UNIVERSITÀ DEGLI STUDI DI PADOVA
FACOLTÀ DI INGEGNERIA INFORMATICA

TESI DI LAUREA IN
INGEGNERIA INFORMATICA

Sviluppo di un Motore Grafico 3D real-time per applicazioni videoludiche

Relatore: *prof. Ezio Stagnaro*

Laureando: *Nicola Marchesan*

ANNO ACCADEMICO 2010-2011

ai miei genitori,
a tutti i miei cari.

Indice

Immagine tratta dal motore grafico sviluppato

1. Introduzione

1.1 Motori Grafici 3D

Oggigiorno si sente sempre più spesso parlare di grafica 3D non solo in contesti tecnici, ma anche nella la vita quotidiana, basti pensare alla recente rivoluzione del cinema e TV 3D. La grafica 3D è passata dal ricoprire un ruolo di nicchia ai suoi albori negli anni sessanta e settanza, fino a diventare oggi la componente primaria di una larga e crescente fetta del mercato globale dell'intrattenimento (si pensi al campo dei videogiochi, al settore cinematografico, alla telefonia), nonchè la struttura base di innumerevoli sistemi di supporto alle persone, in primis il settore della simulazione, largamente sfruttata per scopi educativi, e del campo medico, in cui avanzati sensori trasmettono dati ad un sistema che elabora in tempo reale un'immagine corrispondente alla realtà.

Parallelamente all'incessante nascita di nuovi algoritmi, nel corso degli anni si è vista crescere enormemente la capacità di calcolo dei sistemi dedicati all'elaborazione grafica. Ciò, a partire dagli anni ottanta, ha permesso la realizzazione dei primi sistemi grafici tridimensionali in real-time, ovvero i sistemi in cui le immagini vengono elaborate e presentate con una velocità tale da fornire l'illusione del movimento. Tale caratteristica è misurata con un parametro detto *frame rate*, che indica quante immagini vengono prodotte o visualizzate al secondo. Esso viene misurato in hertz [Hz], sebbene nel campo della computer grafica venga definito in frame per secondo (fps). Mentre ad esempio nel campo cinematografico tale valore è costante e standardizzato, e si aggira sui 25 fps, nei calcolatori esso è completamente variabile, funzione non solo della complessità della scena che si intende visualizzare, ma in maggior misura dell'efficienza degli algoritmi utilizzati e della resa grafica desiderata. Poichè è dimostrato che la percezione del movimento si ottiene con un valore minimo di 30 fps, è compito del progettista del sistema grafico garantire tale valore in ogni situazione. Tuttavia a causa del diverso sistema di produzione delle immagini (particolarmente per il fatto che le immagini artificiali sono in genere prodotte indipendentemente l'una dall'altra, senza alcun meccanismo di

impressionamento di una pellicola), le immagini prodotte da un calcolatore risultano più "fredde" con la conseguenza che la soglia minima si sposta dai 30 fps ad almeno 40-50 fps.

Un altro importante aspetto, per il quale però non esiste un parametro di misura, è la resa grafica delle immagini. Nei sistemi di computer grafica 3D non real-time, le immagini vengono elaborate con un dettaglio grafico molto elevato, in particolare sfruttando accurati modeli di illuminazione ed eventualmente dettagliati modelli 3D, ovvero gli oggetti matematici che rappresentano il database di ogni scena. Dipendentemente dalle capacità computazionali, ciò non è mai possibile nei sistemi real-time. Infatti, mentre spesso non c'è limite di tempo per la produzione di un'immagine 3D ad alta resa (eventualmente fotorealistica), la quale può richiedere da alcuni secondi a molte ore, nei sistemi real-time il tempo massimo concesso è 1/30, cioè circa 30 millisecondi. Per comparazione, nel film Disney Toy Story (1995), ogni singolo frame (dei 114.240 totali) richiedeva un tempo di elaborazione variabile tra 2 e 15 ore.

Esistono quindi due famiglie di sistemi di generazioni di immagini:

- Sistemi *non-realt-time* usati per produrre immagini ad altissima qualità in tempi dell'ordine di minuti/ore
- Sistemi *real-time* usati per produrre immagini di qualità inferiore in tempi dell'ordine dei millisecondi

Generalmente, i sistemi per la produzione di immagini 3D fotorealistiche si basano sulla tecnologia *ray tracing*, in cui il colore di ogni punto dell'immagine è calcolato tracciando un raggio che parte dall'osservatore e passante per il punto considerato. Per ogni raggio, sono considerate tutte le interazioni con tutte le superfici e luci presenti. Nei sistemi real-time, invece, il principio è opposto: i modelli presenti nella scena, possibilmente solo quelli visibili, vengono renderizzati senza preoccuparsi di quale area nell'immagine finale occuperanno. Quest'ultimo approccio è detto *world-to-screen*, al contrario del primo approccio, denominato *screen-to-world*.

A questo punto è essenziale parlare di quell'aspetto che ha, nella seconda metà degli anni novanta, cambiato radicalmente il modo di vivere l'esperienza videoulica (e non solo questa): l'accelerazione hardware. Con accelerazione hardware si intende la capacità di una periferica hardware, il cui chip è chiamato GPU (Graphics Processing Unit), di eseguire autonomamente

tutti i pesanti calcoli concernenti la grafica 3D, sgravando completamente (o quasi) da ogni compito la CPU. Fino all'avvento delle schede acceleratrici, infatti, tutti i calcoli erano eseguiti dalla CPU, limitando fortemente le potenzialità di resa grafica e costringendo i programmatori alla scrittura di complicato codice a basso livello (spesso in assembler) al fine di spremere al massimo le possibilità della CPU. I primi giochi 3D, come ad esempio DooM (1993), Quake (1996) o Unreal (1998), funzionavano in questo modo. Questi gioco erano sviluppati su motori grafici denominati "software render" per indicare che il rendering non era eseguito da un apposito hardware, ma via software, cioè attraverso codice fortemente ottimizzato scritto dal programmatore stesso.

Un motore grafico è un sistema software composto da varie componenti scritte con la possibilità di essere riutilizzabili, il cui scopo è la visualizzazione ed eventualmente l'interazione di un ambiente tridimensionale (definito esternamente). Spesso un motore grafico è definito un sistema software middleware, poichè fornisce gli strumenti necessari per lo sviluppo di un videogioco, anzichè consistere di un videogioco in senso stretto.

Il lavoro della tesi precedente riguardò la costruzione di un motore grafico basato su render software. Un motore grafico software produce risultati mediocri se paragonati alle moderne tecnologie, e paradossalmente risulta forse più complicato da realizzare, nonchè terribilmente meno performante. Tuttavia la sua realizzazione ha comportato l'analisi e lo studio di una grande famiglia di algoritmi molto importanti, alcuni dei quali oggi implementati all'interno delle moderne GPU.

Verso la fine degli anni novanta, con l'introduzione nel mercato delle prime schede acceleratrici a costi accessibili, si decretò la fine dell'era di tali motori grafici, in favore di motori grafici in grado di sfruttare le enormi capacità di calcoli delle nuove GPU. É interessante notare come le GPU contengano attualmente un numero di transistor ben più alto delle CPU, offrendo inoltre la possibilità di parallelizzare un elevato numero di calcoli.

In un motore grafico software, le operazioni di rendering sono scritte manualmente a basso livello. Nei motori grafici accelerati, invece, il sistema poggia su un set di API, ovvero procedure disponibili al programmatore che adempiono a diversi compiti, solitamente fornite da librerie come DirectX (di Microsoft) o OpenGL. Queste, comunicando con il driver il quale poi istruirà infine la GPU sulle operazioni richieste, hanno lo scopo di rendere del tutto trasparente il

substrato a basso livello messo a disposizione dalla GPU. Inoltre, aumentano notevolmente la scalabilità e la portabilità del codice, semplificando il compito del programmatore.

Il motore grafico real-time sviluppato in questa tesi usa la libreria DirectX nella versione 9.0c. Questa non è l'ultima versione disponibile, la quale, fra l'altro, offre potenzialità innovative; ma è stato ritenuto sufficiente, per gli scopi prefissati e per il carico di lavoro previsto per una singola persona, non passare allo studio dell'ultima versione della libreria. Si fa inoltre notare che gran parte dei titoli disponibili oggi nel mercato sono stati sviluppati con questa versione della libreria.

Applicazione (CPU) ➔ API (DirectX) ➔ Driver video ➔ GPU ➔ Output su video

1.2 Motore per Videogiochi

Fin dalla genesi del progetto discusso in questa tesi, è stato scelto di sviluppare un sistema che non fosse solamente un motore grafico, ma bensì qualcosa che si avvicinasse il più possibile ad un videogioco completo, che è il traguardo che l'autore cerca di raggiungere con i suoi studi.

Il sistema che sottosta ad un moderno videogioco è chiamato *Game Engine*, e il motore grafico rappresenta una parte di esso, anche se talvolta i nomi tendono a confondersi. Tuttavia la differenza fra i due termini tende talvolta ad assottigliarsi, fino a diventare sinonimi. Un game engine completo include componenti quali un renderer (i.e. il motore grafico), un physics engine (simulazione della fisica e interazione fra gli oggetti), gestione del suono e della musica, sistema di scripting, sistema per l'intelligenza artificiale, sistema per il networking (usato per il multiplayer), e componenti più specifiche come un gestore della memoria, gestione dei files, gestore input, sistema di threading, etc...

Il progetto sviluppato comprende un renderer, ovvero il modulo dedito alla visualizzazione della scena, un modulo per la gestione della fisica e un modulo per la gestione della logica, in cui è inclusa un'AI di base. In questa tesi sarà dedicata maggiore attenzione al renderer, il cui sviluppo ha coperto un arco di tempo di circa un anno. Fisica e logica, si ripete, di per se non sono indispensabili per lo sviluppo di un mero motore grafico. Tuttavia nessun videogioco potrebbe esistere senza queste componenti essenziali; inoltre un game engine è più

fruibile, presentabile e distribuibile di un graphic engine, poichè più completo. Da un punto di vista di completezza, si ritiene più conveniente e gratificante costruire un game engine magari con qualche qualità grafica in meno, rispetto ad un discreto motore grafico con il quale non è possibile interagire in alcun modo.

Va sottolineato, comunque, che lo sviluppo di un motore grafico completo (e quindi di tutte le componenti sopra citate) è in genere affidato a team composti da molte decine di programmatori con forte esperienza, con tempi per lo sviluppo che variano (per gli attuali titoli commerciali) dai due ai cinque anni. Pertanto il lavoro presentato in questa tesi, che è stato sviluppato da una singola persona, è da considerarsi parziale e non concluso, e sicuramente non comparabile con i moderni titoli presenti nel mercato.

1. 3 Deferred Renderer

Per quanto concerne l'aspetto tecnico del motore grafico, per il suo sviluppo è stato scelto il recente paradigma *Deferred Rendering*, il quale si contrappone al classico paradigma *Forward Rendering*, che ha dominato la struttura dei motori grafici fino alla seconda metà degli anni duemila.

Mentre nel forward rendering ogni oggetto è renderizzato e illuminato da ogni luce che lo influisce, nel deferred rendering vengono renderizzati tutti gli oggetti visibili una sola volta ignorando l'illuminazione, riempendo in un primo step un buffer detto G-Buffer (geometric buffer) con i parametri necessari al modello di illuminazione. Il valore di luminosità dei pixel del Gbuffer è in seguito calcolato effettuando un passaggio per ogni luce accumulando i risultati sul light buffer. Da qui il termine Deferred.

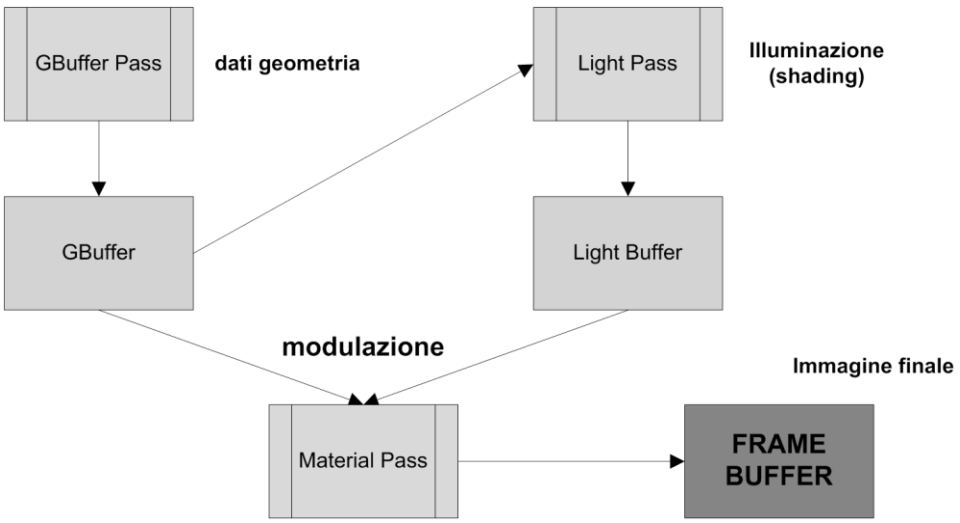


Figura 1.1: schema di un renderer di tipo deferred. Si esegue il GBuffer Pass, poi Light Pass e infine si compone l'immagine finale.

Siccome il G-Buffer ha una dimensione costante pari al frame buffer (immagine finale), ne consegue che la complessità del calcolo dell'illuminazione, ovvero la parte più importante e lenta del motore grafico, non dipende dalla complessità della scena. Ovvero, nel deferred rendering solamente ciò che è visibile è illuminato. Si chiama *Albedo Buffer* il buffer compreso nel GBuffer che contiene "il colore" non illuminato degli oggetti. Questi sono i passaggi:

1. Rendering scena senza luci nel GBuffer
2. Rendering del Light Buffer
3. Modulazione Albedo Buffer con Light Buffer
4. Effetti post-process e presentazione immagine finale

Ad esempio, se un oggetto è illuminato da sei luci, nel forward rendering l'oggetto deve essere renderizzato sei volte, al fine di accumulare il contributo di ogni luce nel buffer. Se accade che una parte dell'oggetto è coperta da un altro, essa verrà inutilmente considerata. Infatti, siccome molti oggetti vengono considerati come indivisibili, e rimanendo difficile individuare quali sono visibili nella scena finale e quali luci coinvolgono ogni oggetto, è frequente il caso in cui si va a calcolare la luminosità di svariati oggetti o porzioni di scena che poi non saranno visibili. Il deferred rendering elimina questo problema disaccoppiando la renderizzazione della

scena (1), e quindi la sua complessità, dal calcolo delle luci (2). Così gli oggetti non visibili che verranno velocemente renderizzati nel passaggio (1) verranno automaticamente ignorati nel passaggio (2).

Definendo N il numero di oggetti presenti, e L il numero di luci presenti, e supponendo che O(1) sia il tempo necessario al rendering di un oggetto, il vantaggio del paradigma deferred rendering è evidente osservando che la complessità del calcolo passa da O(N*L) a O(N+L).

Fra i principali svantaggi si fa notare una grande richiesta di bandwidth e l'impossibilità di trattare semplicemente oggetti trasparenti. Tali problematiche saranno discusse in seguito.

1.4 Il Mercato dei Videogiochi

Ci sarebbe molto da dire in merito a questo aspetto, ma ciò non è l'ambito di questa tesi. Molto velocemente diremo che il mercato dei videogiochi ha da qualche anno superato in fatturato il mercato musicale e cinematografico. Esso domina l'intero settore dell'intrattenimento, producendo un fatturato nel 2008 di un miliardo di euro. Negli ultimi tre anni, inoltre, esso ha raddoppiato gli introiti, traiato dall'incessante ascesa del mercato delle console. Il settore videoludico appare come una proficua applicazione delle tecniche ingegneristiche volte alla sviluppo di prodotti di educazione e intrattenimento.

2. Matematica della grafica 3D

Va premesso che questa non è una tesi di matematica, e il suo scopo è analizzare il funzionamento dell'engine sviluppato e non studiare approfonditamente la teoria degli spazi vettoriali e del calcolo matriciale. Il corpo di algoritmi e nozioni dell'engine è estremamente vasto, pertanto nella seguente parte i concetti saranno chiaramente esposti ma non dimostrati. Inoltre, non tutto ciò che riguarda la teoria degli spazi vettoriali e del calcolo matriciale è stato riportato e per questa ed altri approfondimenti, si rimanda alla bibliografia.

2.1 Vettori

2.1.1 Definizione

Il vettore è, in matematica, un segmento orientato, impiegato per rappresentare una grandezza avente verso, intensità (modulo) e direzione. Estendendo il concetto, si può dire che con vettore si intende l'insieme contenente tutti i segmenti orientati equipollenti fra di loro, cioè tutti i segmenti orientati aventi medesimo verso, direzione ed intensità ma aventi posizioni differenti nello spazio.

Nell'ambito della grafica 3D, si avrà a che fare con vettori a tre dimensioni, definiti quindi da tre numeri reali (x, y, z):

$$\vec{V} = (V_x, V_y, V_z)$$

Semplicisticamente, ogni vettore va pensato come un segmento avente l'origine come punto di partenza e le coordinate del vettore stesso come punto finale. In realtà si utilizzeranno i vettori anche per rappresentare dei semplici punti, proprio come nell'esempio del cubo. Il fatto è che i vettori possono essere pensati come un'estensione del concetto di punto, i quali godono di molte proprietà ulteriori che si analizzeranno.

2.1.2 Operazioni fra vettori

Innanzi tutto è opportuno ricavare l'equazione di un vettore dai due punti A e B. Il vettore risultante V è così definito:

$$\vec{V} = (B_x - A_x, B_y - A_y, B_z - A_z)$$

e sarà un vettore che avrà come direzione la retta congiungente **A** e **B**, come verso quello da **A** a **B** e come modulo la distanza fra i due punti, che è così definita partendo dal teorema di Pitagora esteso al caso tridimensionale:

$$M = \sqrt{Ax^2 + Ay^2 + Az^2}$$

sovente, infatti, quando si parla di lunghezza di un vettore si intende il suo modulo.

Prima di analizzare le principali operazioni fra i vettori, è bene definire cosa sia un vettore normalizzato. Normalizzare un vettore significa sostanzialmente mantenere inalterata la sua direzione ed il suo verso e portare il suo modulo a uno. Un vettore così formato è detto anche vettore unitario o versore.

L'operazione di normalizzazione è così definita:

$$\vec{B} = \vec{A} \cdot \frac{1}{|\vec{A}|}$$

I vettori normalizzati godono di una vasta diffusione nell'ambito della computer grafica poiché il loro impiego permette di semplificare molte operazioni, aumentando la velocità di calcolo..

Le operazioni di addizione e sottrazione sono così definite (si tralasciano eventuali applicazioni o dimostrazioni) :

$$\vec{A} \pm \vec{B} = (A_x \pm B_x, A_y \pm B_y, A_z \pm B_z)$$

Un'altra operazione usata è la moltiplicazione per scalare, la quale agisce sul modulo del vettore, incrementandolo o diminuendolo, o sul verso, negandolo:

$$k\vec{A} = (k \cdot A_x, k \cdot A_y, k \cdot A_z)$$

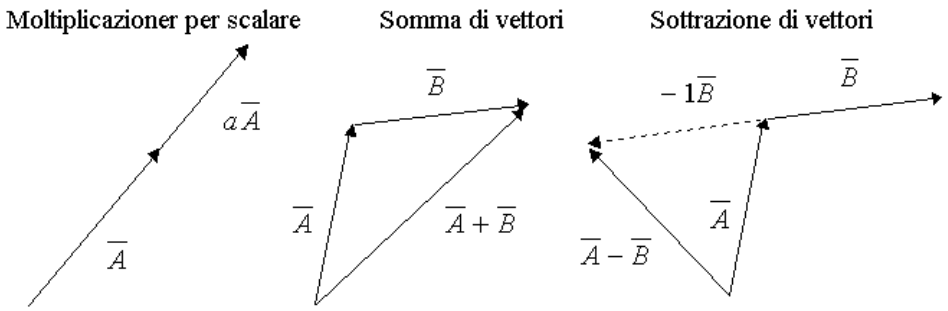


Figura 2.1: Operazioni di somma, sottrazione e moltiplicazione per scalare su vettori

Per quanto riguarda la moltiplicazione fra vettori la situazione è diversa. Esistono due tipi di moltiplicazioni: *prodotto scalare* (da non confondere con il prodotto per scalare) e *prodotto vettoriale*.

Il prodotto scalare è un numero reale così definito:

$$\vec{A} \cdot \vec{B} = \sum_{i=1}^{n} A_i B_i$$

ovvero è un numero reale definito come la somma del prodotto di ogni componente. Nel caso di tre dimensioni, diventa quindi:

$$\vec{A} \cdot \vec{B} = A_x B_x + A_y B_y + A_z B_z$$

Esso è una delle operazioni più usate nell'ambito della grafica 3D perché permette di ottenere una misura della differenza delle direzioni dei due vettori, e vale la proprietà commutativa, associativa e distributiva.

Più dettagliatamente, è vera questa proprietà:

$$\vec{A} \cdot \vec{B} = |\vec{A}||\vec{B}|\cos\alpha$$

Geometricamente, rappresenta la lunghezza della proiezione di A su B normalizzata. Ma questa, nel caso di due vettori normalizzati, è pari quindi al valore del coseno fra i due angoli. Si

può quindi decomporre facilmente un vettore nelle componenti che sono parallele e perpendicolari ad un altro vettore, come in figura.

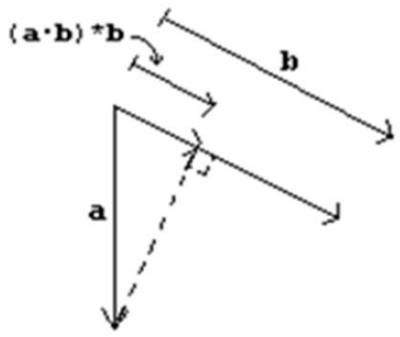


Figura 2.2: Proiezione di A su B mediante il prodotto scalare

La scrittura più usata diventa:

$$\cos\alpha = \frac{\vec{A}\cdot\vec{B}}{|\vec{A}|\cdot|\vec{B}|}$$

Nella pratica, quindi, assume un ruolo fondamentale in quanto permette di calcolare il coseno fra due vettori. Una proprietà molto usata del prodotto scalare è il segno. Si ha pertanto:

$\vec{A}\cdot\vec{B} > 0$ se $\alpha < 90°$
$\vec{A}\cdot\vec{B} = 0$ se $\alpha = 90°$
$\vec{A}\cdot\vec{B} < 0$ se $\alpha > 90°$

Il calcolo è utile, fra le varie cose, per determinare se un vettore giace da un lato di un piano o invece punta nell'altro, calcolo ad esempio utilizzato nel calcolo delle luci. Oppure se il risultato è pari a zero, si evince che i due vettori sono perpendicolari fra loro, ovvero ortogonali. Si vedrà poi in seguito che, per velocizzare le operazioni, si avrà spesso a che fare con vettori normalizzati, poiché con questi non è necessaria la divisione per il prodotto dei moduli.

L'altra operazione è il prodotto vettoriale. Il prodotto vettoriale fra due vettori tridimensionali ha come risultato un altro vettore, *ortogonale* ai due di origine. È così definito:

$$|\vec{A}x\vec{B}| = |\vec{A}|\,|\vec{B}|\sin\beta = (A_yB_z - B_yA_z, A_zB_x - B_zA_x, A_xB_y - B_xA_y)$$

Il vettore risultante sarà perpendicolare ai due di origine, quindi ortogonale al piano definito dai vettori d'origine, e il suo modulo sarà pari all'area del parallelepipedo che si viene prolungando i vettori d'origine. Il verso è definito dalla ben nota regola della mano destra. Tale vettore viene anche detto vettore normale del piano, il cui uso è molto utile nel campo della grafica tridimensionale:

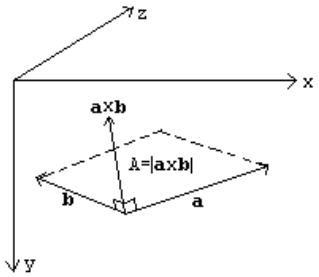


Figura 2.3: Esempio di prodotto vettoriale. Esso crea un vettore perpendicolare ai due di partenza

Il vettore normale pertanto definisce univocamente tutte le caratteristiche di un piano eccetto la distanza dall'origine. Un generico piano, quindi, può essere rappresentato dalla sua normale, più un altro vettore che definisce la distanza dall'origine degli assi.

Il prodotto vettoriale gode della proprietà distributiva secondo uno scalare, ma non della proprietà commutativa.

2.2 Matrici

2.2.1 Definizione

In un engine 3D, i calcoli sono eseguiti su una moltitudine di diversi sistemi di riferimento cartesiani. Passare da un sistema ad un altro richiede l'uso delle matrici di trasformazione, il cui approccio permette di ridurre al minimo i calcoli richiesti, richiedendo nove moltiplicazioni per ogni punto. Descrivendo con una sola matrice quadrata di ordine quattro un intero sistema di riferimento (diversamente orientato rispetto a quello assoluto, ad esempio la

posizione dell'osservatore) è sufficiente una moltiplicazione per trasformare un vettore, ma più generalmente un oggetto, nel nuovo sistema di coordinate.

Si da in seguito un veloce ripasso delle proprietà fondamentali delle matrici, in quando questa tesi non ha lo scopo di approfondire in dettaglio tutti gli aspetti matematici, per la cui analisi si rimanda alla bibliografia.

Una matrice è una tabella di numeri reali o complessi. Essa avrà una lunghezza n ed un'altezza m corrispondente alle dimensioni della tabella dei valori. Una matrice per cui n = m è detta matrice quadrata ed n è detto ordine della matrice. Si usa la scrittura Mnm per indicare l'elemento che risiede alla riga n-esima e alla colonna m-esima.

$$M = \begin{bmatrix} M_{1,1} & M_{1,2} & \dots & M_{1,m} \\ M_{2,1} & M_{2,2} & \dots & M_{2,m} \\ \dots & \dots & \dots & \dots \\ M_{n,1} & M_{n,2} & \dots & M_{n,m} \end{bmatrix}$$

Le matrici sono largamente usate in quanto, come si vedrà poi, permettono di rappresentare completamente i sistemi di riferimento che compongono la totalità dell'engine 3D. Permettono, inoltre, di agire facilmente sui singoli vettori, qualora sia necessario operare una trasformazione.

2.2.2 Operazioni fra matrici

La somma di due matrici n x m si esegue banalmente, secondo la quale ogni elemento della matrice risultante è pari alla somma dei rispettivi elementi delle matrici addende:

$$A + M = M + A = \begin{bmatrix} A_{1,1} + M_{1,1} & A_{1,2} + M_{1,2} & \dots & A_{1,m} + M_{1,m} \\ A_{2,1} + M_{2,1} & A_{2,2} + M_{2,2} & \dots & A_{2,m} + M_{2,m} \\ \dots & \dots & \dots & \dots \\ A_{n,1} + M_{n,1} & A_{n,2} + M_{n,2} & \dots & A_{n,m} + M_{n,m} \end{bmatrix}$$

Anche per le matrici è definito il prodotto per scalare, per cui ogni elemento è moltiplicato per lo scalare stesso:

$$aM = Ma = \begin{bmatrix} aM_{1,1} & aM_{1,2} & ... & aM_{1,m} \\ aM_{2,1} & aM_{2,2} & ... & aM_{2,m} \\ ... & ... & ... & ... \\ aM_{n,1} & aM_{n,2} & ... & aM_{n,m} \end{bmatrix}$$

Due matrici A ed M possono essere moltiplicate fra loro, ammesso che il numero di colonne di A sia lo stesso del numero di righe di M. Se A è una matrice n x m e M una matrice m x p, il prodotto AM è una matrice n x p i cui elementi (i, j) sono così definiti:

$$(AM)_{ij} = \sum_{k=1}^{m} A_{ik} M_{kj}$$

Un altro modo, più semplice, per vedere la moltiplicazione è pensare ogni elemento (i,j) di AM come il prodotto scalare della i-esima riga di A e della j-esima colonna di M.

Molte sono le proprietà soddisfatte dalle operazioni fra matrici, in particolare vige la proprietà commutativa, associativa e distributiva rispetto all'addizione e rispetto alla moltiplicazione per scalare. Al contrario, per il caso della moltiplicazione fra matrici non vige la proprietà commutativa né la distributiva.

Esiste una matrice n x n chiamata matrice identità, identificata da In, la quale rappresenta l'elemento neutro della moltiplicazione, ovvero MIn = InM = M per qualsiasi matrice quadrata M. La matrice identità ha questo aspetto:

$$I_n = \begin{bmatrix} 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & 1 \end{bmatrix}$$

Normalmente la matrice $\mathbf{I_n}$ è più semplicemente identificata solo con **I**.

Una matrice *n x n* **M** è invertibile se esiste una matrice, che denoteremo $\mathbf{M^{-1}}$, tale per cui $\mathbf{MM^{-1} = M^{-1}M = I}$. Non tutte le matrici ammettono inversa, ed inoltre il prodotto di due matrici invertibili è un'altra matrice invertibile anch'essa.

La matrice inversa è utilizzata in aspetti particolari dell'engine 3D, d'altra parte però lo stesso risultato può essere ottenuto percorrendo una via più semplice e più veloce, e pertanto non

sarà analizzato l'algoritmo per ricavare la matrice inversa di una data. Per un'implementazione alternativa, si rimanda alla bibliografia.

Il *determinante* di una matrice è una quantità scalare derivata dagli elementi della matrice, ed è chiamato *det**M***. Usando una simbologia classica, quando si rappresentano gli elementi di una matrice, si sostituiscono le parentesi quadre con due barre verticali per indicare che si sta calcolando il determinante. Ad esempio, nel caso di una matrice **M** *3 x 3*:

$$\det M = \begin{vmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{vmatrix}$$

L'algoritmo di calcolo del determinante segue una formula ricorsiva che diventa complicata già per matrici piccole. In ogni caso nell'engine 3D si avrà a che fare con sistemi a tre dimensioni, quindi con matrici rappresentative di *3 x 3*, pertanto si è scelto di riportare più semplicemente solo la formula usata per il calcolo del determinante di una matrice *3 x 3*.

Il determinante di una matrice *1 x 1*, ovvero uno scalare, è lo scalare stesso. Per una matrice *2 x 2* e *3 x 3* il determinante assume questa formula:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12}\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13}\begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

$$= a_{11}(a_{22}a_{33} - a_{23}a_{32}) - a_{12}(a_{21}a_{33} - a_{23}a_{31}) + a_{13}(a_{21}a_{32} - a_{22}a_{31})$$

Il determinante è utile perché da informazioni riguardo ai vettori linea (o colonna) che compongono la matrice, cosa che si vedrà in dettaglio in seguito. Inoltre, una matrice è invertibile solamente se il suo determinante è diverso da zero.

Banalmente, una matrice può quindi essere utilizzata per rappresentare un vettore in questo modo:

$$A = \begin{bmatrix} A_x & A_y & A_z \end{bmatrix}$$

In questo caso siamo in presenza di una matrice notevole, detta vettore riga. Esiste anche il vettore colonna, che è il vettore riga trasposto, indicato con $\mathbf{A^T}$. Se moltiplichiamo fra loro un vettore riga e un vettore colonna otteniamo quindi uno scalare, pari al prodotto scalare fra i due vettori:

$$\begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = ax + by + cz$$

Le matrici per cui $n = p$ vengono dette *quadrate* di ordine n e possono quindi contenere n vettori ad n dimensioni. Sfruttando questa particolarità, possiamo pensare di rappresentare un intero spazio vettoriale semplicemente scrivendo nella matrice i suoi vettori generatori. Nel caso di uno spazio 3D:

$$\begin{bmatrix} Xx & Xy & Xz \\ Yx & Yy & Yz \\ Zx & Zy & Zz \end{bmatrix}$$

dove **X**, **Y** e **Z** sono i vettori generatori dello spazio 3D.

L'operazione senza dubbio più importante è la moltiplicazione. Nella matrice risultante, ogni elemento è ottenuto moltiplicando rispettivamente la riga e la colonna dell'argomento. Dato che colonne e vettori devono avere la stessa dimensione per poter essere moltiplicati, risultano apparenti delle restrizioni riguardo alle dimensioni delle due matrici da moltiplicare. Infatti, la lunghezza della prima matrice dev'essere pari all'altezza della seconda matrice. In questo caso il prodotto è dato da:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & i & j \end{bmatrix} \begin{bmatrix} x & k \\ y & l \\ z & m \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} & \begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} k \\ l \\ m \end{bmatrix} \\ \begin{bmatrix} d & e & f \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} & \begin{bmatrix} d & e & f \end{bmatrix} \begin{bmatrix} k \\ l \\ m \end{bmatrix} \\ \begin{bmatrix} g & i & j \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} & \begin{bmatrix} g & i & j \end{bmatrix} \begin{bmatrix} k \\ l \\ m \end{bmatrix} \end{bmatrix}$$

Interessante è notare come il prodotto di un vettore per una matrice da come risultato un altro vettore. Questa è una caratteristica fondamentale, in quando permette di trasformare il vettore, e quindi un intero oggetto. Le trasformazioni possibili, però, sono solo quelle lineari. Per ovviare a questa limitazione, e quindi permettere anche operazioni come la traslazione, si è introdotta una nuova notazione per i vettori:

$$A = \begin{bmatrix} x & y & z & w \end{bmatrix}$$

dove la variabile *w* varrà generalmente *1*. In questo modo è possibile introdurre nella matrice di trasformazione il vettore traslazione, le cui componenti saranno sommate al vettore di partenza. Ciò verrà spiegato in dettaglio nei paragrafi successivi.

2.3 Spazi vettoriali

Nella geometria, le forme delle entità sono rappresentate fondamentalmente da coordinate di alcuni punti fra loro connessi. Per esempio, quattro punti opportunamente collegati formano un rettangolo, mentre tre un triangolo (figura che poi si scoprirà essere fondamentale).

Il sistema di coordinate usato è quindi lo spazio *euclidiano* e, ciò detto, si definisce uno spazio vettoriale come un corpo *V* di componenti detti vettori, per il quale è definita l'addizione e la moltiplicazione, e per il quale le seguenti proprietà sono soddisfatte, sapendo che **P** e **Q** sono due vettori appartenenti a *V* e *z* è uno scalare:

- V è chiuso secondo l'addizione
- V è chiuso secondo la moltiplicazione
- Esiste un elemento in *V* detto **0** in virtù del quale per ogni elemento **P** si ha **P + 0 = 0 + P = P**
- Per ogni elemento **P** in *V* esiste un elemento **Q** in *V* tale che **P + Q = 0**
- L'addizione è associativa
- La moltiplicazione per scalare è associativa
- La moltiplicazione per scalare è distributiva secondo l'addizione. Ovvero z(**P + Q**) = z**P** + **z****Q**
- L'addizione di scalari è distributiva secondo la moltiplicazione per scalare. Ovvero (a + b)**P** = a**P** + b**P**

Si definisce inoltre un set di *n* vettori $\{\mathbf{e}_1, \mathbf{e}_2,\ldots,\mathbf{e}_n\}$ linearmente indipendenti se non esiste un set di numeri reali $a_1, a_2,\ldots,a_n$, dove almeno uno degli aj è diverso da zero, per cui:

$$a_1\mathbf{e}_1 + a_2\mathbf{e}_2 + \ldots + a_n\mathbf{e}_n = 0$$

Altrimenti il set $\{\mathbf{e}_1, \mathbf{e}_2,\ldots,\mathbf{e}_n\}$ è linearmente dipendente. Uno spazio vettoriale n-dimensionale è tale se può essere generato da un set di *n* vettori linearmente indipendenti, i quali comporranno la *base* dello spazio vettoriale, ovvero i vettori usati per generare tutti gli altri possibili, partendo da combinazioni lineari di questi. Queste combinazioni lineari sono la base delle trasformazioni che *tutti gli oggetti* subiscono nel percorso della pipeline grafica.

Più precisamente, una base *B* per uno spazio vettoriale *V* è un set di *n* vettori linearmente indipendenti $B = \{\mathbf{e}_1, \mathbf{e}_2,\ldots,\mathbf{e}_n\}$ per cui, dato un qualsiasi elemento **P** in *V*, esiste una serie di numero reali $a_1, a_2,\ldots,a_n$ per i quali:

$$\mathbf{P} = a_1\mathbf{e}_1 + a_2\mathbf{e}_2 + \ldots + a_n\mathbf{e}_n$$

Ogni base di uno spazio n-dimensionale ha esattamente n vettori generatori. Ad esempio è impossibile trovare una base di quattro vettori linearmente indipendenti per $\Re^3$, poiché uno di essi sarà generato dagli altri.

Sono infinite le scelte per definire una base per uno spazio vettoriale, però nel contesto seguente verrà scelta una base che soddisfi due proprietà in particolare, scelta che praticamente viene fatta per ogni engine 3D:

- *I vettori generatori devono avere lunghezza unitaria*
- *I vettori generatori devono essere ortogonali fra loro*

Una delle base più usate nel campo degli engine 3D che soddisfi alle condizioni date, e quella che sarà usata in questa sede, è la seguente:

$$\vec{i} = <1,0,0>$$
$$\vec{j} = <0,1,0>$$
$$\vec{k} = <0,0,1>$$

Tale base andrà a rappresentare il sistema di coordinate assoluto dell'intero mondo di oggetti tridimensionali.

2.4 Trasformazioni

Come già accennato nei capitoli precedenti, nel percorso delle pipeline i vettori subiscono varie trasformazioni geometriche. Le trasformazioni più largamente usate sono quelle per le quali i vertici che compongono gli oggetti vengono trasformati da un sistema di riferimento ad un altro con lo scopo di collocarli dinamicamente nello spazio, finendo poi con la trasformazione secondo le coordinate dell'osservatore, o telecamera (generalmente questo è l'ultimo passo). Altre trasformazioni, la cui applicazione purtroppo non sarà trattata in questa sede, riguardano i calcoli necessari per individuare le collisioni fra gli oggetti. Fra queste trasformazioni figurano operazioni ben note quali la traslazione, il cambio di scala e le rotazioni.

Il concetto di trasformazione lineare e l'idea di poter rappresentare uno spazio vettoriale con una matrice è un punto cardine degli engine 3D.

2.4.1 Trasformazioni Lineari

Si supponga di aver creato un sistema di riferimento vettoriale *C* definito da un'origine e tre assi per le tre dimensioni, nel quale un punto **P** ha le coordinate <*x*, *y*, *z*>. I valori *x*, *y* e *z*

possono essere pensati come la distanza che si deve percorrere lungo ogni asse partendo dall'origine con lo scopo di giungere al punto P.

Si supponga ora di introdurre un nuovo sistema di riferimento *C'*, nel quale le coordinate *<x', y', z'>* possono essere espresse con una combinazione lineare delle coordinate *<x, y, z>* in *C*.

Pertanto si potrà scrivere:

$$x'(x,y,z) = U_1 x + V_1 y + W_1 z + T_1$$
$$y'(x,y,z) = U_2 x + V_2 y + W_2 z + T_2$$
$$z'(x,y,z) = U_3 x + V_3 y + W_3 z + T_3$$

Tale procedimento costituisce una trasformazione lineare da *C* a *C'* e può più agevolmente essere scritta in forma matriciale in questo modo:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} U_1 & V_1 & W_1 \\ U_2 & V_2 & W_2 \\ U_3 & V_3 & W_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix}$$

Le coordinate *x'*, *y'* e *z'* possono essere pensate come la distanza che si deve percorrere lungo gli assi in *C'* per raggiungere il punto **P**. In altre parole la trasformazione correla i due sistemi di riferimento e se applicata ad un punto del primo sistema, produce le coordinate dello stesso punto però relative al secondo sistema di riferimento.

Il vettore **T** rappresenta la traslazione dall'origine di *C* all'origine di *C'*, e la matrice i cui vettori colonna sono **U**, **V** e **W** rappresenta come l'orientamento degli assi sia cambiato passando da *C* a *C'*. È possibile il procedimento inverso, assumendo che la trasformazione sia invertibile, il calcolo diviene:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} U_1 & V_1 & W_1 \\ U_2 & V_2 & W_2 \\ U_3 & V_3 & W_3 \end{bmatrix}^{-1} \left(\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} - \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} \right)$$

In seguito si combinerà una matrice *3 x 3* con un vettore traslazione **T** in un'unica matrice *4 x 4*. Prima però di giungere a quel punto, si analizzeranno le trasformazioni per le quali il

vettore **T** è nullo, ovvero i cambi di scala e le rotazioni. Vale a dire nei casi in cui i due sistemi di riferimento hanno la stessa origine, ma gli assi orientati in maniera differente.

Più trasformazioni possono essere concatenate e rappresentate da un'unica matrice e una traslazione. Ad esempio, le coordinate dei vertici possono dover essere trasformate dallo spazio dell'oggetto allo spazio del mondo (assoluto), e da questo allo spazio della telecamera. Le due trasformazioni possono essere concatenate in una sola matrice che mappi direttamente le coordinate dell'oggetto in quelle della telecamera, ottenendo di fatto un notevole decremento dei calcoli da eseguire, cosa necessaria per ogni buon engine 3D in real-time.

Va sottolineato che nell'ambito degli engine 3D abitualmente si fa uso di basi ortonormali, nelle quali i vettori generatori hanno lunghezza unitaria e sono ortogonali (perpendicolari) fra loro. Tale caratteristica permette di preservare le lunghezze e gli angoli di tutti i vettori interessati dalle trasformazioni, che nel contesto della grafica si traduce in un'assenza di distorsione degli oggetti.

Nelle tre dimensioni, quindi, una base *B* è definita da tre vettori generatori $\mathbf{V_1}$, $\mathbf{V_2}$ e $\mathbf{V_3}$, quindi da una matrice *3 x 3*. Essendo la base ortonormale, vale la seguente proprietà $\mathbf{V_1} \times \mathbf{V_2} = \mathbf{V_3}$, la quale garantisce l'ortogonalità dei vettori generatori.

Infine, quindi, una base è definita da una matrice *3 x 3* e il passaggio da una base all'altra è ottenuto mediante una moltiplicazione da matrici, la quale esplica la trasformazione lineare vista in precedenza. Le matrici usate per trasformare gli oggetti vengono dette matrici di trasformazione.

- **Matrice di scala**

Per scalare un vettore **P** di un fattore *a*, è sufficiente calcolare **P'** = *a***P**. In tre dimensioni, l'operazione può essere esplicata mediante un prodotto per matrice

$$P' = \begin{bmatrix} P_x & P_y & P_z \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} = \begin{bmatrix} P_x \cdot s_x & P_y \cdot s_y & P_z \cdot s_z \end{bmatrix}$$

Il cambio di scala è detto uniforme se $s_x = s_y = s_z = s$.

- **Matrice di rotazione**

È possibile trovare una matrice *3 x 3* che ruoti il sistema di coordinate, ovvero i vettori generatori, di un angolo θ rispetto agli assi *x, y* e *z* senza troppa difficoltà. Anzitutto si considera una singola rotazione per un angolo positivo attorno ad un asse *S*. Pertanto si troverà dapprima una formula generica per una rotazione nel caso bidimensionale, poiché se si ruota attorno ad un asse, supposto quell'asse perpendicolare all'osservatore, si opera un'operazione bidimensionale.

Si supponga di avere quindi un punto $\boldsymbol{A} = (A_x, A_y)$ e di volerlo ruotare di θ gradi. Se si fa riferimento a quanto detto riguardo ai vettori generatori, è semplice trovare le formule di rotazione. Siano **X** e **Y** i vettori generatori ortonormali di un piano bidimensionale standard, con θ quindi pari a zero:

$$\bar{X} = (\cos(\theta), \sin(\theta)) = (1,0)$$
$$\bar{Y} = (\cos(\theta + 90), \sin(\theta + 90)) = (-\sin(\theta), \cos(\theta)) = (0,1)$$

Allora le coordinate finali del punto saranno:

$$\bar{A} = (A_x\bar{X}, A_y\bar{Y}) = A_x(\cos(\theta), \sin(\theta)) + A_y(-\sin(\theta), \cos(\theta))$$

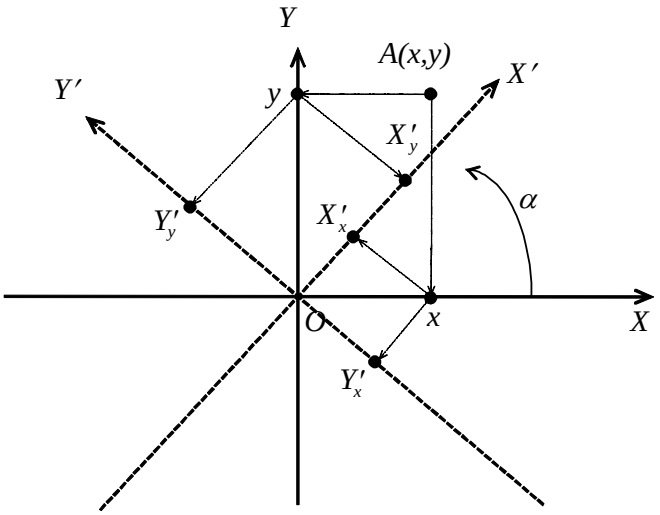


Figura 2.4: Rotazione su due assi. I vettori generatori vengono ruotati, e conseguentemente anche il punto A

In forma matriciale, la matrice della base del piano considerato è la seguente:

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}\Bigg|_{\theta=0} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Il sistema può essere liberamente ruotato variando l'angolo θ, il quale varia i vettori generatori. Tute le rotazioni sono basate su questa matrice, e ogni rotazione attorno ad un asse può essere effettuata partendo da tale matrice applicandola ai due assi rimanenti tenendo invariato quello di rotazione. Infatti, come già detto prima, è possibile pensare la rotazione come un cambiamento di orientamento del sistema di riferimento.

Matrice di rotazione per l'asse *Z*:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & sen\theta & 0 \\ -sen\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Si noti come non vengano modificate le proprietà del vettore generatore dell'asse Z.

Matrice di rotazione per l'asse X:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & sen\theta \\ 0 & -sen\theta & \cos\theta \end{bmatrix}$$

Matrice di rotazione per l'asse Y:

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & sen\theta \\ 0 & 1 & 0 \\ -sen\theta & 0 & \cos\theta \end{bmatrix}$$

Una rotazione generale è ottenuta concatenando, tramite moltiplicazione, le tre matrici di rotazione.

2.4.2 Sistema di coordinate omogeneo

Fino a questo punto si ha avuto a che fare solo con trasformazioni che potevano essere espresse da un'operazione riguardante una matrice *3 x 3* e un vettore tridimensionale. Una serie di trasformazioni analoghe può essere rappresentata da un'unica matrice quadrata di ordine tre, uguale al prodotto delle singole matrici. Un'importante trasformazione, però, è stata esclusa, e si tratta della traslazione. Una base di uno spazio vettoriale può semplicemente essere traslata, senza modificare l'orientamento degli assi né le loro lunghezze, sommando un vettore traslazione (offset). Questa operazione non può essere espressa mediante una matrice *3 x 3*. Quindi, per trasformare un vettore **P** da un sistema di coordinate ad un altro (cambio base) è necessario eseguire questa operazione:

$$\overline{P}' = \overline{MP} + \overline{T}$$

Dove **M** è la matrice invertibile *3 x 3* e **T** è il vettore 3D di traslazione. Eseguire in successione due operazioni simili alla precedente, o equivalentemente operare due trasformazioni ad un vettore, è una cosa tutt'altro che semplice, e la cosa peggiora ben presto se la serie di trasformazioni da eseguire aumenta.

Fortunatamente è stata introdotta una forma compatta ed elegante per poter rappresentare l'intera trasformazione con un'unica matrice *4 x 4*, estendendo i vettori ad un *sistema di coordinate omogeneo* a quattro dimensioni. Un punto 3D **P** è esteso a quattro dimensioni impostando la quarta coordinata, che si chiamerà coordinata *w*, al valore unitario. Si può ora procedere alla costruzione della matrice **F** *4 x 4* corrispondente alla matrice **M** di scala e rotazione e al vettore **T** di traslazione in questo modo.

$$F = \left[\begin{array}{c|c} M & T \\ \hline 0 & 1 \end{array}\right]$$

Nel caso tridimensionale si ottiene:

$$F=\begin{bmatrix} M_{11} & M_{12} & M_{13} & T_x \\ M_{21} & M_{22} & M_{23} & T_y \\ M_{31} & M_{32} & M_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La quale permette facilmente di moltiplicare un vettore, o un punto 3D, ottenendo la rotazione, il cambio di scala e la traslazione in un solo passaggio. Ovviamente le matrici siffatte possono essere concatenate.

3. La grafica tridimensionale

Il problema nasce dal fatto di dover visualizzare su uno schermo piatto delle figure a tre dimensioni. Due sono le principali tecniche utilizzate. Prima di una loro trattazione però è bene definire alcuni termini.

Nell'ambito della grafica 3D spesso si parlerà di *oggetti* e di *mondo*. Nel primo caso si intende una figura 3D più o meno complessa, definita da punti nello spazio fra loro connessi. Per esempio un cubo (figura semplice) o un'automobile (figura complessa) potrebbero essere degli oggetti 3D. Per mondo si intende invece l'insieme, o il database, di tutti gli oggetti presenti, cioè la *scena*. Nella scena tutti gli oggetti hanno una dimensione ed una collocazione spaziale, eventualmente variabile.

La scena è quindi l'entità principale, ovvero il contenitore di tutte le entità. Generalmente si chiama *telecamera* (o *camera*) un punto di vista *orientato* perfettamente *collocato* nella scena. In questa sede spesso si userà il termine *osservatore* in sostituzione del termine telecamera. Per *luce*, poi, si intende una fonte di luce di un determinato tipo anch'essa spazialmente collocata ed orientata, anche se la questione dell'illuminazione non è trattata in questa sede. Sono molti i tipi di oggetti che compongono la scena, ma lo scopo di questa tesi non è la gestione di una scena complicata, ma bensì la sua rappresentazione su display.

Ora, disponendo di una scena 3D, il problema è rappresentarla in un display a due dimensioni. Come già accennato, esistono sostanzialmente due famiglie di metodi in grado di adempiere a questo compito:

- **World to Screen**
- **Screen to World**

I metodi *World to Screen* sono i più utilizzati sebbene siano molto difficili da implementare, ed abbiano generalmente una resa grafica peggiore. Infatti non vengono usati la dove la qualità grafica è indispensabile, mentre hanno moltissimo successo nel campo dei sistemi real-time, quali la simulazione e il settore videoludico. Il principio su cui si basano è il seguente.

Ogni oggetto 3D è trasformato per mezzo delle proiezioni in una figura a due dimensioni, che poi verrà renderizzata e rasterizzata a video (*da oggetto a schermo* appunto). Notare che quando si parla di engine 3D, il termine renderizzare ingloba il termine rasterizzare.

I metodi *Screen to World* lavorano invece sotto un'altra ottica. In questo caso, piuttosto che partire dall'oggetto nello spazio, si parte da ogni punto del display - il numero di punti è pari al prodotto dell'altezza e della larghezza del display, detta risoluzione. Da esso si traccia un raggio che intersecherà qualche oggetto nel mondo (ecco perché screen-to-world) e, una volta individuata la collisione, si ricaverà il suo punto che verrà poi riportato sul display. Solitamente gli algoritmi di questo tipo vengono definiti con il nome di *ray-tracing*.

Il principale vantaggio è una maggiore resa grafica, quasi foto realistica, a scapito però della velocità computazionale. Questi algoritmi permettono l'introduzione di molte caratteristiche altrimenti molto difficili (se non impossibili) da implementare, quali una simulazione realistica delle luci e delle riflessioni, dell'interazione fra gli oggetti (global illumination) o degli specchi. Questi metodi, molto più *lenti* degli altri, vengono usati per esempio nell'industria cinematografica (si pensi ai film fatti con la computer graphics) o per gli effetti speciali, e nel campo del design industriale, e più generalmente la dove si richiede alta qualità grafica.

3.1 Ray Tracing – Accenno

Il ray tracing è la tecnica di renderizzazione che permette di ottenere i migliori risultati grafici. Generalmente non è però utilizzata nel campo negli applicativi in real-time, poiché la sua implementazione risultata piuttosto esosa in termini computazionali.

Questo paradigma si propone di calcolare l'immagine tracciando dei "raggi" (da qui il termine ray) che partono dal punto di vista fino al mondo virtuale. L'implementazione base consiste nel tracciare un raggio per ogni pixel dello schermo e di calcolare le sue intersezioni con le primitive presenti nel mondo. Di tutti i punti intersecati, si sceglie quello con distanza minore, poiché gli altri saranno naturalmente nascosti da quest'ultimo, tenendo traccia dell'oggetto interessato. Conoscendo le proprietà del materiale di cui è composto l'oggetto, si risale al colore del pixel da visualizzare. In seguito fu introdotta l'interazione con la luce. L'idea base è tracciare un nuovo raggio dal punto in esame (visto dall'osservatore) verso ogni sorgente di luce, cercando

eventuali oggetti che ostacolino il raggio. In questo caso la luce non ha effetto, altrimenti si usa un modello d'illuminazione per illuminare il pixel sommando i contributi di ogni luce. Inoltre, se il materiale è non opaco, il raggio proseguira rifratto o riflesso, a cui segue una nuova iterazione.

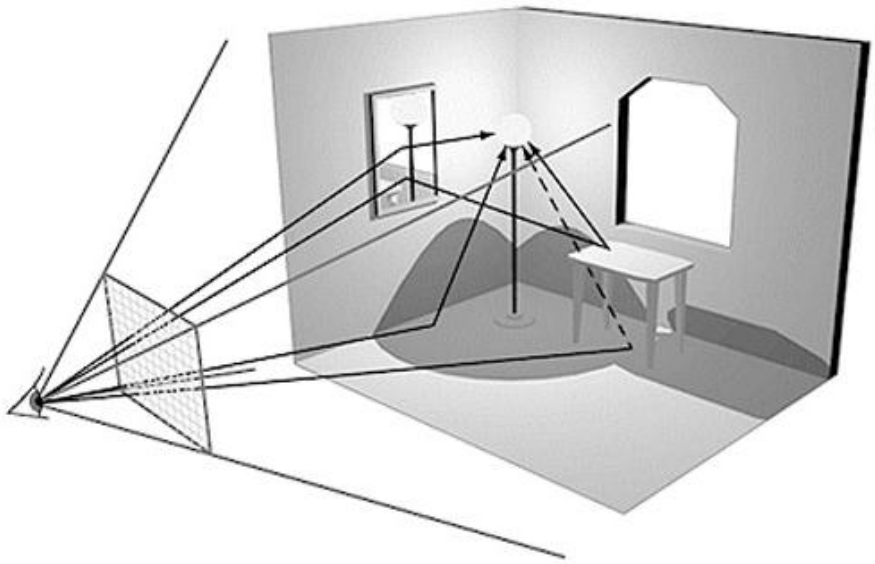


Figura 3.1: Nel ray-tracing, i raggi partono dall'osservatore fino ad intersecare gli oggetti del mondo

Riassumento questi sono gli steps:

- L'immagine è suddivisa un una matrice di punti (pixel)
- Per ogni punto si traccia un raggio dal punto di vista alle superfici della scena
- Considerando il punto visibile, si traccia un raggio dal questo punto ad ogni sorgente di luce diretta: se il raggio non è ostacolato si somma il contributo della sorgente calcolato seguendo un modello di illuminazione
- Se la superficie è non opaca, si prosegue il raggio fino ad incontrare un'altra superficie
- Si ripete il processo fino a raggiungere una superficie opaca o fino a completare un numero massimo di iterazioni

In questa introduzione ci si limiterà a dare uno rapido sguardo a come avviene l'individuazione delle intersezioni.

La telecamera è definita attraverso un punto O, che descrive la posizione del centro del piano di vista, da due vettori ortogonali **S**(S_x, S_y, S_z) e **T**(T_x, T_y, T_z) che descrivono l'orientamento, e da un vettore **V** che descrive la posizione dell'osservatore rispetto al piano di vista. Da questo punto è sufficiente calcolare le coordinate del mondo virtuale partendo da ogni punto dello schermo (cioè il piano di vista):

$$R_x = O_x + sS_x + tT_x$$
$$R_y = O_y + sS_y + tT_y$$
$$R_z = O_z + sS_z + tT_z$$

Il vettore risultante **R** non è altro che il punto nello spazio virtuale nel quale cade l'inizio del raggio. Per trovare l'equazione di quest'ultimo, è sufficiente scrivere l'equazione di una retta con origine **V** e passante per **R**:

$$I = V + t(V - R)$$

Variando il parametro **t** è possibile trovare tutti i punti del raggio. In questo modo, per ogni piano presente nello spazio, si calcola, se presente, la sua intersezione con il raggio e, da questo punto, la sua distanza dal piano di vista. Scegliendo l'intersezione dalla distanza minore, è facilmente possibile tracciare l'intera scena.

In realtà il Ray-Tracing è molto più complesso di quanto descritto. Il suo sviluppo si avvale di modelli d'illuminazione molto raffinati, che non si limitano ad un singolo raggio, ma che si avvalgono di più raggi (riflessione, rifrazione ed ombra) e dell'applicazione ricorsiva di essi. È possibile dire, infine, che la sua implementazione da origine a potenti software di grafica tridimensionale, usati nell'industria cinematografica o la dove la qualità fotografica è un parametro essenziale.

3.2 Pipeline World to Screen

La *pipeline grafica* è il flusso di operazioni a cui un oggetto 3D è sottoposto partendo dalla sua definizione (modello 3D) fino a giungere alla rappresentazione finale nel dispaly. Un motore grafico dovrà sviluppare ognuno dei passaggi della pipeline. Nei metodi *world to screen* gli oggetti vengono proiettati sul campo di vista. Lo scopo è rappresentare oggetti a tre dimensioni su un display a *due* dimensioni. A questo scopo si ricorre alle *proiezioni*, che effettuano una trasformazione di un oggetto 3D su un *piano di proiezione 2D*, di fatto operando un *cambio di base* che porta al sistema di riferimento chiamato *screen space*, il quale è allineato con il display. L'oggetto 2D risultante è infine rasterizzato, ovvero tracciato nello spazio discreto chiamato *frame buffer* che rappresenta l'immagine finale, attraverso *scanline* renderer. Gli elementi che compongono il frame buffer vengono concettualmente chiamati *fragment*, e in genere corrispondono ai pixel del display.

Il colore finale del fragment è definito dal materiale di cui è composto l'oggetto, e dall'interazione di questo con le luci presenti nella scena. Una volta scelto un *modello di illuminazione*, questo è usato per calcolare l'intensità e il colore della luce che incide su ogni punto dell'oggetto. Tale fase è detta *shading.* A partire dall'inizio degli anni duemila, la tecnologia permette di applicare lo shading direttamente ai singoli fragment, ottenendo una illuminazione *per-pixel*, che garantisce risultati molto convincenti. Quella seguente è una pipeline essenziale per un motore grafico 3D di questo tipo:

1. *Trasformazioni oggetti*
    a. *local to world*
    b. *(VSD)*
    c. *world to view*
2. *3D clipping*
3. *Proiezione prospettica*
4. *Rasterizzazione e interpolazione*
    - *Operazioni su pixel (shading per-pixel)*
5. *FRAME BUFFER*

In maniera concisa, gli oggetti vengono trasformati secondo il sistema di riferimento dell'osservatore, vengono proiettati sul piano di proiezione e poi rasterizzati calcolando ogni fragment che andrà scritto sul frame buffer.

La fase (1) non è strettamente legata al rendering, ma fa parte del modo di gestire la scena. La fase (2) è eseguita su CPU e serve ad evitare al renderer di trattare oggetti che poi non saranno visibili. Il lavoro vero e proprio del renderer inizia dalla fase (3), a partire dalla quale i vertici vengono proiettati, vengono generati i fragment e calcolato il colore finale di ognuno di essi. Vengono ora introdotte le varie fasi della pipeline.

3.2.1 Trasformazione oggetti

All'atto della proiezione è necessario disporre di oggetti i cui vertici siano già tutti trasformati in modo spazialmente coerente con l'osservatore, in modo tale che l'origine del sistema di riferimento sia la posizione dell'osservatore, e l'orientamento di quest'ultimo determini l'angolazione del sistema di riferimento degli oggetti. I vertici di un oggetto quindi, prima di essere proiettati e/o illuminati, subiscono tre trasformazioni:

- *Model transformation*
- *World transformation*
- *Viewer transformation*

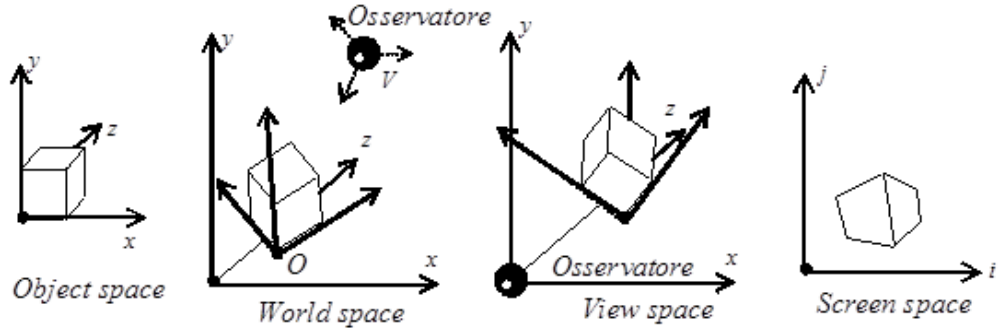


Figura 3.2: Nel ray-tracing, i raggi partono dall'osservatore fino ad intersecare gli oggetti del mondo

Il model space è il sistema di riferimento base dell'oggetto così come è composto dal modellatore. La prima trasformazione lavora su questo spazio e opera una prima trasformazione ruotando, traslando o scalando l'oggetto in base alla sua origine locale. Ad esempio, in un modello di un aereoplano, i movimenti di pitch, roll e yaw sono tutte rotazioni del sistema di riferimento locale.

La world transformation cambia il sistema di coordinate dal model space, dove i vertici sono definiti relativamente all'origine locale del modello, al world space, dove i vertici sono definiti relativamente all'origine di un sistema di coordinate comune a tutti gli oggetti della scena. In termini diversi, essa colloca l'oggetto nel mondo. La posizione di una entità, ad esempio, non è un parametro locale dell'oggetto.

La terza trasformazione porta ogni oggetto al sistema di riferimento dell'osservatore, ovvero trasforma gli oggetti in modo che l'osservatore sia collocato al centro del sistema di riferimento. In altre parole, la viewer transformation porta tutti gli oggetti "davanti" all'osservatore.

Per rappresentare i sistema di coordinate si usano le matrici. Ogni oggetto, quindi, avrà una matrice di orientamento, detta *object matrix*, che ne definirà la posizione, la scala e l'angolazione rispetto ai vertici del modello, definiti nel momento di progettazione e di produzione. In altri termini, tale matrice indica come l'oggetto, nel corso dell'evolversi della scena, venga ruotato, traslato o scalato.

Si noti come l'object matrix ***O*** sia una matrice 4x4 che contiene la matrice di rotazione e il vettore traslazione, in coerenza del sistema di coordinate omogeneo secondo questa struttura:

$$O = \begin{bmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

dove è stato R_{ij} sono gli elementi della matrice di rotazione, e T_{xyz} gli elementi del vettore traslazione. Pertanto, se **O** rappresenta l'object matrix, per ogni punto **P** da trasformare dell'oggetto si ha:

$$P_{local\_space} = P \cdot O = \begin{bmatrix} P_x & P_y & P_z & 1 \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

dove l'object matrix contiene le informazioni rispetto al proprio sistema di coordinate.
A questo punto otteniamo tutti i vettori trasformati nell'local space. Il passo successivo è la trasformazione di questi nel world space, ottenuta similmente, moltiplicando il vettore precedentemente ottenuto con la matrice del sistema world space, detta *world matrix*. Sfruttando le proprietà delle matrici, l'object matrix e la world matrix vengono concatenate in un'unica matrice. Si chiami **W** la world matrix ed **H** la matrice concatenata, definita come **H** = **O**·**W**, allora si ottiene:

$$P_{world\_space} = P \cdot H$$

A questo punto ogni vertice dell'oggetto è stato trasformato, quindi collocato, nel world space. L'ultimo passo è allineare il world space al view space (o *camera space*). Si procede analogamente, dopo aver definito **C** come la camera matrix:

$$P_{view\_space} = P_{world\_space} \cdot C$$

Tale passaggio rappresenta solitamente l'ultimo stadio di trasformazione degli oggetti della scena nell'ambito tridimensionale, in quanto tutta l'iterazione con l'utente e con la scena stessa avviene in un momento precedente delle pipeline.

3.2.2 Clipping e Viewing Frustum

Il termine *clipping* fa riferimento alle procedure che hanno lo scopo di eliminare tutti gli elementi della scena che non sono visibili e quindi non vanno sottoposti al renderer. In tale accezione, il clipping fa parte della famiglia dei metodi di *Visible Surface Determination* (VSD).

Nella fattispecie, il clipping 3D ha lo scopo di:

- Identificare i triangoli completamente non visibili, ovvero esterni al campo di vista, detto *Viewing Frustum.*
- Tagliare solamente l'area esterna al campo di vista nei triangoli parzialmente esterni al viewing frustum. Questa operazione può introdurre nuovi triangoli nella pipeline.

Entrambi sono passaggi fondamentali, anche se molto spesso la seconda fase è ignorata in quanto tutti i moderni processori video sono in gradi di svolgere autonomamente tale operazione. La prima fase, invece, è essenziale in quanto generalmente solo una piccola percentuale degli oggetti della scena è effettivamente inclusa nel viewing frustum, e tale discriminazione allegerisce la mole di informazioni spedita al processore grafico, diminuendo la richiesta di banda e diminuendo il numero di calcoli che la GPU dovrà sostenere.

L'individuazione degli elementi esterni al viewing frustum è spesso gerarchica, e in genere segue questo approccio:

- Controllo di visibilità a livello di oggetto
- Controllo di visibilità dei triangoli di un oggetto

Se un oggetto è esterno, allora sicuramente si possono ignorare tutti i triangoli che lo compongono. Il fatto di verificare singolarmente i triangoli, qualora l'oggetto risulti parzialmente interno, è altresì spesso ignorato e lasciato alla GPU. In genere vengono usate strutture di *partizionamento spaziale* per velocitare tale operazione. In seguito sarà presentato l'approccio scelto per il motore grafico sviluppato.

3.2.2.1 Viewing Frustum

Nella figura è visibile il *viewing frustum*, ovvero il volume dello spazio che contiene tutti gli oggetti visibili della scena, fuori dal quale ogni oggetto è non visibile. La forma del viewing frustum è una piramide la cui punta risiede alla posizione dell'osservatore, e tale forma è scelta

perché rappresenta l'esatto volume che sarebbe visibile ad una telecamera che sta guardando attraverso una finestra rettangolare – il display.

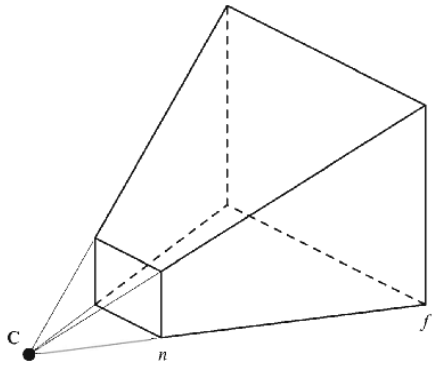


Figura 3.3: Il Viewing Frustum ha una forma piramidale e contiene gli oggetti visibili

Il viewing frustum è delimitato da *sei piani*, quattro dei quali corrispondono ai lati del display, e sono chiamati *left*, *right*, *bottom* e *top frustum plane*. I rimanenti due piani sono chiamati *near* e *far frustum plane*, e definiscono la minima e la massima distanza alla quale gli oggetti della scena sono visibili e considerati dall'osservatore – cioè dalla telecamera. Essi sono particolari in quanto, nel sistema di riferimento della telecamera, sono perpendicolari all'asse *Z*, cioè quello che entra nel display. Il viewing frustum, infatti, è *allineato* al sistema della telecamera che è quello nel quale la telecamera giace all'origine degli assi mentre l'asse *X* punta a destra, l'asse *Y* punta in alto e l'asse *Z* è perpendicolare al display. Nell'engine sviluppato, è entrante.

3.2.2.2 Field of View

Il *piano di proiezione* è un piano che è *perpendicolare alla direzione di vista* della telecamera, ovvero l'asse *Z*, e giace ad una distanza *e* dalla telecamera, dove i left e right frustum planes lo intersecano alle coordinate $x = -1$ e $x = 1$. La distanza *e*, detta *focal lenght* o *focus distance*, dipende dall'*angolo α di vista*, detto angolo di vista, o campo di vista, orizzontale. Le coordinate dei punti del piano di proiezione sono facilmente mappate nel viewport tramite moltiplicazione.

Normalmente l'uomo ha un campo di vista stimato in *60°*, ma nel campo della computer grafica si usa spesso optare per un valore più elevato, tipicamente da *60° a 90°*. La scelta nel progetto è stata *80°*.

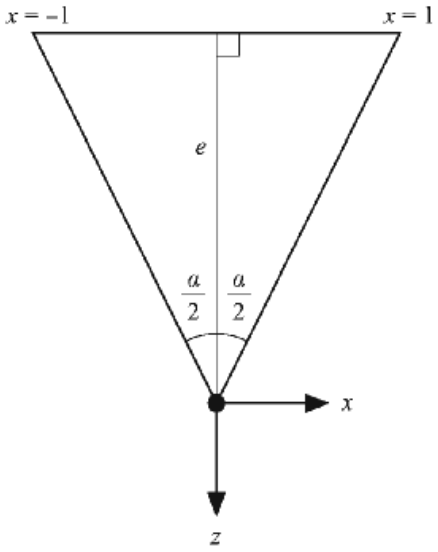


Figura 3.4: Il piano di proiezione giace ad una distanza e relativa all'angolo di vista ed interseca il left ed il right plane ale coordinate x=-1 e x=1

Per un dato angolo di vista, la *focus* dal piano di proiezione è data da una relazione trigonometrica che deriva dall'analisi dei triangoli. Sapendo che per quanto riguarda la dimensione orizzontale, il piano di proiezione può essere suddiviso in due triangoli rettangoli di cui è noto un cateto e l'angolo opposto, si ricava facilmente:

$$e = \frac{1}{\tan(\alpha / 2)}$$

L'*aspect ratio* di un display equivale alla sua altezza rapportata alla sua larghezza. Dal momento che generalmente nessun display è quadrato, il campo di vista verticale è diverso da quello orizzontale. I piani del frustum intersecano il piano di proiezione formando un rettangolo i cui lati giacciono a *y=±a (bottom* e *top) e x=±1 (left* e *right)*, dove *a* è l'aspect ratio.

Campi di vista più grandi equivalgono ad un parametro focus minore. Pertanto l'effetto zoom può essere ottenuto diminuendo l'angolo di vista, provocando un incremento del focus. Questo aspetto è facilmente implementabile negli engine 3D.

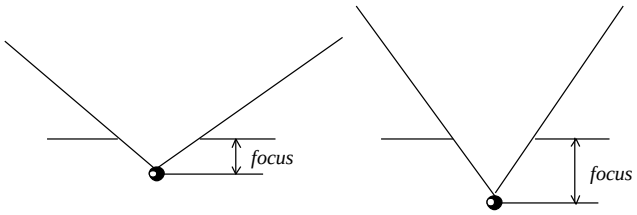


Figura 3.5: Il valore focus è la distanza dell'osservatore dal piano di proiezione ed è relativo alla risoluzione ed all'angolo di vista

3.2.2.3 Frustum Planes

I piani del viewing frustum sono mostrati in figura 3.6. Nella tabella sottostante sono presentate le formule per i piani. La conoscenza dei piani del frustum è importante anche per eseguire il frustum culling. Il modo in cui il frustum è usato per eseguire il frustum culling sarà illustrato in seguito.

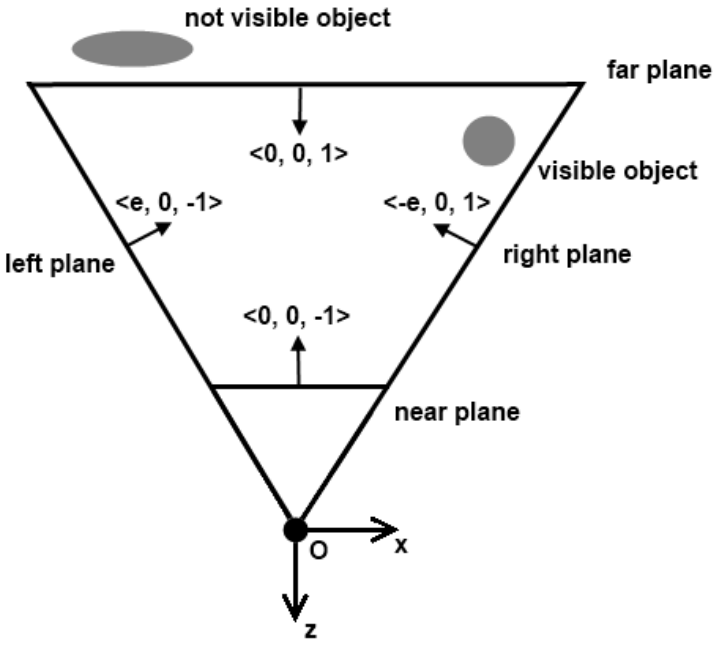


Figura 3.6: Rappresentazione del near, far, righg e left frustum planes

Plane	$\langle N, D \rangle$
Near	$\langle 0, 0, -1, -n \rangle$
Far	$\langle 0, 0, 1, f \rangle$
Left	$\langle \frac{e}{\sqrt{e^2+1}}, 0, -\frac{1}{\sqrt{e^2+1}}, 0 \rangle$
Right	$\langle -\frac{e}{\sqrt{e^2+1}}, 0, -\frac{1}{\sqrt{e^2+1}}, 0 \rangle$
Bottom	$\langle 0, \frac{e}{\sqrt{e^2+a^2}}, -\frac{a}{\sqrt{e^2+a^2}}, 0 \rangle$
Top	$\langle 0, -\frac{e}{\sqrt{e^2+a^2}}, -\frac{a}{\sqrt{e^2+a^2}}, 0 \rangle$

Tabella 3.1: Definizione dei piani del viewing frustum

3.3 Rappresentazione Prospettica

La rappresentazione prospettica permette di trasformare una figura tridimensionale in una bidimensionale, proiettandola nel *piano di proiezione*. Essa crea immagini tanto più piccole quanto più distante è l'oggetto da proiettare, un effetto facilmente osservabile anche nella realtà, permettendo di rappresentare con buona approssimazione un'immagine di una scena reale.

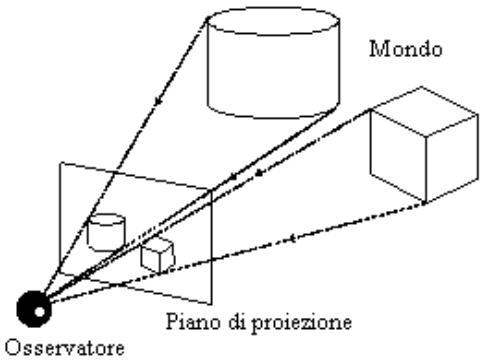
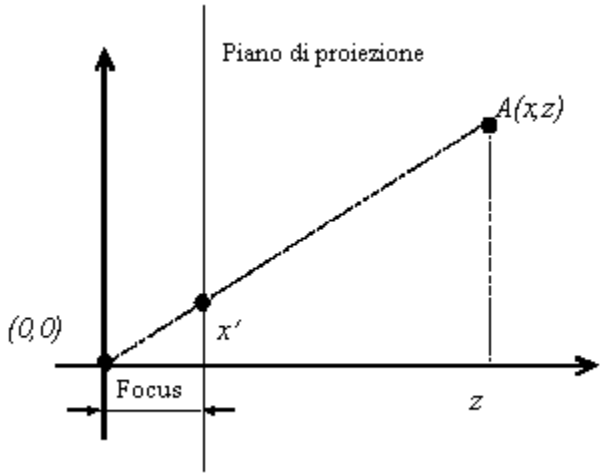


Figura 3.7: Gli oggetti vengono proiettati sul piano di proiezione

È semplice ricavare le formule per la proiezione prospettica se si immagina la proiezione in questo modo, supponendo di dover proiettare sul display un vertice 3D A:



Consideriamo dapprima il caso di una sola dimensione. L'osservatore è posizionato all'origine del sistema di riferimento. Si ribadisce che la distanza fra l'osservatore e il piano di proiezione è la *focus distance*, la quale è già stata calcolata in funzione del campo di vista e della risoluzione del display. Lo scopo è trovare i valori dell'intersezione fra il raggio avente come origine il punto *A* e come destinazione l'osservatore, e il piano di proiezione. Sfruttando le proprietà di similitudine dei triangoli si ottiene:

$$\frac{x'}{e} = \frac{A_x}{A_z} \Rightarrow x' = \frac{A_x e}{A_z}$$

3.3.1 Proiezione Prospettica

Un proiezione prospettica che mappi le coordinate *x* e *y* nel punto corretto del piano di proiezione preservando le informazioni di profondità è ottenuta mappando il view frustum in un cubo, come mostrato in figura 3.9. Il sistema di coordinate del cubo si chiama *homogeneous clip space*. L'osservatore giace al centro del cubo i cui lato hanno lunghezza unitaria. Qualsiasi punto esterno al cubo non sarà visibile nel piano di proiezione.

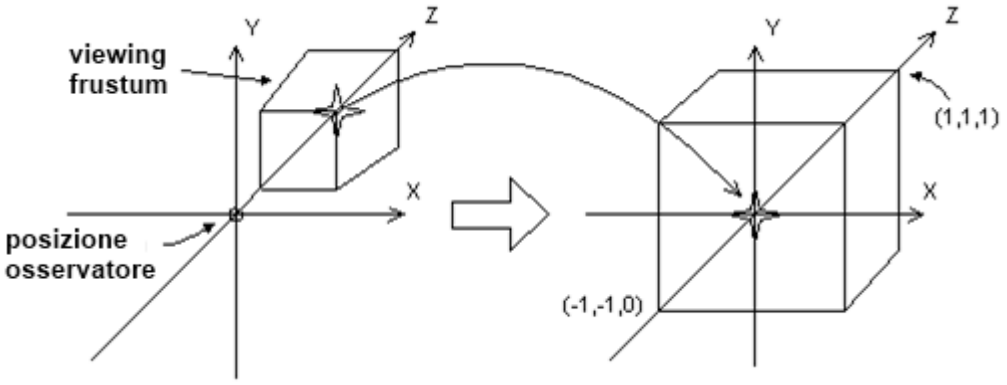


Figura 3.9: Mappatura da view space a homogeous clip space

Si consideri il piano di proiezione a distanza *e* dall'osservatore, e sia **P** = <P_x, P_y, P_z, 1> un punto dentro al frustum. Il vettore proiettato ha coordinate:

$$\boldsymbol{p} = \langle \frac{e}{P_z} P_x, \frac{e}{P_z} P_y \rangle$$

che devono essere entrambe comprese nell'intervallo [-1..1]. Una funzione lineare del tipo

$$z' = \frac{A}{z} + B$$

con

$$A = \frac{2nf}{f-n} \quad e \quad B = \frac{f+n}{f-n}$$

dove f è il valore di profondità del far plane e n è il valore di profondità del near plane, mappa il valore della z in modo tale che il near plane corrisponda a 0 e il far plane a 1. La matrice calcolata dalla libreria di supporto delle DirectX diventa:

$$\begin{bmatrix} cot\frac{fovY}{2} & 0 & 0 & 0 \\ 0 & cot\frac{fovY}{2}/a & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & \frac{nf}{f-n} & 0 \end{bmatrix}$$

Questa matrice ritorna un vettore che preserva la coordinate z nella quarta componente. Questo valore è indispensabile alla GPU in quanto si dimostra che per interpolare linearmente i valori dei vertici lungo lo screen space (scanline process) è necessario disporre dell'inverso della z. Il vettore ottenuto moltiplicando **P** per questa matrice è equivalente al vettore nello screen space dopo averlo diviso per la componente *w*.

3.3.2 Rasterizzazione e interpolazione

Questa è l'ultima macro-fase in cui l'immagine è finalmente scritta nel frame buffer. La rasterizzazione è il processo di conversione di una entità grafica descritta da vettori in una immagine raster, ovvero una immagine bidimensionale composta di punti (pixel). L'interpolazione è la componente della rasterizzazione che si occupa di interpolare i valori assegnati ai vertici su ogni pixel prodotto.

Nella figura 3.10 è presentata la tipica pipeline di una moderna GPU. I blocchi *vertex data* e *primitive data* rappresentano l'informazione descrittiva delle singole mesh. Nella forma più semplice, vertex data rappresenta l'insieme dei vertici che compongono la mesh, mentre primitive data contiene gli indici dei vertici che compongono ogni faccia. Si evidenziano poi due componenti fondamentali: il *Vertex Shader* e il *Pixel Shader*.

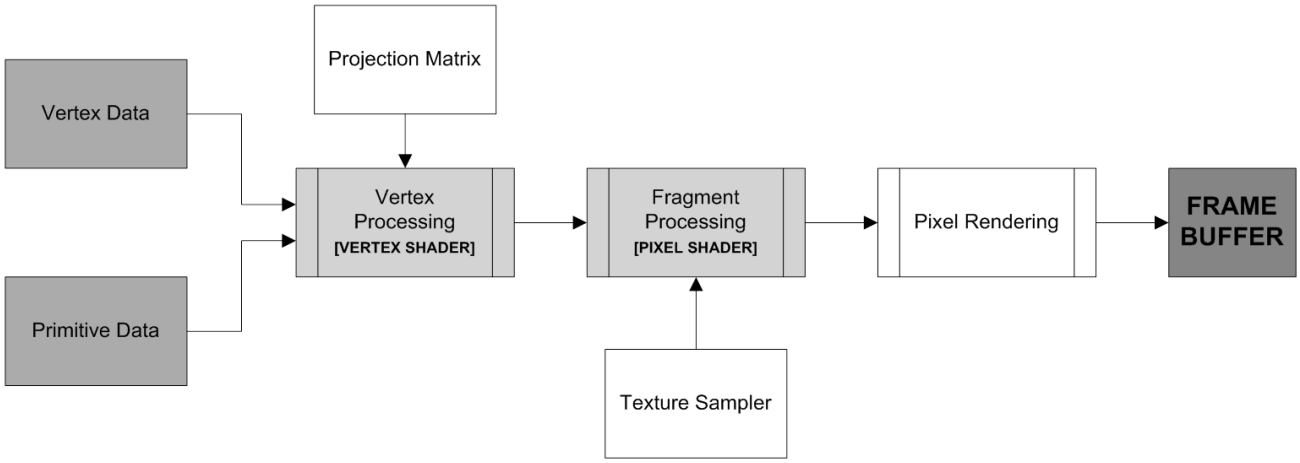


Figura 3.10: Pipeline essenziale di una GPU

Gli shader sono un set di istruzioni software (programmi) usati per calcolare effetti e parametri nel processo di rendering. Essi sono scritti sfruttando una API apposita ed eseguiti in maniera fortemente parallelizzata nella GPU. Hanno lo scopo di personalizzare i calcoli eseguiti dalla GPU per ottenere l'implementazione desiderata di tutti gli algoritmi di rendering. Nello schema sono evidenziati due tipi di shader, sebbene le ultime generazioni di GPU accettino un terzo stadio, che non è stato sfruttato nell'engine sviluppato.

- **Vertex shader**: è la prima parte della pipeline di rendering vero e proprio, e si occupa di trasformare i vertici per poi eseguire la rasterizzazione. In questo passaggio i vertici vengono proiettati, passando quindi allo spazio bidimensionale, e i parametri assegnati ad essi vengono interpolati lungo i fragment prodotti. Un vertex shader è eseguito per ogni vertice del set di vertici richiesti, in maniera parallela. In questo modo più primitive possono essere renderizzare contemporaneamente. I fragment prodotti vengono passati al pixel shader per eseguire le operaioni per-pixel.

- **Pixel Shader**: è l'ultimo stadio programmabile della pipeline, e il suo scopo è elaborare i singoli fragment per determiarne il colore finale. Come si evince dalla figura 3.10, il pixel shader può avvalersi dell'uso di uno o più samplers per ricavare delle informazioni dalle textures assegnate alla primitiva renderizzata. Il pixel shader riceve i parametri dei vertici interpolati, fra i quali le coordinate delle texture, e usa

queste come parametro per i samplers, i quali accedono a textures definite prima dell'esecuzione della pipeline secondo modalità programmate a livello di codice. Il colore della luce diffusa, ad esempio, è determinato in questo stadio campionando la *diffuse texture* della superficie che si sta renderizzando. Si fa notare che in larga parte dei sistemi il test dello ZBuffer è eseguito *dopo* l'esecuzione del pixel shader.

4. Illuminazione

Una volta che la geometria è stata rasterizzata producendo i fragment delle superfici, è necessario calcolare l'ammontare di luce che incide in ognuno di essi. Gli algoritmi di illuminazione usati sono uno degli aspetti più complicati e rilevanti per quanto riguarda la resa grafica finale, nonchè la parte che impegna maggiormente il rendering. Non esiste una soluzione (algoritmo) ottimale, ma piuttosto una serie di approssimazioni che spesse volte vengono fatte lavorare in sinergia per ottenere la resa desiderata. Il calcolo della luce riflessa (intensità e colore) di un fragment è detto ***shading***.

4.1 Equazione di Rendering

Ogni fragment si porta a presso una serie di informazioni, fra cui la posizione, la normale, e le proprietà del materiale che lo compongono. La posizione e la normale sono sufficienti per determinare l'ammontare di radiazione luminosa incidente. In seguito, le proprietà del materiale vengono combinate con questa per ottenere il colore finale. Ogni modello di shading dovrebbe basarsi sul vero comportamento fisico della luce, che obbliga un richiamo alla radiometria, ovvero lo studio della misura della radiazione elettromagnetica, qui trattata solo concettualmente.

La *potenza radiante* o *flusso radiante* è la totalità dell'energia irradiata da un corpo in tutte le direzioni nell'unità di tempo e si misura in watts [W]. La potenza emessa da una sorgente di luce per unità di area è detta *emittanza* (radiant emittance) e si misura in watts per metro quadrato [$W \cdot m^{-2}$], ed indica la capacità di un corpo di emettere energia. Quando la potenza è ricevuta si chiama invece *irraggiamento* (irradiance). L'intensità radiante è la potenza radiante per unità di angolo solido e si misura in [$W \cdot sr^{-1}$]. La *radianza* è la potenza radiante emessa (o attraversata) da una superficie all'interno di un angolo solido, misurata in [$W \cdot m^{-2} \cdot sr^{-1}$]

L'*equazione di rendering* permette di calcolare la radianza uscente da un elemento di superficie con posizione x e direzione $\vec{w}$:

$$L_o(x,\vec{w}) = L_e(x,\vec{w}) + \int_\Omega f_r(x,\vec{w}',\vec{w}) L_i(x,\vec{w}')\,(\vec{w}' \cdot \vec{n}) d\vec{w}'$$

dove L_o è la luce uscente (percepita), L_e è la luce emessa e l'integrale rappresenta la luce riflessa, ottenuta moltiplicando la luce incidente L_i nel punto con il coefficiente di riflessione f_r e l'angolo di arrivo.

Purtroppo non esiste alcuna soluzione traducibile in algoritmo, e tutti i modelli utilizzati nella computer grafica sono delle soluzioni semplifici di questa equazione. Il problema fondamentale è il calcolo di L_i, poichè l'integrale nella semisfera centrata in x presuppone la conoscenza di L_i per ogni angolo, ovvero un valore che non è calcolato localmente nell'elemento di superficie ma riguarda invece l'interazione di tutte le sorgenti di luce e tutti gli elementi della scena. Gli altri parametri sono caratteristiche del materiale e sono noti o comunque facilmente modellabili.
La radiazione incidente è quindi distinguibile in:

- *Illuminazione diretta* (direct illumination), ovvero la radiazione proveniente da una sorgente di luce.
- *Illuminazione indiretta* (indirect illumination), ovvero la radiazione riflessa da altre superfici.

Tale problema porta alla suddivisione degli algoritmi di shading in due famiglie:

- Illuminazione diretta
- Illuminazione globale (global illumination)

Per risolvere il problema dell'illuminazione diretta esistono vari algoritmi, quasi tutti soluzioni semplici ma convincenti dell'equazione di render. Nel motore grafico è usato il modello *Phong Shading*, che è diventato lo standard *de facto*. Nell'implementazione ci si avvale inoltre di una serie di tecniche, in seguito discusse, per incrementare il realismo del rendering. Il problema dell'illuminazione globale, invece, è un tema aperto per il quale

esistono varie proposte ma nessuna soluzione de facto. Sebbene il calcolo delle ombre rientri negli algoritmi di illuminazione globale, in quanto lega la geometria della scena alla luce ricevuta da una superficie, in realtà si considerano facendi parte di questa categoria solo gli algoritmi che modellano l'*inter-riflessione*, ovvero l'effetto della luce riflessa o assorbita dalle superfici. Algoritmi come il ray-tracing risolvono efficacement questo problema, mentre per quanto riguarda gli approcci real-time esistono algoritmi quali il *photon mapping, beam tracing, e ambient occlusion*. Comunque il problema dell'illuminazione globale non era primario per lo sviluppo del motore, quindi le tecniche più raffinate, come il photon mapping, non sono state considerate.

4.2 Superfici Lambertiane

Il modello usato nel motore grafico semplifica l'equazione di rendering trattando le superfici come *diffusori ideali* (semplificando f_r) e considerando solo l'illuminazione diretta (semplificando L_i). Tuttavia alcune tecniche sono state introdotte per simulare in parte l'illuminazione globale, e saranno illustrate in seguito. Un diffusore ideale è una superficie che manifesta la proprietà di riflessione diffusa ideale, per la quale la luce incidente in un punto della superficie viene riflessa in infinite direzioni casuali all'interno della semisfera centrata nel punto, ottenendo una diffusione uniforme del colore della luce riflessa. Gran parte dei materiali sono approssimativamente dei diffusori ideali.

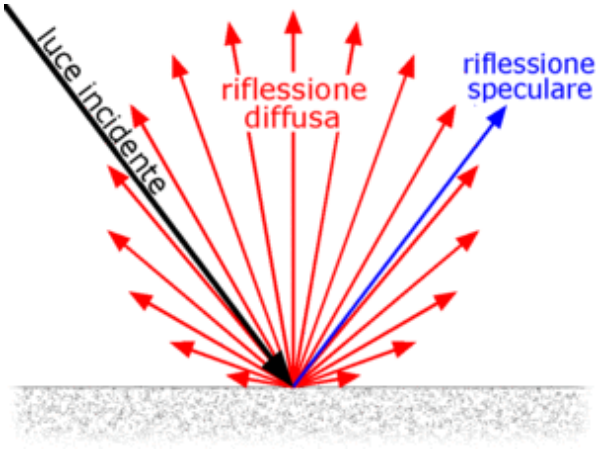


Figura 4.1: La luce incidente si scompone in due componenti: diffusa e speculare

Tali superfici sono dette *superfici lambertiane* in quanto seguono la legge di Lambert, la quale afferma che: *nei diffusori ideali l'intensità radiante osservata è proporzionale al coseno dell'angolo α fra l'osservatore e la normale della superficie.*

Questa legge ha come conseguenza il fatto che una superficie ha la stessa radianza a prescindere dall'angolazione dell'osservatore. In figura, l'elemento di superficie *dA* è interessato da una certa potenza radiante W.

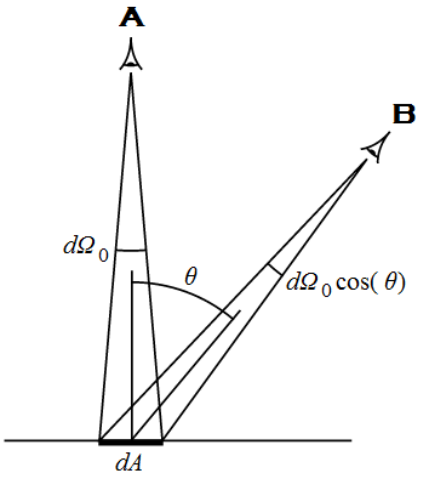


Figura 4.2: L'angolo solido dell'osservatore A è maggiore dell'angolo solido dell'osservatore B

Ogni osservatore vede la scena da un'apertura di superficie *dB*. L'elemento *dA* sottende con l'osservatore A un angolo solido pari a $d\Omega_0$ e pertanto si misura una radianza di:

$$I = \frac{W}{d\Omega_0 dB}$$

Nel caso dell'osservatore B, l'elemento dA sottende un angolo minore, ridotto di un fattore cosΩ, ma anche l'angolo solido da cui vede l'osservatore è ridotto dello stesso fattore, pertanto la radianza misurata vale:

$$I = \frac{W cos\Omega}{d\Omega_0 dB cos\Omega} = \frac{W}{d\Omega_0 dB}$$

La radianza è massima in prossimità della normale della superficie.

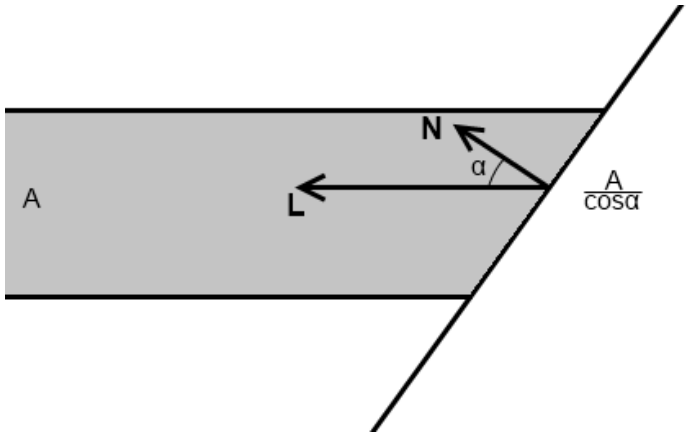


Figura 4.3: il fascio di luce A investe un'area di superficie proporzionale al coseno dell'angolo formato con la normale della superficie

Una conseguenza è che una sorgente di luce imprime una radianza proporzionale al coseno angolo che essa forma con la normale della superficie. Più l'angolo diminuisce, più l'angolo solito associato è maggiore e quindi aumenta la radiazione ricevuta.
In termini pratici, il valore del coseno dell'angolo è facilmente determinato dal prodotto scalare fra la normale della superficie **N** e la direzione della luce incidente **L**. Valori negativi indicano che la superficie non è colpita dalla sorgente di luce. Se il colore di riflessione diffusa della superficie è *D* e l'intensità della luce ricevuta è *C*, allora il colore diffuso riflesso è

$$K = DCmax\{\boldsymbol{N} \cdot \boldsymbol{L}\}$$

4.3 Riflessione speculare

Molti materiali esibiscono l'effetto di riflessione speculare, ovvero la riflessione della radiazione luminosa da parte di una superficie verso un determinato angolo diverso dall'angolo di arrivo. Tale fenomeno è descritto dalla legge di riflessione, la quale afferma che la direzione della luce in arrivo e la direzione della luce in uscita formano lo stesso angolo rispetto alla normale della superficie.

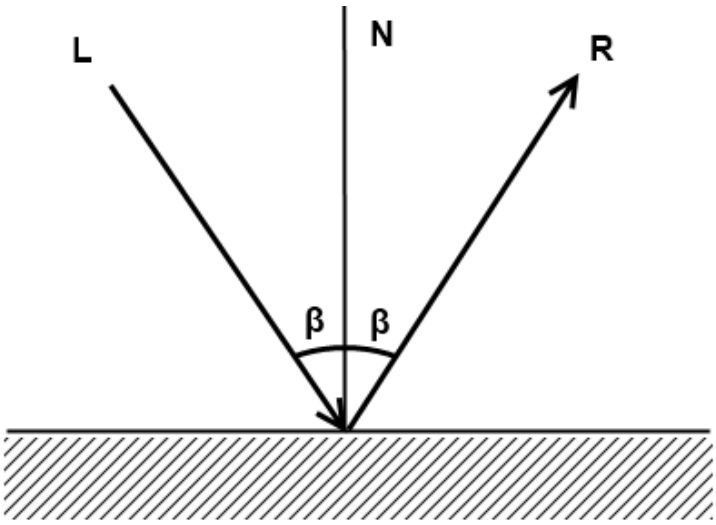


Figura 4.4: il raggio di riflessione speculare esce dalla superficie formando il medesimo angolo rispetto alla normale

La formula per la direzione di riflessione **R** è

$$\boldsymbol{R} = 2(\boldsymbol{L} \cdot \boldsymbol{N})b - \boldsymbol{L}$$

Tutti i materiali la cui superficie sia priva di micro irregolarità di dimensione paragonabile alla lunghezza d'onda della luce possono esibire l'effetto di riflessione speculare. Ad esempio i liquidi, la cui superficie è perfettamente piana, esibiscono chiaramente questo effetto (si pensi al paesaggio riflesso sulle superficie dell'acqua). Varibilmente in intensità, anche i metalli sono materiali interessati da questo fenomeno.

4.4 Sorgenti di luce

Il colore calcolato per ogni punto di una superficie è determinato dal contributo di tutte le luci che interessano la superficie. Di seguito sono elencati i tipi di luce comunemente supportati da tutti i motori grafici, incluso il motore sviluppato.

4.4.1 Ambient light

Non si tratta di una vera e propria sorgente di luce, ma piuttosto di un termine, detto di illuminazione ambientale, che è introdotto per simulare l'effetto di illuminazione globale di inter-riflessione, che, come detto, non è facilmente calcolabile. La luce ambientale proviene da ogni direzione con la medesima intensità e colore, impedendo l'esistenza di zone totalmente buie. Tuttavia è possibile rendere dipendente dalla posizione nella scena tale parametro.

4.4.2 Directional Light

Le directional light sono sorgenti di luce che irradiano luce in una singola direzione da una sorgente infinitamente lontana. Non hanno una posizione spaziale ma sono caratterizzate solamente dalla direzione degli infiniti raggi che emanano. Sono usate per modellare oggetti quali il sole, i cui raggi possono essere considerati paralleli.

4.4.3 Point lights

Le point lights sono sorgenti di luce che irradiano luce in ogni direzione da un punto dello spazio, producendo una diffusione sferica che diminuisce di intensità allontanandosi dal centro. L'intensità è governata dalla legge dell'inverso del quadrato, che approssima fedelmente il comportamento fisico. Tuttavia i motori grafici mettono a disposizione tre parametri per la definizione della funzione dell'intensità, rappresentata da un polinomio di secondo grado.
Si supponga che una point light sia collocata nel punto **P**. L'intensità *C* della luce ricevuta da un punto nello spazio di posizione **Q** è data da:

$$C = \frac{1}{k_c + k_l d + k_q d^2} C_0$$

dove C_0 è il colore della luce, d è la distanza fra luce e punto calcolata come $\|\boldsymbol{P} - \boldsymbol{Q}\|$, e le costanti k_c, k_l, k_q sono dette costanti di attenuazione lineare costante, lineare e quadratica.

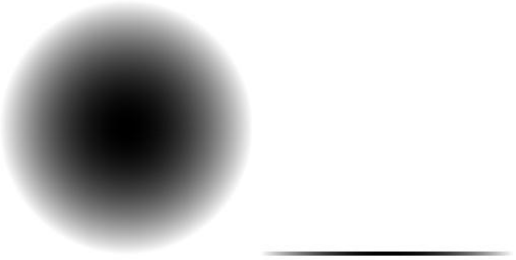


Figura 4.5: esempio di texture 2D e 1D usate per determinare il fallout di una sorgente di luce

4.4.4 Spot Lights

Una spot light è come una point light ma con una direzione precisa di irraggiamento. Può essere pensata come alla luce prodotta da una torcia. La funzione di intensità è composta dalla stessa componente delle point lights più un valore detto effetto spot-light. Si supponga che una spot light sia stata posta alla posizione **P** e che abbia una direzione **R**.

L'intensità *C* della luce ricevuta da un punto nello spazio di posizione **Q** è data da:

$$C = \frac{\max\{-\boldsymbol{R} \cdot \boldsymbol{L}, 0\}^p}{k_c + k_l + k_q d^2} C_0$$

dove **L** è il vettore normalizzato:

$$L = \frac{\boldsymbol{P} - \boldsymbol{Q}}{\|\boldsymbol{P} - \boldsymbol{Q}\|}$$

L'esponente *p* controlla quando acuto è l'angolo solido sotteso dalla luce.

4.5 Phong Shading

A questo punto è possibile descrivere il modello usato nel motore grafico sviluppato, chiamato *Phong Shading*. In questo modello, la normale è interpolata a partire dai vertici dei triangoli, rendendola disponibile per ogni fragment. A livello di singolo fragment, poi, viene

calcolata l'intensità di luce ricevuta (*per-pixel shading*). Nel modello di Phong l'intensità di luce è calcolata con la seguente formula:

$$K = K_{emission} + K_{diffuse} + K_{specular}$$

La componente $K_{emission}$ rappresenta l'eventuale caratteristica intrinseca del materiale di emettere luce propria. Ovviamente questa luce non influirà sulla scena.
La componente $K_{diffuse}$ rappresenta la luce riflessa dal materiale in ogni direzione, secondo quanto illustrato precedentemente. É necessario considerare il contributo di ogni luce, ottenendo la funzione

$$K_{diffuse} = DA + D\sum_{i=1}^{n} C_i \max\{\boldsymbol{N} \cdot \boldsymbol{L}_i, 0\}$$

La componente $K_{specular}$ modella la riflessione speculare della superficie. Contrariamente al caso precedente, in questo caso è richiesta la posizione dell'osservatore. Nella figura 4.6, è illuminato un elemento di superficie alla posizione **O**, dove **N** è la normale della superficie, **L** il raggio di luce entrante, **R** il raggio speculare e **V** la direzione verso l'osservatore.

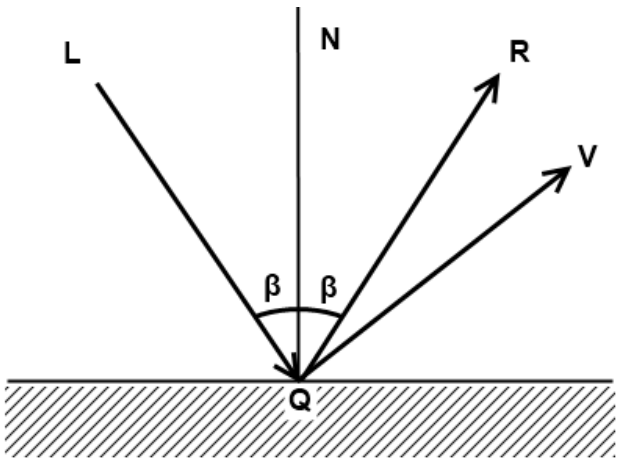


Figura 4.6: lo specular highlight dipende dall'angolo fra il vettore R (riflessione speculare) e il vettore V (posizione dell'osservatore)

Il modello di phong propone la seguente formula:

$$SCmax\{\boldsymbol{R}\cdot\boldsymbol{V},0\}^{m}(\boldsymbol{N}\cdot\boldsymbol{L}>0)$$

dove S è il colore di riflessione speculare, C è l'intensità della luce incidente e m è il l'esponente speculare (*specular exponent* o *shihiness*). Questo valore controlla la forma del riflesso speculare (*specular highlight*). Un valore basso produce un riflesso molto ampio, mentre un valore alto produce un riflesso molto piccolo e concentrato.

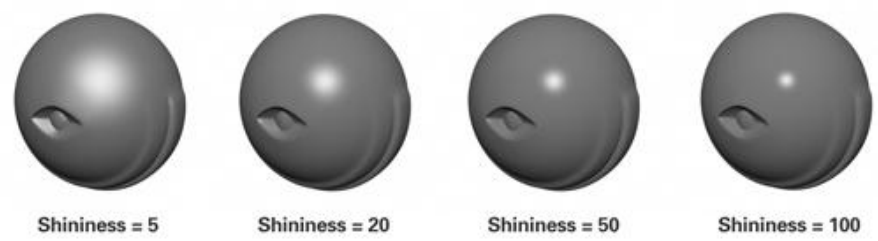


Figura 4.7: esempio di specular highlight in funzione dell'esponente

Tuttavia esiste una formulazione differente che trova maggior impiego pratico, detta B*linn–Phong shading model*. In questo caso viene introdotto l'*halfway vector*, chiamato **H** nella figura 4.8, che rappresenta un vettore che risiede a metà fra **V** e **L**. Lo specular highlight sarà tanto maggiore quanto più il vettore **H** punta al vettore **N.**

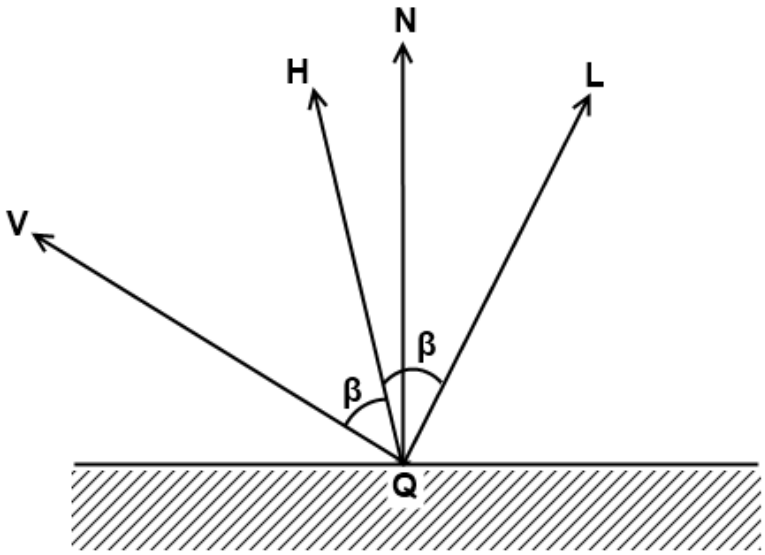


Figura 4.8: il vettore H è a metà fra V (osservatore) e L (sorgente luce), e l'angolo che forma con la normale determina l'intensità della riflessione speculare

Per introdurre questo modello nella formula, si sostituisce $\boldsymbol{R} \cdot \boldsymbol{V}$ con $\boldsymbol{N} \cdot \boldsymbol{H}$, dove

$$\boldsymbol{H} = \frac{\boldsymbol{L} + \boldsymbol{V}}{\|\boldsymbol{L} + \boldsymbol{V}\|}$$

Infine si ottiene la formula generale

$$K_{specular} = S \sum_{i=1}^{n} C_i \max\{\boldsymbol{N} \cdot \boldsymbol{H}_i, 0\}^m \, (\boldsymbol{N} \cdot \boldsymbol{L}_i > 0)$$

4.6 Texture Mapping

Nell'ambito del rendering, la superficie è una istanza geometria piana, generalmente un triangolo, che descrive una parte di un oggetto. Una o più *texture maps* sono applicate alla superficie per descriverne le proprietà a livello dei singoli punti. Ad ogni punto della superficie, determinato durante la fase di rasterizzazione, è associato un valore della texture

map (*texel*) che ne descrive le proprietà. Nel caso più semplice, una texture map è usata per modulare il colore di riflessione diffusa.

Sia T il valore campionato da una texture per un punto della superficie. Usando questo colore per modulare il colore di riflessione diffusa si ottiene la formula

$$K_{diffuse} = DTA + DT \sum_{i=1}^{n} C_i \max\{\boldsymbol{N} \cdot \boldsymbol{L}_i, 0\}$$

Il valore campionato dalla texture map è determinato dalle coordinate della texture applicate ad un oggetto. Ogni triangolo della mesh conserva nei vertici le coordinate di una o più texture, le quali vengono interpolate (prospetticamente) durante la renderizzazione, producento una coppia di valori (nel caso comune di una texture 2D) per ogni fragment.
Questa coppia di valori viene usata dal sampler per ricavare un valore dalla texture in genere avvantaggiandosi di qualche filtraggio per aumentarne la qualità e ovviare al problema dell'aliasing. Infatti la risoluzione della texture map in pratica non combacia mai con la risoluzione della superficie renderizzata. Se un modello si avvicina alla telecamera, la risoluzione della sue superfici diventa molto più alta di quella delle texture assegnate. Per risolvere questo problema, si ricorre alla tecnica del *bilinear filtering*. In questo caso, più elementi della texture vengono pesati secondo una maschera per ricavare un valore medio, sfruttando l'interpolazione bilineare (in due dimensioni). Nel caso contrario, quando il rapporto delle risoluzione si inverte, si usa la tecnica del *mip mapping*, in ad ogni texture è associata una lista di texture precalcolate di dimensioni minori. Questa tecnica è molto conveniente poichè è veloce e richiede poca memoria. L'unione delle due tecniche porta al *trilinear filtering*. Tutti questi filtri sono nativamente supportati da tutte le librerie grafiche commerciali.
Normalmente nei motori grafici, per incrementare il dettaglio e la verosimilità dei materiali, ad ognuno di essi sono assegnate più texture maps, che ne descrivono le caratteristiche fisiche. In seguito sarà illustrato l'approccio scelto nel motore sviluppato.

4.7 Modelli Fisici di Riflessione

Il modello *Phong Shading* rappresenta certamente l'opzione più usata in qualsiasi motore grafico commerciale, specialmente per videogiochi, dove la qualità grafica è importante ma dove i limiti imposti dal frame rate non permettono di usare algoritmi più sofisticati. Tuttavia esistono approcci più fedeli al comportamento fisico della luce, basati sul calcolo della *bidirectional reflectance distribution function* (BRDF). Essa permette di descrivere il modo in cui l'energia radiante contenuta in un raggio di luce venga distrubuita al momento di collisione con la superficie. Parte dell'energia è assorbita dalla superficie, parte può passare attraverso e parte è riflessa. La BRDF è una funzione a quattro variabili che prende in ingresso la direzione della luce in ingresso L e la direzione di riflessione R definendo l'ammontare di luce riflessa in tale direzione.

Modelli come *Cook-Torrance Illumination* e *Fresnel Factor* definiscono la BRDF di un materiale secondo alcune precise osservazioni fisiche concernenti la struttura microscopica della superficie. In pratica, tuttavia, è stato scelto di non considerarli perchè a scapito di un forte aumento di richieste computazionali, non producono alcun evidente miglioramento rispetto all'approssimazione del Phong shading, specie se affiancato ad un set di texture maps che ne aumentano il dettaglio.

5. Struttura Engine

L'engine è un complesso sistema di algoritmi che permette ad uno o più utenti di interagire in tempo reale con un mondo virtuale rappresentato in un display. I componenti principali di un engine sono molti e tali che ognuno meriterebbe una discussione dedicata, e includono un motore di rendering (renderer), un motore fisico, un sistema di animazioni, intelligenza artificiale, un sistema di networking, un sistema audio, un sistema di input. Durante la realizzazione dell'engine, gli sforzi si sono concentrati sullo sviluppo del renderer, tuttavia anche un semplice motore fisico e un sistema di intelligenza artificiale sono stati implementati. L'architettura è tale per cui tutti i componenti possano semplicemente comunicare l'uno con l'altro, semplificando il lavoro del programmatore e aumentando la scalabilità

5.1 Struttura

Il concetto base dell'engine è la scena, che rappresenta ciò che compone il mondo visualizzato e con cui è possibile interagire. La scena è solo concettuale, in quanto il mondo è materialmente composto da una collezione di elementi detti entità. Ad esempio se la scena è un stanza, le entità che la comporranno saranno le pareti, gli oggetti d'arredo, le luci presenti ed eventualmente le persone che ci sono dentro. Le luci meritano un discorso a parte, poichè esse influiscono il rendering finale, ma non si vedono. Infatti una lampada è composta da un elemento (mesh) che rappresenta il modello di lampada, più una sorgente di luce, posta in prossimità della lampadina.

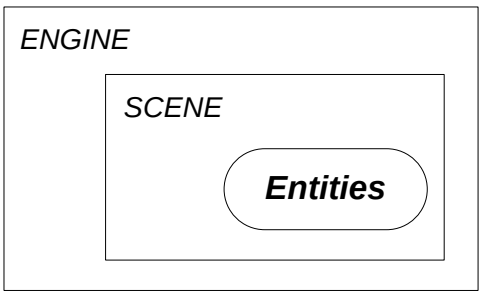


Figura 5.1: l'Engine contiene una Scena, composta da una lista di Entità

Ogni entità è composta da alcune variabili generali (fra cui il nome, il tipo e il padre) e da tre componenti individualmente assegnati al fine di caratterizzarla completamente. Questi componenti vengono detti render, fisico e logico e vengono assegnati ad una entità in un punto detto socket. D'ora in avanti con il termine socket si farà riferimento alla sua istanza.

- *RENDER SOCKET* → risponde alla domanda "*cosa si vede*". É la mesh di triangoli e materiali che determinano il modello 3D di un elemento.
- *PHYSICS SOCKET* → risponde alla domanda "*come interagisce*". É un modello fisico molto semplificato rispetto alla complessità della mesh, che descrive comportamento fisico della mesh, e include parametri quali la posizione, orientamento, peso, etc...
- *LOGIC SOCKET* → risponde alla domanda "*cosa fa*". É un modello logico che descrive il comportamento logico/pensante dell'entità. Ad esempio, se si tratta di un ascensore, il modello si occupa di muoverlo ai diversi piani in relazione ai tasti premuti. Se si tratta di una persona, ci sarà un modello logico che ne descriverà il comportamento.

All'interno di ogni socket è presente un puntatore all'entità padre, che garantisce la possibilità ad ogni socket di poter comunicare con gli altri. Non è obbligatorio che tutte e tre le componenti vengano assegnate. Il socket è detto nullo qualora non venga assegnato. Ad esempio se nella scena si vuole introdurre una entità regola di gioco, questa avrà solamente il socket logico, mentre una entità sedia ne è sprovvista (una sedia tutt'al più subisce ma non agisce). Un mobile in una stanza avrà nel socket render la mesh di un determinato mobile (disegnato da un modellatore), nel socket fisico un modello di tipo parallelepipedo e non avrà alcun socket logico. Un'entità persona, infine, avrà tutte e tre i socket assegnati.

La seguente figura illustra quanto fin'ora esposto.

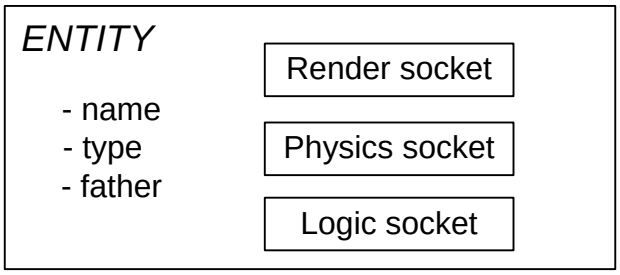


Figura 5.2: ogni entità dispone di tre componenti fondamentali (socket): render, fisico e logico

A questo punto si ha un database di entità, ognuna composta al più da tre componenti, che definisce l'intera scena. Per gestire i tre socket (di tutte le entità) sono presenti tre componenti nell'engine detti modulo render, modulo fisico e modulo logico.

Ogni modulo è logicamente associato ad un socket, pertanto esso si occupa solamente di un socket di ogni entità, potendo però richiedere alcune informazioni agli altri due sockets. Un modulo, inoltre, non solo gestisce il relativo socket, ma si occupa anche dell'assegnazione. Le operazioni di assegnazione e gestione vengono dette rispettivamente loading e update.

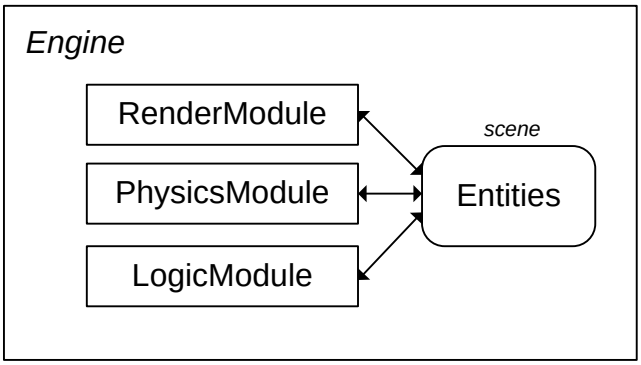


Figura 5.3: i tre moduli dell'engine interagiscono con le entità attraverso i sockets

Il modulo Renderer si occupa di visualizzare la scena, ovvero di renderizzare tutte le entità che la compongono. In altre parole, è il modulo responsabile di ciò che si vede nel display. É anche il modulo che assegna una rappresentazione visiva ad una entità. Per far questo, il modulo mantiene un database interno di modelli (mesh) e pertanto è anche responsabile del loro caricamento da files.

Il modulo Physics gestisce l'interazione dei modelli fisici, sottoposti a forze, delle entità della scena. In altre parole, governa l'interazione spaziale delle varie entità controllando quando

esse collidano ed evitando situazioni impossibili (ad esempio che una palla passi attraverso un muro). Questo modulo è anche responsabile dell'istanziazione del modello fisico desiderato.

Il modulo Logic gestisce le regole e l'intelligenza artificiale delle single entità. Ogni scelta, o meglio ogni evento, determinato da una entità è scatenato dalla sua logica. Il modulo logico è ciò che rende dinamica e viva la scena, senza il quale l'engine non potrebbe che rappresentare una scena totalmente statica, salvo qualche eventuale forza inizialmente applicata ad alcune entità. Come negli altri casi, anche questo modulo è responsabile dell'assegnazione della logica desiderata ad una entità.

5.2 Ciclo Principale di Update

L'esecuzione dell'engine si distingue sommariamente in cinque fasi:

1. *Inizializzazione del sistema*. In questa fase vengolo allocale le risorse di sistema necessarie all'esecuzione dell'engine.
2. *Loading della scena*. In questa fase vengono caricate in memoria tutte le entità e ad ognuna assegnati i rispettivi sockets.
3. *Esecuzione del ciclo principale di update* (real-time). Questa è la fase cruciale poichè viene ripetuta ciclicamente ad ogni frame, che rappresenta l'unità atomica temporale. In una condizione di 60fps, cioè avviene ogni 17ms circa.
4. *Eliminazione contenuti*. Le risorse vengono rilasciate e la memoria liberata.
5. *Chiusura del sistema*

La fase (3) è la più delicata, poichè dev'essere eseguita nel minor tempo possibile, nell'ordine della decina di ms. In questa fase il sistema provvede ad eseguire l'update di tutti i moduli. Nel breve lasso di tempo disponibile, i tre moduli dovranno occuparsi di gestire l'interazione fisica delle entità (physics socket), eseguire le loro scelte (logic socket), e rappresentarle a video (render socket).

In questa fase è anche gestita la lettura degli input, ovvero dei comandi che l'utente impartisce al sistema. Il componente che adempie a questo compito, dopo aver letto gli input, normalmente comunica le sue scelte solamente al modulo logico, o tutt'al più al socket logico delle entità. Ciò preserva una certa "coerenza strutturale" fra gli elementi che compongono il

sistema, poichè l'utente non dovrebbe mai aver bisogno di comunicare direttamente con il socket fisico e di render. Se ad esempio l'utente desidera far saltare il personaggio che sta controllando, invierà al suo socket logico il comando "salta". Il socket logico eseguirà poi internamente una serie di operazioni fra cui applicare una spinta verso l'alto all'entità richiedendolo al socket fisico, e aggiornare la raffigurazione in modo che rappresenti una persona intenta a saltare, richiedendolo al socket di render.

Questo è il flow chart che riassume quanto esposto:

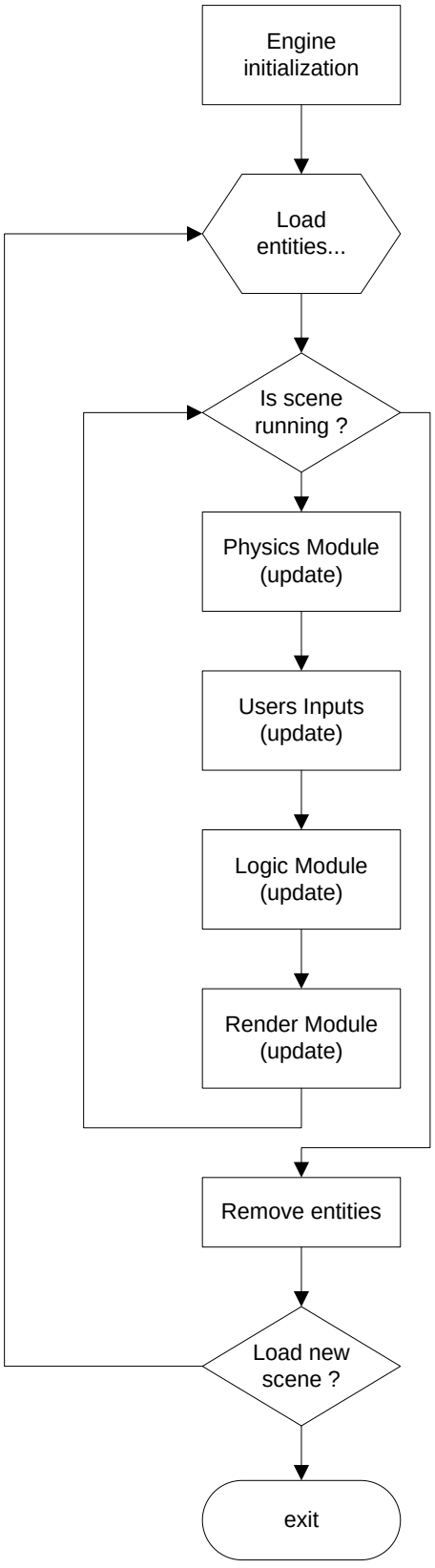


Figura 5.4: ciclo principale del motore grafico

5.3 Interfacce Entità

A livello di codice, ogni entità è rappresentata dalla classe Entity che deriva dall'interfaccia IEntity, la quale, fra gli altri, permette l'accesso ai tre sockets. Una lista di entità è presente dentro la classe Scene, che rappresenta il database della scena caricata e mette a disposizione una serie di metodi di supporto. Ogni socket è rappresentato da un puntatore di tipo interfaccia, in coerenza con il paradigma abstract factory. Ad esempio il socket fisico è rappresentato da un puntatore di tipo *IPhysicalSocket, e il modulo fisico si occuperà di allocare un modello ed assegnarlo tramite questo puntatore. Questo meccanismo, applicato a tutti i socket, permette una gestione analoga e similare per tutte le entità a prescindere da come esse siano internamente costituite. Ciò semplifica molto la struttura del codice e innalza fortemente la scalabilità del sistema. La figura 5.5 mostra in maniera sommaria l'interfaccia dei tre sockets, dove i metodi, per compattezza grafica, sono privi dei parametri:

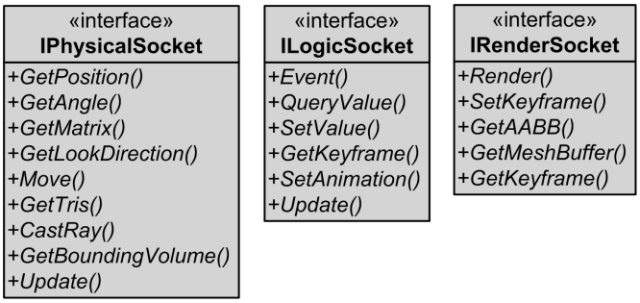


Figura 5.5: componenti principali delle interfacce dei tre sockets

Verranno ora illustrati rapidamente i tre sockets.

IRenderSocket

Concerne il rendering dell'entità, ovvero tutto ciò che riguarda la sua visualizzazione. Sono presenti metodi per le animazioni keyframe (il motore grafico non supporta ancora le animazioni scheletriche) e due metodi molti importanti:

- ***Render***, che ha questo prototipo:

```
virtual enmExitCode Render(CCamera *camera = NULL) = 0
```

Questo metodo renderizza la mesh assegnata al socket, trasformando i vertici secondo la matrice dell'oggetto *camera*, assegnando le texture maps degli eventuali materiali secondo una mappatura definita esternamente dal modulo render. Anche gli shaders e il render target (ovvero la superficie su cui renderizzare) sono specificati esternamente. In altre parole, l'entità renderizza se stessa ma secondo dei criteri imposti dal modulo render. Questi, riassumendo, comprendono:

- *Texture slot*, ovvero in quale canale assegnare le texture maps
- *Shaders*, ovvero i programmi che manipolano i vertici e scrivono i fragment
- *Render target*, ovvero dove renderizzare la mesh

- ***GetAABB***, che ha questo prototipo:

```
virtual void GetAABB(CAABB &AABB) = 0
```

Questo metodo scrive nel parametro la dimensione del bounding box axis alligned della mesh corrente.

ILogicSocket

Concerne la sola logica dell'entità, ovvero le regole che la governano, che nel caso di entità intelligenti è l'AI assegnata all'entità. I quattro metodi più importanti sono:

- *QueryValue*, che ha questo prototipo:

```
virtual enmExitCode QueryValue(enmLogicQuery logicQuery, float
                               *value, int index = 0) = 0
```

Questo metodo ritorna il valore di una proprietà logica dell'oggetto. Ad esempio il numero di colpi in un'arma, il danno di un proiettile, lo stato di un ascensore, etc..

- *SetValue*, che ha questo prototipo:

```
virtual enmExitCode SetValue(enmLogicQuery logicQuery, float
                             *value, int index = 0) = 0
```

Questo metodo è esattamente il duale del precedente, serve per specificare il valore di una proprietà logica.

- *Action*, che ha questo prototipo:

```
virtual enmExitCode Action(enmLogicCommand action, float
                           parameter = 0) = 0
```

Questo metodo scatena gli eventi. Un'evento è un qualcosa che accade all'entità o che l'entità è chiamata a fare. Ad esempio, se l'entità è una arma, l'evento è usato per chiedere all'arma di sparare un colpo.

- *Update*, che ha questo prototipo:

```
virtual enmExitCode Update() = 0
```

Questo metodo è il ciclo di update logico, e serve all'entità per far aggiornare il suo stato nel corso del tempo. Questo metodo permette alle entità di tenere traccia del tempo che scorre ed eseguire delle azioni in risposta ad eventi o a seguito della scadenza di un intervallo di tempo. Ad esempio un'entità ascensore può determinare di chiudere le porte dopo che un tempo massimo è trascorso. Un'entità soldato può decidere di sparare un colpo, o un'entità regola può decidere che la partita è terminata.

IPhysicsSocket

Concerne le proprietà fisiche, ovvero gestisce il modello fisico associato. I metodi più importanti sono:

- *SetPosition/SetPosition*, SetAngle/GetAngle, che hanno questo prototipo:

```
virtual const vec3f  &GetPosition() = 0
virtual void          SetPosition(const vec3f &pos) = 0
virtual const vec3f  &GetAngle() = 0
virtual void          SetAngle(const vec3f &angle) = 0
```

  Ritorna o specifica la posizione/angolazione in world space dell'entità.

- *GetTris*, che ha questo prototipo:

```
virtual void GetTris(SMeshTrisList &trisList, CEllipsoid
                     *ellipsoid = NULL) = 0
```

  Questo metodo restituisce la lista di triangoli della mesh dell'entità inclusi nel volume dell'ellisse specificato. Questo metodo è accellerato dall'uso di un octree, e la descrizione dettagliata sarà esposta in seguito.

- *Action*, che ha questo prototipo:

```
virtual enmExitCode Action(enmLogicCommand action, float
                           parameter = 0) = 0
```

  Questo metodo scatena gli eventi. Un'evento è un qualcosa che accade all'entità o che l'entità è chiamata a fare. Ad esempio, se l'entità è una arma, l'evento è usato per chiedere all'arma di sparare un colpo.

- *CastRay*, che ha questo prototipo:

```
virtual void CastRay(SCollision &result, SRay *ray,
                     enmRayTestType testType = rttMesh) = 0
```

  Questo metodo calcola l'intersezione di un raggio specificato con la mesh associata all'entità. É usato per molti scopi nell'engine. Si consideri ad esempio un proiettile. Questo metodo è chiamato specificando come raggio il vettore velocità del proiettile, ed esso restituisce le proprietà di un'eventuale collisione. Il numero di triangoli di un'entità è generalmente molto alto, e quindi dei metodi di ottimizzazione sono implementati per accellerare il processo. Verranno approfonditi in seguito.
  Il modulo fisico usa il metodo *CastRay* di ogni entità per determinare a livello globale se un raggio interseca con qualche entità della scena.

- *GetLookDirection*, che ha questo prototipo:

```
virtual const vec3f &GetLookDirection() = 0
```

  Ritorna il vettore direzione verso cui l'entità è orientata.

- *Move*, che ha questo prototipo:

```
virtual void Move(enmObjectMovement movement, float par) = 0
```

  Esegue un movimento dell'entità eventualmente secondo l'orientamento della stessa. Ad esempio, il comando *Move(omForward, 10)* sposta l'entità in avanti secondo la sua direzione. Contrariamente, il comando *Move(omX, 10)* sposterà l'entità rispetto all'asse X del sistema di riferimento world, ignorando quindi l'orientamento dell'entità. Nel caso di entità dotate di AI, questo metodo è chiamato dalla logica del logic socket.

- *GetMatrix*, che ha questo prototipo:

```
virtual matrix44f GetMatrix() = 0
```

Ritorna la matrice del sistema di riferimento dell'entità.

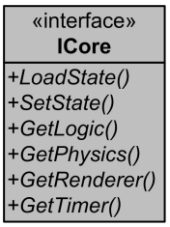
- *Update*, che ha questo prototipo:

```
virtual void Update(DWORD sceneTime,
            EntityList *worldEntities = NULL,
            EntityList *objectEntities = NULL,
            EntityList *triggerVolumes = NULL) = 0
```

L'Update esegue due compiti fondamentali:

- aggiorna le proprietà fisiche in funzione del tempo, quali la velocità, la posizione, le forze.
- determina le eventuali collisioni con altre entità della scena

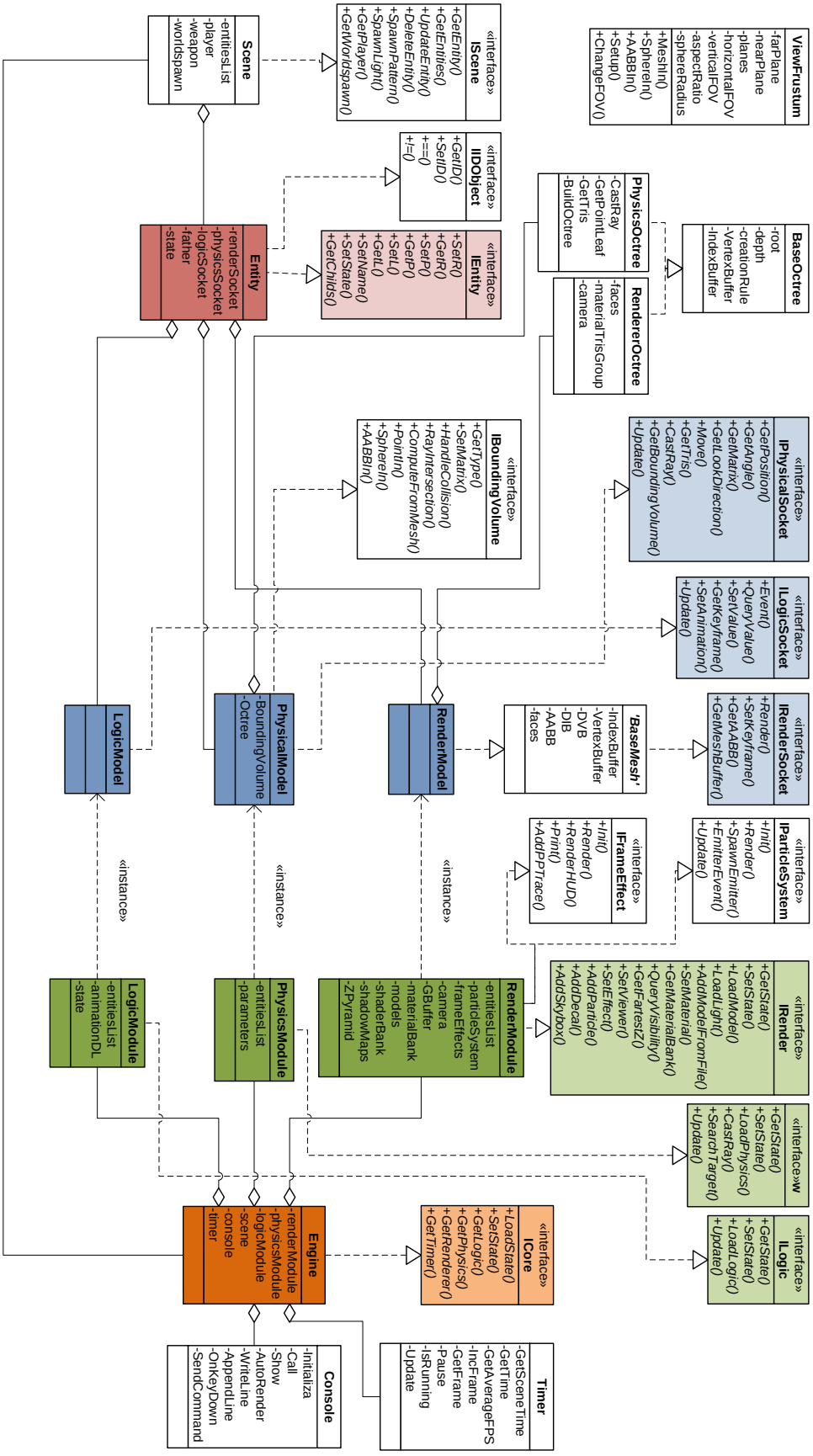
5.4 Interfaccia Core



La classe *Engine*, che deriva da *ICore*, è il blocco che contiene i componenti principali del sistema, fra cui le tre istanze dei tre moduli, ognuno derivante dalla relativa interfaccia (*IPhysicsModule*, *IRenderModule*, *ILogicModule*). Contiene inoltre un'istanza di *Scene*, un'istanza di *Timer* e un oggetto *Console*, usato per agevolare la comunicazione utente-engine. Il metodo *Run()* di Engine rappresenta la fase di update principale, precedentemente illustrata.

L'interfaccia *ICore* è usata anche per mantenere un puntatore globale all'istanza di *Engine*. Tale interfaccia permette di accedere a vari metodi e ai vari moduli i quali, sempre attraverso le loro interfacce, mettono a disposizione una serie di metodi specifici. In questo modo è possibile accedere e agire da qualsiasi punto del codice ad una vasta quantità di informazioni in maniera controllata.

5.5 Grafico Architettura Globale


ViewFrustum
-farPlane
-nearPlane
-planes
-horizontalFOV
-verticalFOV
-aspectRatio
-sphereRadius
+MeshIn()
+SphereIn()
+AABBin()
+Setup()
+ChangeFOV()
BaseOctree
-root
-depth
-creationRule
-VertexBuffer
-IndexBuffer
PhysicsOctree
-CastRay
-GetPointLeaf
-GetTris
-BuildOctree
RendererOctree
-faces
-materialTrisGroup
-camera
«interface»
IScene
+GetEntity()
+GetEntities()
+UpdateEntity()
+DeleteEntity()
+SpawnPattern()
+SpawnLight()
+GetPlayer()
+GetWorldspawn()
Scene
-entitiesList
-player
-weapon
-worldspawn
«interface»
IIDObject
+GetID()
+SetID()
+==()
+!=()
«interface»
IEntity
+SetR()
+GetR()
+SetP()
+GetP()
+SetL()
+GetL()
+SetName()
+SetState()
+GetChilds()
Entity
-renderSocket
-physicsSocket
-logicSocket
-father
-state
«interface»
IPhysicalSocket
+GetPosition()
+GetAngle()
+GetMatrix()
+GetLookDirection()
+Move()
+GetTris()
+CastRay()
+GetBoundingVolume()
+Update()
«interface»
IBoundingVolume
+GetType()
+SetMatrix()
+HandleCollision()
+RayIntersection()
+ComputeFromMesh()
+PointIn()
+SphereIn()
+AABBIn()
«interface»
ILogicSocket
+Event()
+QueryValue()
+SetValue()
+GetKeyframe()
+SetAnimation()
+Update()
«interface»
IRenderSocket
+Render()
+SetKeyframe()
+GetAABB()
+GetMeshBuffer()
'BaseMesh'
-IndexBuffer
-VertexBuffer
-DVB
-DIB
-AABB
-faces
«interface»
IParticleSystem
+Init()
+Render()
+SpawnEmitter()
+EmitterEvent()
+Update()
«interface»
IFrameEffect
+Init()
+Render()
+RenderHUD()
+Print()
+AddPPTrace()
«interface»
IRender
+GetState()
+SetState()
+LoadModel()
+LoadLight()
+AddModelFromFile()
+SetMaterial()
+GetMaterialBank()
+QueryVisibility()
+GetFartestZ()
+SetViewer()
+SetEffect()
+AddParticle()
+AddDecal()
+AddSkybox()
«interface»w
+GetState()
+SetState()
+LoadPhysics()
+CastRay()
+SearchTarget()
+Update()
«interface»
ILogic
+GetState()
+SetState()
+LoadLogic()
+Update()
LogicModel
PhysicalModel
-Octree
BoundingVolume
RenderModel
«instance»
«instance»
«instance»
LogicModule
-entitiesList
-animationDL
-state
PhysicsModule
-entitiesList
-parameters
RenderModule
-entitiesList
-particleSystem
-frameEffects
-camera
-GBuffer
-materialBank
-models
-shaderBank
-shadowMaps
-ZPyramid
«interface»
ICore
+LoadState()
+SetState()
+GetLogic()
+GetPhysics()
+GetRenderer()
+GetTimer()
Engine
-renderModule
-physicsModule
-logicModule
-scene
-console
-timer
Timer
-GetSceneTime
-GetTime
-GetAverageFPS
-IncFrame
-GetFrame
-Pause
-IsRunning
-Update
Console
-Initializa
-Call
-Show
-AutoRender
-WriteLine
-AppendLine
-OnKeyDown
-SendCommand

6. Deferred Rendering

Il dereffer rendering è un paradigma di rendering basato sui sistemi a scan-conversion che ha tovato larga diffusione negli ultimi anni, poichè a scapito di alcuni inconveniente fra i quali un alto uso di memoria e bandwidth, agevola la renderizzazione di scene che presentano un alto contenuto di sorgenti di luce e semplifica l'applicazione di molti effetti speciali.

6. 1 Standard Pipeline

L'osservazione principale è che il peso computazionale del calcolo del colore finale di un pixel è in gran parte dovuto al calcolo dell'illuminazione cui il fragment è sottoposto. In una pipeline *forward rendering* (non deferred rendering), per renderizzare un oggetto si sommano i contributi di tutte le luci che lo influiscono. I contributi delle luci possono essere algebricamente sommati (come si evince analizando il modello di illuminazione phong shading), pertanto l'oggetto è renderizzato una volta per ogni luce usando un buffer come accumulatore dei risultati calcolati. Lo zbuffering è sempre attivo durante questo processo, pertanto fragment non visibili non verranno scritti, mentre fragments più vicini all'osservatore sostituiranno un eventuale fragment scritto precedentemente.

Renderizzando ogni oggetto della scena con questo metodo, si ottiene un rendering corretto, dove ogni oggetto è stato disegnato tenendo in considerazione tutte le luci che lo interessano. Se si considera che alcune tecniche di illuminazione raffinate possono richiedere più di un pass per luce, si ottiene la seguente procedura:

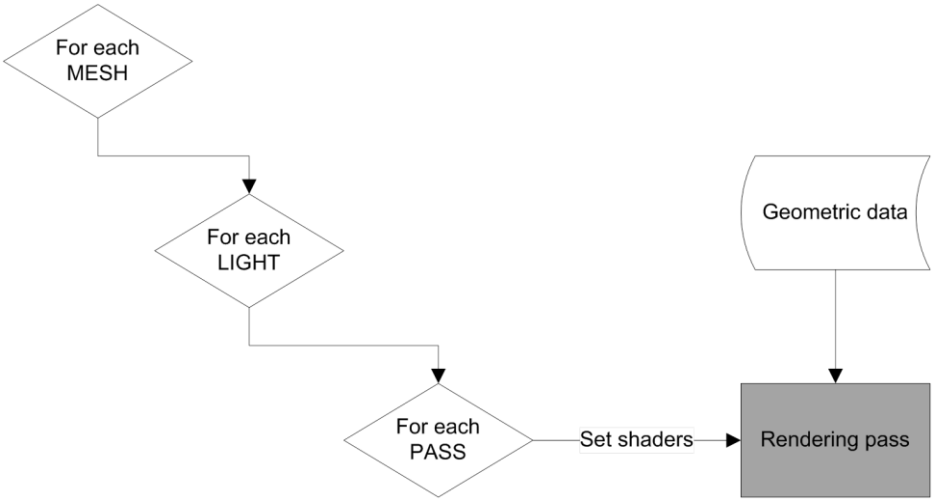


Figura 6.1: pipeline forward rendering. Si esegue un ciclo di luci per ogni oggetto

Se si considera di avere M mesh, L luci (medie per oggetto) e un solo pass per luce, la complessità diventa O(M·L).

Un altro notevole inconveniente di questo modo è l'individuazione delle sole luci che interessano ogni oggetto. Ad esempio se sono presento cento luci nella scena, mentre un oggetto è interessato solamente da cinque luci, non ha alcun senso eseguire un pass per luce, poichè il 95% del lavoro svolto dalla GPU sarebbe inutile. Esistono vari metodi per indiduare quali luci interessano ogni oggetto, ma non verranno considerati in questa trattazione. Tuttavia molto spesso il fatto che una luce influenzi una piccola parte di un modello, comporta che l'intero modello venga renderizzato, con notevole spreco di risorse.

6.2 Deferred Rendering - modus operandi

Il paradigma deferred rendering risolve l'inconveniente del caso precedente disaccoppiando il rendering della geometia dal rendering delle luci, eseguendo quindi il calcolo delle luci in differita (da qui il nome del paradigma). Questo viene ottenuto isolando i parametri dell'equazione di shading, che vengono, nella prima fase detta *GBuffer Pass*, renderizzati individualmente in una serie di buffers, senza eseguire alcun calcolo. Nella seconda fase, detta *Light Pass*, il GBuffer è usato come input per eseguire il calcolo vero e proprio della luminosità di ogni pixel.

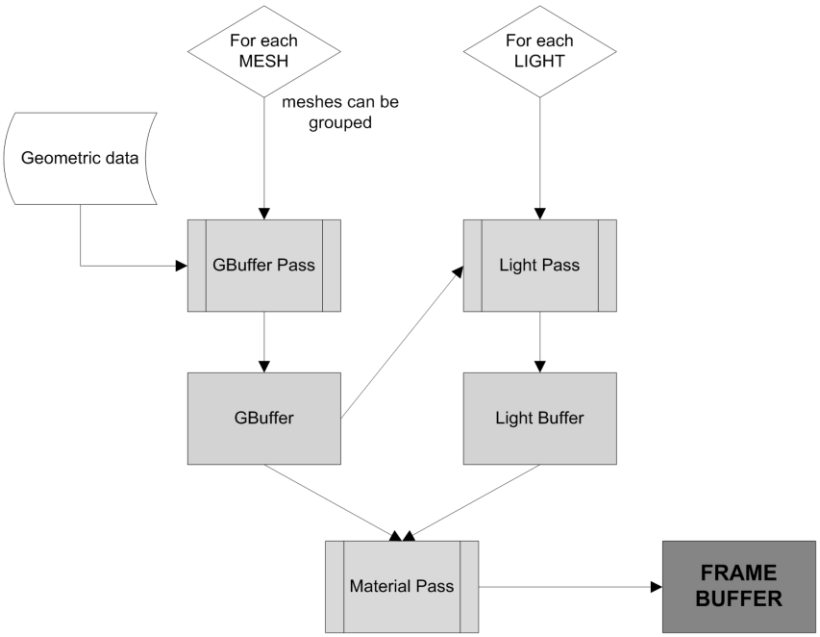


Figura 6.2: pipeline deferred rendering, in cui si disaccoppia la geometria dallo shading

Il light buffer ha lo scopo di accumulare il contributo di ogni luce, e rappresenta la sommatoria della formula di phong shading. Alla fine, in un unico passaggio detto *Material Pass*, il frame finale è composto modulando il colore della luce con il colore delle superfici. Questo paradigma porta la complessità ad un livello lineare. Supponendo di avere M meshes e L luci, la complessità diventa O(M+L).

Il GBuffer (da *Geometric Buffer*) si chiama in questo modo proprio perchè non conteniene alcuna informazione sulla luce, ma solamente sulla geometria e sul materiale delle superfici. Il GBuffer è composto da una serie di buffer della stessa risoluzione del frame buffer, dove ogni elemento corrisponde ad un pixel del frame buffer. Essendo il GBuffer Pass eseguito con zbuffering attivo, esso conterrà le informazioni solo di ciò che è visibile nella scena finale. Dacchè il light pass prende come input il GBuffer, è evidente il vantaggio di dover calcolare l'equazione di shading (che si ricorda essere il processo più esoso in termini di prestazioni) solamente dei fragment visibili, senza sprecare alcuna risorsa nel calcolare la luminosità di oggetti o porzioni di essi non visibili nella scena finale.

É importante notare che una luce raramente interesserà l'intera area dei buffer. In altre parole, una qualsiasi luce nella scena in genere influisce solo alcuni degli oggetti visibili. Ad esempio la luce evidenziata nella figura 6.3 influenza un'area piuttosto limitata della scena visibile, ed è quindi inutile calcolare lo shading per tutti i pixel esterni all'area di influenza della luce (approssimata in figura), poichè riceverebbero un valore nullo.

Figura 6.3: generalmente una luce interessa solo una parte dei fragment della scena finale

Il problema è risolto in modo conservativo con la seguente procedura. Nel momento del caricamento della scena, per ogni luce viene calcolata una bounding mesh che racchiude il volume i cui punti interni sono sufficientemente vicini alla sorgente di luce da risentirne dell'effetto.

Le mesh sono:

- *Sfere* per luci point lights
- *Coni* per spot lights
- *Fullscreen quad* per directional lights

Nel light pass, per ogni luce viene renderizzata la sua mesh. In questo modo la GPU calcolerà automaticamente (e molto velocemente) tutti i pixel potenzialmente interessati dalla luce (nel caso della figura 6.4 si tratta dei punti interni alla circonferenza tracciata). Infine, per ogni pixel, sfruttando la ricostruzione della posizione di un punto nota la sua profondità, viene risolta la formula di shading. Tuttavia, una ulteriore ottimizzazione è stata introdotta nel motore grafico.

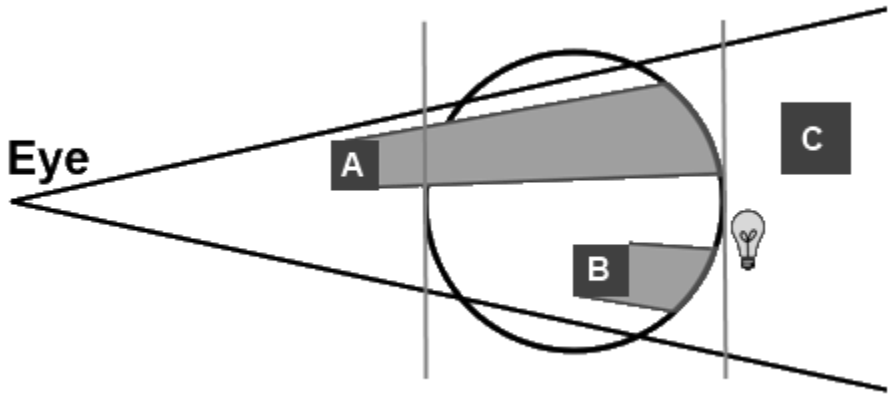


Figura 6.4: alcuni punti inclusi nella proiezione della light mesh possono essere esterni all'influenza della luce

Com'è visibile dalla figura, la bounding sphere della luce ha una tale dimensione per cui la sua proiezione interesserà tutti i punti del viewport. Tuttavia gli oggetti *A* e *C* sono esterni alla bounding sphere. É possibile inserire un test conservativo ma molto veloce che si occupa di individuare i punti la cui proiezione è interna alla proiezione del bounding volume di una luce, che però sono troppo lontani (rispetto all'asse Z) o troppo vicini allo spettatore, per essere influenzati dalla luce. Una coppia di valori limite di profondità, chiamata range limits e rappresentata dalle due linee verticali nella figura, è ricalcolata ad ogni frame per ogni luce con una semplice formula:

$$range\_limits = light\_viewspace\_z \pm light\_radius$$

Poi, nel pixel shader, è confrontata la profondità di ogni fragment. Se questa è esterna a tale intervallo, il fragment viene ignorato.

É un caso frequente quando ci si avvicina tanto ad una sorgente luminosa da rendere la mesh associata molto grande, fino ad occupare l'intero viewport. In questo caso tutti i punti della scena verranno considerati, ma fra essi quelli che rappresentano un eventuale sfondo o oggetti lontani, non verranno sicuramente raggiunti da questa sorgente di luce.

Si supponga di voler implementare un modello molto semplificato del phong shading, per ottenere il seguente modello di illuminazione:

$$K = K_{diffuse} = T \sum_{i=1}^{n} C_i$$

É sufficiente considerare solamente:

- la distanza dalla sorgente di luce da ogni punto. É usata per calcolare il parametro *C*, ovvero l'intensità e il colore della luce ricevuta
- la componente di luce diffusa della superficie
- il colore della sorgente di luce. É usato per calcolare il parametro *C*

Per questo semplice modello è sufficiente un GBuffer composto da due soli buffer, contenenti (per ogni fragment):

- la posizione spaziale del punto, usata per determinare la distanza dalla sorgente di luce.
- il colore diffuso dei punti della superficie, usato per ricavare il parametro *T* nella formula, che modula la luce ricevuta. Usualmente questo buffer si chiama *albedo buffer*.

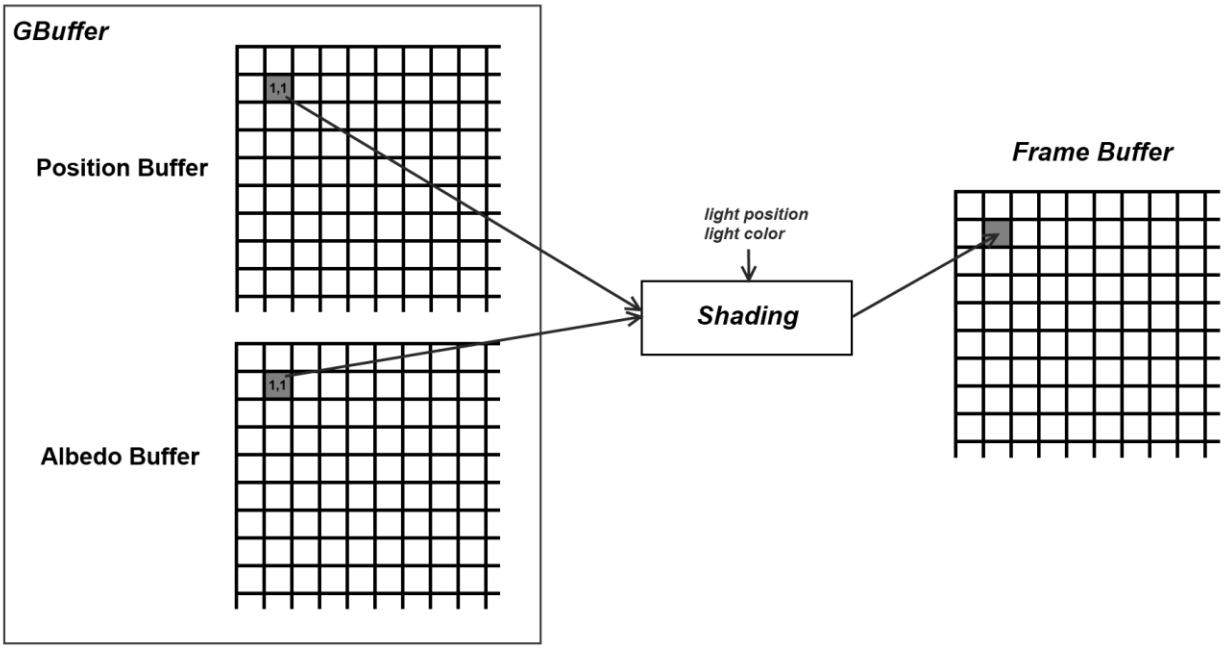


Figura 6.5: elementi corrispondenti dei buffers vengono combinati per calcolare la formula di shading

Il GBuffer è riempito nella prima fase, in cui la geometria è renderizzata ignorando le sorgenti di luce. Nella seconda fase, per ogni luce, viene calcolato C_i prendendo in considerazione la posizione e il colore della luce corrente e la posizione del fragment, ricavata dal GBuffer. In questa fase è calcolato:

$$L = \sum_{i=1}^{n} C_i$$

Infine, per comporre la formula completa di $K_{diffuse}$ viene eseguito il material pass che moltiplica gli elementi del light buffer con quelli dell'albedo buffer, ottenendo:

$$K = T \cdot L = T \sum_{i=1}^{n} C_i$$

6.3 Svantaggi

Il deferrer rendering presenta tuttavia alcuni svantaggi che è importante tenere in considerazione:

- impossibilità di renderizzare *superfici trasparenti*. Nel paradigma forward rendering le superfici sono facilmente renderizzate ordinandole secondo la profondità e renderizzandole a partire dalla più vicina all'osservatore grazia all'alpha blending. Nel deferred rendering ciò non è possibile perchè il GBuffer contiene in ogni elemento le informazioni relative ad un solo livello di profondità.

- bandwidth richiesta molto alta, a causa della grande dimensione del GBuffer che dev'essere riempito e letto più volte per frame

- impossibilità di applicare l'antialiasing hardware, poichè sebbene sia possibile applicare l'antialiasing nel GBuffer pass, ciò non è possibile farlo nel light pass, a causa del modo in cui questo è costruito. Tuttavia è possibile ovviare a questo inconveniente applicando un *edge detecting* eseguendo infine uno *smoothing* nel frame buffer in corrispondenza dei bordi individuati. L'edge detection in questo caso è di semplice realizzazione poichè è sufficiente considerare il depth buffer e individuae i bordi in corrispondenda di differenza di profondità superiori ad una soglia minima.

7. Engine Renderer

Il renderer è il componente che si occupa di visualizzare la scena, e pertanto è il cuore dell'intero engine. Lo sviluppo di un renderer è tutt'altro che semplice poichè la trasposizione in algoritmo dell'equazione di render richiede al programmatore di determinare un tradeoff che garantisca:

- buona resa grafica
- alto frame rate

La complessità che deriva nell'incontrare queste due richieste contrastanti ha frammentato il problema, generando, anzichè una unica soluzione, una famiglia molto vasta di algoritmi che si occupano di gestire i vari aspetti di rendering. Lo scopo dello sviluppatore è scegliere fra le possibiliti alternative quelle che, lavorando in sinergia, producono il risultato desiderato.

7.1 Quadro Generale

Il renderer sviluppato è basato su un modello di shading che simula la luce diretta. Un vero e proprio sistema di modellazione dell'illuminazione indiretta non è stato possibile svilupparlo in quanto si tratta di tematiche talmente complesse che meriterebbero un completo studio a parte. Tuttavia sono state implementate alcune funzionalità ulteriori che si affiancano al modello di shading per aumentare la resa grafica finale, presenti in gran parte dei moderni motori grafici:

- *SSAO* (screen space ambient occlusion), una tecnica che approssima l'ambient occlusion, la quale è una approssimazione dell'illuminazione indiretta
- *Bloom*, una tecnica che simula l'effetto di sovraesposizione che accade nelle lenti di dispositivi quali telecamere e fotocamere

- *Shadow Map*, una tecnica che permette di generare le ombre
- *Depth Of Field* (DOF), è una tecnica che simula la messa a fuoco di un singolo oggetto a scapito del resto della scena

Infine, è stato aggiunto una algoritmo per simulare il *motion blur*, ovvero l'effetto che accade durante la ripresa di una scena quando un oggetto inquadrato è tanto veloce che la sua immagine cambia durante la registrazione di un singolo frame.

Nella figura 7.1 è rappresentato lo schema dell'intera pipeline del renderer. Questi sono tutti i passaggi, esposti in maniera molto sintetica:

- la scena è renderizzata nel GBuffer
- uno zpyramid è derivato dal depth buffer per agevolare l'individuazione di entità e luci non visibili nella scena
- è applicata l'illuminazione
    - viene eventualmente eseguito il shadow pass qualora il tipo di lucerichieda il calcolo delle ombre
- è calcolato lo SSAO, il cui risultato viene fuso assieme alla luce calcolata precedentemente per comporre l'immagine finale (fragment RGB nel grafico).
- vengono aggiunti gli effetti speciali (particle system) e un'eventuale sfondo (Skybox)
- viene applicato il bloom all'immagine appena prodotta
- è calcolato il blurred frame, che verrà usato in questa fase sia per il DOF che per il motion blur

Gli ultimi due passaggi non sono visualizzati in figura:

- sono applicati effetti postumi all'immagine finale, tipo l'effetto cinematografico di temporanea macchiatura della lente
- è applicato l'HUD per completare l'immagine da visualizzare nel display

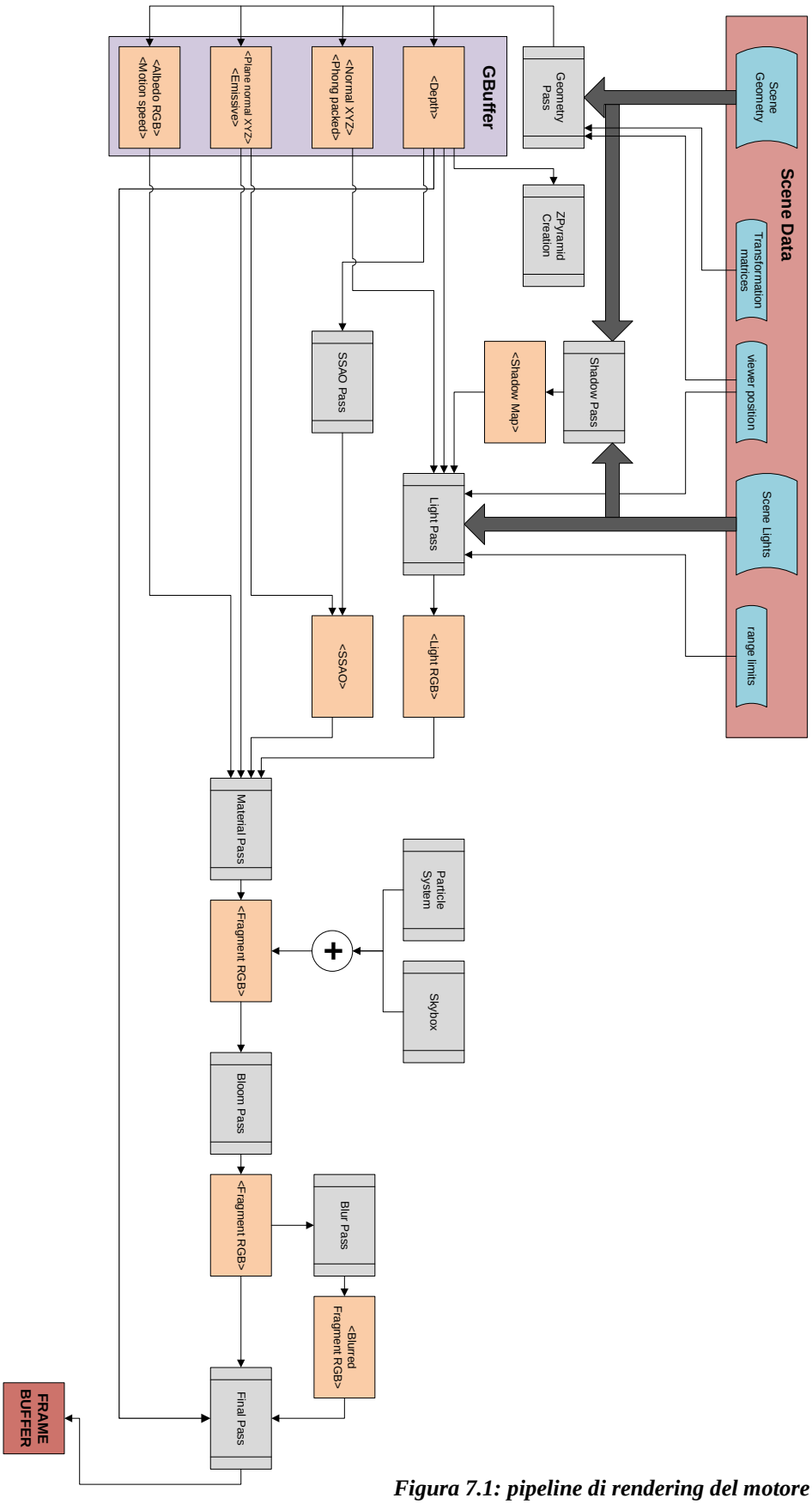


Figura 7.1: pipeline di rendering del motore grafico sviluppato

7.2 Modello di shading

É ora presentato il modello di illuminazione diretta implementato nel renderer sviluppato, basato sul modello phong shading. L'equazione base di phong, illustrata precedentemente, è la seguente:

$$K = K_{emission} + K_{diffuse} + K_{specular}$$

L'equazione espansa usata nel motore grafico, basata sulla precedente, è:

$$K = E + DTA + \sum_{i=1}^{n} HC_i[DT\max\{\boldsymbol{N} \cdot \boldsymbol{L}_i, 0\} + SG\max\{\boldsymbol{N} \cdot \boldsymbol{H}_i, 0\}^m(\boldsymbol{N} \cdot \boldsymbol{L}_i > 0)]$$

Dove *D* è il colore di riflessione diffusa del materiale, mentre *T* è il colore di riflessione diffusa dei singoli punti, ottenuto campionando una texture map detta *diffuse map*. Il modello è ampliato modulando la componente di riflessione speculare con il valore *S*, che specifica il colore di riflessione speculare del materiale, e con il valore *G*, che specifica l'entità della riflessione speculare per ogni punto. Questo valore è ottenuto campionando una texture map chiamata *gloss map*, che è associata ad ogni materiale. Qualora il materiale non dispoga di una gloss map, ne viene assegnata una di default in cui valore *G* è costante e pari a 0,5. Il parametro *H* è presente solo per le luci che supportano il calcolo delle ombre (attualmente solo le spot lights), e può valere zero o uno. Il parametro *E* rappresenta l'illuminazione propria di ogni punto del materiale, ed è ottenuto campionando una texture map detta *emissive map*.

Nel modello di shading usato nel motore grafico è stata introdotta un'altra variante. Normalmente il calcolo della C_i è basato sulla nota legge dell'inverso del quadrato. Questo è un comportamento tipico dei fenomeni elettromagnetici in cui una quantità di energia è irradiata in tutte le direzioni in uno spazio tridimensionale a partire da una sorgente puntiforme. Siccome il la superficie di una sfera è proporzionale al quadrato del raggio, più la radiazione si allontana dalla sorgente, più essa attraversa un'area sempre maggiore. Tuttavia, usando la formula di intensità di luce:

$$I = \frac{1}{k_c + k_l d + k_q d^2}$$

dove d è la distanza della sorgente di luce da un punto nello spazio, non verrà mai raggiunto il valore zero.

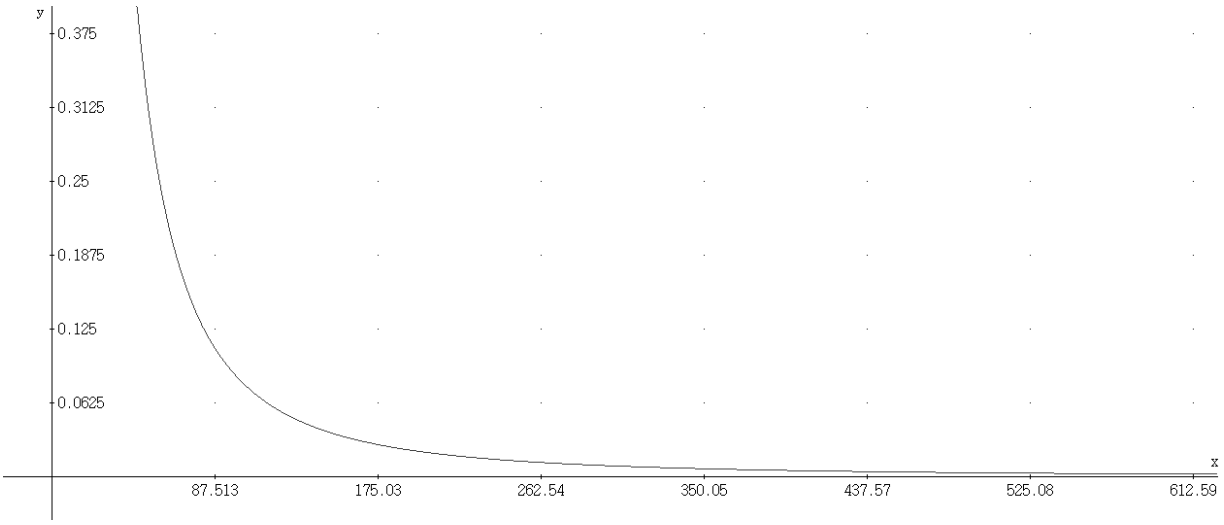


Figura 7.2: il fallout della legge dell'inverso del quadrato è fisicamente corretto ma non raggiunge mai lo zero

Ciò impone una soglia minima sotto alla quale i fragment non verranno illuminati. É stato notato che questa soluzione comporta qualche problema. L'occhio è sensibile a piccole variazioni di luce, quindi la soglia va scelta molto bassa per evitare artefatti (salti netti di luminosità). Purtroppo questo comporta il calcolo di luminosità per un numero molto alto di punti. Per risolvere il problema sarebbe possibile sfruttare una texture map unidimensiole i cui valori siano il campionamento di una prestabilità funzione di fallout. Tuttavia questo metodo impone una cerca rigidità nella scelta dei parametri. La soluzione proposta, invece, è una modifica alla formula dell'intensità in modo che:

- sia conservato un fallout analogo alla formula originale
- la funzione passi per lo zero

La nuova formula diventa:

$$I = \frac{\frac{w - d^2}{w}}{k_c + k_l d + k_q d^2}$$

dove w è il raggio di influenza desiderato per la luce. Per non modificare il fallout, tale valore è scelto risolvendo la formula

$$\frac{1}{k_c + k_l d + k_q d^2} = s$$

usando per s un valore ragionevolmente basso, prossimo alla soglia di percezione dell'effetto della luce in un punto. In questo modo si forza la formula a passare per lo zero quando, nella forma originaria, sarebbe stata molto prossima ad esso. La figura seguente mostra il grafico di una funzione con $k_c = 0,\ k_l = 0,\ k_q = 0{,}12$. Il valore *w* calcolato è circa 30.000.

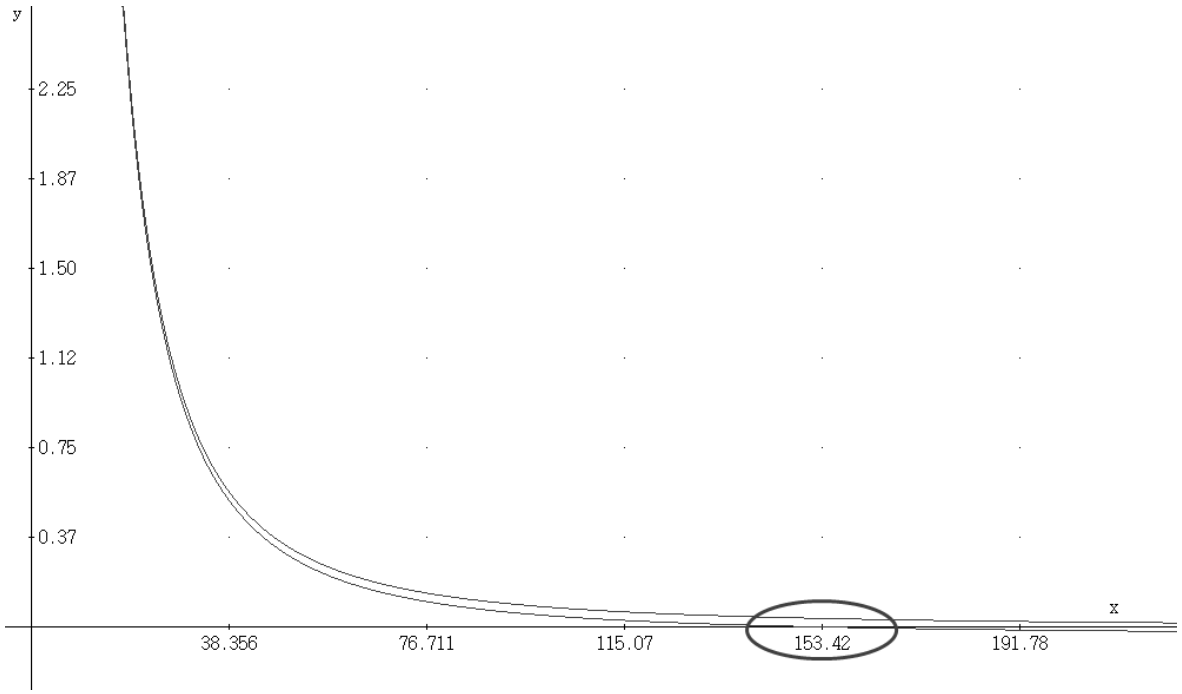


Figura 7.3: la funzione proposta ha un fallout del tutto simile, ma raggiunge lo zero(zona evidenziata)

Come si osserva dalla figura, le due funzioni sono praticamente identiche, ma la formula proposta passa per lo zero, quindi evitando discotinuità nello shadind e permettendo di variare manualmente il raggio della sorgente di luce.

7.3 Layout GBuffer

Per ottenere questi parametri è necessario un GBuffer che mantenga i seguenti parametri per ogni fragment:

- posizione
- normale
- albedo RGB (*T*)
- specular value (*G*)
- specular exponent (shininess)
- emissive value

I parametri relativi alla superficie, cioè quelli non relativi ai singoli punti, come il colore di riflessione diffusa e speculare, non sono salvati nel GBuffer ma piuttosto vengono istruiti agli shader nel momento della chiamata di rendering.

Siccome lo scopo era raggiungere una dimensione ragionevole del GBuffer, in modo da ridurre la bandwidth richiesta al minimo, dopo una serie di prove è stato scelto di usare un formato a 32bit per elemento. Ogni componente è salvata con un numero in virgola mobile a 8bit, che ha dimostrato di avere una precisione sufficiente, molto spesso paragonabile a quella ottenuta con 16bit per elemento. Il campo Phong Packed identifica un valore numerico che rappresenta sia il valore di intensità speculare che l'esponente di phong. Sono dedicati 6bit al primo, permettendo un intervallo di 64 valori, e 2bit all'esponente, quindi 4 valori che verranno usato per mappare un set di quattro esponenti prestabiliti (30, 60, 120, 250).

É assente un buffer che memorizzi la posizione dei fragment perchè è più conveniente ricavarla all'occorrenza partendo dalla profondità del fragment, che è disponibile dal depth buffer. Si nota che è conveniente aumentare la complessità di calcolo anzichè aumentare la

bandwidth. Ripercorrendo al contrario la formula di proiezione, la posizione in view space è determinata in questo modo:

$$P_{vs} = < \frac{S_x \cdot Z}{M_{00}}, \frac{S_y \cdot Z}{M_{11}}, Z, 1 >$$

dove *S* è il vettore 2D delle coordinate in screen space.

Questo è il formato del GBuffer usato nel motore grafico sviluppato, assieme agli altri buffer di supporto che verrano in seguito illustrati:

R8	***G8***	***B8***	***A8***	***name***
Diffuse Albedo RGB			**Pixel Velocity**	RT0
Plane Normal XYZ			**Emissive**	RT1
Normal XYZ			**Phong Packed**	RT2
Depth				RT3
GPU Depth			**Stencil**	DS
R	**G**	**B**	**Motion Speed**	Frame Buffer
R	**G**	**B**		Bloom Buffer
SSAO Value				SSAO Buffer
R	**G**	**B**		Blurred Frame
Depth				Shadow Map

Tabella 7.1: formato del GBuffer (azzurro) e dei buffer di supporto (rosa). Ogni buffer ha la stessa dimensione del frame buffer (risoluzione display), ed ogni elemento è a 32 bit, composto da quattro componenti float a 8 bit. In una risoluzione di 1280x1024 la dimensione del GBuffer è di 20Mb, che con 40fps corrispondono a 800Mb/s.

7.4 Struttura Materiale

La struttura di un materiale nel motore grafico è la seguente:

```
struct SMaterial
{
	string		name;

	STexture	*pDiffuse;
	STexture	*pNormal;
	STexture	*pDetail;
	color32		diffuseColor;
	color32		specularColor;

	int		specularExponentIdx;
[…]
}
```

Sono presenti i due colori di riflessione del materiale. Il valore *specularExponentIdx* è usato per ricavare il valore di shininess per la riflessione speculare, ovvero il valore *m* nella formula, ed è il medesimo per tutto il materiale. Il fatto che il valore sia lo stesso per ogni punto è parso un'accettabile approssimazione, poichè lavorando su questo parametro e sulla gloss map, è possibile ottenere un buon modello di riflessione speculare per gran parte dei tipi di materiali.

Al fine di ottimizzare lo spazio richiesto, i valori RGB della emissive map sono stati ridotti ad un singolo valore di luminosità, che amplificherà il rispettivo valore della diffuse map. Tale scelta comporta una leggera riduione di versatilità nell'uso nel canale emissive, ma comporta un alto risparmio di memoria. Questo valore è scritto nel quarto canale della diffuse map, il cui formato diventa:

R	G	B	Emissive

La normal map contiene le normali dei punti, usate per il normal mapping e il parallax mapping, ma nel quarto canale è salvato il valore di profondità dei punti, ottenendo il formato:

X	Y	Z	Height

É presente inoltre una terza texture map, detta *detail map*. La detail map è una finezza presente in molti motori grafici, e contiene delle informazioni in alta frequenza, che seguono un certo pattern randomico, per ogni materiale. Questo serve a mantenere una buona e convincente resa grafica anche qualora la risoluzione delle texture del materiale diventi insufficiente. Questo fenomeno di aliasing, che è stato accennato anche in precedenza, è evidente quando un elemento di una texture del materiale viene ad occupare una grande sezione del viewport, ad esempio quando l'osservatore si avvicina molto ad un oggetto. In questo caso l'area renderizzata ha una risoluzione troppo alta rispetto alla rispettiva area della texture map (frequenza troppo alta). Il trilinear filtering si comporta molto bene nel mitigare il fenomeno, ma in molti casi è insufficiente. Per applicare la detail map in alta frequenza è sufficiente far variare molto rapidamente le coordinate di campionamento. Questa è la funzione di campionamento usata nel pixel shader:

```
float3 detail = tex2D(detailMap, (In.Texture*detailFreq)).xyz * 2 – 1
```

Nel caso dell'engine, vengono usate le coordinate UV della diffuse map moltiplicate per un fattore *detailFreq* pari a sei. Il valore ricavato dalla detail map è usato per perturbare:

- le componenti RGB della diffuse map
- le componenti XYZ della normal map
- la componente di riflessione speculare (*G* nella formula)

Il risultato è estremamente convincente dal punto di vista della resa e del realismo, in quanto vengono ad essere modellate tutte la casuali imperfezioni presenti sulla superficie di molti materiali. Inoltre, la detail map è molto semplice da implementare.

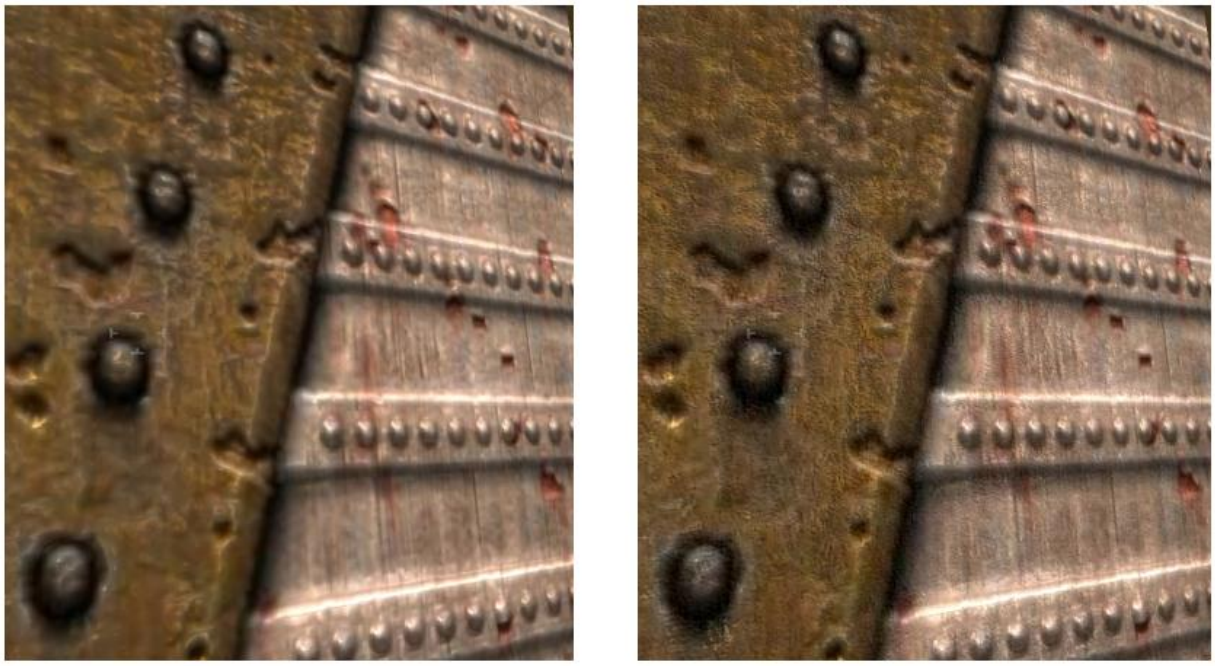


Figura 7.4: la detail map introduce elementi ad alta frequenza che simulano le imperfezioni dei materiali

7.5 Rendering Pipeline

La pipeline completa del motore sviluppato è mostrata in figura 7.1. I blocchi grigi rappresentano le fasi (pass) di rendering, mentre i blocchi arancioni rappresentano i risultati sotto forma di buffer. I blocchi dei pass sono disegnati in ordine temporale. I pass fondamentali sono i seguenti:

- GBuffer Pass
- Light Pass
- Material Pass

Le altre fasi sono degli effetti che verranno illustrati in seguito. Di seguito sono elencati i passaggi di inizializzazione comuni ad ogni pass:

- Reset dei buffers

- Imposta i render targets, istruendo lo shader su dove scrivere i valori
- Imposta i texture slots, istruendo le entità su come assegnare le texture maps dei materiali
- Renderizzazione modello (*RS_RenderEntity(IEntity *entity, CCamera *camera)*). Il modello è una mesh se si tratta di un'operazione mesh→buffer (es: GBuffer pass, Shadow map pass), oppure un quad 2D della dimensione della viewport se si tratta di un'operazione buffer→buffer (es: light pass

GBuffer Pass

In questa fase ogni modello è renderizzato nel GBuffer. I parametri spaziali del modello e le proprietà dei materiali sono specificate prima della chiamata di rendering. In particolare, le chiamate di rendering (*draw call*) sono raggruppate in modo che ogni chiamata renderizzi tutti i triangoli del modello formati dallo stesso materiale, dal momento che le texture maps possono essere assegnate una sola volta prima della draw call. Questo ha soprattutto lo scopo di ridurre al minimo (*batching*) il numero di draw calls.
Il vertex e pixel shader di questo pass si occupano principalmente di:

- Trasformare e proiettare i vertici
- Trasformare le normali da tangent space a world space
- Scrivere i parametri compressi per la riflessione speculare (specular intensity + specular exponent nello stesso valore)
- Applicare la normal map alle texture maps

La struttura input del vertex shader per il GBuffer, che riceve in ingresso i vertici del modello, contiene i valori propri di ogni vertice del modello:

```
struct VS_INPUT
{
   float4 Position   : POSITION;
   float2 Texture    : TEXCOORD0;
   float4 Normal     : NORMAL;
```

```
    float4 Tangent    : TANGENT;
}
```

La struttura di output del vertex shader è la stessa di input del pixel shader, e contiene i seguenti valori:

```
struct VS_OUTPUT
{
    float4 Position   : POSITION;
    float2 Texture    : TEXCOORD0;
    half3  Normal     : TEXCOORD1;
    half3  Tangent    : TEXCOORD2;
    half3  Bitangent  : TEXCOORD3;
    float4 Data       : TEXCOORD4;
}
```

Tutti i campi sono autoesplicativi, eccetto il campo data. Questo vettore a quattro dimensioni è usato dal pixel shader, e contiene le componenti x e y della direzione di vista, la profondità del fragment e la sua velocità (alcuni parametri saranno approfonditi in seguito) secondo il seguente formato:

Dx	Dy	Z	velocity

La struttura di output del pixel shader contiene i valori che verranno salvati su ogni buffer del GBuffer:

```
struct PS_OUTPUT
{
    float4 Normal     : COLOR0
    float4 Diffuse    : COLOR1
    float4 PlaneNormal : COLOR2
    float4 Depth      : COLOR3;
}
```

É evidente la corrispondenza esatta fra i valori di output del pixel shader del GBuffer e la struttura del GBuffer presentata nella figura.

Light Pass

In questa fase tutto il lavoro è svolto da pixel shader, in quanto il vertex shader riceve in ingresso solamente un quad che rappresenta l'intera area dei buffer in ingresso. Come si vede dalla figura 7.5, il pixel shader riceve in ingresso due categorie di informazioni:

- GBuffer
- Parametri sorgente di luce

Per quanto riguarda i parametric di ingress degli shader, figurano solamente i vertici dei punti. Il GBuffer e i parametri delle luci sono passati esternamente, ad ogni pass per luce. Un pass è eseguito per ogni luce, usando il light buffer come accumulatore dei risultati. É in questa fase che l'equazione di shading viene calcolata. Usando l'algoritmo spiegato in precedenza, si individuano solamente i pixel potenzialmente interessati da ogni luce. Nella seguente figura, a sinistra, è illustrato l'effetto di una point light, mentre a destra è rappresentata la sua bounding sphere. I punti colorati di rosso sono quelli che potenzialmente possono essere influenzati.

Figura 7.5: visualizzazione della light mesh (rossa). Solamente i punti rossi verranno considerati per il calcolo della luce il cui effetto è visibile a sinistra

Il vertex shader del light pass si occupa di determinare tali fragment, mentre il pixel shader esegue i seguenti passaggi per ognuno di essi, ricavando tutti gli elementi necessari per la formula di shading dai buffer del GBuffer:

- Range-limit test (introdotto precedentemente)
- Calcolo della posizione in world space
- Calcolo del fattore di ombra, se previsto dal modello di luce
- Calcolo dell'intensità della riflessione diffusa
- Calcolo dell'intensità della riflessione speculare
- Modulazione del colore diffuso del fragment con il colore della luce

La seguente figura illustra come appare il light buffer dopo aver accomulato tutti i contributi delle luci presenti nel viewing frustum di una scena:

Figura 7.6: light buffer, con normal mapping applicato

Material Pass

In questa fase, nella forma più semplice, viene moltiplicato il light buffer per l'albedo buffer:

Figura 7.7: albedo buffer, ovvero la scena renderizzata senza luci con solo la componente di luce diffusa

Ottenendo il seguente risultato:

Figura 7.8: immagine composta modulando l'albedo buffer con il light buffer

7.6 Ambient Occlusion

L'ambient occlusion è un metodo di shading che aggiunge realismo al rendering calcolando il valore di attenuazione della luce che raggiunge un punto della scena a causa di elementi occlusori. A differenza dei metodi di shadowing, che determinano le ombre considerando tutti gli occlusori posti fra una sorgente di luce e un punto nello spazio, l'ambient occlusion gestisce solamente gli occlusori posti nelle *vicinanze* del punto in esame. Per questo motivo, l'ambient occlusion di per sè è solamente una forte approssimazione dell'illuminazione indiretta, che comunque aumenta di molto la percezione della forma e profondità degli oggetti.

L'idea alla base dell'algoritmo è che, dato un punto nello spazio, la radiazione luminosa giunge ad esso da *ogni direzione* inclusa in una semisfera centrata nel punto e orientata secondo la sua normale. Nella figura 7.8 è visualizzato un punto **P**, la sua normale **N**, ed una serie di raggi

che, partendo da **N**, vanno in direzione dei punti della superficie della sfera di raggio prefissato. Ogni volta che un raggio incontra un elemento, il valore di occlusione del punto **P** aumenta.

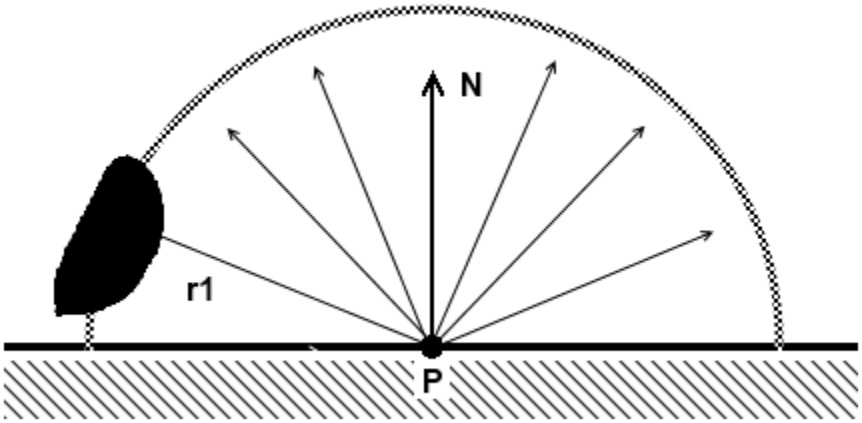


Figura 7.8: rappresentazione 2D della semisfera di radiazione luminosa sul punto P

Disponendo di una funzione booleana di visibilità *V* per ogni punto dello spazio, l'ambient occlusion di un punto P può essere calcolato integrando sulla semisfera:

$$A(p) = \frac{1}{\pi}\int_{\Omega} V(p,\omega)(\omega \cdot \boldsymbol{N})d\omega$$

In genere la funzione *V* è pesata secondo la distanza dal punto **P**, nel senso che un occlusore vicino comporta una più alta attenuazione. Per un motore grafico real-time questa formula non può essere calcolata nemmeno usando un metodo Monte Carlo, a causa della complessità del calcolo di *V*. La figura 7.9 mostra come appare l'ambient occlusion calcolato attraverso ray tracing.

Figura 7.9: componente ambient occlusion di una scena. Zone scure equivalgono a zone con poca luce

7.6.1 Screen Space Ambient Occlusion

Lo screen space ambient occlusion (SSAO) è una tecnica che è stata introdotta a partire dal 2008, che permette di calcolare l'ambient occlusion a partire solamente dal depth buffer. Nasce dall'osservazione che il depth buffer, di fatto, è una rappresentazione della scena e contiene informazioni sufficienti per stabilire l'ambient occlusion. In seguito è stata introdotta una variante che include l'uso del normal buffer, che ne aumenta la qualità. Quest'ultima è stata implementata nel motore grafico. Il fatto di usare solamente i punti screen space semplifica drasticamente il calcolo della funzione di visibilità, a scapito però di una forte approssimazione. A causa della limitata quantità di informazione, tuttavia, queste approssimazioni introducono degli errori talvolta ben visibili che verrano discussi in seguito.

L'idea di base dello SSAO è espressa nella figura 7.10. Per ogni fragment, si genera un numero N di raggi che determinano dei punti nello spazio. Questi punti sono usati per

campionare il depth buffer, al fine di determinare se esiste un occlusore. Se il valore di profondità ricavato dallo zbuffer è maggiore della profondità del punto, allora si considera che il raggio ha intersecato un occlusore. Nella figura, la profondità del raggio r1 è minore della profondità scritta nello zbuffer (area grigia). Infatti il raggio ha colliso con un occlusore. In questo caso la funzione di visibilità associata al raggio ha valore uno, altrimenti zero. La somma dei valori di questa funzione divisa per il numero di raggi da il valore di ambient occlusion del punto.

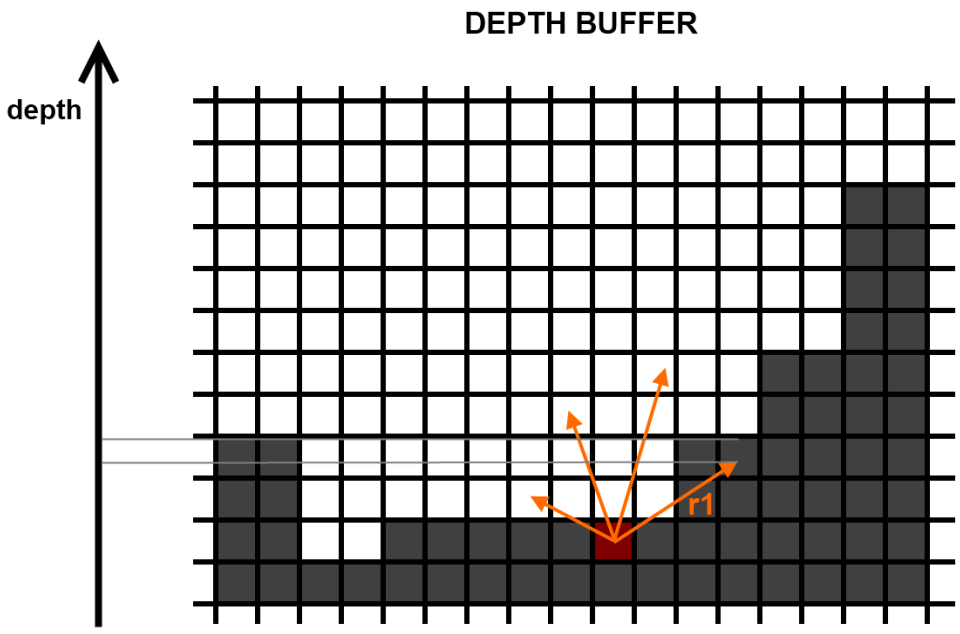


Figura 7.10: rappresentazione del depth buffer per SSAO. I raggi che terminano con valore Z minore del valore del depth buffer sono considerati occlusi

Questo metodo presenta un inconveniente. Non dispondendo della normale dei punti, è impossibile determinare la semisfera associata, e pertanto i raggi saranno generati in ogni direzione. In una superficie piana, metà dei raggi individueranno una occlusione. Il problema è evidente in prossimità degli spigoli, che in questo modo appariranno più chiari del resto dell'oggetto, poichè nell'intorno degli spigoli, essendo presente una discontinuità della geometria, in genere la profondità è minore. Questo artefatto è evidenziato nella seguente figura, in cui gli spigoli degli oggetti sembrano ricevere più luce della superfici piane:

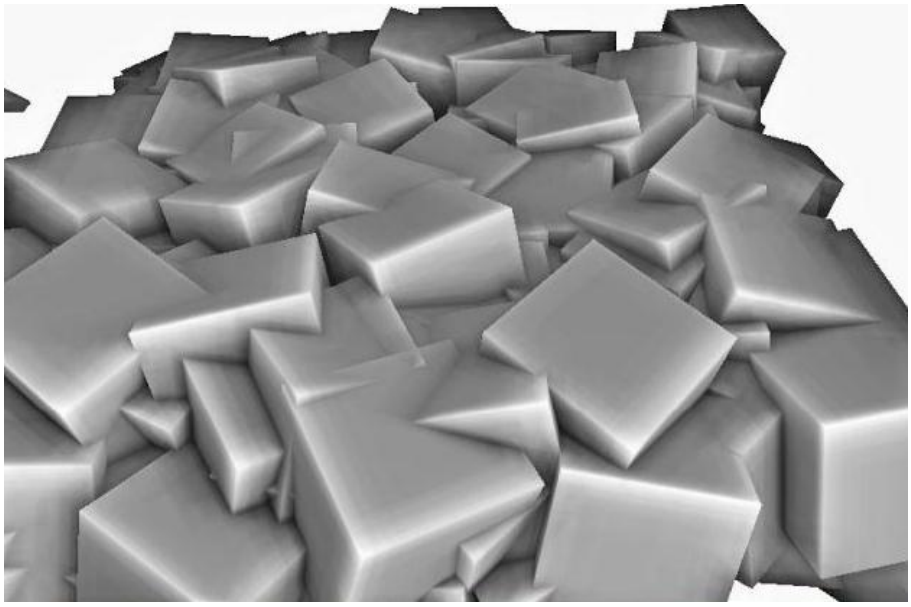


Figura 7.11: componente SSAO calcolata ignorando le normali delle superfici. Gli spigoli appaiono più luminosi

Nella pratica, tuttavia, l'effetto visivo risultante rimane convincente, seppur errato, e per questo tale tecnica è stata inserita in molti motori grafici commerciali. La variante che utilizza le normali dei punti produce risultati più convincenti. L'algoritmo è praticamente identico eccetto che al momento della creazione del raggio, questo è moltiplicato per il segno del prodotto scalare fra la normale del punto in esame e la direzione del raggio. In questo modo se un raggio è diretto contro la superficie, viene eliminato. La figura 7.12 mostra la componente SSAO della scena di figura 7.8. Come si nota, in virtù dell'uso delle normali, le superfici piane sono correttamente prive di attenuazione (colore bianco).

Figura 7.12: componente SSAO del motore grafico sviluppato

Lo shader implementato nel motore grafico esegue i seguenti passaggi:

- ricava posizione viewspace del fragment
- ricava normale viewspace del fragment
- per N raggi:
    - ricava raggio randomico a partire da un kernel predefinito
    - w = sign(dot (raggio, normale))
    - moltiplica raggio per w
    - ricava punto da campionare = posizione+raggio
    - calcola coordinate screen space del punto
    - campiona zbuffer
    - calcola differenza fra profondità fragment e profondità campionata
    - se differenza > soglia_minima, determina peso occlusione e incrementa occlusione totale
- dividi per numero di raggi
- calcola occlusione finale

Per quanto riguarda il calcolo dei raggi, è possibile operare nel dominio 2D o 3D. Quest'ultimo approccio è leggermente più lento, ma molto più semplice da realizzare, ed è stato scelto per il motore sviluppato. Nella pratica si possono usare delle formule raffinate per il calcolo del peso dell'occlusore e del valore di occlusione finale. Nello shader sono state usate queste formule:

```
occlusion += (1. + depthWeight) * (1. - smoothstep(minDist, maxDist, depthDiff * depthDiff))
occlusion = occlusion / SAMPLE_COUNT;
Out.Occlusion = float4( max(0, 1 - borderAdjust * pow(abs(occlusion), fParams.z)), 1, 0, 0);
```

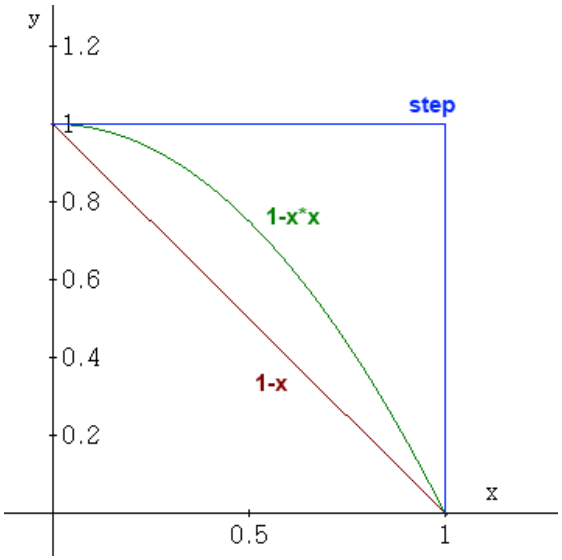


Figura 7.13: fallout dell'occlusion per i raggi calcolati da SSAO

Nell'implementazione il peso è calcolato secondo la distanza al quadrato (funzione verde in figura), anzichè secondo una funzione step. La funzione smoothstep è usata per bilanciare il peso secondo due valori limite. Questi parametri, assieme a alcuni altri introdotti in seguito, sono definiti dal programmatore in funzione della resa grafica desiderata. Il valore borderAdjust è stato introdotto per ovviare al problema di non avere più informazione in prossimità dei punti più esterni dello screen space. In questo caso i raggi che cadono all'esterno dello screen space vengono ignorati, mentre quelli interni riceveranno un peso maggiore per compensare. Il valore *fParams.z* determina l'intensità globale dell'occlusione. *depthWeight* è un valore che è stato introdotto per aumentare il peso dell'occlusione di punti molto lontani dall'osservatore. Questo è

stato fatto perchè la risoluzione finita dello zbuffer determina una quantità di informazione minore per i punti lontani dall'osservatore.

Trattandosi di uno shader molto lento, per evitare di influire troppo sul frame rate è possibile eseguire lo shader ad una risoluzione ridotta rispetto al frame buffer, in genere dimezzata. Contenendo solo informazioni a bassa frequenza, il risultato finale è solo lievemente inferiore, tuttavia il tempo richiesto è un quarto.

7.6.2 Filtering

Il limitato numero di raggi che è possibile generare per ogni fragment è limitato dalla necessità di ottenere dei tempi di esecuzione molto bassi. Nella pratica, il numero varia da otto a sedici. Usare un basso numero di raggi casuali per ogni punto produce un risultato parziale, che genera una mappa di ambient occlusion molto frammentata. Per ovviare a tale inconveniente è necessario ricorrere ad un filtro passa basso per eliminare le discontinuità dei valori di occlusion dei punti.

Un metodo molto veloce che produce risultati accettabili è applicare un filtro gaussiano (effetto blur). Il filtro gaussiano è un'operazione che algoritmicamente può essere scomposta in due passaggi, uno per asse, riducendo la complessità da O(n2) a O(2n), ed è quindi molto veloce. Il filtraggio rende omogenea la distribuzione della luminosità ma ha il grosso svantaggio di generare una errata distribuzione dell'occlusione (bleeding) in molti casi. Tipicamente il blurring elimina i bordi, com'è dimostrato in figura 7.14:

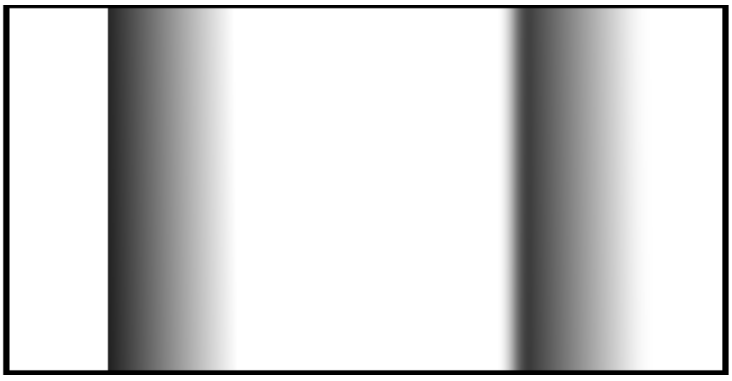


Figura 7.14: un filtro passa basso non preserva i bordi

Per ovviare a questo problema si usa un filtro bilaterale, che si usa proprio quando si vuole preservare una forma di edge in qualche distribuzione. Nel filtro bilaterale, i valori del kernel non sono pesati solamente in funzione della distanza, ma anche secondo altri parametri. In questo caso, si considera la differenza della profondità dei punti del kernel con quella del punto in esame. Questo codice è tratto dallo shader del filtro:

```
float i = tex2D(buffer, coord);
float diff = abs(i - ifrag);
float POW = 1;
if (diff > 0.5)
      w = 0;
else
      w = POW / (1 + diff);
value += i * w;
```

Il grosso inconveniente è l'impossibilità di scomporre il filtro, che quindi rimane di complessità *O(n2).* In pratica è comunque possibile applicare la scomposizione anche se in questo modo il risultato ottenuto non sarà perfetto.

Figura 7.15: componente SSAO (A), filtrata con gauss filter (B), filtrata con bilateral filter (C)

La figura 7.16 mostra il risultato del filtraggio applicato nel motore grafico. Teoricamente l'attenuazione SSAO andrebbe applicata solo alla componente di luce ambientale, ma nel motore sviluppato è stato scelto di dare maggiore peso alla componente SSAO, usando questa formula per determinare la quantità di luce finale per un pixel:

```
float directLightSSAOAtt = (7 * SSAOvalue + 3) * 0.1
```

```
float3 fragment = albedo * (light * directLightSSAOAtt +
                      ambient * SSAOvalue + emissive * 2)
```

Figura 7.16: esempio di SSAO buffer prodotto dal renderer sviluppato per la scena di figura 7.8

7.7 Bloom

Il bloom è un artefatto tipico dei dispositivi ottici che causa aloni nei contorni degli oggetti fortemente illuminati. É dovuto al fatto che nessuna lente è in grado di mettere perfettamente a fuoco un oggetto, e sorgenti di luce poste a distanza da essa causano effetti di rifrazione.

Figura 7.17: effetto bloom in una fotografia. Le pale del mulino vengono cancellate dal bagliore

Per ricreare l'effetto bloom è ricreato sommando all'immagine renderizzata l'alone di luce, determinato applicando un filtro di smoothing ai punti che causano l'effetto ottico. Questi possono essere specificati da chi modella la scena, attraverso una texture map, oppure autonomamente determinati. Si tratta quindi di tre fasi:

- individuazione punti bloom
- smoothing
- somma immagini

Nel motore grafico l'individuazione dei punti che causano il bloom è automatica, e consiste nel selezionare i punti più luminosi, che si suppone caratterizzino delle sorgenti di luce intensa, dell'immagine finale. Si tratta quindi di un effetto post-process. La luminosità di un pixel è calcolata con una formula che deriva dall'ottica:

$$L = Color \cdot < 0.8;\ 0.587;\ 0.144 >$$

dove color è il vettore delle tre componenti RGB del pixel. Nel renderer sviluppato, uno shader applica questa formula a tutti i pixel dell'immagine renderizzata secondo un kernel di

dimensione 4x4. Al valore medio, ottenuto dividendo per la dimensione del kernel, è applicata la formula di luminosità. Una soglia minima è richiesta per considerare il pixel come sorgente di luce. Infine un parametro, bloomStrength, è usato per determinare l'intensità globale del bloom. Il codice seguente, tratto dallo shader del motore grafico, esegue quest'ultima parte:

```
color *= 0.0625;
float luminance = dot (color, float4(0.3, 0.56, 0.2, 0));
float fragmentWeight = luminance - bloomLumaThresold;
if (fragmentWeight > 0)
      Out.Value = color * bloomStrength * fragmentWeight;
else
      Out.Value = 0;
```

L'immagine prodotta presenta però delle componenti ad alta frequenza, poco verosimili perchè non variano omogeneamente. Per migliorare la resa è applicato un filtro passa basso, di smoothing, con un kernel 5x5. La figure seguenti illustrano quanto esposto:

Figura 7.18: dettaglio di tre luci senza effetto bloom

Figura 7.19: stesso dettaglio dei figura 7.18, ma con bloom attivo. Si nota il bagliore attorno alle luci

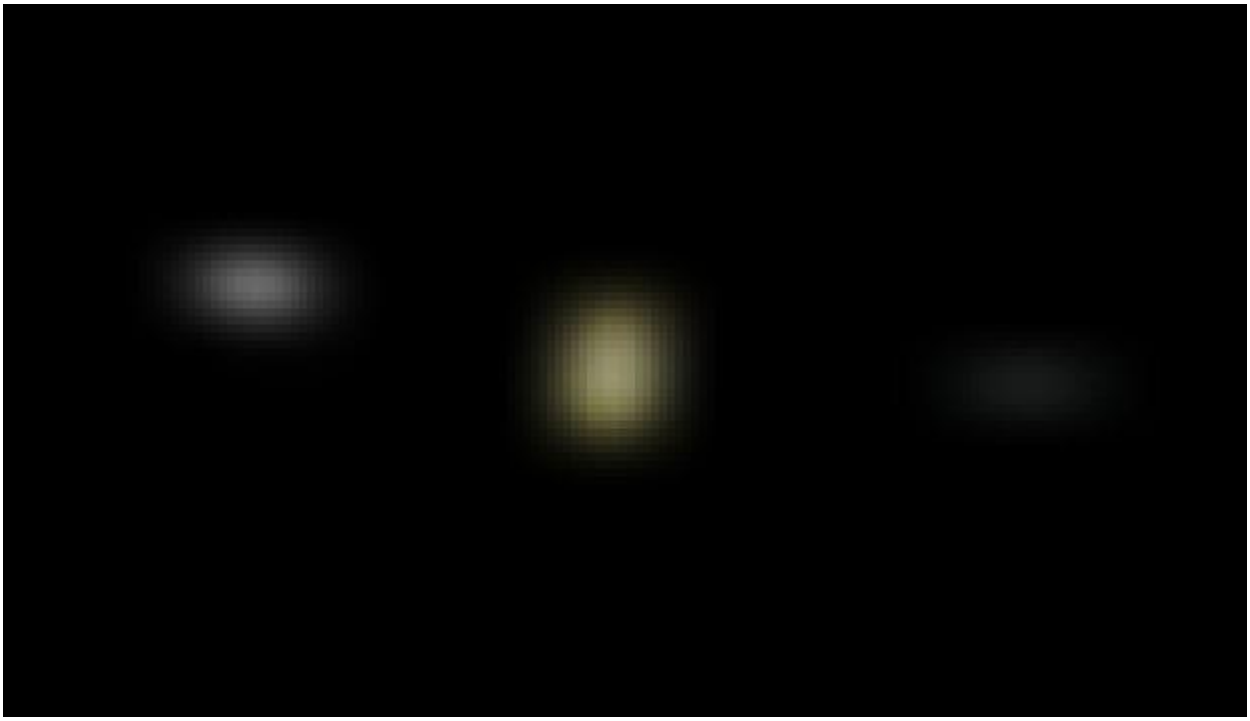


Figura 7.20: bloom buffer prodotto dal renderer sviluppato per la scena di figura 7.19

7.8 Motion Blur

Il motion blur è l'effetto che accade durante la ripresa di una scena quando un oggetto inquadrato è tanto veloce che la sua immagine cambia durante la registrazione di un singolo frame. Fa parte degli effetti detti cinematici, perchè simulano il comportamento delle telecamere reali. Quando l'otturatore è aperto e l'oggetto di sposta velocemente, ogni punto del CCD è impresso con più punto dell'oggetto, allineati alla direzione della velocità. Per simulare il motion blur esiste un approccio basato su una operazione analoga, componendo il colore del pixel finale con i colori dei pixel del frame buffer allineati secondo la sua velocità. L'algoritmo si compone quindi di due passaggi:

- determinazione della per-pixel velocity
- per ogni pixel, campionamento del frame buffer secondo il vettore velocità

Nel motore grafico è stato scelto un approccio completamente diverso, in quanto non c'era abbastanza spazio nel GBuffer per scrivere un vettore velocità senza dover aggiungere un nuovo layer, che avrebbe appesantito il renderer. Esiste infatti un approccio alternativo, che consiste nell'interpolare il frame buffer appena generato con quello del frame precedente, secondo un parametro che ha range [0..1] che determina l'entità del motion blur. In questo modo un pixel sarà composto dal suo attuale colore e dal colore assunto nelmomento precedente, simulando l'effetto di eccessivo tempo di apertura dell'otturatore. In questo caso diventa sufficiente uno scalare per la velocità, che nel GBuffer è salvato nel canale alpha dell'albedo buffer.

La velocità è calcolata per vertice, ed interpolata lungo la superficie. Il pixel shader calcola la velocità partendo dalla posizione attuale e precedente in screen space. La vecchia posizione è determinata attraverso una matrice, mantenuta da ogni entità, che mantiene la posizione e l'orientamento del frame precedente. Il vertice moltiplicato per questa matrice da la posizione spaziale cercata. Queste due righe dello shader compiono quanto appena esposto:

```
float4 prevPosition = mul(mPrevLocalWorldViewProj, In.Position)
float fragmentVelocity = length(prevPosition - Out.Position) * motionBlurAmount
```

dove il parametro motionBlurAmount specifica l'ammontare globale dell'effetto. Nel motore grafico l'interpolazione con il frame precedente è attuata mediante alpha blending nel material pass.

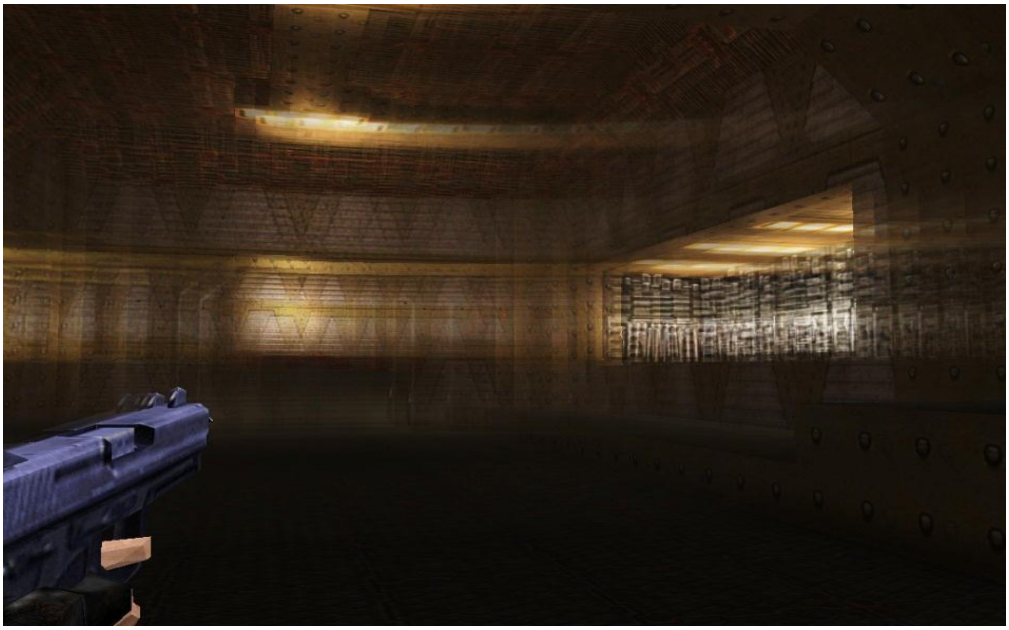


Figura 7.21: effetto motion blur. Si nota che non è applicato all'arma in primo piano

8. Partizionamento Spaziale

Il partizionamento spaziale è essenzialmente una tecnica che permette di ottimizzare notevolmente la gestione degli oggetti del mondo 3D, permettendo di raggrupparli e definire delle regole che possano correlare la loro collocazione rispetto ad altre entità o all'osservatore.

8.1 Introduzione

A questo punto, infatti, la scena è una lista di entità, le quali dispongono di una mesh, ovvero una collezione di vertici opportunamente connessi. Non c'è però alcuna struttura che possa correlare gli oggetti l'uno con l'altro o legarli a qualche proprietà spaziale. Il partizionamento dello spazio è tale per cui si possono definire dei rapporti tra i vari oggetti presenti, spesso, e potenzialmente, in relazione alla posizione dell'osservatore e in relazione alla loro collocazione.

Sinteticamente tale tecnica ha lo scopo, più che di introdurre nella pipeline nuove potenzialità prima impossibili, di *accelerare* molte operazioni. Un punto cardine di questa tecnica è che permette di avere strutture *gerarchiche* in virtù delle quali, una volta identificata una proprietà generale di un nodo, è possibile *estendere* tale proprietà a tutti i figli di quel nodo. Ad esempio, se un'entità edificio è invisibile, allora lo saranno anche tutte le stanze interne, le quali saranno sistematicamente ignorate, facendo risparmiare dapprima ai test di occlusion, e poi eventualmente al render, un numero alto di poligoni.
Generalmente, quindi, è usato per due adempiere a due scopi:

1. eliminare dalla rendering pipeline tutti gli oggetti non visibili in modo *veloce.*
2. accelerare le operazioni inter-entità svolte dal modulo fisico, in quanto è ora possibile determinare velocemente quali entità potenzialmente possono interessare la dinamica di un'altra entità.

Il partizionamento può essere svolto a livello di entità, o a livello di poligoni della mesh. Quest'ultima modalità era diffusa nei motori grafici di vecchia generazione, quando era indispensabile minimizzare al massimo il numero di poligoni inviati al renderer. Oggi questo livello di dettaglio è usato solamente nella gestione della fisica, mentre si tende ad ignorare questo livello di suddivisione a livello di rendering, in quanto le moderne schede video possono peremettersi di renderizzare un numero di poligoni inutili senza risentirne troppo in velocità. Pertanto nel motore grafico sviluppato sono stati adoperati due approcci diversi, uno per il renderer e uno per la fisica. Nel renderer la suddivisione è molto approssimata, mentre nella fisica è dettagliata a livello di singoli poligoni.

Esistono quattro metodi diffusi di partizionamento: celle, alberi BSP, PVS, Octree.

8.1.1 Celle

Il partizionamento a celle è il metodo più semplice, e consiste nel suddividere l'intero spazio in tante aree cubiche di dimensione fissa, tutte adiacenti, associando ad ognuna di esse i poligoni che risultano interni. A questo punto è sufficiente fare un test conservativo di visibilità di una cella per scoprire se i suoi poligoni siano visibili o meno.

Il partizionamento a celle permette di eliminare dalla pipeline molti poligoni esterni al viewing frustum con estrema rapidità, diminuendo il peso computazionale che graverebbe sul processore.

8.1.2 Alberi BSP

Era forse l'algoritmo più usato, responsabile del successo di tante applicazioni real-time, mentre oggi ha perso parte della sua utilità a causa dell'incremento delle prestazioni delle CPU e delle GPU. L'albero BSP permette di ottenere molto velocemente una lista perfettamente ordinata dei poligoni, dal più vicino al più lontano o viceversa, relativamente ad una posizione spaziale qualsiasi. Questo era un forte vantaggio per i vecchi renderer ma, come già osservato, non è più necessario sottostare a questo vincolo per avere frame rate real-time.

8.1.3 PVS

Il *Potentially Visibile Set* (PVS) è una tecnica che permette di *precalcolare* i poligoni visibili dentro una determinata area prestabilità, per poi richiamare la lista in tempi velocissimi. Pertanto necessita che il modello venga in qualche modo pre-compilato, e soprattutto *non permette* il dinamismo agli oggetti, perché qualora uno di questi cambiasse posizione sarebbe necessari ricalcolare il PVS nel quale esso giace.

8.1.4 Octree

L'octree è una struttura dati ad albero con otti figli per nodo, che rappresenta un'ottimizzazione del partizionamento a celle. É usata nel motore grafico e verrà illustrata nel prossimo paragrafo.

8.2 Octree

L'octree è una struttura dati ad albero, che rappresenta un'ottimizzazione del partizionamento a celle e che risulta molo interessante perchè permette di creare abbastanza facilmente una relazione *gerarchica* fra le entità della scena. I vantaggi sono molti, e vanno dall'*Hidden Surface Removal* (HSR) all'ottimizzazione delle collisioni.

Nell'octree, un volume di spazio (radice dell'albero) è suddiviso in otto volumi cubici (sebbene sia possibile usare dei parallelepipedi arbitrari) della stessa dimensione, ricorsivamente, in modo che ognuno dei volumi partizionati possa essere ulteriormente suddiviso in altri otto volumi. Ogni volume è assegnato ad un nodo. Ogni nodo può avere otto nodi figlio o essere una foglia. La procedura si suddivisione continua fino a quando non è soddisfatto qualche criterio, che può essere:

- raggiunta la dimensione minima per un volume
- non raggiunto il numero massimo di oggetti contenuti (entità o poligoni)

É evidente come sia possibile stabilire una gerarchia di proprietà che interessa i nodi verticalmente. Se ad esempio un nodo è esterno al campo di vista, sicuramente lo saranno anche

tutti i nodi figli. Se un nodo non collide con una entità, allora nessuna entità contenuta nel nodo colliderà con l'entità in esame. Un altro vantaggio dell'octree è che permette di gestire oggetti anche se sono in movimento nella scena, ovvero la struttura dell'octree può essere aggiornata ad ogni frame affinchè sia sempre coerente con il contenuto della scena. Ciò rilassa il vincolo di molte struttura di partizionamento che richiedono che gli elementi gestiti siano statici.

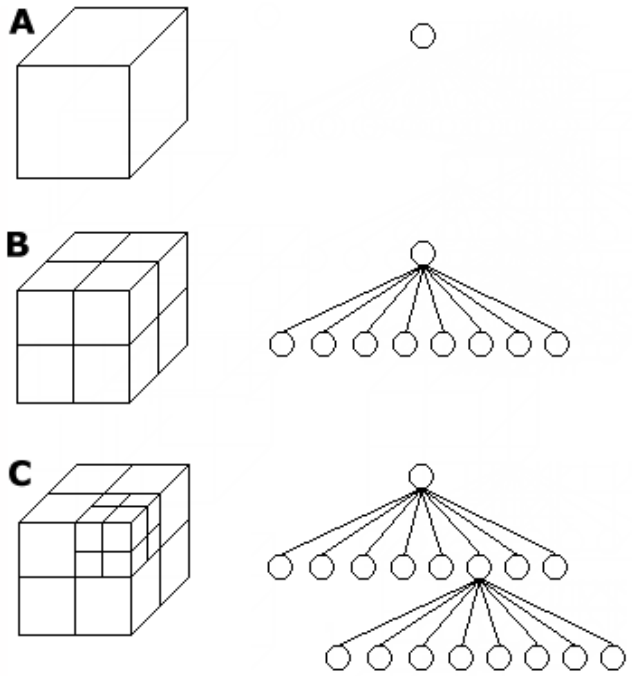


Figura 8.1: Sviluppo di un Octree. Ad ogni nodo è associata un'area in genere cubica.

Lo scopo dell'octree nel motore grafico è permettere di:

A. individuare velocemente gli oggetti interni ad un volume di spazio specificato, o prossimi ad un punto dello spazio specificato
B. eseguire dei test sui nodi i cui risultati valgono anche i nodi figli e quindi le entità ivi contenute
C. ottenere una lista di entità ordinate secondo la distanza da un punto specificato
D. accellerare il ray-casting

Le proprietà *A*, *B* e *D* sono sfruttate dal modulo fisico. Le proprietà *B* e *C* sono sfruttate dal modulo renderer.

8.2 Implementazione

Nel motore grafico un octree può essere creato per il render socket e per il physics socket. Pertanto fino a due octree possono essere creati per ogni entità.

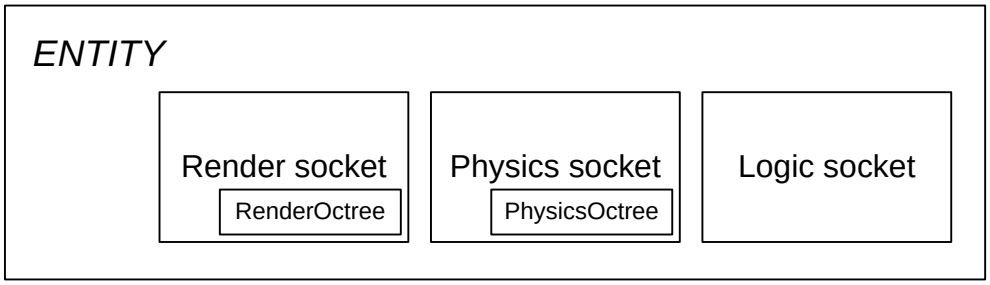


Figura 8.2: un octree può essere assegnato al render socket e al physics socket

Questa scelta è dovuta al fatto che entrambi i moduli possono necessitare di un octree, ma di caratteristiche e scopi diversi. Le differenze più importanti sono evidenziate nella figura 8.3, che descrive le classi. I metodi *FillLeaf*, *Build* e *Delete*, esposti dalla classe base, sono virtuali e quindi specifici delle classi derivate.

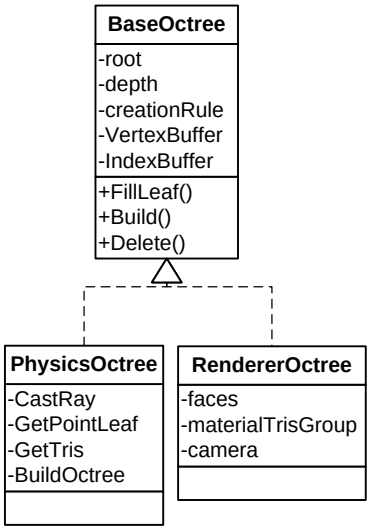


Figura 8.3: i due tipi di octree derivano da una classe base

Attualmente non è prevista alcuna struttura globale di partizionamento spaziale, cioè per l'intera scena. Questa scelta è dovuta al fatto che per quanto riguarda le collisioni, gran parte di esse riguardano la mappa statica, ovvero l'entità worldspawn, che è anche l'entità con il più elevato numero di triangoli e per la quale l'octree è previsto. Per le altre entità è stato ritenuto sufficiente, per ora, basarsi sul veloce test fra i bounding-ellipsoids. In seguito sarà specificato come sono calcolate le collisioni (fra solidi e fra raggi).

L'Octree è una struttura dati ad *albero* nel quale ad ogni nodo è associata un *volume*. Tale volume nell'engine è stato scelto di forma cubica poiché in questo modo è possibile sfruttare il bounding-sphere per una rapida verifica conservativa dell'estraneità del nodo rispetto a qualche oggetto. Questo sarà il viewing frustum nel caso sia necessario sapere se un nodo è visibile, o una entità nel caso delle collisioni. I nodi interni hanno tutti otto figli, detti *octant*, poiché ogni cubo viene suddiviso in otto cubi più piccoli, eccetto le foglie, che rappresentano un volume cubico non ulteriormente suddivisibile. La radice dell'abero rappresenta l'intera mesh, mentre i figli rappresentano le successive suddivisioni spaziali costruite secondo il criterio scelto. Sia i nodi interni che foglia contengono la lista dei triangoli interni o parzialmente interni. Ogni nodo ha un riferimento anche al padre. La posizione di un nodo è definita come la posizione al punto centrale del cubo.

Per identificare in modo chiaro e ordinato gli otto figli di ogni nodo interno, si usa la cosiddetta notazione per octree ZYX. Secondo questa convenzione, l'indice (numerico) di ogni figlio corrisponde alla sua collocazione nel volume suddivisa, secondo il numero ottenuto agendo sui tre bit del numero binario ZYX. In questa, ogni bit indica se il nodo figlio ha la coordinata, relativa all'asse identificato dal bit, minore o maggiore a quella media (centrale) del padre.

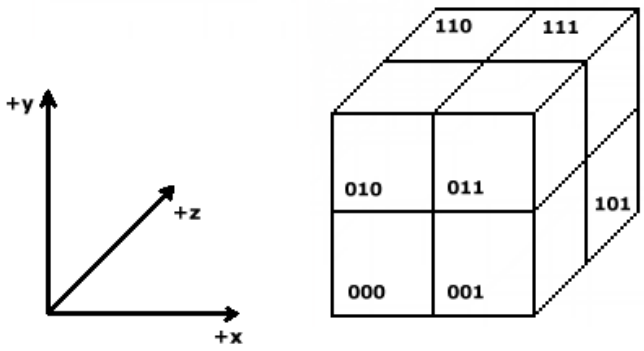


***Figura 8.4: Si usa la convenzione ZYX per identificare gli otto figli. Essa è basata sulla posizione dei cubi* figli nei confronti del cento del cubo padre.**

Nella figura, il cubo piccolo *000*, in basso a sinistra, ha una posizione tale che la coordinata z è minore di quella centrale del padre (cubo grande), pertanto il suo bit sarà pari a zero. Il cubo figlio *110* è tale perché il valore *z* è maggiore (più in fondo) rispetto al valore del padre, proprietà che vale anche per le *y*, essendo il cubo *110* collocato in alto rispetto al centro del padre.

In questo modo:

- variando un bit si ottengono nodi aventi un piano in comune
- variando due bit si ottengono nodi aventi una retta in comune
- variando tre bit si ottiene il nodo avente un solo punto in comune

Tale proprietà è fondamentale per la visita dell'albero in grado di restituire una lista parzialmente ordinata di tutti i poligoni visibili. Questa è la struttura di un nodo octree:

```
struct ONode
{
      ONode*      pChild;
      ONode*      pFather;
      uint8       level;
      vec3f       center;
      vec3f       vmin;
      vec3f       vmax;
      float       sphereRadius;

      int32array  trisList;
      int32array  trisInside;

      void *pData;
}
```

É presente un puntatore *void *pData* nella definizione di un nodo octree, che permette al metodo *FillLeaf* implementato nella classe derivata di allegare una quantità specifica di informazioni. *sphereRadius* è il raggio della bounding-sphere, mentre *trisList* è un array di interi che contiene gli indici dei triangoli contenuti.

8.3 Costruzione

Al nodo radide è assegnato il set di triangoli della mesh. Il processo di creazione è basato su un processo ricorsivo che può così essere riassunto:

- Si definisce un cubo che ingloba l'intera mesh, ottenuto passando in rassegna la lista di vertici, tenendo traccia degli estremi. Tale cubo sarà associato alla radice dell'albero octree.
- Si procede ricorsivamente seguendo questi passi per ogni nodo:
    - si aggiorna la lista di poligoni contenuti nel volume del nodo
    - si conta il numero di poligoni interni
    - viene richiamata *FillLeaf* per riempire il nodo con le informazioni *specifiche*
    - si stabilisce la rispondenza al *criterio* scelto
        - numero massimo superato
        - dimensione minima non raggiunta

- se il criterio è verificato, il volume viene suddiviso e per ogni nodo è rieseguito il processo.

All'atto di dividere un nodo in otto parti, ci si pone il problema dei poligoni che diventeranno parte di più figli. È il caso di poligoni che intersecano i piani di divisione del nodo. Le scelte possibili sono due:

- il poligono viene tagliato dal piano di divisione del nodo padre
- il poligono viene fatto entrare nella lista di tutti i nodi figli interessati

La prima soluzione è stata scelta nel motore grafico. Sono ora introdotte le due implementazioni.

8.4 Renderer Octree

Lo scopo dell'octree usato dal renderer è:

A. ottenere un approssimativo ordinamento dei poligoni
B. individuare velocemente i poligoni esterni al frustum

Siccome le GPU gesticono velocemente un alto numero di triangoli, le soglie in questo caso sono molto alte e ogni foglia contiene mediamente migliaia o decine di migliaia di poligoni. Questo octree si occupa non solo di raggruppare i triangoli della mesh all'interno dei nodi, ma anche di raggruppare ulteriormente i triangoli secondo il tipo di materiale. Questo è dovuto al fatto che i materiali sono specificati prima della chiamata alla draw call, e quindi per avere il minor numero di chiamate possibili i triangoli vengono partizionati secondo il materiale. Il rendering della mesh è effettuato attraverso la visita dell'octree (che soddisfa al contempo i punti A e B) ed è illustrata nel paragrafo seguente.
Questa è la struttura che il RenderOctree appende al nodo octree:

```
struct RONodeData
{
      struct materialIB
```

```
        {
              uint32     startIBIdx
              uint32     endIBIdx;
              uint32     maxIndex
              uint32     minIndex;
              SMaterial* material
              CAABB      aabb
              LPD3DIB9   DIB;
        };
        List<materialIB> MIB;
        materialIB &GetAllTriangles () { return MIB[0]; }
  }
```

La struttura *materialIB* identifica un gruppo di triangoli dello stesso materiale. Un array di *materialIB* contiene i set di triangoli, dove la posizione zero contiene invece la lista totale, ritornata da *GetAllTriangles()*.

8.4.1 Visita

SCOPO

La visita dell'octree può ritornare una lista *parzialmente* ordinata di poligoni, ed è usata primariamente del renderer octree. Il termine parzialmente è usato poiché l'ordinamento è effettuato nei confronti dei nodi dell'octree e non rispetto ai loro poligoni interni. La visita dell'albero segue un ordine *front to back*, ed ogni volta che un nodo è determinato essere esterno al campo di vista, viene ignorato.

MODUS OPERANDI

L'algoritmo ricorsivo della visita dell'octree si basa sull'idea si partire dalla radice e raggiungere il nodo più piccolo al cui interno giace l'osservatore, e da quello visitare tutti i nodi sempre meno adiacenti, ovvero partendo da quelli con un piano in comune terminando con quelli con un solo punto in comune. Per quanto riguarda un nodo, la visita, a livello geometrico, può essere così schematizzata:

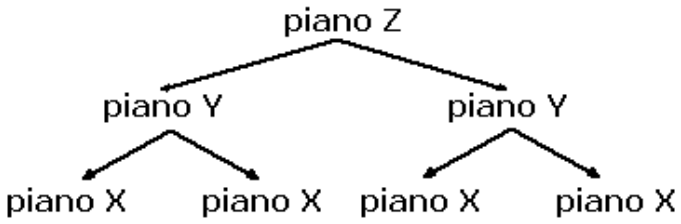


Figura 8.5: La visita è eseguita percorrendo una precisa suddivisione dello spazio rispetto ai piani Z, Y e X

All'inizio si hanno a disposizione otto figli. Si considera un piano, in questo caso il piano Z, e si controlla la posizione dell'osservatore rispetto a quel piano. Si considerano conseguentemente ora solo i nodi da quella parte del piano, che sono quattro. Ora si procede analogamente, in figura con il piano Y, rimanendo con due nodi. Infine si procede per il piano X. Una volta visitato un nodo, si ritorna indietro per visitare gli altri sette fratelli, seguendo sempre lo stesso schema. La scelta del nodo da visitare è fata settando ogni bit di *ZYX* in funzione della posizione rispetto al piano.

L'algoritmo usato nell'engine può essere così riassunto, dove *oss* è la posizione dell'osservatore e *nodo* è usato per ottenere la posizione del punto centrale dell'area rappresentata dal nodo:

- *oss.x > nodo.mx ? Se si, X = 1, altrimenti X = 0*
- *oss.y > nodo.my ? Se si, Y = 1, altrimenti Y = 0*
- *oss.z > nodo.mz ? Se si, Z = 1, altrimenti Z = 0*
- *si compone il numero n = ZYX*
- *si visita il figlio[n]*
- *si visitano i tre figli con un piano in comune a figlio[n], invertendo una alla volta i bit di ogni piano*
- *si visitano i tre figli con una retta in comune a figlio[n], invertendo due bit alla volta*
- *si visitano l'ultimo figlio rimasto, con un solo punto in comune, invertendo i tre bit*

Si evince che l'ordine *back to front* è ottenuto partendo dai nodi più vicini all'osservatore procedendo verso quelli più lontani.

8.5 Physics Octree

Lo scopo del *PhysicsOctree* è permettere al modulo fisico di calcolare molto velocemente le collisioni. A tal proposito la soglia del numero di triangoli massimo è molto ridotta, ed è sull'ordine della decina di triangoli per foglia, valore che è risultato offrire i minori tempi di risposta. Conseguentemente, anche le dimensioni dei volumi sono molto più piccole di quelle tipiche del *RendererOctree*.

Nel motore grafico esistono due tipi di collisioni:

A. ***bounding volume / bounding volume***

Sono usate per gestire la normale fisica degli oggetti, vale a dire le interazioni fra oggetti solidi. In questo caso lo scopo dell'octree è, data un volume nello spazio (un elissoide nell'implementazione), ricavare tutti i triangoli prossimi ad esso. Questi saranno usati poi dal sistema di collisioni per determinare che cosa una entità sta toccando.
L'algoritmo che è stato sviluppato per l'engine decompone l'elissoide il cui volume è richiesto di considerare, in tanti cubi la cui dimensione è la minima dimensione dei nodi presenti nell'octree. Ovvero, il volume richiesto è discretizzato secondo la risoluzione massima dell'octree. In seguito, sono ricercati tutti i nodi al cui interno giace il centro di ogni volume ricavato, e considerata la lista dei triangoli. Nella figura 8.6, l'ellisse è discretizzato secondo la risoluzione dell'octree, e partendo dal punto **A** (vertice con coordinate minime) fino al punto **B** (vertice con coordinate massime) vengono considerati tutti i nodi racchiusi. Questo sono gli step:

a. *determina punto di partenza*
b. *determina step X-Y-Z*
c. *per X, per Y, per Z*
    i. *ricava nodo*
    ii. *ricava lista triangoli*
    iii. *merge con la lista corrente*

iv. *incrementa step*

d. *torna al punto c*

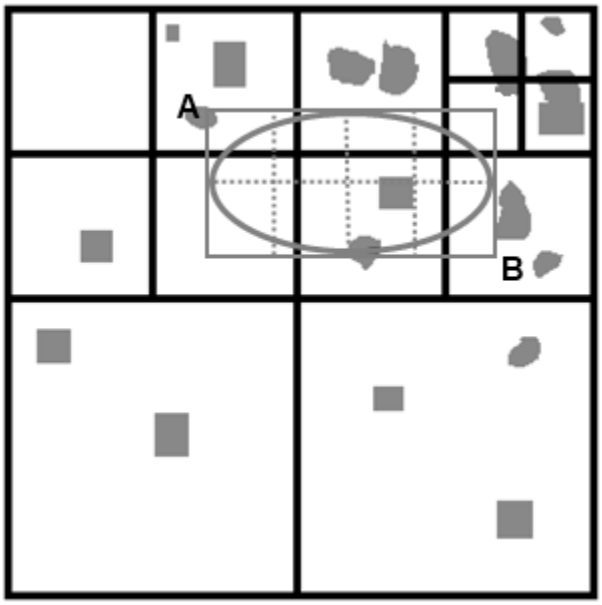


Figura 8.6: partendo dal punto A fino al punto B vengono considerati tutti i nodi rinchiusi nel volume

La funzione:

```
void CPhysicsOctree::GetTris(SMeshTrisList &trisList, CEllipsoid *ellipsoid)
```

esegue questo algoritmo ed è primariamente usata dal modulo fisico per ottenere tutti i triangoli prossimi ad una entità, il cui bounding ellipsoid è passato come parametro (in genere per quanto riguarda l'entità worldspawn). Si ricorda nuovamente che non esiste ancora una struttura gerarchica per la gestione delle collisioni quando una delle entità è worldspawn (mappa statica). Il modulo fisico è ottimizzato per evitare di ricalcolare nuovamente la lista dei triangoli quando entrambe le entità non hanno subito trasformazioni nell'ultimo frame. Quindi:

Entità A	Entità B	Metodo
generica	worldspawn	si ottengono i triangoli con l'algoritmo appena esposto, e si esegue una procedura di collisione bounding_ellipsoid/lista_triangoli
generica	generica	vengono tutte considerate tutte le entità con un test molto veloce sfera-sfera. Se c'è collisione, viene eseguito il test ellipsoid(A) con AABB(B)

B. ***raggio / bounding volume***

Servono a molteplici scopi nel motore grafico, ad esempio per calcolare se un oggetto è visibile da un osservatore, o per calcolare che cosa un proiettile colpisce. Differentemente dal caso precedente, questo algoritmo ha come input sia i nodi che i triangoli, non solamente i triangoli come nel caso precedente. Per questa ragione, l'algoritmo è scritto internamente nell'octree ed è richiamato dal metodo *CastRay*:

```
void CPhysicsOctree::CastRay(SCollision &result, SRay *ray)
```

L'algoritmo si avvale dell'octree in quanto vengono di volta in volta calcolati i nodi foglia in cui il raggio passa, dal punto di partenza. Per ogni nodo, viene calcolata l'intersezione con ogni triangolo contenuto. Siccome lo stesso triangolo può appartenere a più nodi, i triangoli vengono marcati per evitare di eseguire il test di intersezione più volte. La figura 8.7 mostra una situazione di esempio.

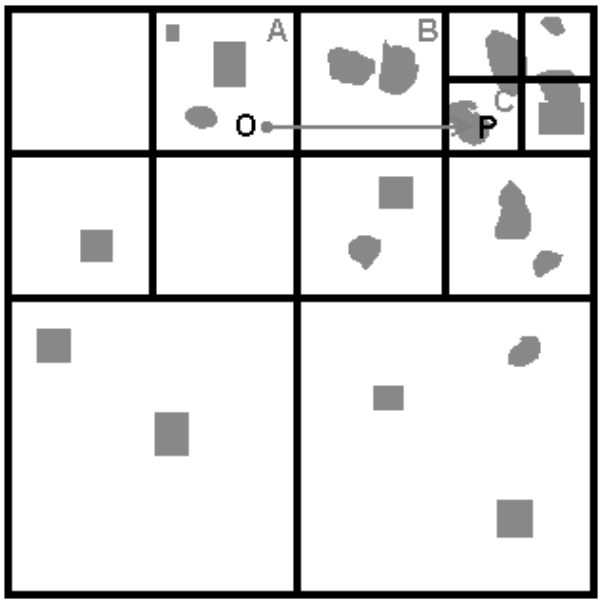


Figura 8.7: algoritmo di ray-tracing che sfrutta l'octree. Solo i triangoli prossimi al raggio vengono considerati

Il raggio è lanciato dal punto **O** ed è descritto dall'equazione:

$$r(t) = \boldsymbol{O} + \boldsymbol{D} \cdot t$$

L'algoritmo individua il nodo foglia che contiene il punto O, che in questo caso è il nodo *A*. Il test triangolo/raggio è eseguito per tutti i triangoli interni al nodo. Nessuna collisione è individuata, quindi è calcolata l'intersezione del raggio con l'AABB del nodo *A*. In seguito il parametro *t* dell'equazione è incrementato di un valore molto basso (il raggio è spostato in avanti), in modo da portare il raggio al nodo adiacente, cioè il nodo *B*. A questo punto l'algoritmo si ripete.
Questi sono gli step:

a. p = ray.start
b. nodo = GetPointLeaf(p)
c. per ogni triangolo $\subset$ nodo
    i. triangolo già controllato ?
    ii. test triangolo/raggio. Mantieni collisione con *t* minore
d. collisione avvenuta ?

i. se si, termina
ii. se no:
    1. esegui intersezione AABB/raggio
    2. incrementa *t* (t = 1.001t)
    3. calcola p = ray.start + ray.D · t
    4. torna al punto *b*

L'algoritmo accelera il processo in quanto solamente i triangoli prossimi al percorso del raggio vengono controllati. Nel motore grafico un valore di triangoli medio compreso fra 5 e 20 è risultato ottimale.

9. Engine Physics

La fisica dell'engine è interamente gestita dal modulo physics, avvalendosi dei physic sockets delle entità. Esso si occupa di aggiornare le proprietà fisiche delle entità (forze, posizione, angolazione) evitando situazioni impossibili, ovvero entità i cui modelli fisici penetrano negli altri. Si fa notare che non è stato scritto un vero e proprio sistema fisico, in quanto le possibilità sono piuttosto limitate e oggigiorno sono disponibili molte librerie commerciali molto performanti. Tuttavia è stato scritto da zero un sistema che permette di gestire le collisioni in un alto numero di entità in tempi accettabili. Il modulo fisico è introdotto in questo capitolo.

9.1 Bounding Volume

I bounding volumes sono dei modelli matematici che approssimano la reale geometria di un oggetto. Vengono usati in quanto la gestione delle collisioni fra essi è estremamente più semplice che la gestione a livello di singoli triangoli. Inoltre, i risultati che si ottengono sono, se ben implementati, molto convincenti. In figura %%% sono illustrati i tre più usati (e semplici) bounding volume: *AABB, Ellipsoid, Sphere*.

Figura 9.1: bounding sphere, ellipsoid e AABB.

9.1.1 Sphere

É in assoluto il bounding volume più semplice, essendo il più approssimativo e il più veloce da calcolare. É spesso usato come test preliminare, in quanto il test di collisione di due sfere si riduce ad un calcolo di distanza dei rispettivi centri. Una sfera è definita dal raggio e dal centro.

```
collision = (centerA – centerB).Length() < radiusA + radiusB
```

La derivazione della sfera da una mesh è molto semplice. Si calcola anzitutti l'AABB, si trasforma l'AABB in cubo eguagliando i lati al lato maggiore ed infine si usa come raggio la distanza dal centro ad uno spigolo (diagonale).

9.1.2 AABB

L'axis alligned bounding box è un bounding volume molto usato perchè è sia la derivazione che i test di collisione sono semplici da calcolare, e spesse volte rappresenta una sufficiente approssimazione del modello. Un AABB è un parallelepipedo i cui lati sono allineati agli assi del sistema di riferimento world (non locale all'oggetto). Pertanto un AABB non può subire inclinazioni. Per definire un AABB sono sufficienti due vertici, di cui uno avrà le coordinate minori, e l'altro le coordinate maggiori.

Dati due AABB *A* e *B*, il test di collisione si riduce al controllare che almeno uno dei vertici di *A* sia contenuto nell'AABB *B*.

- Per ogni vertice $V \subset A$
    - collision = $B_{min} < V < B_{max}$

La derivazione dell'AABB è eseguita controllando tutti i vertici della mesh, tenendo traccia del più lontano dal centro di massa relativamente ai singoli assi.

- Per ogni vertice $V \subset$ Mesh
    - aggiorna X minore e maggiore
    - aggiorna Y minore e maggiore
    - aggiorna Z minore e maggiore

9.1.3 Ellipsoid

L'ellissoide presenta caratteristiche interessanti perchè spesso è una buona approssimazione di una mesh, e al contempo presenta dei metodi di collisione molto veloci. Infatti, passando dal sistema di riferimento world a quello dell'elissoide, in cui l'ellissoide diventa una sfera di raggio unitario, i metodi di collisione diventano quelli del bounding sphere. Il fatto di preferire spesso l'ellissoid ad un AABB risiede nella semplicità di effettuare lo slides tra le superfici, cosa molta gradita nei motori grafici per videogiochi. Un ellissoide è definito dal suo centro e il suo raggio, dove il raggio è un vettore a tre dimensioni.

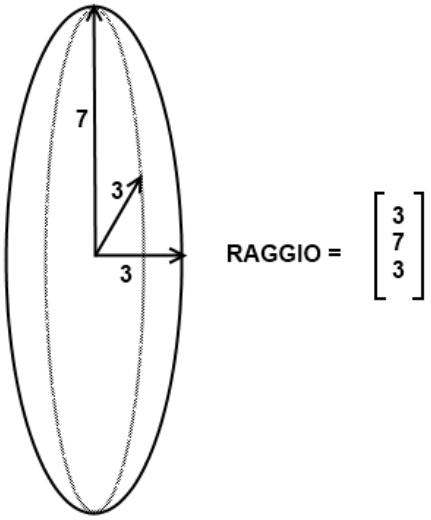


Figura 9.2: ellissoide e raggio a tre dimensioni

Per passare al sistema di riferimento dell'ellissoide è sufficiente moltiplicare un vettore per l'inverso del raggio, in modo da trasformare l'ellissoide in una sfera di raggio unitario (pari al vettore ***<1, 1, 1>)***:

$$\boldsymbol{P}_{ellipsoid} = \boldsymbol{P}_{world} \cdot \boldsymbol{I}_{radius}$$

Dopo aver trasformato il centro dell'ellissoide, i vettori velocità e i vertici di un'eventuale mesh, il metodo di collisione è lo stesso del bounding sphere.

9.2 Interfaccia IBoundingVolume

Tutti i bounding volume implementano un set di metodi necessari per gestire le collisioni. Le collisioni nel motore grafico sono descritte dalla struttura *SCollision* qui riportata:

```
struct SCollision
{
	bool	occurred;

	//  intersection
	vec3f	normal;
	vec3f	vt;

	vec3f	oldPosition;
	vec3f	vtr;
	vec3f	velocity;
	float	t;

	//  for bounding volume
	bool	crushed;
	bool	bounded[3];
	bool	groundGrip;
	bool	onEdge;
}
```

I valori più importanti sono il punto di collisione, la normale del punto di collisione e il parametro *t* del vettore velocità. Il valore *vtr* è un punto dello spazio lievemente anticipato rispetto alla collisione, usato per alcuni effetti. Il valore *crushed* è vero quando il volume è stato schiacchiato da qualche entità di massa maggiore.

Nella figura 9.3 è esposta l'interfaccia *IBoundingVolume*:

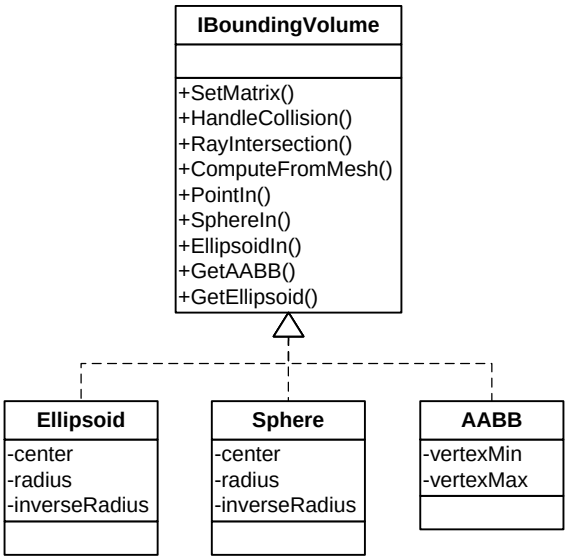


Figura 9.3: interfaccia dei bounding volumes

I metodi più importanti sono ora illustrati:

- ***PointIn***, che ha questo prototipo:

```
virtual bool PointIn(const vec3f &point) = 0
```

  Ritorna *true* se il punto dato è interno al bounding volume.

- ***SphereIn***, che ha questo prototipo:

```
virtual char SphereIn(const SSphere &sphere) = 0
```

  Ritorna un valore che indica se la sfera, rispetto al bounding volume:

    - è completamente interna
    - lo interseca
    - è completamente esterna

- ***ComputeFromMesh***, che ha questo prototipo:

```
virtual void ComputeFromMesh(evertex *VB, int count) = 0
```

Costruire il bounding volume partendo dai vertici della mesh data.

- ***RayIntersection***, che ha questo prototipo:

```
virtual bool RayIntersection(SRay &ray, SCollision &result) = 0
```

Ritorna *true* se il raggio interseca la sfera. Ritorna il punto di collisione e il valore *t* dell'equazione parametrica del raggio.

- ***SetMatrix***, che ha questo prototipo:

```
virtual void SetMatrix(matrix44f *matrix) = 0
```

Assegna la matrice che porta il bounding volume al sistema di riferimento dell'entità a cui è assegnato.

- ***HandleCollision***, che ha questo prototipo:

```
virtual char HandleCollision(SCollision &result,
                SMeshTrisList *triList, vec3f velocity,
                SMeshTrisList *usedTriList = NULL,
                bool hasMoved = false) = 0
```

Gestisce la collisione fra il bounding volume e una lista di triangoli. Questo è il metodo più complicato, e non verrà approfondito in questa tesi. L'implementazione più importante è quella della classe *Ellipsoid*, che gestisce le collisioni fra i triangoli di una mesh, e la sfera derivata dalla trasformazione dell'ellisse. Poichè gran parte delle entità dinamiche ha un bounding volume di tipo ellipsoid, questo algoritmo è

usato per gestire la fisica della maggior parte delle entità della scena. Un'entità persona che cammina, o qualsiasi oggetti dinamico, è gestito attraverso questo metodo.

L'algoritmo (qui solo introdotto) è basato sui metodi di gestione di collisioni fra sfera e triangolo, ed è di tipo CCD (*Continuous Collision Detection*). Data la sfera, per ogni triangolo si valuta:

- Collisione *sfera/piano triangolo*. É calcolata la distanza sfera/piano con la formula per la distanza *piano/punto*, e se essa è minore del raggio della sfera, si valuta se il punto è interno al triangolo. Nella figura 9.3 è illustrata una sfera di centro **P** e con velocità **V** la cui destinazione sarebbe **D**.

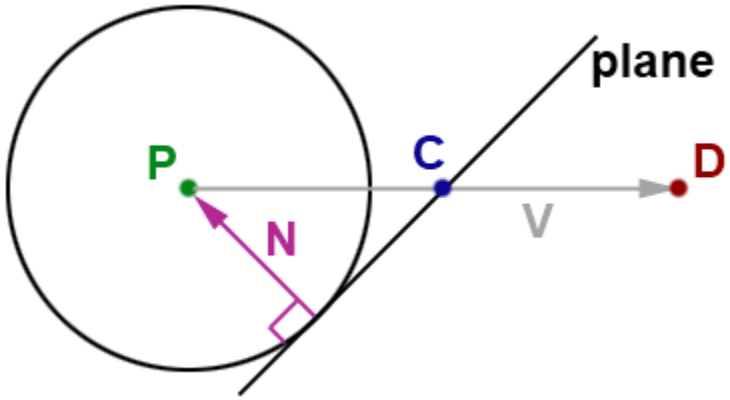


Figura 9.4: vettori interessati per il calcolo della collisione sfera/piano

Il punto **C** è determinato a seguito del calcolo della *t* per cui il vettore velocità interseca il piano, con questa formula tratta dal motore grafico (la cui derivazione non è trattata):

```
eintersection = eposition + evelocity * t - cp.plane.normal
```

A seguito di questo calcolo, la nuova posizione della sfera implementa automaticamente lo slides nelle superfici.

- Collision *sfera/lati triangolo*. Viene calcolata la distanza fra la sfera e i lati del triangolo, eventualmente calcolando un nuovo punto di intersezione (procedura non trattata in questa tesi).

- Collisione *sfera/vertici triangolo*. Analogo al caso precedente (procedura non trattata in questa tesi).

Quando questi tre vincoli sono soddisfatti, il punto di collisione individuato risulta corretto.

9.3 Ciclo di Collisioni

La gestione delle collisioni è basata sui metodi appena descritti. Il modulo fisico richiama il metodo *HandleCollision* del socket fisico di ogni entità, curandosi di passare come parametro la lista delle entità ordinate per tipologia. Per quanto riguarda le entità dinamiche (ovvero quelle che necessitano di un vero e proprio sistema di collisioni), queste vengono modellate con bounding ellipsoid, e il sistema fisico passa come parametro le entità suddivise in:

1. entità di tipo *world* , suddivise in *statiche* (es: geometria mondo) e *dinamiche* (es: ascensore). Il test di collisione segue due fasi:
   - test ellipsoid/sphere
   - test ellispoid/mesh (per ogni singolo triangolo)

2. entità di tipo *dinamico* (es: soldato). Il test è unico e a seconda del modello scelto può essere:
   - ellipsoid/AABB
   - ellipsoid/ellipsoid

Il modello fisico usato si cura di calcolare la corretta posizione eseguendo l'algoritmo esposto al paragrafo precedente più volte, fino ad un massimo di tre iterazioni. Questo è necessario perchè in certe circostanze il risultato ottenuto da una singola esecuzione potrebbe

essere non preciso. Per le entità di tipo world, l'algoritmo si avvale del physics octree per ottenere solo la lista di triangoli vicini ad un volume dato. Nel caso di entità statiche, la lista non viene ricalcolata ad ogni frame.

Figura 9.5: i soldati sono modellati come ellissoidi e camminano sul pavimento modellato con dei triangoli

Conclusioni

Lo sviluppo del motore grafico ha richiesto quasi due anni di lavoro, durante i quali sono stati approfonditi numerosi aspetti algoritmici e matematici. Numerose componenti che hanno richiesto mesi di lavoro non sono stati incluse in questa trattazione, e chi fosse interessato è invitato a rivolgersi all'autore. Fra queste menzioniamo il *Particle System*, lo *Shadow Mapping*, il *Normal e Parallax Mapping*, la gestione delle collisioni (qui solo accennata). Complessivamente questi elementi lavorano sinergicamente e hanno permesso di ottenere un motore grafico piuttosto soddisfacente, soprattutto se si considera che la mole di lavoro tipica per un sistema analogo è distribuita su team di decine di persone per anni di sviluppo. Anche se il sistema sviluppato non potrà, per ovvie ragioni (fra cui la stabilità e le performances), mai competere con i titoli commerciali, esso rappresenta sicuramente un ottimo punto di partenza verso un prodotto commerciale, e un ottimo biglietto da visita per l'autore. L'elevato numero di argomenti trattati, inoltre, rende il progetto qualcosa di didatticamente notevole e completo.

L'engine è incluso nel Cd-Rom allegato, ed è stato scritto in C++ con Visual Studio 2005. É stato usato l'SDK di Directx9c come libreria grafica.

Ringrazio tutti quanti mi sono stati vicini durante la realizzazione del progetto.
In particolar modo il mio relatore, il Prof. Ezio Stagnaro, per la sua disponibilità e fiducia nelle mie potenzialità.

Bibliografia

1. Mike McShaffry, *Game Coding Complete, Third Edition*
2. *The DirectX book, Wolfgang Engel*
3. Alan, Watt. *3D Computer Graphics third edition*
4. Sergei Savchenko. *3Dgpl's – 3D Graphics Reference*
5. Andrea Fasano. *Programmazione grafica 3D*
6. Gianluca Alberico. *Engine 3D tutorial for Personal Computer*
7. Eric Lengyel. *Mathematics For 3D Game Programming And Computer Graphics 2E*
8. Sébastien Loisel. *Zed3D – A compact reference for 3d computer graphics programming*

Siti consultati durante lo sviluppo:

www.realtimerendering.com

www.gamedev.net