DOTTORATO DI RICERCA

in

SCIENZE COMPUTAZIONALI E INFORMATICHE

Ciclo XXIII

Consorzio tra Università di Catania, Università di Napoli Federico II,
Seconda Università di Napoli, Università di Palermo, Università di Salerno

SEDE AMMINISTRATIVA: UNIVERSITÀ DI NAPOLI FEDERICO II

Diego Romano

Il problema dell'illuminazione globale per il rendering e sua soluzione in ambiente GPU

TESI DI DOTTORATO DI RICERCA

"Tu e i tuoi minori compagni avete degnamente eseguito il vostro ultimo incarico; ma vi devo adoperare in un altro simile stratagemma. Va' e conduci qui tutta la masnada che ho messo sotto la tua potestà e incitala a muoversi prestamente. Devo offrire agli sguardi di questi due giovani una qualche illusione della mia arte, ed essi l'aspettano da me"

WILLIAM SHAKESPEARE, *La Tempesta*

Nota dell'autore

Ritengo doveroso esprimere i miei più sentiti e sinceri ringraziamenti al Prof. Murli, Responsabile Scientifico per questa tesi, che nell'ultimo decennio è stato il mio mentore con sprone imperterrito, nonché fonte di ispirazione verso l'eccellenza. Devo a lui lo sviluppo delle mie competenze.

Desidero altresì ringraziare: la Prof. D'Amore, per i dettagliati, preziosi e pazienti suggerimenti nella complessa fase di formalizzazione matematica di questo lavoro; il Prof. Lapegna, per le utili indicazioni nello studio del problema dell'integrazione numerica; il Prof. Laccetti, il mio Tutor, che sin dai tempi della tesi di laurea ha sempre supportato le mie idee e le mie iniziative lavorative.

Impossibile non ringraziare le ragazze del laboratorio*: le dottoresse Rosanna Campagna, Rossella Arcucci, e Valeria Mele, che con la loro sensibilità, il loro umorismo e le loro competenze mi hanno incoraggiato nei momenti difficili del lavoro, e della vita.*

Un particolare ringraziamento va alla mia collega/amica di un decennio, colei che più di ogni altro ha condiviso con me gioie e dolori di questo mestiere: la dottoressa Livia Marcellino, con la speranza di poter continuare il nostro percorso lavorativo e di amicizia con imperituro successo.

Voglio poi esprimere la mia riconoscenza a coloro che nella sfera privata mi hanno supportato nelle scelte e mi hanno aiutato ad affrontare le difficoltà, sostenendomi durante il lungo percorso verso il riconoscimento futuro: i miei genitori. A mio padre, che sin dalla nascita continua imperterrito a spingermi verso la comprensione della verità nelle cose e nei fatti. A mia madre, che per tutta la vita ha cercato di convincermi a non demordere nei miei obiettivi, e che continua ancora ora a farlo attraverso il ricordo che ho di lei.

Vorrei rivolgere un affettuoso ringraziamento alle mie zie Annamaria, Maria Antonietta, Maria Pia, a cui devo sia la mia sensibilità nei confronti della cultura in tutte le sue forme, sia il valore nello studio che lentamente ha preso piede dentro me. Ai miei fratelli Attilio e Marco, nonché a mio cugino Dario, dico grazie per lo spirito competitivo che, inconsapevolmente, mi hanno aiutato a sviluppare nell'ambiente familiare protetto.

Vorrei concludere con un indispensabile ringraziamento agli amici più cari, quelli che mi hanno sostenuto nei momenti più bassi, ma che hanno condiviso con me alcuni dei momenti più alti: Margherita e Stefania, per il pluridecennale legame fraterno che non mi è mancato mai; Paologiorgio, che con la sua

vicinanza da lontano mi ha sempre ricordato che bisogna provarci anche se sembra impossibile; Ada, per gli intensi stimoli nell'indagine, la speculazione e la creatività.

Dedico questo lavoro ai miei nipoti lontani: Beatrice, Vincenzo, Jurriaan, Leonie, nella speranza che in un futuro lontano possano accorgersi del mio contributo, sebbene infinitesimo, al miglioramento della loro qualità di vita. Perché uno zio del Sud Italia potrebbe anche essere meglio di uno zio d'America.

Novembre 2011

Indice

1 Introduzione **8**
1.1 La sintesi di immagini, rendering e illuminazione globale . . . 8
1.2 Illuminazione locale e GPU 9
1.3 Illuminazione globale su GPU e Piano di lavoro 12

2 Il modello matematico **14**
2.1 Il problema del rendering con illuminazione globale 14
2.2 Alcuni concetti preliminari 14
2.3 L'equazione di rendering 18
2.3.1 Radianza 18
2.3.1.1 La funzione di distribuzione di riflettanza bidirezionale 20
2.3.1.2 L'equazione della Radianza 22
2.3.1.3 Flusso radiante 23
2.3.2 Potenziale 24
2.3.2.1 Flusso 24
2.4 Equazione di Kajiya 25

3 I metodi di risoluzione **28**
3.1 Classificazione per percorsi di luce 28
3.2 Soluzioni incomplete 31
3.2.1 Raytracing 31
3.2.2 Radiosity 33

3.3 Metodi stocastici 36
3.3.1 Cammini casuali dal sensore alla luce 39
3.3.2 Cammini casuali dalla luce al sensore 39
3.4 Metodi iterativi del punto fisso 40
3.4.1 Metodo delle direzioni 43

4 Implementazione su GPU **51**
4.1 Caratteristiche dell'ambiente computazionale 51
4.1.1 Cenni sull'architettura GPU 51
4.1.2 Modello di programmazione 53
4.1.2.1 Blocchi di thread 56
4.2 Parametri di valutazione delle prestazioni 58
4.3 Individuazione dei metodi con le migliori prospettive prestazionali 61
4.3.1 Raytracing 61
4.3.2 Radiosity 62
4.3.3 Metodi Stocastici 64
4.3.4 Metodo delle direzioni 64
4.3.5 Conclusioni 66
4.4 Algoritmo sequenziale ibrido 67
4.5 Algoritmo parallelo 74

5 Analisi dei risultati **77**
5.1 Qualità delle immagini ottenute con il software sequenziale . . 77
5.2 Valutazioni delle prestazioni su GPU del software parallelo . . 84
5.2.1 Occupazione 87
5.2.1.1 Percorsi di luce 87
5.2.1.2 Calcolo dei numeri random 88
5.2.1.3 Cammini casuali 88
5.2.2 Speed-up ed Efficienza 89

6 Sommario e Conclusioni **93**

6.1 Sommario 93

6.2 Conclusioni 94

6.2.1 Direzioni per ricerche future 94

Capitolo 1

Introduzione

1.1 La sintesi di immagini, rendering e illuminazione globale

L'obiettivo finale della sintesi di immagini tramite calcolatore è quello di dare all'osservatore la percezione visiva di star osservando un'immagine fotografica, anche detta foto-realistica, riprodotta sullo schermo video o su un altro dispositivo di output.

Il procedimento di generazione dell'immagine, detto *rendering*, consiste in una sequenza di operazioni che trasformano il modello matematico della scena artificiale da visualizzare, nel quale ogni oggetto è descritto attraverso le sue proprietà geometriche e fisiche, in una immagine discreta.

Per poter quindi sintetizzare una immagine foto realistica è necessario completare tre fasi distinte: modellazione degli oggetti, rendering, riproduzione dell'immagine attraverso un dispositivo.

Nella fase di modellazione si deve descrivere la scena da visualizzare, specificando gli oggetti, le sorgenti luminose, i materiali, i punti di vista. Questa descrizione include le proprietà geometriche degli oggetti, la loro posizione e l'orientamento; le proprietà ottiche dei materiali che descrivono l'interazione della luce con le superfici degli oggetti; la definizione della posizione dell'osservatore e delle luci con le loro proprietà.

Nella fase di rendering si genera l'immagine rappresentante il modello della scena. Molti sono i procedimenti disponibili, ma ad un alto livello esistono due categorie di rendering, relative ad altrettanti modelli di trasporto della luce, chiamati ***illuminazione locale*** e ***illuminazione globale***.

Nell'illuminazione locale i singoli oggetti vengono visualizzati tenendo conto unicamente della loro interazione ottica con le sorgenti luminose della scena, come se si trovassero isolati in un spazio vuoto. In tale condizione nell'immagine non sono visibili ombre proiettate, o riflessioni/rifrazioni di alcun tipo.

Nell'illuminazione globale, invece, viene simulata la propagazione dell'energia luminosa nell'ambiente della scena. In questo caso saranno visibili tutti gli effetti ottici rappresentabili utilizzando il modello matematico del trasporto della luce, come ad esempio le ombre, i riflessi, le rifrazioni.

1.2 Illuminazione locale e GPU

Sebbene sia un modello semplificato nel suo foto-realismo, l'illuminazione locale permette di visualizzare abbastanza bene le forme degli oggetti. Considerando che i tempi di rendering per il modello locale sono assai ridotti rispetto al modello globale, l'illuminazione locale è molto utilizzata nelle applicazioni interattive (real-time). Ad esempio, i software per la modellazione interattiva di una scena utilizzano la visualizzazione con illuminazione locale. Combinandola con altre tecniche di resa realistica delle immagini, l'illuminazione locale è utilizzata per il rendering nei videogiochi, nei simulatori di realtà virtuale, nel Computer Aided Design (CAD), e in tutte le applicazioni interattive di Grafica Computazionale.

Nell'ultimo decennio del secolo scorso, con la crescita di richiesta per applicazioni di visualizzazione interattiva tridimensionale, sono state realizzate apposite apparecchiature hardware per il rendering con modello di illuminazione locale. Dovendo gestire tutta la pipeline di calcolo dalla creazione del modello alla visualizzazione dell'immagine sintetica sullo schermo (Fig.1.2.1), i produttori di hardware hanno promosso lo sviluppo di interfacce di programmazione (API) per l'illuminazione locale interattiva, tra cui spiccano OpenGL e DirectX.

La pipeline per il rendering con illuminazione locale, prendendo come input la definizione del modello tramite le API, consiste in quattro passi di calcolo:

1. Trasformazioni geometriche di proiezione sul modello;
2. Creazione dei frammenti, ovvero delle porzioni di immagine rappresentanti i singoli oggetti della scena;
3. Colorazione dei frammenti;

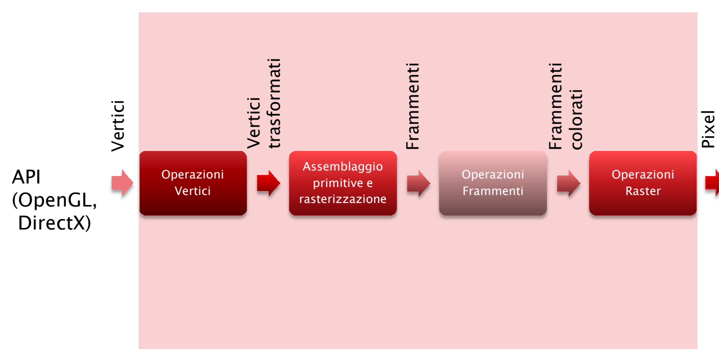

Figura 1.2.1: Pipeline per il rendering con modello di illuminazione locale

4. Composizione dei frammenti nell'immagine finale (*Raster*).

Essendo le operazioni 1,3,4 le più costose in termini di tempo di calcolo, i produttori di hardware, volendo commercializzare componenti dalle prestazioni sempre migliori, hanno introdotto nell'architettura delle schede grafiche 3D il paradigma del parallelismo a livello dei singoli passi della pipeline.

Nel 2004 la NVIDIA ha prodotto la scheda grafica GeForce 6 (Fig.1.2.2) che prevedeva sei unità di calcolo specializzate per le trasformazioni geometriche, quattro unità multi-core programmate per le operazioni sui frammenti, e sedici unità di calcolo dedicate alle operazioni di composizione dei frammenti.

Data l'eterogeneità delle possibili scene da visualizzare, il bilanciamento del carico tra le differenti unità di calcolo non era sempre garantito. Ad esempio, nelle scene con molti oggetti dalle caratteristiche ottiche semplici, il throughput della pipeline, ovvero la quantità di istruzioni eseguite in una data quantità di tempo, era determinato dalle unità specializzate per le trasformazioni geometriche. Mentre nel caso di scene con pochi oggetti ma con caratteristiche ottiche molto complesse, le unità più impegnate erano quelle dedicate alle operazioni sui frammenti.

Per questo motivo, nel 2006 NVIDIA ha introdotto nella sua scheda GeForce 8 (Fig.1.2.3) sedici unità di calcolo multi-core generiche, cioè non programma-

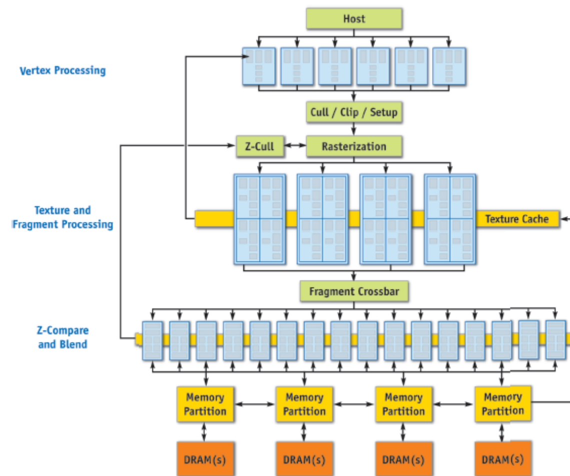

Figura 1.2.2: Schema dell'architettura della scheda GeForce 6

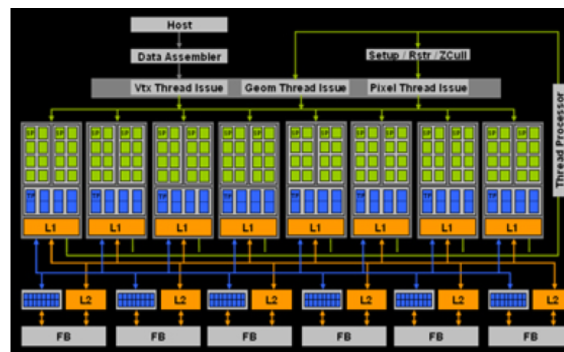

Figura 1.2.3: Schema dell'architettura della scheda GeForce 8

te per eseguire solo specifiche operazioni, ma in grado di operare in qualsiasi passo della pipeline. In tal modo il bilanciamento del carico computazionale tra le varie unità era garantito. Così è nata l'architettura delle ***Graphics Processing Units*** (***GPU***) attuali, con molte unità di calcolo a disposizione per il rendering con illuminazione locale. La soluzione di problemi di natura diversa, con quello che viene chiamato *General-Purpose computation on Graphics Processing Units* (*GPGPU*), è stata una successiva reazione della comunità del Calcolo ad Alte Prestazioni alla disponibilità di tale potenza di calcolo.

I primi esperimenti di GPGPU venivano realizzati sfruttando estensioni di OpenGL o altri linguaggi grafici, utilizzando le unità di calcolo con espedienti per interferire nella pipeline grafica, allo scopo di far eseguire alle GPU calcoli relativi ad altri problemi.

L'ultimo passo significativo nell'evoluzione delle schede grafiche è stato quello di introdurre linguaggi di programmazione indipendenti dalle esigenze della pipeline grafica. Con l'introduzione di CUDA e OpenCL, l'utilizzo delle GPU per problemi generici si è avvicinato al tradizionale paradigma di programmazione parallela, definendo quello che ora è conosciuto come *GPU Computing*.

1.3 Illuminazione globale su GPU e Piano di lavoro

Similarmente al caso dell'illuminazione locale, è possibile schematizzare il rendering con illuminazione globale attraverso una sequenza di azioni: partendo dal modello fino all'immagine.

Avendo come obiettivo la giusta illusione visiva nell'osservatore, la sensazione derivante dal guardare l'immagine artificiale dovrebbe essere approssimativamente equivalente alla percezione che si avrebbe guardando gli oggetti nel mondo reale. La percezione della scena dipende dalla funzionalità dell'occhio e dall'intensità luminosa che lo colpisce provenendo da certe direzioni; l'intensità luminosa, a sua volta, è determinata dalla *radianza* dei punti visibili. Quest'ultima, misurando la quantità di luce emessa, riflessa o trasmessa attraverso una sezione e diretta verso una direzione, dipende dalla forma e dalle proprietà ottiche degli oggetti, così come dall'intensità delle sorgenti luminose.

Il modello utilizzato per la sintesi dell'immagine deve quindi comprendere elementi per descrivere la geometria degli oggetti, le proprietà ottiche dei

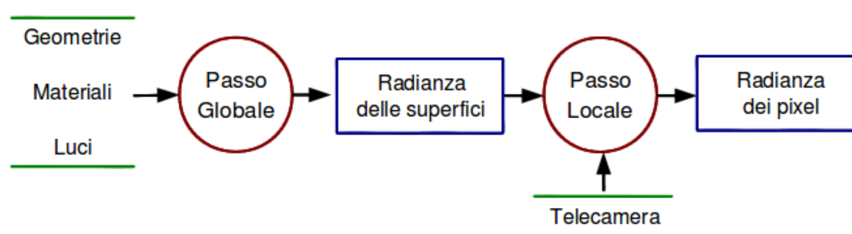

Figura 1.3.1: Flusso di dati nel rendering con illuminazione globale

materiali e le luci nella scena. Applicando le leggi della fisica a questi elementi è possibile simulare i fenomeni ottici del mondo reale allo scopo di ricostruire la distribuzione della luce nella scena. Questa operazione è chiamata ***passo globale*** del rendering, o *fase indipendente dal punto di vista*. In seguito, posizionando nella scena uno strumento di misurazione del tipo telecamera o occhio, è possibile misurare la distribuzione della luce in un punto preciso con un'orientazione specifica. Questo è quello che si chiama ***passo locale***, o *fase dipendente dal punto di vista* (Fig.1.3.1). È da notare che non tutti gli algoritmi per il rendering fanno una netta distinzione tra le due fasi, calcolando invece simultaneamente sia la distribuzione della luce, sia il suo effetto sulla telecamera.

In questa tesi si descrive il progetto di un algoritmo parallelo che sfrutta la potenza di calcolo delle GPU per realizzare il rendering con illuminazione globale con tempi di esecuzione accettabili. L'algoritmo si basa su un metodo numerico che ben si presta alle caratteristiche architetturali della GPU.

Nel Capitolo 2 verrà presentato il modello matematico che descrive i fenomeni ottici dell'illuminazione globale: ne verranno date varie formulazioni.

Il Capitolo 3 si focalizza sui metodi classici per la risoluzione del problema dell'illuminazione globale, introducendo anche un metodo iterativo basato sui percorsi di luce.

Nel Capitolo 4 verrà descritta l'implementazione dell'algoritmo prescelto, motivandone la preferenza in base alle caratteristiche della GPU.

Infine, nel Capitolo 5 verranno analizzati i risultati sia dal punto di vista qualitativo, sia dal punto di vista prestazionale.

Capitolo 2

Il modello matematico

2.1 Il problema del rendering con illuminazione globale

Eseguire il rendering foto-realistico di una scena con illuminazione globale vuol dire calcolare l'energia luminosa che raggiunge l'osservatore, supposto che questi si trovi in una posizione in cui la scena sia visibile.

Quindi, preso un punto qualsiasi della scena, è necessario calcolare l'energia luminosa che lascia il punto in direzione dell'osservatore. L'espressione di questa quantità verrà formulata dall'*equazione di rendering*.

Avendo come obiettivo la misurazione di questa energia al fine di riprodurre una immagine sintetica, è necessario interporre uno sensore discreto tra l'osservatore e la scena, ovvero uno schermo costituito da *pixel* (picture element).

Poiché l'osservatore vede infiniti punti della scena attraverso un singolo pixel, sarà necessario calcolare l'energia luminosa complessiva emessa da ogni pixel, ovvero il *flusso luminoso* che passa attraversa il pixel.

2.2 Alcuni concetti preliminari

Prima di procedere alla descrizione del modello matematico utilizzato per il rendering foto-realistico con illuminazione globale, è conveniente introdurre alcune definizioni utili per le formulazioni successive.

Poiché le scene che si vogliono visualizzare sono rappresentazioni del mondo materiale visibile all'occhio umano, d'ora innanzi ci porremo nello spazio euclideo $\mathbb{R}^3$ supponendo che valgano le leggi della meccanica classica e dell'ottica. Dato che i corpi, o oggetti materiali, occupano una porzione di spazio, o volume, decidiamo di rappresentare esclusivamente gli insiemi chiusi $C_i \subset \mathbb{R}^3$. Sia S_i la frontiera di C_i, chiameremo d'ora innanzi:

- C_i oggetto solido,
- S_i superficie dell'oggetto C_i,
- $\cup C_i$ scena (da rappresentare).

Sia $A \subset \mathbb{R}^3$, tale che $A = \cup S_i$, $\forall x \in A$, chiameremo con n_x la normale nel punto x alla superficie S_i a cui x appartiene. Si definisce altresì dA_x come l'area di superficie infinitesima intorno a x.

Volendo analizzare l'energia luminosa emessa da un punto x di una superficie S_i, al fine di descrivere le proprietà direzionali dell'emissione energetica, è utile considerare la seguente:

Definizione 2.1. Sia $x \epsilon A$, la sfera di raggio unitario Ω_x di centro x è detta *sfera d'illuminazione*.

Attraverso la sfera d'illuminazione, si può pensare di descrivere la propagazione della luce.

Definizione 2.2. Sia n_x la normale in x alla superficie S_i. Dato un sistema di coordinate sferiche, una *direzione* Θ su Ω_x è una semiretta con origine in x determinata da due angoli: la latitudine θ, formata tra Θ e n_x, e la longitudine φ, formata tra la proiezione di Θ sulla superficie ed un asse fissato a. (Fig.2.2.1)

Le superfici *trasparenti* possono emettere energia nelle direzioni individuate da tutti i punti della sfera, mentre per le superfici *opache* si ottiene la *semisfera* Ω_{H_x} che si trova dal lato *esterno* all'oggetto, ovvero tale che $\Omega_{H_x} \cap C_i = \emptyset$.

Al fine di poter interpretare la propagazione della luce a livello infinitesimale, risulta utile la seguente:

Definizione 2.3. Si definisce *angolo solido infinitesimo* la quantità $d\omega_\Theta$ definita come segue:

$$d\omega_\Theta = sin\theta \cdot d\theta d\varphi.$$

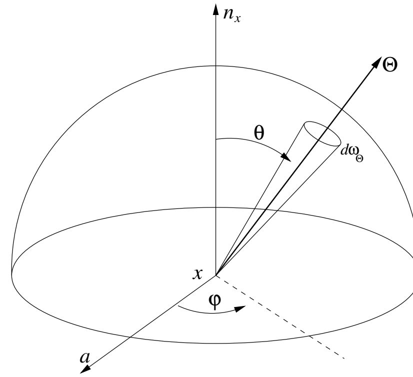

Figura 2.2.1: Sistema di coordinate sferiche (θ, φ)

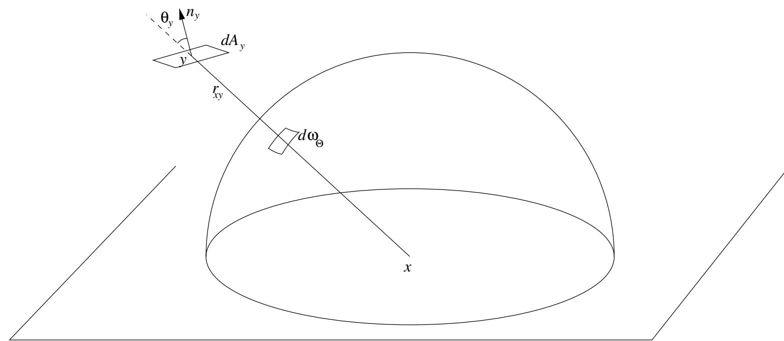

Figura 2.2.2: Angolo solido da superficie infinitesima dA_y con proiezione su sfera unitaria

Un altro modo per definire l'angolo solido infinitesimo è quello di considerare un generico punto $y\epsilon S_j$, con normale n_y, la superficie infinitesima dA_y intorno ad y (Fig.2.2.2).

Definizione 2.4. L'angolo solido sotteso da dA_y rispetto ad un punto $x\epsilon S_i$ distante $r_{\overline{xy}}$ è dato da:

$$d\omega_{\Theta} = \frac{dA_y \cdot cos\theta_y}{r_{\overline{xy}}^2} \tag{2.2.1}$$

dove θ_y coincide con l'angolo formato da n_y con la retta congiungente x e y.

Un concetto importante nella formulazione del problema di illuminazione globale è quello del ray casting.

Definizione 2.5. Si prenda una semiretta con origine in $x\epsilon A$ e direzione Θ. La funzione $r:\ A \times \Omega_x \rightarrow A$ tale che

$$r\left(x, \Theta\right) = x + t_{inf} \cdot \Theta \tag{2.2.2}$$

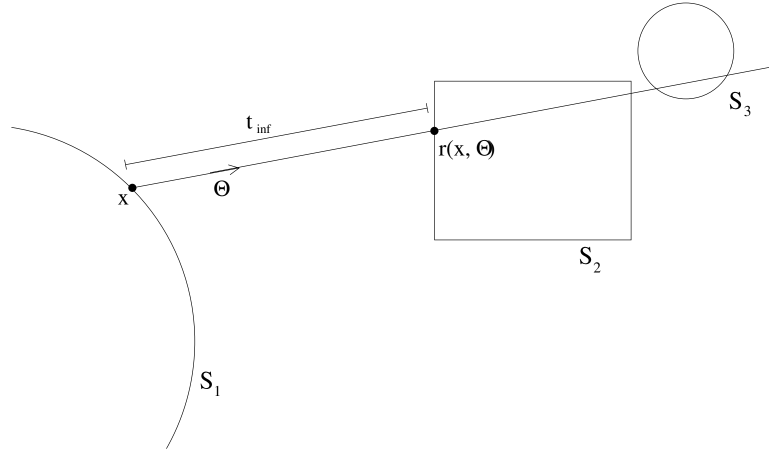

Figura 2.2.3: Funzione di ray casting in una scena con tre superfici

con

$$t_{inf} = inf \{t > 0 : (x + t \cdot \Theta) \in A\}$$

è detta *funzione di ray casting.*

La funzione r è molto utile perché determina il punto di A più vicino a x incontrato da un ipotetico raggio emesso da x verso una direzione Θ (Fig.2.2.3).

Volendo verificare l'insieme di definizione della funzione r, supponiamo che $\exists x \in A$ ed $\exists \Theta \in \Omega_x$ tale che $\forall t > 0$ si ha che $(x + t \cdot \Theta) \notin A$. In tal caso t_{inf} ed il valore di $r\,(x, \Theta)$ non sono definiti, e si dice che l'*ambiente della scena non è chiuso.* Un esempio di ambiente non chiuso è dato da una scena che riproduce una stanza con un grosso lucernario aperto: esisterà almeno un punto $x \in A$ ed una direzione $\Theta \in \Omega_x$ tale che un raggio emesso da x in direzione Θ vada verso l'infinito senza incontrare alcun altro oggetto.

Nel contesto dell'illuminazione globale, però, l'ambiente viene solitamente considerato chiuso.

Proposizione 2.1. *Sia $y = r\,(x, \Theta)$ con $x \epsilon A$. Allora si ha che*

$$x = r\,(y, -\Theta) = r\,(r\,(x, \Theta)\,, -\Theta) \qquad (2.2.3)$$

Con la finalità di poter individuare le superfici S_i coinvolte in un determinato fenomeno ottico, risulta utile introdurre il concetto di *visibilità.*

Definizione 2.6. Dati due punti $x, y \in A$, si definisce la funzione di visibilità $V : A \times A \to \{0, 1\}$ tale che $V(x, y) = 1$ se i due punti sono mutualmente visibili tra loro, ovvero se è possibile tracciare il segmento $\overline{xy}$ senza intersecare alcun altro punto di A. Altrimenti $V(x, y) = 0$.

2.3 L'equazione di rendering

Per poter giungere alla definizione del modello di rendering con illuminazione globale attraverso le leggi della radiometria, è necessario introdurre alcuni concetti di base.

L'intensità del trasferimento energetico è caratterizzato da differenti misure a seconda che si considerino o meno le proprietà posizionali e direzionali.

L'unità di base dell'energia luminosa è l'***energia radiante***. Viene denotata da Q ed è solitamente misurata in Joule.

Definizione 2.7. Il ***flusso radiante***, o flusso luminoso, Φ è l'energia radiante che attraversa (o arriva, o lascia) una superficie per unità di tempo; viene solitamente espressa in Watt ed è proporzionale al numero di fotoni che passano attraverso la frontiera nell'unità di tempo:

$$\Phi = \frac{dQ}{dt}.$$

Si possono individuare le misure di flusso incidente ed uscente rapportandole alle superfici infinitesime.

Definizione 2.8. L'***irradianza*** (E) misura il flusso incidente per unità di area superficiale. L'***emettenza radiante*** (M), anche detta ***radiosity*** (B) nel contesto dell'illuminazione globale, misura il flusso uscente per unità di area superficiale:

$$E = \frac{d\Phi}{dA} \qquad M = B = \frac{d\Phi}{dA}.$$

Quando si vuole misurare il flusso relativamente ad una direzione risulta utile la seguente definizione:

Definizione 2.9. L'***intensità radiante***, o luminosa, è il flusso radiante emesso in un angolo solido infinitesimo $d\omega$ lungo la direzione $\omega \epsilon \Omega$:

$$I = \frac{d\Phi}{d\omega}.$$

2.3.1 Radianza

Definizione 2.10. La **radianza** L è l'intensità radiante I che lascia la superficie infinitesima dA intorno al punto x, lungo la direzione ω, divisa per l'area della superficie proiettata. (Fig.2.3.1)

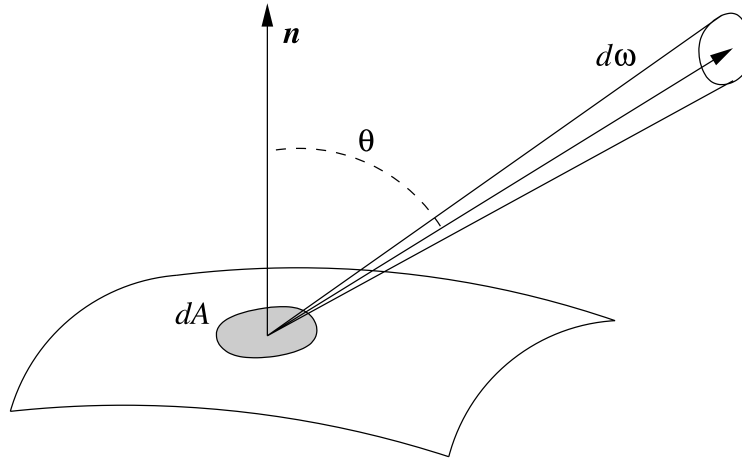

Figura 2.3.1: Area della superficie infinitesima proiettata lungo la direzione ω

Proposizione 2.2. *Se l'angolo tra la normale alla superficie e la direzione ω è θ, allora l'area proiettata è $dA \cdot cos\theta$, quindi la radianza si può esprimere come:*

$$L = \frac{d^2\Phi\left(x, dA, \omega, d\omega\right)}{d\omega} \cdot \frac{1}{dA \cdot cos\theta}. \tag{2.3.1}$$

La radianza quindi può essere interpretata come l'intensità percepita da un osservatore o da un sensore (fotocamera, videocamera,...). Infatti un osservatore non percepisce l'intensità in relazione alla grandezza dell'emettitore. Se due oggetti, uno di $10m^2$ e l'altro di $1m^2$, entrambi emettono un flusso radiante di 1000 Watt, certamente il secondo più piccolo sembrerà molto più luminoso visto che l'ammontare di energia per unità d'area sarà molto maggiore.

Poiché la radianza in un punto x è definita rispetto ad una certa direzione Θ, e può essere sia incidente sia emessa, dobbiamo introdurre una notazione che rappresenti questi concetti:

- $L\left(x, \Theta\right)$ è la radianza che lascia il punto x con direzione Θ
- $L\left(\Theta, x\right)$ è la radianza che raggiunge il punto x con direzione Θ
- $L\left(x, y\right)$ è la radianza che lascia il punto x verso il punto y
- $L\left(y, x\right)$ è la radianza che raggiunge il punto x proveniente dalla direzione del punto y

Una proprietà importante della radianza è la sua invarianza lungo percorsi rettilinei.

Proposizione 2.3. *La radianza che lascia il punto x verso il punto y è uguale alla radianza che raggiunge il punto y dalla direzione in cui si osserva il punto x.*

Questo ci permette di dire che:

Corollario 2.1. *$L(y, x)$ è la radianza che lascia il punto y verso il punto x*

ovvero che la radianza non si attenua con la distanza.

2.3.1.1 La funzione di distribuzione di riflettanza bidirezionale

Al fine di descrivere come la superficie di un oggetto disperda l'energia luminosa ricevuta da tutto l'ambiente, è utile introdurre la funzione di distribuzione di riflettanza bidirezionale (***BRDF***) .

Definizione 2.11. La BRDF di un punto $x \in A$ è il rapporto tra la radianza riflessa in una direzione uscente Θ_r, e la irradianza incidente dalla direzione Θ_i, ovvero la derivata della radianza riflessa sull'irradianza incidente:

$$f_r(x, \Theta_i, \Theta_r) = \frac{dL(x, \Theta_r)}{dE(\Theta_i, x)} = \frac{dL(x, \Theta_r)}{L(\Theta_i, x)\cos\theta_i d\omega_{\Theta_i}}. \qquad (2.3.2)$$

Osservazione 2.1. La BRDF assume sempre valori positivi.

Proposizione 2.4. *La BRDF rimane inalterata se si invertono le direzioni incidente ed uscente, ovvero*

$$f_r(x, \Theta_i, \Theta_r) = f_r(x, \Theta_r, \Theta_i).$$

Questa è chiamata reciprocità di Helmholtz.

Osservazione 2.2. La f_r in generale dipende dall'orientamento della superficie (anisotropica), ovvero se si ruota la superficie intorno alla normale nel punto x, la f_r cambia. Ad ogni modo la maggior parte dei materiali comuni sono isotropici, ed in tali casi la f_r non dipende dall'orientamento della superficie a cui appartiene x.

Proposizione 2.5. *I valori assunti dalla BRDF rispetto ad una direzione incidente non dipendono dalla possibile presenza di irradianza lungo altri angoli incidenti.*

Al fine di conoscere la radianza totale riflessa da x, punto su di una superficie opaca e non emettente luce propria, rispetto ad una direzione Θ_r e dovuta alla irradianza ricevuta da tutte le direzioni sulla sfera Ω_x, è necessario integrare la 2.3.2 su Ω_x.

Teorema 2.1. *Sia $x \epsilon A$, $\Theta_i, \Theta_r \epsilon \Omega_x$ e n_x normale ad A in x. Sussiste la seguente relazione:*

$$L(x, \Theta_r) = \int_{\Omega_x} d\omega_\Theta f_r(x, \Theta, \Theta_r) L(\Theta, x) \cos(n_x, \Theta) \tag{2.3.3}$$

dove $\cos(n_x, \Theta)$ *è il coseno dell'angolo formato dai vettori n_x e Θ.*

Dimostrazione. Dalla 2.3.2 si ha

$$dL(x, \Theta_r) = f_r(x, \Theta_i, \Theta_r) dE(\Theta_i, x)$$

Integrando su tutte le direzioni incidenti:

$$L(x, \Theta_r) = \int_{\Omega_x} f_r(x, \Theta, \Theta_r) dE(\Theta, x)$$

da cui, sempre dalla 2.3.2, si ha la 2.3.3. □

La 2.3.3 viene chiamata *equazione della riflettanza.*

Poiché deve essere soddisfatto il bilanciamento energetico, ovvero il numero di fotoni emessi non può essere maggiore del numero di fotoni ricevuti, risulta utile introdurre una nuova quantità:

Definizione 2.12. Posto

$$a(x, \Theta_i) = \int_{\Omega_x} d\omega_\Theta f_r(x, \Theta, \Theta_i) \cos(n_x, \Theta) \tag{2.3.4}$$

la quantità $a(x, \Theta_i)$ è detta *albedo*, con $0 \leq a(x, \Theta_i) \leq 1$.

Osservazione 2.3. L'albedo misura la frazione di radiazione incidente che viene riflessa in tutte le direzioni: è massima quando tutta la radiazione incidente viene riflessa, cioè misura 1 e la superficie appare bianca. L'albedo è minima, cioè 0, quando non viene riflessa nessuna radiazione incidente e la superficie appare nera.

2.3.1.2 L'equazione della Radianza

Definizione 2.13. Sia $x\epsilon A_S$ sorgente luminosa, e sia $\Theta\epsilon\Omega_x$. Si definisce *emittanza* $L_e(x,\Theta)$ la radianza emessa da x in direzione Θ per unità di superficie.

Partendo dall'equazione 2.3.3, ma prendendo anche in considerazione l'emittanza come funzione di inizializzazione, è possibile formulare quella che viene detta ***equazione di rendering***:

Definizione 2.14. La radianza che lascia un punto x in direzione Θ_{out} si può esprimere come:

$$L(x,\Theta_{out}) = L_e(x,\Theta_{out}) + \int_{\Omega_x} d\omega_\Theta L(\Theta,x)\, f_r(x,\Theta,\Theta_{out}) \cos(n_x,\Theta) .$$

Si può notare quindi che la radianza uscente è uguale all'emittanza più la radianza incidente proveniente da ogni direzione visibile e riflessa da x in direzione Θ_{out} .

Teorema 2.2. *Sia $x\epsilon A$ e sia $\Theta_{out}\epsilon\Omega_x$. Si ha che:*

$$L(x,\Theta_{out}) = L_e(x,\Theta_{out}) + \int_{\Omega_x} d\omega_\Theta L(y,-\Theta)\, f_r(x,\Theta,\Theta_{out}) \cos(n_x,\Theta) \tag{2.3.5}$$

dove $y = r(x,\Theta)$.

Dimostrazione. Ricordando la 2.2.3 si ha che $L(-\Theta,x) = L(r(x,\Theta),-\Theta)$, da cui la 2.3.5. □

Invece di integrare sulla sfera Ω_x delle direzioni ω_Θ, ricordando l'eq.2.2.1, a volte può essere più conveniente integrare su tutti i punti visibili della scena.

Teorema 2.3. *Siano $x, y\epsilon A$, e sia $z = r(x,\Theta)$ con $\Theta\epsilon\Omega_x$. Si ha che:*

$$L(x,y) = L_e(x,y) + \int_A dA_z L(z,x)\, f_r(x,z,y)\, G(x,z) \tag{2.3.6}$$

con

$$G(x,z) = \frac{\cos(n_x,\Theta)\cos(n_z,-\Theta)\, V(x,z)}{r_{\overline{xz}}^2}$$

n_xe n_z normali ad A in x e z rispettivamente, $V(x,z)$ funzione di visibilità, e $r_{\overline{xz}}$ distanza tra i punti x e z.

Dimostrazione. Ricordando l'eq. 2.2.1, si può scrivere:

$$L(x,y) = L_e(x,y) + \int_{A_x} dA_z L(z,x) f_r(x,z,y) \frac{\cos(n_x,\Theta)\cos(n_z,-\Theta)}{r_{\overline{xz}}^2}$$

da cui la 2.3.6. □

Proposizione 2.6. *Sia $x \epsilon A$, e sia $\Theta_{in} \epsilon \Omega_x$. Si ha:*

$$L(\Theta_{in},x) = L_e(\Theta_{in},x) + \int_{\Omega_{H_y}} d\omega_\Theta L(\Theta,y) f_r(y,\Theta,\Theta_{in}) \cos(n_y,\Theta) \quad (2.3.7)$$

dove $y = r(x,\Theta)$.

La 2.3.7 è detta *equazione della radianza incidente in un punto x con direzione Θ_{in}.*

2.3.1.3 Flusso radiante

La risoluzione del problema dell'illuminazione globale necessita del calcolo di tutti i valori della funzione radianza per tutti i punti di A e per tutte le direzioni relativamente a quei punti, ovvero calcolare i valori della funzione per tutti i punti di un insieme quadridimensionale nello spazio $A \times \Omega$. Una discretizzazione ci permette di calcolare la radianza media su alcuni sottoinsiemi di punti e direzioni, ad esempio calcolandone il flusso radiante attraverso i pixel dello schermo. Nel prossimo capitolo vedremo alcuni approcci in termini algoritmici.

Il flusso radiante si può esprimere in termini di radianza integrando la funzione $L(x,\Theta)$ su tutti i punti e le direzioni appartenenti all'insieme su cui il flusso deve essere calcolato.

Proposizione 2.7. *Sia $S = A_s \times \Omega_s \subset A \times \Omega$. Il flusso $\Phi(S)$ attraverso S è dato da:*

$$\Phi(S) = \int_{A_s} dA_x \int_{\Omega_S} d\omega_\Theta L(x,\Theta) \cos(n_x,\Theta)$$

Integrando sull'intero insieme delle superfici della scena A:

Proposizione 2.8. *Vale la:*

$$\Phi(S) = \int_A dA_x \int_{\Omega_H} d\omega_\Theta L(x,\Theta)\, g(x,\Theta) \cos(n_x,\Theta) \quad (2.3.8)$$

con

$$g(x,\Theta) = \begin{cases} 1 & se\,(x,\Theta) \in S \\ 0 & se\,(x,\Theta) \notin S \end{cases}$$

2.3.2 Potenziale

In letteratura sono presenti alcuni lavori ([Szirmay-Kalos06, Szirmay-Kalos99]) in cui viene utilizzata la funzione potenziale per formulare il problema di illuminazione globale.

Definizione 2.15. Sia $y \epsilon A$, e sia $\Psi \epsilon \Omega_y$. La *funzione potenziale* è data da:

$$W(\Psi, y) = \frac{d^2\Phi(S)}{d^2\Phi(y,\Psi)} = \frac{d^2\Phi(S)}{L(y,\Psi)\cos(n_y,\Psi)\,d\omega_\Psi dA_y} \tag{2.3.9}$$

dove $(y,\Psi) \in S = A_s \times \Omega_s \subset A \times \Omega$ e $\Phi(S)$ è il flusso radiante attraverso S.

La funzione potenziale misura il flusso attraverso S in termini del flusso che lascia (y,Ψ) se vi è stata posta un'unica luce con radianza $L(y,\Psi)$.

Si può dimostrare che è possibile esprimere la funzione potenziale in forma integrale:

Proposizione 2.9. *Sia $y \epsilon A$, e sia $\Psi \epsilon \Omega_y$. La funzione potenziale è data da:*

$$W(\Psi, y) = g(y,\Psi) + \int_{\Omega_{H_z}} d\omega_\Theta W(\Theta, z)\, f_r(y, \Theta, -\Psi)\cos(n_z,\Theta) \tag{2.3.10}$$

dove $z = r(y,\Psi)$ e

$$g(y,\Psi) = \begin{cases} 1 & se\,(y,\Psi) \in S \\ 0 & se\,(y,\Psi) \notin S \end{cases}$$

Osservando la 2.3.10 si nota facilmente che questa è identica alla 2.3.7, di cui preferiremo la formulazione in questo testo.

2.3.2.1 Flusso

Una espressione del flusso di un insieme, basata sulla funzione potenziale, si può ottenere integrando la 2.3.9. Considerato che le sorgenti luminose ($L_e \neq 0$) sono gli unici punti che emettono energia luminosa nella scena, i rispettivi valori della radianza sono gli unici che contribuiscono all'illuminazione della scena.

Teorema 2.4. *Sia $S = A_s \times \Omega_s \subset A \times \Omega$. Il flusso $\Phi(S)$ attraverso S è dato da:*

$$\Phi(S) = \int_A dA_x \int_{\Omega_{H_x}} d\omega_\Theta W(\Theta, x) L_e(x,\Theta)\cos(n_x,\Theta) \tag{2.3.11}$$

Dimostrazione. Dalla 2.3.9 si ha:

$$d^2\Phi(S) = W(\Psi, y)L(y, \Psi)\cos(n_y, \Psi)\, d\omega_\Psi dA_y =$$

$$= W(\Theta, x)L_e(x, \Theta)\cos(n_x, \Theta)\, d\omega_\Theta dA_x$$

dove $(y, \Psi) \in light = A_{light} \times \Omega_{light}$ e $(x, \Theta) \in S = A_S \times \Omega_S$. Da cui la 2.3.11. □

Teorema 2.5. *Vale la:*

$$\Phi(S) = \int_A dA_x \int_{\Omega_{H_x}} d\omega_\Theta W_e(\Theta, x)L(x, \Theta)\cos(n_x, \Theta) \tag{2.3.12}$$

Dimostrazione. Posto $W_e(\Theta, x) = g(x, \Theta)$ si ha che la 2.3.8 diventa la 2.3.12. □

2.4 Equazione di Kajiya

Abbiamo visto la formulazione dell'equazione di rendering basata sulle leggi della radiometria. Il primo modello introdotto nella letteratura della Grafica Computazionale è stato quello di Kajiya del 1986, meglio noto come *equazione di rendering di Kajiya* [Kajiya86]:

$$I(x, x') = g(x, x')\left[\varepsilon(x, x') + \int_A \rho(x, x', x'')\, I(x', x'')\, dx''\right] \tag{2.4.1}$$

dove:

- $I(x, x')$ è l'intensità della luce che passa dal punto x' al punto x, anche detta intensità di trasporto tra due punti non occlusi.
- $g(x, x')$ è la funzione di visibilità tra x e x'. Se tra x e x' non è possibile tracciare un raggio di luce diretto, ovvero un segmento che non interseca altre superfici, allora $g(x, x') = 0$. Se invece sono visibili, allora g varia come l'inverso del quadrato della distanza tra i due punti. La g è quindi diversa da V definita in def.2.6.
- $\varepsilon(x, x')$ è l'emittanza trasferita da x' a x ed è relativa all'intensità di tutta la luce emessa da x' in direzione di x.

- $\rho(x, x', x'')$ è il termine diffusivo rispetto alle direzioni x' e x'', e consiste nell'intensità di energia proveniente dal punto x'' e diffusa verso x dal un punto x'. Kajiya la chiama riflettanza di trasporto tra tre punti non occlusi.

La funzione ρ può essere messa in relazione con la funzione di distribuzione di riflettanza bidirezionale *BRDF* (bi-directional reflectance distribution function), che rappresenta la luce riflessa da un punto x' di una superficie S verso una direzione $D(\theta_{ref}, \varphi_{ref})$ in presenza di una luce proveniente dalla direzione $P(\theta_{in}, \varphi_{in})$:

$$\rho(x, x', x'') = BRDF\left(P(\theta_{in}, \phi_{in}), D(\theta_{ref}, \phi_{ref})\right) \cos\theta' \cos\theta_{ref}$$

dove θ' è la latitudine della linea $x'x$ rispetto al punto x.

L'integrale al secondo membro di 2.4.1 è definito su A, cioè tutti i punti di tutte le superfici nella scena, o equivalentemente tutte le direzioni su $\Omega_{x'}$.

Dal punto di vista fisico, l'equazione 2.4.1 vuole rappresentare l'intensità di trasporto luminoso dal punto x' al punto x, che risulta essere uguale alla quantità di luce emessa da x' verso x sommata alla quantità di luce proveniente da tutte le altre superfici nella scena e diffusa da x' verso x.

Per la risoluzione di questa equazione è necessario conoscere:

- un modello per la luce emessa, ε, dalla superficie;
- una rappresentazione della BRDF per ogni superficie;
- un metodo per la valutazione della funzione di visibilità g.

In letteratura sono presenti varie metodologie per descrivere e valutare queste funzioni ([Goral84, Cook84, Whitted80, Cook86]). Nel considerare l'equazione 2.4.1 si può però osservare che:

1. non ha una soluzione esplicitamente calcolabile. Gran parte dei metodi di risoluzione, come vedremo nel capitolo successivo, impongono ipotesi restrittive che implicano risultati poco accurati, ovvero che producono immagini di scarsa qualità visiva. Molti algoritmi utilizzano i metodi Monte Carlo per la valutazione dell'integrale.

2. questa formulazione è indipendente dal punto di vista dell'osservatore, poiché il punto x' è un punto qualsiasi della scena. Più in generale il problema dell'illuminazione globale può essere affrontato in maniera sia indipendente (es. Radiosity) sia dipendente dal punto di vista. In quest'ultimo caso si valutano i soli punti visibili della scena, e la complessità dell'equazione di rendering si riduce.

3. si tratta, come in tutte le formulazioni di questo capitolo, di una equazione di Fredholm del secondo tipo, ovvero del tipo: $a(x) = b(x) + \int_D K(x,y)\, a(y)\, dy$, con $b(x)$ e $K(x,y)$ funzioni date, e $a(x)$funzione incognita. Un approccio molto utilizzato nella Grafica Computazionale per risolvere quest'equazione è quello di tracciare dei raggi all'inverso dal piano dell'immagine, costruendo un percorso di oggetto in oggetto fino ad ottenere una stima del valore del pixel. La complessità dipende dal numero di percorsi tracciati.

Poiché risulta generalmente difficile utilizzare l'equazione di rendering di Kajiya, tratteremo i metodi di risoluzione utilizzando le più rigorose definizioni di equazione di radianza e di flusso ottico.

Capitolo 3

I metodi di risoluzione

Abbiamo introdotto nel capitolo precedente diverse formulazioni del modello matematico per il rendering con illuminazione globale: attraverso la valutazione dell'intensità di trasporto luminoso, del flusso di radianza, o ancora della funzione potenziale.

In tutte le forme, si tratta di una equazione di Fredholm del secondo tipo, ovvero del tipo:

$$a(x) = b(x) + \int_S K(x, y)\, a(y)\, dy \tag{3.0.1}$$

con $b(x)$ e $K(x, y)$ funzioni date, S dominio di integrazione, e $a(x)$funzione incognita. In genere si vuole valutare $a(x)$ in un numero di punti $x_1, \ldots, x_n \in S$.

In questo capitolo saranno introdotti i principali metodi di risoluzione presenti in letteratura, e verrà presentato un nuovo metodo iterativo che, a parità di efficienza, è dotato di una maggiore accuratezza, oltre a prestarsi ad un efficiente implementazione sulle GPU.

3.1 Classificazione per percorsi di luce

Per poter comprendere rapidamente le caratteristiche qualitative dei vari metodi che prenderemo in esame, introduciamo la notazione per percorsi di luce, che descrive il tipo di interazione luminosa utilizzata nei vari metodi.

Anzitutto osserviamo che la luce che raggiunge una superficie può essere riflessa specularmente o diffusamente:

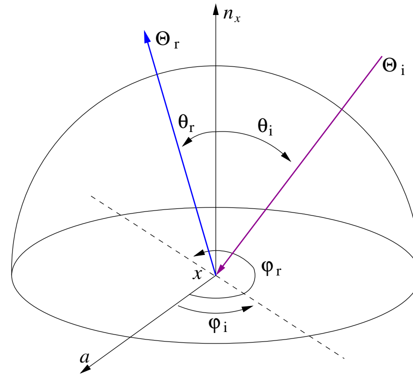

Figura 3.1.1: Riflessione speculare

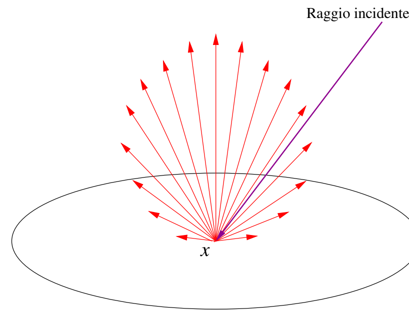

Figura 3.1.2: Riflessione diffusa

- La *riflessione speculare* si ha quando un singolo raggio di luce incidente la superficie A con direzione Θ_i , individuata dagli angoli (θ_i, φ_i), in un punto x produce un singolo raggio riflesso con direzione Θ_r , determinata da $(\theta_r, \varphi_r) = (-\theta_i, \varphi_i + \pi)$. (Fig. 3.1.1)
- La *riflessione diffusa* avviene quando un singolo raggio di luce incidente la superficie A in un punto x non viene riflesso con una direzione specifica, bensì viene diffuso su molte direzioni casuali. (Fig. 3.1.2)

Questo comportamento è dovuto alla natura microscopica del materiale che costituisce la superficie dell'oggetto. Ad esempio i materiali metallici levigati tendono a comportarsi specularmente, mentre i materiali porosi diffondono la luce senza alcuna riflessione. In realtà molti materiali reali hanno entrambe le proprietà, essendo diffusivi ma con una riflessione speculare in un intorno di Θ_r. Questo concetto è stato difatti utilizzato da Phong [Phong75] nel suo modello di riflessione per il rendering con illuminazione locale.

Quindi la luce che proviene da una superficie A e che raggiunge una superficie B può essere originata da un fenomeno speculare o diffusivo, ed altresì può

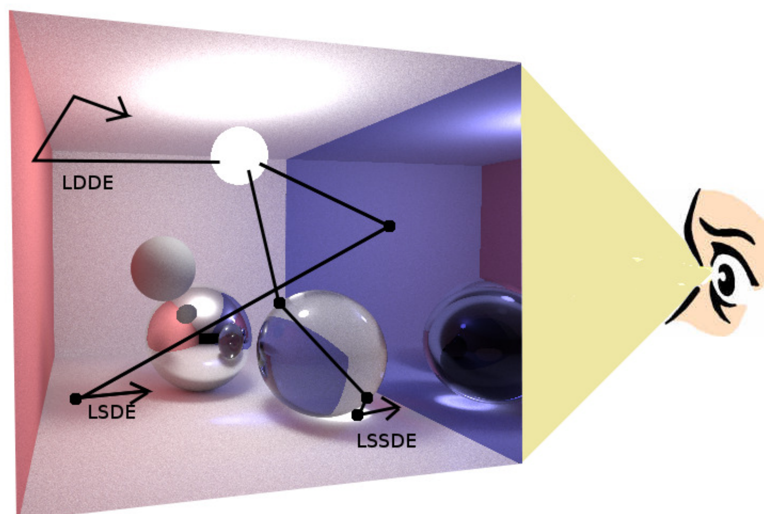

Figura 3.1.3: Esempi di percorsi di luce

proseguire dalla superficie B specularmente o diffusamente. Si può quindi dire che per coppie di superfici consecutive in un percorso di luce, esistono quattro tipi di trasferimento:

- da diffusivo a diffusivo,
- da speculare a diffusivo,
- da diffusivo a speculare,
- da speculare a speculare.

Alcuni metodi risolutivi presuppongono che ci siano solo interazioni tra superfici diffusive, come il metodo radiosity. Altri che ci siano solo superfici speculari, come il metodo raytracing. Altri ancora prevedono interazioni complesse tra vari tipi di superfici. A tal fine Heckbert [Heckbert90] ha introdotto una notazione mediante stringhe che permette di elencare tutte le interazioni che un percorso di luce compie dalla sorgente luminosa (L) fino all'occhio (E). Nell'ordine in cui sono stati elencati prima, le varie interazioni saranno categorizzate come DD, SD, DS, SS.

Alcuni esempi di percorsi di luce possono essere (Fig. 3.1.3):

1. LDDE, dove un raggio luminoso colpisce prima un oggetto diffusivo e poi ancora un altro oggetto diffusivo prima di raggiungere l'occhio dell'osservatore.

2. LSDE, quando ad esempio in una zona d'ombra su di una superficie diffusiva viene riflessa la luce tramite uno specchio.

3. LSSDE, che può essere presente quando una luce attraversa una sfera di cristallo e colpisce una superficie diffusiva. Il primo contatto del raggio luminoso con la sfera è di tipo speculare con rifrazione della luce all'interno, il secondo è speculare con uscita dalla sfera, ed il terzo avviene quando la luce colpisce la superficie diffusiva creando un effetto concentrato (anche noto come effetto *caustico*).

In generale un metodo di risoluzione dell'illuminazione globale che possa essere considerato completo dovrà includere un qualsiasi percorso luminoso, scritto come L(D|S)*E, dove | indica *oppure*, e * indica ripetizioni.

3.2 Soluzioni incomplete

Molti degli approcci alla risoluzione del problema dell'illuminazione globale in realtà apportano delle significative restrizioni nelle caratteristiche della scena che può essere rappresentata, come ad esempio limitazione sui possibili percorsi di luce.

Verranno ora proposti i metodi più significativi, da cui poi sono state generate numerose variazioni più o meno generali o restrittive.

3.2.1 Raytracing

Si consideri una scena in cui gli oggetti sono tutti speculari (come se ad esempio fossero di plastica lucida).

In questo contesto si posso tralasciare le interazioni diffusive tra le superfici, e quindi si può pensare ad un metodo che calcoli solo i percorsi di luce di tipo LS*E.

Poiché rimane impraticabile la soluzione dell'equazione di rendering calcolando il percorso di ogni singolo raggio di luce emesso dalle sorgenti presenti nella scena, l'intuizione di Whitted [Whitted80] è stata quella di ragionare al contrario, ovvero partire dal sensore, ad esempio dall'occhio e dal piano di

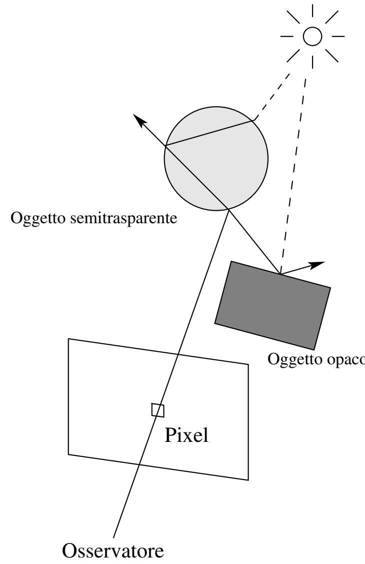

Figura 3.2.1: Ray Tracing di Whitted

visibilità in cui sono i pixel. Questo è un tipico esempio in cui il passo locale e quello globale sono fusi in uno.

A partire dall'occhio, vengono tracciati dei raggi che passano per alcuni punti del piano di visibilità e incontrano gli oggetti nella scena. Quando un oggetto viene colpito dal raggio, questo è riflesso specularmente secondo le leggi dell'ottica con un certo fattore di degradazione di intensità, e similarmente viene in parte rifratto all'interno dell'oggetto. Per entrambi i casi, si può pensare di proseguire finché l'energia trasmessa non scende al di sotto di una certa soglia. Infine per ogni punto toccato nel percorso di luce, viene effettuato un controllo dell'ombra per assicurarsi che il punto sia illuminato o meno.

Nei termini dell'equazione di rendering 2.3.5, il metodo del raytracing può essere descritto così:

$$L\left(x,\Theta_{out}\right)=L_{e}\left(x,\Theta_{out}\right)+\int_{\Omega_{H_x}}d\omega_{\Theta}L_{source}\left(y,-\Theta\right)f_{r}\left(x,\Theta,\Theta_{out}\right)\cos\left(n_{x},\Theta\right)+$$

$$+k_{r}\left(x,\Theta,\Theta_{r}\right)\cdot L\left(r\left(x,\Theta_{r}\right),-\Theta_{r}\right)+k_{t}\left(x,\Theta,\Theta_{t}\right)\cdot L\left(r\left(x,\Theta_{t}\right),-\Theta_{t}\right) \quad (3.2.1)$$

dove $y=r\left(x,\Theta\right)$, L_{source} è la radianza emessa dalla sorgente luminosa, Θ_r e Θ_t sono le direzioni ideali di riflessione e rifrazione luminosa, e k_r e k_t sono i coefficienti di riflessione e rifrazione.

3.2.2 Radiosity

Prendiamo ora in considerazione una scena in cui ci siano solo superfici diffusive, ovvero in cui i percorsi di luce siano solo del tipo LD*E, e che anche le luci abbiano solamente proprietà diffusive. Questo vuol dire che la radianza uscente è indipendente dalla direzione verso cui esce, poiché la BRDF è costante in tutte le direzioni per ogni punto $x \in A$.

In questo contesto possiamo introdurre il metodo Radiosity.

Anzitutto si suddivide l'intera scena in elementi geometrici superficiali di forma qualsiasi.

Lemma 3.1. *Sia A l'insieme delle superfici di una scena. Esiste una partizione di A tale che:*

$$A = \bigcup_i A_i$$

e tale che:

$$\forall i, j \epsilon \{1, ..., n\} \quad e \ (i \neq j) \quad \Rightarrow \quad A_i \cap A_j = \emptyset$$

Le superfici A_i sono dette patch di A.

Osservando che ci siamo posti in una scena con sole superfici diffusive, prendendo una partizione in patch, supponiamo che sia la radianza, sia la BRDF assumano valori costanti per ogni $x \in A_i$, con i fissato.

Teorema 3.1. *Sia A l'insieme delle superfici di una scena con sole proprietà di riflessione diffusa, e sia $\{A_i\}$ con $i = 1, ..., n$ una partizione in patch di A. Date n funzioni di base $N_i(x, \Theta)$ che ricoprono l'intero dominio $A \times \Omega_H$, con*

$$N_i(x, \Theta) = \begin{cases} 0 & se\ x \notin A_i \\ 1 & se\ x \in A_i \end{cases}$$

e tali che:

$$\begin{aligned} L(x, \Theta) &\approx \tilde{L}(x, \Theta) = \sum_{j=1}^{n} L_j N_j(x, \Theta) \\ L_e(x, \Theta) &\approx \tilde{L_e}(x, \Theta) = \sum_{j=1}^{n} L_{ej} N_j(x, \Theta) \end{aligned} \tag{3.2.2}$$

Si ha allora che

$$L_i = L_{ei} + \rho_i \sum_{j=1}^{n} L_j F_{ij} \tag{3.2.3}$$

con

$$F_{ij} = \frac{\int_{A_i} dA_x \int_{A_j} dA_z G(x,z)}{A_i \pi}$$

$$L_{ei} = \frac{\int_{A_i} dA_x \int_{\Omega_x} d\omega_\Theta L_e(x,\Theta) \cos(n_x,\Theta)}{A_i \pi}$$

$$\rho_i = \int_{\Omega_x} d\omega_\Theta f_r(x) \cos(n_x,\Theta) = \pi f_{ri}$$

dove f_{ri} *è la BRDF diffusiva per la patch* i *e* $G(x,z)$ *è quella definita in 2.3.6.*

Dimostrazione. Al fine di determinare i coefficienti L_j , utilizziamo l'approssimazione nel senso dei minimi quadrati (metodo di Galerkin) e consideriamo l'equazione 2.3.6 su ognuna delle funzioni di base $N_i(x,\Theta)$.

Definiamo il prodotto scalare tra funzioni come:

$$< F_i, F_j >= \int_A dA_x \int_{\Omega_x} d\omega_\Theta F_i(x,\Theta) F_j(x,\Theta) \cos(n_x,\Theta)$$

Dalla 2.3.6 si avrà allora che:

$$\forall i: \quad \int_A dA_x \int_{\Omega_x} d\omega_\Theta L(x,\Theta) N_i(x,\Theta) \cos(n_x,\Theta) =$$

$$= \int_A dA_x \int_{\Omega_x} d\omega_\Theta L_e(x,\Theta) N_i(x,\Theta) \cos(n_x,\Theta) +$$

$$+ \int_A dA_x \int_{\Omega_x} d\omega_\Theta \left(\int_A dA_z L(z,x) f_r(x) G(x,z) \right) N_i(x,\Theta) \cos(n_x,\Theta)$$

Se sostituiamo $L(x,\Theta)$ e $L_e(x,\Theta)$ con le approssimazione date in 3.2.2, possiamo imporre che l'eguaglianza precedente sussista. In tal caso avremmo:

$$\forall i: \quad \int_A dA_x \int_{\Omega_x} d\omega_\Theta \left(\sum_{j=1}^{n} L_j N_j(x,\Theta) \right) N_i(x,\Theta) \cos(n_x,\Theta) =$$

$$= \int_A dA_x \int_{\Omega_x} d\omega_\Theta \left(\sum_{j=1}^{n} L_{ej} N_j(x,\Theta) \right) N_i(x,\Theta) \cos(n_x,\Theta) +$$

$$+ \int_A dA_x \int_{\Omega_x} d\omega_\Theta \left(\int_A dA_z \left(\sum_{j=1}^{n} L_j N_j(z,\overline{zx}) \right) f_r(x) G(x,z) \right) N_i(x,\Theta) \cos(n_x,\Theta)$$

Ricordando che le funzioni di base sono tutte nulle tranne che sulle loro rispettive patch, si ha:

$$\forall i: \quad \int_A dA_x \int_{\Omega_x} d\omega_\Theta L_i N_i(x,\Theta) N_i(x,\Theta) \cos(n_x,\Theta) =$$

$$= \int_A dA_x \int_{\Omega_x} d\omega_\Theta L_{ei} N_i(x,\Theta) N_i(x,\Theta) \cos(n_x,\Theta) +$$

$$+ \sum_{j=1}^{n} \int_A dA_x \int_{\Omega_x} d\omega_\Theta \left(\int_{A_j} dA_z L_j N_j(z,\overline{zx}) f(x) G(x,z) \right) N_i(x,\Theta) \cos(n_x,\Theta)$$

Osservando che:

$$\int_A dA_x \int_{\Omega_x} d\omega_\Theta N(x,\Theta) N_i(x,\Theta) \cos(n_x,\Theta) = \begin{cases} A_i \pi & se\ i = j \\ 0 & se\ i \neq j \end{cases}$$

posto:

$$L_{ei} = \frac{\int_A dA_x \int_{\Omega_x} d\omega_\Theta L_e(x,\Theta) N_i(x,\Theta) \cos(n_x,\Theta)}{A_i \pi} =$$

$$= \frac{\int_{A_i} dA_x \int_{\Omega_x} d\omega_\Theta L_e(x,\Theta) \cos(n_x,\Theta)}{A_i \pi}$$

si ha infine:

$$\forall i \quad A_i \pi L_i = A_i \pi L_{ei} + \sum_{j=1}^{n} \pi f_{ri} L_j \int_{A_i} dA_x \int_{A_j} dA_z G(x,z)$$

$$L_i = L_{ei} + \rho_i \sum_{j=1}^{n} L_j \frac{\int_{A_i} dA_x \int_{A_j} dA_z G(x,z)}{A_i \pi}$$

da cui poi la 3.2.3. □

La 3.2.3 è detta *equazione di radiosity*, dove,

- F_{ij} è chiamato *fattore di forma* tra le patch i e j. Nel caso di patch planari si ha che $F_{ii} = 0$, non potendo una patch scambiare energia con sé stessa;
- ρ_i è la riflettanza emisferica, ovvero la BRDF integrata sull'emisfera:
- L_{ei} è chiamata radiosity emessa dalla patch i.

Il metodo radiosity consiste nella risoluzione dell'equazione 3.2.3, ovvero nel passo globale del rendering. In forma matriciale l'eq.3.2.3 si può scrivere come:

$$\begin{bmatrix} 1 & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1 & \cdots & -\rho_2 F_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \ldots & 1 \end{bmatrix} \begin{bmatrix} L_1 \\ L_2 \\ \cdots \\ L_n \end{bmatrix} = \begin{bmatrix} L_{e1} \\ L_{e2} \\ \cdots \\ L_{en} \end{bmatrix}$$

Le difficoltà legate all'utilizzo di questo metodo per il rendering risiedono nel calcolo dei fattori di forma, e nel passo locale del rendering da eseguire in una seconda fase.

Per il calcolo dei fattori di forma generalmente si utilizza il metodo geometrico di Nusselt [Cohen85] che definisce una formula di quadratura mediante la discretizzazione con un emicubo unitario di centro coincidente con il centro della patch.

Per il passo locale, ovvero la proiezione della scena sul sensore posto innanzi all'osservatore, risulta invece necessario operare una interpolazione tra le patch.

3.3 Metodi stocastici

L'introduzione di metodi stocastici per la soluzione del problema dell'illuminazione globale è motivata dalla necessità di rappresentare scene in cui sono presenti superfici di varia natura. Volendo superare le limitazioni dei metodi di Raytracing e Radiosity, appare necessario introdurre un metodo che permetta il calcolo di equazioni del tipo 3.0.1.

In letteratura sono presenti molti esempi di metodi Monte Carlo utilizzati per l'illuminazione globale [Szirmay-Kalos99, Sanjurjo09, Doucet10].

Per la descrizione di un approccio generale alla soluzione di un equazione di Fredholm del secondo tipo con il metodo Monte Carlo si veda [Doucet10].

Partendo dall'equazione 3.0.1, posto $x_0 = x$, $x_1 = y$, con la tecnica dell'espansione si perviene alla forma:

$$a(x_0) = b(x_0) + \sum_{n=1}^{\infty} \int_{D^n} \left(\prod_{k=1}^{n} K(x_{k-1}, x_k) \right) b(x_n)\, dx_{1:n}$$

che risulta essere una somma infinita di integrali.

Applicando il metodo Monte Carlo si può stimare il valore di a in un punto x_0 come:

$$\langle a(x_0)\rangle = b(x_0) + \frac{1}{N}\sum_{i=1}^{N}\frac{K(x_0, y_i)\, a(y_i)}{p_1(y_i)} \tag{3.3.1}$$

dove $p_1(y)$ è una funzione di densità di probabilità (PDF) su dominio D con la quale generiamo N campioni y_i.

Applicando l'espansione anche alla 3.3.1, si ha la:

$$\langle a(x_0)\rangle = b(x_0) + \frac{1}{N_1}\sum_{i=1}^{N_1}\frac{K(x_0, y_i)}{p_1(y_i)}\left(b(y_i) + \frac{1}{N_2}\sum_{j=1}^{N_2}\frac{K(y_i, z_j)}{p_2(z_j|y_i)}\left(b(z_j) + ...\right)\right) \tag{3.3.2}$$

Proposizione 3.1. *La serie 3.3.2 converge se è verificata la seguente condizione:*

$$\|K\| = \int_D |K(x, y)|\, dy < 1.$$

Il caso dell'equazione di rendering rientra in questa condizione, essendo $K(x, y)$ il prodotto tra la BRDF e un fattore coseno.

Poiché siamo interessati a valutare lo stimatore 3.3.2 in un tempo finito, è necessario introdurre un criterio di arresto per la sommatoria.

Per ovviare alla distorsione che si introdurrebbe nell'interruzione arbitraria dopo un certo numero di iterazioni, è opportuno utilizzare il concetto di *assorbimento* per determinare se un certo termine della somma deve essere valutato, oppure se bisogna interrompere il procedimento.

Definizione 3.1. Si consideri un procedimento iterativo infinito. Sia u una variabile casuale uniforme in $[0, 1]$. Se all'iterazione $i - ma$ la variabile $u(i)$ è minore di un certo coefficiente α, detto di *assorbimento*, allora il procedimento si interrompe a quella iterazione. Altrimenti il procedimento continua con l'iterazione successiva.

Nel nostro caso, presa una variabile casuale u uniforme, la sommatoria si interrompe al passo i se $u(i) < \alpha$. Altrimenti si procede con l'operazione di somma.

Questa procedura garantisce che la stima con assorbimento non cambia il valore atteso dell'esperimento.

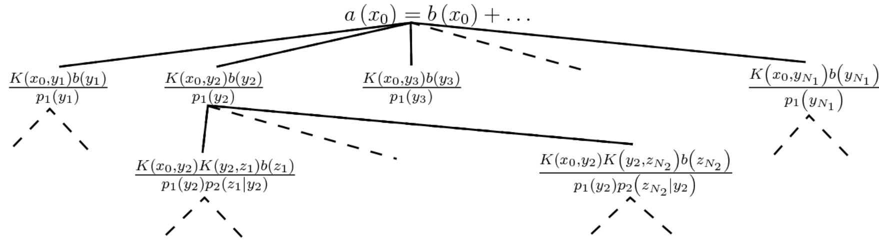

Figura 3.3.1: Albero per la valutazione di $a(x_0)$

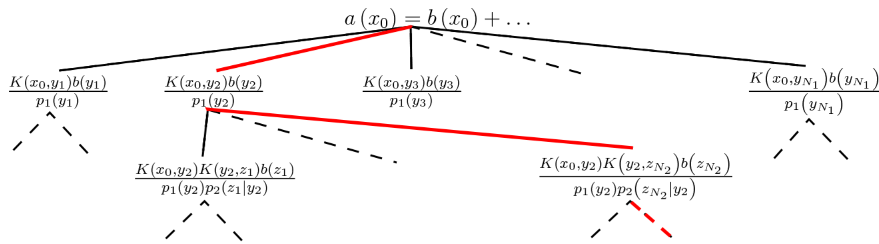

Figura 3.3.2: Esempio di cammino casuale

Si può notare che il numero di campioni richiesti per la stima di ogni termine della somma cresce molto rapidamente. Per stimare, ad esempio, la somma dei primi due termini sono necessari $N_1 \cdot N_2$ nuovi campioni.

Generando nuovi campioni ad ogni passo si costruisce un albero di campioni che si ramifica per ogni elemento (Fig.3.3.1). Scendendo nell'albero di campioni, i termini più bassi contribuiscono sempre meno alla stima di $a(x_0)$, per effetto della convergenza. Ma troncare questi termini dopo un fissato numero di iterazioni introdurrebbe una distorsione inaccettabile nella valutazione.

Un approccio per ovviare a questa distorsione può essere quello di valutare un solo campione per ogni termine, per ogni ramo dell'albero, ovvero generare una Catena di Markov, che qui chiameremo *cammino casuale* (Fig.3.3.2). In tal caso la stima diverrebbe:

$$\langle a(x_0) \rangle = b(x_0) + \frac{K(x_0, x')}{p'(x')} b(x') + \frac{K(x_0, x') K(x', x'')}{p'(x') p''(x''|x')} b(x'') + \ldots$$

Ci si aspetta che la varianza associata a questa stima sia molto alta, poiché risente dell'approssimazione di una sommatoria di N numeri con un solo valore.

Ma poiché in questo modo sono stati generati molti meno campioni che nell'intero albero, si può cercare di ottenere una stima più accurata generando altri cammini casuali.

Generando N_{camm} cammini casuali originati in $a(x_0)$, e poi calcolando la media delle stime ottenute, si ha che lo stimatore diventa:

$$\langle a(x_0)\rangle = \frac{1}{N_{camm}} \sum_{i=1}^{N_{camm}} \left(b(x_0) + \frac{K(x_0, x')}{p'(x')} b(x') + \frac{K(x_0, x') K(x', x'')}{p'(x') p''(x''|x')} b(x'') + ... \right) \tag{3.3.3}$$

In letteratura esistono molti algoritmi basati sui cammini casuali. Una classificazione interessante si può trovare in [Szirmay-Kalos99]. Qui introdurremo solo i due approcci principali.

3.3.1 Cammini casuali dal sensore alla luce

Per visualizzare una scena è necessario calcolare per ogni pixel dell'immagine finale la 2.3.12. Si procede quindi eseguendo un campionamento nell'area del pixel. Esistono varie strategie in letteratura per eseguire questo campionamento [Kajiya86][Lee85][Cook86]. In seguito si stimeranno la radianza dei punti scelti e delle relative direzioni.

Ricordando la 3.3.1, si può scrivere una stima della 2.3.5 come:

$$\langle L(x, \Theta_{out})\rangle = L_e(x, \Theta_{out}) + \frac{1}{N} \sum_{i=1}^{N} \frac{L(y, -\Theta_i) f_r(x, \Theta_i, \Theta_{out}) cos(n_x, \Theta_i)}{p(\Theta_i)}$$

La scelta della $p(\Theta)$ influenza la varianza della stima. Anche in questo ambito esistono vari approcci in letteratura [Shirley91, Zimmerman95].

Applicando la 3.3.3, si può determinare il valore di $\langle L(x, \Theta_{out})\rangle$ per ogni punto x_j campionato nell'area del pixel, per cui, dalla 2.3.8:

$$\langle \Phi(pixel)\rangle = \frac{1}{M} \sum_{j=1}^{M} \frac{\langle L(x_j, \Theta_j)\rangle g(\Theta_j, x_j) \cos\left(n_{x_j}, \Theta_j\right)}{p(x_j)}$$

3.3.2 Cammini casuali dalla luce al sensore

Si vuole calcolare il flusso che raggiunge un pixel utilizzando la 2.3.11. In questo caso stiamo assumendo che gli unici contributi in termini di energia luminosa al flusso nel pixel siano dovuti alla sorgente luminosa, ovvero quando $L_e \neq 0$.

Una PDF che campioni uniformemente punti sulla sorgente luminosa, e poi campioni direzioni uniformi sulla semisfera intorno i punti delle superfici, ha la seguente forma:

$$p\left(x, \Theta\right) = \frac{1}{superficie\, luminosa} \cdot \frac{1}{2\pi} = \frac{1}{\sum_{l=1}^{Q} A_l} \cdot \frac{1}{2\pi}.$$

Una stima del flusso quindi può essere:

$$\left\langle \Phi\left(pixel\right)\right\rangle = \frac{1}{M}\sum_{j=1}^{M}\left\langle W\left(\Theta_j, x_j\right)\right\rangle L_e\left(x_j, \Theta_j\right)\cos\left(n_{x_j}, \Theta_j\right)\cdot 2\pi\sum_{l=1}^{Nl} A_l$$

quando si prendono M campioni.

Ricordando la 3.3.1, si può scrivere una stima della 2.3.10:

$$\left\langle W\left(\Theta_j, x_j\right)\right\rangle = W_e\left(\Theta_j, x_j\right) + \frac{1}{N}\sum_{i=1}^{N} W\left(-\Psi_i, z_j\right) f_r\left(z_j, \Psi_i, \Theta_j\right) cos\left(n_{z_j}, \Psi_i\right)\cdot 2\pi$$

3.4 Metodi iterativi del punto fisso

Posto:

$$\mathcal{H}L\left(x, \Theta_{out}\right) = \int_{\Omega_{H_x}} d\omega_\Theta L\left(y, -\Theta\right) f_r\left(x, \Theta, \Theta_{out}\right)\cos\left(n_x, \Theta\right).$$

si ha che la 2.3.5 si può quindi scrivere come:

$$L = L_e + \mathcal{H}L$$

Questo ci permette di osservare che la soluzione dell'equazione di rendering si può vedere come il punto fisso dello schema iterativo

$$L_n = L_e + \mathcal{H}L_{n-1}$$

In questa sezione verrà presentato un metodo iterativo che, calcolando approssimazioni intermedie per ogni punto, se è garantita la convergenza, progressivamente tende al valore cercato.

Anzitutto individuiamo quali siano i criteri per garantire la convergenza del metodo iterativo.

Definizione 3.2. L'operatore $\mathcal{H}$ è una contrazione se, data una data norma $\|\|$, si ha che:

$$\|\mathcal{H}L\| < \lambda \cdot \|L\|$$

con $\lambda < 1$

Lemma 3.2. *Sia $\mathcal{H}$ una contrazione, posto $L_0 = 0$, si ha che:*

$$\|L_n - L_{n-1}\| = \left\|\mathcal{H}^{n-1}L_e\right\| < \lambda^{n-1} \cdot \|L_e\|$$

Dimostrazione. Si prendano due iterazioni successive:

$$L_n = L_e + \mathcal{H}L_{n-1}$$

$$L_{n-1} = L_e + \mathcal{H}L_{n-2}$$

Sottraendo le due equazioni, e ricordando che $L_0 = 0$, otteniamo che:

$$L_n - L_{n-1} = \mathcal{H}\left(L_{n-1} - L_{n-2}\right) = \mathcal{H}^{n-1}\left(L_1 - L_0\right) = \mathcal{H}^{n-1}L_e$$

da cui la tesi. □

La costante λ è detta *fattore di contrazione*, e dipende sia dalla geometria sia dalla riflettività media della scena. L'ipotesi restrittiva su L_0 indica che il ciclo iterativo deve partire da una sorgente luminosa.

Nel caso di un ambiente chiuso, cioè il caso elettivo per il problema dell'illuminazione globale, λ corrisponde all'albedo media. Questo significa, ricordando l'oss.2.3, che se è necessario calcolare i valore della radianza di ambienti altamente riflettenti, il metodo iterativo converge lentamente.

Il valore calcolato tramite l'operatore integrale $\mathcal{H}$ inevitabilmente contiene un errore ad ogni passo dell'iterazione; valutiamo l'errore nella soluzione.

Lemma 3.3. *Sia $\mathcal{H}^*$ una discretizzazione di $\mathcal{H}$, e supponiamo che entrambi gli operatori siano contrazioni e che quindi le iterazioni convergano qualunque sia la funzione iniziale. Supponiamo altresì che $L_0 = L$ e che*

$$L^* = \lim_{n\to\infty} L_n$$

Allora vale:

$$\|L^* - L\| \leq \frac{\|\mathcal{H}^*L - \mathcal{H}L\|}{1 - \lambda}$$

Dimostrazione. L'errore al passo n sarà:

$$\|L_n - L\| = \|L_n - L_{n-1} + L_{n-1} - \ldots + L_1 - L\| \leq \sum_{i=1}^{n} \|L_i - L_{i-1}\|$$

dove

$$\|L_i - L_{i-1}\| = \|\mathcal{H}^* L_{i-1} - \mathcal{H}^* L_{i-2}\| = \|\mathcal{H}^* (L_{i-1} - L_{i-2})\| \leq$$

$$\leq \lambda \cdot \|L_{i-1} - L_{i-2}\| \leq \lambda^{i-1} \cdot \|L_1 - L_0\|$$

quindi

$$\sum_{i=1}^{n} \|L_i - L_{i-1}\| \leq \|L_1 - L_0\| \cdot \left(1 + \lambda + \lambda^2 + \ldots + \lambda^{n-1}\right).$$

Se $n \to \infty$ si ha:

$$\|L^* - L\| \leq \|L_1 - L_0\| \cdot \left(1 + \lambda + \lambda^2 + \ldots + \lambda^{n-1}\right) = \frac{\|L_1 - L_0\|}{1 - \lambda}.$$

Infine, ricordando che $L_0 = L$, si ha che $L_1 - L = \mathcal{H}^* L - \mathcal{H} L$. Da cui la tesi. □

Questo risultato indica che nel caso in cui l'albedo $\lambda \to 1^-$(cioè nelle scene con superfici molto riflettenti) l'errore è massimo.

Ipotizzando di aver trovato i valori della radianza di una scena, ovvero aver completato il passo globale del rendering (cfr. Fig.1.3.1), per poter visualizzare la scena stessa è necessario calcolare il flusso radiante che passa attraverso i pixel dello schermo, ovvero eseguire il passo locale.

Dato un *pixel*, cioè la superficie rettangolare che rappresenta l'unità di suddivisione dell'area del piano di vista su cui giace lo schermo, per esso passano infiniti raggi luminosi. Dato che lo schermo è completamente trasparente, non assorbe o riflette alcuna energia. Poiché siamo interessati esclusivamente alla radianza in direzione dell'osservatore, dalla 2.3.8 segue:

$$\bar{\Phi}\,(pixel) = \int_{A_{pixel}} dA_z L\left(z, \hat{\Theta}\right) \cos\left(n_z, \hat{\Theta}\right) \tag{3.4.1}$$

dove $\hat{\Theta}$ è la direzione del raggio che, passando per z, raggiunge l'osservatore con radianza L.

Per calcolare l'integrale nella eq.3.4.1 bisogna individuare una formula di quadratura adatta. Anzitutto abbiamo supposto che il pixel abbia un'area unitaria. La formula più semplice applicabile è quella del punto medio:

$$\bar{\Phi}\left(pixel\right) = L\left(z, \hat{\Theta}\right) + O\left(h^2\right)$$

dove z è il punto medio del pixel e h è la lunghezza del lato del pixel.

Se invece si vuole maggiore accuratezza, è preferibile una formula di quadratura con più nodi.

Definizione 3.3. Sia $x_1, x_2, x_3, \ldots$ una sequenza di punti all'interno di un pixel. Tale sequenza si dice equidistribuita se

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} f\left(x_i\right) = \int_{pixel} f\left(x\right) dx \tag{3.4.2}$$

con f funzione integrabile secondo Riemann.

Come mostrato in [Krommer98]:

Proposizione 3.2. *Sia $x_1, x_2, x_3, \ldots$ una sequenza di punti all'interno di un pixel, sia $\mathcal{O}\left(E, n\right)$ una funzione che conta il numero dei punti $x_i \in E \subseteq pixel$. Allora:*

$$\lim_{n\to\infty} \frac{\mathcal{O}\left(E, n\right)}{n} = area\left(E\right) \tag{3.4.3}$$

è condizione sufficiente perché la sequenza x_i sia equidistribuita.

Quindi, nel caso in cui ci trovassimo in una situazione in cui vale la 3.4.3, allora potremmo applicare la 3.4.2.

3.4.1 Metodo delle direzioni

Cerchiamo ora di definire un metodo iterativo per scene chiuse e principalmente diffusive.

Proposizione 3.3. *Sia Θ_{pixel} il raggio che passa per il centro di un dato pixel e raggiunge l'occhio (puntiforme) dell'osservatore, e sia*

$$x_0 = r(osservatore, -\Theta_{pixel})$$

il punto della scena da cui parte il raggio Θ_{pixel} . Allora si ha che:

$$L\left(\Theta_{pixel}, pixel\right) = L\left(x_0, \Theta_{pixel}\right) = L_e\left(x_0, \Theta_{pixel}\right) +$$

$$+ \int\limits_{\Omega_{H_{x_0}}} d\omega_{\Theta'} L\left(r\left(x_0, \Theta'\right), -\Theta'\right) \cdot f_r\left(\Theta', x_0, \Theta_{pixel}\right) \cdot \cos\left(n_{x_0}, \Theta'\right)$$

Dimostrazione. Consegue direttamente dalla 2.3.5 □

La prop.3.3 ci dice che per poter calcolare $L\left(\Theta_{pixel}, pixel\right)$ è necessario conoscere la radianza nel punto $r\left(x_0, \Theta'\right)$ con direzione $-\Theta'$.

Decidiamo quindi di utilizzare il metodo del punto fisso per trovare una stima di $L\left(\Theta_{pixel}, pixel\right)$.

Proposizione 3.4. *Sia* $\Theta^{[i]} \epsilon \Omega_{H_{x_{i-1}}}$ *con* $i = \{1, ..., n\}$ *una sequenza di direzioni tali che* $x_{i+1} = r\left(x_i, -\Theta^{[i]}\right)$*, e sia* $\Theta^{[0]} = \Theta_{pixel}$ *. Allora si può assumere che:*

$$L\left(x_i, \bar{\Theta}^{[i]}\right) = L_e\left(x_i, \bar{\Theta}^{[i]}\right) + \mathcal{H}\left[x_i, \bar{\Theta}^{[i]}, L\left(x_{i+1}, \bar{\Theta}^{[i+1]}\right)\right] \tag{3.4.4}$$

con

$$\mathcal{H}\left[x_i, \bar{\Theta}^{[i]}, L\left(x_{i+1}, \bar{\Theta}^{[i+1]}\right)\right] =$$
$$= L\left(x_{i+1}, \bar{\Theta}^{[i+1]}\right) \int\limits_{\Omega_{x_i}} d\omega_{\Theta} f_r\left(\Theta, x_i, \bar{\Theta}^{[i]}\right) \cdot \cos\left(n_{x_i}, \Theta\right)$$

Dimostrazione. Si procede per costruzione. Fissata una direzione $\bar{\Theta}^{[1]}$ su $\Omega_{H_{x_0}}$, si consideri il punto $x_1 = r\left(x_0, \bar{\Theta}^{[1]}\right)$. Poiché supponiamo che la f_r sia nota per ogni punto e direzione della scena, si può assumere che:

$$L\left(x_0, \Theta_{pixel}\right) = L_e\left(x_0, \Theta_{pixel}\right) + \mathcal{H}\left[x_0, \Theta_{pixel}, L\left(x_1, \bar{\Theta}^{[1]}\right)\right]$$

con

$$\mathcal{H}\left[x_0, \Theta_{pixel}, L\left(x_1, \bar{\Theta}^{[1]}\right)\right] = L\left(x_1, \bar{\Theta}^{[1]}\right) \int\limits_{\Omega_{H_{x_0}}} d\omega_{\Theta} f_r\left(\Theta, x_0, \Theta_{pixel}\right) \cdot \cos\left(n_{x_0}, \Theta\right)$$

Tale stima è possibile se si conosce $L\left(x_1, \bar{\Theta}^{[1]}\right)$. Ma, fissata una direzione $\bar{\Theta}^{[2]}$ su $\Omega_{H_{x_1}}$

$$L\left(x_1, \bar{\Theta}^{[1]}\right) = L_e\left(x_1, \bar{\Theta}^{[1]}\right) + \mathcal{H}\left[x_1, \bar{\Theta}^{[1]}, L\left(x_2, \bar{\Theta}^{[2]}\right)\right]$$

con

$$\mathcal{H}\left[x_1, \bar{\Theta}^{[1]}, L\left(x_2, \bar{\Theta}^{[2]}\right)\right] = L\left(x_2, \bar{\Theta}^{[2]}\right) \int\limits_{\Omega_{H_{x_1}}} d\omega_{\Theta} f_r\left(\Theta, x_1, \Theta^{[2]}\right) \cdot \cos\left(n_{x_1}, \Theta\right),$$

Generalizzando, con le ipotesi poste, si ha che al passo i vale la 3.4.4. □

Se al passo x_{n+1} si trova su una sorgente luminosa si ha che:

$$L\left(x_{n+1}, \bar{\Theta}^{[n+1]}\right) = L_e\left(x_{n+1}, \bar{\Theta}^{[n+1]}\right)$$

e quindi il procedimento iterativo si può fermare. Per cui, sostituendo i valori di $L\left(x_i, \bar{\Theta}^{[i]}\right)$ calcolati, e procedendo all'indietro, si ottiene $L\left(x_0, \Theta_{pixel}\right)$.

Prendendo invece il punto iniziale su una sorgente luminosa lux, si possono scegliere i $\bar{\Theta}^{[i]}$ alla ricerca dell'$osservatore$, intersecando il sensore in un $pixel$.

Proposizione 3.5. *Sia* $x_0 \epsilon lux$, *e siano* $\Theta^{[i]} \epsilon \Omega_{H_{x_i}}$ *con* $i = \{0, ..., n\}$ *una sequenza di direzioni tali che* $x_{i+1} = r\left(x_i, \Theta^{[i]}\right)$. *Si ha che:*

$$L\left(x_0, \bar{\Theta}^{[0]}\right) = L_e\left(x_0, \bar{\Theta}^{[0]}\right)$$

Allora, per $i = \{1, ..., n\}$, *si può assumere che:*

$$L\left(x_i, \bar{\Theta}^{[i]}\right) = L_e\left(x_i, \bar{\Theta}^{[i]}\right) + \mathcal{H}\left[x_i, \bar{\Theta}^{[i]}, L\left(x_{i-1}, \bar{\Theta}^{[i-1]}\right)\right] \tag{3.4.5}$$

con

$$\mathcal{H}\left[x_i, \bar{\Theta}^{[i]}, L\left(x_{i-1}, \bar{\Theta}^{[i-1]}\right)\right] =$$
$$= L\left(x_{i-1}, \bar{\Theta}^{[i-1]}\right) \int\limits_{\Omega_{x_i}} d\omega_\Theta f_r\left(\Theta, x_i, \bar{\Theta}^{[i]}\right) \cdot \cos\left(n_{x_i}, \Theta\right)$$

Supponendo che la direzione $\bar{\Theta}^{[i]}$ è determinata dalle variabili casuali uniformi $\theta \in U\left(0, \frac{\pi}{2}\right)$ e $\varphi \in U\left(0, 2\pi\right)$, allora la sequenza di punti e direzioni generati dal metodo iterativo della prop.3.5 traccia un percorso di luce casuale del tipo L(D|S)*E (Fig.3.4.1). Non è quindi possibile determinare se verrà raggiunto un pixel sul sensore in pochi passi.

Nel paragrafo precedente abbiamo individuato una proprietà che ci può mostrare se il procedimento iterativo descritto è convergente.

Dal lemma 3.2, basta verificare che $\mathcal{H}$ è una contrazione.

Proposizione 3.6. *Sia* $\mathcal{H}$ *l'operatore definito dalla eq.3.4.5.* $\mathcal{H}$ *è una contrazione e si ha che:*

$$\left\|\mathcal{H}\left[x_i, \bar{\Theta}^{[i]}, L\left(x_{i+1}, \bar{\Theta}^{[i+1]}\right)\right]\right\| < \lambda \cdot \left\|L\left(x_{i+1}, \bar{\Theta}^{[i+1]}\right)\right\|, \quad \lambda = k_i \pi < 1.$$

Dimostrazione. Notiamo anzitutto che

$$\int\limits_{\Omega_{x_i}} d\omega_\Theta f_r\left(\Theta, x_i, \bar{\Theta}^{[i]}\right) \cdot \cos\left(n_{x_0}, \Theta\right) = a\left(x_i, \bar{\Theta}^{[i]}\right)$$

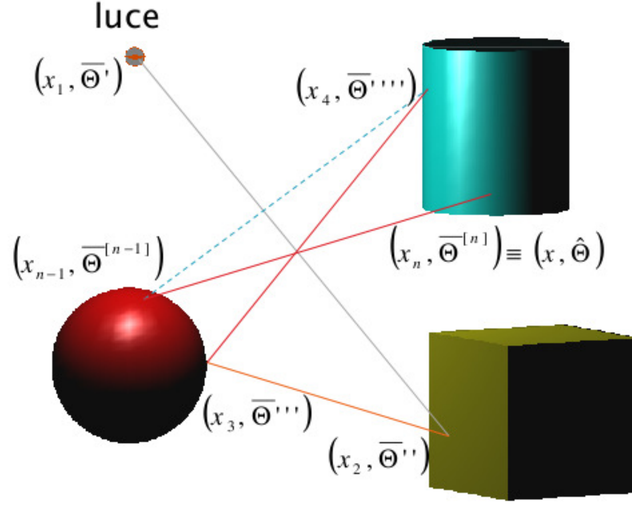

Figura 3.4.1: Percorso di luce con punto iniziale nella sorgente luminosa

dove $a\left(x_i, \bar{\Theta}^{[i]}\right)$ è l'albedo nel punto x_i con direzione $\bar{\Theta}^{[i]}$. Infatti per le superfici esclusivamente diffusive si ha che la BRDF è costante in tutte le direzioni, ovvero

$$f_r\left(\Theta, x_i, \bar{\Theta}^{[i]}\right) = k_i \ , \ \forall \Theta \epsilon \Omega_{H_{x_i}}$$

da cui

$$a\left(x_i, \bar{\Theta}^{[i]}\right) = k_i \pi$$

Per la conservazione dell'energia di ha che

$$k_i < \frac{1}{\pi}$$

e pertanto la tesi. □

Poiché $x_i = r\left(x_{i-1}, \bar{\Theta}^{[i]}\right)$, la velocità di convergenza dipende strettamente dalla scelta dei $\bar{\Theta}^{[i]}$ e dagli oggetti intersecati dai raggi emessi. Infatti nel caso di una scena con superfici esclusivamente diffusive la velocità di convergenza è massima, mentre nel caso di presenza di riflessioni la luce tende a propagarsi senza decadimento energetico, e la convergenza rallenta.

Proposizione 3.7. *Sia* $L\left(x_i, \bar{\Theta}^{[i]}\right)$ *definita come nell'eq.3.4.5. Si ha che:*

$$\lim_{i \to \infty} L\left(x_i, \bar{\Theta}^{[i]}\right) = 0$$

Dimostrazione. Segue dalla legge della conservazione dell'energia. □

Ora ricordiamo che il nostro obiettivo è quello di calcolare il flusso radiante attraverso ogni pixel, considerato il piano di vista su cui sono rappresentati i pixel dello schermo, come descritto dalla eq.3.4.1, dobbiamo individuare una opportuna formula di quadratura.

Poiché la scena rappresenta un ambiente chiuso e gli oggetti sono principalmente diffusivi, l'energia luminosa tende a disperdersi in ogni direzione e non ci sono punti visibili che non siano colpiti da raggi luminosi, anche se ognuno con intensità diversa.

Pertanto, tornando alla 3.4.5, abbiamo che in ogni x_i è possibile scegliere una direzione $\tilde{\Theta}^{[i]}$ che rappresenta il raggio luminoso uscente da x_i e che raggiunge l'osservatore.

Proposizione 3.8. *Sia x_i un punto della scena, e sia V la funzione di visibilità della def.2.6. Detta $\tilde{\Theta}^{[i]}$ la direzione della semiretta che congiunge x_i con l'osservatore, allora si ha che:*

$$V(x_i, osservatore) = 1 \Rightarrow \exists z_j \in pixel_k : r\left(z_j, \tilde{\Theta}^{[i]}\right) = osservatore$$

con $pixel_k$ appartenente al piano di vista.

Ciò vuol dire che se un punto x_i della scena è visibile all'osservatore, esiste sempre un punto di un $pixel_k$ attraverso cui è visibile x_i.

Proposizione 3.9. *Sia x_i una sequenza di punti definita come nella prop.3.5 con $i = 1, .., n$. Fissato un $pixel_k$, sia z_j una sequenza di punti definita come nella prop.3.8, con $j = 1, .., n_k < n$. Si ha che:*

$$\bar{\Phi}\left(pixel_k, n_k\right) = \frac{1}{n_k} \sum_{j=0}^{n_k} L\left(z_j, \tilde{\Theta}^{[j]}\right) \tag{3.4.6}$$

Dimostrazione. Poiché per ogni sequenza di $z_j \in pixel_k$ vale la 3.4.3, si ha la tesi. □

Procedendo nel calcolo dei $L\left(x_i, \bar{\Theta}^{[i]}\right)$, possiamo eseguire le sommatorie in 3.4.6 in maniera non ordinata (Fig.3.4.2).

Dalla prop.3.7 si può fissare il seguente criterio di arresto:

Condizione 1. Arresta il procedimento iterativo per il calcolo di $L\left(x_i, \bar{\Theta}^{[i]}\right)$ se è verificato che:

$$L\left(x_i, \bar{\Theta}^{[i]}\right) \leq \max\left(\varepsilon_{rel}, tol\right)$$

dove ε_{rel} è la massima accuratezza relativa [Murli07] a $L\left(x_i, \bar{\Theta}^{[i]}\right)$ e *tol* è una tolleranza richiesta.

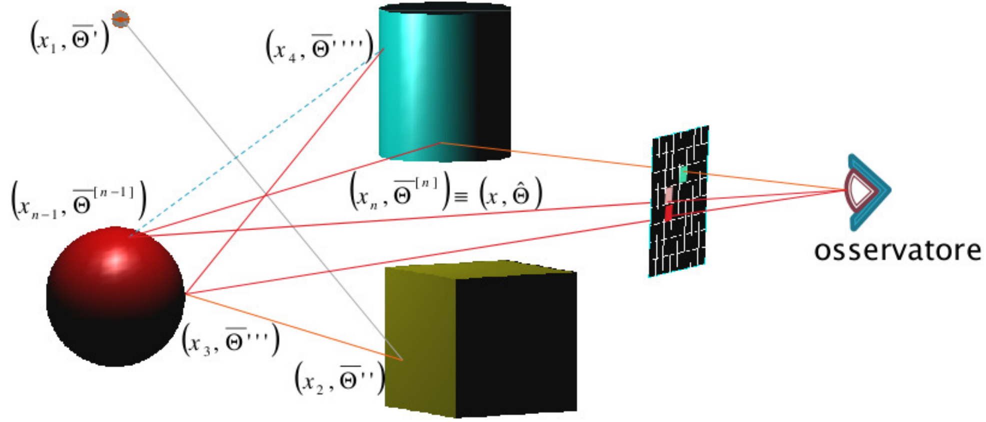

Figura 3.4.2: Calcolo del flusso in corrispondenza di ogni x_i

Generando più percorsi di luce, e quindi applicando più volte il procedimento iterativo, si aumenta progressivamente l'accuratezza dei valori calcolati del flusso in ogni $pixel_k$. Infatti generando un nuovo percorso di luce, la formula di quadratura per alcuni pixel sarà più accurata, ed il valore calcolato tenderà sempre più al valore dell'integrale.

Proposizione 3.10. *Sia $\bar{\Phi}(pixel_k, n_k)$ il valore calcolato del flusso nel $pixel_k$ all'n-mo percorso di luce. Allora si ha che:*

$$\Phi(pixel_k) = \lim_{n_k \to \infty} \bar{\Phi}(pixel_k, n_k)$$

Lemma 3.4. *Se vale la 3.4.6, allora si ha:*

$$\lim_{n_k \to \infty} \left\| \bar{\Phi}(pixel_k, n_k) - \bar{\Phi}(pixel_k, n_k - 1) \right\| = 0$$

Dimostrazione. Dalla 3.4.6, $\forall n_k \epsilon \mathbb{N}$ si ha:

$$\left\| \bar{\Phi}(pixel_k, n_k) - \bar{\Phi}(pixel_k, n_k - 1) \right\| \leq \left\| \bar{\Phi}(pixel_k, n_k - 1) - \bar{\Phi}(pixel_k, n_k - 2) \right\|$$

Ma poiché

$$\left\| \bar{\Phi}(pixel_k, n_k) - \bar{\Phi}(pixel_k, n_k - 1) \right\| = \left\| \frac{1}{n_k} L\left(z_{n_k}, \tilde{\Theta}^{[n_k]}\right) \right\|$$

e poiché

$$\lim_{n_k \to \infty} \left\| \frac{1}{n_k} L\left(z_{n_k}, \tilde{\Theta}^{[n_k]}\right) \right\| = 0 \tag{3.4.7}$$

allora si ha la tesi. □

Proposizione 3.11. *Sia Npixel il numero di pixel presenti nell'immagine risultato, sia*

$$n_{tot} = \sum_{k=1}^{Npixel} n_k$$

e sia

$$\bar{\Phi}_{n_{tot}}^{med} = \frac{1}{Npixel} \sum_{k=1}^{Npixel} \left\| \bar{\Phi}\left(pixel_k, n_k\right) - \bar{\Phi}\left(pixel_k, n_k - 1\right) \right\|$$

Allora si ha che:

$$\lim_{n \to \infty} \bar{\Phi}_{n_{tot}}^{med} = 0$$

La quantità $\bar{\Phi}_{n_{tot}}^{med}$ rappresenta il contributo medio al flusso calcolato nei $pixel_k$, apportato dalla radianza in n_{tot} punti di A raggiunti dai percorsi di luce.

Questa proprietà ci potrebbe portare a pensare ad un criterio di arresto per il procedimento iterativo di generazione dei percorsi di luce del seguente tipo:

Condizione 2. Arresta il procedimento iterativo di generazione di percorsi di luce dopo n_{tot} valutazioni se

$$\bar{\Phi}_{n_{tot}}^{med} \leq \max\left(\varepsilon_{rel}, tol\right)$$

dove ε_{rel} è la massima accuratezza relativa a $\bar{\Phi}_{n_{tot}}^{med}$ e tol è una tolleranza richiesta.

Data però la 3.4.7, si comprende che la convergenza di $\bar{\Phi}_{n_{tot}}^{med}$ dipende strettamente dal crescere degli n_k ad ogni iterazione.

Si può quindi pensare di semplificare la cond.2 con la seguente:

Condizione 3. Sia

$$m = \frac{1}{n_{tot}} \sum_{l=1}^{Npixel} n_k \tag{3.4.8}$$

la media delle valutazioni eseguite su tutti i pixel. Arresta il procedimento iterativo di generazione di percorsi di luce dopo n_{tot} valutazioni se

$$\frac{1}{m} \leq \max\left(\varepsilon_{rel}, tol\right) \tag{3.4.9}$$

dove ε_{rel} è la massima accuratezza relativa a $\frac{1}{m}$ e tol è una tolleranza richiesta.

L'algoritmo del metodo delle direzioni si può descrivere come di seguito:

```
ripeti

    SEGUI UN PERCORSO DI LUCE:
    calcola la radianza della sorgente di luce
            in una direzione theta

    ripeti

        trova il punto x intersecato dal raggio con
               direzione theta

        calcola la radianza in x

        trova il pixel k attraverso cui x è visibile
               all'osservatore

        aggiorna il valore del flusso in k

        scegli una nuova direzione theta

     finché la radianza in x è minore di una tolleranza

finché il contributo medio al flusso nei pixel è minore
       di una tolleranza
```

Capitolo 4

Implementazione su GPU

4.1 Caratteristiche dell'ambiente computazionale

L'ambiente computazionale scelto per l'implementazione del metodo risolutivo è quello tipico delle applicazioni interattive della grafica computazionale. Si tratta di una workstation dotata di processore multi-core e Graphics Processing Unit (GPU).

Le caratteristiche tecniche della macchina utilizzata sono elencate in tabella 4.1.1.

4.1.1 Cenni sull'architettura GPU

La scheda grafica NVIDIA Quadro FX 3800 è dotata di GPU con architettura denominata GT200.

L'unità di calcolo, ovvero un core della GPU, è lo *Streaming Processor* (SP), rappresentato in figura 4.1.1. Al suo interno sono disponibili tre unità di

CPU	Intel Core i5-650: 2 core, Hyperthreading (4 thread indipendenti), 3.2 GHz, 64-bit
Memoria	6 GB DDR3-1066
Hard Disk	280,6 GB
GPU	NVIDIA Quadro FX 3800: 192 core, 1GB memoria locale, 1.2 GHz

Tabella 4.1.1: Caratteristiche tecniche della workstation

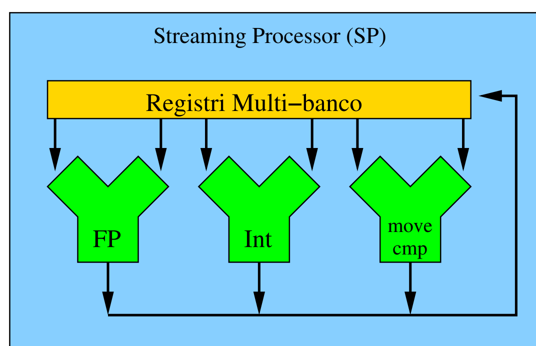

Figura 4.1.1: Schema dello Streaming Processor (SP)

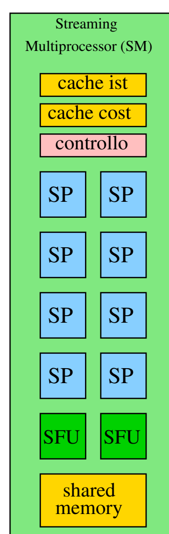

Figura 4.1.2: Schema dello Streaming Multi Processor (SM)

esecuzione, anche se non è presente alcuna memoria cache. Infatti ricordiamo che è stata progettata per eseguire, nell'ambito dell'illuminazione locale, operazioni ripetitive su stream di vertici o di pixel.

Di per sé un SP non è autonomo: per poter eseguire operazioni necessita di altri componenti quali una memoria cache e l'unità di controllo, come se fosse l'equivalente di una ALU (Unità Logico Aritmetica) all'interno di una CPU. Poiché le operazioni della pipeline grafica per l'illuminazione locale sono altamente parallelizzabili, NVIDIA ha progettato delle unità chiamate *Streaming Multi Processor* (SM), ognuna delle quali contiene (Fig.4.1.2):

- otto SP,

- due unità per le funzioni speciale (SFU), cioè per il calcolo trascendente (sen, cos, ...) e l'interpolazione (per i filtri sulle texture),
- una unità per il controllo delle operazioni (CU),
- una memoria condivisa (shared memory di 16 KB) tra gli SP,
- una memoria cache per per le costanti (di sola lettura, 8 KB),
- una memoria cache per le istruzioni (8 KB).

È da notare che le aree di memoria sono di piccola dimensione poiché il problema dell'illuminazione locale implementato nella pipeline grafica non richiede grande località di dati.

All'interno di un SM, tutti gli SP eseguono lo stesso identico set di istruzioni contemporaneamente, oppure rimangono inattivi, mentre gli SFU possono operare in maniera asincrona.

Anche gli SM sono raggruppati in blocchi di tre unità. Un blocco è chiamato Texture/Processor Cluster (TPC) e, oltre ai tre SM, contiene una unità di controllo per le geometrie, ed un blocco di memoria per le texture[1] (Fig.4.1.3).

Nel contesto della pipeline grafica per il rendering con illuminazione locale, la proiezione di una texture su di un oggetto è una operazione abbastanza onerosa in termini computazionali. Da cui si comprende la necessità di una unità parallela specifica come il TPC.

Nella NVIDIA Quadro FX 3800 sono raggruppati otto TPC in un grande Streaming Processor Array, che include anche una memoria globale condivisa da tutte le unità (Fig.4.1.4).

4.1.2 Modello di programmazione

Dal punto di vista della programmazione, l'ambiente computazionale scelto si può interpretare come una macchina costituita da un *host*, che sarebbe un calcolatore dotato di CPU tradizionale, e da un *device*, che è una GPU come quella descritta precedentemente. Nel nostro caso (tab.4.1.1), le due componenti comunicano tramite un canale PCIe con larghezza di banda di 12.8 GB/s.

[1] Le texture sono immagini che possono essere proiettate sulle superfici degli oggetti tridimensionali per aumentare il realismo del dettaglio: in genere sono costituite da fotografie ritraenti il materiale da rappresentare.

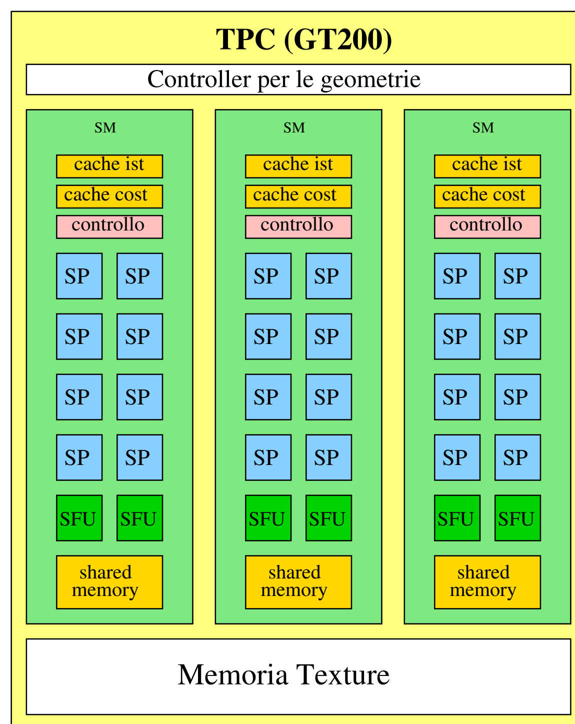

Figura 4.1.3: Schema del Texture/Processor Cluster (TPC)

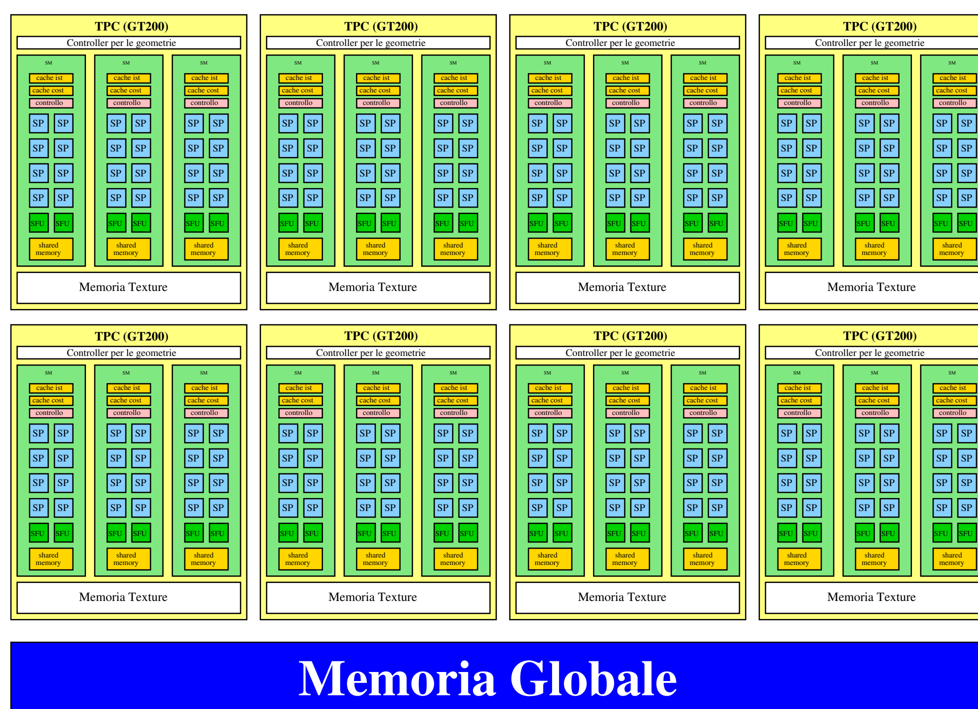

Figura 4.1.4: Schema dello Streaming Processor Array

Le opportunità date da un siffatto ambiente permettono di sviluppare software in grado di sfruttare la potenza di calcolo del device per sequenze di codice altamente parallelizzabili, ovvero con proprietà intrinseche di *data-parallelism*. Risulterebbe così possibile eseguire, con più SM contemporaneamente, sequenze di operazioni indipendenti su grandi strutture di dati.

Un tipico esempio di data-parallelism è dato dalle operazioni del prodotto tra matrici (Fig.4.1.5). Se, ad esempio, si vogliono moltiplicare due matrici M e N, con $dim\,(M) = dim\,(N) = 1000$, si dovranno eseguire 1000000 prodotti scalari indipendenti, ognuno consistente in 1000 moltiplicazioni e 1000 addizioni.

Il linguaggio scelto per l'implementazione del software di questo lavoro è *CUDA* (Compute Unified Device Architecture), creato per programmare i device NVIDIA. È strutturato in modo da poter organizzare il codice in sezioni chiamate *kernel*.

CUDA è un linguaggio che ben si integra con il linguaggio C. Infatti nel codice per l'host costituisce una estensione del C, fornendo comandi sia per la gestione del trasferimento dei dati tra host e device, sia per il controllo ad alto livello delle esecuzioni dei kernel sul device.

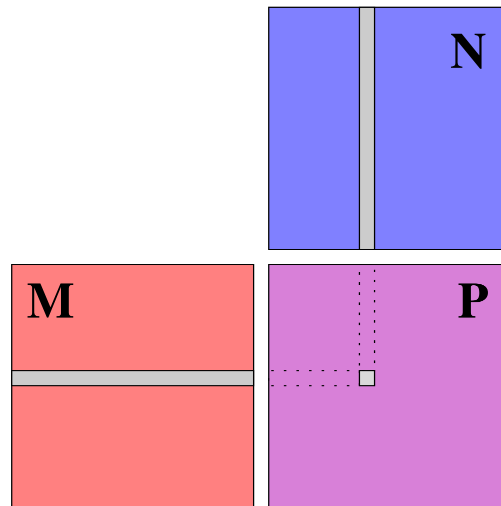

Figura 4.1.5: Data Parallelism nel prodotto tra matrici

Un tipico software eseguibile su GPU prevede diverse fasi: le sequenze di operazioni che non presentano un evidente data-parallelism sono in genere eseguite sull'host, ovvero dalla CPU; le operazioni suddivisibili in diversi task indipendenti invece sono eseguite in parallelo sul device, ovvero dalla GPU (Fig. 4.1.6).

4.1.2.1 Blocchi di thread

L'esecuzione dei kernel sulla GPU è organizzata in una struttura di *thread*, cioè entità di flusso di istruzioni eseguibili indipendentemente.

I thread sono raggruppati formalmente in *blocchi*, i quali costituiscono poi la *griglia* di tutti i thread che eseguono un kernel. Al momento del lancio dell'esecuzione di un kernel è quindi necessario specificare la dimensione della griglia in termini di numero di blocchi, e la dimensione dei singoli blocchi in termini del numero di thread. Quando l'esecuzione di tutti i thread di una griglia termina, il controllo torna al codice sull'host.

Ogni blocco viene eseguito da un singolo SM, pertanto l'organizzazione in blocchi di thread è utile per la gestione dei differenti livelli di memoria e la sincronizzazione:

- i singoli thread hanno accesso esclusivo a registri e memorie locali (lettura/scrittura);

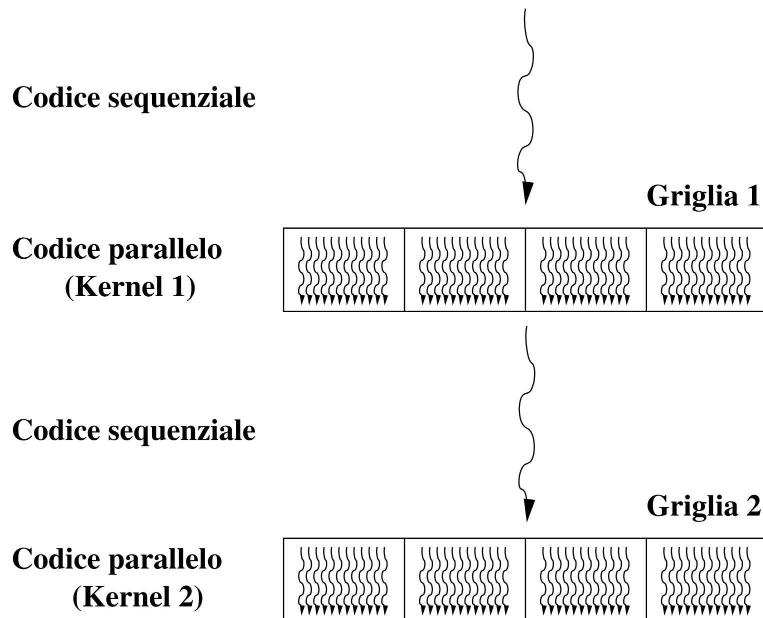

Figura 4.1.6: Esecuzione di un programma su GPU con CUDA

- i thread all'interno di un blocco condividono la shared-memory (lettura/scrittura) e possono sincronizzare la loro esecuzione;
- tutti i thread all'interno di una griglia condividono la memoria globale (lettura/scrittura) e la memoria costante (solo lettura);
- due thread in blocchi differenti non possono cooperare nell'esecuzione;
- Il codice host può solo eseguire operazioni di lettura e scrittura nella memoria globale e nella memoria costante (operazioni asincrone).

Un'ultima caratteristica utile in merito alla programmazione delle GPU è la possibilità di scrivere programmi in C che integrino sia CUDA sia OpenGL per il controllo delle esecuzioni grafiche sulla GPU. In tal modo è possibile visualizzare direttamente i risultati di operazioni eseguite sul device senza la necessità di trasferirli prima sull'host e poi di nuovo sul device con OpenGL. Il processo di visualizzazione dei dati grafici può essere quindi eseguito con grande efficienza.

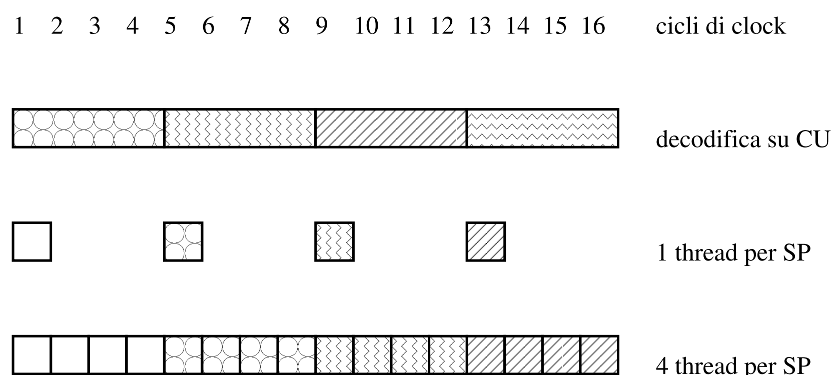

Figura 4.2.1: Esecuzione di un flusso di istruzioni su un SP. Prima riga: tempo di decodifica delle istruzioni. Seconda riga: tempo di esecuzione di un thread. Terza riga: tempo di esecuzione di quattro thread.

4.2 Parametri di valutazione delle prestazioni

Cerchiamo ora di comprendere quanto l'organizzazione in blocchi di thread influisca sulle prestazioni dei software su GPU.

Come abbiamo visto nella sez.4.1.1, in ogni SM è presente una sola CU. Per poter eseguire una nuova operazione, gli SP devono aspettare la decodifica della successiva istruzione da parte della CU.

Esiste però un divario tra la velocità di esecuzione di una operazione MAD (Multiply-And-Add) da parte dell'SP (1 ciclo di clock), e la velocità di decodifica di una istruzione da parte della CU (4 cicli di clock) (Fig.4.2.1).

Per ottimizzare l'utilizzo degli SM, quindi, un kernel in esecuzione è organizzato in raggruppamenti di thread:

Definizione 4.1. Sia q il numero di Streaming Processor (SP) all'interno di uno Streaming Multi Processor (SM). Sia d il numero di cicli di clock necessari alla unità di controllo (CU) dello SM per decodificare una istruzione ed inviarla agli SP. La quantità:

$$w = d * q$$

è il numero di thread attivi sullo SM durante la decodifica di una istruzione. Un gruppo di w thread in esecuzione su di un SM durante la decodifica di una istruzione è detto *warp*.

Il warp rappresenta il numero di thread che un SM può eseguire contemporaneamente. Se si richiede che vengano eseguiti meno di w thread, il

tempo impiegato non cambierà. Pertanto si può dire che *il warp è l'unità di esecuzione di un SM.*

Cerchiamo ora di definire il massimo numero di thread in esecuzione su una GPU:

Definizione 4.2. Sia p il numero di Streaming Multi Processor (SM) presenti su di una GPU, e sia w la dimensione di un warp su di un SM. Allora:

$$TH_{max} = p * w$$

è il numero di massimo di thread in esecuzione concorrente sulla GPU nel tempo di decodifica di una istruzione.

Ricordando le definizioni di Speed-up ed Efficienza di un algoritmo parallelo, come descritte in [Murli06], possiamo affermare che TH_{max} rappresenta lo speed-up ideale raggiungibile su una GPU.

A questo punto possiamo introdurre i seguenti concetti:

Definizione 4.3. Sia A_{seq} un algoritmo sequenziale per risolvere con una CPU un problema P costituito da n_{seq} operazioni e m_{seq} accessi in memoria. Se t^c_{seq} è il tempo per eseguire una operazione e t^m_{seq} è il tempo per un accesso in memoria, allora il tempo di esecuzione di A_{seq} è:

$$T(A_{seq}) = n_{seq} \cdot t^c_{seq} + m_{seq} \cdot t^m_{seq}$$

Allo stesso modo:

Definizione 4.4. Sia A_{gpu} un algoritmo parallelo su GPU per risolvere, utilizzando SP core, un problema P costituito da n_{gpu} operazioni per thread e m_{gpu} accessi in memoria per thread. Se t^c_{gpu} è il tempo per eseguire w operazioni e t^m_{gpu} è il tempo per un accesso in memoria, allora il tempo di esecuzione di A_{gpu} è:

$$T(A_{gpu}) = \left(\frac{n_{gpu}}{w} \cdot t^c_{gpu} + m_{gpu} \cdot t^m_{gpu}\right) \cdot SP + O_h$$

dove O_h è l'*Overhead totale* dell'esecuzione.

Lo Speed-up per algoritmi su GPU diventa quindi:

$$S_{SP} = \frac{T(A_{seq})}{T(A_{gpu})} \tag{4.2.1}$$

Mentre l'Efficienza si misura con:

$$E_{SP} = \frac{S_{SP}}{SP} \tag{4.2.2}$$

Studi sperimentali ([D'Amore11, Demmel08, Volkov10-1, Volkov10-2]) hanno mostrato che $O_h$varia in relazione ad alcuni parametri, quali:

1. la frazione di operazioni eseguite sequenzialmente;
2. l'occupazione degli SM da parte dei blocchi di thread assegnati;
3. il numero di registri usati dal kernel;
4. la quantità di shared memory allocata dal kernel:
5. la coalescenza degli accessi in memoria globale da parte di un warp;
6. la divergenza dei thread all'interno di un blocco.

Cerchiamo di analizzare brevemente ogni punto:

1. Nell'organizzazione dell'algoritmo parallelo non è possibile distribuire tra le unità processanti tutte le operazioni. Questo è particolarmente vero per gli algoritmi su GPU, dove il controllo di flusso ad alto livello è eseguito dalla CPU. La legge di *Ware-Amdhal* [Ware83] ci aiuta a comprendere come la frazione sequenziale pesi molto nel calcolo dello speed-up quando si utilizzi un elevato numero di unità processanti.
2. La situazione di occupazione ideale di un SM si ha quando un blocco di thread occupa il massimo numero di warp gestibili contemporaneamente dallo scheduler dell'SM. In tal caso si potrebbe verificare un mascheramento delle latenze per gli accessi alla memoria ed un overhead basso.
3. Se un kernel utilizza pochi registri di memoria, probabilmente fa maggior uso di locazioni di memoria più lente. Pertanto ci può essere una degradazione delle prestazioni. D'altro canto una maggior richiesta di registri di memoria da parte dei thread può limitare l'occupazione di un SM, perché potrebbe non essere possibile gestire il massimo numero di warp a causa di un esaurimento di locazioni disponibili.

4. La shared memory è una memoria molto veloce. Quando è allocata da un kernel può essere uno strumento utile nel caso in cui i thread di un blocco debbano scambiare dati tra loro. Nel caso però in cui più thread cerchino di accedere contemporaneamente a locazioni di memoria di uno stesso segmento, si verifica un conflitto. Questo introduce una latenza che può influire negativamente sul tempo di esecuzione. Per alcuni algoritmi una riorganizzazione delle operazioni può limitare questo problema.

5. La memoria globale è sia la memoria di maggiori dimensioni, sia quella più lenta a disposizione dei thread. Se più thread si un warp cercano di leggere locazioni di memoria di uno stesso segmento (coalescenza), la lettura avviene più rapidamente.

6. Ogni SM ha un counter di avanzamento per ogni thread, pertanto i thread possono eseguire operazioni diverse. Se però si considera l'organizzazione in warp, si capisce che la situazione ottimale si ha quando tutti i thread di un warp eseguono la stessa operazione contemporaneamente (SIMD [Flynn66]). In caso contrario, se t thread seguono flussi di operazioni divergenti, le esecuzioni sono intercalate, ed è necessario aspettare $d \cdot t$ cicli di clock per eseguire l'operazione successiva.

4.3 Individuazione dei metodi con le migliori prospettive prestazionali

Dopo aver considerato le caratteristiche architetturali del sistema su cui deve essere implementato l'algoritmo, e tenuto conto sia del modello di programmazione utilizzato, sia dei parametri di valutazione, si procede ora all'analisi dei vari metodi introdotti nel Cap. 3 dal punto di vista sia qualitativo, sia prestazionale.

4.3.1 Raytracing

Si tratta di un metodo basato sul calcolo diretto dei valori dell'immagine risultato (cfr. sez.3.2.1).

Da una prima analisi è immediatamente immaginabile un partizionamento del carico di lavoro tra thread: il flusso di operazioni di raytracing per un pixel può essere assegnato ad un thread.

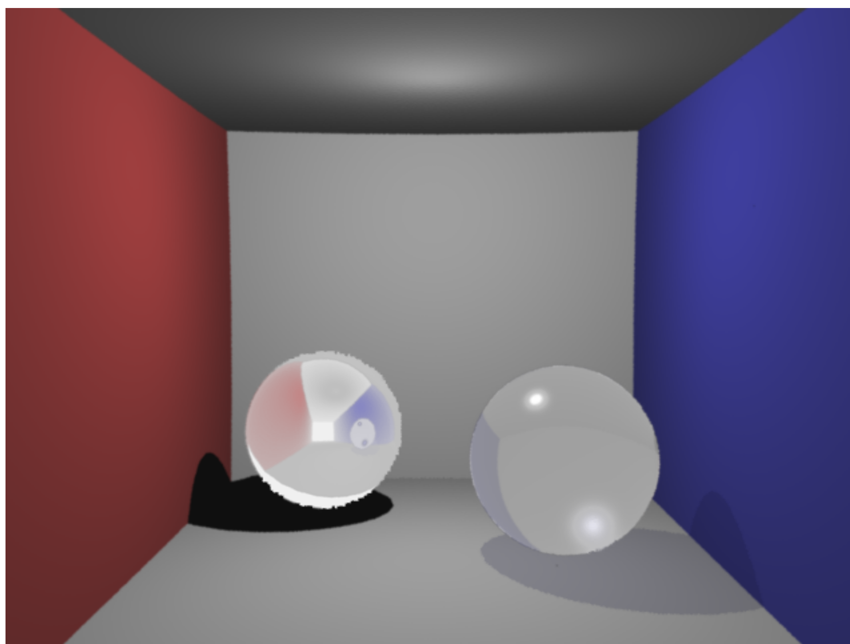

Figura 4.3.1: Scena con rendering di tipo raytracing

Avendo abbastanza memoria per conservare il modello descrittivo della scena nella memoria globale, i thread possono aggiornare ad ogni passo i valori dei pixel dell'immagine. La coalescenza per gli accessi alla memoria dei pixel è garantita, mentre non è realizzata per l'accesso ai dati della scena.

Ma la divergenza indotta dal moltiplicarsi dei raggi ad ogni passo è un fattore di grande inefficienza. Difatti CUDA supporta gli algoritmi ricorsivi solo per le GPU di ultima generazione.

Inoltre qualitativamente il modello del raytracing non permette la visualizzazione di scene con superfici diffusive (Fig. 4.3.1).

4.3.2 Radiosity

Si tratta di un metodo basato sulla suddivisione in patch del modello della scena e del calcolo di una soluzione indipendente dal punto di vista (cfr. sez.3.2.2).

Memorizzando il modello rappresentante la scena nella memoria globale, è anzitutto necessario partizionare il modello in patch. Questa operazione non è ben parallelizzabile, e spesso richiede di conoscere a priori le zone in cui procedere con una partizionamento a grana fine.

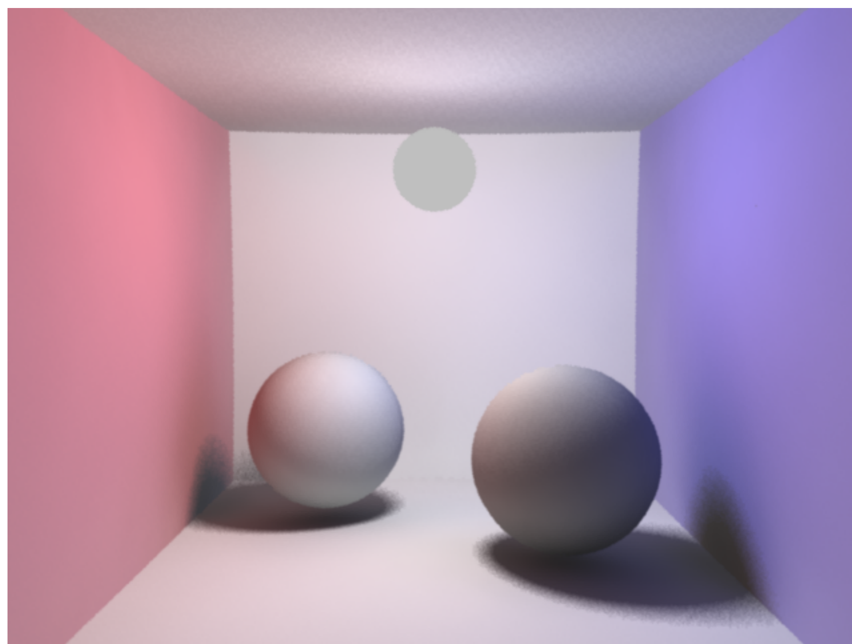

Figura 4.3.2: Scena con Rendering di tipo Radiosity

Successivamente è possibile distribuire il calcolo dei fattori di forma tra thread indipendenti. Le sequenze di operazioni in questa fase possono essere implementate in maniera ottimale per un calcolatore SIMD, pertanto ci si aspetta che le prestazioni siano buone sulle GPU.

La fase della soluzione del sistema lineare per il calcolo del valore di radiosity per tutte le patch ha una grande complessità computazionale. Gli algoritmi per questo tipo di problemi non sono ottimali per le GPU, ma esistono diverse strategie per l'implementazione di software efficienti [Tomov10, Krüger05].

Infine è necessaria una fase per la visualizzazione delle patch e l'interpolazione dei valori di radiosity. A tal fine è possibile utilizzare gli strumenti OpenGL implementati a livello hardware nella GPU.

Sebbene le prospettive prestazionali siano interessanti, perché raffinando il metodo e sfruttando alcune caratteristiche hardware è possibile progettare un algoritmo dalle prestazioni promettenti, purtroppo la resa grafica del metodo radiosity è del tutto insufficiente, essendo inadatta a rappresentare superfici speculari o trasparenti (Fig.4.3.2).

4.3.3 Metodi Stocastici

In questo caso (cfr. sez.3.3) la resa grafica delle scene è molto foto-realistica e preferibile a quella dei metodi precedenti (Fig.4.3.3).

Il metodo consiste nel seguire un certo numero di cammini casuali dai pixel dell'immagine fino alla sorgente luminosa. Ogni cammino deve eseguire una sequenza di operazioni indipendenti, e pertanto l'esecuzione si presta molto bene alla suddivisione in thread indipendenti.

Supposto che il modello della scena sia memorizzato nella memoria globale, le operazioni di lettura del modello non possono sfruttare alcun criterio di coalescenza, data la natura casuale dei cammini: ogni thread di un blocco deve accedere ai dati relativi ad un punto diverso del modello.

Le istruzioni per il calcolo dei contributi al valore finale del flusso nel pixel sono le stesse per ogni cammino, e perciò possono essere organizzate come se si trattasse di un algoritmo SIMD. Può accadere, però, che thread nello stesso warp colpiscano oggetti con BRDF differenti. In tal caso si avrebbe una divergenza nel flusso di istruzioni influenzando negativamente sui tempi di esecuzione.

Per di più, ogni cammino si interrompe casualmente dopo un certo numero di passi. Quindi da quel momento l'occupazione dell'SM peggiorerebbe, aumentando l'overhead di esecuzione.

Infine, essendo la scrittura dei risultati nei pixel una operazione che avviene con tempistica casuale, nessun tipo di coalescenza può essere sfruttata in questa fase.

Si può quindi dire che, sebbene ci siano varie problematiche che peggiorano le prestazioni a causa di un elevato overhead totale nell'implementazione parallela su GPU, i metodi stocastici garantiscono una buona resa grafica e ci si aspetta comunque un miglioramento prestazionale sensibile.

4.3.4 Metodo delle direzioni

Anche in questo caso le scene rappresentabili con questo metodo possono essere di qualsiasi natura, anche se si ha una migliore convergenza nel caso di BRDF prevalentemente diffusive (cfr. sez.3.4.1).

Il metodo consiste nel seguire un certo numero di percorsi di luce, dalla sorgente fino a dispersione completa dell'energia. Ogni percorso deve eseguire una sequenza di operazioni indipendenti, e anche in questo caso l'esecuzione si presta molto bene alla suddivisione in thread indipendenti.

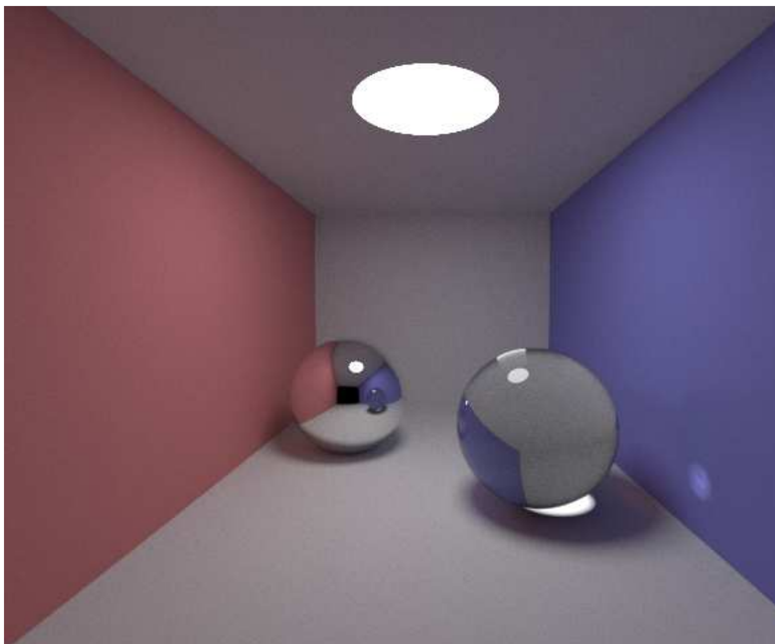

Figura 4.3.3: Scena con Rendering di tipo Stocastico

Come per i metodi stocastici, supposto che il modello della scena sia memorizzato nella memoria globale, le operazioni di lettura del modello non sfruttano alcun criterio di coalescenza, data la natura casuale delle direzioni scelte: ogni thread di un blocco leggerà i dati relativi al punto colpito dal raggio in una certa direzione, coinvolgendo un oggetto del modello non predeterminabile.

Le istruzioni per il calcolo della radianza al passo n, se gli oggetti colpiti hanno tutti la stessa BRDF, sono le stesse per ogni percorso di luce. Pertanto possono essere eseguite come se si trattasse di un algoritmo SIMD. Ma può accadere che thread nello stesso warp emettano raggi che colpiscono oggetti con BRDF differenti. In tal caso si ha una divergenza nel flusso di istruzioni che influisce negativamente sui tempi di esecuzione.

Poiché, però, l'energia in media si disperde dopo un numero di iterazioni che dipende dall'albedo media della scena, l'occupazione degli SM è mantenuta al massimo per quasi tutti i passi iterativi.

Infine, essendo la scrittura dei contributi al flusso dei pixel una operazione che avviene in maniera imprevedibile ad ogni passo, nessun tipo di coalescenza può essere garantita in questa fase.

4.3.5 Conclusioni

Con questa analisi a priori, possiamo affermare che certamente i metodi di Raytracing e Radiosity siano da scartare data la resa inaccurata dei fenomeni ottici dell'illuminazione globale.

Sia i Metodi Stocastici, sia il Metodo delle Direzioni sono molto simili dal punto di vista dell'impatto sull'overhead totale e delle prestazioni attese, anche se il Metodo delle Direzioni sembra essere più promettente dal punto di vista dell'occupazione degli SM.

Poiché però, nel caso di presenza di superfici riflettenti o trasparenti, i Metodi Stocastici producono una resa grafica migliore, sceglieremo di implementare un algoritmo ibrido:

- per il calcolo del flusso nei pixel in cui sono visibili superfici diffusive, si utilizza il Metodo delle Direzioni;
- per il calcolo del flusso nei pixel in cui sono visibili superfici speculari o trasparenti, si utilizza il Metodo Stocastico dei Cammini dal Sensore alla Luce.

Algorithm 4.1 Algoritmo macroscopico del metodo delle direzioni

```
repeat
        theta=rand(direzione)
        PercorsoLuce(scena,theta,pixel,indiceQuad,
                num_pix,media)
until ((1/media)<max(eps_rel,tol))
```

4.4 Algoritmo sequenziale ibrido

Il metodo delle direzioni, che è quello preferito per l'implementazione nell'ambiente con GPU, si presta molto bene al caso delle scene dotate di superfici esclusivamente diffusive.

Nell'algoritmo descritto in Alg.4.1:

- `theta` rappresenta la direzione iniziale del percorso di luce originato in una sorgente luminosa;
- `PercorsoLuce` è la procedura (Alg.4.2) che esegue il percorso di luce attribuendo, per ogni punto colpito, il relativo valore di radianza al pixel corrispondente;
- `scena` (input) è la struttura dati contenente la descrizione del modello della scena;
- `pixel` (input/output) è l'array dei pixel che costituiscono l'immagine risultante dal rendering della scena;
- `indiceQuad` (input/ouput) è l'array che, ad ogni $pixel_k$, associa l'intero corrispondente al numero di contributi ricevuti nel calcolo del flusso;
- `num_pix` (input) è la lunghezza degli array `pixel` e `indiceQuad`;
- `media` (output) è la media del numero di contributi ricevuti da tutti i pixel dell'immagine durante il calcolo del flusso;
- `eps_rel` è la massima accuratezza relativa a `(1/media)`;
- `tol` è la tolleranza richiesta.

Osservando nel dettaglio il codice della procedura PercorsoLuce in Alg.4.2:

Algorithm 4.2 Algoritmo del metodo delle direzioni per ogni percorso di luce

```
procedure PercorsoLuce
 media=0
 x=rand(luce)
 L=emit(x,theta)
 repeat
        x=r(x,theta)
        if V(x,osservatore)=1 then
                ind_p=trova_pixel(x,osservatore)
                theta_pixel=trova_theta(x,osservatore)
                indiceQuad[ind_p]=indiceQuad[ind_p]+1
                pixel_new=(pixel[ind_p]+
                        +emit(x,theta_pixel)+
                        +H(x,theta_pixel,L))/
                        /indiceQuad[ind_p]
                endif
                pixel[ind_p]=pixel_new
        endif
        theta=nuova_direzione(diffusiva o riflessa)
        L=emit(x,theta)+H(x,theta,L)
 until (L<max(eps_rel,tol))
 for i=0 to num_pix
        media=media+indiceQuad[ind_p]
 endfor
 media=media/num_pix
end procedure
```

- `x` è il punto da cui origina la direzione verso la quale si vuole calcolare la radianza;
- `theta` è la direzione verso la quale si vuole calcolare la radianza;
- `r` è la funzione di raycasting, come definita nell'eq.2.2.2
- `L` è la radianza in `x` con direzione `theta`;
- `emit` è la funzione che restituisce la radianza emessa da `x`, ovvero L_e della eq.3.4.4;
- `H` è l'operatore $\mathcal{H}$ definito nell'eq.3.4.4;
- `V` è la funzione di visibilità, come definita in Def.2.6;
- `trova_pixel` è una funzione che identifica il pixel attraverso cui passa il raggio che dal punto `x` raggiunge l'`osservatore`;
- `trova_theta` è una funziona che identifica la direzione del raggio che dal punto `x` raggiunge l'`osservatore`;
- `pixel_new` è il nuovo valore del flusso passante per il `pixel[ind_p]`;
- `nuova_direzione` è una funzione che calcola una nuova direzione per il percorso di luce a seconda che il materiale sia diffusivo, con un metodo casuale, o sia speculare, e quindi con le leggi ottiche della riflessione/rifrazione;
- `eps_rel` è la massima accuratezza relativa a `L`;
- `tol` è la tolleranza richiesta.

Come abbiamo potuto vedere nel capitolo precedente, il metodo delle direzioni converge rapidamente nel caso di superfici diffusive, poiché ad ogni passo è possibile contribuire a calcolare il flusso luminoso che attraversa i pixel.

Nel caso però che venga incontrato un punto su di una superficie riflettente, a meno che il raggio riflesso non raggiunga esattamente l'osservatore, non è possibile calcolare il flusso per il pixel relativo.

In tal caso è più efficace applicare il metodo stocastico dei cammini casuali dal sensore alla luce, come descritto nella sez. 3.3.1.

L'algoritmo per questo caso è descritto in Alg.4.3:

Algorithm 4.3 Algoritmo dei cammini casuali dal sensore alla luce

```
repeat
  maxdiff=0
  for i=0 to num_pix
    theta=trova_theta(osservatore , pixel)
    x=r(osservatore , theta)
    throughput=0
    repeat
          theta_lux=trova_theta(x,luce)
          lux=r(x,theta_lux)
          L=emit(lux , theta_lux)*BRDF(x,theta_lux , theta)*
              *coseno(x,theta_lux)
          throughput=throughput+L
          theta=nuova_direzione(diffusiva o riflessa)
          x=r(x,theta)
    until assorbimento
    if abs(throughput-pixel[i])>maxdiff
                maxdiff=abs(throughput-pixel[i])
    endif
    pixel[i]=throughput
  endfor
until (maxdiff<max(eps_rel , tol))
```

- `theta_lux` è la direzione del raggio che dalla sorgente luminosa raggiunge il punto `x`;
- `lux` è un punto sulla sorgente luminosa;
- `L` è la radianza che lascia il punto `x` nella direzione `theta`;
- `BRDF` è la funzione di distribuzione di riflettanza, come definita nell'equazione 2.3.2;
- `coseno` è la funzione che calcola il coseno dell'angolo formato tra la normale alla superficie in `x` e la direzione `theta_lux`;
- `throughput` è il valore progressivo della radianza in `pixel[i]`;
- `assorbimento` è una funzione che è vera se viene estratto un numero prescelto con un procedimento di roulette russa;
- `maxdiff` (output) è il valore del massimo contributo ricevuto tra tutti i pixel dell'immagine durante il calcolo del flusso;
- `eps_rel` è la massima accuratezza relativa a `maxdiff`;
- `tol` è la tolleranza richiesta.

Per far sì che si possa ottenere l'immagine di migliore qualità con le migliori prestazioni, è stato progettato un algoritmo ibrido che includa entrambi i metodi.

Uno schema completo è proposto in Alg.4.4 e Alg.4.5. In questo procedimento si alternano le due fasi di calcolo per il flusso nei pixel relativi alle superfici diffusive, e per il flusso nei pixel relativi alle superfici speculari:

- `maxdiff` è il valore del massimo contributo ricevuto tra tutti i pixel dell'immagine durante il calcolo del flusso in entrambe le fasi. All'inizio di ogni fase viene azzerato, ed è quindi una misura relativa alle operazioni eseguite durante quella fase;
- `maxdiff_gen` è il valore del massimo contributo ottenuto nella fase precedente. Si richiede che nella fase attiva si continui a raffinare il valore del flusso nei pixel fino al raggiungimento del valore ottenuto nella fase precedente. È inizializzato al massimo valore numerico assegnabile ai pixel;

Algorithm 4.4 Algoritmo ibrido (parte 1): combinazione del metodo delle direzioni e del metodo dei cammini casuali dal sensore alla luce

```
flag=0
maxdiff_gen=255
repeat
  media=0
  if flag=0 then
   repeat
    maxdiff=0
    x=rand(luce)
    theta=rand(direzione)
    L=emit(x,theta)
    repeat
     x=r(x,theta)
     if V(x,osservatore)=1 AND diffuse(x) then
      ind_p=trova_pixel(x,osservatore)
      theta_pixel=trova_theta(x,osservatore)
      indiceQuad[ind_p]=indiceQuad[ind_p]+1
      pixel_new=(pixel[ind_p]+emit(x,theta_pixel)+
                +H(x,theta_pixel,L))/indiceQuad[ind_p]
      if abs(pixel_new-pixel[ind_p])>maxdiff
          maxdiff=abs(pixel_new-pixel[ind_p])
      endif
      pixel[ind_p]=pixel_new
     endif
     theta=nuova_direzione(diffusiva o riflessa)
     L=emit(x,theta)+H(x,theta,L)
    until (L<max(eps_L,tol_L))
   until (maxdiff<maxdiff_gen)
   maxdiff_gen=maxdiff
   flag=1
  else
```

Algorithm 4.5 Algoritmo ibrido (parte 2): combinazione del metodo delle direzioni e del metodo dei cammini casuali dal sensore alla luce

```
  else
   repeat
    maxdiff=0
    for i=1 to num_pix
     theta=trova_theta(osservatore,pixel)
     x=r(osservatore,theta)
     if reflect(x) then
      throughput=0
      repeat
        theta_lux=trova_theta(x,luce)
        lux=r(x,theta_lux)
        L=emit(lux,theta_lux)*
            *BRDF(x,theta_lux,theta)*coseno(x,theta_lux)
            throughput=throughput+L
            theta=nuova_direzione(diffusiva o riflessa)
            x=r(x,theta)
      until assorbimento
      if abs(throughput-pixel[i])>maxdiff
                  maxdiff=abs(throughput-pixel[i])
      endif
      pixel[i]=throughput
     endif
    endfor
   until (maxdiff<maxdiff_gen)
   maxdiff_gen=maxdiff
   flag=0
  endif
  for i=0 to num_pix
        media=media+indiceQuad[ind_p]
  endfor
  media=media/num_pix
until ((1/media)<max(eps_med,tol_med))
```

- `flag` è utilizzato per alternare le due fasi. In tal modo la progressiva qualità visiva dell'immagine è omogenea, con lo stesso livello di precisione sia per le superfici diffusive, sia per le superfici speculari;
- `eps_med` è la massima accuratezza relativa a `(1/media)`;
- `tol_med` è la tolleranza richiesta per `(1/media)`.
- `eps_L` è la massima accuratezza relativa a `L`;
- `tol_L` è la tolleranza richiesta per `L`;

Si noti che il criterio di arresto di entrambe le fasi è lo stesso, basandosi sul principio di progressivo raffinamento della formula di quadratura. Le due fasi si alternano in modo da mantenere omogenea la qualità dell'immagine con il procedere delle iterazioni. Poiché la natura dei metodi nelle due fasi non è la stessa, si è preferito usare questa tecnica per approssimare qualitativamente il risultato visivo.

L'algoritmo termina quando la formula di quadratura su tutti i pixel ha raggiunto in media una precisione relativa ad una tolleranza, o alla massima accuratezza relativa (eq. 3.4.9).

4.5 Algoritmo parallelo

Nel ricercare le procedure che possono trarre vantaggio da una parallelizzazione, e ricordando le considerazioni fatte nella sez.4.2, si possono immediatamente individuare in Alg.4.4 due cicli iterativi che eseguono operazioni indipendenti.

Il primo è quello relativo ai percorsi di luce: ogni percorso di luce può essere eseguito da un thread indipendente, come evidenziato in Alg.4.2.

Il secondo è quello relativo ai cammini casuali dal sensore alla luce: ogni percorso calcola un contributo alla formula di quadratura del flusso di uno specifico pixel, e può essere eseguito anch'esso indipendentemente da un processore, come visto in Alg.4.3.

Entrambi i cicli iterativi fanno gran uso di numeri casuali con distribuzione uniforme. Poiché in CUDA non è implementata una funzione per la generazione dei numeri casuali, una strategia possibile è quella di generare in parallelo una lunga sequenza di numeri casuali con distribuzione uniforme $U(0,1)$, da far utilizzare poi parzialmente ad ogni thread.

Uno schema dell'algoritmo parallelo organizzato in kernel è proposto in Alg.4.6:

Algorithm 4.6 Algoritmo ibrido parallelo organizzato in kernel

```
flag=0
maxdiff_gen=255
repeat
  media=0
  Kernel_GenRand(random_arr,num_rand)
  if flag=0 then
        repeat
          maxdiff=0
      ind_ran=rand(num_rand)
      Kernel_emit(luce,theta_arr,L_arr,random_arr,ind_rand)
      Kernel_PercorsoLuce(scena,theta_arr,L_arr,pixel,indiceQuad,
              num_pix,maxdiff,random_arr,num_rand,ind_rand)
    until (maxdiff<maxdiff_gen)
    maxdiff_gen=maxdiff
    flag=1
  else
    repeat
      maxdiff=0
      ind_ran=rand(num_rand)
      Kernel_CamminoCasuale(scena,pixel,indiceQuad,num_pix,
              maxdiff,random_arr,ind_rand)
    until (maxdiff<maxdiff_gen)
   maxdiff_gen=maxdiff
   flag=0
  endif
  for i=0 to num_pix
        media=media+indiceQuad[ind_p]
  endfor
  media=media/num_pix
until ((1/media)<max(eps_med,tol_med))
```

`Kernel_GenRand` è il kernel che si occupa della generazione di una sequenza di numeri casuali con distribuzione uniforme $U(0,1)$:

- `random_arr` (output) è l'array della sequenza di numeri casuali
- `num_rand` (input) è la lunghezza dell'array `random_arr`.

`Kernel_emit` è il kernel che si occupa di inizializzare un array di valori di radianza, emettendo raggi dalla sorgente luminosa:

- `luce` (input) è il modello della sorgente luminosa;
- `theta_arr` (output) è l'array che contiene le direzioni dei raggi luminosi generati;
- `L_arr` (output) è l'array che contiene la radianza iniziale dei percorsi di luce;
- `ind_rand` (input) è un indice casuale che identifica l'inizio della sequenza all'interno di `random_arr` che verrà utilizzata nel kernel.

`Kernel_PercorsoLuce` è il kernel che genera i percorsi di luce fino all'esaurimento dell'energia radiante. Ad ogni passo viene aggiornato il valore del flusso per i pixel corrispondenti agli oggetti diffusivi incontrati dal percorso di luce.

- `scena` (input) è il modello che descrive la scena;
- `pixel` (input/output) è l'array dei pixel dell'immagine finale;
- `indiceQuad` (input/output) è l'array che ad ogni pixel associa il numero di contributi ricevuti nel calcolo del flusso;
- `num_pix` (input) è la lunghezza degli array `pixel` e `indiceQuad`;
- `maxdiff` (output) è il valore del massimo contributo ricevuto tra tutti i pixel coinvolti durante il calcolo del flusso.

`Kernel_CamminoCasuale` è il kernel che segue i cammini casuali a partire dai pixel verso le sorgenti luminose. Viene aggiornato solo il flusso dei pixel corrispondenti a oggetti speculari/trasparenti. Al termine dell'esecuzione il flusso dei pixel sarà stato aggiornato per un numero casuale di volte.

Ogni kernel viene eseguito da una griglia di blocchi di thread. La configurazione di questa griglia sarà uno dei soggetti del prossimo capitolo.

Capitolo 5

Analisi dei risultati

5.1 Qualità delle immagini ottenute con il software sequenziale

È stato verificato sperimentalmente che la percezione della qualità delle immagini ottenute tramite il software ibrido dipende dalla tolleranza imposta nel criterio di arresto (Eq. 3.4.9)

Con valori alti si hanno immagini con più rumore, mentre con valori più bassi il rumore si attenua fino a scomparire.

In generale una tolleranza accettabile, dal punto di vista della percezione della qualità dell'immagine risultante, dipende dalla natura della scena, dal tipo di materiali utilizzati, e dalla posizione dell'osservatore e delle luci.

Nella Fig.5.1.1, vengono presentati i risultati per la scena test ($test1$), nota come Cornell Box [Goral84] con sole superfici diffusive, per differenti tolleranze. Data la natura della scena, il software esegue solo il codice relativo ai percorsi di luce.

Si può immediatamente notare che con una tolleranza molto alta, ovvero eseguendo poche iterazioni, si può già percepire la natura ottica della scena. All'aumentare della precisione richiesta, ovvero diminuendo la tolleranza, le immagini diventano sempre meno rumorose.

Nella Fig.5.1.2, vengono presentati i risultati per una scena test ($test2$) contenente sia superfici diffusive, sia superfici speculari, sia superfici trasparenti, per differenti tolleranze. In questo caso il software esegue entrambi i metodi: quello dei percorsi di luce e quello dei cammini casuali.

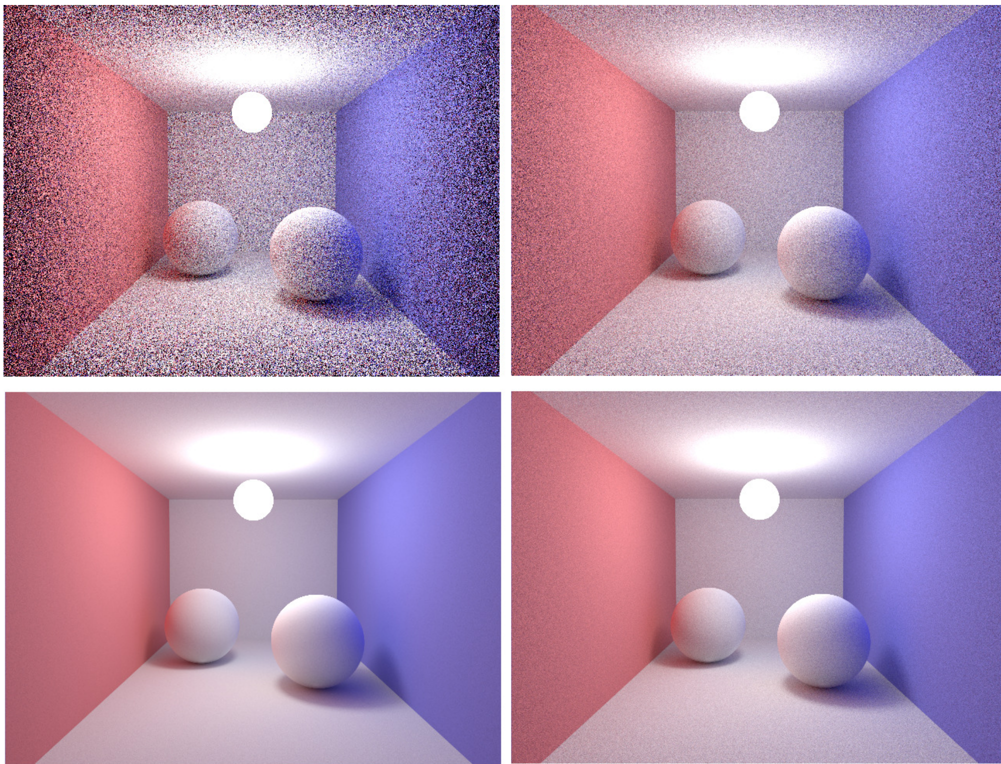

Figura 5.1.1: Scena con tutte superfici diffusive. In senso orario partendo dall'alto a sinistra: $tol = 0.1$, $tol = 0.01$, $tol = 0.001$, $tol = 0.0001$.

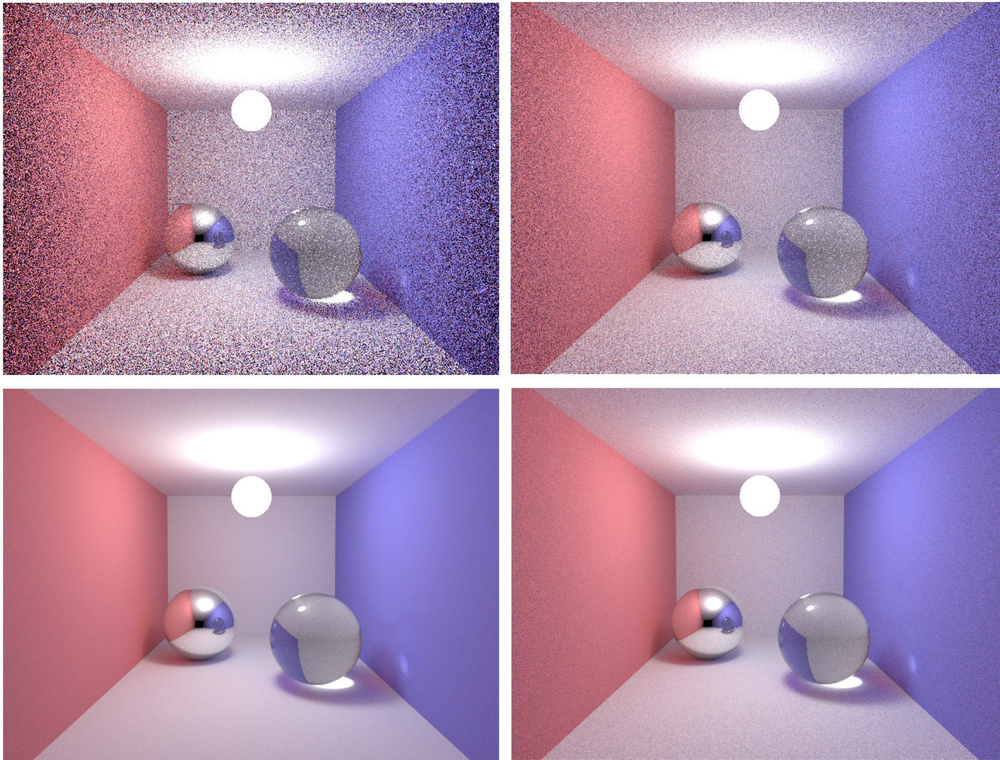

Figura 5.1.2: Scena con superfici diffusive, speculari e trasparenti. In senso orario partendo dall'alto a sinistra: $tol = 0.1$, $tol = 0.01$, $tol = 0.001$, $tol = 0.0001$.

Come prima, anche attraverso questi risultati si può osservare la robustezza del software, che infatti risulta in grado di evidenziare diversi effetti ottici già con pochissime iterazioni.

Nella Fig.5.1.3 si possono osservare due scene particolarmente complesse per la posizione della luce e per la presenza di superfici specchiate.

In entrambi i casi, valori di tolleranza alti danno risultati abbastanza insoddisfacenti. Nella scena con la sorgente di luce nascosta, quella a sinistra, le pareti sono quasi completamente in ombra. L'unica luminosità visibile è quella derivante dalle superfici diffuse che ricevono radianza dall'unica superficie speculare. Questa caratteristica richiede molte iterazioni affinché l'energia luminosa possa essere distribuita accuratamente su tutta la scena. Accade quindi che la media del numero di contributi al flusso nei pixel, m, crescerà molto lentamente. Ciò richiede tempi di esecuzione più lunghi.

Nel caso della scena con grosse superfici specchiate, quella a destra, i percorsi di luce disperdono poca energia nel loro tragitto, pertanto con poche iterazioni risultano macchie di luce vistose sulle superfici diffusive. Con l'avanzare dell'esecuzione, la maggior parte del tempo sarà dedicata ai cammini casuali per il calcolo del flusso sulle superfici non diffusive.

I risultati di un ulteriore test qualitativo sono visibili in Fig.5.1.4 dove è rappresentata una scena con superfici di diverso materiale e con due sorgenti luminose. È possibile evidenziare i fenomeni ottici della penombra correttamente visualizzati, così come le doppie rifrazioni di luce nella sfera trasparente, sia originate dalle sorgenti stesse, sia riflesse dalla sfera metallica.

Nella Fig.5.1.5 sono rappresentate alcune sfere diffusive di colori differenti su di un pavimento specchiato in una stanza bianca. Al centro è anche posizionata una sfera trasparente. È possibile osservare come le varie tonalità di colore cambino in relazione alla posizione della sorgente luminosa e alla luce riflessa dal pavimento. In particolare è possibile notare sulle pareti bianche la penombra causata dalla luce riflessa dal pavimento che illumina le sfere lungo le pareti con una direzione proveniente dal basso. Inoltre è possibile vedere come le sfere sul fondo siano rifratte nella sfera centrale trasparente in ordine capovolto.

Infine nella Fig.5.1.6 è presentata l'immagine risultato di una scena test ($test3$) contenente un oggetto estremamente complesso, il cui modello è stato generato ricorsivamente ed è costituito da 780 sfere di diversa dimensione, colore e materiale. Si può notare come, anche in questo caso, gli oggetti e le singole componenti siano rappresentati molto accuratamente, rispettando anche le proprietà ottiche della penombra estremamente complessa.

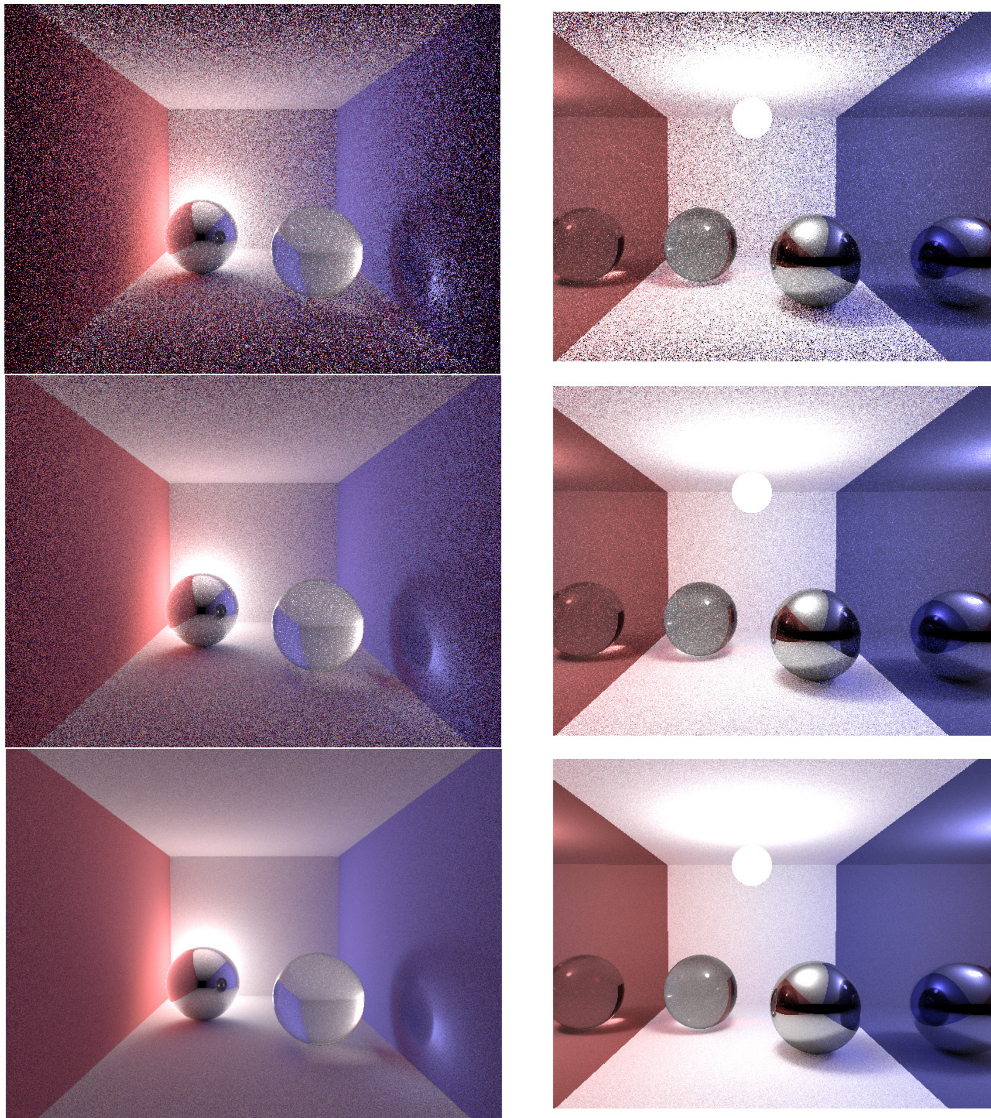

Figura 5.1.3: A sinistra: scena con sorgente luminosa non visibile direttamente e con luminosità diffusa dalla superficie speculare. A destra: scena con grandi superfici speculari. Per entrambi i test sono state utilizzate diverse tolleranze per le tre immagini.

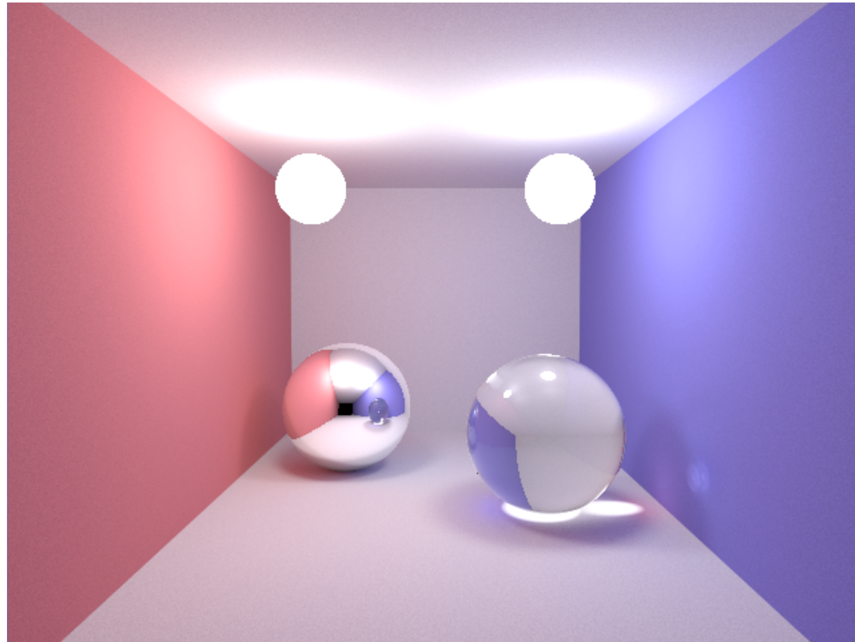

Figura 5.1.4: Scena con superfici diffusive, speculari, trasparenti e due sorgenti luminose. La tolleranza utilizzata è 0.0001.

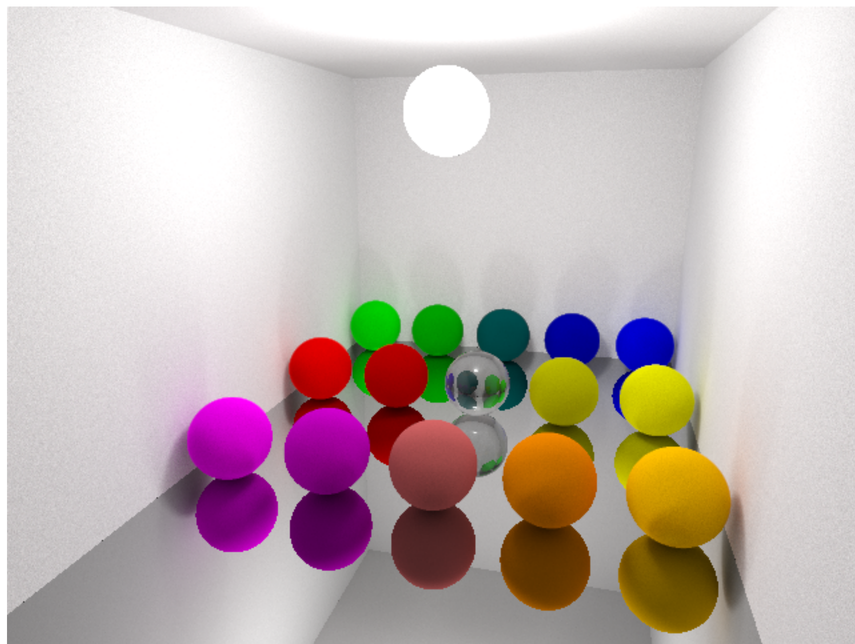

Figura 5.1.5: Scena con superfici diffusive, speculari, trasparenti. La tolleranza utilizzata è 0.0005.

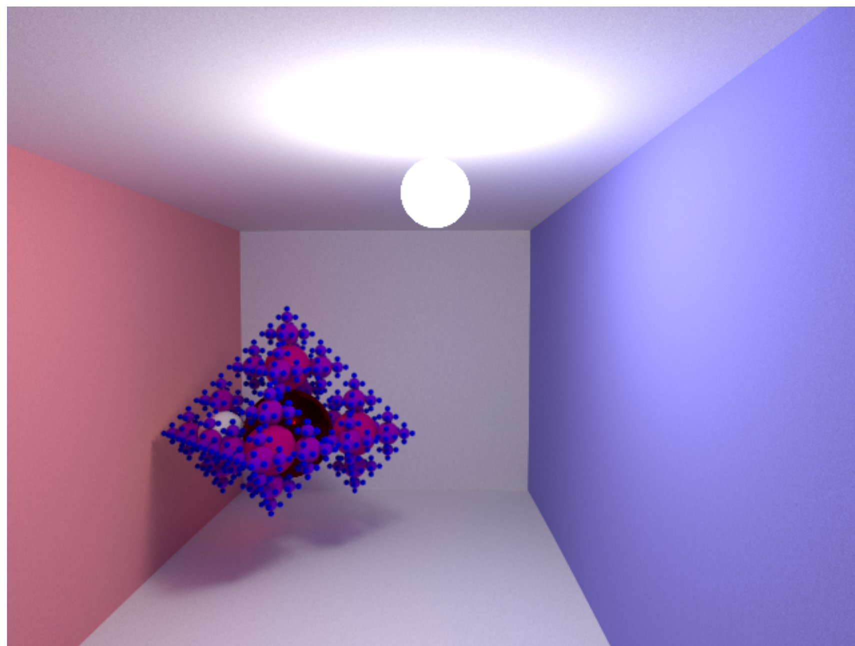

Figura 5.1.6: Scena con superfici diffusive, speculari, trasparenti. La tolleranza utilizzata è 0.0001. Il modello dell'oggetto rappresentato al centro della scena è stato generato ricorsivamente ed è costituito da 780 sfere di materiali differenti.

CPU	Intel Core 2 Quad: 4 core, 2.4 GHz, 64-bit
Memoria	8 GB DDR3-1066
Hard Disk	101,4 GB
GPU	NVIDIA Quadro 600:96 core, 1GB memoria locale, 1.2 GHz

Tabella 5.2.1: Caratteristiche tecniche della workstation 2

Alg.4.6	Codice CUDA
`Kernel_GenRand`	RandomGPU
`Kernel_emit`	get_ray
`Kernel_PercorsoLuce`	RadianceLightTracing_dev
`Kernel_CamminoCasuale`	RadiancePathTracing_dev

Tabella 5.2.2: Corrispondenza dei nomi dei kernel tra Alg.4.6 ed il codice CUDA

5.2 Valutazioni delle prestazioni su GPU del software parallelo

Il software parallelo è stato sviluppando utilizzando CUDA 4.0 su Ubuntu Linux 10.10.

I test sono stati eseguiti sulla macchina di riferimento (cfr. Sez.4.1) e su una macchina dotata di GPU di nuova generazione, con architettura Fermi (Tab.5.2.1).

In questo paragrafo faremo riferimento ai kernel descritti nella sez.4.5 con i nomi utilizzati nel codice sviluppato. La corrispondenza è chiarita in Tab.5.2.2.

Come si può vedere dalle Fig.5.2.1 e Fig.5.2.2, lo stesso software viene fatto eseguire sulle due macchine, con le stesse configurazioni di blocchi di thread, riportando carichi molto diversi (in termini percentuali) per i kernel. Ciò è dovuto alla diversa organizzazione dei livelli di memoria e delle unità di controllo di ogni SM, nonché al diverso numero di SP.

Considerando il $test2$ (Fig.5.1.2), mantenute le stesse configurazioni di blocchi di thread, i tempi di esecuzione sono migliori sulla workstation con Quadro 600 nonostante il minor numero di SP (Tempi riportati in Fig.5.2.3).

Lo stesso comportamento si può osservare nella Fig.5.2.4, dove sono rappresentati i tempi di esecuzione per il $test1$ con sole superfici diffusive (Fig.5.1.1), anche se con un divario tra i tempi sulle due workstation leggermente maggiore dell'esempio precedente.

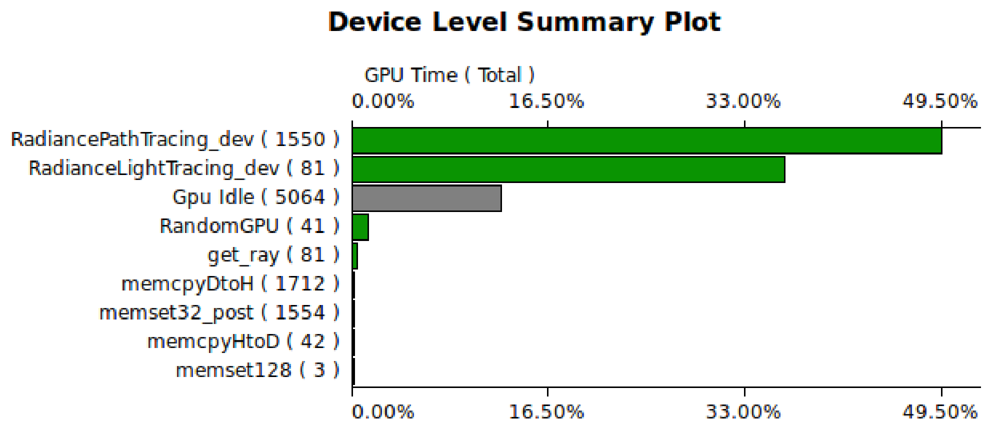

Figura 5.2.1: Percentuali di utilizzo della GPU Quadro 600, per kernel

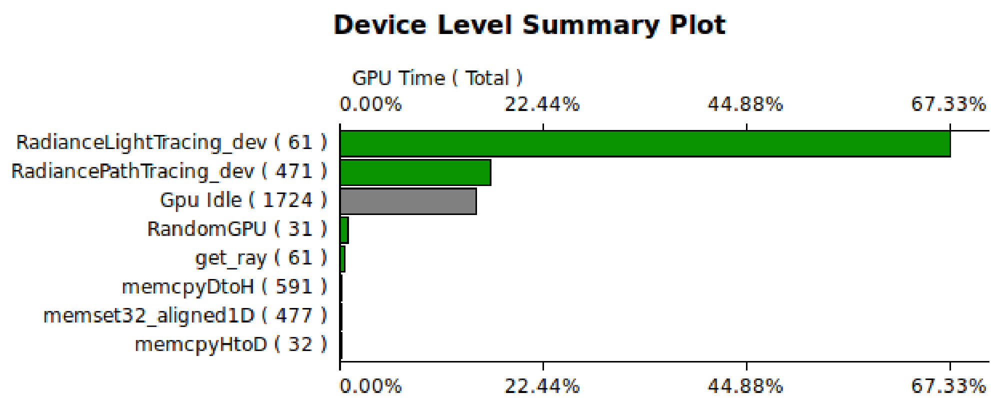

Figura 5.2.2: Percentuali di utilizzo della GPU Quadro FX3800, per kernel

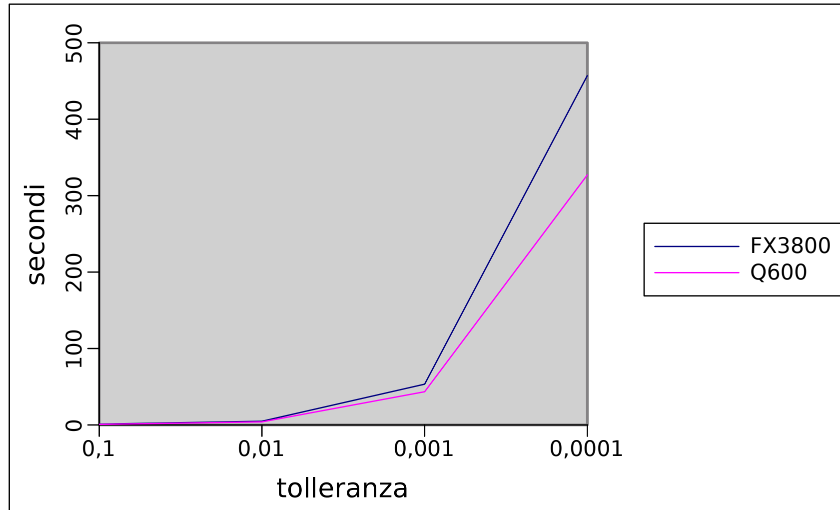

Figura 5.2.3: Tempi di esecuzione dell'esempio in Fig.5.1.2, su Quadro FX3800 e Quadro 600.

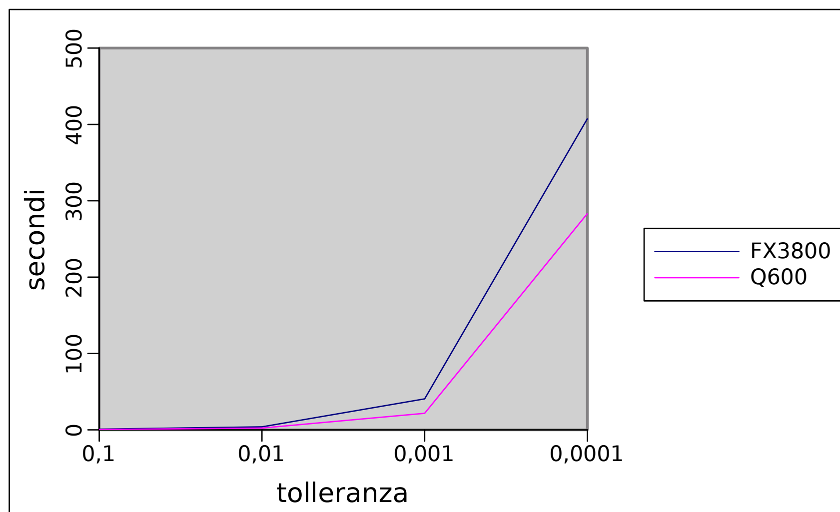

Figura 5.2.4: Tempi di esecuzione dell'esempio in Fig.5.1.1, su Quadro FX3800 e Quadro 600.

(dimGrid,dimBlock)	`get_ray`	`RadianceLightTracing_dev`
$(3200, 64)$	0.333333	0.333333
$(1600, 128)$	0.666667	0.583333
$(800, 256)$	0.833333	0.5
$(400, 512)$	0.666667	0.333333

Tabella 5.2.3: Occupazione degli SM per i kernel `get_ray` e `RadianceLightTracing_dev` in corrispondenza di varie configurazioni dei thread (valori tra 0 e 1)

Possiamo quindi affermare che il software sviluppato ha tempi di esecuzione leggermente più brevi sulla nuova workstation (Tab.5.2.1).

Procederemo quindi allo studio delle configurazioni dei blocchi di thread considerando questa macchina, che prospetta risultati in tempi più brevi.

Poiché la configurazione ideale della griglia di blocchi di thread non è conosciuta a priori, è necessario eseguire alcuni test per cercare di ottenere le performance migliori.

5.2.1 Occupazione

5.2.1.1 Percorsi di luce

Il software sviluppato ha la caratteristica di produrre fotogrammi successivi rappresentanti i risultati intermedi, fino all'immagine finale. All'avanzare del tempo di esecuzione le immagini sono sempre più accurate nella rappresentazione della scena. Quindi, nel decidere quanti thread utilizzare per la generazione dei percorsi di luce, si è cercato di mantenere un *framerate* intorno ai $4 frame/sec$.

Questa velocità di visualizzazione dei fotogrammi corrisponde, sulle workstation utilizzate per i test, all'esecuzione di circa 205000 percorsi di luce in 1/4 secondi.

Per questo motivo i kernel `RadianceLightTracing_dev` e `get_ray` utilizzano configurazioni con questo numero totale di thread.

Nella Tab.5.2.3 sono riportate le misurazioni dell'occupazione per quattro configurazioni differenti. In tutti i casi è stata utilizzata una griglia monodimensionale, contenente blocchi monodimensionali. Questo approccio è preferibile vista la totale indipendenza dei thread e la non coalescenza degli accessi alla memoria globale (cfr. sez.4.3-Metodo delle direzioni). Pertanto

ci si riferisce alle diverse configurazioni con la coppia ($dimGrid, dimBlock$) dove *dimGrid* è il numero di blocchi di thread utilizzati, e *dimBlock* è il numero di thread per blocco.

Si può evidenziare quindi che per il kernel `get_ray` la configurazione ottimale è $(800, 256)$, mentre per `RadianceLightTracing_dev` è preferibile $(1600, 128)$. Ciò è dovuto al maggior numero di registri utilizzato nel secondo kernel, che di conseguenza limita il numero ottimale di thread di un blocco. Con meno thread le risorse degli SM sarebbero parzialmente inutilizzate, mentre con più thread sarebbero saturati i registri.

5.2.1.2 Calcolo dei numeri random

L'algoritmo utilizzato per la generazione del vettore di numeri casuali è il Mersenne Twister [Matsumoto98]. La migliore configurazione possibile, che garantisce una totale occupazione degli SM, è data dalla coppia $(16, 256)$. Ad ogni esecuzione vengono generati 7.5 milioni di numeri random, che è una quantità sufficiente per l'esecuzione del kernel dei percorsi di luce. Grazie, però, all'utilizzo di `ind_rand` (cfr. Alg.4.6), è possibile assegnare al kernel dei cammini casuali una sequenza di numeri diversa a partire dagli stessi numeri random generati, garantendo così una bassa distorsione statistica nei risultati.

5.2.1.3 Cammini casuali

In questo caso, poiché l'accesso alle locazioni di memoria globale dei pixel potrebbe avvenire con coalescenza, si è preferito utilizzare una griglia bidimensionale per la configurazione dei blocchi di thread. In tutti i test in cui i blocchi avevano dimensioni che erano potenze di 2, l'occupazione è risultata sempre pari a 0.333333. In caso contrario l'occupazione era molto minore.

Anche `RadiancePathTracing_dev`, come nel caso del kernel dei percorsi di luce, ha una grossa limitazione in termini di occupazione a causa dell'elevato numero di registri.

È importante notare, però, che secondo alcuni studi [Volkov10-1, Volkov10-2], si possono ottenere ottime performance sfruttando maggiormente i registri, senza saturare l'occupazione degli SM.

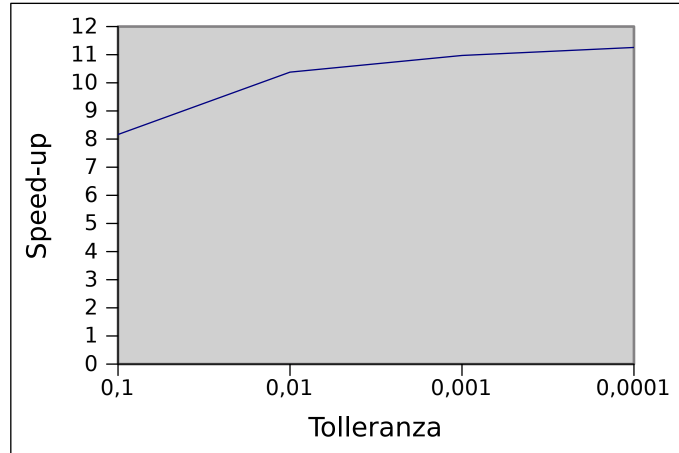

Figura 5.2.5: Speed-up relativo al $test1$ (Fig.5.1.1), per diverse tolleranze

5.2.2 Speed-up ed Efficienza

Per il Calcolo dello speed-up e dell'efficienza sono state utilizzate due versioni dello stesso software: quella per GPU, e quella per CPU multi-core.

Considerata, quindi l'eq.4.2.1, il termine $T\left(A_{seq}\right)$ in realtà va sostituito con $T\left(A_{CPU}\right)$, dove la nostra CPU è descritta in Tab.5.2.1, ed il software utilizzato sfrutta tutti i core a disposizione. La strategia adottata per la parallelizzazione su CPU multi-core è la stessa di quella descritta in sez.4.5. L'implementazione è stata realizzata con l'uso di OpenMP [Chandra00], che è una estensione del linguaggio C inclusa nel compilatore utilizzato (gcc versione 4.4.5).

Utilizzando le configurazioni che assicurano il massimo dell'occupazione degli SM, così come descritto in sez.5.2.1, si ha che per il test della scena con sole superfici diffusive (cfr. Fig.5.1.1) lo speed-up è:

$$S_{SP}(test1) = 11$$

utilizzando i 96 SP a disposizione sulla GPU ed una tolleranza di 0.0001.

Nella Fig.5.2.5 si può notare che lo speed-up relativo a diverse tolleranze nel criterio di arresto (eq.3.4.9) è:

$$S_{SP}(test1) \approx 10$$

L'inclinazione della curva è dovuta alle operazioni di inizializzazione delle strutture dati e di OpenGL, che essendo eseguite in sequenziale una sola

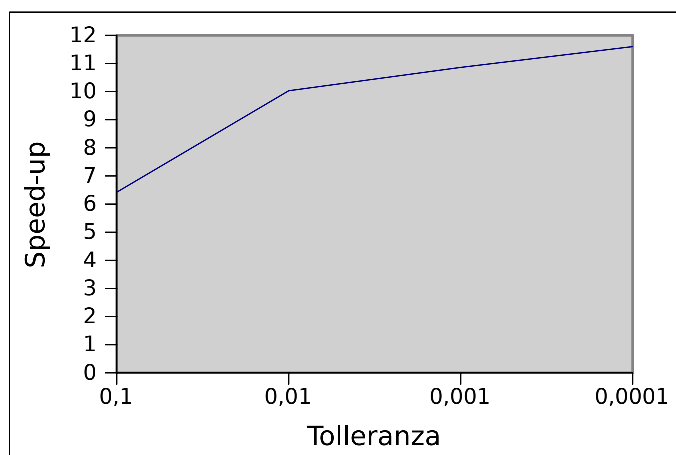

Figura 5.2.6: Speed-up relativo al *test*2 (Fig.5.1.2), per diverse tolleranze

volta allo start-up, diminuiscono di peso in valore percentuale all'aumentare del numero di iterazioni, e quindi dei tempi di esecuzione.

Un risultato analogo si ha per il test della scena con superfici sia diffusive sia speculari/trasparenti (cfr. Fig.5.1.2). Nella Fig.5.2.6 si può osservare che lo speed-up è:

$$S_{SP}(test2) \approx 10$$

Per quanto riguarda la misura dell'efficienza del software parallelo su GPU, in Fig.5.2.7 e Fig.5.2.8 si possono osservare i valori relativi ai due test di riferimento. In entrambi i casi si può vedere che l'efficienza, anche dopo numerose iterazioni, è:

$$E_{SP} \approx 0,1$$

ovvero si aggira intorno ad 1/10 dell'efficienza ideale, che è rappresentata dal valore 1.

Bisogna però ricordare che l'idea alla base di questo lavoro è quella di sfruttare risorse di calcolo progettate per l'illuminazione locale, solitamente a disposizione sulle workstation grafiche, al fine di ottenere un risultato accurato per il rendering con illuminazione globale in tempi brevi.

Nell'uso tipico delle risorse di calcolo sulle workstation grafiche, le GPU vengono utilizzate unicamente per la modellazione interattiva con illuminazione locale. Mentre per il rendering con illuminazione globale vengono impiegati software che sfruttano solo le CPU multicore.

Per questo lavoro, quindi, lo speed-up e l'efficienza misurano lo sfruttamento di una risorsa di calcolo che in genere non è utilizzata per risolvere questo

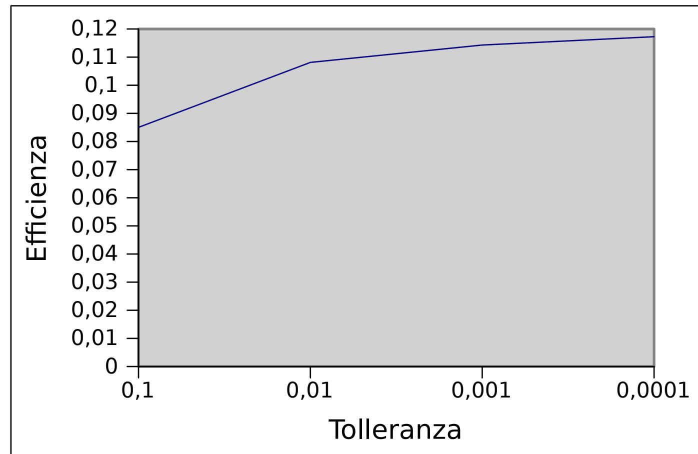

Figura 5.2.7: Efficienza relativa alla scena in Fig.5.1.1, per diverse tolleranze

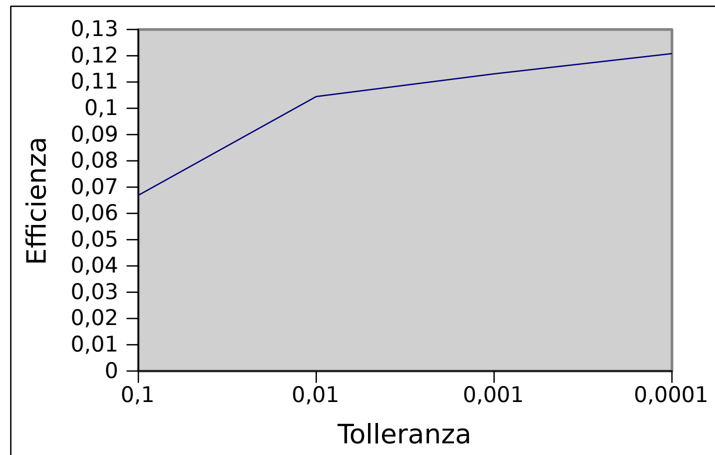

Figura 5.2.8: Efficienza relativa alla scena in Fig.5.1.2, per diverse tolleranze

problema, anche se disponibile per il calcolo. Pertanto non ha senso pensare di rapportare le prestazioni di questo software per GPU con quelle di un ipotetico software per calcolatori a memoria distribuita, che rimane oggetto di un altro filone di ricerca.

Capitolo 6

Sommario e Conclusioni

6.1 Sommario

In questo lavoro è stato preso in esame il problema dell'illuminazione globale nell'ambito della grafica computazionale, con l'obiettivo di sviluppare un software che sfrutti la potenza di calcolo delle GPU.

Trattandosi della generazione di immagini sintetiche rappresentanti una scena artificiale, utilizzando il procedimento detto di rendering, sono stati descritti i passi necessari (Cap.1): dalla creazione di un modello descrittivo della scena, fino alla visualizzazione dell'immagine nei termini di una matrice di pixel.

Concentrando l'attenzione sulla fase di rendering, è stato descritto (Cap.2) il modello matematico sottostante i fenomeni di propagazione della luce in un ambiente. Tra le formulazioni presenti in letteratura, è stata preferita quella legata alla radianza luminosa, che appare essere la più versatile per descrivere una maggiore varietà di fenomeni ottici in un'ampia gamma di scene.

Al fine di individuare un opportuno metodo adatto all'implementazione efficiente su GPU, sono stati studiati (Cap.3) gli approcci più diffusi, evidenziando le relative approssimazioni, e le loro limitazioni in termini qualitativi dei risultati. È stato poi introdotto un nuovo metodo iterativo di cui si è provata la convergenza.

Passando poi allo studio dell'implementazione sulle GPU (Cap.4), è stata introdotta l'architettura parallela con le caratteristiche più rilevanti per la valutazione del software che si vuole sviluppare. Sono poi stati analizzati i vari metodi risolutivi alla luce di nuovi parametri di valutazione delle prestazioni

e delle aspettative qualitative. Quindi è stato implementato un algoritmo ibrido sia in versione sequenziale, sia in versione parallela multithreading.

Infine sono stati analizzati i risultati dell'implementazione (Cap.5) dal punto di vista della qualità delle immagini ottenute, e delle prestazioni raggiunte con l'utilizzo delle GPU.

6.2 Conclusioni

I risultati visivi dell'algoritmo ibrido sono soddisfacenti sia nella versione sequenziale, sia nella versione multithreading con e senza GPU, già con tolleranze molto maggiori della massima precisione relativa ottenibile dalle unità di calcolo a singola precisione.

Per di più i metodi implementati tendono a convergere rapidamente nelle scene rappresentanti ambienti chiusi, che sono quelle in cui è più interessante visualizzare i fenomeni diffusivi.

È stato mostrato che l'algoritmo è molto robusto, in quanto è possibile percepire le qualità dell'immagine risultato già dopo poche iterazioni, aumentando l'accuratezza con il numero di iterazioni eseguite.

Per quanto riguarda i tempi di esecuzione del software multithreading su GPU, il guadagno rispetto all'esecuzione del software multithreading su CPU è evidente, arrivando a diventare circa dieci volte più veloce sull'hardware dei casi test presi in esame.

Rispetto ai parametri classici di valutazione dei software paralleli, il software sviluppato ha un ampio margine di ottimizzazione in termini di speed-up ed efficienza relativi al numero di core utilizzati.

6.2.1 Direzioni per ricerche future

L'immagine del *test*3 (cfr.Fig.5.1.6) è stata realizzata, utilizzando la GPU di riferimento della Tab.5.2.1, in un tempo di esecuzione di 2 ore e 26 minuti, mentre sfruttando il solo multithreading della CPU il tempo di esecuzione è stato di circa 21 ore.

Risulta quindi chiaro che l'esecuzione su GPU del software sviluppato permette la realizzazione di immagini sintetiche foto-realistiche in tempi tali da poter influire significativamente sui processi produttivi di design, come ad esempio nel caso di prove di giuria per modelli automobilistici, o per la presentazione di diverse opzioni in architettura di interni.

Ma avendo come altro obiettivo anche l'interattività, ad esempio per tutti i casi in cui la visualizzazione è attualmente affidata a metodi di illuminazione locale, è necessario approfondire alcune problematiche emerse dal presente studio.

La riorganizzazione delle operazioni nei kernel può condurre ad un miglior utilizzo delle risorse computazionali a disposizione, con l'obiettivo di portare il parametro di efficienza verso valori ideali, quando si confrontano le implementazioni per CPU e GPU.

Con la produzione di nuove architetture GPU e la conseguente riorganizzazione dei core, i parametri caratterizzanti l'overhead totale nell'esecuzione di un software multithreading sono in continua evoluzione. È necessario ed auspicabile lo studio sistematico delle caratteristiche responsabili del degrado delle prestazioni ideali, e la formulazione di parametri chiari per la valutazione delle prestazioni dei software che utilizzano le GPU come risorse di calcolo addizionali.

Infine è necessario testare l'efficienza del metodo ibrido e la qualità dei risultati nel caso di BRDF più complesse, che coinvolgano fenomeni misti di diffusione, specularità e/o trasparenza.

Bibliografia

[Bala11] K. Bala, P. Dutre, P. Bekaer, *Advanced Global Illumination*, Second Edition, A K Peters/CRC Press, 2011

[Carr03] N. A. Carr, J. D. Hall, J. C. Hart, *GPU Algorithms for Radiosity and Subsurface Scattering*, Proceedings of the ACM SIGGRAPH-EUROGRAPHICS conference on Graphics hardware, 2003

[Chalmers98] A. Chalmers, E. Reinhard, *Parallel and Distributed Photo-Realistic Rendering (course notes)*, ACM SIGGRAPH, 1998

[Chandra00] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald, *Parallel Programming in OpenMP*, Morgan Kaufmann, 2000

[Cohen85] M. F. Cohen, D. P. Greenberg, *The hemi-cube: a radiosity solution for complex environments*, SIGGRAPH Comput. Graph. 19, 3, 1985

[Cohen88] M. F. Cohen, S. E. Chen, J. R. Wallace, D. P. Greenberg, *A progressive refinement approach to fast radiosity image generation*, SIGGRAPH '88 Comput. Graph., 1988

[Cohen93] M. Cohen , J. Wallace, *Radiosity and Realistic Image Synthesis*, Academic Press Prof., 1993

[Cook84] R.L. Cook, T. Porter, L. Carpenter, *Distributed ray tracing*, Computer Graphics (Proceedings of SIGGRAPH'84), 1984

[Cook86] R. L. Cook, *Stochastic sampling in computer graphics*, ACM Trans. on Computer Graphics, 1986

[Coombe04] G. Coombe, M. Harris, A. Lastra, *Radiosity on Graphics Hardware*, Graphics Interface, 2004

[D'Amore11] L. D'Amore, L. Marcellino, V. Mele, D. Romano, *Deconvolution of 3D Fluorescence Microscopy Images using Graphics Processing Units*, Workshop on Models, Algorithms and Methodologies for Hierarchical Parallelism in new HPC Systems, PPAM 2011

[Demmel08] V. Volkov, J.W. Demmel, *Benchmarking GPUs to tune dense linear algebra*, ACM/IEEE Conference on Supercomputing, 2008.

[Doucet10] A. Doucet, A. M. Johansen, V. B. Tadić, *On solving integral equations using Markov chain Monte Carlo methods*, Applied Mathematics and Computation, 2010

[Flynn66] M.J. Flynn, *Very High-Speed Computing Systems*, Proceedings of the IEEE, vol. 54, p.p. 1901-1909, December 1966

[Gautron05] P. Gautron, J. Křivánek, K. Bouatouch, S. Pattanaik, *Radiance cache splatting: a GPU-friendly global illumination algorithm*, ACM SIGGRAPH 2005 Sketches (SIGGRAPH '05), 2005

[Glassner95] A. S. Glassner, *Principles of digital image synthesis*, Coyote Wind LLC, 1995

[Goral84] C.M. Goral, K.E. Torrance, D.P. Greenberg, B. Battaile, *Modeling the interaction of light between diffuse surfaces*, SIGGRAPH'84, 1984

[Gortler93] S. J. Gortler, P. Schroeder, M. F. Cohen, P. Hanrahan, *Wavelet radiosity*, In Proceedings of the 20th annual conference on Computer graphics and interactive techniques (SIGGRAPH '93), ACM, 1993

[Greger98] G. Greger, P. Shirley, P.M. Hubbard, D.P. Greenberg, *The Irradiance Volume*, IEEE Computer Graphics and Applications, 1998

[Heckbert90] P. S. Heckbert, A*daptive radiosity textures for bi-directional ray tracing*, Proc. SIGGRAPH '90, 1990

[Kajiya86] J. T. Kajiya, *The rendering equation*, Proceedings of the 13th annual conference on Computer graphics and interactive techniques (SIGGRAPH '86), 1986

[Keller97] A. Keller, *Instant radiosity*, SIGGRAPH '97 Proceedings, 1997

[Kolmogorov80] A. N. Kolmogorov, S. V. Fomin, *Elementi di teoria delle funzioni e di analisi funzionale*, Ed. Mir, 1980

[Krommer98] A. R. Krommer, C. W. Ueberhuber, *Computational Integration*, SIAM, 1998

[Krüger05] J. Krüger, R. Westermann, *A GPU Framework for Solving Systems of Linear Equations*, GPU gems 2: programming techniques for high-performance graphics and general-purpose computation, Randima Fernando, series editor, 2005

[Kurç97] T.M.Kurç, C. Ayakant, B. Özgüç, *A parallel scaled conjugate-gradient algorithm for the solution phase of gathering radiosity on hypercubes*, The Visual Computer, 1997, 13

[Laine07] S. Laine, H. Saransaari, J. Kontkanen, J. Lehtinen, T. Aila, *Incremental Instant Radiosity for Real-Time Indirect Illumination*, Eurographics Symposium on Rendering, 2007

[Leblond02] M. Leblond, *H-self-adjoint matrices. Application to radiosity*, Numerical Linear Algebra with Applications, 2002

[Lee85] M. E. Lee, R. A. Redner, S. P. Uselton, *Statistically optimized sampling for distributed ray tracing*, SIGGRAPH Comput. Graph. 19, 3, 1985

[Matsumoto98] M. Matsumoto, T. Nishimura, *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Trans. on Modeling and Computer Simulation Vol. 8, 1998

[Murli06] A. Murli, *Lezioni di Calcolo Parallelo*, Liguori Editore, 2006.

[Murli07] A. Murli, *Matematica Numerica: metodi, algoritmi e software - Parte Prima*, Liguori Editore, 2007

[Nichols09] G. Nichols, J. Shopf, C. Wyman, *Hierarchical Image-Space Radiosity for Interactive Global Illumination*, Computer Graphics Forum, 2009

[Nijasure05] M. Nijasure, S. N. Pattanaik, V. Goel, *Real-time Global Illumination on GPU*, Journal of Graphics Tools, 2005

[Phong75] B. T. Phong, *Illumination for computer generated pictures*, Commun. ACM 18, 6 , 1975

[Sanjurjo09] J. R. Sanjurjo, M. Amor, M. Bóo, R. Doallo, J. Casares, *Optimizing Monte Carlo radiosity on graphics hardware*, The Journal of Supercomputing, 2009

[Shirley91] P. Shirley, C. Wang, *Direct lighting by monte carlo integration*, Proceedings of the Second Eurographics Workshop on Rendering, 1991

[Szirmay-Kalos99] L. Szirmay-Kalos, *Monte-Carlo Methods in Global Illumination*, Institute of Computer Graphics, Vienna University of Technology, 1999

[Szirmay-Kalos06] L. Szirmay-Kalos, L. Szecsi, M. Sbert, *GPUGI: Global Illumination Effects on the GPU*, Technology, 2006

[Tomov10] S. Tomov, R. Nath, H. Ltaief, J. Dongarra, *Dense linear algebra solvers for multicore with GPU accelerators*, Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, 2010

[Volkov10-1] V. Volkov, *Better performance at lower occupancy*, GPU Technology Conference, 2010

[Volkov10-2] V. Volkov, *Use registers and multiple outputs per thread on GPU*, International Workshop on Parallel Matrix Algorithms and Applications, 2010.

[Wallace87] J. R. Wallace, M. F. Cohen, D. P. Greenberg, *A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods*, Proceedings of the 14th annual conference on Computer graphics and interactive techniques (SIGGRAPH '87), 1987

[Wallace89] J. R. Wallace, K. A. Elmquist, E. A. Haines, *A Ray tracing algorithm for progressive radiosity*, Proceedings of the 16th annual conference on Computer graphics and interactive techniques , 1989

[Ware83] W.Ware, *The ultimate computer*, IEEE Spectrum, 1983

[Whitted80] T. Whitted, *An improved illumination model for shaded display*, CACM, 1980

[Zimmerman95] K. Zimmerman, P. Shirley, *A two-pass realistic image synthesis method for complex scenes*, Rendering Techniques '95, Springer-Verlag, 1995

Elenco delle figure

1.2.1 Pipeline per il rendering con modello di illuminazione locale . 10
1.2.2 Schema dell'architettura della scheda GeForce 6 11
1.2.3 Schema dell'architettura della scheda GeForce 8 11
1.3.1 Flusso di dati nel rendering con illuminazione globale 13

2.2.1 Sistema di coordinate sferiche (θ, φ) 16
2.2.2 Angolo solido da superficie infinitesima dA_y con proiezione su sfera unitaria 16
2.2.3 Funzione di ray casting in una scena con tre superfici 17
2.3.1 Area della superficie infinitesima proiettata lungo la direzione ω 19

3.1.1 Riflessione speculare 29
3.1.2 Riflessione diffusa 29
3.1.3 Esempi di percorsi di luce 30
3.2.1 Ray Tracing di Whitted 32
3.3.1 Albero per la valutazione di $a(x_0)$ 38
3.3.2 Esempio di cammino casuale 38
3.4.1 Percorso di luce con punto iniziale nella sorgente luminosa . . 46
3.4.2 Calcolo del flusso in corrispondenza di ogni x_i 48

4.1.1 Schema dello Streaming Processor (SP) 52
4.1.2 Schema dello Streaming Multi Processor (SM) 52
4.1.3 Schema del Texture/Processor Cluster (TPC) 54

4.1.4 Schema dello Streaming Processor Array 55
4.1.5 Data Parallelism nel prodotto tra matrici 56
4.1.6 Esecuzione di un programma su GPU con CUDA 57
4.2.1 Esecuzione di un flusso di istruzioni su un SP. Prima riga: tempo di decodifica delle istruzioni. Seconda riga: tempo di esecuzione di un thread. Terza riga: tempo di esecuzione di quattro thread. 58
4.3.1 Scena con rendering di tipo raytracing 62
4.3.2 Scena con Rendering di tipo Radiosity 63
4.3.3 Scena con Rendering di tipo Stocastico 65

5.1.1 Scena con tutte superfici diffusive. In senso orario partendo dall'alto a sinistra: $tol = 0.1$, $tol = 0.01$, $tol = 0.001$, $tol = 0.0001$. 78
5.1.2 Scena con superfici diffusive, speculari e trasparenti. In senso orario partendo dall'alto a sinistra: $tol = 0.1$, $tol = 0.01$, $tol = 0.001$, $tol = 0.0001$. 79
5.1.3 A sinistra: scena con sorgente luminosa non visibile direttamente e con luminosità diffusa dalla superficie speculare. A destra: scena con grandi superfici speculari. Per entrambi i test sono state utilizzate diverse tolleranze per le tre immagini. 81
5.1.4 Scena con superfici diffusive, speculari, trasparenti e due sorgenti luminose. La tolleranza utilizzata è 0.0001. 82
5.1.5 Scena con superfici diffusive, speculari, trasparenti. La tolleranza utilizzata è 0.0005. 82
5.1.6 Scena con superfici diffusive, speculari, trasparenti. La tolleranza utilizzata è 0.0001. Il modello dell'oggetto rappresentato al centro della scena è stato generato ricorsivamente ed è costituito da 780 sfere di materiali differenti. 83
5.2.1 Percentuali di utilizzo della GPU Quadro 600, per kernel . . . 85
5.2.2 Percentuali di utilizzo della GPU Quadro FX3800, per kernel . 85
5.2.3 Tempi di esecuzione dell'esempio in Fig.5.1.2, su Quadro FX3800 e Quadro 600. 86
5.2.4 Tempi di esecuzione dell'esempio in Fig.5.1.1, su Quadro FX3800 e Quadro 600. 86

5.2.5 Speed-up relativo al *test*1 (Fig.5.1.1), per diverse tolleranze . . 89
5.2.6 Speed-up relativo al *test*2 (Fig.5.1.2), per diverse tolleranze . . 90
5.2.7 Efficienza relativa alla scena in Fig.5.1.1, per diverse tolleranze 91
5.2.8 Efficienza relativa alla scena in Fig.5.1.2, per diverse tolleranze 91