# UNIVERSITÀ DEGLI STUDI DI PARMA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

---

## VISUALIZZAZIONE IN GRAFICA 3D DI RIFLESSIONI DINAMICHE MEDIANTE ENVIRONMENT MAPPING

Relatore

Chiar.mo Prof. STEFANO CASELLI

Correlatore

Ing. JACOPO ALEOTTI

Laureando

LUCA METTI

ANNO ACCADEMICO 2008-2009

*Alla mia famiglia.*

# Indice

INTRODUZIONE . . . . . 3

Obiettivi della Tesi . . . . . 4

Organizzazione della Tesi . . . . . 5

CAPITOLO 1: TEXTURE MAPPING . . . . . 6

1.1 Texture su superfici piane . . . . . 6

1.2 Mipmapping . . . . . 8

1.3 Modellazione di riflessioni speculari . . . . . 10

1.3.1 Generazione delle coordinate della texture . . . . . 11

1.3.2 Sphere Mapping . . . . . 13

1.3.3 Cube Mapping . . . . . 14

1.4 Riflessioni speculari dinamiche . . . . . 18

CAPITOLO 2: GENERAZIONE DI TEXTURE DINAMICHE . . . . . 21

2.1 Gestione del Framebuffer . . . . . 22

2.2 Tecniche di rendering su texture . . . . . 24

2.2.1 Architettura del Framebuffer Object . . . . . 26

2.2.2 Configurazione di un Framebuffer Object . . . . . 27

2.3 Realizzazione di un Environment Mapping dinamico . . . . . 29

CAPITOLO 3: PROGETTAZIONE DEL SISTEMA . . . . . 31

3.1 Stato dell'arte . . . . . 31

3.2 La libreria OpenGL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.3 Descrizione dell'applicazione realizzata . . . . . . . . . . . . . . . . . . . . . . . 35
3.3.1 Inizializzazione dell'ambiente grafico . . . . . . . . . . . . . . . . . 35
3.3.2 Inizializzazione della cubemap . . . . . . . . . . . . . . . . . . . . . . . 36
3.3.3 Inizializzazione del Framebuffer Object . . . . . . . . . . . . . . . . 38
3.3.4 Interfaccia grafica . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.3.5 Generazione della cubemap dinamica . . . . . . . . . . . . . . . . . . 43
3.3.6 Rendering della scena . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

CONCLUSIONI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

Bibliografia . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

Ringraziamenti . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

# Introduzione

Nella disciplina dell'informatica uno dei campi che ha avuto una notevole evoluzione negli ultimi anni è la computer grafica 3D. Questo termine sta a indicare la scienza che si occupa di studiare i metodi di proiezione della rappresentazione matematica di oggetti tridimensionali su un'immagine bidimensionale, attraverso l'uso di tecniche come la prospettiva e l'ombreggiatura che simulano la percezione di questi oggetti da parte dell'occhio umano.

Ogni sistema 3D deve fornire due elementi: un metodo di descrizione del sistema 3D stesso ("scena"), composto da rappresentazioni matematiche di oggetti tridimensionali, detti "modelli", e un meccanismo di produzione di un'immagine 2D dalla scena, detto "renderer".

Visto l'impiego sempre più massiccio della computer grafica 3D in una moltitudine di ambiti professionali e di consumo, come i videogiochi, la progettazione grafica (CAD) o l'industria cinematografica, si è resa necessaria la realizzazione di tecniche sempre più avanzate volte a incrementare il realismo della scena visualizzata.

Un metodo per migliorare il realismo di una scena è quello di introdurre l'illuminazione, intesa come modellazione degli effetti che la luce reale produce sugli oggetti tridimensionali. Tra questi effetti è particolarmente rilevante la riflessione speculare, cioè quel fenomeno fisico per cui i raggi luminosi che colpiscono una superficie levigata (es. uno specchio) con un certo angolo di incidenza vengono riflessi in una direzione avente un angolo di riflessione di pari ampiezza.

# Obiettivi della tesi

Nell'ambito della robotica la visione artificiale riveste un ruolo essenziale, dato che tra i sensori che vengono impiegati quelli che forniscono la maggior parte delle informazioni sono di tipo visivo. A questo proposito una tipologia di sensorialità di grande interesse è la percezione omnidirezionale, cioè in grado di fornire una visione dell'ambiente a 360°. Questi sensori trovano un impiego ideale nei robot mobili per le problematiche relative alla localizzazione e alla navigazione nell'ambiente circostante.

Nel corso degli anni i sensori omnidirezionali sono stati costruiti in diversi modi, con un numero variabile di specchi e telecamere: in [1] si impiegano delle telecamere puntate su diversi specchi disposti in modo da formare una piramide; in [2] è presente un numero limitato di telecamere mobili che ruotano intorno ad alcuni specchi; in [3] viene impiegata una lente fisheye con un elevato angolo di campo; infine sono di particolare interesse i sensori omnidirezionali catadiottrici costruiti con un'unica telecamera fissa puntata verso uno specchio di forma parabolica o iperbolica [4, 5].

Considerata l'importanza di questi sensori, è nata la richiesta di simulatori in grado di riprodurre il comportamento dei sensori omnidirezionali con lo scopo di ridurre l'impiego di hardware per le fasi di testing. In questo ambito trova applicazione la computer grafica 3D, con particolare riferimento agli effetti di riflessione speculare per la simulazione degli specchi.

L'obiettivo che si pone questa tesi è di realizzare un'applicazione grafica, in linguaggio C, che simuli il comportamento di un sensore (telecamera) omnidirezionale mobile composta da uno specchio sferico all'interno di un ambiente tridimensionale virtuale. Mediante l'impiego delle *texture* e dell'*environment mapping*, una funzionalità avanzata della libreria OpenGL, l'applicazione può simulare la riflessione speculare che produrrebbe la luce reale su una superficie curva levigata. Inoltre la riflessione simulata è dinamica, cioè riproduce eventuali cambiamenti nell'ambiente circostante, come per esempio un

oggetto in movimento. Per ottenere questo effetto è stata impiegata una recente tecnica di rendering con la quale è possibile rigenerare le texture in modo da ottenere delle riflessioni dinamiche in tempo reale.

Le operazioni da compiere per realizzare un simulatore di questo tipo si possono riassumere molto schematicamente in:

1. Inizializzazione dell'ambiente grafico
2. Creazione e configurazione di una cubemap texture
3. Rendering su cubemap dell'ambiente circostante al sensore simulato
4. Orientamento della cubemap coerentemente con il punto di vista
5. Rendering a video della texture applicata alla sfera
6. Rendering a video dell'ambiente

# Organizzazione della tesi

Il primo capitolo della tesi è dedicato alle tecniche per l'applicazione di texture sulle superfici di oggetti tridimensionali, con particolare riferimento alle modalità con cui si può realizzare l'environment mapping necessario alla modellazione di riflessioni speculari.

Nel secondo capitolo viene illustrata una modalità alternativa al rendering tradizionale, detta rendering su texture, ottenuta grazie all'impiego del Framebuffer Object e fondamentale per la realizzazione di riflessioni dinamiche.

Nel terzo capitolo vengono presentati la libreria OpenGL, le principali caratteristiche di un'applicazione che ne fa uso e il sistema che è stato realizzato.

Infine vengono indicate le possibili applicazioni del sistema e le funzionalità che possono essere perfezionate o introdotte in futuro.

# Capitolo 1

# Texture Mapping

Il texture mapping è la tecnica fondamentale che viene usata per definire l'aspetto degli oggetti che si renderizzano. Uno degli utilizzi abituali che si fa del texturing è quello di aggiungere dettagli a una forma geometrica modificando il colore di ogni pixel della superficie. Solitamente le informazioni sul colore da applicare provengono da un'immagine digitale che viene caricata in memoria.

Questo capitolo tratta la modalità più diffusa di texture mapping e una modalità più avanzata, l'environment mapping, usata per simulare effetti grafici più complessi, come le riflessioni su una superficie curva.

## 1.1 Texture su superfici piane

Il primo passo per poter applicare una texture sulla superficie di una geometria è quello di avere in memoria la texture stessa. Quindi è necessario caricare in memoria le informazioni relative al colore di ogni pixel di un'immagine digitale e, successivamente, copiarle in una texture di pari dimensioni. Una texture è rappresentata da un array di texel, esattamente come un'immagine digitale è

rappresentata da un array di pixel. In OpenGL il comando per creare una texture è:

```
glTexImage2D(GLenum target, GLint level, GLint internalformat,
GLsizei width, GLsizei height, GLint border, GLenum format,
GLenum type, void *data);
```

Questa funzione fornisce a OpenGL tutto quello che deve sapere per poter interpretare correttamente le informazioni dell'immagine digitale, contenute in `data`, e crearne una texture. Nel caso più comune, il parametro `target` vale `GL_TEXTURE_2D` e la texture che viene generata è bidimensionale.

Una volta creata la texture va indicato come applicarla alla superficie della geometria. Per farlo non ci si riferisce ai texel della texture mediante indirizzi di memoria, ma più astrattamente mediante delle coordinate [6]. Tali coordinate sono espresse in valori decimali nell'intervallo [0.0, 1.0] e indicate con i simboli *s*, *t*, *r* (Figura 1.1) similarmente a come si fa con le coordinate spaziali *x*, *y*, *z*.

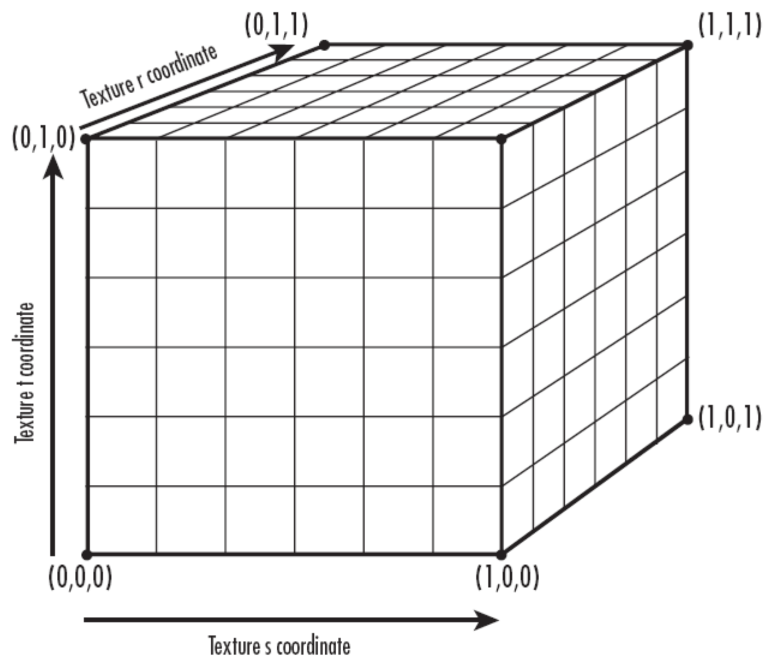


**Figura 1.1 I texel vengono indirizzati mediante le coordinate *s*, *t*, *r***

Nel caso di una texture bidimensionale, le coordinate *s*, *t* si specificano tramite il comando `glTexCoord2f(Glfloat s, GLfloat t)` e vanno fatte seguire dal comando che indica le coordinate di un vertice della geometria. La texture verrà automaticamente ridimensionata per poter essere applicata alla geometria. Nella figura 2.2 si può osservare un esempio di come una texture bidimensionale può essere applicata a due diverse figure geometriche.

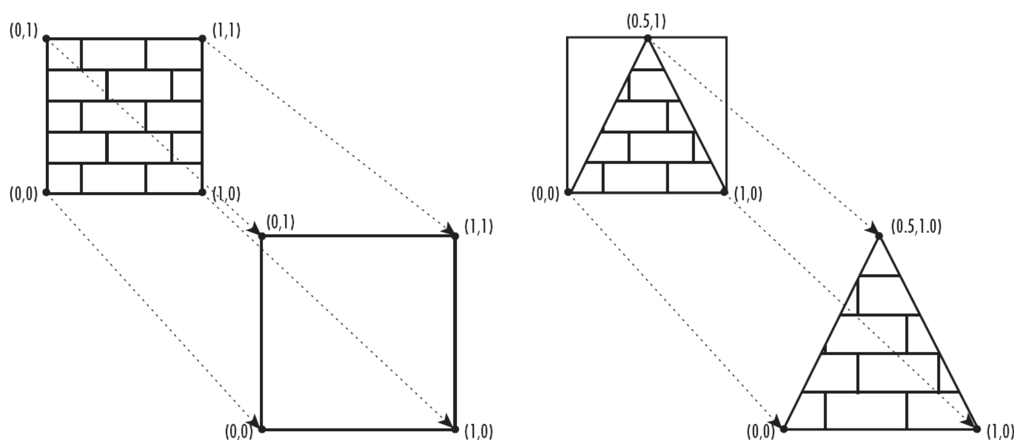


**Figura 1.2 Una texture bidimensionale applicata a un quadrato e a un triangolo**

# 1.2 Mipmapping

Il *mipmapping* è una tecnica che incrementa sia la qualità visiva di una scena che le prestazioni del rendering, risolvendo due problemi riscontrabili frequentemente quando un oggetto in lontananza viene disegnato molto piccolo rispetto alle dimensioni della texture applicata. Il primo problema è la comparsa di artefatti sulla superficie dell'oggetto che si presentano come pixel con una colorazione errata e sono maggiormente evidenti quando la vista o l'oggetto sono in movimento. Il secondo problema, che porta a un degrado delle prestazioni, è la necessità di applicare alla geometria un numero ridotto di texel che, richiedendo una quantità aggiuntiva di calcoli, incide sulla velocità del rendering [7].

Una soluzione a questi problemi potrebbe essere l'utilizzo di una texture più piccola ma, quando l'oggetto viene disegnato più grande poiché più vicino, la texture apparirebbe inevitabilmente più confusa e meno definita. La soluzione ideale è l'impiego del mipmapping con cui si generano diverse copie della texture, le mipmap, ognuna di dimensioni dimezzate rispetto alla precedente (Figura 1.3). Così facendo, OpenGL è in grado di scegliere tra le varie mipmap quella che di volta in volta meglio si adatta alle dimensioni della geometria. In questo modo si riescono a eliminare gli artefatti grafici quando l'oggetto è lontano e allo stesso tempo si mantiene la texture ad alta risoluzione da applicare quando l'oggetto è vicino.

**Figura 1.3 Una serie di mipmap**

Per avvalersi delle funzionalità del mipmapping è necessario, dopo aver caricato una texture, creare le varie mipmap. Per questo scopo è consigliabile l'utilizzo del comando `glGenerateMipmapEXT(GLenum target)` che genera le mipmap in modo rapido e automatico avvalendosi dell'accelerazione hardware della scheda video [8].

# 1.3 Modellazione di riflessioni speculari

Il realismo di una scena può essere migliorato modellando gli effetti della luce che risultano dalle riflessioni degli oggetti. OpenGL fornisce degli strumenti per modificare, ad esempio, l'effetto della luce ambientale sugli oggetti, ma è solo un'approssimazione di una riflessione reale. Un approccio più sofisticato si può ottenere sfruttando le funzionalità avanzate di texturing che OpenGL mette a disposizione. Con il termine *environment mapping* si intende una tecnica di texturing utilizzata per modellare le influenze che l'ambiente circostante produce sull'aspetto di un oggetto.

L'environment mapping, come il tradizionale surface texturing, modifica l'aspetto di un oggetto mediante l'applicazione di una texture map sulla sua superficie. Una environment texture map, però, prende in considerazione la vista circostante dell'ambiente in cui è l'oggetto. Se l'oggetto ha una superficie con elevata specularità, la texture map mostrerà l'ambiente riflesso dalla superficie. Dal momento che si sta simulando un effetto della luce, i texel vengono selezionati in funzione del *vettore normale* o del *vettore riflessione* per ogni punto della superficie [6]. Questi vettori vengono convertiti in coordinate texture per ogni vertice e poi interpolati ad ogni punto della superficie, come nel caso di una texture tradizionale. Utilizzando questi vettori come input alla funzione di generazione della texture, si rende possibile la simulazione del comportamento della luce nei casi di riflessione speculare e riflessione diffusa.

Le tecniche per implementare l'environment mapping, trattate nei prossimi paragrafi, sono lo sphere mapping e il cube mapping, entrambe caratterizzate da alcuni pregi e difetti.

### 1.3.1 Generazione delle coordinate della texture

In OpenGL la generazione delle coordinate di una texture map per environment mapping può non essere fatta manualmente dal programmatore, ma gestita in modo autonomo da un set di tre funzioni distinte: *normal vector mapping*, *reflection vector mapping* e *sphere mapping*. La scelta della funzione da utilizzare si effettua passando al comando `glTexGen` il parametro appropriato: `GL_NORMAL_MAP`, `GL_REFLECTION_MAP`, o `GL_SPHERE_MAP`.

Nella prima funzione, la texture map è generata e applicata ad una superficie in base alla direzione delle normali della superficie stessa. Vengono utilizzate le tre componenti della vertex normal[1] *N* come coordinate della texture tramite il mapping di $N_X$, $N_Y$, $N_Z$ rispettivamente in *s*, *t*, *r*. I vettori vengono assunti di lunghezza unitaria in modo che le coordinate generate varino da -1 a 1. Questa funzione è adatta per la simulazione di una riflessione diffusa su una superficie.

Nella seconda funzione, la generazione della texture map si basa sul reflection vector della superficie: questo vettore è calcolato considerando la vertex normal *N* e il vettore del punto di vista *I* [6]. Quest'ultimo vettore è di lunghezza unitaria e diretto dalla posizione del punto di vista verso il vertice che si sta considerando. Il reflection vector *R* (Figura 1.4) è generato secondo l'equazione:

$$R = I - 2N\,(N \cdot I)$$

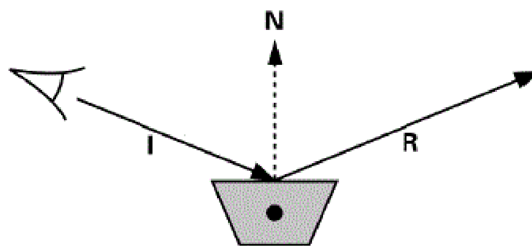


**Figura 1.4 Rappresentazione grafica dei vettori I, N, R**

[1] In computer grafica, la vertex normal di un vertice di un poliedro è un vettore calcolato come media normalizzata delle normali delle facce che contengono quel vertice.

Calcolato il reflection vector, le sue componenti sono convertite nelle coordinate della texture attraverso il mapping di $R_X$, $R_Y$, $R_Z$ rispettivamente in *s*, *t*, *r*. Dato che sia *I* che *N* sono normalizzati, lo è anche il risultante *R* e quindi le coordinate variano nell'intervallo [-1, 1]. Questa funzione è adatta a simulare una riflessione speculare su una superficie.

Mentre nelle due funzioni precedenti le coordinate generate sono tre, la terza funzione ne produce solo due: *s*, *t*. Viene calcolato dapprima un reflection vector, ottenuto come spiegato in precedenza, e poi vengono scalate le componenti $R_X$ e $R_Y$ di un fattore *m*. Tale fattore è calcolato come:

$$m = 2\sqrt{R_X^2 + R_Y^2 + (R_Z + 1)^2}$$

Dividendo $R_X$ e $R_Y$ per $m$ si proiettano le due componenti in un vettore che descrive una circonferenza unitaria nel piano $R_Z = 0$, di raggio 0.5 e centrata in (0.5, 0.5). Quindi $R_X$ e $R_Y$ sono confinati nell'intervallo [0, 1] e possono essere utilizzati come coordinate *s*, *t* (Figura 1.5).

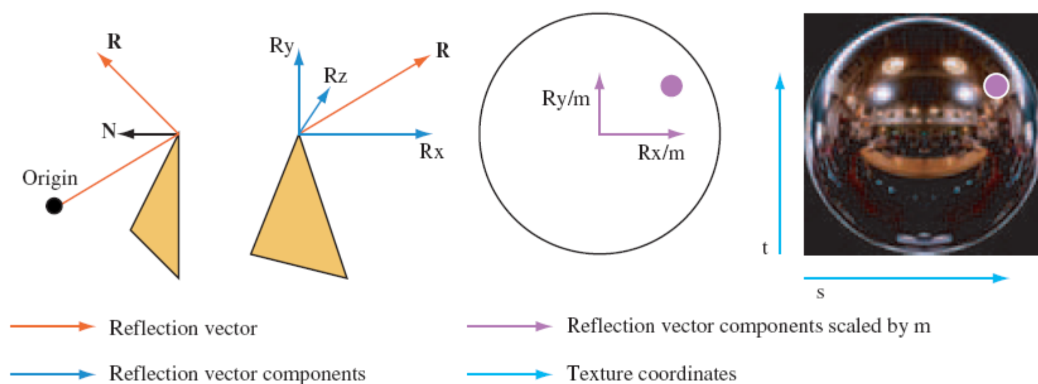


**Figura 1.5 Generazione coordinate della sphere map**

Mentre lo sphere mapping, generando due sole coordinate, può utilizzare una tradizionale texture 2D, il normal mapping e il reflection mapping, che generano tre coordinate, hanno necessità di una texture map a tre dimensioni. La texture map più utilizzata in questi casi è la *cube map*, discussa nei prossimi paragrafi.

### 1.3.2 Sphere Mapping

OpenGL ha supportato nativamente l'environment mapping, ma nelle prime versioni solo attraverso lo sphere mapping. Come già accennato, questa tecnica si avvale di una singola texture 2D ed è quindi semplice e veloce da implementare. Spesso l'immagine della texture è una fotografia scattata utilizzando un obiettivo *fisheye*[2], quindi distorta sia orizzontalmente che verticalmente (Figura 1.6).

Tuttavia una sphere map presenta spesso alcuni difetti nella colorazione dei pixel in corrispondenza dei bordi della superficie su cui viene applicata, soprattutto se in movimento. Inoltre la sphere map è *view-dependent*, cioè viene creata a partire da un punto di vista e di conseguenza il riflesso può considerarsi accurato solo per quello specifico punto di vista oppure sotto l'ipotesi di un ambiente infinitamente lontano [9]. Questa è la limitazione più grave dello sphere mapping, poiché la riflessione che si ottiene non è fisicamente corretta.

A causa dei difetti menzionati lo sphere mapping non è la tecnica preferibile per ottenere una riflessione accurata dell'ambiente, ma può tuttavia essere impiegata quando l'accuratezza non è indispensabile o quando la superficie riflettente non ha un'alta specularità. In tutti gli altri casi si utilizza il cube mapping, una tecnica più recente e affidabile.

---

[2] In fotografia, il fisheye è un obiettivo grandangolare estremo, con un angolo di campo non minore di 180°, che produce foto circolari con una distorsione elevata.

**Figura 1.6 Una tipica foto prodotta da un obiettivo fisheye**

### 1.3.3 Cube Mapping

Le potenzialità dell'environment mapping sono state sfruttate al meglio in versioni successive di OpenGL con l'introduzione del normal mapping e del reflection mapping insieme ad una nuova texture map: la *cube map*.

Una cube map è una texture particolare composta a sua volta da sei texture 2D che possono essere rappresentate come se fossero disposte sulle sei facce di un cubo [10]. In questo caso le coordinate *s*, *t*, *r* rappresentano un vettore che nasce dal centro del cubo e sono limitate nell'intervallo [-1.0, 1.0]. La componente di ampiezza maggiore viene usata per definire quale faccia si sta selezionando, mentre le altre due componenti indicano il texel di quella faccia che è preso in considerazione. Per creare una cube map si utilizza il comando tradizionale `glTexImage2D` passando come `target`, anziché `GL_TEXTURE_2D`, quello relativo alla faccia che si sta generando (Figura 1.7). Anche per generare le mipmap si utilizzano i metodi tradizionali, con l'unica differenza del `target`.

```
GL_TEXTURE_CUBE_MAP_POSITIVE_X
GL_TEXTURE_CUBE_MAP_NEGATIVE_X
GL_TEXTURE_CUBE_MAP_POSITIVE_Y
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y
GL_TEXTURE_CUBE_MAP_POSITIVE_Z
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z
```

**Figura 1.7 Valori assunti dal target per una cube map**

Supponendo di disporre di sei texture bidimensionali, ognuna raffigurante la vista dell'ambiente in una delle sei direzioni e con un angolo di campo di 90°, è possibile creare una cube map contenente una vista omnidirezionale (Figura 1.8).

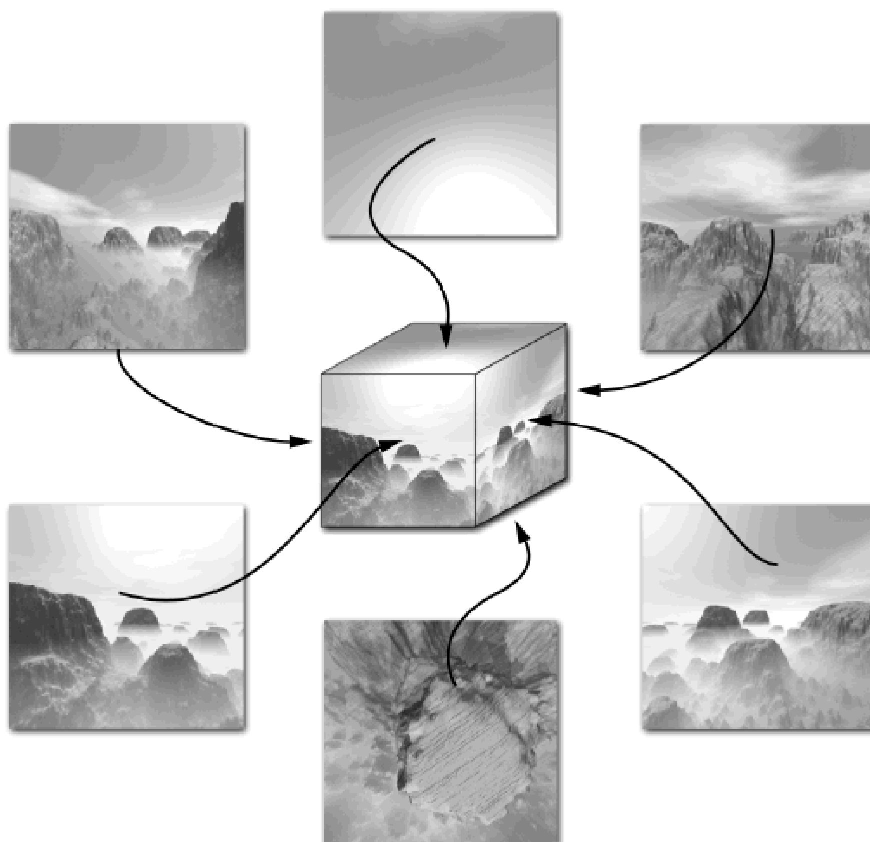

**Figura 1.8 Una cube map, composta da sei texture 2D**

Si può pensare di applicare la cube map a una superficie curva, come una sfera: per farlo va abilitata la generazione automatica delle coordinate e vanno configurati i relativi parametri. In questo modo OpenGL calcolerà autonomamente i texel da applicare ai vertici della geometria [11]. I comandi sono i seguenti:

```
glEnable(GL_TEXTURE_CUBE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
```

A questo punto è sufficiente disegnare la sfera e la texture sarà applicata sulla sua superficie. La figura 1.9 mostra l'effetto finale che si ottiene dal rendering: la sfera sembra riflettere l'ambiente che la circonda.

**Figura 1.9 Una cubemap applicata a una sfera: la superficie "riflette" l'ambiente**

Il modo in cui la cube map viene applicata alla sfera non è scontato: si tratta infatti di sei texture bidimensionali su una superficie tridimensionale curva. In pratica la sfera viene suddivisa in sei parti ad ognuna delle quali viene applicata una delle sei texture della cube map. La figura 1.10 mostra tale suddivisione e la deformazione subita da ogni texture: si noti come la deformazione non sia uniforme, ma più accentuata in corrispondenza dei bordi delle regioni [6].

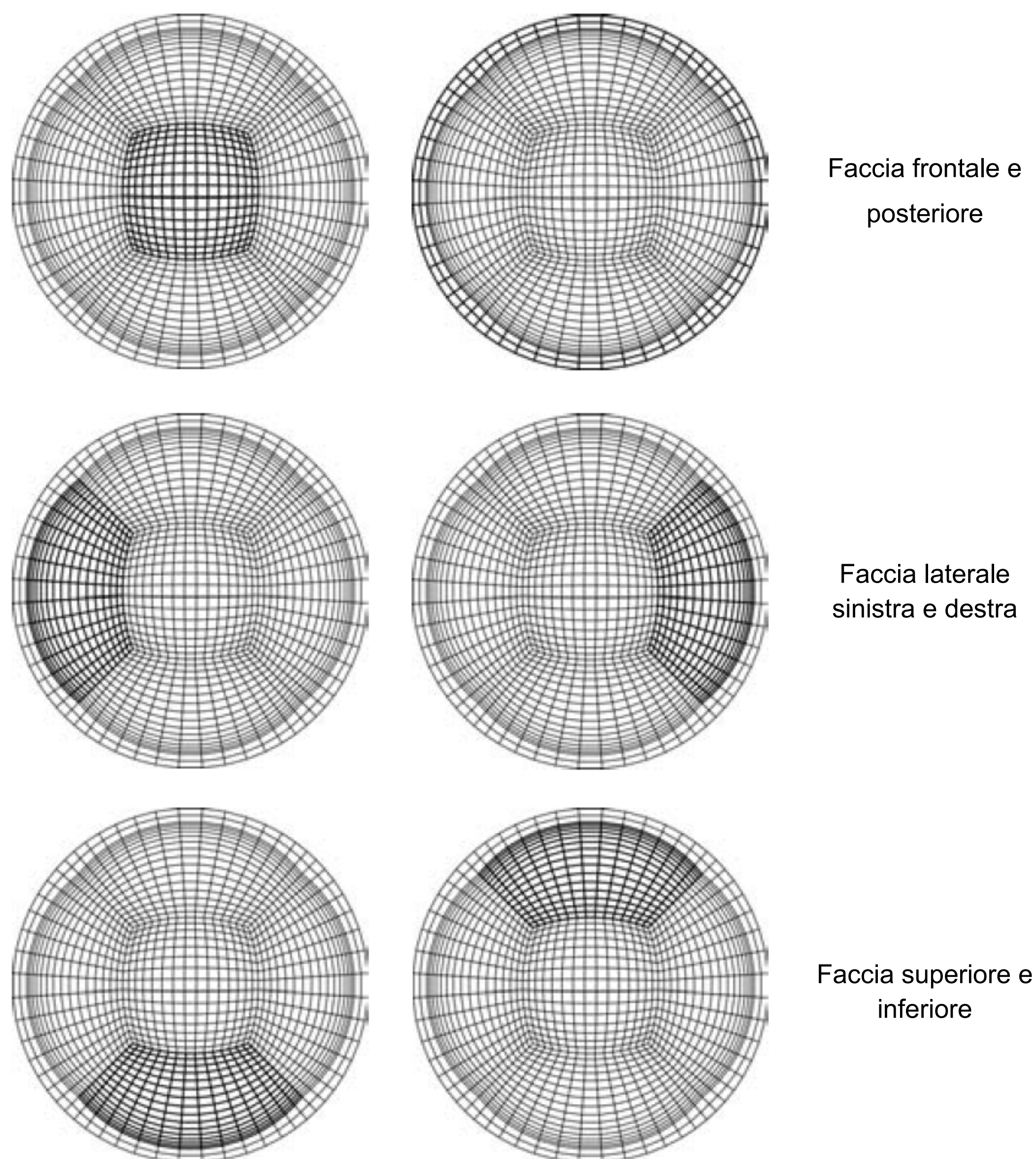


**Figura 1.10 Suddivisione di una sfera in sei regioni. Ogni texture della cube map viene opportunamente deformata e applicata a una regione della sfera.**

Prima di concludere questa panoramica sul cube mapping sono fondamentali alcune considerazioni: innanzitutto si può notare come una cube map, composta da sei texture, occupi significativamente più memoria di una sphere map composta da una sola texture. In secondo luogo è molto importante rilevare che una cube map, al contrario di una sphere map, è *view-independent*, cioè il modo in cui viene generata non dipende dal punto di vista dell'osservatore. Questo pregio è dovuto al fatto che la cube map complessivamente copre sempre un angolo di campo di 360° ed è valida qualsiasi sia la posizione dell'osservatore.

Date le sue caratteristiche, il cube mapping viene considerato il metodo più adatto a simulare effetti di riflessione speculare realistici su superfici curve ed è quello che si è scelto di utilizzare nel software sviluppato in questa tesi.

# 1.4 Riflessioni speculari dinamiche

A prescindere dalla scelta tra sphere o cube mapping, l'environment mapping che si realizza presenta un limite rilevante: è statico, cioè non riproduce eventuali cambiamenti nell'ambiente poichè si limita ad applicare le medesime texture. Per riflettere una variazione nell'ambiente, come ad esempio un oggetto in movimento, vanno rigenerate le texture ogni qualvolta si verifica la variazione. Si parla allora di *environment mapping dinamico*.

Va puntualizzato che una sphere map, essendo view-dependent, andrebbe rigenerata non solo quando l'ambiente subisce una modificazione, ma anche quando si sposta solamente il punto di vista. Questa necessità richiede un notevole incremento della quantità di calcoli da effettuare ed è un altro motivo per il quale è preferibile adottare il cube mapping, che è view-independent. Infatti è sufficiente applicare una rotazione alla *texture matrix* per orientare una cube map coerentemente con il nuovo punto di vista.

Per calcolare l'angolo di tale rotazione e l'asse su cui avviene bisogna conoscere due vettori [12]:

1. Un vettore “direzione”, che va dal centro della cube map alla posizione della vista;
2. Un vettore “orientamento”, che indica l’attuale disposizione della cube map.

Il prodotto vettoriale dei due vettori restituisce l’asse di rotazione, mentre l’arcoseno del prodotto scalare restituisce l’angolo di rotazione. Ricalcolando i vettori ogni volta che varia il punto di vista si trova la rotazione da applicare alla texture matrix per ottenere la riflessione corretta.

Per ottenere una cube map dinamica, invece, vanno rigenerate le sei texture che la compongono, un procedimento in cui la sorgente delle texture non è un’immagine digitale memorizzata, ma è il rendering in tempo reale della scena. Le operazioni fondamentali da compiere sono le seguenti:

1. Impostare una vista prospettica con angolo di campo a 90°;
2. Orientare il punto di vista verso uno degli assi;
3. Eseguire il rendering della scena;
4. Generare la texture.

Eseguendo questo procedimento sei volte, ognuna con un differente orientamento ortogonale del punto di vista, si riescono a rigenerare tutte le facce della cube map. Infine si esegue il rendering finale della scena applicando la cube map all’oggetto che deve riflettere l’ambiente.

Naturalmente la necessità di dover eseguire sei rendering della scena solo per la rigenerazione della cube map ha un impatto significativo sulle prestazioni complessive dell’applicazione e può essere conveniente valutare e variare la frequenza dell’aggiornamento. Ad esempio, si può optare di eseguire un aggiornamento delle texture ad intervalli di tempo regolari, piuttosto che ad ogni variazione dell’ambiente. Comunque è un aspetto che va valutato caso per caso, tenendo in considerazione la complessità della scena da renderizzare e la velocità di variazione dell’ambiente.

Il punto cruciale di tutta la procedura per ottenere un environment mapping dinamico è la generazione delle texture a partire dal rendering della scena, argomento chiave del prossimo capitolo.

# Capitolo 2

# Generazione di texture dinamiche

Le diverse texture map viste nel capitolo precedente hanno una caratteristica in comune: sono statiche, cioè generate da un'immagine digitale e prima che l'applicazione inizi il rendering. Con il termine *texturing dinamico* si intende la creazione "al volo" di texture avendo come sorgente il rendering di una scena anziché un'immagine. Questo processo si può sintetizzare molto genericamente in tre passi principali:

1. Eseguire un rendering preliminare della scena;
2. Creare una texture dal rendering effettuato;
3. Eseguire il rendering della texture applicandola a una geometria.

L'impiego di texture dinamiche rende possibile la realizzazione di effetti grafici particolari, come *motion blur*, *anti-aliasing, dynamic environment mapping* e molti altri. In questo capitolo viene illustrato il funzionamento generale del Framebuffer e vengono esaminate le modalità con cui si può attuare il texturing dinamico, con particolare riferimento all'impiego del *Framebuffer Object*.

# 2.1 Gestione del Framebuffer

OpenGL utilizza diversi buffer per mantenere in memoria le informazioni riguardanti i pixel dell'area di rendering. I buffer differiscono tra loro per il tipo di informazioni che contengono e il loro insieme si definisce *Framebuffer*.
I buffer messi a disposizone da OpenGL sono principalmente quattro:

- *Color Buffer*
- *Depth Buffer*
- *Stencil Buffer*
- *Accumulation Buffer*

Il color buffer memorizza il colore di ogni pixel della scena per ognuna delle componenti RGBA (Red, Green, Blue e Alpha). Normalmente la dimensione del color buffer è di 32 bit, 8 bit per ognuna delle componenti, ma è possibile creare buffer non simmetrici o addirittura privi di un canale, ad esempio impostando il canale Alpha a 0 bit se non si ha necessità di effettuare trasparenze o effetti simili.

Un miglioramento considerevole della qualità del rendering si ottiene impiegando più di un color buffer mediante il *Double Buffering*. Utilizzando un double buffer vengono creati due buffer colore, uno principale chiamato *Front buffer* e uno nascosto chiamato *Back buffer*. Le operazioni di rendering avvengono sul back buffer, mentre il front buffer viene visualizzato sullo schermo. Alla fine del rendering i due buffer vengono scambiati trasformando il buffer nascosto nel buffer principale e viceversa. Dato che il buffer visualizzato sullo schermo è sempre quello principale, la tecnica del double buffering permette di visualizzare il rendering completo della scena, anziché mostrarla mentre viene disegnata.

Il depth buffer memorizza la profondità di ogni pixel, valore che varia a seconda della posizione all'interno della scena e che è maggiore quanto più è distante il pixel dal punto di vista. Le informazioni sulla profondità sono indispensabili per

visualizzare gli oggetti in base alla distanza dal punto di vista invece che in base all'ordine nel quale vengono disegnati. Di norma tutte le applicazioni OpenGL utilizzano il depth buffer e la sua dimensione è di solito di 16 bit.

Lo stencil buffer consente di disegnare selettivamente su alcune porzioni della scena e di bloccare il rendering sul resto della scena. Va creata una "maschera" per definire le zone in cui è consentito disegnare facendo il rendering preventivo di una geometria sullo stencil buffer. Per ogni pixel modificato da questo rendering verrà memorizzato un valore pari a 1 nello stencil buffer. A questo punto renderizzando la scena è possibile specificare di modificare un pixel solo se il valore relativo al dato pixel nello stencil buffer è uguale a 1, ovvero appartenente alla maschera. In questo modo il rendering avverrà solo all'interno dell'area precedentemente delimitata. In base alla dimensione in bit dello stencil buffer è possibile creare diversi strati di maschere e ottenere diversi effetti grafici, come ombre realistiche, riflessioni, rendering all'interno di forme irregolari.

L'accumulation buffer è un buffer aggiuntivo che si usa per "accumulare" il contenuto del color buffer: l'idea è quella di eseguire diversi rendering sul color buffer e copiarli man mano nell'accumulation buffer, sommandoli. Ad esempio è possibile generare la media di diversi rendering eseguiti sul color buffer. Copiando poi il contenuto dell'accumulation buffer sul color buffer si visualizza l'effetto prodotto. Un utilizzo tipico che si fa dell'accumulation buffer è quello di impiegarlo per realizzare semplici effetti di anti-aliasing.

# 2.2 Tecniche di rendering su texture

L'uso crescente di effetti grafici avanzati nelle applicazioni grafiche ha reso necessario lo studio di tecniche sempre più efficienti per la creazione di texture dinamiche. Infatti, come mostrato da Simon Green di NVIDIA [13], i metodi di lettura del framebuffer pixel per pixel e la successiva scrittura su texture non erano sufficientemente veloci per implementare gli effetti grafici desiderati. Il collo di bottiglia principale era rappresentato dalla necessità di dover fare un rendering e poi una copia del contenuto del framebuffer in una texture.

Una svolta decisiva è stata compiuta con l'introduzione di tecniche denominate *Render-To-Texture (RTT)* in cui il rendering della scena viene fatto direttamente su texture anziché sul framebuffer. Le operazioni che si compiono in tal caso si possono così sintetizzare:

1. Allocazione di una texture
2. Creazione di un oggetto "Render-To-Texture"
3. Selezione dell'oggetto RTT come target del rendering
4. Rendering della scena
5. Selezione del framebuffer come target del rendering
6. Associazione dell'oggetto RTT alla texture
7. Rendering della texture applicata a una geometria

Fino a pochi anni fa non esisteva un vero e proprio oggetto RTT e si ricorreva ad altri metodi, come l'uso del *Pixel Buffer Object* (*PBO*). Il vantaggio principale di un PBO è la velocità di trasferimento dei dati provenienti e diretti alla scheda video grazie all'accesso diretto alla memoria (DMA).

Il diagramma in figura 2.1 mostra il modo convenzionale di caricare una texture: la sorgente (es. un'immagine) viene prima caricata nella memoria di sistema e poi copiata in un oggetto texture. Entrambe le operazioni sono effettuate dalla CPU. Nel diagramma in figura 2.2, invece, la sorgente viene caricata direttamente su un

PBO, controllato da OpenGL. In questo caso non è la CPU a occuparsi della copia tra il PBO e l'oggetto texture, ma è la GPU tramite un accesso diretto alla memoria. Così facendo la CPU ha un minor carico di lavoro e può eseguire altre operazioni.

Più recentemente è stato introdotto il Framebuffer Object che permette di eseguire un vero e proprio rendering su texture e sostituisce in modo definitivo il PBO per questo specifico tipo di operazioni [14].

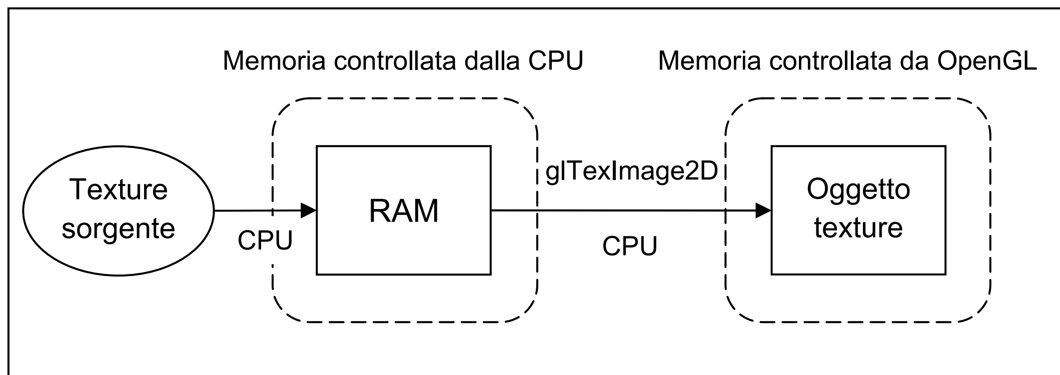


**Figura 2.1 Diagramma del caricamento di una texture senza l'impiego del PBO**

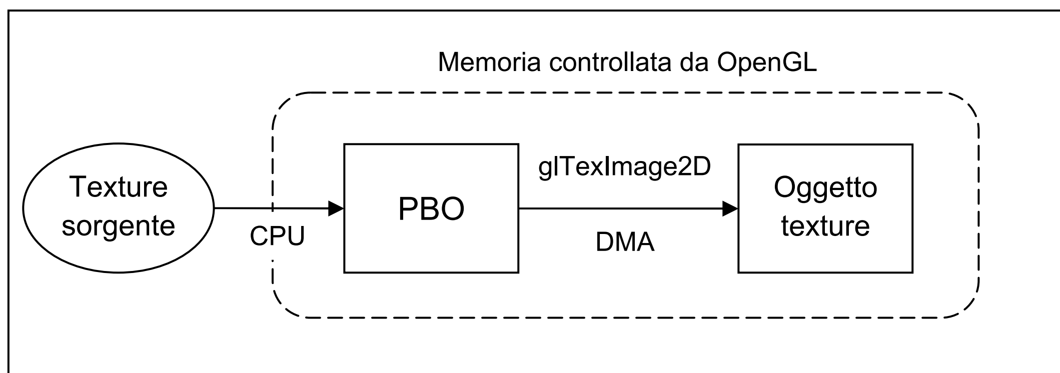


**Figura 2.2 Diagramma del caricamento di una texture con l'impiego del PBO**

### 2.2.1 Architettura del Framebuffer Object

Come illustrato precedentemente, il rendering viene fatto su un framebuffer predefinito che viene chiamato *window-system-provided* framebuffer, cioè creato e gestito interamente dal sistema e il cui contenuto è visualizzato direttamente a video.

Utilizzando il Framebuffer Object è possibile indirizzare il rendering su un *application-created* framebuffer che è controllato direttamente da OpenGL e il cui contenuto non viene visualizzato a video.

Analogamente ai diversi buffer di cui è composto il framebuffer tradizionale, anche un Framebuffer Object è formato da diversi elementi, chiamati *attachment point*, che si suddividono in quattro categorie:

- `GL_COLOR_ATTACHMENT`
- `GL_DEPTH_ATTACHMENT`
- `GL_STENCIL_ATTACHMENT`
- `GL_DEPTH_STENCIL_ATTACHMENT`

Per poter essere utilizzato, un attachment point deve essere associato a un oggetto chiamato *framebuffer attachable image* che può essere di due tipi:

- *Texture*
- *Renderbuffer*

La texture è l'oggetto utilizzato comunemente per la memorizzazione di immagini e in questo caso si sfrutta per eseguire dei rendering direttamente su texture; il renderbuffer è un oggetto che è stato introdotto contestualmente al Framebuffer Object e si usa per eseguire dei rendering *offscreen*, che non vengono visualizzati a video. Solitamente il renderbuffer si associa ai buffer per cui non ha senso l'associazione con una texture, come depth buffer e stencil buffer.

In figura 2.3 si può osservare uno schema dell'architettura del Framebuffer Object che è stata presentata al Game Developers Conference 2005 tenutosi a San Francisco, California.

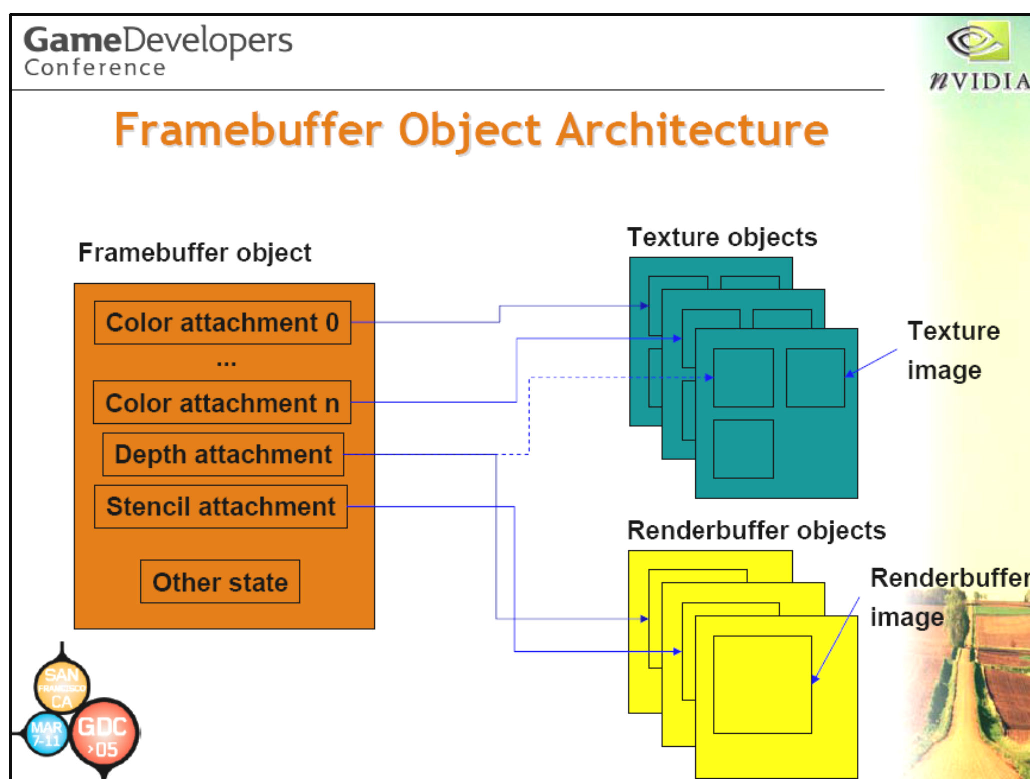


**Figura 2.3 L'architettura del Framebuffer Object [13]**

## 2.2.2 Configurazione di un Framebuffer Object

I primi passi da compiere per disporre di un Framebuffer Object sono la creazione tramite il comando `glGenFramebuffersEXT()` e la sua abilitazione con `glBindFramebufferEXT()`. Da questo momento tutte le operazioni OpenGL vengono effettuate sul Framebuffer Object finché non viene disabilitato. Il passo successivo consiste nell'effettuare le associazioni degli attachment point con renderbuffer e texture. Per creare un renderbuffer e abilitarlo si usano i comandi `glGenRenderbuffersEXT()` e `glBindRenderbufferEXT()`, ma non viene

allocato automaticamente uno spazio di memoria. Il comando `glRenderbufferStorageEXT()` serve per definire le dimensioni della memoria da allocare e il tipo di dati che conterrà. Infine per specificare a quale attachment point viene associato il renderbuffer si usa il comando `glFramebufferRenderbufferEXT()`.

Per l'associazione con una texture, invece, si usa il comando `glFramebufferTexture2DEXT()` in cui si specifica sia l'attachment point da associare, sia la texture dove eseguire il rendering.

A questo punto la configurazione è completa, ma va verificato lo stato del Framebuffer Object con `glCheckFramebufferStatusEXT()` che restituisce `GL_FRAMEBUFFER_COMPLETE_EXT` se la configurazione è corretta. In caso contrario il valore restituito dipende dal tipo di errore che è stato rilevato. In generale un Framebuffer Object ben configurato deve presentare le seguenti caratteristiche fondamentali:

- Le dimensioni delle attachable image sono diverse da zero;
- Un attachable image associata a un color attachment point definita con un formato adeguato (es. `GL_RGB`, `GL_RGBA`, ...);
- Un attachable image associata a un depth attachment point definita con un formato adeguato (es. `GL_DEPTH_COMPONENT`);
- Un attachable image associata a uno stencil attachment point definita con un formato adeguato (es. `GL_STENCIL_INDEX`);
- Al Framebuffer Object è associato almeno un attachable image;
- Tutte le attachable image hanno le stesse dimensioni.

Nel caso in cui queste condizioni fossero soddisfatte ma lo stato del Framebuffer Object risultasse `GL_FRAMEBUFFER_UNSUPPORTED_EXT` significherebbe che la combinazione di formati e parametri scelti non è supportata dalla scheda video.

Il codice completo della configurazione del Framebuffer Object che è stata impiegata in questa tesi verrà presentato nel prossimo capitolo.

## 2.3 Realizzazione di un environment mapping dinamico

Una volta creato un Framebuffer Object è possibile sfruttarlo per realizzare un environment mapping dinamico. Nella configurazione il depth attachment deve essere associato con un renderbuffer e il color attachment con una cube map texture. A questo punto è necessario creare una funzione da chiamare all'interno della funzione display di OpenGL ogni qual volta si rende necessario rigenerare la cube map. Le operazioni che la funzione deve compiere sono:

1. Attivare il rendering su FBO;
2. Impostare le dimensioni dell'area di rendering (viewport);
3. Impostare una vista prospettica con angolo di campo a 90°;
4. Orientare il punto di vista verso uno degli assi;
5. Associare una delle sei texture della cubemap al color attachment;
6. Eseguire il rendering della scena;
7. Cancellare il contenuto dei buffer del FBO;
8. Rimuovere l'associazione del color attachment con la cubemap;
9. Ripristinare il rendering su framebuffer;
10. Ripristinare l'area di rendering e la vista prospettica precedenti.

Le operazioni dalla 4 alla 7 vanno ripetute per ogni texture della cubemap orientando il punto di vista verso un asse differente.

Per quanto riguarda l'impostazione dell'area di rendering (punto 2), le dimensioni devono coincidere con quelle impostate per il renderbuffer e la texture nella configurazione del FBO, ma possono essere superiori a quelle della finestra dell'applicazione [15]. Ad esempio, se le dimensioni della finestra dell'applicazione sono impostate a 800x600, il rendering su FBO può comunque essere eseguito su un'area di 1024x1024 per generare una texture con una

risoluzione superiore. Questo è un indubbio vantaggio offerto dal Framebuffer Object rispetto alle precedenti tecniche di rendering su texture.

La rimozione dell'associazione del color attachment con la cubemap (punto 8) è indispensabile poiché la cubemap è sia la destinazione in scrittura del rendering del FBO, sia la sorgente in lettura del rendering della superficie su cui viene applicata. Quindi bisogna sempre assicurarsi che la texture non sia contemporaneamente utilizzata sia in lettura che in scrittura, altrimenti si avrebbe un comportamento imprevedibile durante il rendering [16].

Infine la cubemap che è stata rigenerata può essere utilizzata normalmente abilitando l'environment mapping e la generazione delle coordinate, come illustrato nel precedente capitolo.

# Capitolo 3

# Progettazione del sistema

Questo capitolo è dedicato al sistema che è stato progettato e realizzato in questa tesi. Dopo una rapida panoramica sullo stato dell'arte, vengono illustrate le caratteristiche della libreria OpenGL e i motivi per cui si è scelto di impiegarla nella realizzazione del sistema. Successivamente viene descritta l'applicazione realizzata, presentando e commentando le parti di codice fondamentali, e infine si mostrano i risultati ottenuti.

## 3.1 Stato dell'arte

La possibilità di simulare una visione omnidirezionale dell'ambiente è un argomento di particolare interesse e studio, soprattutto in ambito robotico, ma i simulatori che sono stati progettati e realizzati non sono numerosi.

Tra questi va menzionato Webots [17], un simulatore commerciale in grado di simulare la fisica dei robot mobili oltre a sensori visivi, sonar e laser. Tuttavia non sono facilmente implementabili gli specchi catadiottrici, indispensabili nella visione omnidirezionale.

Un altro progetto degno di nota è quello illustrato in [18] in cui viene simulato un sistema omnidirezionale catadiottrico con specchio parabolico e si impiega il cube mapping per la generazione delle texture necessarie alla simulazione della riflessione speculare.

Uno studio con un approccio simile a quello seguito in questa tesi viene presentato in [19] in cui si fa uso della libreria OpenGL per generare, mediante cube mapping, delle riflessioni speculari che vengono applicate a uno specchio sferico.

Infine in [20, 21] si realizza un sistema di visione omnidirezionale stereo per la navigazione e localizzazione in ambito robotico. Il sistema si avvale di algoritmi *raytracing* per la simulazione delle riflessioni speculari su uno specchio conico. L'impiego del raytracing al posto di librerie grafiche come OpenGL offre dei vantaggi concreti in termini di fotorealismo della scena e di accuratezza delle riflessioni dal punto di vista fisico. Tuttavia l'elevata richiesta di potenza di calcolo da parte degli algoritmi raytracing rende il sistema inadatto all'utilizzo in applicazioni real-time.

## 3.2 La libreria OpenGL

In ambiente grafico, OpenGL (Open Graphics Library) è una API di sviluppo largamente utilizzata sia per applicazioni 2D che 3D ed è impiegata in molteplici ambiti, dalle applicazioni CAD ai videogiochi. La sua grande diffusione ha diverse ragioni: è potente, nel senso che può gestire il rendering di scene complesse con effetti di illuminazione, texture mapping e altri effetti grafici; è supportato nativamente da molti produttori di hardware; è multi-piattaforma e disponibile per le più importanti architetture attuali (Apple Macintosh, Microsoft Windows, Unix, OS/2, etc.); è gratuito e ben documentato.

Un'altra caratteristica molto apprezzata in ambito professionale è la retrocompatibilità tra le diverse versioni di OpenGL: programmi scritti per la

versione 1.0 della libreria devono funzionare senza modifiche su implementazioni che si avvalgono di versioni successive.

OpenGL assolve due funzioni fondamentali: nasconde la complessità di interfacciamento con acceleratori 3D differenti, offrendo al programmatore una API unica e uniforme; nasconde le capacità offerte dai diversi acceleratori 3D, richiedendo che tutte le implementazioni supportino completamente l'insieme di funzioni OpenGL, ricorrendo ad un'emulazione software se necessario.

Il compito di OpenGL è quello di ricevere primitive come punti, linee e poligoni, e di convertirli in pixel (procedimento chiamato *rasterizing* o *rasterizzazione*). Questo procedimento è realizzato da una pipeline grafica nota come *OpenGL State Machine*. La maggior parte dei comandi OpenGL forniscono primitive alla pipeline grafica o istruiscono la pipeline su come elaborarle.

Storicamente, OpenGL ha esercitato una notevole influenza sullo sviluppo degli acceleratori 3D, promuovendo un livello base di funzionalità che è oggi comune nelle schede video destinate al grande pubblico:

- punti, linee e poligoni disegnati come primitive base;
- una pipeline per il transform and lighting;
- Z-buffering;
- Texture mapping;
- Alpha blending.

A livello più basso OpenGL è una specifica, ovvero si tratta semplicemente di un documento che descrive un insieme di funzioni e il comportamento preciso che queste devono avere. Da questa specifica, i produttori di hardware creano implementazioni, ovvero librerie di funzioni create rispettando quanto riportato sulla specifica OpenGL, facendo uso dell'accelerazione hardware ove possibile. I produttori devono comunque superare dei test specifici per poter fregiare i loro prodotti della qualifica di implementazioni OpenGL.

La specifica di OpenGL è stata inizialmente supervisionata dall'OpenGL Architecture Review Board (ARB), formatosi nel 1992. L'ARB era compoosto da

un gruppo di aziende interessate a creare una API coerente e ampiamente disponibile. I membri fondatori dell'ARB comprendevano aziende del calibro di 3Dlabs, Apple Computer, ATI Technologies, Dell, IBM, Intel, NVIDIA, SGI, Sun Microsystems e Microsoft. Il coinvolgimento di così tante aziende con interessi molto diversificati, ha portato OpenGL a diventare nel tempo una API ad uso generico, con un ampio ventaglio di capacità.

Lo standard OpenGL permette ai produttori individuali di fornire funzionalità aggiuntive tramite delle *estensioni* alla libreria man mano che vengono create nuove tecnologie. Un'estensione viene poi distribuita in due parti: come file di intestazione che contiene i prototipi di funzione dell'estensione e come *driver* del produttore. Ogni produttore ha un'abbreviazione alfabetica che viene usata nel denominare le proprie funzioni e costanti. Per esempio, l'abbreviazione di NVIDIA (*NV*) viene usata nel definire la funzione proprietaria `glCombinerParameterfvNV()` e la costante `GL_NORMAL_MAP_NV`.

Può succedere che più produttori si accordino per implementare la stessa funzionalità e in tal caso si usa l'abbreviazione *EXT*. Può anche accadere che l'ARB approvi l'estensione. Allora essa diventa nota come una *estensione standard* e si usa l'abbreviazione *ARB*. La prima estensione ARB è stata la `GL_ARB_multitexture`. Avendo seguito il percorso di promozione per le estensioni ufficiali, la multitexture non è più un'estensione ARB implementata opzionalmente, ma è stata integrata nella API base di OpenGL.

# 3.3 Descrizione dell'applicazione realizzata

Come di consueto, in un'applicazione OpenGL prima del rendering vero e proprio è necessaria una fase di inizializzazione e configurazione dell'ambiente grafico e delle funzionalità che verranno impiegate. In questa fase vengono anche indicate tutte le funzioni che implementano il comportamento dei dispositivi di input/output. Infine la gestione passa alla funzione che si occupa del rendering, che viene richiamata ciclicamente fino al termine dell'esecuzione dell'applicazione.

## 3.3.1 Inizializzazione dell'ambiente grafico

Il primo passo nella realizzazione del sistema è l'inizializzazione dell'applicazione, ovvero delle dimensioni della finestra e della modalità di visualizzazione, e l'indicazione delle funzioni per la gestione dei dispositivi di input/output:

```
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
glutCreateWindow("Dynamic Environment Mapping using FBO");
glutInitWindowSize(WINDOW_WIDTH, WINDOW_HEIGHT);
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutKeyboardFunc(keyboard);
glutSpecialFunc(specialKeys);
```

Si può notare dalla prima istruzione che per la visualizzazione è stato impiegato sia il double buffering, per ottenere un rendering più accurato, sia il depth buffer grazie al quale è possibile disegnare gli oggetti in base alla loro distanza dal punto

di vista. Per quest'ultimo aspetto è necessario attivare il controllo della profondità della scena mediante le seguenti istruzioni:

```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LEQUAL);
```

Un controllo di fondamentale importanza è quello relativo al supporto delle estensioni OpenGL da parte dei driver della scheda video: infatti molte funzionalità grafiche non fanno parte della libreria standard OpenGL, ma sono sviluppate da diversi produttori di hardware e messe a disposizione come estensioni. Questa applicazione fa uso dell'estensione relativa al Framebuffer Object per il rendering su texture e il supporto a questa estensione è strettamente necessario per un corretto funzionamento. Di conseguenza se l'estensione non è supportata non sarà possibile eseguire l'applicazione.
Un'altra estensione che viene impiegata è quella relativa al filtro anisotropico[3], ma non è indispensabile per l'esecuzione dell'applicazione. Le funzionalità del cube mapping, invece, sono entrate a far parte della libreria standard OpenGL e non è più necessaria la relativa estensione. Le istruzioni per controllare il supporto alle estensioni menzionate sono:

```
glInfo.isExtensionSupported("GL_EXT_framebuffer_object")
glutExtensionSupported("GL_EXT_texture_filter_anisotropic")
```

### 3.3.2 Inizializzazione della cubemap

Dopo l'inizializzazione dell'ambiente grafico, l'operazione successiva è l'inizializzazione della cubemap che si utilizzerà per la modellazione dei riflessi speculari. Il primo passo consiste nel creare un oggetto cubemap:

[3] Il filtro anisotropico è un metodo utilizzato nella computer grafica per accrescere la qualità delle immagini che ritraggono texture poste su superfici inclinate rispetto al punto di osservazione.

```
glGenTextures(1, &cubemap);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemap);
```

Vanno poi definite le modalità con cui verranno generate le sei texture della cubemap e come saranno applicate a una geometria. Come descritto nel capitolo 1, nel caso di riflessioni speculari si utilizza la modalità `GL_REFLECTION_MAP` per la generazione delle coordinate:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexParameterf(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
                GL_CLAMP_TO_EDGE);
glTexParameterf(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
                GL_CLAMP_TO_EDGE);
glTexParameterf(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R,
                GL_CLAMP_TO_EDGE);
```

Si passa poi alla scelta del filtraggio da operare sulla texture quando le sue dimensioni non sono uguali a quelle della geometria a cui si applica. In questa fase si abilita anche l'impiego delle mipmap e si definisce la qualità massima del filtraggio anisotropico:

```
glTexParameterf(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR_MIPMAP_LINEAR);
glTexParameterf(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,
                GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_GENERATE_MIPMAP,GL_TRUE);

if (anisotropicSupported)
glTexParameterf(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_MAX_ANISOTROPY_EXT,
                fLargest);
```

Infine va allocato lo spazio di memoria per le sei texture della cubemap, specificando il formato e le dimensioni che avrà l'immagine. Per farlo si utilizza il comando `glTexImage2D`, che viene solitamente usato per copiare i dati di un'immagine digitale in una texture, passando il parametro `NULL` invece del puntatore all'area di memoria che contiene i dati. Così facendo si riserva solamente uno spazio di memoria di dimensioni adeguate, senza creare una texture vera e propria:

```
for (int i=0; i<6; i++)
glTexImage2D(faceTarget[i], 0, GL_RGBA, TEXTURE_WIDTH,
             TEXTURE_HEIGHT, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
```

### 3.3.3 Inizializzazione del Framebuffer Object

Prima dell'inizializzazione del Framebuffer Object viene interrogato il sistema circa le funzionalità OpenGL supportate dalla scheda video e le impostazioni correntemente in uso, tramite le istruzioni:

```
glInfo glInfo;
glInfo.getInfo();
glInfo.printSelf();
```

Il risultato viene mostrato in una finestra DOS (figura 3.1) contenente le informazioni di maggior interesse: è importante che la versione della libreria OpenGL in uso non sia inferiore alla 2.0, in caso contrario l'applicazione potrebbe non funzionare in modo corretto.

Vale la pena ricordare che la libreria OpenGL è parte integrante dei driver della scheda video e di conseguenza è possibile disporre della versione più recente mediante l'aggiornamento dei driver della scheda video, forniti dal produttore dell'hardware.

Nella finestra DOS viene anche mostrata una lista con tutte le estensioni supportate dalla scheda video in modo da fornire all'utente un riscontro immediato circa la disponibilità di impiego del Framebuffer Object (Figura 3.2), indispensabile al funzionamento dell'applicazione.

```
OpenGL Driver Info
==================
Vendor: NVIDIA Corporation
Version: 3.0.0
Renderer: GeForce 9600M GT/PCI/SSE2

Color Bits(R,G,B,A): (8, 8, 8, 0)
Depth Bits: 24
Stencil Bits: 0

Max Texture Size: 8192x8192
Max Lights: 8
Max Clip Planes: 6
Max Modelview Matrix Stacks: 32
Max Projection Matrix Stacks: 4
Max Attribute Stacks: 16
Max Texture Stacks: 10

Total Number of Extensions: 161
==============================
```

**Figura 3.1 Le funzionalità OpenGL supportate dalla scheda video**

```
GL_EXT_blend_subtract
GL_EXT_compiled_vertex_array
GL_EXT_depth_bounds_test
GL_EXT_direct_state_access
GL_EXT_draw_buffers2
GL_EXT_draw_instanced
GL_EXT_draw_range_elements
GL_EXT_fog_coord
GL_EXT_framebuffer_blit
GL_EXT_framebuffer_multisample
GL_EXT_framebuffer_object
GL_EXT_framebuffer_sRGB
GL_EXT_geometry_shader4
GL_EXT_gpu_program_parameters
GL_EXT_gpu_shader4
GL_EXT_multi_draw_arrays
GL_EXT_packed_depth_stencil
GL_EXT_packed_float
GL_EXT_packed_pixels
GL_EXT_pixel_buffer_object
```

**Figura 3.2 Alcune delle estensioni supportate dalla scheda video**

Se il Framebuffer Object è supportato, si passa alla sua creazione e abilitazione con due istruzioni simili a quelle usate per le texture:

```
glGenFramebuffersEXT(1, &fboID);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboID);
```

Come illustrato nel capitolo 2, il Framebuffer Object è composto da diversi attachment point che vanno configurati. In questa applicazione ci si avvale esclusivamente del depth attachment e del color attachment e si può quindi tralasciare la configurazione dello stencil attachment.

L'utilizzo del depth attachment è indispensabile per ottenere un rendering che tenga conto della distanza degli oggetti dal punto di vista e deve essere associato a un oggetto renderbuffer:

```
glGenRenderbuffersEXT(1, &rboID);
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, rboID);
glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT,
                         GL_DEPTH_COMPONENT16,
                         TEXTURE_WIDTH, TEXTURE_HEIGHT);
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, 0);
glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
                             GL_DEPTH_ATTACHMENT_EXT,
                             GL_RENDERBUFFER_EXT, rboID);
```

Al contrario il color attachment deve essere associato con la cubemap creata precedentemente, in modo da poter eseguire in seguito un rendering su texture:

```
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT,
                          faceTarget[0], cubemap, 0);
```

Infine è necessario controllare che il Framebuffer Object sia configurato correttamente:

```
bool status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
```

Si rammenta che una condizione necessaria per una corretta configurazione è l'esatta corrispondenza delle dimensioni del renderbuffer con le dimensioni della texture.

Nel caso in cui la configurazione non fosse corretta, viene mostrato in finestra DOS un messaggio indicante la causa dell'errore. In tal caso per avere informazioni più dettagliate si può fare riferimento alle specifiche del Framebuffer Object [22]. Se invece la configurazione risulta corretta (Figura 3.3) il Framebuffer Object può essere impiegato con successo per il rendering su texture.

```
==================================================================
Video card supports GL_EXT_framebuffer_object.

***** FBO STATUS *****
Max Number of Color Buffer Attachment Points: 8
Color Attachment 0: GL_TEXTURE, 512x512, GL_RGBA
Depth Attachment: GL_RENDERBUFFER_EXT, 512x512, GL_DEPTH_COMPONENT

[SUCCESS] Framebuffer complete.
```

**Figura 3.3 La configurazione del Framebuffer Object risulta corretta**

## 3.3.4 Interfaccia grafica

L'interfaccia grafica dell'applicazione si presenta in una finestra di 1200x450 pixel suddivisa in due viewport affiancate in cui vengono eseguiti due rendering della scena differenti. Nella viewport a sinistra il punto di vista è orientato inizialmente in direzione dello specchio sferico, ma può essere variato (ruotato e traslato) per simulare un ipotetico osservatore in grado di navigare liberamente nell'ambiente virtuale. Al contrario, il punto di vista nella viewport a destra è

fisso e sempre posizionato sotto la sfera, in modo da simulare un sensore visivo composto da uno specchio e una telecamera che lo inquadra.

È presente anche una modalità con una viewport singola: in questo caso la finestra viene ridimensionata a 800x600 pixel ed è possibile scegliere quale dei due punti di vista si desidera utilizzare. In qualsiasi momento è possibile passare da una modalità all'altra, attraverso un menù a tendina richiamabile con il mouse. Naturalmente l'utilizzo di due viewport richiede due rendering separati e dunque è più oneroso rispetto alla modalità con viewport singola.

Per la creazione delle due viewport le istruzioni sono le seguenti:

```
for (int i = 0; i < viewports; i++) {
  switch (i) {

    case 0:
    glViewport(0, 0, VIEWPORT_WIDTH, VIEWPORT_HEIGHT);
    glLoadIdentity();
    if (multiview) staticcamera = false;
    renderAll();
    break;

    case 1:
    glViewport(VIEWPORT_WIDTH+1, 0,
               VIEWPORT_WIDTH-1, VIEWPORT_HEIGHT);
    glLoadIdentity();
    if (multiview) staticcamera = true;
    renderAll();
    break;
  }
}
```

La funzione `renderAll()` si occupa del rendering della scena per entrambe le viewport ed è illustrata nel paragrafo 3.3.6.

### 3.3.5 Generazione della cubemap dinamica

Completate le fasi preliminari di inizializzazione e configurazione, si passa alla gestione del rendering su texture con cui si genera la cubemap dinamica da applicare successivamente alla sfera, in modo da simulare la riflessione speculare dinamica su uno specchio sferico.

Innanzitutto è necessario salvare le impostazioni relative al rendering su framebuffer (*viewport* e matrice *modelview*), in modo da ripristinarle una volta terminato il rendering su texture:

```
glMatrixMode(GL_MODELVIEW);
glPushAttrib(GL_VIEWPORT_BIT);
glPushMatrix();
```

Ora si possono abilitare il rendering su Framebuffer Object e definire le dimensioni dell'area di rendering (viewport), che devono essere le stesse della texture e del renderbuffer che si stanno utilizzando. In questa applicazione si è scelto di utilizzare una texture quadrata di 512x512 pixel, dimensione sufficiente per ottenere una riflessione di buona qualità.

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboID);
glViewport(0, 0, TEXTURE_WIDTH, TEXTURE_HEIGHT);
```

Un'operazione fondamentale è l'impostazione di una vista prospettica con un angolo di campo di 90°, in modo da riuscire ad avere una visuale a 360° orientando semplicemente la vista lungo gli assi x, y, z.

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
gluPerspective( 90.0f,
                (GLfloat)TEXTURE_WIDTH/(GLfloat)TEXTURE_HEIGHT,
                0.5f, 100.0f );
```

I rendering su texture si eseguono mediante un ciclo in cui si imposta ogni volta un punto di vista orientato verso un certo asse, diverso a seconda della faccia della cubemap da renderizzare. Si associa poi la faccia della cubemap al Framebuffer Object, in modo che il rendering sia eseguito su di essa, e si esegue il rendering su texture della scena. Ovviamente la scena deve essere composta da tutto l'ambiente e gli oggetti tridimensionali ad esclusione della sfera su cui si applicherà la cubemap che si sta generando, poiché la sfera non deve riflettere anche se stessa ma solo l'ambiente circostante.

Per semplicità, in questa applicazione l'ambiente circostante la sfera è composto esclusivamente da una stanza (un cubo) sulle cui pareti vengono applicate delle texture in modo da simulare un ambiente tridimensionale.

Il codice completo del ciclo per eseguire i sei rendering sulle texture della cubemap è il seguente:

```
for (int i=0; i<6; i++) {
  glLoadIdentity();
  switch (faceTarget[i]) {
    case GL_TEXTURE_CUBE_MAP_POSITIVE_X:
      gluLookAt(X,      Y,      Z,
                X+5.0f, Y+0.0f, Z+0.0f,
                0.0,    -1.0,   0.0);
      break;

    case GL_TEXTURE_CUBE_MAP_NEGATIVE_X:
      gluLookAt(X,      Y,      Z,
                X-5.0f, Y+0.0f, Z+0.0f,
                0.0,    -1.0,   0.0);
      break;

    case GL_TEXTURE_CUBE_MAP_POSITIVE_Y:
      gluLookAt(X,      Y,      Z,
                X+0.0f, Y+5.0f, Z+0.0f,
                0.0,    0.0,    1.0);
      break;
```

```
    case GL_TEXTURE_CUBE_MAP_NEGATIVE_Y:
      gluLookAt(X,      Y,      Z,
                X+0.0f, Y-5.0f, Z+0.0f,
                0.0,    0.0,    -1.0);
      break;

    case GL_TEXTURE_CUBE_MAP_POSITIVE_Z:
      gluLookAt(X,      Y,      Z,
                X+0.0f, Y+0.0f, Z+5.0f,
                0.0,    -1.0,   0.0);
      break;

    case GL_TEXTURE_CUBE_MAP_NEGATIVE_Z:
      gluLookAt(X,      Y,      Z,
                X+0.0f, Y+0.0f, Z-5.0f,
                0.0,    -1.0,   0.0);
      break;
  }
  glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                            GL_COLOR_ATTACHMENT0_EXT,
                            faceTarget[i], cubemap, 0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  renderRoom();
}
```

A questo punto la cubemap è stata generata interamente. Rimane la creazione delle mipmap, la disattivazione del rendering su Framebuffer Object e il ripristino delle impostazioni del rendering su framebuffer:

```
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT, 0, 0, 0);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
glEnable(GL_TEXTURE_CUBE_MAP);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemap);
glGenerateMipmapEXT(GL_TEXTURE_CUBE_MAP);
glBindTexture(GL_TEXTURE_CUBE_MAP, 0);
```

```
glDisable(GL_TEXTURE_CUBE_MAP);
glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
glPopMatrix();
glPopAttrib();
```

## 3.3.6 Rendering della scena

Come già accennato, deve essere eseguito un rendering diverso per ogni viewport, dato che il punto di vista è differente.

Per quanto riguarda l'osservatore vengono memorizzate le coordinate della sua posizione e la direzione verso cui è rivolto. L'osservatore può "muoversi" liberamente all'interno della scena attraverso l'uso della tastiera, variando così posizione e direzione del campo visivo. Queste informazioni vengono utilizzate per la traslazione e rotazione del punto di vista della prima viewport.

Anche la sfera, che rappresenta un ipotetico sensore visivo omnidirezionale mobile, può essere spostato all'interno della scena tramite tastiera. In questo caso le informazioni sulla posizione della sfera sono utilizzate per la traslazione del punto di vista della seconda viewport, in cui la visuale rimane sempre fissa sulla sfera.

I due punti di vista vengono definiti semplicemente dalle istruzioni:

```
if (!staticcamera) {
  glRotatef(-yrot, 0.0f, 1.0f, 0.0f);
  glTranslatef(-xpos, -ypos, -zpos);
}
else {
  glRotatef(-90, 1.0f, 0.0f, 0.0f);
  glTranslatef(-X, -(Y-2.5), -Z);
}
```

A questo punto nel caso in cui la sfera si sia spostata o sia presente nella scena un oggetto in movimento, va richiamata la funzione per la generazione della cubemap, in modo da aggiornare la riflessione che deve apparire sulla superficie dello specchio. A questo scopo si utilizza un flag per indicare se è necessario o meno l'aggiornamento della cubemap.

L'ultima operazione prima del rendering delle geometrie consiste nell'orientare correttamente la cubemap in modo che la riflessione simulata sulla sfera tenga conto della posizione del punto di vista. In caso contrario, la riflessione rimarrebbe la medesima per qualsiasi posizione e angolazione del punto di vista. Come illustrato nel paragrafo 1.4, è necessario definire due vettori "direzione" e "orientamento" e calcolarne il prodotto vettoriale e l'arcoseno del prodotto scalare, per conoscere l'asse e l'angolo di rotazione da applicare alla cubemap. Il vettore "orientamento" è fisso e coincidente con l'asse `Z+`, mentre il vettore "direzione" è variabile e va dal centro della sfera al punto di vista. Una volta noti l'asse e l'angolo di rotazione viene applicata una rotazione alla *texture matrix*. Le istruzioni necessarie per orientare correttamente la cubemap sono le seguenti:

```
void orientCubemap(float x, float y, float z) {

  float orientation[3] = {0.0f, 0.0f, 1.0f};
  float dir[3];
  dir[0] = x;
  dir[1] = y;
  dir[2] = z;
  normalize(dir);
  float dot = dotProduct(orientation, dir);
  float angle = acos(dot) * 180 / PI;
  float cross[3];
  crossProduct(dir, orientation, cross);
  glMatrixMode(GL_TEXTURE);
  glRotatef(-angle, cross[0], cross[1], cross[2]);
  glMatrixMode(GL_MODELVIEW);
}
```

Infine si può eseguire il rendering completo della scena, comprensivo di tutti gli oggetti tridimensionali. In questo caso la scena è composta da una stanza (un cubo) al cui interno è posto il sensore omnidirezionale (una sfera).

Per renderizzare la sfera va attivato l'impiego del cube mapping sulla sua superficie e la generazione delle coordinate della cubemap:

```
glEnable(GL_TEXTURE_CUBE_MAP);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemap);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
glutSolidSphere(1.0, 100, 100);
glDisable(GL_TEXTURE_GEN_S);
glDisable(GL_TEXTURE_GEN_T);
glDisable(GL_TEXTURE_GEN_R);
glDisable(GL_TEXTURE_CUBE_MAP);
glBindTexture(GL_TEXTURE_CUBE_MAP, 0);
```

Il risultato del rendering finale è visibile nelle immagini proposte nelle prossime pagine.

NVIDIA

# Conclusioni

Questa tesi ha avuto come scopo la progettazione e realizzazione di un'applicazione grafica 3D in grado di simulare il comportamento di un sensore (telecamera) omnidirezionale composto da uno specchio sferico e posto all'interno di un ambiente tridimensionale virtuale.

L'applicazione è stata realizzata in linguaggio C e le funzionalità grafiche sono state implementate grazie alla libreria grafica OpenGL.

Per simulare il comportamento di uno specchio sferico che riflette l'ambiente circostante sulla sua superficie è stata impiegata una tecnica denominata *cube environment mapping*, grazie alla quale è possibile memorizzare l'aspetto dell'ambiente su sei texture separate che coprono globalmente uno spazio a 360°.

Inoltre per far sì che la riflessione fosse dinamica, ovvero riproducesse eventuali variazioni dovute allo spostamento dello specchio oppure oggetti in movimento nell'ambiente, è stato impiegato il Framebuffer Object grazie al quale è possibile effettuare i rendering su texture necessari ad aggiornare la cubemap.

Il rendering finale mostra come la superficie dello specchio sferico presenti una riflessione accurata e realistica dell'ambiente circostante.

L'applicazione realizzata si presta a ulteriori ottimizzazioni:

- In primo luogo è di fondamentale importanza inserire il sensore in un ambiente più complesso, composto da numerosi modelli tridimensionali e oggetti in movimento, in modo da determinare più meticolosamente la

realisticità della riflessione e valutare le prestazioni dell'applicazione in un ambiente dinamico.

- Dal punto di vista prestazionale è senz'altro conveniente sfruttare le potenzialità *multi-rendering* del Framebuffer Object, così da rigenerare la cubemap con un solo rendering su texture anziché sei. Per implementare questa funzionalità è necessaria la programmazione di *shader* in linguaggio GLSL.

Inoltre il sistema realizzato si presta allo sviluppo di algoritmi per la localizzazione, navigazione o pianificazione del moto all'interno di ambienti non strutturati. Per quest'ultimo aspetto è indispensabile una fase preventiva in cui si memorizza la riflessione visualizzata sulla superficie dello specchio in diverse zone dell'ambiente, che fungono da punti di riferimento. Successivamente, dal confronto tra le riflessioni memorizzate e la riflessione corrente si riuscirebbe a determinare un percorso da seguire per raggiungere una certa zona dell'ambiente.

# Bibliografia


[1] **Tan, Kar-Han, Hua, Hong and Ahuja, Narendra.** "Multiview Panoramic Cameras Using a Mirror Pyramid". [Online] 2002. http://vision.ai.uiuc.edu/publications/Tan-Omnivis02.pdf.

[2] **Kang, Sing Bing and Szeliski, Richard.** "3-D Scene Data Recovery Using Omnidirectional Multibaseline Stereo". [Online] 1997. http://research.microsoft.com/pubs/75576/Kang-IJCV97.pdf.

[3] **Swaminathan, Rahul and Nayar, Shree K.** "Nonmetric Calibration of Wide-Angle Lenses and Polycameras". [Online] 2000. http://www.cs.columbia.edu/CAVE/publications/pdfs/Swaminathan_PAMI00.pdf.

[4] **Gluckman, Joshua, Nayar, Shree K. and Thoresz, Keith J.** "Real-Time Omnidirectional and Panoramic Stereo". [Online] 1998. http://cis.poly.edu/~gluckman/papers/iuw98a.pdf.

[5] **Yi, Sooyeong and Ahuja, Narendra.** "An Omnidirectional Stereo Vision System Using a Single Camera". [Online] 2006. http://vision.ai.uiuc.edu/publications/sooyeong_camera.pdf.

[6] **McReynolds, Tom and Blythe, David.** *Advanced Graphics Programming Using OpenGL.* s.l. : Morgan Kaufmann, 2005.

[7] **Shreiner, Dave, et al.** *OpenGL Programming Guide.* 6th ed. s.l. : Addison-Wesley, 2007.

[8] **Benstead, Luke.** *Beginning OpenGL Game Programming.* 2nd ed. s.l. : Course Technology PTR, 2009.

[9] **Astle, David.** *More OpenGL Game Programming.* 2nd ed. s.l. : Course Technology PTR, 2005.

[10] **NVIDIA Corporation.** "OpenGL Cube Map Texturing". [Online] 1999. http://developer.nvidia.com/object/cube_map_ogl_tutorial.html.

[11] **Wright, Richard S., Lipchak, Benjamin and Haemel, Nicholas.** *OpenGL SuperBible.* 4th ed. s.l. : Addison-Wesley, 2007.

[12] **OpenGL Architecture Review Board.** "Texture Cube Map Specifications". [Online] 1999. http://www.opengl.org/registry/specs/ARB/texture_cube_map.txt.

[13] **Green, Simon.** "The OpenGL Framebuffer Object Extension". [Online] 2005. http://download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf.

[14] **Ahn, Song Ho.** "OpenGL Frame Buffer Object (FBO)". [Online] 2008. http://songho.ca/opengl/gl_fbo.html.

[15] **Jones, Rob.** "OpenGL FrameBuffer Object 201". [Online] 2006. http://www.gamedev.net/reference/programming/features/fbo2/.

[16] **Jones, Rob.** "OpenGL FrameBuffer Object 101". [Online] 2006. http://www.gamedev.net/reference/programming/features/fbo1/.

[17] **Michel, Olivier.** "Webots: Professional Mobile Robot Simulation". [Online] 2004. http://www.cyberbotics.com/publications/ars.pdf.

[18] **Schmits, Tijn and Visser, Arnoud.** "An Omnidirectional Camera for the USARSim Environment". [Online] 2008. http://staff.science.uva.nl/~arnoud/publications/main_RP.pdf.

[19] **Panagiotis, Palantas, et al.** "Monocular Omnidirectional Vision Simulator for Robot Navigation". [Online] 2008. http://www.fh-kiel.de/fileadmin/data/IuE/forschung/Konferenzen/AMIES_2008/George_Papadourakis_2_AMIES_Kiel_2008.pdf.

[20] **Burbridge, Christopher and Spacek, Libor.** "Instantaneous robot self-localization and motion estimation with omnidirectional vision". [Online] 2007. http://www.infm.ulst.ac.uk/~chrisb/spacek-burbridge07.pdf.

[21] **Burbridge, Christopher and Spacek, Libor.** "Omnidirectional Vision Simulation and Robot Localisation". [Online] 2006. http://www.infm.ulst.ac.uk/~chrisb/burbridge-spacek06.pdf.

[22] **OpenGL Architecture Review Board.** "Framebuffer Object Specifications". [Online] 2008. http://www.opengl.org/registry/specs/ARB/framebuffer_object.txt.

# Ringraziamenti

Innanzitutto desidero ringraziare di cuore la mia famiglia per avermi costantemente sostenuto e incoraggiato, non solo nello svolgimento di questa tesi ma durante tutto il (lungo e accidentato) percorso di studi.

Rivolgo un sentito ringraziamento anche a tutti coloro che non sono stati semplici compagni di studio, ma si sono rivelati ottimi amici con cui condividere piacevolmente una serata, una bevuta, una vacanza, una risata. Ben più di una, a essere sinceri...