots,\ \lambda a_n)$$

Geometricamente, λ agisce come un fattore di scala. Se $\lambda > 1$ il vettore viene allungato, se $0 < \lambda < 1$ viene accorciato, mentre se $\lambda < 0$ viene anche invertito di direzione.

Norma di un vettore

La *norma* (detta anche *lunghezza* o *modulo*) di un vettore $a \in \mathbb{R}^n$ è definita come:

$$\|a\| = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}$$

Questa rappresenta la distanza del punto identificato da a dall'origine. Geometricamente, è la lunghezza del segmento che va dall'origine all'estremo finale del vettore.

Proiezione di vettori

Data una coppia di vettori $a, b \in \mathbb{R}^n$ con $b \neq 0$, la *proiezione* di a sulla direzione di b è definita come:

$$\mathrm{proj}_b(a) = \frac{a \cdot b}{\|b\|^2}\, b$$

Si tratta quindi di un vettore, parallelo a b, che rappresenta la componente di a lungo la direzione individuata da b.

La lunghezza di tale proiezione, ossia il modulo della componente di a lungo b, è:

$$\|\text{proj}_b(a)\| = \frac{|a \cdot b|}{\|b\|}$$

Geometricamente, la proiezione rappresenta l'ombra che il vettore a getta sulla retta generata da b, se la luce è ortogonale a b.

4.1.3 Prodotto scalare tra vettori

Definizione formale

Siano $a, b \in \mathbb{R}^n$ due vettori con componenti $a = (a_1, a_2, \ldots, a_n)$ e $b = (b_1, b_2, \ldots, b_n)$. Il *prodotto scalare canonico* (o *prodotto interno*) tra a e b, indicato con $a \cdot b$ (o con $\langle a, b \rangle$), è definito come la somma dei prodotti delle componenti corrispondenti dei due vettori:

$$a \cdot b = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = \sum_{i=1}^{n} a_i b_i \quad (a \cdot b \in \mathbb{R})$$

Proprietà

In generale, un prodotto scalare definito sullo spazio vettoriale $\mathbb{R}^n$ è un'applicazione $\mathbb{R}^n \times \mathbb{R}^n \longrightarrow \mathbb{R}$ che gode delle seguenti proprietà:

- *Simmetria*: $a \cdot b = b \cdot a \quad \forall a, b \in \mathbb{R}^n$
- *Distributività rispetto alla somma*: $a \cdot (b + c) = a \cdot b + a \cdot c \quad \forall a, b, c \in \mathbb{R}^n$
- *Linearità rispetto a valori scalari*: $a \cdot (\lambda b) = \lambda (a \cdot b) \quad \forall a, b \in \mathbb{R}^n, \ \forall \lambda \in \mathbb{R}$
- *Positività definita*: $a \cdot a \geq 0 \quad \text{e} \quad a \cdot a = 0 \iff a = 0$

Norma secondo il prodotto scalare

Per comprendere il significato geometrico del prodotto scalare, è utile ridefinire il concetto di *norma*, sulla base di quanto appena detto. La norma di un vettore $a \in \mathbb{R}^n$ può infatti essere espressa in termini di prodotto scalare come la radice quadrata del prodotto del vettore con se stesso:

$$\|a\| = \sqrt{a \cdot a} = \sqrt{\sum_{i=1}^{n} a_i^2}$$

Questa formulazione è resa possibile dalla proprietà di *positività definita* del prodotto scalare, che garantisce che $a \cdot a \geq 0$, rendendo ben definita la radice quadrata.

Significato geometrico

Il prodotto scalare tra due vettori può essere espresso in termini della loro norma e dell'angolo θ compreso tra essi. Dati due vettori $a, b \in \mathbb{R}^n$, vale la relazione:

$$a \cdot b = \|a\| \, \|b\| \cos\theta$$

Questa formula permette di interpretare il prodotto scalare come una misura di quanto un vettore si *proietta* nella direzione dell'altro e di stabilire la relazione angolare tra essi:

- Se $\theta = 0$, i vettori sono paralleli e il prodotto scalare è massimo e positivo ($a \cdot b = \|a\|\|b\|$).
- Se $\theta = \frac{\pi}{2}$, i vettori sono ortogonali e il prodotto scalare è nullo ($a \cdot b = 0$).
- Se $\theta = \pi$, i vettori sono antiparalleli e il prodotto scalare è massimo in valore assoluto ma negativo ($a \cdot b = -\|a\|\|b\|$).

4.1.4 Prodotto Vettoriale

Definizione

Siano $a, b \in \mathbb{R}^3$. Si indichi con e_1, e_2, e_3 la *base canonica* di $\mathbb{R}^3$, dove:

$$e_1 = (1, 0, 0), \quad e_2 = (0, 1, 0), \quad e_3 = (0, 0, 1)$$

Ogni vettore $x \in \mathbb{R}^3$ può essere scritto come combinazione lineare dei vettori della base canonica. In particolare, a e b si potranno esprimere come:

$$a = a_1e_1 + a_2e_2 + a_3e_3 \qquad b = b_1e_1 + b_2e_2 + b_3e_3$$

Il *prodotto vettoriale* (anche detto *cross product* o *prodotto esterno*), indicato con $a \times b$, è definito come un nuovo vettore ottenuto dalla seguente formula:

$$a \times b = \begin{vmatrix} e_1 & e_2 & e_3 \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = (a_2b_3 - a_3b_2)e_1 - (a_1b_3 - a_3b_1)e_2 + (a_1b_2 - a_2b_1)e_3$$

Proprietà principali

Il prodotto vettoriale gode delle seguenti proprietà fondamentali:

- *Antisimmetria*: $a \times b = -(b \times a)$
- *Linearità*: $(\lambda a) \times b = \lambda(a \times b)$ per ogni $\lambda \in \mathbb{R}$
- *Ortogonalità*: $a \times b \perp a$ e $a \times b \perp b$

Significato geometrico

Il vettore $a \times b$ rappresenta il vettore ortogonale al piano individuato da a e b. Il suo verso è determinato dalla *regola della mano destra*: si dispongono pollice, indice e medio in modo che il pollice indichi la direzione di a, l'indice quella di b, e il medio (perpendicolare al palmo) fornisce il verso del prodotto $a \times b$.

Il modulo del prodotto vettoriale è dato dalla formula:

$$\|a \times b\| = \|a\|\|b\| \sin(\theta)$$

che, geometricamente, corrisponde all'area del parallelogramma costruito sui vettori a e b, dove θ è l'angolo tra questi due vettori. Da questa proprietà derivano alcune conseguenze:

- $\|a \times b\| = 0 \iff a$ e b sono vettori paralleli.
- $\|a \times b\|$ ha valore massimo quando a e b sono ortogonali ($\|a\|\|b\|$).

4.2 Sistemi di riferimento e coordinate omogenee

Le coordinate omogenee estendono la rappresentazione dei punti e dei vettori nello spazio, consentendo di trattare traslazioni e proiezioni come trasformazioni lineari mediante matrici 4x4. Questa formalizzazione è essenziale per la gestione delle trasformazioni geometriche nelle pipeline di rendering.

4.2.1 Sistema di riferimento

La sola specifica di una base non è sufficiente per determinare la posizione di un punto, ma occorre individuare anche un punto di riferimento. Il concetto di base vettoriale viene quindi esteso a quello di *riferimento* (o *frame*), definito come una quaterna $F = (e_1, e_2, e_3, P_0)$, dove e_1, e_2, e_3 rappresenta la base e P_0 un punto, chiamato *origine del sistema di riferimento*.

Dato quindi un riferimento F e un punto $P \in \mathbb{R}^3$, quest'ultimo può essere espresso come:

$$P = P_0 + v = P_0 + v_1e_1 + v_2e_2 + v_3e_3$$

dove $v = v_1e_1 + v_2e_2 + v_3e_3$ è il vettore che collega P_0 a P e dove gli scalari v_1, v_2, v_3 rappresentano le coordinate del punto P rispetto al riferimento F.

4.2.2 Coordinate omogenee

Nella rappresentazione classica, vettori e punti possiedono lo stesso numero di componenti, ma vengono rappresentate in maniera differente ($v = v_1e_1 + v_2e_2 + v_3e_3$ e $P = v_1e_1 + v_2e_2 + v_3e_3 + P_0$). Per evitare tale ambiguità, si introduce una rappresentazione univoca dei due elementi, nota con il nome di *coordinate omogenee*.

L'idea consiste nell'associare a ciascun elemento un vettore a quattro componenti, in cui l'ultima assume valore 0 per i vettori e 1 per i punti:

$$v = v_1e_1 + v_2e_2 + v_3e_3 + 0 \cdot P_0 \qquad P = v_1e_1 + v_2e_2 + v_3e_3 + 1 \cdot P_0$$

In questo modo punti e vettori vengono trattati e rappresentati in maniera uniforme: $(v_1, v_2, v_3, 0)^T$ per i vettori e $(v_1, v_2, v_3, 1)^T$ per i punti. Ciò rende possibile l'utilizzo di un unico formalismo matriciale per descrivere le varie trasformazioni geometriche.

4.2.3 Cambio di sistema di riferimento

Si considerino due riferimenti $F_1 = (x, y, z, O)$ e $F_2 = (u, v, w, E)$. Un punto P espresso in riferimento ad F_1, avrà coordinate omogenee $P = (x_p, y_p, z_p, 1)^T$ e si esprimerà come:

$$P = x_p x + y_p y + z_p z + 1 \cdot O$$

Lo stesso punto P, espresso in riferimento ad F_2, avrà coordinate omogenee $P = (u_p, v_p, w_p, 1)^T$ e si esprimerà come:

$$P = u_p u + v_p v + w_p w + 1 \cdot E$$

La relazione che lega le coordinate di P nei due riferimenti può essere espressa tramite una matrice di cambiamento di base:

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_w & x_E \\ y_u & y_v & y_w & y_E \\ z_u & z_v & z_w & z_E \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ w_p \\ 1 \end{bmatrix}$$

dove (x_u, y_u, z_u) rappresentano le coordinate del vettore u rispetto a F_1, (x_v, y_v, z_v) sono le coordinate di v e (x_w, y_w, z_w) quelle di w. La colonna (x_E, y_E, z_E) corrisponde invece alle coordinate del nuovo punto di origine E espresse nel sistema F_1.

4.3 Trasformazioni affini

Le trasformazioni affini rappresentano uno strumento centrale nella geometria computazionale e nella computer grafica, in quanto consentono di collocare gli oggetti all'interno della scena tridimensionale, modificarne la forma, generarne copie e supportare la realizzazione di animazioni basate su variazioni temporali delle trasformazioni stesse. Inoltre, le trasformazioni affini costituiscono la base delle operazioni di trasformazione dei vertici nella pipeline di rendering 3D.

4.3.1 Definizione

Una trasformazione geometrica lineare $f : \mathbb{R}^n \to \mathbb{R}^n$ si dice ***affine*** se può essere espressa come:

$$f(\mathbf{x}) = A\mathbf{x} + \mathbf{b} \quad \text{con } \mathbf{x} \in \mathbb{R}^n$$

dove $A \in \mathbb{R}^{n \times n}$ è una matrice che rappresenta la componente lineare della trasformazione e $\mathbf{b} \in \mathbb{R}^n$ è un vettore che rappresenta la traslazione.

Tale tipo di trasformazione presenta alcune proprietà fondamentali:

- *Collinearità*: i punti di una linea giacciono ancora sulla linea dopo la trasformazione.
- *Rapporto tra le distanze*: il punto medio di un segmento rimane tale anche dopo la trasformazione.

Per rappresentare le trasformazioni affini in maniera compatta, è conveniente usare le coordinate omogenee. Questo permette di ridurre tutte le trasformazioni (comprese le traslazioni) a una singola moltiplicazione matriciale, come avviene per le trasformazioni lineari. In pratica, utilizzando le coordinate omogenee, ogni trasformazione affine può essere riscritta come:

$$f(\mathbf{x})_h = \begin{bmatrix} A & \mathbf{b} \\ \mathbf{0}^T & 1 \end{bmatrix} \mathbf{x}_h \quad \text{con } \mathbf{x}_h = \begin{bmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{bmatrix}$$

dove A rappresenta la componente lineare, $\mathbf{0}^T$ un vettore riga nullo di lunghezza n e $\mathbf{b}$ la traslazione. In questo modo, anche la traslazione, che non è una trasformazione lineare pura, è inglobata nella matrice.

4.3.2 Traslazione

Traslare una primitiva geometrica nello spazio, significa spostare tutti i sui punti $P = (x, y, z)$ di uno stesso vettore $T = (d_x, d_y, d_z)$, fino a raggiungere la nuova posizione $P' = (x', y', z')$, dove:

$$x' = x + d_x \qquad y' = y + d_y \qquad z' = z + d_z$$

In notazione matriciale, questa trasformazione può essere espressa come:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix}$$

Nella notazione in coordinate omogenee è possibile riscrivere la trasformazione come:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

4.3.3 Scalatura

Scelto un punto C di riferimento (*punto fisso*), *scalare* una primitiva geometrica significa riposizionare tutti i suoi punti $P = (x, y, z)$, rispetto a C, in accordo ai *fattori di scala* $s = (s_x, s_y, s_z)$.

In base ai valori assegnati ai fattori di scala, se $s_i < 1$ le coordinate dei punti dell'oggetto lungo l'asse i vengono avvicinate al punto di riferimento, se $s_i > 1$ vengono invece allontanate. Inoltre, se i fattori di scala non sono uguali, le proporzioni dell'oggetto non sono mantenute e l'operazione prende il nome di *scalatura non uniforme*, altrimenti, se le proporzioni sono mantenute, si parla di *scalatura uniforme*.

Supponendo che il punto fisso sia l'origine $O = (0, 0, 0)$, la scalatura può essere espressa in forma matriciale come:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Nella notazione in coordinate omogenee è possibile riscrivere la trasformazione come:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

4.3.4 Rotazione

Fissato un punto C di riferimento (detto *pivot*) e un verso di rotazione (orario o antiorario), *ruotare* una primitiva geometrica attorno a C significa spostare di un angolo θ tutti i suoi punti nel verso assegnato, in maniera che per ognuno di essi si conservi la distanza da C.

Le rotazioni tridimensionali attorno ai tre assi cartesiani possono essere espresse nella forma generale:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = R(\theta) \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

dove la matrice $R(\theta)$ varia a seconda dell'asse attorno al quale viene effettuata la rotazione:

$$\underset{\text{(asse } x\text{)}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}} \quad \underset{\text{(asse } y\text{)}}{\begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}} \quad \underset{\text{(asse } z\text{)}}{\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}}$$

Come per le altre trasformazioni, anche le rotazioni possono essere espresse in coordinate omogenee, assumendo la forma:

$$\underset{\text{(asse x)}}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \quad \underset{\text{(asse y)}}{\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \quad \underset{\text{(asse z)}}{\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}$$

4.3.5 Composizione di trasformazioni

Una delle principali potenzialità della rappresentazione in coordinate omogenee è la possibilità di comporre trasformazioni mediante la moltiplicazione di matrici. Se f_1 e f_2 sono due trasformazioni affini con matrici M_1 e M_2, allora la trasformazione composta è:

$$f(\mathbf{x}) = M_2 M_1 \mathbf{x}$$

È importante notare che l'ordine di applicazione delle trasformazioni influisce sul risultato finale, poiché la moltiplicazione di matrici non è commutativa. In particolare, l'ordine di applicazione è determinato dalla vicinanza della matrice al vettore: la matrice più vicina al vettore viene applicata per prima.

4.4 Interpolazione

L'interpolazione è una tecnica matematica che, per una funzione discreta, consente di stimare valori intermedi non definiti dalla funzione a partire da un insieme finito di valori noti definiti dalla funzione. Questo processo è cruciale in computer grafica, dove molte proprietà (colori, texture, normali) sono definite solo in punti discreti e devono invece essere calcolate per ogni punto del rendering.

4.4.1 Interpolazione lineare

Sia data una funzione $f : [x_0, x_1] \to \mathbb{R}$ definita in due punti x_0 e x_1, con valori $f(x_0) = y_0$ e $f(x_1) = y_1$. L'*interpolazione lineare* permette di calcolare un valore approssimato $f(x)$ per $x \in [x_0, x_1]$ tramite la formula:

$$f(x) \approx y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$$

Definendo il parametro adimensionale $t = \frac{x - x_0}{x_1 - x_0} \in [0, 1]$, la formula si riscrive nella forma compatta:

$$\text{lerp}(y_0, y_1, t) = (1 - t)y_0 + ty_1$$

4.4.2 Interpolazione bilineare

Per funzioni di due variabili, l'interpolazione bilineare estende la linearità su due dimensioni. Siano dati quattro punti $(x_0, y_0), (x_1, y_0), (x_0, y_1), (x_1, y_1)$, con valori corrispondenti $f_{00}, f_{10}, f_{01}, f_{11}$. Il valore interpolato in un punto interno (x, y) è:

$$f(x, y) = (1 - t)(1 - s)f_{00} + t(1 - s)f_{10} + (1 - t)sf_{01} + tsf_{11}$$

dove:

$$t = \frac{x - x_0}{x_1 - x_0} \quad \text{(interpolazione orizzontale)} \qquad s = \frac{y - y_0}{y_1 - y_0} \quad \text{(interpolazione verticale)}$$

4.4.3 Interpolazione baricentrica

L'interpolazione baricentrica è utilizzata per stimare valori all'interno di un triangolo in $\mathbb{R}^2$ a partire dai valori noti nei suoi vertici. Sia dato un triangolo con vertici V_0, V_1, V_2 e valori associati f_0, f_1, f_2. Per un punto P interno al triangolo, si definiscono le *coordinate baricentriche* (u, v, w) rispetto ai vertici tali che:

$$P = uV_0 + vV_1 + wV_2, \quad \text{con} \quad u + v + w = 1, \quad u, v, w \geq 0$$

Il valore interpolato $f(P)$ è allora espresso come combinazione lineare dei valori dei vertici pesata dalle coordinate baricentriche:

$$f(P) = uf_0 + vf_1 + wf_2$$

4.5 Derivate e gradiente

Le derivate e il gradiente sono strumenti chiave in computer grafica per analizzare variazioni locali di funzioni scalari, come intensità di colore, altezza di un terreno o valori di rumore procedurale. Conoscere la direzione e il tasso massimo di variazione permette di calcolare pendenze, normali e ottimizzare l'interpolazione nelle superfici 3D.

4.5.1 Derivata in una dimensione

Si consideri una funzione scalare $f : \mathbb{R} \to \mathbb{R}$ definita su un intervallo reale. La *derivata* di f in un punto $x \in \mathbb{R}$, indicata con $f'(x)$, misura la variazione istantanea del valore della funzione al variare della variabile indipendente x:

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Dal punto di vista geometrico, $f'(x)$ rappresenta la pendenza della retta tangente al grafico di f nel punto $(x, f(x))$, ossia la velocità con cui f cresce o decresce localmente.

4.5.2 Derivate parziali in più dimensioni

Si consideri ora una funzione scalare $f : \mathbb{R}^n \to \mathbb{R}$, che associa a ogni punto $\mathbf{x} \in \mathbb{R}^n$ un valore reale $f(\mathbf{x}) \in \mathbb{R}$. Per funzioni di più variabili, ad esempio $f : \mathbb{R}^2 \to \mathbb{R}$, il concetto di derivata in una dimensione va estesa alle *derivate parziali*. Queste descrivono la variazione di f rispetto a una singola variabile, mantenendo costanti le altre. In un punto (x, y) si definiscono:

$$\frac{\partial f}{\partial x}(x, y) = \lim_{h \to 0} \frac{f(x + h, y) - f(x, y)}{h} \qquad \frac{\partial f}{\partial y}(x, y) = \lim_{k \to 0} \frac{f(x, y + k) - f(x, y)}{k}$$

In generale, per $f : \mathbb{R}^n \to \mathbb{R}$ e $\mathbf{x} = (x_1, \ldots, x_n)$, si ha che:

$$\frac{\partial f}{\partial x_i}(\mathbf{x}) = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_n) - f(\mathbf{x})}{h}$$

Se tutte le derivate parziali esistono e variano in modo continuo, la funzione f è detta *differenziabile*. In tal caso, ad esempio con $n = 2$, il suo grafico risulta localmente approssimabile da un piano tangente (o un iperpiano, in caso di $n > 2$) alla superficie di equazione $z = f(x, y)$. Le derivate parziali forniscono le pendenze di questo piano lungo le direzioni degli assi x e y.

4.5.3 Gradiente

Definizione

Il *gradiente* di una funzione $f : \mathbb{R}^n \to \mathbb{R}$ in un punto $\mathbf{x} \in \mathbb{R}^n$ è il vettore in $\mathbb{R}^n$ formato da tutte le sue derivate parziali:

$$\nabla f(x, y) = \left(\frac{\partial f}{\partial x}(x, y), \frac{\partial f}{\partial y}(x, y) \right)$$

In generale, per una data dimensione $n > 2$ il gradiente si estende a:

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{x}), \frac{\partial f}{\partial x_2}(\mathbf{x}), \ldots, \frac{\partial f}{\partial x_n}(\mathbf{x}) \right)$$

Proprietà

Il gradiente possiede proprietà geometriche di grande rilevanza:

- *Direzione di massima crescita*: il vettore $\nabla f(\mathbf{x})$ indica la direzione lungo la quale la funzione f cresce più rapidamente nel punto $\mathbf{x}$.
- *Modulo*: la norma $\|\nabla f(\mathbf{x})\|$ rappresenta la massima velocità di variazione della funzione. Ciò significa che spostandosi dal punto $\mathbf{x}$ nella direzione del gradiente, f aumenta più rapidamente che in qualsiasi altra direzione.
- *Ortogonalità alle curve di livello*: il gradiente è perpendicolare alle *curve di livello* $\{(x, y) : f(x, y) = c\}$ in $\mathbb{R}^2$ e, più in generale, alle *superfici di livello* $\{\mathbf{x} \in \mathbb{R}^n : f(\mathbf{x}) = c\}$ (che costituiscono le regioni geometriche dove la funzione risulta costante).

4.6 Curve parametriche

Le curve parametriche forniscono una descrizione continua di percorsi e superfici nello spazio e sono utilizzate per modellare geometrie fluide e animazioni. Spline come Catmull-Rom o B-spline permettono di generare forme lisce e controllabili, fondamentali per funzioni grafiche quali tessellazione, LOD e geometria procedurale.

4.6.1 Definizione

In geometria, una *curva parametrica* nello spazio tridimensionale è una funzione vettoriale che associa a un parametro reale t, appartenente a un intervallo $[a, b] \subseteq \mathbb{R}$, un punto nello spazio euclideo $\mathbb{R}^3$. Formalmente, si definisce come:

$$C(t) = (x(t), y(t), z(t)) \quad \text{con } t \in [a, b]$$

dove $x(t), y(t), z(t)$ (le *componenti parametriche* della curva) sono funzioni reali continue che descrivono le coordinate cartesiane del punto al variare di t. Il parametro t rappresenta l'andamento della curva e consente di descrivere forme che non sarebbero rappresentabili come funzioni cartesiane tradizionali (come cerchi, spirali o curve complesse).

4.6.2 Continuità

Quando due segmenti di curva si incontrano in un punto (detto *punto di contatto*), si parla di *continuità* per indicare quanto il passaggio da un segmento all'altro sia regolare. La continuità può essere classificata in diversi livelli:

- *Continuità C^0*: si dice che due segmenti di curva hanno continuità C^0 quando si uniscono ad un estremo senza alcun salto. In altre parole, le due curve condividono il punto di contatto P_0, garantendo che la curva risultante sia connessa.
- *Continuità parametrica C^1*: considerando il punto di contatto P_0 e denotando con v_1 e v_2 le derivate prime dei due segmenti di curva nel punto P_0, si dice che la curva ha continuità parametrica C^1 se la direzione e il modulo dei vettori tangenti sono uguali. Questo assicura che la curva sia liscia, senza la presenza di cambiamenti improvvisi della tangente lungo la curva.
- *Continuità geometrica G^1*: due segmenti di curva hanno continuità geometrica G^1 se le direzioni dei vettori tangenti coincidono nel punto di contatto, anche se i moduli possono differire. In questo caso la curva appare visivamente continua, ma la tangente lungo la curva può variare tra i due segmenti.
- *Continuità parametrica C^n*: la continuità parametrica di ordine n richiede che tutte le derivate fino all'ordine n dei due segmenti coincidano nel punto di contatto, cioè:

$$C_1^{(k)}(P_0) = C_2^{(k)}(P_0) \quad k = 1, \ldots, n$$

4.6.3 Curve interpolanti e approssimanti

Quando le curve parametriche sono utilizzate per modellare forme geometriche a partire da insiemi discreti di punti, detti *punti di controllo*, esse possono essere classificate in base al modo in cui trattano i punti di controllo, dividendosi in due gruppi principali: *interpolanti*, se passano esattamente per i punti dati, o *approssimanti*, se li seguono senza necessariamente attraversarli.

Curve interpolanti

Una curva parametrica $\mathbf{C}(t) = (x(t), y(t), z(t))$ è detta *interpolante* rispetto a un insieme di punti $\{P_0, P_1, \ldots, P_n\}$ (detti *punti di controllo*) se essa soddisfa la condizione di interpolazione:

$$\mathbf{C}(t_i) = P_i, \quad i = 0, \ldots, n$$

dove $\{t_0, t_1, \ldots, t_n\}$ è un insieme di parametri associati ai punti di controllo.

In termini matematici, l'interpolazione può essere formulata separatamente per ciascuna coordinata della curva:

$$x(t_i) = x_i, \quad y(t_i) = y_i, \quad z(t_i) = z_i, \quad i = 0, \ldots, n$$

Una possibile costruzione consiste nell'esprimere ciascuna funzione come combinazione lineare di funzioni base $\{\varphi_0(t), \ldots, \varphi_n(t)\}$:

$$x(t) = \sum_{i=0}^{n} \alpha_i \varphi_i(t), \quad y(t) = \sum_{i=0}^{n} \beta_i \varphi_i(t), \quad z(t) = \sum_{i=0}^{n} \gamma_i \varphi_i(t)$$

scegliendo i coefficienti $\alpha_i, \beta_i, \gamma_i$ in modo che vengano soddisfatte le condizioni di interpolazione:

$$x(t_i) = x_i, \quad y(t_i) = y_i, \quad z(t_i) = z_i, \quad i = 0, \ldots, n$$

In questo modo, la costruzione della curva interpolante si riduce a risolvere tre problemi di interpolazione scalare, uno per ciascuna coordinata.

Curve approssimanti

Una curva parametrica $\mathbf{C}(t) = (x(t), y(t), z(t))$ è detta *approssimante* rispetto a un insieme di punti $\{P_0, P_1, \ldots, P_n\}$ (detti *punti di controllo*) se essa non necessariamente passa per tutti i punti dati, ma ne definisce la forma generale in maniera controllata. In termini formali, una curva approssimante è una curva generata mediante combinazioni lineari di funzioni base, dove i punti di controllo influenzano la forma della curva senza costringerla ad attraversarli.

Matematicamente, ogni coordinata della curva può essere espressa come combinazione lineare di funzioni base $\{\varphi_0(t), \ldots, \varphi_n(t)\}$:

$$x(t) = \sum_{i=0}^{n} \alpha_i \varphi_i(t), \quad y(t) = \sum_{i=0}^{n} \beta_i \varphi_i(t), \quad z(t) = \sum_{i=0}^{n} \gamma_i \varphi_i(t),$$

dove i coefficienti $\alpha_i, \beta_i, \gamma_i$ sono determinati in modo da ottenere la migliore approssimazione dei punti di controllo secondo criteri di continuità e regolarità.

A differenza delle curve interpolanti, quindi, le curve approssimanti consentono un maggiore controllo sulla forma globale e sulla regolarità della curva, sacrificando il passaggio esatto per i punti di controllo.

4.6.4 Curve di Hermite

Le *curve di Hermite* sono un tipo particolare di curve parametriche cubiche interpolanti definite nello spazio tridimensionale. Una *curva parametrica cubica* è una curva $\mathbf{C}(t) = (x(t), y(t), z(t))$ in cui ciascuna coordinata è una funzione polinomiale di terzo grado del parametro t.

Polinomio interpolatore di Hermite

Un *polinomio interpolatore di Hermite* è un polinomio cubico che descrive un tratto di curva costruito in modo da interpolare due valori dati $f_0, f_1 \in \mathbb{R}$ e le rispettive derivate $d_0, d_1 \in \mathbb{R}$ in due punti $t_0, t_1 \in [0, 1]$. In altre parole, il polinomio identifica un segmento di curva che passa esattamente per i valori dati e allinea la sua pendenza alle estremità secondo le derivate specificate.

Il polinomio interpolatore di Hermite si scrive come:

$$P_H(t) = \varphi_0(t)f_0 + \varphi_1(t)d_0 + \psi_0(t)f_1 + \psi_1(t)d_1$$

dove, per un generico $t \in [0, 1]$, le funzioni base cubiche sono definite da:

$$\varphi_0(t) = 2t^3 - 3t^2 + 1, \quad \varphi_1(t) = t^3 - 2t^2 + t, \quad \psi_0(t) = -2t^3 + 3t^2, \quad \psi_1(t) = t^3 - t^2$$

Le funzioni φ_0 e ψ_0 garantiscono che il polinomio assuma i valori f_0 e f_1 agli estremi, mentre φ_1 e ψ_1 modulano la tangente, assicurando che le derivate siano rispettate. In particolare, si ha:

$$P_H(t_0) = f_0, \quad P_H(t_1) = f_1, \quad P'_H(t_0) = d_0, \quad P'_H(t_1) = d_1,$$

il che permette di concatenare più polinomi interpolatori di Hermite senza creare discontinuità nella derivata prima, ottenendo curve continue e regolari.

Curve di Hermite

Un *segmento di curva cubica di Hermite* nello spazio tridimensionale è la naturale estensione del polinomio interpolatore di Hermite al caso vettoriale. Invece di due valori scalari f_0, f_1 e delle loro derivate d_0, d_1, si considerano due punti di controllo $P_0, P_1 \in \mathbb{R}^3$ e i corrispondenti vettori tangenti $T_0, T_1 \in \mathbb{R}^3$. In questo modo il segmento si esprime come:

$$\mathbf{C}(t) = \varphi_0(t)P_0 + \varphi_1(t)T_0 + \psi_0(t)P_1 + \psi_1(t)T_1, \quad t \in [0, 1]$$

dove i punti di controllo P_0, P_1 definiscono le estremità del segmento, mentre i vettori tangenti T_0, T_1 controllano la direzione e la pendenza della curva nei punti di controllo. Le funzioni base di Hermite $\varphi_0, \varphi_1, \psi_0, \psi_1$ garantiscono interpolazione e continuità C^1 tra segmenti adiacenti.

Dato un insieme di punti di controllo da interpolare $\{P_0, \ldots, P_n\} \subset \mathbb{R}^3$, con $P_i = (x_i, y_i, z_i)$, e le rispettive tangenti $\{T_0, \ldots, T_n\} \subset \mathbb{R}^3$, con $T_i = (T_i^x, T_i^y, T_i^z)$, la curva di Hermite complessiva $\mathbf{C}(t) = (C_x(t), C_y(t), C_z(t))$ si ottiene concatenando segmenti Hermite tra punti consecutivi P_i e P_{i+1}, con interpolazione esatta delle tangenti, garantendo continuità C^1 lungo tutta la curva.

Ogni segmento tra P_i e P_{i+1} è definito da:

$$\mathbf{C}_i(t) = \varphi_0(s)P_i + \varphi_1(s)T_i + \psi_0(s)P_{i+1} + \psi_1(s)T_{i+1}, \quad s \in [0, 1],$$

dove s è la normalizzazione del parametro t sull'intervallo $[t_i, t_{i+1}]$:

$$s = \frac{t - t_i}{t_{i+1} - t_i}, \quad t \in [t_i, t_{i+1}] \subset \mathbb{R}$$

Esplicitando le componenti della curva in coordinate cartesiane, otteniamo:

$$C_x(t) = \sum_{i=0}^{n-1} \phi_{[t_i,t_{i+1}]}(t) \Big[\varphi_0(s)x_i + \varphi_1(s)T_i^x + \psi_0(s)x_{i+1} + \psi_1(s)T_{i+1}^x \Big]$$
$$C_y(t) = \sum_{i=0}^{n-1} \phi_{[t_i,t_{i+1}]}(t) \Big[\varphi_0(s)y_i + \varphi_1(s)T_i^y + \psi_0(s)y_{i+1} + \psi_1(s)T_{i+1}^y \Big]$$
$$C_z(t) = \sum_{i=0}^{n-1} \phi_{[t_i,t_{i+1}]}(t) \Big[\varphi_0(s)z_i + \varphi_1(s)T_i^z + \psi_0(s)z_{i+1} + \psi_1(s)T_{i+1}^z \Big]$$

dove $\phi_{[t_i,t_{i+1}]}(t)$ è la *funzione indicatrice* dell'intervallo $[t_i, t_{i+1}]$, che assicura che solo il segmento corrispondente contribuisca per ogni $t \in [t_0, t_n]$:

$$\phi_{[t_i,t_{i+1}]}(t) = \begin{cases} 1, & \text{se } t \in [t_i,\, t_{i+1}] \\ 0, & \text{altrimenti} \end{cases}$$

4.6.5 Catmull-Rom Spline

Le *Catmull-Rom spline* sono una classe di curve parametriche cubiche interpolanti definite nello spazio tridimensionale. Dato un insieme di punti di controllo $\{P_0, P_1, \ldots, P_n\} \subset \mathbb{R}^3$, una Catmull-Rom spline costruisce un segmento cubico tra ogni coppia di punti consecutivi P_i e P_{i+1} e ogni segmento può essere interpretato come un *segmento di curva di Hermite*:

$$\mathbf{C}_i(t) = \varphi_0(s)P_i + \varphi_1(s)T_i + \psi_0(s)P_{i+1} + \psi_1(s)T_{i+1}, \quad s \in [0, 1]$$

dove $\varphi_0, \varphi_1, \psi_0, \psi_1$ sono le funzioni base di Hermite e s è il parametro normalizzato sull'intervallo $[t_i, t_{i+1}]$.

E' importante sottolinare che ogni segmento di Catmull-Rom è però definito da quattro punti consecutivi $(P_{i-1}, P_i, P_{i+1}, P_{i+2})$, dove P_i e P_{i+1} sono i punti interpolati dal segmento, mentre P_{i-1} e P_{i+2} determinano la direzione delle tangenti ai punti interpolati e quindi la forma del segmento.

A differenza delle curve di Hermite infatti, le tangenti T_i e T_{i+1} non sono specificate manualmente, ma calcolate automaticamente dai punti adiacenti. In particolare, ciascuna tangente è parallela al segmento che congiunge il punto precedente con quello successivo:

$$T_i = \frac{1}{2}(P_{i+1} - P_{i-1}) \quad T_{i+1} = \frac{1}{2}(P_{i+2} - P_i)$$

Sostituendo queste tangenti nella formulazione di Hermite, la curva può essere riscritta come polinomio cubico in $t \in [0, 1]$:

$$\mathbf{C}(t) = \frac{1}{2}\Big[(2P_1) + (-P_0 + P_2)t + (2P_0 - 5P_1 + 4P_2 - P_3)t^2 + (-P_0 + 3P_1 - 3P_2 + P_3)t^3\Big]$$

In generale, le Catmull-Rom spline garantiscono continuità C^1 lungo tutta la spline e un importante *controllo locale* della curva dove la modifica di un punto di controllo influisce principalmente sui segmenti adiacenti.

Capitolo 5

Progetto

In questo capitolo viene descritta l'organizzazione complessiva del progetto, con particolare attenzione agli aspetti pratici che ne hanno guidato la realizzazione.

Saranno analizzati, in primo luogo, lo scopo del progetto, le tecnologie adottate e la struttura generale del sistema. Successivamente verranno descritti i meccanismi di gestione della telecamera e le modalità di interazione con l'utente. Si passerà quindi all'organizzazione delle geometrie impiegate per la costruzione degli oggetti e all'inserimento di un personaggio animato importato da file esterni. Seguiranno la trattazione dei buffer di memoria utilizzati per il passaggio dei dati e l'approfondimento delle tecniche di rumore procedurale. Verranno poi analizzati la gestione delle texture e i sistemi di rilevamento delle collisioni. Infine, sarà dedicata particolare attenzione all'utilizzo degli shader, già introdotti nei capitoli precedenti, con un focus sul loro ruolo nella gestione dinamica del livello di dettaglio.

Per mantenere la trattazione chiara e discorsiva, si è scelto di riportare solo alcuni estratti di codice, selezionati come esempi significativi, evitando di appesantire il testo con sequenze di codice troppo dettagliate.

5.1 Introduzione

5.1.1 Scopo del progetto

Il progetto mira ad utilizzare gli shader standard (Vertex Shader e Fragment Shader) e gli shader opzionali (Tessellation Shaders e Geometry Shaders), descritti nei capitoli precedenti, per la realizzazione di due ambientazioni paesaggistiche, composte da elementi di scena e da un personaggio animato controllabile.

In particolare, il progetto sfrutta l'utilizzo dei nuovi shader opzionali per dettagliare in maniera dinamica (LOD dinamico) ogni oggetto presente nella scena, in base alla posizione della telecamera virtuale o a quella del personaggio animato.

5.1.2 Tecnologie utilizzate

Linguaggio C/C++

Il linguaggio di programmazione scelto per lo sviluppo del progetto è il C/C++, ampiamente utilizzato in ambito grafico grazie alla sua efficienza e al controllo diretto sulla memoria. Queste caratteristiche permettono di gestire in maniera precisa le strutture dati e le risorse grafiche, garantendo prestazioni elevate. La scelta di questo linguaggio è inoltre dovuta al fatto che la maggior parte delle librerie moderne per la computer grafica forniscono interfacce sviluppate per C e C++.

OpenGL

Come già accennato, OpenGL (*Open Graphics Library*) è una libreria open source e multipiattaforma che definisce un'API per la grafica 2D e 3D. È gestita dal *Khronos Group* ed è ampiamente supportata nei sistemi operativi moderni, poichè permette di utilizzare la potenza di calcolo delle GPU per eseguire rendering in tempo reale. Nel progetto è stata utilizzata nella sua versione più recente, la 4.6, che permette l'utilizzo di funzionalità avanzate per le pipeline grafiche programmabili e l'utilizzo degli shader più avanzati, come i Tessellation Shaders e il Geometry Shader.

GLSL

GLSL (*OpenGL Shading Language*) è il linguaggio di shading associato a OpenGL, che presenta una sintassi simile a quella del C. Questo linguaggio è progettato per scrivere programmi eseguiti direttamente sulla GPU (chiamati *shader*), che permettono di controllare in maniera flessibile la pipeline grafica. Nel progetto sono stati utilizzati diversi tipi di shader, tra cui *Vertex*, *Tessellation*, *Geometry* e *Fragment Shader*, con l'obiettivo di realizzare un rendering dettagliato e dinamico.

Glad

Glad (*OpenGL Loading Library*) è una libreria fondamentale per lo sviluppo di applicazioni in OpenGL. La sua funzione principale consiste nel caricare automaticamente, al momento dell'esecuzione del programma, tutte le funzioni di OpenGL necessarie per l'applicazione. In questo modo, Glad permette di verificare quali estensioni siano disponibili su un determinato sistema operativo, garantendo la portabilità e il corretto funzionamento dell'applicazione su diverse piattaforme.

GLFW

GLFW è una libreria open source multipiattaforma progettata per semplificare lo sviluppo di applicazioni grafiche basate su OpenGL. Fornisce un'interfaccia che permette la creazione di finestre grafiche personalizzate, la gestione di eventi come la chiusura della finestra o il suo ridimensionamento, e la creazione di *contesti grafici*, ossia l'ambiente necessario per eseguire le operazioni grafiche. Inoltre, GLFW si occupa di gestire l'interazione con il sistema operativo sottostante, rendendo più semplice l'input da tastiera, mouse e altri dispositivi.

GLM

GLM (*OpenGL Mathematics*) è una libreria di funzioni matematiche progettate appositamente per applicazioni grafiche. È scritta interamente in C++ e segue le stesse convenzioni utilizzate in GLSL. Questa libreria fornisce strumenti per gestire vettori e matrici, indispensabili per operazioni come trasformazioni geometriche, calcolo di proiezioni prospettiche e gestione di rotazioni nello spazio tridimensionale.

Assimp

Assimp (*Open Asset Import Library*) è una libreria che consente di importare una vasta gamma di formati di file 3D, come ad esempio OBJ e FBX. L'obiettivo di questa libreria è fornire una rappresentazione unificata dei modelli caricati, senza che lo sviluppatore debba preoccuparsi delle differenze fra i vari formati. Oltre al caricamento della geometria, Assimp supporta anche l'importazione di materiali, texture e dati per l'animazione scheletrica.

Dear ImGui

Dear ImGui è una libreria grafica che implementa il paradigma delle *Immediate Mode GUI*. A differenza delle interfacce utente tradizionali (basate su oggetti persistenti), con ImGui ogni elemento grafico dell'interfaccia viene ridisegnato a ogni frame in maniera immediata. Questo approccio semplifica notevolmente la creazione di pannelli di debug, strumenti di ispezione e interfacce di controllo integrate nel programma. Nel progetto è stata utilizzata per implementare un'interfaccia grafica di supporto, utile a monitorare e modificare parametri durante l'esecuzione.

5.1.3 Struttura generale

Il progetto è stato organizzato secondo un'architettura modulare, in cui ogni componente è stato sviluppato come unità indipendente con responsabilità ben definite. Questo approccio segue il principio della *Single Responsibility*, secondo cui ciascun modulo deve occuparsi di un compito specifico e non sovrapporsi alle funzioni degli altri.

In pratica, per ogni sezione descritta nei paragrafi successivi è stato sviluppato un modulo dedicato, così da separare in maniera chiara la logica di gestione dei vari aspetti del progetto. Questa suddivisione facilita la leggibilità e la manutenzione del codice, oltre a permettere eventuali modifiche o estensioni senza dover intervenire sull'intero progetto. Per quanto riguarda gli shader, questi sono stati organizzati in moduli distinti, uno per ciascun tipo di shader della pipeline grafica (vertex, tessellation, geometry, fragment), rispecchiando così la struttura della pipeline di rendering.

5.2 Telecamera

La telecamera virtuale rappresenta uno degli elementi fondamentali nelle applicazioni di computer grafica, poichè solo grazie ad essa è possibile determinare quale porzione di scena risulterà visibile all'osservatore e che quindi dovrà essere proiettata sullo schermo. La telecamera, infatti, definisce il punto di vista e l'orientamento dell'osservatore virtuale nello spazio tridimensionale.

Come già spiegato, la *View Transformation* si occupa di trasformare le coordinate dei punti dal sistema di riferimento del mondo (*World Space*) a quello relativo alla posizione e alla direzione della telecamera (*View Space*).

5.2.1 Concetti fondamentali

Dal punto di vista teorico, per descrivere una telecamera virtuale è necessario definire un oggetto caratterizzato da una posizione, una direzione di vista e due assi (u e v) che puntano rispettivamente verso la destra e verso l'alto della telecamera. Questi tre elementi tra loro ortogonali costituiscono il sistema di riferimento locale della telecamera.

La posizione della telecamera è rappresentata da un vettore espresso nel sistema di coordinate del mondo e identifica il punto in cui si trova l'osservatore. La direzione di vista è data dal vettore che si ottiene sottraendo al vettore posizione del *target* (il punto verso cui la telecamera è orientata) il vettore posizione della telecamera stessa. Per convenzione in OpenGL, l'osservatore guarda lungo l'asse z negativo. L'asse u, che punta a destra della telecamera, viene calcolato come prodotto vettoriale tra il vettore direzione e il vettore VUP, ossia il vettore definito nello spazio del mondo, che punta verso l'alto. Una volta determinato u, è possibile calcolare l'asse v (rivolto verso l'alto nel sistema della telecamera) come prodotto vettoriale tra u e la direzione della telecamera. In questo modo si ottiene una terna che descrive univocamente l'orientamento della telecamera.

Da tali vettori è possibile costruire la matrice di vista, che converte coordinate espresse nel sistema di riferimento del mondo in coordinate nel sistema di riferimento della telecamera. Poiché la trasformazione di base è però quella che porta dallo spazio della telecamera a quello del mondo, la matrice calcolata deve essere invertita per poter essere applicata correttamente nella pipeline di rendering.

A livello implementativo, la libreria GLM fornisce la funzione `lookAt`, che consente di costruire la matrice di vista a partire dalla posizione della telecamera, dal vettore che ne definisce l'orientamento verso l'alto e dal punto target.

La telecamera viene descritta tramite la struttura `ViewSetup`, che contiene i campi essenziali per definire i suoi parametri:

```
1 typedef struct {
2     vec3 position; // Posizione della camera nello spazio 3D
3     vec3 target; // Punto verso cui punta la camera
4     vec3 upVector; // Direzione verso l'alto della camera
```

```
    vec3 direction; // Direzione di vista della camera
} ViewSetup;
```

5.2.2 Proiezione prospettica

La telecamera non si limita a definire il sistema di riferimento locale per l'osservatore, ma deve anche gestire la **proiezione** della scena sul piano di vista dell'utente. In computer grafica, come già accennato, vengono tipicamente utilizzati due tipi di proiezione: quella ortogonale, che mantiene i parallelismi e i rapporti fra le distanze, e quella prospettica, che cerca di riprodurre il modo in cui l'occhio umano percepisce la realtà, secondo cui gli oggetti più lontani appaiono più piccoli e quelli più vicini appaiono più grandi.

Come tipo di prospettiva, all'interno del progetto, è stata scelta la proiezione prospettica. Questa è definita da quattro parametri principali: il campo visivo verticale ($fovY$) che indica l'ampiezza del cono visivo espresso in gradi, il *rapporto d'aspetto* (*aspect ratio*) che corrisponde al rapporto tra larghezza e altezza della finestra di visualizzazione, il *piano di clipping vicino* (*near plane*) che rappresenta la distanza minima alla quale un oggetto deve trovarsi per poter essere visualizzato e il *piano di clipping lontano* (*far plane*) che rappresenta invece la distanza massima oltre la quale gli oggetti vengono eliminati dal rendering.

Anche in questo caso, GLM fornisce la funzione `perspective`, che costruisce automaticamente la matrice di proiezione prospettica a partire da questi valori.

Per rappresentare i parametri della proiezione viene utilizzata la struttura `PerspectiveSetup`:

```
typedef struct {
    float fovY;        // Campo visivo verticale (in gradi)
    float aspect;      // Rapporto tra larghezza e altezza del viewport
    float near_plane;  // Piano di clipping vicino
    float far_plane;   // Piano di clipping lontano
} PerspectiveSetup;
```

5.2.3 Movimento della telecamera

Oltre a definire posizione, orientamento e proiezione, è fondamentale poter gestire il movimento della telecamera all'interno della scena, così da consentire all'osservatore di esplorare l'ambiente 3D.

La logica di movimento implementata permette di traslare la telecamera lungo i tre assi principali: avanti/indietro, destra/sinistra e su/giù. Questi movimenti vengono realizzati tramite un insieme di funzioni che aggiornano i vettori `position` e `target` della struttura `ViewSetup` in base alla direzione di movimento desiderata e alla velocità di spostamento della telecamera, espressa dal parametro `cameraSpeed`.

Il controllo è gestito dalle funzioni `cameraUp`, `cameraDown`, `cameraLeft`, `cameraRight`, `cameraForward` e `cameraBack`. Di seguito vengono descritte le logiche di funzionamento dei tre gruppi pincipali di movimento, riportandone l'implementazione.

Movimenti verticali

Il movimento verticale viene realizzato sfruttando la relazione tra la direzione di vista e il vettore *up*. In particolare, si calcola dapprima un vettore laterale detto *slide vector*, ottenuto come prodotto vettoriale tra direzione e `up`. Questo vettore descrive lo spostamento orizzontale perpendicolare alla direzione di vista, cioè quello che la telecamera percepisce come destra/sinistra.

Successivamente, per ottenere lo spostamento verticale reale, si calcola il prodotto vettoriale tra la direzione e lo slide vector. In questo modo si ricava un vettore ortogonale sia alla direzione di vista che allo spostamento laterale, quindi perfettamente allineato con l'asse verticale percepito dalla telecamera (che non necessariamente coincide con l'asse globale y).

Nella funzione `cameraUp` tale vettore verticale viene sottratto alla posizione della telecamera e al target relativo, spostando la telecamera verso l'alto, mentre in `cameraDown` viene sommato, realizzando così la discesa.

```
1  void cameraUp(void) {
2      SetupTelecamera.direction = SetupTelecamera.target - SetupTelecamera.
       position;
3      slide_vector = normalize(cross(SetupTelecamera.direction,
       SetupTelecamera.upVector));
4      vec3 upDirection = cross(SetupTelecamera.direction, slide_vector) *
       cameraSpeed;
5      SetupTelecamera.position -= upDirection;
6      SetupTelecamera.target -= upDirection;
7  }
8
9  void cameraDown(void) {
10     SetupTelecamera.direction = SetupTelecamera.target - SetupTelecamera.
       position;
11     slide_vector = normalize(cross(SetupTelecamera.direction,
       SetupTelecamera.upVector));
12     vec3 upDirection = cross(SetupTelecamera.direction, slide_vector) *
       cameraSpeed;
13     SetupTelecamera.position += upDirection;
14     SetupTelecamera.target += upDirection;
15 }
```

Movimenti orizzontali

Il movimento laterale sfrutta direttamente lo *slide vector* definito in precedenza. In `cameraLeft`, questo vettore viene sottratto alla posizione della telecamera e al target

relativo, in modo da ottenere lo spostamento a sinistra. La funzione `cameraRight` si comporta allo stesso modo, ma aggiunge lo slide vector, realizzando il movimento verso destra.

```
void cameraLeft(void) {
    SetupTelecamera.direction = SetupTelecamera.target - SetupTelecamera.
    position;
    slide_vector = cross(SetupTelecamera.direction, SetupTelecamera.
    upVector) * cameraSpeed;
    SetupTelecamera.position -= slide_vector;
    SetupTelecamera.target -= slide_vector;
}

void cameraRight(void) {
    SetupTelecamera.direction = SetupTelecamera.target - SetupTelecamera.
    position;
    slide_vector = cross(SetupTelecamera.direction, SetupTelecamera.
    upVector) * cameraSpeed;
    SetupTelecamera.position += slide_vector;
    SetupTelecamera.target += slide_vector;
}
```

Movimenti longitudinali

Infine, il movimento lungo la direzione di vista sfrutta direttamente il vettore direzione della telecamera, senza dover calcolare alcun nuovo vettore. La funzione `cameraForward` avanza lungo questa direzione, sommano la direzione (scalata tramite la velocità) alla posizione e si aggiorna il target di conseguenza. Al contrario, la funzione `cameraBack` sottrae la direzione, permettendo di arretrare nella scena.

```
void cameraForward(void) {
    SetupTelecamera.direction = SetupTelecamera.target - SetupTelecamera.
    position;
    SetupTelecamera.position += SetupTelecamera.direction * cameraSpeed;
    SetupTelecamera.target = SetupTelecamera.position + SetupTelecamera.
    direction;
}

void cameraBack(void) {
    SetupTelecamera.direction = SetupTelecamera.target - SetupTelecamera.
    position;
    SetupTelecamera.position -= SetupTelecamera.direction * cameraSpeed;
    SetupTelecamera.target = SetupTelecamera.position + SetupTelecamera.
    direction;
}
```

5.3 Interazioni con l'utente

Un aspetto molto importante del progetto è dato dalla possibilità di interagire con esso attraverso i più comuni dispositivi di input messi a disposizione dell'utente: il mouse e la tastiera.

Per gestire correttamente tali input, la libreria GLFW mette a disposizione una serie di strumenti che permettono di ricevere ed elaborare gli eventi generati da questi dispositivi, come ad esempio la pressione di un tasto su mouse o tastiera, lo spostamento del cursore del mouse o lo scorrimento della sua rotella. Questi strumenti si basano sul meccanismo delle *callback functions*, cioè funzioni definite dal programmatore che vengono richiamate automaticamente da GLFW nel momento in cui si verifica un determinato evento.

Affinché GLFW sappia a quale funzione rivolgersi per la gestione di uno specifico evento, è necessario registrare all'inizio del programma le callback di interesse tramite le apposite funzioni di assegnazione. Ad esempio, per eventi generati dal mouse, `glfwSetCursorPosCallback` consente di associare una funzione alla gestione degli spostamenti del cursore, mentre `glfwSetScrollCallback` permette di registrare la funzione che gestirà lo scorrimento della rotella.

Ogni funzione di callback deve rispettare la signature prevista da GLFW, la quale specifica gli argomenti che vengono passati alla funzione al momento della chiamata. Tali argomenti contengono le informazioni utili all'elaborazione dell'evento, come la finestra coinvolta e i dati specifici legati al tipo di input.

Per la gestione del mouse, nel progetto sono state utilizzate principalmente due tipologie di funzioni di callback: una per la gestione del movimento del cursore e una per lo scorrimento della rotella del mouse.

La funzione associata allo spostamento del cursore (`cursor_position_callback`) gestisce l'orientamento della telecamera a partire dai movimenti del mouse. Ogni variazione di posizione del cursore viene tradotta in due offset, orizzontale e verticale, che vanno ad aggiornare rispettivamente gli angoli di rotazione della telecamera (possibili solo in senso orizzontale o in verticale). In questo modo l'utente può esplorare la scena liberamente, simulando il classico comportamento di una visuale in prima persona.

Per evitare effetti indesiderati, come la rotazione completa attorno all'asse verticale (che causerebbe l'inversione della visuale quando si guarda troppo verso l'alto o verso il basso), l'angolo verticale viene limitato ad un intervallo compreso tra $-89°$ e $+89°$. Infine, dai nuovi valori ottenuti viene calcolata la direzione della telecamera, che viene poi normalizzata e aggiornata all'interno della struttura di gestione della telecamera.

La funzione legata allo scroll (`scroll_callback`), invece, viene utilizzata per modificare il campo visivo verticale (`fovY`) della proiezione prospettica, simulando un effetto di zoom, avvicinando o allontanando la visuale senza alterare la posizione della telecamera.

Di seguito viene riportato il codice relativo all'implementazione di tali funzioni:

```
void cursor_position_callback(GLFWwindow* window, double xposIn, double
    yposIn) {
    if (!mouseLocked)
        return;

    int width, height;
    glfwGetFramebufferSize(window, &width, &height);

    float sensitivity = 0.05f;
    static bool firstCall = true;
    static float lastX = width / 2.0f;
    static float lastY = height / 2.0f;

    float xpos = float(xposIn);
    float ypos = float(yposIn);

    ypos = height - ypos;

    if (firstCall) {
        lastX = xpos;
        lastY = ypos;
        firstCall = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = ypos - lastY;

    lastX = xpos;
    lastY = ypos;

    xoffset *= sensitivity;
    yoffset *= sensitivity;

    Theta += xoffset;
    Phi += yoffset;

    if (Phi > 89.0f)
        Phi = 89.0f;
    if (Phi < -89.0f)
        Phi = -89.0f;

    vec3 newDirection;
    newDirection.x = cos(radians(Theta)) * cos(radians(Phi));
    newDirection.y = sin(radians(Phi));
    newDirection.z = sin(radians(Theta)) * cos(radians(Phi));
    SetupTelecamera.direction = normalize(newDirection);
    SetupTelecamera.target = SetupTelecamera.position + SetupTelecamera.
```

```
    direction;
}

void scroll_callback(GLFWwindow* window, double xoffset, double yoffset) {
    if (yoffset < 0)
        SetupProspettiva.fovY -= 1; //Rotella del mouse indietro
    else
        SetupProspettiva.fovY += 1;  //Rotella del mouse in avanti
}
```

Oltre alle funzioni di callback per la gestione del mouse, all'interno del progetto è stata inoltre definita una funzione `process_input`, che simula il comportamento di una normale funzione di callback, senza però esserlo realmente. A differenza delle callback fornite da GLFW, che vengono invocate automaticamente al verificarsi di un evento, questa funzione viene richiamata esplicitamente nel ciclo principale del programma per verificare lo stato corrente di mouse e tastiera ed elaborare di conseguenza le azioni da intraprendere.

Per quanto riguarda la gestione del mouse, la funzione si occupa di gestire la visibilità del cursore e la modalità di controllo della visuale mediante il tasto destro del mouse. Quando il cursore è visibile, la telecamera non segue i movimenti del mouse e l'utente può interagire liberamente con eventuali interfacce grafiche. Al contrario, quando il cursore è nascosto, i movimenti del mouse vengono interpretati come variazioni dell'orientamento della telecamera, permettendo un controllo diretto della scena.

Per quanto riguarda la gestione della tastiera, `process_input` associa ai tasti standard di movimento (`W`, `A`, `S`, `D`) lo spostamento della telecamera sul piano orizzontale, mentre i tasti `SPACE` e `LEFT SHIFT` consentono di muoversi rispettivamente verso l'alto e verso il basso, completando così un controllo tridimensionale. La funzione permette inoltre di abilitare o disabilitare il rendering in modalità wireframe tramite i tasti `L` e `F`, e di terminare il programma con il tasto `ESC`.

La stessa funzione gestisce infine anche il movimento del personaggio animato tramite le frecce direzionali, traducendo i comandi in spostamenti coerenti con le direzioni indicate (avanti, indietro, sinistra e destra). Contemporaneamente allo spostamento lineare, viene calcolato l'angolo di rotazione del modello in modo che l'orientamento del personaggio risulti sempre coerente con la direzione del moto.

La funzione `process_input` non è stata implementata come funzione di callback perchè non si limita a reagire ad un singolo evento, ma deve gestire in maniera continua lo stato dei dispositivi di input. Per questo motivo viene richiamata ad ogni iterazione del ciclo di rendering, così da garantire un controllo fluido e costante della scena e delle entità presenti in essa.

5.4 Geometrie

Come anticipato nei capitoli iniziali di questa tesi, il compito principale dell'applicazione lato CPU nei confronti della pipeline grafica è quello di definire la geometria

degli oggetti nella scena, andandone a calcolare le coordinate dei vertici e organizzandoli secondo il tipo di primitive che il resto della pipeline si aspetta di ricevere in ingresso.

In particolare, poiché il progetto utilizza gli shader di tessellazione e di geometria per arricchire e dettagliare ulteriormente le forme, le geometrie vengono definite lato CPU in maniera semplificata, con un numero ridotto di vertici che rappresentano la forma di base degli oggetti. Questo consente di demandare alla GPU il compito di generare la complessità finale, alleggerendo così il carico computazionale iniziale.

Di seguito vengono presentate le geometrie principali utilizzate, divise in base alle due ambientazioni realizzate.

5.4.1 Ambientazione 1: paesaggio montuoso

Terreno

Il terreno è stato definito come una griglia regolare di punti appartenenti ad un piano. La funzione `simplePlane` genera i vertici di questa griglia a partire da due parametri: la dimensione complessiva del piano e il numero di suddivisioni desiderate. In questo modo il piano viene suddiviso in celle quadrate, costituite ciascuna da quattro vertici.

Successivamente, tramite la funzione `generatePatches`, tali vertici vengono organizzati in gruppi da quattro, così da poter essere elaborati come patch quadrilaterali dalla pipeline. Ogni patch corrisponde a una singola cella della griglia e rappresenta l'unità di base che sarà poi sottoposta ai Tessellation Shaders e Geometry Shader, responsabili di aumentarne il dettaglio e deformarla per ottenere un terreno realistico.

```
vector<float> generatePatches(const vector<float>& plane, int division) {
    vector<float> patches;
    int rowLength = division + 1;

    for (int row = 0; row < division; ++row) {
        for (int col = 0; col < division; ++col) {
            int v0 = ((row + 1) * rowLength + col) * 3;
            int v1 = ((row + 1) * rowLength + col + 1) * 3;
            int v2 = (row * rowLength + col + 1) * 3;
            int v3 = (row * rowLength + col) * 3;

            patches.push_back(plane[v0]);
            patches.push_back(plane[v0 + 1]);
            patches.push_back(plane[v0 + 2]);

            patches.push_back(plane[v1]);
            patches.push_back(plane[v1 + 1]);
            patches.push_back(plane[v1 + 2]);

```

```
            patches.push_back(plane[v2]);
            patches.push_back(plane[v2 + 1]);
            patches.push_back(plane[v2 + 2]);

            patches.push_back(plane[v3]);
            patches.push_back(plane[v3 + 1]);
            patches.push_back(plane[v3 + 2]);
        }
    }

    return patches;
}
```

In particolare, si anticipa che, poiché il terreno è stato definito come completamente piatto, quindi privo di dislivelli lungo l'asse verticale, l'aspetto montuoso verrà generato successivamente attraverso l'utilizzo di una speciale mappa di altezza. Quest'ultima, descritta nel dettaglio nella sezione 5.7, permetterà di attribuire a ciascun vertice un valore di elevazione, così da ottenere un terreno irregolare e realistico una volta che la geometria sarà amplificata dagli shader.

Stelle

Per la generazione delle stelle si è scelto di partire da una rappresentazione sferica semplificata. Viene innanzitutto definito un insieme di direzioni corrispondenti ai sei assi principali dello spazio tridimensionale (`sphereCorners`). Combinando opportunamente questi punti attraverso la tabella `ottanteTriangles`, si ottengono otto triangoli che corrispondono agli ottanti di una sfera.

La funzione `generateSphericalBase` utilizza tali direzioni per costruire un insieme di vertici distribuiti su una superficie sferica di raggio prefissato, centrata in una posizione arbitraria. In questo modo si ottiene una base poligonale approssimata della sfera, che costituisce il punto di partenza per la generazione delle stelle nella scena.

```
vector<vec3> sphereCorners = {
    vec3(0,  1,  0),
    vec3(0,  -1,  0),
    vec3(0, 0,  1),
    vec3(0, 0,  -1),
    vec3(1,  0, 0),
    vec3(-1,  0, 0)
};

const int ottanteTriangles[8][3] = {
    {0, 2, 4}, // Ottante 1
    {0, 3, 4}, // Ottante 2
    {0, 3, 5}, // Ottante 3
    {0, 2, 5}, // Ottante 4
```

```
    {1, 2, 4}, // Ottante 5
    {1, 3, 4}, // Ottante 6
    {1, 3, 5}, // Ottante 7
    {1, 2, 5}  // Ottante 8
};

vector<vec3> generateSphericalBase(const vec3& center, float radius) {
    vector<vec3> verts;

    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 3; ++j) {
            vec3 dir = normalize(sphereCorners[ottanteTriangles[i][j]]);
            vec3 offset = dir * radius;
            verts.push_back(center + offset);
        }
    }

    return verts;
}
```

5.4.2 Ambientazione 2: paesaggio urbano

Terreno

Analogamente a quanto fatto per il paesaggio montuoso, il terreno di questa ambientazione è organizzato come una griglia regolare, ma con una suddivisione funzionale che permette di distinguere le aree stradali da quelle coperte di erba.

La funzione `roadAndGrass` genera i vertici della griglia e associa a ciascun punto un indicatore booleano che definisce se quel punto appartiene alla strada o all'erba circostante. La strada principale viene posizionata lungo l'asse centrale della griglia, formando una croce, e la sua larghezza è definita da un parametro della funzione, consentendone in questo modo la regolazione.

Successivamente, la funzione `generatePatches` organizza i vertici in patch quadrilaterali come già visto per il terreno precedente. Ciascun patch viene classificata fra patch di erba o patch di strada e viene calcolato un array di flag che identifica i bordi della patch che confinano con una patch di tipologia diversa. Queste informazioni saranno poi utilizzate dagli shader per applicare correttamente i materiali e generare gli effetti visivi di transizione tra strada e terreno verde.

Edifici e siepi

Per la rappresentazione degli edifici e delle siepi della scena urbana sono stati utilizzati dei blocchi a forma di parallelepipedo. La funzione `generateBlocks` si occupa di generare i vertici dei parallelepipedi a partire da un insieme di posizioni centrali.

Per ciascun blocco viene calcolata un'altezza casuale compresa in un intervallo prestabilito, in modo da conferire variazione e realismo agli edifici e alle siepi. Gli edi-

fici hanno dimensioni standard e altezza variabile, mentre le siepi vengono generate con una scala minore e proporzioni leggermente irregolari, ottenute introducendo casualità nella larghezza e nella profondità dei parallelepipedi.

Ogni blocco può poi essere suddiviso verticalmente in più segmenti, permettendo di ampliare ulteriormente il dettaglio a livello di shader senza incrementare eccessivamente il numero di vertici lato CPU.

La funzione `generatePatchesFromBlocks` organizza infine i vertici dei blocchi in patch quadrilaterali e calcola le normali delle facce. Le facce superiori e inferiori possono essere opzionalmente escluse nel caso degli edifici, poiché coperte dal tetto (generato separatamente) o a contatto col terreno, mentre le facce laterali vengono mantenute per garantire la corretta illuminazione e la resa dei materiali.

Tetti

I tetti sono generati separatamente rispetto ai blocchi che costituiscono i corpi degli edifici, in modo da poter avere una maggior possibilità di personalizzazione e per una gestione più precisa dell'organizzazione dei vertici delle geometrie.

La funzione `generateRoofs` costruisce i tetti come tronchi di piramide, con la base superiore leggermente ridotta rispetto a quella inferiore. Per ciascun tetto viene definita la posizione centrale, calcolata a partire dalla base superiore del blocco corrispondente e da un'altezza prefissata. La funzione suddivide inoltre ogni faccia in porzioni più piccole, così da aumentare il successivo dettaglio dinamico fornito dagli shader e rendere l'oggetto più realistico una volta applicate le texture.

Successivamente, la funzione `generatePatchesFromRoofs` organizza i vertici in patch quadrilaterali e calcola le normali per ciascuna faccia. Queste ultime sono orientate in modo coerente con la geometria, così che quelle relative alle basi puntino verso l'alto o verso il basso (a seconda che si tratti della base superiore o inferiore), mentre quelle relative ai lati inclinati siano orientate verso l'esterno.

Lampioni

Per generare i lampioni sono state utilizzate due funzioni, una per la geometria relativa al palo e una per la sfera luminosa.

La funzione `generateLampLinesFromBases` costruisce i vertici dei pali dei lampioni a partire da un'insieme di posizioni di base e dalla direzione desiderata, in modo da essere coerenti con l'andamento della strada. I vertici del palo sono disposti in modo da creare tre curve principali, che definiscono l'asta verticale e la curva superiore che sostiene la luce. Questa scelta è stata fatta perchè successivamente la geometria del palo viene amplificata e disposta lungo curve Catmull-Rom grazie agli shader, ottenendo un effetto più organico e dettagliato. In questa fase, i pali non sono stati inoltre suddivisi in patch quadrilaterali, ma in patch da due vertici, poichè nei Tessellation Shaders verrà utilizzato un tipo di patch differente.

La funzione `generateSphericalBasesFromPositions` genera le sfere luminose che si trovano all'estremità superiore del palo. La definizione della geometria delle sfere

utilizza lo stesso metodo già utilizzato per le stelle della prima ambientazione, ma utilizzando un posizionamento leggermente differente. Il centro della sfera è infatti calcolato in modo che la base superiore della sfera coincida con l'estremità finale del palo. Successivamente, per ciascuna sfera vengono generati otto triangoli che rappresentano gli ottanti della sfera, scalati in base a un raggio casuale scelto per ogni lampione.

```
pair<vector<vec3>, vector<vec3>> generateLampLinesFromBases(const vector<
    vec3>& basePositions, const vector<vec3>& directions, vector<pair<vec3,
     vec3>>& verticalRods) {

    vector<vec3> result;
    vector<vec3> lightPositions;

    float height = 1.3f;
    float width = 0.3f;
    float halfWidth = width * 0.5f;

    for (size_t i = 0; i < basePositions.size(); ++i) {
        const vec3& base = basePositions[i];
        const vec3& dir = directions[i];
        float angle = atan2(dir.x, dir.z);

        auto rotateY = [&](const vec3& offset) -> vec3 {
            return vec3(
                offset.x * cos(angle) + offset.z * sin(angle),
                offset.y,
                -offset.x * sin(angle) + offset.z * cos(angle)
            );
        };

        vec3 baseLeft = base + rotateY(vec3(-halfWidth, 0.0f, 0.0f));
        vec3 topLeft = base + rotateY(vec3(-halfWidth, height, 0.0f));
        vec3 topRight = base + rotateY(vec3(halfWidth, height, 0.0f));
        float shortLeg = height * 0.15f;
        vec3 baseRight = base + rotateY(
            vec3(halfWidth, height - shortLeg, 0.0f)
        );

        verticalRods.push_back({ baseLeft, topLeft });
        result.insert(result.end(),{baseLeft,topLeft,topRight,baseRight});
        result.insert(result.end(),{topLeft,topRight,baseRight,baseRight});
        result.insert(result.end(),{baseLeft,baseLeft,topLeft,topRight});
        lightPositions.push_back(baseRight);
    }

    return { result, lightPositions };
}
```

5.5 Personaggio animato

Un problema comune ad entrambe le ambientazioni è stato quello di integrare nella scena un modello tridimensionale di un personaggio animato, capace di muoversi all'interno del paesaggio. Per questo scopo è stato utilizzato un modello esterno in formato *FBX*, nel quale erano già presenti le animazioni corrispondenti ai movimenti del personaggio (respirazione e camminata).

La sfida principale è consistita nella gestione dei dati di animazione: è stato necessario determinare come estrarre correttamente tutte le informazioni incorporate all'interno di un file FBX e come organizzarle all'interno del codice del progetto, in modo da poter riprodurre fedelmente l'animazione durante l'esecuzione [10].

L'intero processo si è concretizzato nello studio e nell'utilizzo della libreria Assimp, che mette a disposizione funzioni, strutture dati e classi progettate per leggere e interpretare la gerarchia dei nodi, le matrici di trasformazione e i pesi delle ossa. Comprendere come sfruttare queste funzionalità ha permesso di salvare e organizzare efficacemente tutti i dati di animazione all'interno del progetto, garantendo la corretta riproduzione delle animazioni durante il rendering [12].

Figura 5.1: Personaggio animato in modalità `FILL` (sinistra) e in modalità `LINE` (destra).

5.5.1 Concetti fondamentali

Come anticipato, il modello del personaggio animato è contenuto in un file *FBX*, che non fornisce solamente la geometria del modello, ma include informazioni essenziali per l'animazione, come ad esempio il *rigging*.

Il ***rigging*** consiste nell'inserimento di uno scheletro virtuale all'interno del modello, composto da ossa (dette *bones*) che ne definiscono la struttura interna e ne con-

trollano la deformazione e il movimento. Queste ossa sono organizzate secondo una gerarchia ad albero, in cui ogni osso può avere un *parent* (o *padre*) e più *child* (o *figli*). L'osso parent si trova a un livello più alto nella gerarchia e rappresenta il riferimento per i movimenti dei figli; le ossa child, collocate a livelli più bassi, ereditano le trasformazioni applicate al parent, ma non viceversa.

Questa struttura gerarchica implica che una trasformazione applicata a un osso non si limita a influenzare la sua posizione locale, ma si propaga a tutti i figli lungo la catena, garantendo coerenza nei movimenti complessivi dello scheletro. Facendo un esempio concreto, muovendo il braccio (padre), tutto l'avambraccio e la mano (figli) si muoveranno di conseguenza. Se invece si muove solo la mano, il braccio rimane invariato.

La gerarchia ad albero permette quindi di rappresentare relazioni complesse in modo efficiente, riducendo la necessità di calcolare separatamente ogni singola trasformazione e assicurando che la mesh si deformi in modo realistico durante l'animazione. Ogni osso memorizza informazioni sulle proprie trasformazioni locali rispetto al parent, e la posizione finale nello spazio globale viene calcolata concatenando le trasformazioni lungo l'intera catena gerarchica fino al root bone.

Lo scheletro così definito costituisce la base della ***skeletal animation***, ovvero il processo mediante il quale le animazioni del modello vengono generate facendo muovere le ossa. Le trasformazioni delle ossa non si limitano al loro spazio locale, ma si propagano ai vertici del modello, indicato in questo contesto come *skin* (cioè la *pelle* visibile del personaggio), attraverso la tecnica dello *skinning*. Tale tecnica prevede che ogni vertice possa essere influenzato da più ossa, e per ognuno di questi venga assegnato un peso w_i, che determina quanto il movimento di quell'osso incide sulla trasformazione finale del vertice. La somma di tutti i pesi che influenzano un vertice deve essere pari a 1, in modo da garantire che la deformazione sia coerente e naturale.

Dato quindi un vertice della pelle v_i, influenzato da n ossa b_j (con $j = 0, \ldots, n$), la trasformazione finale di tale vertice è calcolata come combinazione lineare dei contributi delle ossa che lo influenzano:

$$\text{trasformazione\_}v_i = \sum_{j=0}^{n} w_j \cdot \text{trasformazione\_}b_j$$

Questo meccanismo garantisce che il modello si deformi in maniera coerente e naturale, consentendo animazioni fluide e realistiche.

5.5.2 Parsing del modello con Assimp

Per semplificare la gestione dei modelli tridimensionali, in particolare dei file in formato FBX, è stata utilizzata la libreria *Assimp* (Open Asset Import Library). Questa libreria si occupa di leggere i dati presenti nel file di input e di convertirli in strutture dati gerarchiche, che possono poi essere direttamente sfruttate per il rendering e per l'animazione.

L'elemento centrale attorno a cui ruota l'organizzazione delle informazioni è l'oggetto `aiScene`, restituito dal metodo `Importer.readFile(filename, flags)` della classe `Importer`. Quest'ultima gestisce l'intero processo di importazione e mette a disposizione una serie di parametri di post-processing (i cosiddetti `flags`) che permettono di adattare i dati alle necessità del rendering, evitando di dover implementare manualmente procedure di conversione o pulizia. Fra i più comuni si trovano, ad esempio, `aiProcess_Triangulate`, che converte tutte le primitive in triangoli, `aiProcess_JoinIdenticalVertices`, che elimina i vertici duplicati riducendo la memoria occupata, e `aiProcess_GenNormals`, che genera automaticamente le normali quando queste non sono presenti.

L'oggetto `aiScene` rappresenta quindi il contenitore principale di tutte le informazioni importate e raccoglie sia i dati geometrici dei modelli, sia quelli relativi alle ossa e alle animazioni. Al suo interno, uno degli attributi più importanti è l'array `mMeshes[]`, che contiene tutte le mesh della scena e la cui dimensione è definita dal valore `mNumMesh`. Una mesh, rappresentata dall'oggetto `aiMesh`, descrive la geometria del modello tramite collezioni di vertici connessi da primitive (tipicamente triangoli), accompagnati da informazioni aggiuntive come normali e coordinate di texture.

5.5.3 Mesh e ossa

Tra i campi principali di `aiMesh`, di particolare interesse per la skeletal animation, vi sono: l'array `mVertices[]` che contiene le coordinate di ciascun vertice, l'array `mNormals[]` che memorizza le normali, l'array di puntatori `mTextureCoords[][]` che conserva, per ogni canale di texture, le coordinate corrispondenti ai vertici, e infine i campi `mNumBones` e `mBones[]`, che indicano rispettivamente il numero di ossa associate alla mesh e l'array degli oggetti `aiBone` che descrivono tali ossa.

Ciascun osso è quindi rappresentato da un oggetto `aiBone`, che fornisce tutti i dati necessari per legare la mesh allo scheletro. In particolare, l'attributo `mName` identifica l'osso all'interno della gerarchia, specificandone il nome univoco, mentre l'array `mWeights[]` descrive l'elenco dei vertici della mesh influenzati da quell'osso. Ogni elemento di questo array è un oggetto della struttura `aiVertexWeight`, la quale contiene due campi fondamentali: `mVertexId`, che specifica l'indice del vertice influenzato, e `mWeight`, che rappresenta il peso con cui l'osso contribuisce al movimento di quel vertice.

Un ruolo cruciale è poi svolto dalla matrice `mOffsetMatrix`, che definisce la trasformazione necessaria per portare i vertici dallo spazio della mesh allo spazio locale dell'osso nella *bind pose*, ovvero nella configurazione iniziale del modello.

5.5.4 Gerarchia delle ossa

L'oggetto `aiScene` contiene al suo interno anche le informazioni riguardanti l'organizzazione gerarchica delle ossa che compongono lo scheletro di ciascuna mesh presente. In particolare, fra gli ulteriori campi fondamentali messi a disposizione da

`aiScene`, è presente `mRootNode`,che rappresenta il nodo radice della gerarchia ed è implementato come un oggetto `aiNode`.

Ogni oggetto `aiNode` rappresenta un nodo della gerarchia corrispondente ad un osso dello scheletro ed è caratterizzato da diversi attributi: il campo `mName` che identifica il nodo mediante un nome univoco (consentendo di stabilire la corrispondenza con l'osso che rappresenta), il campo `mParent` che mantiene un riferimento al nodo padre, il campo `mNumChildren` che specifica il numero di nodi figli e l'array `mChildren[]` che contiene i riferimenti a ciascun nodo figlio. In questo modo l'insieme dei nodi viene organizzato in una struttura ad albero, con `mRootNode` come radice e tutti gli altri nodi collegati gerarchicamente.

5.5.5 Sistemi di riferimento dello scheletro

Per poter animare correttamente una mesh, è fondamentale gestire i diversi sistemi di riferimento messi a disposizione da Assimp. In particolare, possiamo distinguere tre spazi principali: il *local space*, che rappresenta il sistema di coordinate della mesh originale, il *root bone space*, che rappresenta il sistema di riferimento dell'osso corrispondente al nodo radice, e il *child bone space*, che rappresenta il sistema di riferimento delle ossa corrispondenti ai nodi figli, ereditando le trasformazioni dai nodi genitori.

Ogni nodo della gerarchia è rappresentato da un oggetto `aiNode`, che contiene informazioni fondamentali per determinare la posizione dei vertici nello spazio globale. Tra queste, il campo `mTransformation` definisce la matrice di trasformazione locale del nodo rispetto al proprio padre, descrivendo come il sistema di coordinate del nodo deve essere trasformato per allinearsi a quello del padre. La trasformazione globale di un nodo si ottiene moltiplicando ricorsivamente le trasformazioni lungo il percorso che va dalla radice al nodo stesso. Ad esempio, consideriamo una gerarchia di nodi A, B e C (dove A è la radice, B è figlio di A e C è figlio di B), la trasformazione globale del nodo C si ottiene come:

$$globalTransform_C = mTransformation_A \cdot mTransformation_B \cdot mTransformation_C$$

Nella maggior parte dei modelli importati, la matrice `mTransformation` del nodo radice coincide con la matrice identità, poiché la scena è modellata direttamente rispetto a tale nodo.

Combinando la trasformazione locale di ciascun nodo (`mTransformation`) con la matrice `mOffsetMatrix` delle ossa (descritta nella sezione precedente), è possibile calcolare la posizione finale dei vertici nello spazio globale della scena (cioè nel sistema di riferimento del nodo radice) durante l'animazione:

$$boneSpacePosition = mOffsetMatrix \cdot localSpacePosition$$
$$globalSpacePosition = globalTransform_{node} \cdot bone\_space\_position$$

dove *localSpacePosition* sono le coordinate del vertice nello spazio della mesh originale, *mOffsetMatrix* è la matrice che trasforma il vertice dallo spazio della mesh allo spazio locale dell'osso nella bind pose, $globalTransform_{node}$ rappresenta la trasformazione globale del nodo (ottenuta moltiplicando tutte le `mTransformation` lungo il

percorso dalla radice al nodo corrente), *boneSpacePosition* è la posizione del vertice nello spazio dell'osso e *globalSpacePosition* è la posizione finale del vertice nello spazio globale.

5.5.6 Animazioni

Infine, l'oggetto `aiScene` raccoglie le informazioni sulle animazioni tramite l'array `mAnimations`, la cui dimensione è specificata dal campo `mNumAnimations`.

Ogni elemento di questo array è un oggetto `aiAnimation`, che descrive una sequenza di movimento completa. I campi principali di `aiAnimation` sono `mDuration`, che definisce la durata dell'animazione espressa in *tick*, `mTicksPerSecond`, che indica il numero di *tick* per secondo, e l'array `mChannels[]`, che raccoglie i canali di animazione.

Ciascun canale è rappresentato da un oggetto `aiNodeAnim` che definisce l'animazione di un nodo specifico, identificato dal campo `mNodeName`. Ogni `aiNodeAnim` contiene quindi tre array: `mPositionKeys[]`, `mRotationKeys[]` e `mScalingKeys[]`, i cui elementi descrivono matematicamente i valori di traslazione, rotazione e scalatura nel tempo. Le rispettive dimensioni sono specificate dai campi `mNumPositionKeys`, `mNumRotationKeys` e `mNumScalingKeys`.

Grazie ai campi relativi all'unità di tempo, è possibile convertire la durata dell'animazione in secondi mediante la formula:

$$\text{animation\_time\_in\_seconds} = \text{mDuration} \,/\, \text{mTicksPerSecond}$$

Durante l'esecuzione dell'animazione, i valori contenuti negli array `mPositionKeys[]`, `mRotationKeys[]` e `mScalingKeys[]` vengono interpolati in base al tempo corrente, così da ottenere la trasformazione locale del nodo in ogni istante. Questa trasformazione locale viene poi combinata con le trasformazioni dei nodi parent lungo la gerarchia (`mTransformation` di ciascun `aiNode`) e con la matrice `mOffsetMatrix` dell'osso (`aiBone`), al fine di calcolare la posizione finale dei vertici nello spazio globale durante l'animazione.

5.5.7 Implementazione del modello animato

Per integrare un modello animato all'interno del progetto, è stato sviluppato un sistema di caricamento e gestione dei dati forniti dai file FBX, sfruttando le funzionalità offerte dalla libreria Assimp che sono state descritte precedentemente. L'implementazione si articola in due fasi principali: caricamento dei dati geometrici della mesh e delle ossa (con associazione dei relativi vertici alle ossa) e calcolo delle trasformazioni delle ossa durante l'animazione.

Caricamento delle mesh e delle ossa

Il caricamento dei modelli avviene tramite la funzione `loadModel`, che si occupa di leggere il file FBX e di costruire le strutture dati necessarie per la gestione delle animazioni scheletriche. Per ogni mesh presente nella scena, le posizioni, le normali

e le coordinate di texture vengono salvate in array globali, mantenendo un offset che permette di riferirsi ai vertici correttamente anche in presenza di più mesh.

Parallelamente, per ciascun osso associato alla mesh, viene registrata la sua matrice di offset, che trasforma le coordinate dei vertici dalla posizione nello spazio della mesh allo spazio locale dell'osso nella bind pose. Ogni vertice riceve inoltre una lista di ossa che lo influenzano, insieme ai pesi corrispondenti, tramite la struttura `VertexBoneData`. Questo permette di calcolare successivamente le deformazioni della mesh in base alle animazioni applicate allo scheletro.

In sintesi, questa fase garantisce che tutte le informazioni necessarie per animare correttamente la mesh siano disponibili: la geometria è memorizzata, ogni vertice è associato alle ossa che ne determinano il movimento e le matrici di offset delle ossa sono salavate per calcolare le trasformazioni globali durante l'animazione.

```
void loadMeshBones(const int meshIndex, const aiMesh* mesh, ModelState
    state) {
    for (int i = 0; i < mesh->mNumBones; i++) {
        const aiBone* bone = mesh->mBones[i];

        int boneID = getBoneID(bone, state);

        for (int j = 0; j < bone->mNumWeights; j++) {
            const aiVertexWeight& vertexWeight = bone->mWeights[j];
            unsigned int globalVertexID = mesh_vertices[meshIndex] +
                vertexWeight.mVertexId;
            vertices_to_bones[globalVertexID].addBone(
                boneID,
                vertexWeight.mWeight
            );
        }
    }
}
```

```
void loadSceneData(const aiScene* scene, ModelState state) {
    int total_vertices = 0;
    int total_indices = 0;
    int total_bones = 0;

    mesh_vertices.resize(scene->mNumMeshes);

    for (int i = 0; i < scene->mNumMeshes; i++) {
        const aiMesh* mesh = scene->mMeshes[i];
        total_vertices += mesh->mNumVertices;
        total_indices += mesh->mNumFaces * 3;
        total_bones += mesh->mNumBones;
    }

    positions.reserve(total_vertices);
```

```
    normals.reserve(total_vertices);
    texCoords.reserve(total_vertices);
    indices.reserve(total_indices);

    vertices_to_bones.resize(total_vertices);

    int vertex_offset = 0;
    for (int i = 0; i < scene->mNumMeshes; i++) {
        const aiMesh* mesh = scene->mMeshes[i];
        int num_vertices = mesh->mNumVertices;
        int num_indices = mesh->mNumFaces * 3;
        int num_bones = mesh->mNumBones;

        mesh_vertices[i] = vertex_offset;

        for (int v = 0; v < num_vertices; v++) {
            unsigned int globalVertexID = mesh_vertices[i] + v;
            VertexBoneData& vertex = vertices_to_bones[globalVertexID];

            // Position
            aiVector3D pos = mesh->mVertices[v];
            positions.push_back(vec3(pos.x, pos.y, pos.z));

            // Normal
            if (mesh->HasNormals()) {
                aiVector3D normal = mesh->mNormals[v];
                normals.push_back(vec3(normal.x, normal.y, normal.z));
            }
            else {
                aiVector3D backupNormal(0.0f, 1.0f, 0.0f);
                normals.push_back(vec3(
                    backupNormal.x,
                    backupNormal.y,
                    backupNormal.z
                ));
            }

            // Texture coords
            if (mesh->HasTextureCoords(0)) {
                aiVector3D uv = mesh->mTextureCoords[0][v];
                texCoords.push_back(vec2(uv.x, uv.y));
            }
            else {
                aiVector3D backupUV(0.0f, 0.0f, 0.0f);
                texCoords.push_back(vec2(backupUV.x, backupUV.y));
            }
        }

        for (int f = 0; f < mesh->mNumFaces; f++) {
```

```
            const aiFace& face = mesh->mFaces[f];
            indices.push_back(mesh_vertices[i] + face.mIndices[0]);
            indices.push_back(mesh_vertices[i] + face.mIndices[1]);
            indices.push_back(mesh_vertices[i] + face.mIndices[2]);
        }

        if (mesh->HasBones()) {
            loadMeshBones(i, mesh, state);
        }

        vertex_offset += num_vertices;
    }

    for (auto& v : vertices_to_bones) {
        v.normalize();
    }
}
```

```
void loadModel(string modelPath, ModelState state) {
    aiMatrix4x4 transform;

    if (state == WALKING) {
        scene_walking = importerWalking.ReadFile(
            modelPath,
            aiProcess_Triangulate |
            aiProcess_GenSmoothNormals |
            aiProcess_JoinIdenticalVertices
        );

        if (!scene_walking || !scene_walking->HasMeshes()) {
            return;
        }

        transform = scene_walking->mRootNode->mTransformation;
        globalInverseTransformation = inverse(
            aiMatrix4x4_to_mat4(transform)
        );

        loadSceneData(scene_walking, state);
    }
    else {
        scene_standing = importerStanding.ReadFile(
            modelPath,
            aiProcess_Triangulate |
            aiProcess_GenSmoothNormals |
            aiProcess_JoinIdenticalVertices
        );

        if (!scene_standing || !scene_standing->HasMeshes()) {
```

```
32              return;
33          }
34
35          transform = scene_standing->mRootNode->mTransformation;
36          globalInverseTransformation = inverse(
37              aiMatrix4x4_to_mat4(transform)
38          );
39
40          loadSceneData(scene_standing, state);
41      }
42  }
```

Calcolo delle trasformazioni delle ossa

Il cuore dell'animazione scheletrica è la funzione `readNodeHierarchy`, che percorre ricorsivamente la gerarchia dei nodi dello scheletro, calcolando la trasformazione globale di ciascun nodo a partire dalle trasformazioni locali e dai dati di animazione.

Per ogni nodo, la funzione verifica se è presente un canale di animazione (`aiNodeAnim`) associato. Se presente, vengono calcolati tramite interpolazione i valori di traslazione, rotazione e scalatura in base al tempo corrente. Tali interpolazioni sono gestite da funzioni dedicate (`CalcInterpolatedTranslation`, `CalcInterpolatedRotation` e `CalcInterpolatedScaling`), che si occupano di individuare i *keyframe* adiacenti e calcolare una media ponderata tra i valori, ottenendo così transizioni fluide tra le pose definite dall'animazione. Con il termine *keyframe* si intendono punti nel tempo in cui vengono definite esplicitamente le trasformazioni di un nodo: tra due keyframe consecutivi, le trasformazioni vengono interpolate per generare movimenti continui e realistici.

Le trasformazioni interpolate vengono quindi combinate per ottenere la trasformazione locale del nodo, che rappresenta lo spostamento, l'orientamento e la scala del nodo rispetto al nodo padre. Questa trasformazione locale viene poi moltiplicata per la trasformazione globale del nodo genitore, generando così la trasformazione globale del nodo nello spazio del modello.

Per ogni osso associato a un nodo, la trasformazione globale calcolata viene combinata con la matrice di offset dell'osso (`offsetMatrix`), riportando i vertici dallo spazio della bind pose allo spazio locale dell'osso. Questo passaggio è fondamentale per garantire che le deformazioni della mesh seguano correttamente i movimenti dello scheletro e rispettino la gerarchia dei nodi.

Il processo viene eseguito ricorsivamente su tutti i figli del nodo, aggiornando in modo coerente tutte le ossa dello scheletro. Grazie a questa procedura, ogni vertice della mesh può essere trasformato correttamente in base alla combinazione delle ossa che lo influenzano, permettendo animazioni realistiche e fluide in tempo reale.

```
1  void readNodeHierarchy(float animationTimeTicks, const aiNode* node, const
       mat4& parentTransform, ModelState state) {
2      string nodeName = node->mName.data;
```

```
    aiAnimation* animation;
    if (state == WALKING) {
        animation = scene_walking->mAnimations[0];
    }
    else{
        animation = scene_standing->mAnimations[0];
    }

    mat4 nodeTransformation = aiMatrix4x4_to_mat4(node->mTransformation);
    aiNodeAnim* nodeAnimation = findNodeAnim(animation, nodeName);

    if (nodeAnimation) {
        aiVector3D scaling;
        mat4 scalingMatrix = mat4(1.0f);
        CalcInterpolatedScaling(
            scaling,
            animationTimeTicks,
            nodeAnimation
        );
        scalingMatrix = scale(
            scalingMatrix,
            vec3(scaling.x, scaling.y, scaling.z)
        );

        aiQuaternion rotation;
        CalcInterpolatedRotation(
            rotation,
            animationTimeTicks,
            nodeAnimation
        );
        quat glmRotation = quat(
            rotation.w,
            rotation.x,
            rotation.y,
            rotation.z
        );
        mat4 rotationMatrix = toMat4(glmRotation);

        aiVector3D translation;
        mat4 translationMatrix = mat4(1.0f);;
        CalcInterpolatedTranslation(
            translation,
            animationTimeTicks,
            nodeAnimation
        );
        translationMatrix = translate(
            translationMatrix,
            vec3(translation.x, translation.y, translation.z)
        );
```

```
52
53          nodeTransformation = translationMatrix *
54              rotationMatrix *
55              scalingMatrix;
56      }
57
58      mat4 globalTransform = parentTransform * nodeTransformation;
59
60      if (state == WALKING) {
61          if (bone_name_to_index_walking.find(nodeName) !=
    bone_name_to_index_walking.end()) {
62              int boneIndex = bone_name_to_index_walking[nodeName];
63              bone_info_walking[boneIndex].finalTransform =
64                  globalInverseTransformation *
65                  globalTransform *
66                  bone_info_walking[boneIndex].offsetMatrix;
67          }
68      }
69      else {
70          if (bone_name_to_index_standing.find(nodeName) !=
    bone_name_to_index_standing.end()) {
71              int boneIndex = bone_name_to_index_standing[nodeName];
72              bone_info_standing[boneIndex].finalTransform =
73                  globalInverseTransformation *
74                  globalTransform *
75                  bone_info_standing[boneIndex].offsetMatrix;
76          }
77      }
78
79      for (unsigned int i = 0; i < node->mNumChildren; i++) {
80          readNodeHierarchy(
81              animationTimeTicks,
82              node->mChildren[i],
83              globalTransform,
84              state
85          );
86      }
87  }
```

Aggiornamento delle animazioni

Durante l'esecuzione, la funzione `updateBoneTransforms` viene chiamata ogni frame per aggiornare le trasformazioni delle ossa in base al tempo di animazione corrente. Grazie al mapping tra nomi delle ossa e indici negli array di trasformazioni, le matrici finali calcolate possono essere trasferite direttamente agli shader per applicare lo *skinning* dei vertici.

Gestione delle texture incorporate

Infine, la funzione `extractEmbeddedTextures` consente di salvare su disco eventuali texture incorporate all'interno del file FBX. Questo permette di riutilizzare tali immagini durante il rendering, garantendo che il modello venga visualizzato correttamente anche se il file originale non fornisce texture esterne separate.

Per ottenere ciò, la funzione sfrutta alcune strutture dati messe a disposizione dalla libreria Assimp. In particolare, si utilizza il campo `mTextures` della classe `aiScene`, un'array di lunghezza `mNumTextures` che contiene tutte le texture salvate internamente al file, rappresentate da oggetti `aiTexture`.

Il contenuto di ciascuna texture viene quindi salvato in un file separato, con estensione specificata dal campo `achFormatHint` di `aiTexture`, utilizzando le normali operazioni di scrittura su file in C/C++. I dati da scrivere sono contenuti nel campo `pcData`, mentre la dimensione del contenuto è indicata dal campo `mWidth`.

```
void extractEmbeddedTextures(const string modelPath, const string&
    outputDirectory) {
    scene_bind_pose = importerBindPose.ReadFile(
        modelPath,
        aiProcess_Triangulate |
        aiProcess_GenSmoothNormals |
        aiProcess_JoinIdenticalVertices
    );

    if (!scene_bind_pose->HasTextures()) {
        return;
    }

    for (unsigned int i = 0; i < scene_bind_pose->mNumTextures; ++i) {
        const aiTexture* texture = scene_bind_pose->mTextures[i];

        if (texture->mHeight == 0) {
            string extension = texture->achFormatHint;
            string fileName = outputDirectory + "/texture_embedded_" +
                to_string(i) + "." + extension;

            ofstream fout(fileName, ios::binary);
            fout.write(
                reinterpret_cast<const char*>(texture->pcData),
                texture->mWidth
            );
            fout.close();
        }
    }
}
```

5.6 Buffer di memoria

Tutti i vertici delle geometrie generate, per poter essere passati e utilizzati dagli shader della pipeline, devono essere organizzati e caricati in un sistema di buffer utilizzato da OpenGL. I buffer più comuni sono i *Vertex Buffer Object* (VBO), che contengono insiemi di dati come le coordinate dei vertici, le normali o le coordinate di texture, e i *Vertex Array Object* (VAO), che fungono da contenitori logici e memorizzano la configurazione che specifica come i dati devono essere letti ed interpretati durante il rendering.

L'associazione tra un VAO e uno o più VBO permette di organizzare le informazioni in maniera ordinata ed efficiente. Ogni VBO può contenere un tipo specifico di attributo: per esempio le posizioni dei vertici, le normali, o ancora i pesi di influenza delle ossa in un modello animato. Tramite la funzione `glVertexAttribPointer` si specifica come i dati sono strutturati (dimensione, tipo, offset, ecc.), mentre con `glEnableVertexAttribArray` si attiva il relativo attributo, che verrà poi utilizzato dagli shader.

Un ulteriore buffer, chiamato *Element Buffer Object* (EBO), viene invece utilizzato per gestire gli indici. In questo modo è possibile evitare la duplicazione di vertici che appartengono a più geometrie di base, migliorando sia la compattezza dei dati sia l'efficienza in fase di rendering.

Un esempio completo di questa organizzazione dei dati si ritrova nella funzione `INIT_MODEL_BUFFERS`, la funzione utilizzata per caricare tutti i dati relativi al personaggio animato nei relativi buffer. Al suo interno viene generato un VAO che contiene diversi VBO, ciascuno dedicato a un tipo specifico di informazione: il primo contiene le posizioni dei vertici, il secondo le normali, il terzo le coordinate di texture, il quarto gli identificatori delle ossa e il quinto i pesi relativi alle ossa.

Infine, viene associato un EBO che memorizza gli indici dei vertici da utilizzare per disegnare i triangoli che compongono la mesh. In questo modo, durante il rendering, OpenGL può ricostruire correttamente la geometria del modello senza ridondanza nei dati.

```
ModelBufferPair INIT_MODEL_BUFFERS() {
        ModelBufferPair pair;

    vector<ivec4> boneIDs(vertices_to_bones.size());
    vector<vec4> weights(vertices_to_bones.size());
    for (size_t i = 0; i < vertices_to_bones.size(); i++) {
        boneIDs[i] = ivec4(
            vertices_to_bones[i].boneIDs[0],
            vertices_to_bones[i].boneIDs[1],
            vertices_to_bones[i].boneIDs[2],
            vertices_to_bones[i].boneIDs[3]
        );

        weights[i] = vec4(
```

```
            vertices_to_bones[i].weights[0],
            vertices_to_bones[i].weights[1],
            vertices_to_bones[i].weights[2],
            vertices_to_bones[i].weights[3]
        );
    }

    glGenVertexArrays(1, &pair.vao);
    glBindVertexArray(pair.vao);

    // Genera i buffer
    glGenBuffers(1, &pair.vboPositions);
    glGenBuffers(1, &pair.vboNormals);
    glGenBuffers(1, &pair.vboTexCoords);
    glGenBuffers(1, &pair.ebo);

    // POSIZIONI
    glBindBuffer(GL_ARRAY_BUFFER, pair.vboPositions);
    glBufferData(GL_ARRAY_BUFFER, positions.size() * sizeof(vec3),
    positions.data(), GL_STATIC_DRAW);
    glEnableVertexAttribArray(0); // location 0
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(vec3),(void*)0);

    // NORMALI
    glBindBuffer(GL_ARRAY_BUFFER, pair.vboNormals);
    glBufferData(GL_ARRAY_BUFFER, normals.size() * sizeof(vec3), normals.
    data(), GL_STATIC_DRAW);
    glEnableVertexAttribArray(1); // location 1
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(vec3),(void*)0);

    // TEX COORDS
    glBindBuffer(GL_ARRAY_BUFFER, pair.vboTexCoords);
    glBufferData(GL_ARRAY_BUFFER, texCoords.size() * sizeof(vec2),
    texCoords.data(), GL_STATIC_DRAW);
    glEnableVertexAttribArray(2); // location 2
    glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(vec2),(void*)0);

    // BONES: boneIDs (interi)
    glGenBuffers(1, &pair.vboBoneIDs);
    glBindBuffer(GL_ARRAY_BUFFER, pair.vboBoneIDs);
    glBufferData(GL_ARRAY_BUFFER, boneIDs.size() * sizeof(ivec4), boneIDs.
    data(), GL_STATIC_DRAW);
    glEnableVertexAttribArray(3); // location 3
    glVertexAttribIPointer(3, 4, GL_INT, sizeof(ivec4), (void*)0);

    // BONES: weights (float)
    glGenBuffers(1, &pair.vboBoneWeights);
    glBindBuffer(GL_ARRAY_BUFFER, pair.vboBoneWeights);
    glBufferData(GL_ARRAY_BUFFER, weights.size() * sizeof(vec4), weights.
```

```
    data(), GL_STATIC_DRAW);
60      glEnableVertexAttribArray(4); // location 4
61      glVertexAttribPointer(4, 4, GL_FLOAT, GL_FALSE, sizeof(vec4),(void*)0);
62
63      // INDICI
64      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, pair.ebo);
65      glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned
        int), indices.data(), GL_STATIC_DRAW);
66
67      glBindVertexArray(0);
68      return pair;
69  }
```

5.7 Rumore procedurale

Un aspetto centrale nello sviluppo della prima ambientazione è stata la definizione di una *displacement map*, utilizzata per deformare dinamicamente la superficie del terreno e generare un paesaggio realistico, con variazioni di altitudine che riproducessero montagne, colline, pianure e aree costiere. Con il termine *displacement map* si intende una mappa bidimensionale di valori numerici che, una volta interpretata dalla pipeline grafica, viene impiegata per spostare i vertici della geometria lungo la direzione normale, modificando effettivamente la forma del modello.

Per generare questa mappa non si è fatto ricorso a dati predefiniti, ma si è scelto un approccio basato sulla *procedural generation* (o *generazione procedurale*). Con questa espressione si intende la creazione di contenuti attraverso algoritmi matematici, anziché tramite modellazione manuale. In questo modo è possibile generare scenari vasti, senza che risultino artificiosi o ripetitivi, mantenendo al contempo un elevato livello di controllo sui parametri che determinano la morfologia del terreno.

Il principio alla base di queste tecniche è l'impiego del cosiddetto *rumore* (o *noise*). In ambito matematico e informatico, il termine indica una funzione matematica che restituisce valori numerici distribuiti secondo un certo grado di complessità e variabilità. Un rumore puramente casuale produce sequenze di valori tra loro indipendenti, senza alcuna correlazione spaziale, e di conseguenza risulta del tutto privo di struttura. Per molte applicazioni grafiche, tuttavia, è preferibile utilizzare un rumore pseudo-casuale, che conserva la componente di imprevedibilità tipica del caso ma introduce al tempo stesso una certa continuità. In questo modo, i valori generati non sono completamente disgiunti, ma presentano transizioni graduali e correlazioni locali [5].

Tra i diversi metodi possibili per generare superfici procedurali, è stato scelto l'algoritmo Perlin Noise, introdotto da Ken Perlin negli anni '80 [13]. Si tratta di una funzione in grado di generare un rumore pseudo-casuale "morbido", caratterizzato da variazioni graduali e prive di discontinuità. Integrato in una displacement map, il Perlin Noise permette di creare terreni realistici, con rilievi e depressioni distribuiti in modo coerente, che richiamano l'andamento del paesaggio reale [3], [11].

Nei paragrafi seguenti vengono descritte le principali fasi dell'algoritmo.

5.7.1 Valori iniziali

L'algoritmo ha inizio assumendo di possedere una griglia quadrata di dimensione $n \cdot n$. Si assume anche che il valore di n sia un numero piccolo, solitamente compreso fra 2 e 10.

Per ogni vertice (i, j) appartenente alla griglia, dove $0 \leq i, j \leq n$, si genera un valore scalare randomico z_{ij}, che rappresenta il valore di base del punto. Per facilitare la continuità sui bordi della griglia, è conveniente applicare gli stessi valori ai punti più esterni sui vari lati della griglia: $z_{in} = z_{i0}$ e $z_{nj} = z_{0j}$ (per ogni valore di i e j).

Dato un punto qualsiasi (x, y), con $0 \leq x, y \leq n$, i vertici appartenenti alla griglia che compongono la cella che contiene il punto sono i seguenti:

$$\begin{array}{ll} x_0 = \lfloor x \rfloor & x_1 = (x_0 + 1) \bmod n \\ y_0 = \lfloor y \rfloor & y_1 = (y_0 + 1) \bmod n \end{array}$$

5.7.2 Random gradients

Nell'algoritmo classico di Perlin Noise, per ogni vertice della griglia si calcola poi un vettore bidimensionale randomico che rappresenta il gradiente del punto. Questo determina la direzione lungo la quale la funzione di noise tenderà ad aumentare o diminuire. In questo modo però, i gradienti relativi a vertici adiacenti possono essere completamente differenti: in un punto il terreno potrebbe pendere verso nord-est, mentre nel vertice confinante potrebbe pendere verso sud-ovest, generando un andamento brusco tra le celle.

Per fare in modo che la funzione di noise abbia un effetto più graduale e non completamente randomico su ogni singolo punto della griglia, l'algoritmo classico opera nel seguente modo. Preso una singola cella della griglia, si indicano i suoi estremi con (x_0, y_0), (x_1, y_0), (x_1, y_1), (x_0, y_1) e i gradienti corrispondenti con $g_{0,0}$, $g_{1,0}$, $g_{1,1}$, $g_{0,1}$. Per ogni punto (x, y) interno alla cella si calcolano i vettori che congiungono ciascun vertice della cella al punto stesso:

$$\begin{array}{ll} v_{0,0} = (x, y) - (x_0, y_0) & v_{0,1} = (x, y) - (x_0, y_1) \\ v_{1,1} = (x, y) - (x_1, y_1) & v_{1,0} = (x, y) - (x_1, y_0) \end{array}$$

e si valuta il *dot product* con i corrispondenti gradienti dei vertici. Il risultato prende il nome di *vertical displacement* e rappresenta l'altezza che il punto avrebbe nello spazio a tre dimensioni. Questo valore è massimo se i vettori puntano nella stessa direzione, minimo se opposti, nullo se ortogonali. Il dot product, oltre che rappresentare la scelta più consona, permette anche di scalare l'effetto in maniera lineare con la distanza dal vertice. I valori di displacement per ogni cella sono i seguenti:

$$\begin{array}{ll} \delta_{0,0} = (v_{0,0} \cdot g_{0,0}) & \delta_{0,1} = (v_{0,1} \cdot g_{0,1}) \\ \delta_{1,1} = (v_{1,1} \cdot g_{1,1}) & \delta_{1,0} = (v_{1,0} \cdot g_{1,0}) \end{array}$$

5.7.3 Fading

Uno dei problemi principali dei valori di displacement è che essi risultano influenzati da tutti gli angoli della cella corrispondente. Man mano che ci si sposta dal vertice della cella, il valore tende ad aumentare, mentre l'effetto desiderato è esattamente il contrario: il gradiente di displacement deve applicarsi con maggiore intensità vicino ai vertici della cella e diminuire progressivamente allontanandosi da essi.

Per ottenere questo comportamento, si definisce una funzione particolare chiamata *fade function*, una funzione parametrica dipendente da un parametro t, che restituisce 0 quando $t = 0$ e 1 quando $t = 1$. Originariamente la funzione era $\phi(t) = 3t^2 - 2t^3$, ma successivamente si è adottata una versione più regolare:

$$\psi(t) = 6t^5 - 15t^4 + 10t^3$$

dove $\psi(0) = 0$, $\psi(1) = 1$ e la derivata seconda è non nulla in entrambi i valori, garantendo una transizione più morbida.

In due dimensioni, si definisce invece la *joint fade function*, ottenuta moltiplicando i valori della fade function classica calcolati sui due assi: $\Psi(x, y) = \psi(x)\psi(y)$.

Il valore finale della noise function al punto (x, y) si ottiene tramite un'interpolazione bilineare pesata dei quattro vertici della cella corrispondente, utilizzando i valori della joint fade function $\Psi(x, y)$ come pesi:

$$noise(x, y) = \Psi(1 - x, 1 - y)\delta_{0,0} + \Psi(x, 1 - y)\delta_{1,0} + \Psi(1 - x, y)\delta_{0,1} + \Psi(x, y)\delta_{1,1}$$

In questo modo, ogni vertice contribuisce in maniera graduale e continua al valore finale, intensificando il proprio contributo man mano che il punto si trova vicino ad esso, evitando brusche discontinuità.

5.7.4 Aggiunta delle altezze random

Per ottenere un ulteriore elemento di casualità, è possibile aggiungere al valore di displacement verticale anche i valori scalari randomici inizialmente assegnati ai punti della griglia come valori di base (vedi sezione 5.7.1). Considerata la cella avente (i, j) come indici del vertice in basso a sinistra, il calcolo dei valori di displacement diventerà:

$$\delta_{0,0} = z_{i,j}(v_{0,0} \cdot g_{0,0}) \qquad \delta_{0,1} = z_{i,j+1}(v_{0,1} \cdot g_{0,1})$$
$$\delta_{1,1} = z_{i+1,j+1}(v_{1,1} \cdot g_{1,1}) \qquad \delta_{1,0} = z_{i+1,j}(v_{1,0} \cdot g_{1,0})$$

5.7.5 Brownian motion

Il risultato del Perlin Noise classico in due dimensioni è una funzione continua e graduale, con variazioni dolci sui due assi. Pur essendo casuale, le transizioni tra i valori adiacenti sono morbide e prive di brusche discontinuità. Tuttavia, l'ampiezza delle variazioni rimane relativamente costante e la frequenza delle oscillazioni si mantiene in un range limitato, il che conferisce al risultato un aspetto regolare e poco frastagliato.

Per rendere la funzione più dettagliata e ricca di variazioni, è possibile sommare tra loro diverse versioni della stessa funzione, scalate mediante l'uso di frequenze e ampiezze diverse. Questo approccio prende il nome di *Fractal Brownian Motion* (FBM).

Nella pratica, ogni iterazione (detta *octave* o *ottava*) di questa somma utilizza un rumore leggermente diverso, con frequenza incrementata e ampiezza ridotta rispetto all'iterazione precedente. L'incremento di frequenza è definito *lacunarity*, mentre la diminuzione dell'ampiezza è definita *gain*.

Aggiungendo un rumore diverso ad ogni iterazione (*octaves*), modificato incrementando la frequenza originale con step regolari (*lacunarity*) e decrementando l'ampiezza (*gain*), è possibile ottenere una granularità più fine e ottenere così maggiori dettagli.

Tipicamente, la frequenza viene raddoppiata e l'ampiezza dimezzata ad ogni ottava, producendo un rumore più granulare e dettagliato. La funzione risultante può quindi essere espressa come:

$$FBM(x, y) = \sum_{i=0}^{k} p^i \cdot noise(2^i \cdot x, 2^i \cdot y)$$

dove p^i rappresenta l'ampiezza dell'ottava i-esima, mentre 2^i scala la frequenza. In questo modo, la combinazione di più ottave produce un terreno digitale con dettagli su più scale, senza perdere la continuità e la regolarità tipiche del Perlin Noise.

5.7.6 Implementazione

Nell'implementazione del progetto, l'algoritmo di generazione del Perlin Noise è stato modificato rispetto alla versione classica per garantire maggiore controllo, continuità tra le celle e semplicità computazionale.

Invece di utilizzare dei gradienti completamente randomici per ogni vertice della griglia, infatti, si è scelto di utilizzare un insieme discreto di quattro direzioni costanti $(1, 1), (-1, 1), (-1, -1), (1, -1)$. La scelta del gradiente per ciascun vertice viene determinata tramite una tabella di permutazione pseudo-casuale di 256 elementi, duplicata per ottenere un array di 512 valori. La tabella di permutazione rappresenta un insieme di valori interi compresi tra 0 e 255 generati casualmente tramite il *generatore Mersenne Twister*, un algoritmo per la generazione di numeri pseudo-casuali ad alta rapidità, messo a disposizione dalla libreria `<random>`.

In questo modo l'effetto casuale del terreno viene preservato, si evitano discontinuità improvvise tra le celle adiacenti e si semplifica la funzione di interpolazione.

Di seguito viene riportata una porzione del codice utilizzato per l'implementazione dell'algoritmo Perlin Noise, comprensivo della gestione delle ottave per la FBM. e dove `MakePermutation()` rappresenta la funzione di riempimento della tabella di permutazione:

```
1 vector<float> textureData;
```

```
static vector<int> permutation = MakePermutation();

vec2 GetConstantVector(int value) {
        switch (value & 3) {
                case 0: return vec2(1.0f, 1.0f);
                case 1: return vec2(-1.0f, 1.0f);
                case 2: return vec2(-1.0f, -1.0f);
                default: return vec2(1.0f, -1.0f);
        }
}

float Fade(float t) {
        return ((6 * t - 15) * t + 10) * t * t * t;
}

float Lerp(float t, float a, float b) {
        return a + t * (b - a);
}

float Noise2D(float x, float y) {
        int X = static_cast<int>(floor(x)) & 255;
        int Y = static_cast<int>(floor(y)) & 255;
        float x_decimal = x - floor(x);
        float y_decimal = y - floor(y);
        vec2 topRight = vec2(x_decimal - 1.0, y_decimal - 1.0);
        vec2 topLeft = vec2(x_decimal, y_decimal - 1.0);
        vec2 bottomRight = vec2(x_decimal - 1.0, y_decimal);
        vec2 bottomLeft = vec2(x_decimal, y_decimal);
        int valueTopRight = permutation[permutation[X + 1] + Y + 1];
        int valueTopLeft = permutation[permutation[X] + Y + 1];
        int valueBottomRight = permutation[permutation[X + 1] + Y];
        int valueBottomLeft = permutation[permutation[X] + Y];
        float dotTopRight = dot(topRight, GetConstantVector(valueTopRight))
    ;
        float dotTopLeft = dot(topLeft, GetConstantVector(valueTopLeft));
        float dotBottomRight = dot(bottomRight, GetConstantVector(
    valueBottomRight));
        float dotBottomLeft = dot(bottomLeft, GetConstantVector(
    valueBottomLeft));
        float u = Fade(x_decimal);
        float v = Fade(y_decimal);
        return Lerp(u,
                Lerp(v, dotBottomLeft, dotTopLeft),
                Lerp(v, dotBottomRight, dotTopRight)
        );
}

float FractalBrownianMotion(float x, float y, int numOctaves) {
        float result = 0.0f;
```

```
48          float amplitude = 0.9f;
49          float frequency = 0.005f;
50          float gain = 0.5f;
51          float lacunarity = 2.0f;
52          for (int i = 0; i < numOctaves; ++i) {
53                  result += amplitude * Noise2D(x * frequency, y * frequency)
    ;
54                  amplitude *= gain;
55                  frequency *= lacunarity;
56          }
57          return result;
58  }
59
60  vector<float> generateFBMData(int width, int height, int numOctaves) {
61          vector<float> data(width * height);
62          float scale = 512.0f * 2.5;
63          for (int y = 0; y < height; ++y) {
64                  for (int x = 0; x < width; ++x) {
65                          float xf = static_cast<float>(x) / width;
66                          float yf = static_cast<float>(y) / height;
67                          float value = FractalBrownianMotion(xf * scale, yf
    * scale, numOctaves);
68                          data[(y * width) + x] = value;
69                  }
70          }
71          return data;
72  }
```

5.8 Texture

Le texture rivestono un ruolo fondamentale all'interno del progetto, in quanto costituiscono la base per la colorazione e il dettaglio visivo dei vari oggetti presenti nella scena, consentendo di ottenere una resa il più possibile realistica. Il loro utilizzo non si è limitato esclusivamente alla componente di colore, ma ha riguardato anche il *displacement*, ovvero lo spostamento dei vertici delle geometrie amplificate in base a determinate condizioni, seguendo la direzione della normale associata a ciascun vertice.

Nel progetto sono stati utilizzati tre tipi di texture: quelle di colore e quelle di displacement, ottenute caricando immagini esterne tramite la libreria `stb_image`, e una ulteriore texture ottenuta a partire dalla mappa di altezze generata con l'algoritmo Perlin Noise combinato con Fractal Brownian Motion (vedi sezione 5.7).

Una volta create o caricate, le texture vengono rese disponibili agli shader attraverso il loro identificatore OpenGL. In pratica, ciascuna texture è associata a un oggetto della GPU che può essere richiamato negli shader sotto forma di *sampler2D*, ovvero un collegamento alla texture che permette di accedere ai valori memorizzati al suo

interno, restituendo il colore o l'informazione associata a una specifica coordinata di texture.

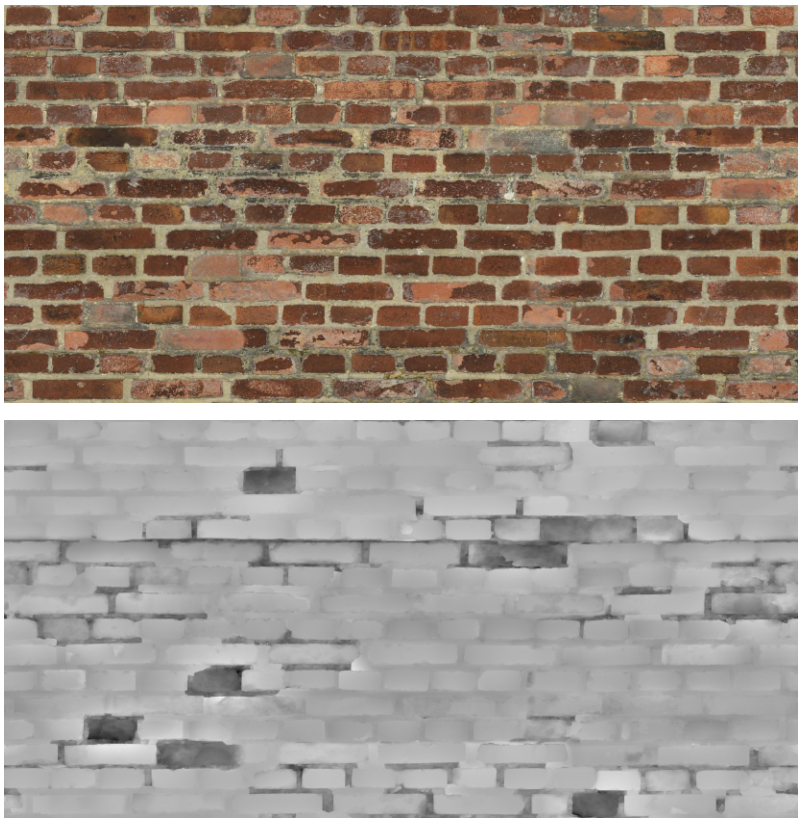

Figura 5.2: Esempio di texture di colore (in alto) e relativa texture di displacement (in basso).

5.8.1 Caricamento di texture di colore e displacement

Per quanto riguarda le texture di colore e di displacement utilizzate negli scenari del progetto, queste sono state caricate da file esterni tramite la funzione `loadSingleTexture`. Questa funzione, partendo dal percorso del file, si occupa inizialmente di leggere i dati dell'immagine tramite la libreria `stb_image`, verificando il tipo di immagine e adattando il formato della texture da utilizzare.

Una volta caricati i dati, viene generato un identificatore per la texture e viene riempita con i valori letti. In questa fase vengono inoltre definiti i parametri fondamentali per la sua gestione. Il parametro `GL_REPEAT` stabilisce che, quando le coordinate di texture fuoriescono dall'intervallo $[0, 1]$, l'immagine viene ripetuta in modo continuo, così da estendere la superficie senza interruzioni. Per quanto riguarda i filtri di interpolazione, `GL_LINEAR` indica che, se le coordinate non corrispondono esattamente a un punto dell'immagine, il colore viene calcolato facendo una media dei pixel più vicini. Il parametro `GL_LINEAR_MIPMAP_LINEAR`, invece, applica la stessa logica anche alle versioni ridotte dell'immagine (chiamate *mipmap*), che vengono generate automaticamente e servono a migliorare la resa visiva e le prestazioni quando la texture viene vista da lontano.

```
GLuint loadSingleTexture(const string& path) {
    int width, height, channels;
    stbi_set_flip_vertically_on_load(true);
    unsigned char* data = stbi_load(
        path.c_str(),
        &width,
        &height,
        &channels,
        0
    );

    if (!data) {
        return 0;
    }

    GLuint textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);

    GLenum format = GL_RGB;
    GLint internalFormat = GL_RGB8;

    if (channels == 1) {
        format = GL_RED;
        internalFormat = GL_R8;
    }
    else if (channels == 3) {
        format = GL_RGB;
        internalFormat = GL_RGB8;
    }
    else if (channels == 4) {
        format = GL_RGBA;
        internalFormat = GL_RGBA8;
    }
    else {
        stbi_image_free(data);
        return 0;
    }

    glTexImage2D(
        GL_TEXTURE_2D, 0, internalFormat, width,
        height, 0, format, GL_UNSIGNED_BYTE, data
    );
    glGenerateMipmap(GL_TEXTURE_2D);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR_MIPMAP_LINEAR);
```

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    stbi_image_free(data);
    return textureID;
}
```

5.8.2 Generazione della texture di altezze

La texture di altezze, utilizzata unicamente per la definizione delle variazioni geometriche del terreno all'interno della prima ambientazione, è stata invece generata direttamente a partire da valori numerici ottenuti dalla funzione `generateFBMData` definita nella sezione precedente.

In pratica, la funzione `createFloatTexture2D` prende un insieme di valori in virgola mobile e li organizza in una texture 2D, dove ogni punto memorizza un singolo valore che rappresenta l'altezza relativa di quella posizione. Queste informazioni vengono poi lette negli shader per determinare come spostare i vertici e ottenere superfici più dettagliate e irregolari.

Come per le altre texture, sono stati definiti i parametri di wrapping e filtraggio. Tuttavia, in questo caso non vengono generate mipmap, poiché la texture di altezze non viene usata per la resa visiva diretta, ma come base per calcoli geometrici che richiedono la massima precisione.

```
GLuint createFloatTexture2D(int width, int height, const vector<float>&
    data) {
    GLuint textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_R32F, width, height, 0, GL_RED,
    GL_FLOAT, data.data());

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    return textureID;
}
```

5.9 Collisioni

All'interno del progetto, in entrambe le ambientazioni, il personaggio animato non può muoversi liberamente nello spazio, ma deve evitare le zone già occupate dagli oggetti della scena.

Nel secondo ambiente questo vincolo è gestito tramite i cosiddetti *bounding volume*, ovvero parallelepipedi che racchiudono l'ingombro massimo di ciascun oggetto (personaggio compreso) lungo i tre assi principali, calcolati a partire dalle coordinate minime e massime dei suoi vertici. Quando il personaggio tenta di muoversi, viene controllato se il suo bounding volume andrebbe a intersecare quello di un altro oggetto. In caso di intersezione lo spostamento viene annullato, poiché significherebbe entrare in uno spazio già occupato.

Nel primo ambiente, invece, la gestione delle collisioni è differente, in quanto l'unico oggetto con cui il personaggio interagisce è il terreno montuoso, del quale deve seguire l'andamento. Qui si sfrutta direttamente la pipeline grafica del terreno: oltre ai dati della superficie, viene passata anche la posizione del personaggio, così che ogni patch possa averla a disposizione e confrontarla con la propria area. In questa pipeline particoalre non è presente un Fragment Shader, ma viene usata una *transform feedback*, che consente di salvare dalla pipeline alcune informazioni utili che possono essere utilizzate lato applicativo. In particolare, viene individuata la patch del terreno su cui si trova il personaggio e da essa si estraggono sia l'altezza (per aggiornare la sua posizione verticale) sia la normale (per attribuire un'inclinazione coerente con la pendenza del terreno).

5.10 Shaders

Gli shader rappresentano la componente principale alla base della realizzazione delle due ambientazioni del progetto. Grazie al loro utilizzo è stato possibile implementare un sistema di LOD dinamico, in cui sia il livello di dettaglio del terreno, sia quello degli oggetti di scena, viene calcolato in maniera adattiva. In questo modo gli elementi risultano più o meno definiti e realistici, in base alla loro distanza dal personaggio o dalla telecamera virtuale.

Nell'implementazione del progetto sono state definite diverse pipeline di rendering, ognuna caratterizzata da specifici shader e da compiti ben precisi. In generale, la maggior parte delle pipeline comprende Vertex Shader, Tessellation Control Shader, Tessellation Evaluation Shader, Geometry Shader e Fragment Shader. Sono presenti tuttavia anche pipeline semplificate: ad esempio, nei casi in cui non è necessario l'utilizzo della tessellazione o dell'amplificazione delle geometrie, i relativi shader non sono stati inclusi. In un caso particolare, già introdotto nella sezione 5.9, la pipeline è stata fatta terminare al Geometry Shader, senza quindi passare nè al Fragment Shader, nè alle fasi di clipping e rasterizzazione.

Per la maggior parte degli oggetti della scena è stata implementata una pipeline dedicata, poiché ciascun tipo di entità richiedeva parametri e comportamenti differenti rispetto agli altri. Questa suddivisione è resa possibile dalla struttura stessa di OpenGL, che rende infatti possibile la definizione di più programmi shader distinti, ognuno costituito dai propri stadi, selezionando di volta in volta quello da utilizzare tramite il comando `glUseProgram`.

Di seguito verrà fornita una spiegazione teorica delle principali pipeline realizzate, suddivise in base all'ambientazione. Verranno inoltre mostrati alcuni esempi di codi-

ce GLSL, limitati però alle parti più significative per comprendere il funzionamento degli shader.

5.10.1 Terreno (ambientazione 1)

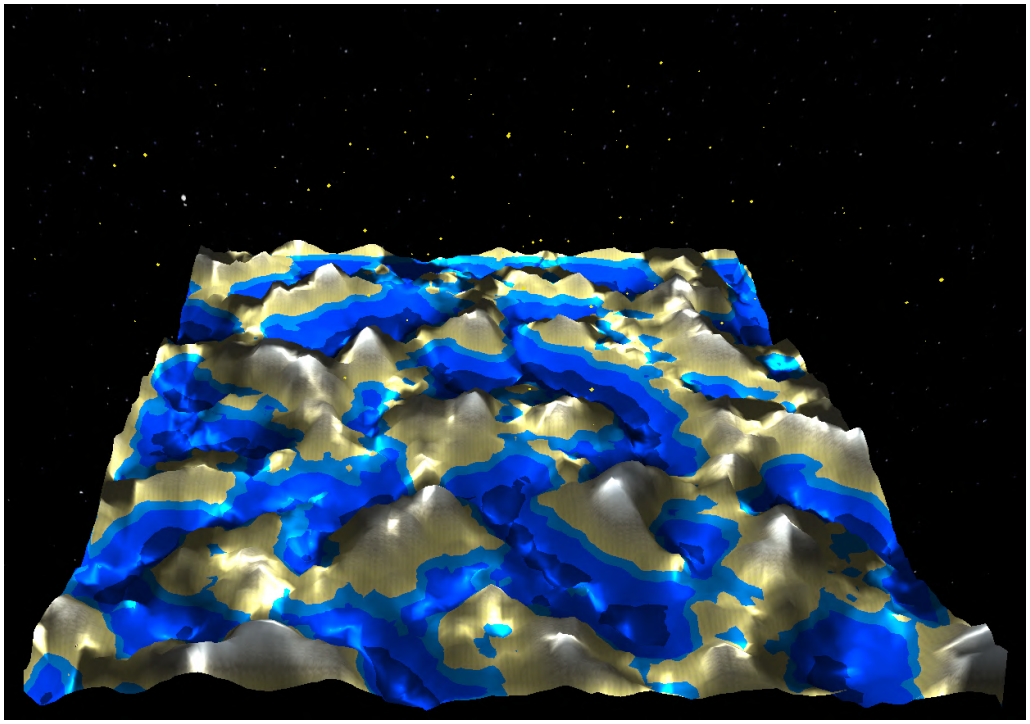

Figura 5.3: Vista dall'alto del terreno montuoso relativo alla prima ambientazione.

La pipeline utilizzata per il terreno montuoso della prima ambientazione mostra in maniera chiara come sia stato implementato il sistema di LOD adattivo grazie agli shader di tessellazione e di geometria.

Il Vertex Shader è stato realizzato in modalità *pass through*, dove cioè i dati dei vertici, già preparati lato CPU, vengono semplicemente inoltrati al resto della pipeline, senza ulteriori elaborazioni. Il cuor del sistema risiede nei Tessellation Shaders.

Nel Tessellation Control Shader si stabilisce il livello di suddivisione della mesh ed è qui che si concretizza il sistema di LOD adattivo. Invece di assegnare un unico valore di tessellazione a tutta la patch, il livello viene definito lato per lato, considerando la distanza tra la telecamera (o il personaggio) e il punto medio di ciascun lato. Questo approccio permette di regolare con precisione il dettaglio di ogni porzione di terreno in funzione della distanza dall'osservatore e garantisce che i lati condivisi tra patch adiacenti abbiano lo stesso livello di tessellazione, evitando fratture o disallineamenti nella mesh, un problema comune nelle pipeline senza gestione della continuità.

Il valore finale di tessellazione per ciascun lato viene determinato tramite interpolazioni lineari e normalizzazioni, così da ottenere transizioni graduali. Per ogni patch si tiene conto sia della distanza media dell'osservatore, sia del dislivello massimo interno, calcolato come differenza tra l'altezza minima e massima dei vertici. In

questo modo, le zone lontane o piatte vengono suddivise meno, mentre quelle vicine o con forti variazioni di quota vengono dettagliate maggiormente, ottenendo un compromesso ottimale tra qualità visiva ed efficienza computazionale. I livelli di tessellazione esterna seguono la logica appena descritta, mentre i livelli interni vengono determinati prendendo il valore massimo tra i livelli di tessellazione dei lati corrispondenti, in modo da garantire coerenza interna.

```
void main() {
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].
    gl_Position;

    if (gl_InvocationID == 0) {

        vec3 p[4];
        float h[4];
        for (int i = 0; i < 4; ++i) {
            p[i] = gl_in[i].gl_Position.xyz;
            vec2 uv = p[i].xz * UV_SCALE;
            h[i] = texture(u_fbmTexture, uv).r * HEIGHT_SCALE;
        }

        float minH = min(min(h[0], h[1]), min(h[2], h[3]));
        float maxH = max(max(h[0], h[1]), max(h[2], h[3]));
        float deltaH = maxH - minH;

        vec3 center[4];
        center[0] = (p[0] + p[3]) * 0.5;
        center[1] = (p[0] + p[1]) * 0.5;
        center[2] = (p[1] + p[2]) * 0.5;
        center[3] = (p[2] + p[3]) * 0.5;

        vec3 basePosition;
        if (useCharacterToTess) {
            basePosition = characterPosition;
        }
        else {
            basePosition = cameraPosition;
        }

        for (int i = 0; i < 4; ++i) {
            float dist = length(basePosition - center[i]);
            float tess;
            if (dist > MAX_DIST) {
                tess = MIN_TES;
            }
            else {
                float distFactor = dist / MAX_DIST;
                float heightFactor = clamp(
                    deltaH / MAX_HEIGHT_DIFF,
```

```
                    0.0,
                    1.0
                );
                float lodFactor = 0.6 * distFactor +
                    0.4 * (1.0 - heightFactor);
                tess = mix(MAX_TES, MIN_TES, lodFactor);
            }
            gl_TessLevelOuter[i] = tess;
        }

        gl_TessLevelInner[0] = max(
            gl_TessLevelOuter[1],
            gl_TessLevelOuter[3]
        );
        gl_TessLevelInner[1] = max(
            gl_TessLevelOuter[0],
            gl_TessLevelOuter[2]
        );
    }
}
```

Nel Tessellation Evaluation Shader, la suddivisione calcolata dal TCS viene tradotta in geometria effettiva. Ogni vertice generato viene posizionato tramite un'interpolazione bilineare dei quattro vertici originali della patch quadrilaterale e successivamente elevato lungo l'asse verticale utilizzando la mappa procedurale di displacement basata su Fractal Brownian Motion di Perlin Noise. L'utilizzo di una funzione continua per la generazione dell'altezza semplifica notevolmente la gestione dei confini, evitando buchi o discontinuità tra patch adiacenti, problema che invece si presenta in pipeline che usano texture locali o segmentate. Inoltre, in questa fase si calcolano le normali dei nuovi vertici tramite differenze locali finite, ottenendo una stima coerente della superficie, fondamentale per l'illuminazione successiva.

```
void main() {
    vec3 p0 = gl_in[0].gl_Position.xyz;
    vec3 p1 = gl_in[1].gl_Position.xyz;
    vec3 p2 = gl_in[2].gl_Position.xyz;
    vec3 p3 = gl_in[3].gl_Position.xyz;

    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;

    vec3 pos = ((1.0 - u) * (1.0 - v) * p0) +
        (u * (1.0 - v) * p1) +
        (u * v * p2) +
        ((1.0 - u) * v * p3);

    vec2 uv = pos.xz * UV_SCALE;
    float height = texture(u_fbmTexture, uv).r;
```

```
    pos.y += height * HEIGHT_SCALE;

    worldPos = model * vec4(pos, 1.0);

    vec2 delta = vec2(1.0 / terrainSize_tes);

    float hL = texture(u_fbmTexture, uv - vec2(delta.x, 0.0)).r *
    HEIGHT_SCALE;
    float hR = texture(u_fbmTexture, uv + vec2(delta.x, 0.0)).r *
    HEIGHT_SCALE;
    float hD = texture(u_fbmTexture, uv - vec2(0.0, delta.y)).r *
    HEIGHT_SCALE;
    float hU = texture(u_fbmTexture, uv + vec2(0.0, delta.y)).r *
    HEIGHT_SCALE;

    vec3 dx = vec3(2.0 * delta.x, hR - hL, 0.0);
    vec3 dz = vec3(0.0, hU - hD, 2.0 * delta.y);
    vec3 normal = normalize(cross(dz, dx));
    tes_normal = mat3(transpose(inverse(model))) * normal;
}
```

Successivamente, all'interno del Geometry Shader avviene la generazione procedurale della vegetazione, composta da erba e alghe. Per ogni triangolo ricevuto, lo shader esegue innanzitutto un culling rispetto alla telecamera, utilizzando la normale del triangolo e la direzione verso l'osservatore, evitando così di generare geometria non visibile e risparmiando risorse computazionali.

La generazione dell'erba (funzione `generateGrass`) sfrutta alcuni punti chiave del triangolo, come i vertici estremi, i centri dei lati e il centro del triangolo. Per ciascun punto vengono create foglie singole, definite come dei triangoli orientati lungo il vettore tangente alla superficie e ruotati casualmente entro un angolo massimo prestabilito. L'oscillazione naturale dovuta al vento è simulata tramite una funzione sinusoidale dipendente dal tempo, con una fase casuale per ogni foglia per evitare pattern ripetitivi.

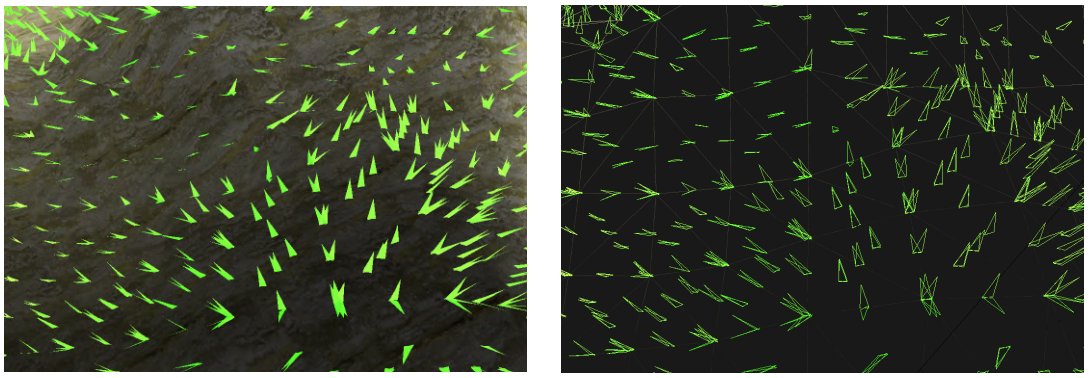

Figura 5.4: Esempio di erba generata tramite gli shader in modalità `FILL` (sinistra) e in modalità `LINE` (destra).

Analogamente, le alghe (funzione `generateKelps`) vengono costruite come segmenti rettangolari disposti uno sopra l'altro lungo una direzione principale, con piccole variazioni angolari pseudo-casuali tra un segmento e l'altro. Anche qui, un'oscillazione sinusoidale conferisce un effetto dinamico realistico, mentre le normali vengono calcolate coerentemente per garantire un'illuminazione corretta.

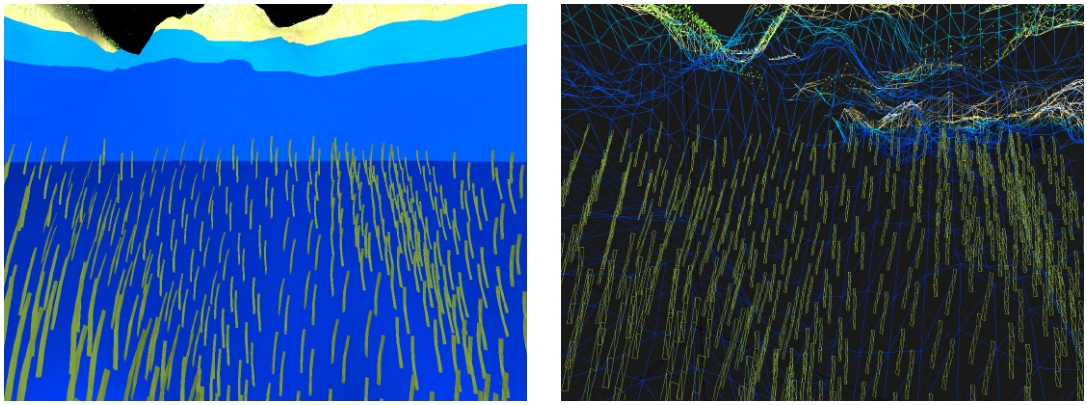

Figura 5.5: Esempio di alghe generate tramite gli shader in modalità `FILL` (sinistra) e in modalità `LINE` (destra).

All'interno del Geometry Shader avviene anche un'altra operazione di LOD basata sulla distanza della patch dal punto di riferimento (telecamera o personaggio). Solamente le patch vicine all'osservatore vengono popolate con erba o alghe, riducendo così il numero di vertici generati per le zone lontane e migliorando le prestazioni complessive. In aggiunta, anche l'altezza del terreno determina la comparsa di erba o alghe: l'erba viene generata solo su superfici comprese tra un minimo e un massimo prefissati, mentre le alghe compaiono esclusivamente nelle zone più basse della mappa.

```
void main() {
    vec3 normal = normalize(cross(
        worldPos[1].xyz - worldPos[0].xyz,
        worldPos[2].xyz - worldPos[0].xyz
    ));
    if (normal.y < 0.0) normal = -normal;

    vec3 triCenter = (
        worldPos[0].xyz +
        worldPos[1].xyz +
        worldPos[2].xyz
    ) / 3.0;
    vec3 toCamera = normalize(cameraPosition - triCenter);

    float visibility = dot(normal, toCamera);
    if (visibility < 0.0) return;

    for (int i = 0; i < 3; ++i) {
        gl_Position = proj * view * worldPos[i];
```

```
        gs_worldPos = worldPos[i];
        gs_normal = normalize(tes_normal[i]);
        gs_isGrass = 0;
        gs_isKelp = 0;
        EmitVertex();
    }
    EndPrimitive();


    vec3 basePosition;
    if (useCharacterToTess) {
        basePosition = characterPosition;
    }
    else {
        basePosition = cameraPosition;
    }


    vec3 center = (
        worldPos[0].xyz +
        worldPos[1].xyz +
        worldPos[2].xyz
    ) / 3.0;
    bool isClose = (length(basePosition - center) < MAX_DISTANCE);

    if (isClose) {
        float centerHeight = center.y;

        if (centerHeight > GRASS_MIN_TERRAIN_HEIGHT &&
            centerHeight < GRASS_MAX_TERRAIN_HEIGHT) {
            generateGrass();
        }
        else if (centerHeight < KELP_MAX_TERRAIN_HEIGHT) {
            generateKelps();
        }
    }
}
```

Infine, all'interno del Fragment Shader viene assegnato il colore ad ogni elemento. Per prima cosa, viene identificato il tipo di elemento da renderizzare tramite l'utilizzo di variabili booleane assegnate ad ogni vertice, che indicano se si tratta di erba, alghe o terreno. In base a queste informazioni è poi possibile assegnare a ciascun vertice il colore appropriato: per l'erba viene utilizzato un colore verde brillante, per le alghe un colore verde scuro, mentre per il terreno un colore determinato tramite una funzione di blending fra texture (`blendTextures`). In questo caso, l'altezza del vertice nello spazio funge da parametro principale per determinare la transizione tra le diverse texture, permettendo di rappresentare in maniera naturale zone acquatiche, sabbiose, erbose, rocciose o innevate.

In questa fase il colore viene modulato dall'effetto dell'illuminazione. Il modello utilizzato è quello di *Phong*, composto da tre contributi principali: ambientale, diffuso e speculare. La componente ambientale rappresenta la luce diffusa presente nell'ambiente, garantendo che anche le zone in ombra siano percepibili. La componente diffusa dipende dall'orientamento della superficie rispetto alla luce, evidenziando pendenze e dettagli del terreno. Infine, la componente speculare, calcolata secondo la variante di *Blinn-Phong*, genera i riflessi brillanti che cambiano in base all'angolo tra la luce, la normale della superficie e la direzione dello sguardo dell'osservatore, conferendo realismo ai materiali. Per ottenere questi effetti, all'interno della funzione `computeLighting` la normale del vertice viene normalizzata e usata per determinare sia la componente diffusa sia quella speculare. La direzione della luce e quella del verso della camera vengono combinate per calcolare correttamente le intensità delle due componenti. Infine, il risultato dell'illuminazione viene sommato al colore base del frammento, ottenendo un output visivo coerente che integra colori, texture e luce.

```
vec3 computeLighting(vec3 normal, vec3 fragPos, vec3 viewPos, vec3
    baseColor) {
    vec3 norm = normalize(normal);

    vec3 lightDir = normalize(light.position - fragPos);

    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * light.color;

    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * light.color;

    vec3 viewDir = normalize(viewPos - fragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32.0);
    float specularStrength = 0.5;
    vec3 specular = specularStrength * spec * light.color;

    vec3 lighting = (ambient + diffuse + specular) * baseColor * light.
    power;
    return lighting;
}
```

```
void main() {
    float h = gs_worldPos.y;
    vec2 uv = gs_worldPos.xz;
    vec3 baseColor;

    if (gs_isGrass == 1) {
        baseColor = vec3(0.2, 0.8, 0.1);
    }
    else if (gs_isKelp == 1) {
```

```
        baseColor = vec3(0.2, 0.25, 0.1);
    }
    else {
        baseColor = blendTextures(h, uv).rgb;
    }

    vec3 lighting = computeLighting(
        normalize(gs_normal),
        gs_worldPos.xyz,
        ViewPos,
        baseColor
    );
    FragColor = vec4(lighting, 1.0);
}
```

5.10.2 Transform Feedback (ambientazione 1)

Il caso più particolare all'interno del progetto è rappresentato dalla pipeline con *Transform Feedback*, già anticipata nella sezione 5.9 in riferimento alla gestione delle collisioni con il terreno montuoso.

Come nella pipeline standard del terreno della prima ambientazione, i primi tre stadi di questa pipeline (Vertex Shader, Tessellation Control Shader e Tessellation Evaluation Shader) restano invariati e operano esattamente come già descritto: i vertici iniziali vengono processati, suddivisi adattivamente e proiettati nello spazio del mondo. La differenza principale riguarda invece la parte finale della pipeline. Qui non viene utilizzato alcun Fragment Shader, poichè non si vuole effettuare un ulteriore rendering a schermo del terreno montuoso, ma piuttosto catturare informazioni geometriche direttamente dalla GPU e renderle disponibili al lato CPU.

Questo è reso possibile dalla modalità *Transform Feedback*, una funzionalità di OpenGL che permette di registrare i valori di output di un determinato stadio della pipeline (in questo caso il Geometry Shader) e memorizzarli all'interno di un buffer dedicato. Invece di inviare i dati al rasterizer e al framebuffer, essi vengono intercettati e salvati in una struttura accessibile dall'applicazione.

Nello specifico, la pipeline è stata configurata affinchè il Geometry Shader emetta, non più primitive grafiche, ma punti che contengono due informazioni fondamentali: la posizione interpolata del personaggio sul terreno (`characterPositionTransformFeedback`) e la normale locale del terreno in quel punto (`characterNormalTransformFeedback`).

Questi valori, dichiarati come *varyings*, vengono registrati tramite la funzione `glTransformFeedbackVaryings`, che specifica ad OpenGL quali variabili devono essere catturate nel buffer assegnato alla Transform Feedback. Quando la pipeline elabora le patch del terreno, il Geometry Shader calcola se la posizione del personaggio, proiettata sul piano XZ, ricade all'interno di un triangolo tessellato. In caso positivo, l'altezza corrispondente (coordinata Y) viene determinata per interpolazione baricentrica e la normale locale del terreno viene calcolata come prodotto

vettoriale tra i lati del triangolo. Entrambe le informazioni sono poi emesse come singolo vertice di tipo `point` e catturate dal Transform Feedback. Ne consegue che, per ciascun frame, il buffer conterrà al più un solo record valido, corrispondente alla patch effettivamente attraversata dal personaggio, mentre in assenza di corrispondenze (ad esempio se il personaggio si trova fuori dai limiti del terreno), nessun valore viene aggiornato.

Lato CPU, i dati vengono letti tramite la funzione `glMapBuffer`, che consente di accedere direttamente al contenuto del buffer. In questo modo è possibile aggiornare in tempo reale l'altezza del personaggio rispetto al terreno e ottenere la normale corrispondente, utile per definirne l'orientamento in accordo con la pendenza locale.

Il buffer, dimensionato per contenere coppie di vettori tridimensionali (posizione e normale), è gestito con aggiornamenti dinamici a ogni frame (`GL_DYNAMIC_READ`), così da poter seguire in tempo reale il movimento del personaggio sulla superficie del terreno. Grazie a questo approccio, il personaggio si muove sul terreno rispettandone fedelmente sia l'altimetria, sia l'inclinazione, senza necessità di implementare esternamente algoritmi di collisione veri e propri.

```
void main() {
    vec3 center = (
        worldPos[0].xyz +
        worldPos[1].xyz +
        worldPos[2].xyz
    ) / 3.0;

    vec3 p0 = worldPos[0].xyz;
    vec3 p1 = worldPos[1].xyz;
    vec3 p2 = worldPos[2].xyz;
    vec2 charXZ = characterPosition.xz;
    bool inside = isPointInTriangle(charXZ, p0.xz, p1.xz, p2.xz);
    float interpolatedY = -100.0;

    if (inside) {
        vec2 v0 = p1.xz - p0.xz;
        vec2 v1 = p2.xz - p0.xz;
        vec2 v2 = charXZ - p0.xz;

        float d00 = dot(v0, v0);
        float d01 = dot(v0, v1);
        float d11 = dot(v1, v1);
        float d20 = dot(v2, v0);
        float d21 = dot(v2, v1);

        float denom = d00 * d11 - d01 * d01;
        float v = (d11 * d20 - d01 * d21) / denom;
        float w = (d00 * d21 - d01 * d20) / denom;
        float u = 1.0 - v - w;

```

```
31          interpolatedY = u * p0.y + v * p1.y + w * p2.y;
32
33          vec3 edge1 = p1 - p0;
34          vec3 edge2 = p2 - p0;
35          vec3 normal = normalize(cross(edge1, edge2));
36
37          characterPositionTransformFeedback = vec3(
38              characterPosition.x,
39              interpolatedY,
40              characterPosition.z
41          );
42          characterNormalTransformFeedback = normal;
43
44          EmitVertex();
45          EndPrimitive();
46      }
47  }
```

Listing 5.1: Geometry Shader per il calcolo della posizione e normale del personaggio sul terreno con Transform Feedback.

5.10.3 Stelle (ambientazione 1)

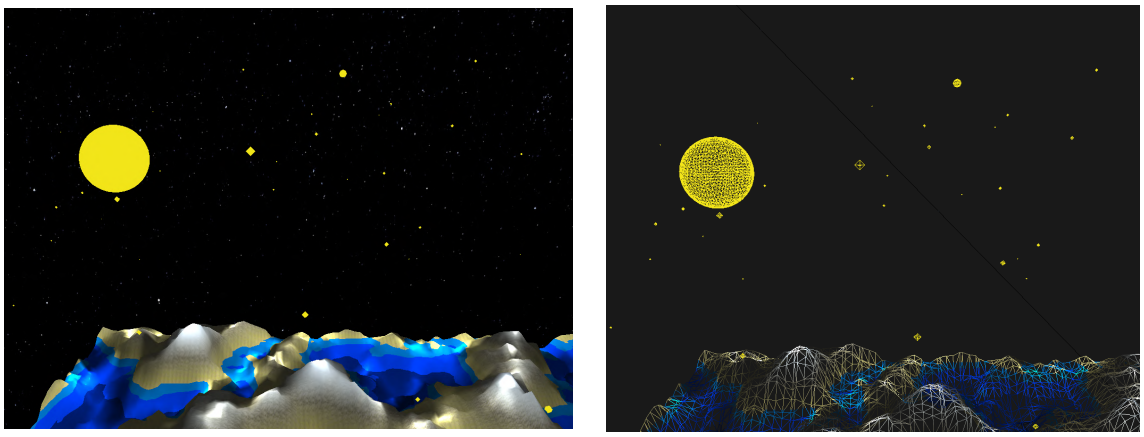

Figura 5.6: Esempio di stelle generata tramite gli shader in modalità `FILL` (sinistra) e in modalità `LINE` (destra).

La pipeline per la generazione di una stella costituisce un caso di pipeline completa, ma di semplice comprensione. Comprende tutti gli stadi descritti: Vertex Shader, Tessellation Control Shader, Tessellation Evaluation Shader, Geometry Shader e Fragment Shader, ognuno con un ruolo preciso nella costruzione della geometria sferica a partire dagli otto ottanti triangolari iniziali. Analogamente a quanto descritto per il terreno montuoso, anche qui è implementato un sistema di LOD adattivo basato sulla distanza, che regola il dettaglio dei triangoli in funzione della vicinanza alla telecamera, garantendo così una sfera visivamente coerente e ben definita.

Il Vertex Shader, come per il terreno montuoso, si limita a riceve le posizioni dei vertici e il centro della sfera corrispondente e ad inoltrarli al Tessellation Control

Shader, convertendo le coordinate nel formato corretto e predisponendo il centro della sfera in output.

Il Tessellation Control Shader riceve le patch triangolari e decide il livello di suddivisione di ciascuna patch in base alla distanza tra la telecamera (o personaggio) e il centro della sfera. I triangoli lontani ricevono una tessellazione minima, mentre quelli più vicini vengono suddivisi maggiormente per aumentare il dettaglio della superficie sferica. La suddivisione è uniforme sia sui lati esterni sia all'interno della patch, assicurando transizioni regolari e senza fratture tra triangoli adiacenti.

All'interno del Tessellation Evaluation Shader ogni vertice viene poi posizionato tramite interpolazione baricentrica dei tre vertici originali della patch e successivamente proiettato sulla superficie della sfera. Per farlo, si calcola il raggio come distanza tra il vertice originale e il centro della sfera, si normalizza il vettore dal centro al vertice e lo si scala in modo da riportare il vertice sulla superficie sferica. In questo modo, tutti i vertici generati vengono distribuiti in maniera coerente sulla sfera, evitando distorsioni o spigoli vivi. L'output dello shader include le coordinate finali nello spazio del mondo, pronte per essere elaborate dal Geometry Shader.

Il Geometry Shader riceve i triangoli generati dal TES e calcola la posizione di ciascun vertice rispetto al centro del triangolo, senza modificare la geometria, ma garantendo che siano compatibili con eventuali trasformazioni successive. Per ogni vertice viene applicata la trasformazione dalle coordinate mondo a quelle di vista, attraverso la matrice di vista e quella di proiezione. L'output comprende sia la posizione trasformata sia una copia delle coordinate nello spazio mondo, utile per eventuali effetti o calcoli aggiuntivi.

Infine, il Fragment Shader assegna il colore finale dei pixel della sfera. In questa implementazione, la stella è resa con un colore uniforme giallo chiaro, che simula la superficie luminosa della stella senza introdurre dettagli complessi.

```
void main() {
    vec3 p0 = gl_in[0].gl_Position.xyz;
    vec3 p1 = gl_in[1].gl_Position.xyz;
    vec3 p2 = gl_in[2].gl_Position.xyz;

    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;
    float w = 1.0 - u - v;

    vec3 pos = w * p0 + u * p1 + v * p2;

    vec3 sphereCenter = tcCenter[0];

    float r = length(p0 - sphereCenter);
    vec3 dir = normalize(pos - sphereCenter);
    vec3 sphericalPos = sphereCenter + dir * r;

    worldPos = model * vec4(sphericalPos, 1.0);
```

```
}
```

Listing 5.2: Tessellation Evaluation Shader per la generazione della sfera.

5.10.4 Personaggio (ambientazioni 1 e 2)

La pipeline dedicata al personaggio animato rappresenta un caso di pipeline più tradizionale e lineare, senza shader di tessellazione e di geometria, in cui l'attenzione è rivolta allo skinning delle ossa e all'illuminazione per il rendering finale.

Il Vertex Shader riceve in ingresso tutti i dati calcolati lato CPU (vedi sezione 5.5) e passati in modo appropriato: i vertici del modello, le normali, le coordinate di texture e i dati per lo skinning, cioè gli indici delle ossa e i pesi associati. La funzione principale di questo shader è applicare la trasformazione delle ossa ai vertici del modello. Ogni vertice viene influenzato dalle quattro ossa più rilevanti, combinando le matrici di trasformazione secondo i pesi associati. In questo modo il modello può deformarsi in modo realistico seguendo l'animazione calcolata dall'applicazione.

Una volta applicato lo skinning, il Vertex Shader calcola la posizione finale del vertice nel World Space e trasforma la normale utilizzando sia la matrice di trasformazione del modello sia la trasformazione delle ossa, garantendo un'illuminazione corretta. Le coordinate di texture, così come posizione e normale, vengono inoltrate al Fragment Shader. Infine, la posizione del vertice viene proiettata nello spazio della telecamera tramite le matrici di vista e di proiezione.

Il Fragment Shader utilizza lo stesso modello di illuminazione già applicato al terreno montuoso (*modello di Phong*), basato su luce puntiforme con componenti ambientale, diffusa e speculare. Il colore finale del modello è ottenuto dalla combinazione tra la texture e la luce, ottenendo un effetto realistico che valorizza le forme del modello.

```
void main() {
    mat4 skinMatrix =
        aWeights[0] * bones[aBoneIDs[0]] +
        aWeights[1] * bones[aBoneIDs[1]] +
        aWeights[2] * bones[aBoneIDs[2]] +
        aWeights[3] * bones[aBoneIDs[3]];

    vec4 skinnedPos = skinMatrix * vec4(aPos, 1.0);
    FragPos = vec3(model * skinnedPos);

    mat3 normalMatrix = transpose(inverse(mat3(model * skinMatrix)));
    Normal = normalize(normalMatrix * aNormal);

    TextCoords = aTextCoords;

    gl_Position = proj * view * vec4(FragPos, 1.0);
}
```

Listing 5.3: Vertex Shader per lo skinning del personaggio.

5.10.5 Terreno (ambientazione 2)

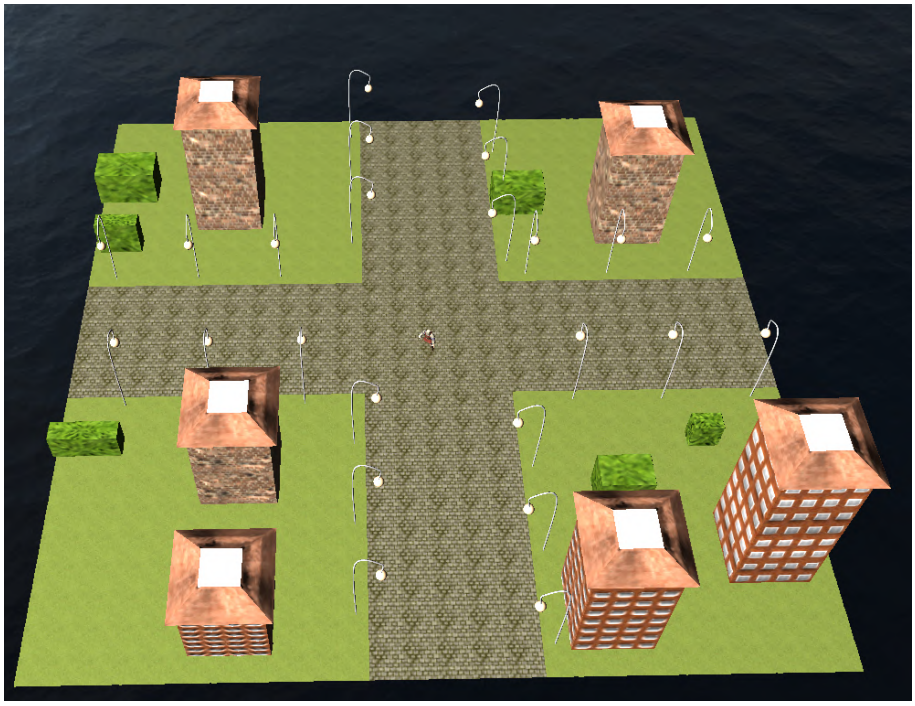

Figura 5.7: Paesaggio urbano della seconda ambientazione, visto dall'alto.

La pipeline dedicata al terreno della seconda ambientazione introduce una gestione particolare delle patch, che vengono elaborate in due passaggi distinti: uno per le aree di erba e uno per le strade.

Il Vertex Shader svolge un ruolo puramente di pass-through. Riceve le coordinate dei vertici della patch e un vettore che indica quali lati confinano con superfici di tipo diverso (erba o strada). Questo vettore, calcolato lato CPU dalla funzione `generatePatches`, specifica per ciascun lato se il displacement debba essere azzerato, così da garantire continuità tra patch diverse ed evitare buchi o disallineamenti visivi. Lo shader non effettua quindi trasformazioni, ma si limita a inoltrare correttamente i dati al Tessellation Control Shader.

Il Tessellation Control Shader mantiene i dati in uscita e imposta i fattori di tessellazione, determinando il livello di suddivisione di ciascuna patch. Come già visto per il terreno montuoso, questi valori vengono scelti dinamicamente in base alla distanza dalla telecamera o dal personaggio, implementando un meccanismo di LOD adattivo. In particolare, lo shader (proprio come per il terreno montuoso) calcola i punti medi dei lati della patch, ne misura la distanza dalla posizione di riferimento e assegna livelli di tessellazione più elevati nelle aree vicine e più bassi nelle zone lontane, garantendo al tempo stesso la continuità tra patch adiacenti ed evitando la formazione di buchi.

Nel Tessellation Evaluation Shader, per ogni nuovo vertice vengono calcolate le coordinate baricentriche normalizzate e utilizzate per ricostruire la posizione del punto all'interno della patch. A partire da queste coordinate viene campionata

la texture di displacement, che definisce l'altezza del terreno. Se però il vertice si trova su un lato che confina con una superficie di tipo diverso, il displacement viene azzerato in base al vettore ricevuto dal Vertex Shader, così da garantire una transizione continua tra erba e strada. Infine, lo shader calcola le normali tramite differenze finite, con la stessa tecnica già adottata per il terreno montuoso.

Il Geometry Shader riceve i triangoli prodotti dal Tessellation Evaluation Shader e li proietta nello spazio della telecamera. Per ciascun vertice inoltra le informazioni necessarie (posizione, normale e coordinate di texture), senza alterare ulteriormente la geometria. In questo modo funge da collegamento diretto con il Fragment Shader.

Il Fragment Shader completa la pipeline applicando il modello di illuminazione di Phong già utilizzato in precedenza. Il colore finale è ottenuto combinando l'effetto della luce con la texture di base, che varia a seconda che la patch rappresenti erba o strada. In questo modo il terreno appare visivamente coerente, con transizioni fluide tra le diverse superfici e un'illuminazione realistica.

```
vec3 getDisplacedPos(vec2 uv, vec3 p0, vec3 p1, vec3 p2, vec3 p3) {
    float u = uv.x;
    float v = uv.y;

    vec3 pos = ((1.0 - u) * (1.0 - v) * p0)
        + (u * (1.0 - v) * p1)
        + (u * v * p2)
        + ((1.0 - u) * v * p3);

    float height = texture(texture1, uv).r;
    float epsilon = 0.001;

    bool onLeft = abs(uv.x) < epsilon;
    bool onBottom = abs(uv.y) < epsilon;
    bool onRight = abs(uv.x - 1.0) < epsilon;
    bool onTop = abs(uv.y - 1.0) < epsilon;

    if ((onTop && tcDisplace[0].x == 1.0f) ||
        (onRight && tcDisplace[0].y == 1.0f) ||
        (onBottom && tcDisplace[0].z == 1.0f) ||
        (onLeft && tcDisplace[0].w == 1.0f)) {
        height = 0.0f;
    }

    if ((onLeft && onBottom) ||
        (onLeft && onTop) ||
        (onRight && onBottom) ||
        (onRight && onTop)) {
        height = 0.0f;
    }

    return pos + vec3(0.0, height * SCALE, 0.0);
```

```
}

void main() {
    vec3 p0 = gl_in[0].gl_Position.xyz;
    vec3 p1 = gl_in[1].gl_Position.xyz;
    vec3 p2 = gl_in[2].gl_Position.xyz;
    vec3 p3 = gl_in[3].gl_Position.xyz;

    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;
    vec2 uv = vec2(u, v);

    vec3 pos = getDisplacedPos(uv, p0, p1, p2, p3);

    vec2 delta = vec2(1.0 / 5.0);

    float hL = texture(texture1, uv - vec2(delta.x, 0.0)).r * SCALE;
    float hR = texture(texture1, uv + vec2(delta.x, 0.0)).r * SCALE;
    float hD = texture(texture1, uv - vec2(0.0, delta.y)).r * SCALE;
    float hU = texture(texture1, uv + vec2(0.0, delta.y)).r * SCALE;

    vec3 dx = vec3(2.0 * delta.x, (hR - hL), 0.0);
    vec3 dz = vec3(0.0, (hU - hD), 2.0 * delta.y);
    vec3 normal = normalize(cross(dz, dx));

    worldPos = model * vec4(pos, 1.0);
    normalTES = normalize(transpose(inverse(mat3(model))) * normal);
    tesUV = uv;
}
```

Listing 5.4: Tessellation Evaluation Shader per determinare il displacement dei vertici delle patch.

5.10.6 Edifici, siepi e tetti (ambientazione 2)

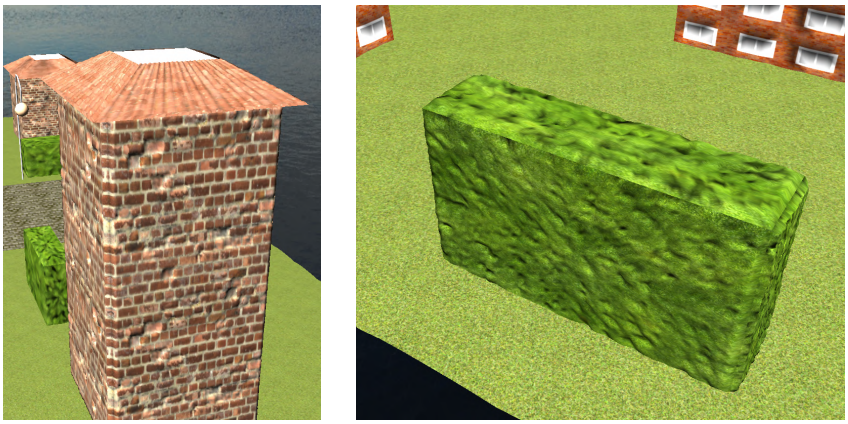

Figura 5.8: Esempio di edificio con tetto (sinistra) e di siepe (destra) generati tramite shader in modalità `FILL`.

Un esempio di gestione simile a quella del terreno della seconda ambientazione è rappresentato dalla pipeline utilizzata per edifici e siepi, ma anche da quella utilizzata per i tetti. Poiché queste pipeline condividono la stessa struttura di base e differiscono solo per alcuni dettagli, verranno descritte insieme, mettendo in evidenza le caratteristiche che le distinguono.

Il Vertex Shader, come già avviene per la maggior parte delle altre pipeline definite, è di tipo *pass-through*, che si limita a ricevere i dati inviati dall'applicazione (posizione dei vertici e relative normali) e a passarli direttamente allo stadio successivo, senza applicare trasformazioni aggiuntive.

Anche in questo caso, come già descritto per il terreno della seconda ambientazione, il Tessellation Control Shader definisce i valori di tessellazione seguendo inizialmente lo stesso approccio usato per il terreno montuoso. In particolare, per ogni patch vengono calcolati i punti medi dei lati e misurata la loro distanza dalla telecamera (o dal personaggio, a seconda del parametro di configurazione). Questa misura non tiene conto della componente verticale, ma viene calcolata solo sul piano orizzontale. In questo modo, anche avvicinandosi alla base di un edificio, è possibile ottenere una suddivisione dettagliata anche delle parti più alte, come i tetti, che il personaggio non potrebbe raggiungere direttamente.

I livelli di tessellazione vengono inizialmente assegnati in funzione di queste distanze, con valori più elevati per le aree vicine e più bassi per quelle lontane, implementando quindi un LOD adattivo. Per garantire però una suddivisione proporzionata, questi valori vengono corretti in base alla lunghezza effettiva dei lati della patch. La procedura consiste nel calcolare la lunghezza di ciascun lato, stimare i rapporti con i livelli iniziali, ricavarne la media per ciascun asse e, infine, moltiplicare questa densità media per le lunghezze dei lati stessi, ottenendo i livelli finali.

Le versioni utilizzate per edifici, siepi e tetti condividono la stessa logica generale, differenziandosi unicamente per i valori dei parametri di distanza minima e massima che regolano il calcolo, che risultano più contenuti per i tetti e leggermente più ampi per edifici e siepi, così da modulare con maggiore precisione il livello di dettaglio in funzione della tipologia di oggetto.

Per ogni vertice ottenuto dalla suddivisione, il Tessellation Evaluation Shader calcola le coordinate baricentriche normalizzate all'interno della patch e utilizza un'interpolazione bilineare per determinare la posizione iniziale del punto. Successivamente viene campionata la texture di displacement e applicata lungo la normale locale del vertice, scalata di un fattore arbitrario.

Una caratteristica importante di questo shader è la gestione dei bordi delle patch e degli spigoli delle facce. Per determinare se un vertice generato debba ricevere il displacement o meno, al TES vengono passati i vertici originali della geometria di partenza: per ogni tetto, tutti i vertici del tronco di piramide corrispondente, e per ogni edificio o siepe, i vertici del parallelepipedo. Lo shader confronta la posizione interpolata del vertice con questi punti originali, verificando se si trova su uno spigolo tramite test geometrici. In particolare, nel caso dei tronchi di piramide si controlla se il punto giace su uno dei segmenti che collegano i vertici, mentre nei parallelepipedi

si verifica se almeno due componenti coincidono con quelle di un vertice originale. Se il vertice si trova su uno spigolo, il displacement viene annullato, garantendo continuità tra le superfici adiacenti ed evitando sovrapposizioni o disallineamenti. Per i tetti, oltre agli spigoli, il displacement viene disabilitato anche sulle facce orizzontali, mentre per edifici e siepi la logica si basa sulla strategia di displacement appena spiegata.

Infine, le normali sono calcolate tramite differenze finite, valutando la posizione dei punti vicini, ottenendo delle normali corrette anche in presenza di displacement. In entrambe le versioni del TES, le coordinate di texture vengono rimappate in funzione della normale della faccia per garantire un orientamento e un mapping corretto delle texture su tutte le facce.

Infine, il Geometry Shader e il Fragment Shader di entrambe le tipologie di geometria sono strutturati e organizzati in maniera analoga alle altre pipeline. Il Geometry Shader si limita a trasferire le informazioni nello spazio della telecamera, applicando le matrici di trasformazione fondamentali, mentre il Fragment Shader gestisce l'illuminazione utilizzando il modello di Phong.

```
vec3 getDisplacedPos(vec2 uv, vec3 p0, vec3 p1, vec3 p2, vec3 p3) {
    float u = uv.x;
    float v = uv.y;
    vec3 pos = ((1.0 - u) * (1.0 - v) * p0) + (u * (1.0 - v) * p1) + (u * v
     * p2) + ((1.0 - u) * v * p3);

    vec2 mappedUV = remapUV(uv, normal_tcs[0]);
    float height = texture(texture1, mappedUV).r;

    float epsilon = 0.001;
    bool onBlockBorder = false;

    int edges[24] = int[](
        0, 1, 1, 2, 2, 3, 3, 0,  // base inferiore
        4, 5, 5, 6, 6, 7, 7, 4,  // base superiore
        0, 4, 1, 5, 2, 6, 3, 7   // spigoli verticali
    );

    int baseIndex = (gl_PrimitiveID / 6) * 8;

    for (int i = 0; i < 24; i += 2) {
        vec3 a = originalPoints[baseIndex + edges[i]];
        vec3 b = originalPoints[baseIndex + edges[i + 1]];

        if (isPointOnSegment(pos, a, b, epsilon)) {
            onBlockBorder = true;
        }
    }

    bool isHorizontalFace = abs(normal_tcs[0].y) > 0.99;
```

```
30
31      if (onBlockBorder || isHorizontalFace) {
32          height = 0.0;
33      }
34
35      return pos + normal_tcs[0] * height * SCALE;
36  }
37
38  void main() {
39      vec3 p0 = gl_in[0].gl_Position.xyz;
40      vec3 p1 = gl_in[1].gl_Position.xyz;
41      vec3 p2 = gl_in[2].gl_Position.xyz;
42      vec3 p3 = gl_in[3].gl_Position.xyz;
43
44      float u = gl_TessCoord.x;
45      float v = gl_TessCoord.y;
46      vec2 uv = vec2(u, v);
47      vec3 pos = getDisplacedPos(uv, p0, p1, p2, p3);
48
49      vec2 delta = vec2(1.0 / 64.0);
50      vec3 posL = getDisplacedPos(uv - vec2(delta.x, 0.0), p0, p1, p2, p3);
51      vec3 posR = getDisplacedPos(uv + vec2(delta.x, 0.0), p0, p1, p2, p3);
52      vec3 posD = getDisplacedPos(uv - vec2(0.0, delta.y), p0, p1, p2, p3);
53      vec3 posU = getDisplacedPos(uv + vec2(0.0, delta.y), p0, p1, p2, p3);
54      vec3 dU = posU - posD;
55      vec3 dR = posR - posL;
56      vec3 normal = normalize(cross(dU, dR));
57
58      if (dot(normal, normal_tcs[0]) < 0.0) {
59          normal = -normal;
60      }
61
62      bool isHorizontalFace = abs(normal_tcs[0].y) > 0.99;
63      worldPos = model * vec4(pos, 1.0);
64      normalTES = normalize(transpose(inverse(mat3(model))) * normal);
65
66      if (isHorizontalFace) {
67          tesUV = vec2(-1.0, -1.0);
68      }
69      else {
70          tesUV = remapUV(uv, normal_tcs[0]);
71      }
72  }
```

Listing 5.5: Esempio di Tessellation Evaluation Shader utilizzato per determinare il displacement dei vertici dei lati del tetto.

5.10.7 Lampioni (ambientazione 2)

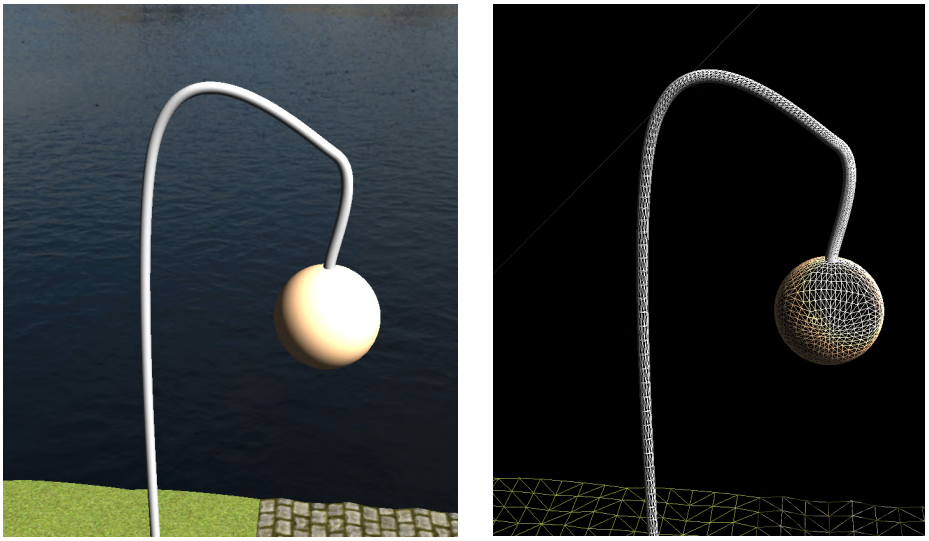

Figura 5.9: Esempio di lampione generato tramite gli shader in modalità `FILL` (sinistra) e in modalità `LINE` (destra).

Un ultimo caso di gestione particolare del sistema di LOD adattivo tramite gli shader è dato dalla pipeline relativa ai lampioni della seconda ambientazione. Qui la geometria di partenza è molto leggera rispetto alle altre pipeline e deve essere amplificata per ottenere un aspetto realistico. La pipeline descritta di seguito riguarda i pali dei lampioni, mentre quella utilizzata per le luci sferiche rimane identica a quella delle stelle della prima ambientazione.

Il Vertex Shader viene ancora una volta utilizzato in modalità *pass-through*, semplicemente per trasportare i vertici della geometria allo stadio successivo.

Il Tessellation Control Shader definisce i livelli di tessellazione in maniera leggermente differente rispetto alle altre pipeline, poichè le patch sono rappresentate da isolines, costituite da segmenti congiunti per ciascun lampione. Per ogni patch, lo shader calcola la distanza dei vertici rispetto alla telecamera (o al personaggio) e ne fa la media, ottenendo un valore rappresentativo della distanza complessiva della patch. Questo valore viene quindi normalizzato e trasformato nel secondo livello di tessellazione tramite una funzione di interpolazione lineare. Il primo livello esterno viene mantenuto a 1, poiché nelle isolines influisce sul numero di linee parallele, mentre il secondo livello determina quanti vertici vengono generati lungo ciascuna curva, permettendo di ottenere pali più definiti quando il personaggio si avvicina. La distanza viene calcolata solo sul piano orizzontale, ignorando la componente verticale, così da reagire anche quando il personaggio si muove vicino alla base del lampione senza dover salire.

```
1 void main() {
2     gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].
    gl_Position;
```

```
3
4       if (gl_InvocationID == 0) {
5           vec3 basePosition = useCharacterToTess ? characterPosition :
        cameraPosition;
6
7           float distSum = 0.0;
8           for (int i = 0; i < 4; ++i) {
9               distSum += length(basePosition - gl_in[i].gl_Position.xyz);
10          }
11
12          float avgDist = distSum / 4.0;
13          float tessLevel = mix(
14              float(maxTessLevel),
15              float(minTessLevel),
16              clamp((avgDist - minDist) / (maxDist - minDist),
17              0.0,
18              1.0)
19          );
20
21          gl_TessLevelOuter[0] = 1.0;
22          gl_TessLevelOuter[1] = tessLevel;
23      }
24  }
```

Listing 5.6: Tessellation Control Shader utilizzato per la tessellazione del palo del lampione.

Nel Tessellation Evaluation Shader, i vertici generati vengono posizionati lungo le curve dei segmenti utilizzando l'interpolazione Catmull-Rom, che permette di ottenere curve lisce e continue tra i punti originali della geometria, distribuendo uniformemente i vertici lungo il percorso, mantenendo la forma del palo.

```
1   vec3 catmullRom(vec3 p0, vec3 p1, vec3 p2, vec3 p3, float t) {
2       return 0.5 * (
3           (2.0 * p1) +
4           (-p0 + p2) * t +
5           (2.0 * p0 - 5.0 * p1 + 4.0 * p2 - p3) * t * t +
6           (-p0 + 3.0 * p1 - 3.0 * p2 + p3) * t * t * t
7       );
8   }
9
10  void main() {
11      float t = gl_TessCoord.x;
12      float curveIndex = gl_TessCoord.y;
13
14      vec3 p0 = gl_in[0].gl_Position.xyz;
15      vec3 p1 = gl_in[1].gl_Position.xyz;
16      vec3 p2 = gl_in[2].gl_Position.xyz;
17      vec3 p3 = gl_in[3].gl_Position.xyz;
18
```

```
    vec3 pos = catmullRom(p0, p1, p2, p3, t);
    pos.y += spacing * curveIndex;
    gl_Position = vec4(pos, 1.0);
}
```

Listing 5.7: Tessellation Evaluation Shader utilizzato per la creazione di curve interpolanti per il palo del lampione.

Il Geometry Shader riceve le linee generate dal TES e costruisce attorno a ciascun segmento un tubo cilindrico, generando sezioni circolari con più vertici per conferire tridimensionalità al palo. Per ciascun vertice della sezione vengono calcolati i due vettori ortogonali al segmento, e la posizione viene trasformata nello spazio della telecamera tramite le matrici di trasformazione fondamentali. Viene inoltre applicato un leggero allungamento ai bordi dei segmenti per evitare interruzioni visibili tra le sezioni, creando l'illusione di un tubo continuo lungo tutto il palo.

```
vec3 orthogonal(vec3 v) {
    if (abs(v.x) > abs(v.z))
        return normalize(vec3(-v.y, v.x, 0.0));
    else
        return normalize(vec3(0.0, -v.z, v.y));
}

void main() {
    vec3 p0 = gl_in[0].gl_Position.xyz;
    vec3 p1 = gl_in[1].gl_Position.xyz;
    vec3 tangent = normalize(p1 - p0);

    vec3 extendedP0 = p0 - tangent * extendLength;
    vec3 extendedP1 = p1 + tangent * extendLength;

    vec3 normal = orthogonal(tangent);
    vec3 binormal = normalize(cross(tangent, normal));

    for (int i = 0; i <= circleSegments; i++) {
        float angle = float(i) / float(circleSegments) * 2.0 *
    3.14159265359;
        float cosA = cos(angle);
        float sinA = sin(angle);

        vec3 offset = normal * cosA * radius + binormal * sinA * radius;
        vec3 pos0 = extendedP0 + offset;
        vec3 pos1 = extendedP1 + offset;

        gsFragPos = pos0;
        gsNormal = normalize(offset);
        gl_Position = proj * view * model * vec4(pos0, 1.0);
        EmitVertex();

```

```
33          gsFragPos = pos1;
34          gsNormal = normalize(offset);
35          gl_Position = proj * view * model * vec4(pos1, 1.0);
36          EmitVertex();
37      }
38
39      EndPrimitive();
40  }
```

Listing 5.8: Geometry Shader utilizzato per la creazione dei tubi cilindrici attorno al palo del lampione.

Infine, il Fragment Shader si occupa di replicare la gestione dell'illuminazione tramite il modello di Phong, in maniera analoga alle altre pipeline, sfruttando le informazioni di posizione e normale passate dal Geometry Shader.

Capitolo 6

Risultati e conclusioni

In questo capitolo vengono presentate le analisi conclusive sui risultati raggiunti dal progetto. Si parte dall'osservazione dei risultati visivi ottenuti, per poi esaminare e confrontare le prestazioni delle diverse varianti sviluppate. Infine, vengono discusse possibili direzioni di sviluppo future, evidenziando gli aspetti principali legati al miglioramento dell'applicazione.

6.1 Risultati visivi

Dopo aver già proposto alcune immagini dei risultati ottenuti e dopo aver descritto le tecniche utilizzate per la realizzazione del sistema di LOD adattivo nelle due diverse ambientazioni, è ora utile fornire una dimostrazione visiva degli effetti grafici globali raggiunti.

Per ogni esempio di seguito proposto vengono prima mostrate le immagini ottenute attraverso la suddivisione con LOD dinamico, con un livello di dettaglio minimo iniziale, in cui la geometria mantiene una forma molto vicina a quella di base generata lato applicativo, e successivamente con dettaglio massimo, così da evidenziare le differenze geometriche e valutare l'impatto visivo degli algoritmi descritti nella sezione dedicata allo sviluppo del progetto.

6.1.1 Ambientazione 1

Montagne e vegetazione

Nella prima ambientazione, l'effetto più evidente della suddivisione e dell'amplificazione geometrica si riscontra sulle montagne che caratterizzano il terreno: esse risultano più definite man mano che, avvicinandosi alla telecamera o al personaggio, aumenta il loro livello di suddivisione.

Dalla figura 6.1 emerge chiaramente come la suddivisione massima della geometria renda le cime più frastagliate e realistiche, grazie ad una maggiore precisione nella rappresentazione dei dislivelli, dovuta all'aumento del dettaglio geometrico.

La figura 6.2 mostra invece l'impatto diretto della tessellazione e dell'amplificazione geometrica sulla struttura di base. Si nota chiaramente come la geometria maggiormente suddivisa risulti più fitta e dettagliata, permettendo una resa più convincente della superficie montuosa. In questo caso, il LOD adattivo porta anche a una distribuzione più ricca della vegetazione nelle aree vicine alla telecamera, mentre le zone più lontane mantengono una geometria semplificata e priva di vegetazione aggiuntiva.

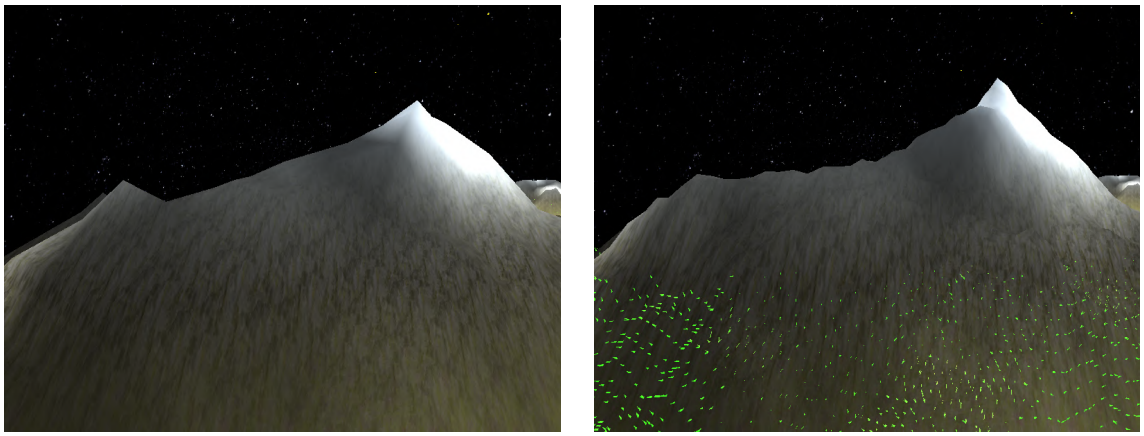

Figura 6.1: Esempio di montagna definita tramite LOD adattivo, in modalità `FILL` (a sinistra la versione base, a destra quella amplificata).

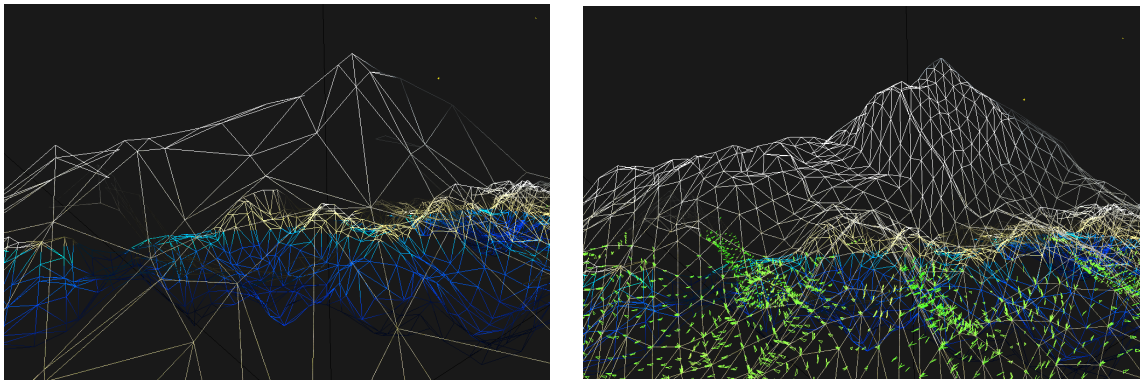

Figura 6.2: Esempio di montagna definita tramite LOD adattivo, in modalità `LINE` (a sinistra la versione base, a destra quella amplificata).

Stelle

Un ulteriore effetto significativo, della prima ambientazione, riguarda la geometria realizzata per rappresentare le stelle sferiche.

In figura 6.3 si osserva come la geometria suddisa tramite un livello di dettaglio minimo, inizialmente più vicina a un solido romboidale, venga progressivamente raffinata fino ad assumere una forma sferica molto più realistica. La figura 6.4 mette invece in evidenza la complessità geometrica introdotta dal processo di tessellazione, grazie al quale la densità delle suddivisioni aumenta in modo significativo, permettendo

di ottenere un numero elevato di vertici che definiscono con precisione la superficie curva della stella.

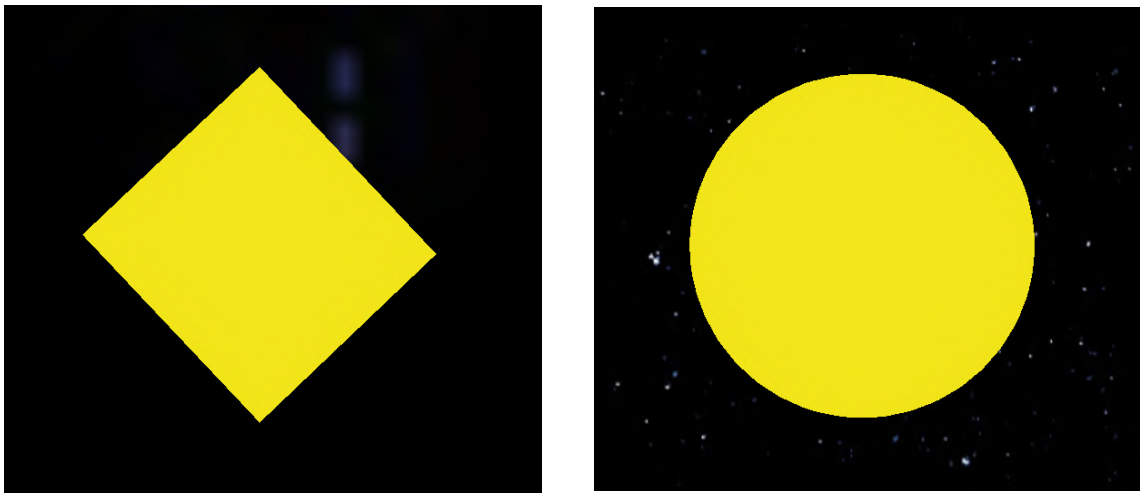

Figura 6.3: Esempio di stella definita tramite LOD adattivo, in modalità `FILL` (a sinistra la versione base, a destra quella amplificata).

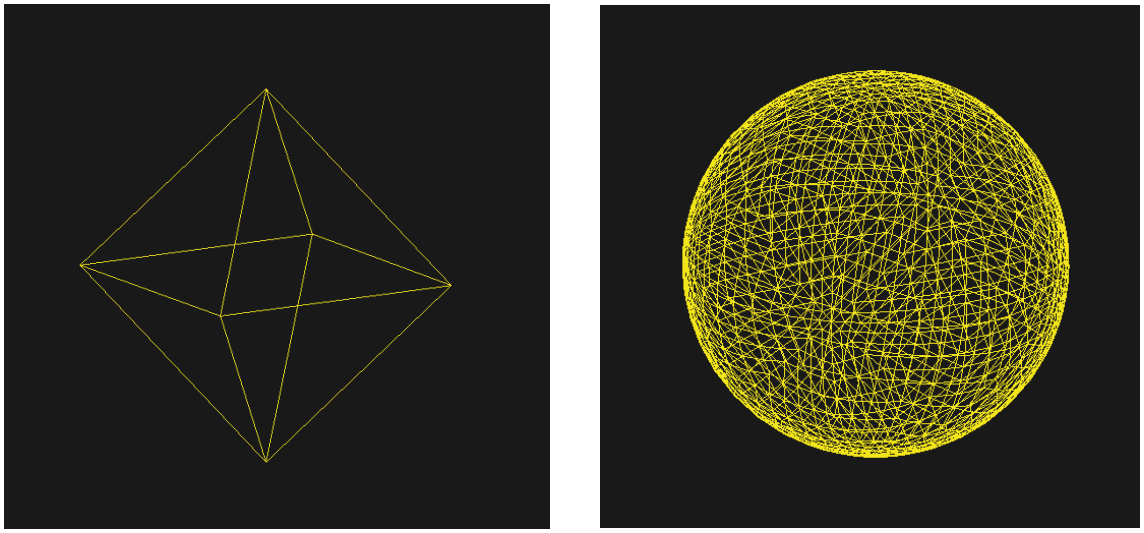

Figura 6.4: Esempio di stella definita tramite LOD adattivo, in modalità `LINE` (a sinistra la versione base, a destra quella amplificata).

6.1.2 Ambientazione 2

Lampioni

Nella seconda ambientazione, caratterizzata dalla presenza di un numero maggiore di oggetti definiti e amplificati dinamicamente, si osservano diversi casi ben visibili di come il LOD adattivo abbia contribuito a rendere la scena maggiormente realistica nelle zone vicine alla telecamera o al personaggio.

Un primo esempio, molto efficace, è dato dalla geometria relativa al lampione. In figura 6.5 si può notare come la geometria, con livello di dettaglio minimo, risulti

spezzata, semplice e innaturale, mentre invece diventi lineare, realistica e continua con livello di dettaglio massimo. Come per gli altri casi, la figura 6.6 mostra invece il grado di suddivisione geometrica che risulta dall'aumento del livello di dettaglio quando l'oggetto risulta vicino alla telecamera.

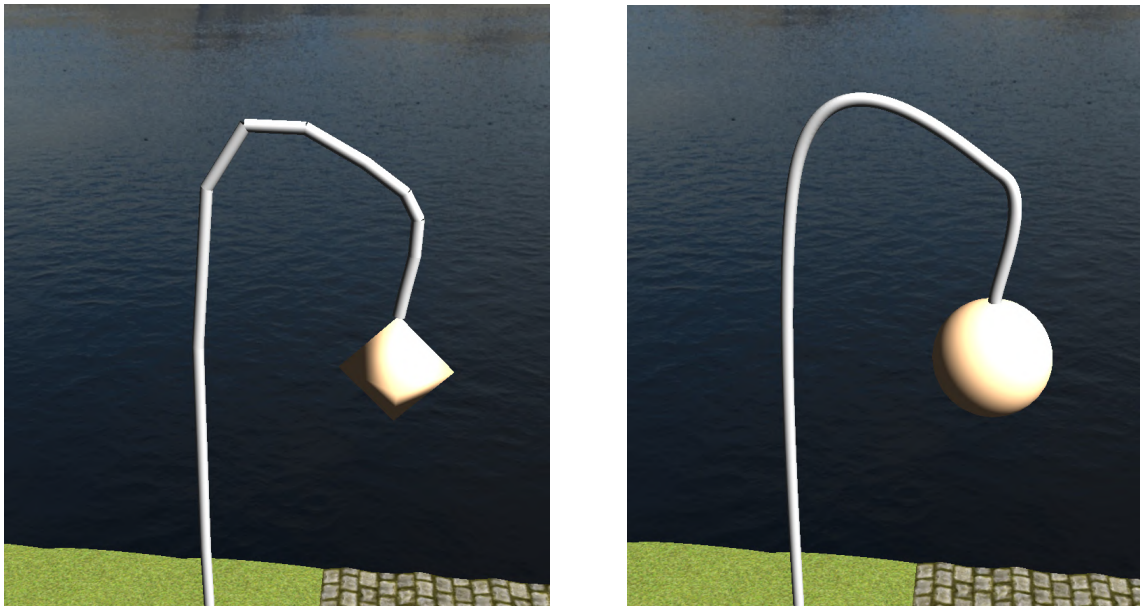

Figura 6.5: Esempio di lampione definito tramite LOD adattivo, in modalità `FILL` (a sinistra la versione base, a destra quella amplificata).

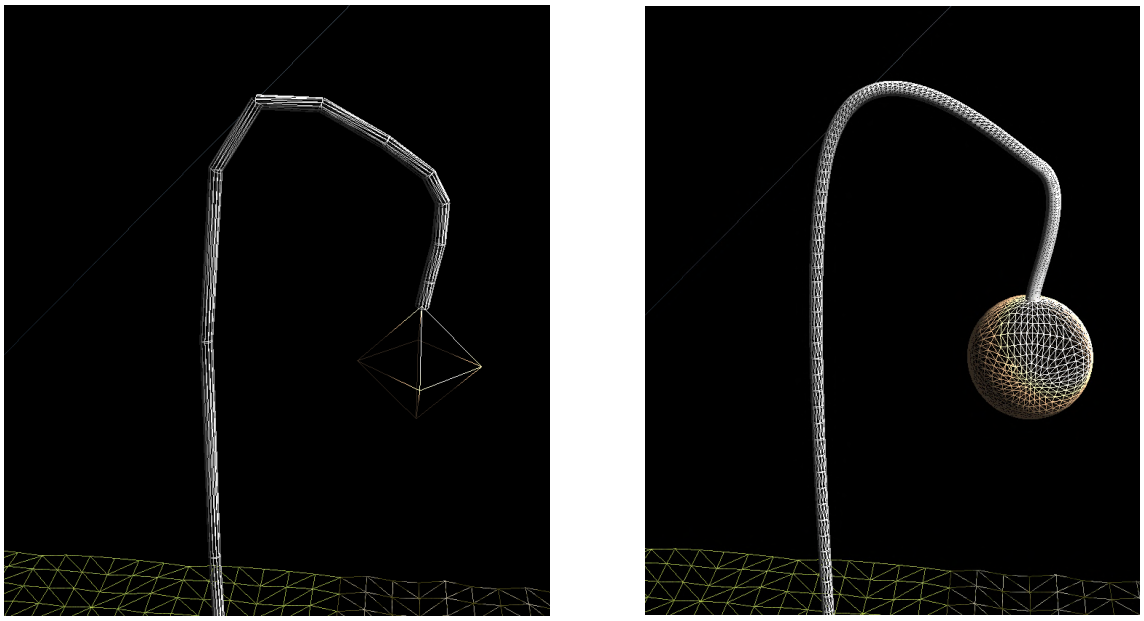

Figura 6.6: Esempio di lampione definito tramite LOD adattivo, in modalità `LINE` (a sinistra la versione base, a destra quella amplificata).

Edifici e tetti

Un secondo caso in cui risulta particolarmente evidente l'apporto del LOD adattivo è rappresentato dagli edifici in cui, sia i tetti sia le facciate laterali beneficiano della suddivisione dinamica della geometria. Nelle figure 6.7 e 6.8 viene confrontata la versione a dettaglio minimo, caratterizzata da superfici semplificate e spigoli poco realistici, e la versione a dettaglio massimo, in cui la tessellazione consente di ottenere geometrie più complesse e convincenti dal punto di vista visivo.

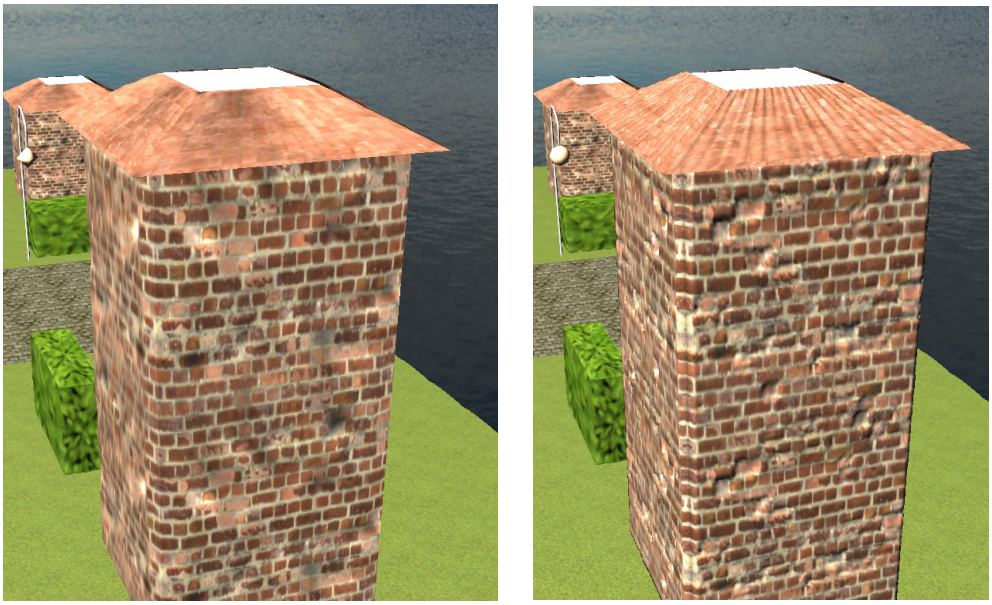

Figura 6.7: Esempio di edificio e di tetto definiti tramite LOD adattivo, in modalità `FILL` (a sinistra la versione base, a destra quella amplificata).

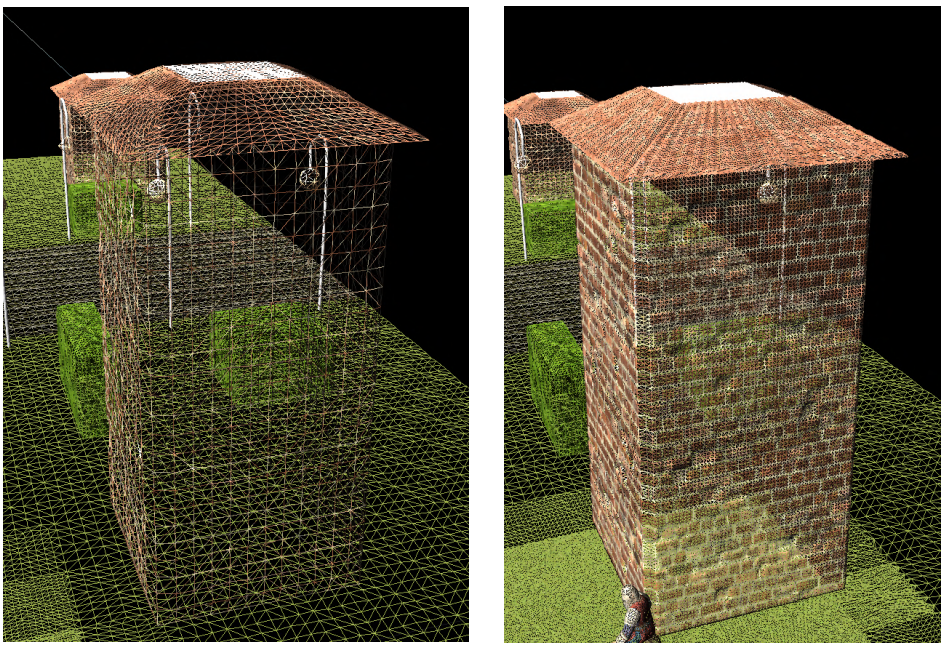

Figura 6.8: Esempio di edificio e di tetto definiti tramite LOD adattivo, in modalità `LINE` (a sinistra la versione base, a destra quella amplificata).

Siepi

Un ulteriore esempio dell'efficacia del LOD adattivo, riguarda le siepi, in cui la suddivisione dinamica della geometria contribuisce a rendere le superfici più realistiche. Nelle figure 6.9 e 6.10 si osserva come, nella versione a dettaglio minimo, la forma della siepe risulti squadrata e poco naturale. Al contrario, nella versione a dettaglio massimo, l'incremento del numero di vertici consente di ottenere superfici più definite e conferisce all'elemento un aspetto complessivo più realistico e coerente con l'ambiente circostante.

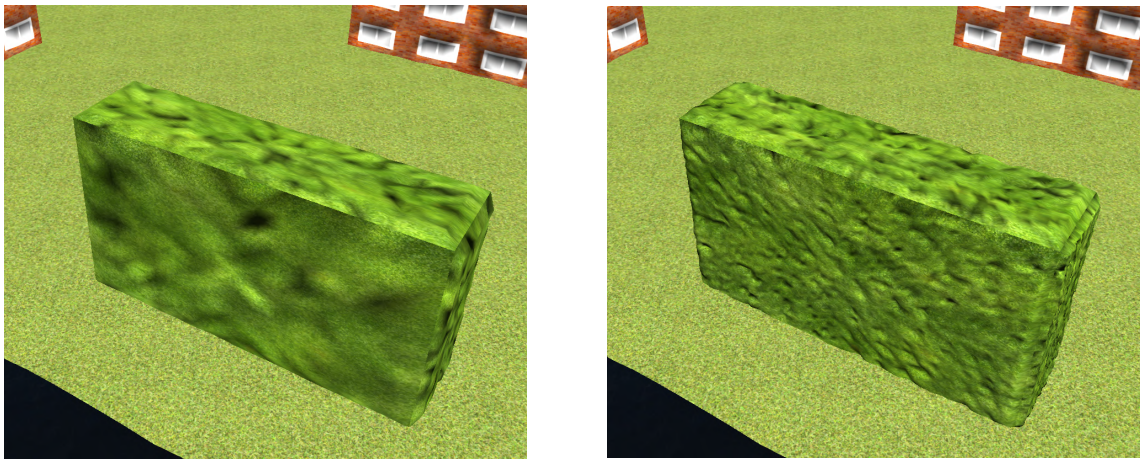

Figura 6.9: Esempio di siepe definita tramite LOD adattivo, in modalità `FILL` (a sinistra la versione base, a destra quella amplificata).

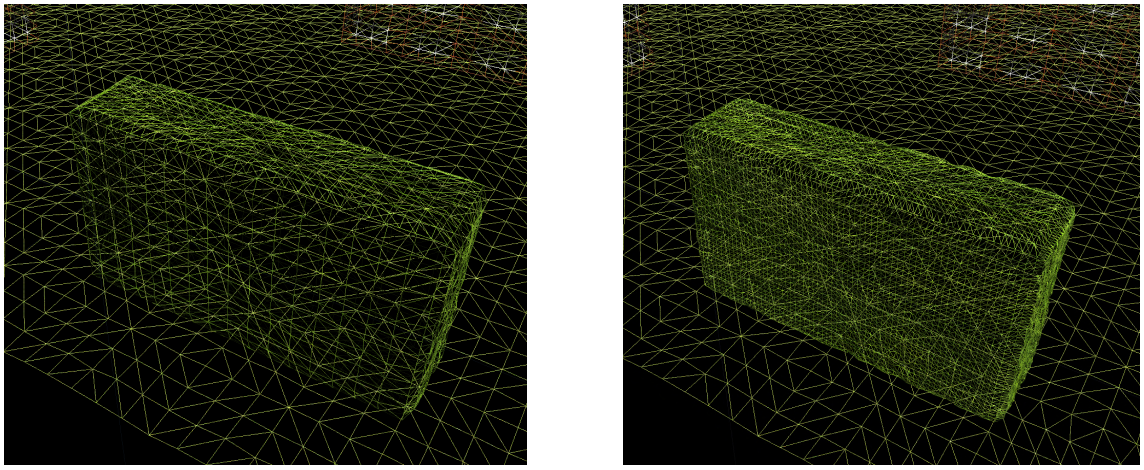

Figura 6.10: Esempio di siepe definita tramite LOD adattivo, in modalità `LINE` (a sinistra la versione base, a destra quella amplificata).

Strada

L'ultimo esempio riguarda invece la strada, su cui l'effetto del LOD adattivo è evidente nelle figure 6.11 e 6.12. Nella versione a dettaglio minimo, la strada appare infatti piatta a causa della geometria limitata a disposizione. Grazie ad un livello di suddivisione maggiore del LOD dinamico, implementato tramite tessellation e

geometry shader, è possibile invece suddividere e amplificare la geometria, generando un numero maggiore di vertici e permettendo una migliore rappresentazione delle irregolarità e dei piccoli dislivelli dovuti alle rocce che compongono la superficie, conferendo alla strada un aspetto più tridimensionale e realistico.

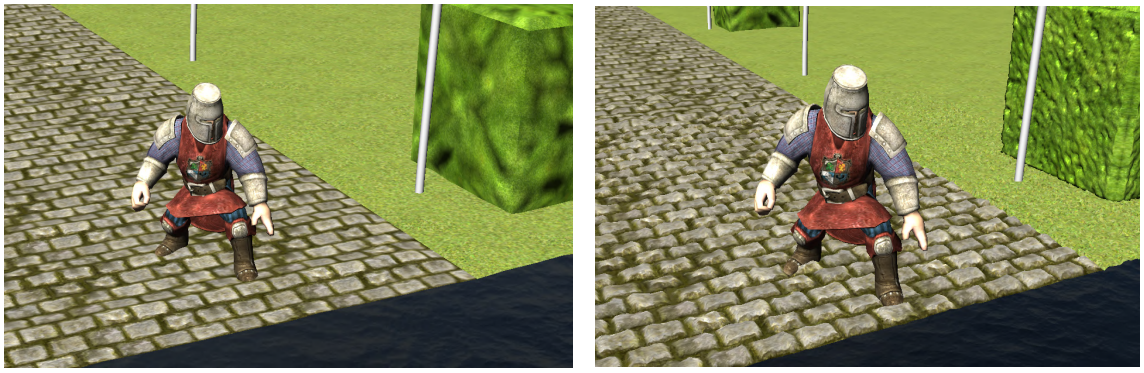

Figura 6.11: Esempio di una parte di strada definita tramite LOD adattivo, in modalità `FILL` (a sinistra la versione base, a destra quella amplificata).

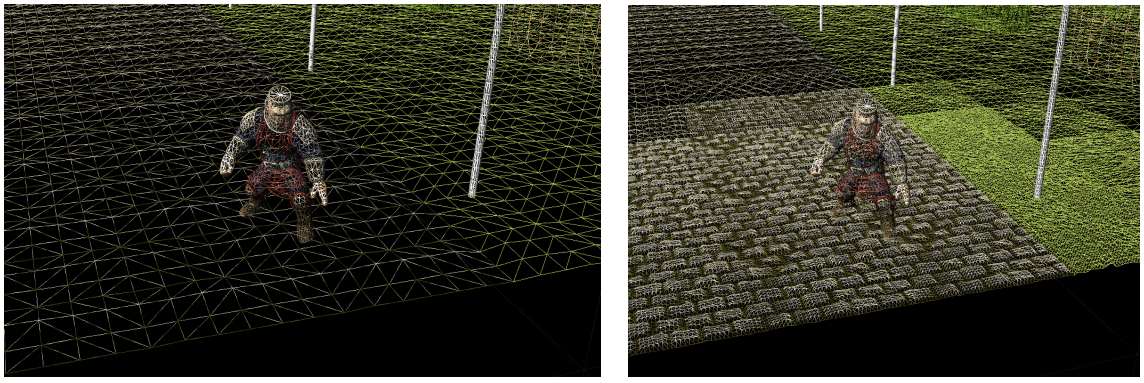

Figura 6.12: Esempio di una parte di strada definita tramite LOD adattivo, in modalità `LINE` (a sinistra la versione base, a destra quella amplificata).

6.2 Analisi delle prestazioni

6.2.1 Configurazioni di test

Per valutare l'efficacia del lavoro svolto e la coerenza con gli obiettivi prefissati, sono state realizzate diverse varianti dell'applicazione, così da sperimentare differenti modalità di esecuzione e gestione del dettaglio geometrico.

In particolare, per entrambe le ambientazioni sviluppate (paesaggio montuoso e urbano), sono state implementate quattro versioni distinte:

- **Versione con LOD dinamico calcolato dalla GPU**: rappresenta il cuore del progetto di tesi. In questa configurazione il livello di dettaglio viene gestito direttamente sulla GPU, tramite *Tessellation Shaders* e *Geometry Shader*,

dove la suddivisione e l'amplificazione della geometria variano dinamicamente in base alla distanza dalla telecamera.

- **Versione con LOD dinamico calcolato dalla CPU**: qui la gestione del dettaglio viene realizzata dalla componente applicativa. In questo caso non si fa uso né di tessellation né di geometry shading, ma è la CPU che calcola dinamicamente il livello di dettaglio delle geometrie da visualizzare. All'avvio del programma (fuori dal ciclo principale) viene definita la geometria di base suddivisa in patch, in modo analogo a quanto avviene in una pipeline con tessellation. Durante ogni iterazione del ciclo principale, per ciascuna patch viene calcolata la distanza dal centro rispetto al punto di riferimento (telecamera o personaggio) e, tramite un'operazione di interpolazione lineare inversa, viene stabilito il livello di suddivisione da applicare (uguale per ciascun lato della patch). In seguito, vengono generati i nuovi vertici di ogni patch in base al livello di suddivisione calcolato, utilizzando una suddivisione bilineare quando si lavora su superfici planari (ad esempio per il terreno o per i muri), o in altri casi, rigenerando l'intera geometria con un numero di vertici variabile in base al LOD, creando mesh più o meno dettagliate (ad esempio per le sfere dei lampioni). Per ciascun vertice generato, vengono inoltre calcolati gli attributi necessari al rendering, come i valori di displacement, le normali e le coordinate di texture. Infine, i buffer della GPU vengono aggiornati con i vertici appena calcolati, in modo che vengano così processati dal flusso tradizionale della pipeline grafica, passando attraverso Vertex Shader e Fragment Shader.

- **Versione con calcoli eseguiti dalla GPU senza LOD dinamico e con geometria al massimo dettaglio**: gli shader opzionali vengono utilizzati per definire sempre il massimo livello di dettaglio per ogni geometria originale. La geometria di partenza viene comunque raffinata dalla GPU, ma senza alcuna logica di adattamento basata sulla distanza dalla telecamera.

- **Versione con calcoli eseguiti dalla CPU senza LOD dinamico e con geometria al massimo dettaglio**: le mesh vengono caricate al massimo livello di dettaglio fin dall'inizio. La pipeline grafica in questo caso è ridotta ai soli *Vertex Shader* e *Fragment Shader*, impiegati per la semplice visualizzazione delle geometrie.

In questo modo sono state ottenute quattro implementazioni per ciascuna ambientazione, utili per confrontare in maniera diretta i diversi approcci e valutarne le prestazioni.

Per l'analisi delle prestazioni è stato utilizzato come indicatore principale il numero di *FPS* (*frames per second*), ovvero il numero di schermate renderizzate e visualizzate nell'arco di un secondo. La misurazione è stata effettuata direttamente all'interno dell'applicazione, considerando le iterazioni del *main loop* (cioè il ciclo principale del programma che aggiorna costantemente lo stato dell'applicazione e ridisegna la scena) e il numero di frame effettivamente presentati.

I quattro casi di test sono stati eseguiti su due distinte configurazioni hardware: una GPU integrata, montata su un computer portatile, e una GPU dedicata, presente

su un computer fisso.

Nel **primo caso**, l'applicazione è stata eseguita su un portatile dotato di GPU integrata *Intel(R) Iris(R) Plus Graphics*, con driver in versione 27.20.100.8681 (rilasciati il 05/09/2020). Questa GPU supporta le *DirectX* 12 con *Feature Level* 12.1, che definiscono il massimo livello di funzionalità grafiche teoricamente disponibili sull'hardware. Le **DirectX** sono un insieme di librerie sviluppate da Microsoft che consentono ai programmi di interagire con l'hardware grafico in modo standardizzato, garantendo compatibilità e prestazioni migliori. Il **Feature Level** rappresenta invece il sottoinsieme di funzionalità effettivamente implementate dall'hardware, che indica il livello massimo di capacità grafiche che una GPU può mettere a disposizione, pur restando conforme a una determinata versione delle DirectX.

Queste informazioni vengono riportate come riferimento tecnico per comprendere le capacità della scheda grafica. Va precisato però che nel progetto non sono state utilizzate direttamente le DirectX, ma sono state sfruttate le funzionalità messe a disposizione da OpenGL e dalle altre librerie impiegate, che operano comunque entro i limiti e le capacità dell'hardware stesso.

La GPU Iris Plus non possiede memoria video dedicata, ma condivide la RAM con il processore centrale. Questo comporta una minore larghezza di banda e una ridotta capacità di calcolo parallelo rispetto a una GPU dedicata, fattori che incidono soprattutto in scenari che richiedono tecniche come la tessellation o il geometry shading. Nonostante questo, le prestazioni ottenute si sono dimostrate sufficienti per lo sviluppo e la sperimentazione di pipeline basate su OpenGL 4.6.

La GPU è integrata in un processore *Intel Core i7-1065G7* da 1.30 GHz, appartenente alla famiglia *Ice Lake* di decima generazione. In questo modello il controller grafico è incluso direttamente nel *die* della CPU, ovvero nello stesso chip fisico che ospita tutti i componenti principali del processore. Questa configurazione rappresenta un tipico esempio di sistema *consumer*, cioè di uso comune, non specializzato per il calcolo grafico avanzato. Testare l'applicazione su questa piattaforma permette quindi di verificare quanto il programma sia efficiente e portabile anche su macchine comuni, senza GPU dedicata.

Nel **secondo caso**, l'applicazione è stata eseguita su un computer fisso dotato di GPU dedicata *NVIDIA GeForce GTX 750 Ti*, con driver in versione 32.0.15.6094 (rilasciati il 14/08/2024). Questa GPU supporta le *DirectX* 12 con *Feature Level* 11.0. A differenza della GPU integrata del primo caso, la GTX 750 Ti dispone di 2 GB di memoria video dedicata con banda di 86.4 GB/s, caratteristiche che consentono una gestione più efficiente delle texture e dei buffer grafici. Tuttavia, il livello di funzionalità 11.0 la limita rispetto a schede più moderne, impedendo l'accesso ad alcune tecniche avanzate introdotte con le versioni successive.

Il sistema utilizza poi un processore *Intel(R) Core(TM) i7-6700K* da 4.00 GHz, appartenente alla famiglia *Skylake* di sesta generazione, con 4 core fisici e 8 processori logici. Rispetto alla CPU del portatile, questa soluzione garantisce prestazioni superiori.

Tale configurazione rappresenta quindi un esempio di sistema *desktop enthusiast*, dove la presenza di una GPU dedicata e di una CPU performante consente di testare l'applicazione in condizioni più favorevoli rispetto a un sistema consumer portatile.

6.2.2 Risultati ottenuti e confronto delle prestazioni

I risultati ottenuti dall'esecuzione dei vari casi di test sulle due configurazioni hardware appena descritte sono riportati nelle tabelle seguenti. In entrambe, i dati sono stati organizzati come segue. Ogni riga corrisponde a uno dei quattro casi di test previsti:

- Dettaglio geometrico dinamico realizzato tramite LOD adattivo sulla GPU.
- Dettaglio geometrico dinamico realizzato tramite LOD adattivo sulla CPU.
- Dettaglio massimo, indipendente dalla posizione della telecamera, elaborato ad ogni iterazione dalla GPU.
- Dettaglio statico della geometria, caricata direttamente al livello massimo, elaborato una sola volta (al di fuori del ciclo principale) dalla CPU.

Sulle colonne, i dati sono differenziati secondo due modalità, ovvero `LINE MODE` (che visualizza le sole linee delle geometrie) e `FILL` (che consente di valutare l'effetto visivo completo, con le geometrie riempite). Ciascuna di queste due modalità è a sua volta suddivisa in due colonne, per confrontare direttamente le prestazioni ottenute su hardware con GPU integrata e su hardware con GPU dedicata.

Sono state realizzate due tabelle distinte, una per ciascuna delle ambientazioni sviluppate, così da analizzare separatamente i risultati e rendere più chiara l'osservazione delle differenze prestazionali all'interno dello stesso contesto. In particolare, la figura 6.13 riporta le prestazioni relative alla prima ambientazione (il paesaggio montuoso), mentre la 6.14 si riferisce alla seconda (il paesaggio urbano).

Questa organizzazione consente di mettere in evidenza, in modo lineare e immediato, le variazioni di performance tra i diversi approcci implementativi e tra le due configurazioni hardware, facilitando sia il confronto diretto sia la lettura complessiva dei dati.

	LINE MODE		FILL MODE	
	GPU integrata	GPU dedicata	GPU integrata	GPU dedicata
Dettaglio dinamico GPU	90-130	400-410	90-150	410-420
Dettaglio dinamico CPU	1-2	3-6	1-2	5-6
Dettaglio statico GPU	2-3	50	2-3	55
Dettaglio statico CPU	10-15	35	45-50	130

Figura 6.13: Confronto delle prestazioni nelle quattro versioni di test realizzate per la prima ambientazione (paesaggio montuoso), con unità di misura espressa in *FPS* (*frames per second*).

	LINE MODE		FILL MODE	
	GPU integrata	GPU dedicata	GPU integrata	GPU dedicata
Dettaglio dinamico GPU	100-130	320-480	110-180	400-450
Dettaglio dinamico CPU	6-8	6-12	6-8	6-12
Dettaglio statico GPU	12	30	28	60
Dettaglio statico CPU	8	20	55-80	170

Figura 6.14: Confronto delle prestazioni nelle quattro versioni di test realizzate per la seconda ambientazione (paesaggio urbano), con unità di misura espressa in *FPS* (*frames per second*).

Dai risultati relativi alla **prima ambientazione**, i test mostrano chiaramente come l'approccio di dettaglio dinamico implementato sulla GPU garantisca prestazioni nettamente superiori rispetto alle altre soluzioni.

Per quanto riguarda la tecnica del dettaglio dinamico delle geometrie: attraverso la GPU dedicata si raggiungono valori stabili intorno ai 400 FPS in entrambe le modalità (`LINE` e `FILL`), mentre con la GPU integrata le prestazioni oscillano fra 90 e 150 FPS, mantenendo comunque una fluidità pienamente utilizzabile. Al contrario, lo stesso approccio dinamico, ma eseguito su CPU, produce valori quasi nulli (con un massimo di 6 FPS tra i risultati ottenuti coi due tipi di hardware), poichè i calcoli del LOD sono eseguiti direttamente nel ciclo principale dell'applicazione, senza poter sfruttare il parallelismo fornito dalla GPU.

In questo scenario, la CPU è costretta a gestire in modo sequenziale un gran numero di operazioni di calcolo per la generazione e l'adattamento dei vertici, simulando l'intero processo di suddivisione e amplificazione geometrica attraverso calcoli software. Questo comporta un dispendio significativo di tempo e di risorse, oltre che un carico computazionale troppo oneroso per poter essere gestito in tempo reale. La GPU, invece, grazie alla sua architettura, può distribuire questi calcoli su più esecuzioni in parallelo e integra nella propria pipeline dei componenti ottimizzati, sia programmabili (come Tessellation Shaders e Geometry Shader) sia fissi (come il Tessellator). Quest'ultimo, in particolare, è in grado di generare nuovi vertici direttamente in hardware, con latenze ridotte, permettendo un adattamento geometrico dinamico che risulta estremamente efficiente. In questo modo, la GPU riesce a garantire prestazioni elevate pur mantenendo un livello qualitativo molto alto.

Nel caso di assegnazione del massimo livello di dettaglio a tutta la geometria della scena, i risultati mostrano limiti significativi. Attraverso la GPU si ottengono prestazioni accettabili con la GPU dedicata (50 FPS in `LINE MODE` e 55 FPS in `FILL MODE`), ma estremamente basse con la GPU integrata ($2 - 3$ FPS). Ciò è dovuto al fatto che l'intera complessità geometrica viene caricata senza alcun adattamento, portando a un numero di vertici molto elevato che la GPU è costretta a processare integralmente, facendo così risaltare le diverse prestazioni dei due tipi di GPU hardware. Nel caso della versione statica lato CPU invece, i dati risultano generalmente migliori, ma ciò dipende dal diverso flusso di elaborazione. Qui, infatti, i vertici

vengono passati già completamente definiti alla pipeline e processati su GPU solo dal Vertex e dal Fragment Shader.

La discreta qualità dei risultati ottenuti nel caso di dettaglio statico elaborato dalla CPU non può però reggere il confronto con il caso di LOD dinamico su GPU, anche considerando il fatto che l'aumento della complessità della scena determinerebbe un repentino decadimento delle prestazioni, a causa del numero sempre crescente di vertici che verrebbero generati e processati integralmente dalla pipeline grafica. Il LOD dinamico su GPU parte invece da un numero molto più contenuto di vertici e genera quelli mancanti direttamente all'interno della pipeline, riducendo il carico iniziale e mantenendo la possibilità di adattare dinamicamente la complessità in funzione della posizione della telecamera. Se le scene aumentassero di complessità (come accadrebbe in un contesto reale, con intere mappe o ambienti completi), il costo di definire fin dall'inizio un numero enorme di vertici diventerebbe insostenibile e proprio in tali scenari emergerebbe in modo ancora più netto il vantaggio dell'approccio dinamico su GPU.

Nella **seconda ambientazione** (paesaggio urbano), i valori confermano lo stesso andamento generale, ma introducono alcune nuove considerazioni legate alla natura più eterogenea della scena. In questo contesto, il dettaglio dinamico su GPU mantiene prestazioni molto elevate, con valori compresi tra 320 e 480 FPS su GPU dedicata e tra 100 e 180 FPS su GPU integrata. Tali risultati dimostrano la robustezza dell'approccio anche in scenari caratterizzati da una maggiore varietà di elementi e da pipeline di gestione più numerose e articolate, riuscendo comunque a garantire un equilibrio ottimale tra qualità visiva e fluidità del rendering.

L'approccio dinamico su CPU mostra prestazioni leggermente migliori rispetto alla prima ambientazione (tra 6 e 12 FPS), ma rimane comunque del tutto inadeguato per un'applicazione in tempo reale. Questo incremento minimo è da attribuire alla diversa distribuzione dei vertici nella scena urbana, ma non rappresenta in alcun modo un vantaggio concreto.

Il dettaglio statico su GPU risente maggiormente della complessità geometrica: le prestazioni scendono a 30–60 FPS su GPU dedicata e a soli 12–28 FPS su GPU integrata, evidenziando la difficoltà di gestire direttamente scene molto dense senza meccanismi di adattamento. Ancora una volta, questo sottolinea l'efficacia del LOD dinamico lato GPU rispetto alle soluzioni statiche.

Infine, il dettaglio statico lato CPU mostra risultati più variegati. In modalità `FILL` con GPU dedicata si osservano valori molto alti (fino a 170 FPS), ma negli altri casi le prestazioni risultano notevolmente inferiori. Anche qui, il fenomeno è spiegabile con il diverso bilanciamento dei carichi: in questa ambientazione, le geometrie urbane, pur numerose, presentano strutture regolari che talvolta agevolano il passaggio diretto dei vertici già definiti. Tuttavia, l'approccio resta poco flessibile e non paragonabile al metodo dinamico su GPU, che garantisce un adattamento più fine e costante delle risorse.

Un aspetto rilevante emerso dai dati complessivi riguarda anche la differenza tra GPU integrata e dedicata. Quest'ultima garantisce sempre un incremento presta-

zionale significativo (anche superiore a cinque volte in alcuni casi), dimostrando quanto la parallelizzazione e la maggiore potenza della GPU dedicata incidano nella resa del sistema. Ciò è particolarmente evidente nei casi di calcolo intensivo, come l'adattamento dinamico della geometria, dove le potenzialità hardware della GPU vengono sfruttate appieno.

6.3 Conclusioni e sviluppi futuri

L'analisi complessiva conferma come l'utilizzo del LOD dinamico, realizzato tramite shader di tessellazione e di geometria, rappresenti una soluzione efficace per coniugare qualità visiva e prestazioni. Questa tecnica permette di sfruttare appieno le potenzialità della GPU e di adattare in tempo reale il dettaglio delle geometrie, garantendo risultati che si rivelano superiori rispetto ad approcci più statici o basati sulla sola CPU.

Tuttavia, l'efficacia di questo metodo non dipende soltanto dal principio alla base del metodo stesso, ma soprattutto dal modo in cui questo viene implementato. Ogni operazione superflua all'interno degli shader, seppur apparentemente irrilevante, viene inevitabilmente ripetuta per milioni di vertici e tende quindi ad amplificarsi fino a compromettere le prestazioni complessive. È quindi fondamentale progettare algoritmi snelli ed efficienti, capaci di evitare sprechi computazionali e di sfruttare al meglio le risorse messe a disposizione dalla pipeline grafica.

In questo senso, il LOD dinamico su GPU rappresenta una scelta solida e versatile, a patto che venga accompagnato da un'attenta progettazione degli shader e da un uso consapevole degli strumenti offerti dall'hardware grafico.

Un possibile sviluppo futuro riguarda l'ottimizzazione del sistema di LOD adattivo per considerare non solo la distanza dal personaggio, ma anche la sua direzione di vista. In questo modo, la suddivisione e l'aumento del dettaglio geometrico verrebbero applicati solo alle parti effettivamente visibili all'utente, in base all'altezza, all'inclinazione e alla posizione all'interno del cono di vista. Questo approccio permette di concentrare le risorse computazionali solo su ciò che l'utente vede realmente, evitando di generare dettagli superflui per geometrie vicine ma non osservate, e migliorando ulteriormente l'efficienza del sistema, soprattutto in ambienti di grandi dimensioni tipici di videogiochi o simulazioni interattive.

Un secondo ambito di sviluppo riguarda l'adattamento dinamico del LOD in funzione delle prestazioni rilevate in tempo reale dall'applicazione. In pratica, il sistema potrebbe regolare automaticamente la densità dei vertici e i livelli di dettaglio in base alla capacità di rendering corrente, mantenendo costante la fluidità percepita dall'utente. In presenza di zone particolarmente complesse o di cali di prestazioni, il sistema ridurrebbe temporaneamente il dettaglio massimo generato, senza compromettere l'esperienza complessiva, bilanciando così qualità visiva e stabilità delle prestazioni.

Complessivamente, il lavoro svolto dimostra quindi come il LOD dinamico su GPU, implementato tramite shader di tessellazione e geometria, sia una strategia efficace

per ottenere alta qualità visiva senza compromettere le prestazioni, anche in scenari complessi e ricchi di dettagli. I risultati sperimentali confermano la validità dell'approccio e sottolineano l'importanza di un'attenta progettazione degli shader per sfruttare al meglio le potenzialità hardware. Gli sviluppi futuri illustrati offrono inoltre indicazioni concrete su come estendere e perfezionare ulteriormente la tecnica, rendendola ancora più adattativa ed efficiente.

Bibliografia

1. Akenine-Möller, Tomas; Haines, Eric; Hoffman, Naty; Pesce, Angelo; Iwanicki, Michal; Hillaire, Sébastien. *Real-Time Rendering, Fourth Edition*. CRC Press, 2018.

2. Bailey, Mike; Cunningham, Steve. *Graphics Shaders: Theory and Practice, Second Edition*. CRC Press, 2012.

3. Gonzalez Vivo, Patricio; Lowe, Jen. *Fractal Brownian Motion*. In *The Book of Shaders*, 2015. `https://thebookofshaders.com/13/`

4. Gordan, Victor. *Modern OpenGL Tutorial – Tessellation Shaders*. YouTube, 2022. `https://www.youtube.com/watch?v=21gfE-zUym8`

5. Gordan, Victor. *Procedural Generation Tutorial*. YouTube, 2022. `https://www.youtube.com/watch?v=FKLbihqDLsg`

6. Kessenich, John; Sellers, Graham; Shreiner, Dave. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3, Eighth Edition*. Addison-Wesley, 2013.

7. LearnOpenGL. *Geometry Shader Tutorial*. `https://learnopengl.com/Advanced-OpenGL/Geometry-Shader`

8. LearnOpenGL. *Guest Article: Tessellation and Height Maps*. `https://learnopengl.com/Guest-Articles/2021/Tessellation/Height-map`

9. LearnOpenGL. *Guest Article: Tessellation Shaders Tutorial*. `https://learnopengl.com/Guest-Articles/2021/Tessellation/Tessellation`

10. Meiri, Etay. *Skeletal Animation with Assimp – Tutorial 38*. OGLDev, 2011. `https://ogldev.org/www/tutorial38/tutorial38.html`

11. Mount, Dave; Eastman, Roger. *Procedural Generation: 2D Perlin Noise*. University of Maryland, Lecture Notes, 2018. `https://www.cs.umd.edu/class/spring2018/cmsc425/Lects/lect13-2d-perlin.pdf`

12. Open Asset Import Library Documentation. *Assimp Data Structures (non ufficiale)*. `https://documentation.help/assimp/data.html`

13. Perlin, Ken. *An Image Synthesizer*. Proceedings of SIGGRAPH '85, 1985, pp. 287–296. `https://dl.acm.org/doi/pdf/10.1145/325165.325247`

14. The Khronos Group. *Geometry Shader – OpenGL Wiki*. https://www.khronos.org/opengl/wiki/Geometry_Shader

15. The Khronos Group. *OpenGL 4.6 Core Profile Specification*. 2017. https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf

16. The Khronos Group. *Tessellation Control Shader – OpenGL Wiki*. https://www.khronos.org/opengl/wiki/Tessellation_Control_Shader

17. Vince, John. *Mathematics for Computer Graphics, Seventh Edition*. Springer-Verlag London, 2025.

18. Yuksel, Cem. *Interactive Computer Graphics – Geometry Shader Lecture*. University of Utah, School of Computing, YouTube, 2020. https://www.youtube.com/watch?v=5Ruv2H9lkGA

19. Yuksel, Cem. *Interactive Computer Graphics – Tessellation Shaders Lecture*. University of Utah, School of Computing, YouTube, 2020. https://www.youtube.com/watch?v=OqRMNrvu6TE