CAPITOLO 1a ~ Applicazioni della Computer Grafica

Che cos'è la Computer Grafica? Si tratta della disciplina che studia i metodi per realizzare immagini di sintesi *(singole o in successione per creare animazioni*), impiegando opportuni algoritmi, strutture di dati innovative ed hardware dedicato.

Nascita della Computer Grafica. La computer grafica è nata subito dopo la nascita dei computer, per la necessità di definire i formalismi con cui rappresentare, in modo visuale, i dati numerici generati dal computer. Nei primi stadi di evoluzione, per motivi tecnologici, le informazioni si rappresentavano solamente su **supporti statici** (come la carta disegnata con il plotter), solo in seguito ha prevalso largamente la rappresentazione su **supporti dinamici** (come schermi televisivi). Oggi la computer grafica è praticamente solamente interattiva, deve riuscire a modellare un contesto entro il quale l'utente ha un controllo continuo sul sistema, l'aspetto e la struttura degli oggetti e sulla maniera con cui vengono rappresentati.

Utilizzabilità della Computer Grafica. Fino agli anni '80 l'utilizzo degli strumenti di computer grafica era limitato ad un numero ristretto di studiosi a causa dell'elevato costo delle attrezzature. In seguito l'introduzione di computer con **grafica raster** ha reso accessibile la computer grafica ad un vasto numero di utenti ad un costo più contenuto.

Applicazioni odierne della Computer Grafica. Esistono diversi ambiti entro i quali è possibile e produttivo sfruttare le nozioni della computer grafica:

- ✓ **Computer Aided Desgin (CAD/CAM).** La computer grafica aiuta nella progettazione e nel disegno assistito di componenti per la meccanica, elettrici ed elettronici. Durante la progettazione è possibile interagire col modello per testarne le caratteristiche prima ancora di realizzarlo fisicamente.
- ✓ **Realtà Virtuale.** Per **realtà virtuale** si intende l'insieme di tecniche e dispositivi che permettono di ricreare un ambiente così simile alla realtà che l'utente immergendosi in esso non riesca a distinguere tale ambiente dalla realtà interagendo normalmente con esso. La **realtà aumentata** invece è un sistema entro il quale si sovrappone alla scena reale tramite dispositivo una scena ricreata artificialmente. SI tende a portare l'utente a convincersi della verosimiliarità della nuova realtà tramite utilizzatori che l'utente indossa per interagire con la realtà virtuale.
- ✓ **Visualizzazione scientifica.** Disciplina che si occupa della rappresentazione visiva dei fenomeni naturali modellati matematicamente e simulati numericamente.
- ✓ **Medical Imaging.** Tramite tecniche grafiche si vogliono rappresentare i dati planari (2D) o volumetrici (3D) acquisiti tramite dispositivi non invasivi. Si vuole aiutare il medico nella diagnosi e nella cura della malattia.
- ✓ **Beni culturali.** Si utilizza la grafica 3D per ricostruire modelli di edifici e gallerie virtuali accessibili ovunque per agevolare la fruizione e la conservazione delle opere d'arte.
- ✓ **Computer Art.** Tramite le tavolette grafiche gli artisti possono "dipingere" su computer sfruttando tutto ciò che la computer grafica mette loro a disposizione.
- ✓ **Industria dell'intrattenimento.** Il cinema ed i videogiochi sfruttano la più ampia gamma di risorse di computer grafica e di finanziamenti ad essa destinata.
- ✓ **Interfacce utente.** Essenziali per aiutare l'utente a familiarizzare con il sistema informatico.

CAPITOLO 1b ~ Modelli e sistemi grafici

Computer Grafica. Essa si occupa di tutti gli aspetti riguardanti la generazione di una immagine mediante l'uso di un personal computer. Un **sistema grafico di base** è realizzato da un gruppo di dispositivi di input (tastiera, mouse, tavoletta grafica, ecc), una unità di computazione e memoria (memoria e CPU), un buffer di memoria per l'immagine generata (Frame Buffer) ed uno o più dispositivi di output (schermo o stampante).

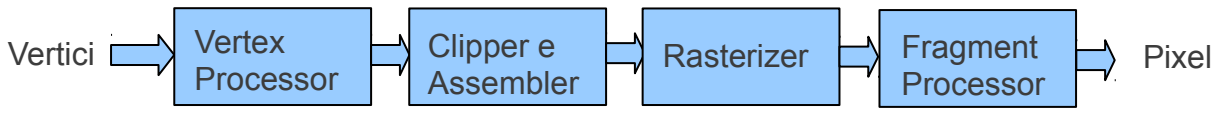
Storia della Computer Grafica. Le fasi della sua evoluzione attraverso gli anni.

- ✓ **1950/1960.** La grafica computerizzata risale alla nascita dei primi dispositivi di calcolo automatico. I computer all'epoca erano lenti e costosi e la rappresentazione grafica era possibile tramite un convertitore A/D collegato ad un monito CRT di tipo vettoriale o calligraphic.
- ✓ **1960/1970.** Si parla di:
  - ✘ *Wireframe Graphics*: si disegnano solo le linee della figura;
  - ✘ *Sketchpad*: è stato sviluppato da una idea di **Ivan Sutherland** durante il suo dottorato al MIT, riconosce le potenzialità dell'interazione tra uomo e macchina. Funziona sull' *idea del loop*: si disegna qualcosa, l'utente sposta la penna ottica ed il computer disegna una nuova immagine. Sutherland è famoso per aver sviluppato moltissimi algoritmi della computer grafica;
  - ✘ *Display Processor*: è un processore dedicato che aggiorna il display al posto del computer. Tutto quello che concerne la grafica è memorizzato in una d*isplay list* memorizzata a sua volta nel display processor, così la macchina ospite compila la lista ed invia tutto al DPU;
  - ✘ *Introduzione dei tubi catodici*.
- ✓ **1970/1980.** Si parla di:
  - ✘ *Grafica Raster*: l'immagine è prodotta da singoli elementi (pixel) secondo uno schema matriciale (raster) all'interno di un frame buffer. La grafica di questo tipo consente il passaggio dal wireframe al disegno "pieno". Nel **frame buffer**, una certa area della memoria, sono memorizzati i pixel, la quantità di memoria dedicata al frame buffer dipende dalla *profondità* (numero di bit per pixel, determina il numero di colori che possiamo utilizzare) e dalla *risoluzione* dell'immagine. I sistemi **True Color RGB** hanno una profondità di 24 bit (8 per il rosso, 8 per il blu ed 8 per il verde). L'**aspect ratio** indica il rapporto tra l'altezza e la larghezza di una immagine.
  - ✘ *Vengono proposti i primi standard grafici*;
  - ✘ *Prime workstation e primi personal computer:* al giorno d'oggi non c'è molta differenza, ma all'inizio le workstation erano caratterizzate da collegamenti di rete client-server e da un elevato grado di interattività. Nei PC parte della memoria era dedicata al frame buffer e si potevano facilmente modificare le immagini modificando il contenuto del frame.
- ✓ **1980/1990.** La computer grafica diventa più realistica.
  - ✘ *Smooth Shading*: la superficie dell'immagine viene come "lisciata" e spariscono i poligoni spigolosi.
  - ✘ *Environment Mapping*: sulla superficie degli oggetti è possibile "specchiare" l'ambiente che li circonda con molto realismo.
  - ✘ *Bump Mapping*: è possibile ricreare un effetto a rilievo sulle superfici degli oggetti.
- ✓ **1990/2000.** Si parla di:
  - ✘ *OpenGl*;
  - ✘ Hardware decisamente più evoluto;
  - ✘ Produzione dei primi film completamente in computer grafica.
- ✓ **2000/oggi.** Si parla di:
  - ✘ nasce il concetto di *fotorealismo*: si producono immagini praticamente indistinguibili dalla realtà, il termine sta ad indicare quanto una immagine assomigli alla realtà;
  - ✘ nasce una vera e propria industria basata sulla computer grafica, attraverso lo sviluppo di videogiochi accattivanti e di film sempre più sbalorditivi;
  - ✘ Pipeline grafiche programmabili e schede grafiche dotate di *Graphic Unit Processor*, microprocessore per le schede video di pc e console, è specializzata nell'eseguire elaborazioni 3D.

Formazione dell'immagine. La computer grafica genera immagini sintetiche utilizzando approcci ispirati alla realtà. Per poter generare una immagine partiamo dalla **scena** composta da oggetti, osservatore e sorgente di luce. All'interno della scena è necessario specificare anche forma, colore e tipo di luce, i parametri che definiscono il tipo di materiale e la posizione e la direzione

dell'osservatore. Bisogna ricordare che oggetti, osservatore e luce sono entità tra loro sempre indipendenti.
La **luce** è la porzione di spettro elettromagnetico visibile all'occhio umano. La natura della luce come onda elettromagnetica non viene mai presa in considerazione dalla computer grafica, bensì come modello dell'ottica geometrica. Il percorso di un raggio può essere complesso da seguire, ed è possibile creare un'immagine seguendo i raggi di luce dalla scena all'obbiettivo (ray tracing).
Una **pinhole camera** (camera oscura) è una piccola scatola scura e chiuse con un foro su un lato per far entrare la luce. La luce che entra si proietta sul lato di fondo della scatola riportando l'immagine al di fuori della scatola in maniera capovolta. Più il foro è stretto più l'immagine e nitida.
Il **field of view** (campo visivo) è l'angolo formato dal più grande oggetto visibile sul fondo della scatola nera.
La **profondità di campo** è la distanza davanti e dietro all'immagine che proiettata sul fondo della scatola appare più a fuoco delle altre.
La **clipping window** è la porzione di immagine che riusciamo a vedere (o la finestra attraverso la quale riusciamo a vedere la scena).

Pipeline Grafica. Gestiamo un oggetto alla volta così come sono stati introdotti nella scena. La pipeline tiene conto solo degli effetti di illuminazione locale. Tutti i passi della pipeline possono essere implementati in maniera grafica. Una Pipeline Grafica realizza il processo di rendering della scena a partire da un insieme di oggetti (definiti da vertici).



Vertex Processor. Siccome gli oggetti sono definiti da vertici, questo modulo si occupa di convertire gli oggetti da un sistema di coordinate ad un altro. Ogni cambio di coordinate è equivalente ad una matrice di trasformazione. Il vertex processor si occupa anche di calcolare il colore dei vertici.
Projection. È un processo che si occupa di utilizzare i parametri dell'osservatore nello spazio per produrre una immagine 2D da oggetti 3D.
Primitive Assembler. I vertici processati sono considerati come parte di oggetti prima che la fase di clipping e raterizzazione possa avvenire.
Clipping. Come la videocamera non può vedere tutto, anche la finestra della camera virtuale deve escludere quelle parti di scena non visibili. Tramite il processo di clipping tutti gli oggetti non presenti nel campo visivo devono essere tagliati fuori dalla scena.
Rasterizzazione. È necessario determinare i colori appropriati dei pixel dell'immagine. Questo modulo produce un insieme di frammenti per ogni pixel, tali frammenti sono dei "potenziali pixel". Il modulo interpola le informazioni di colore contenute nei vertici.
Fragment processing. I frammenti sono processati all'interno del buffer per determinare il colore finale del pixel.

Interfaccia. Il programmatore deve il sistema grafico attraverso una interfaccia software (API). Le API mettono a disposizione del programmatore gli strumenti per la creazione e l'assemblaggio delle scene per produrre l'immagine finale (oltre che a fornire il sistema per il quale è possibile comunicare attraverso input da tastiera).

Primitive di disegno. Un API solitamente supporta un *numero limitato di primitive di disegno*. Ogni primitiva è definita attraverso locazioni nello spazio o vertici.

Rendering. È detto **Rendering** il processo di generazione di una immagine a partire da una descrizione della scena. La complessità della scena ed il tipo di rendering influenzano la quantità di tempo necessaria per produrre l'immagine finita.

CAPITOLO 2 ~ Introduzione a OpenGL

IRIS GL. Agli inizi degli anni '90 la Silicon Graphics era leader indiscusso della progettazione di architetture per lo sviluppo della grafica 3D (IRIS GL era la libreria grafica fornita con le sue workstation). IRIS GL era una buona libreria ma fortemente dipendente dall'hardware e finì con soddisfare più le esigenze visto la nascita sempre più frequente di architetture ad-hoc con hardware e librerie specifiche per ogni produttore. Allora la Silicon Graphics riuscì a mettere tutti d'accordo, nel 1992 nasce **OpenGL**, standard unico aperto a tutti.

OpenGL come macchina a stati. Si definisce **macchina a stati** un modello di funzionamento di un sistema composto da stati (che memorizzano i comportamenti della macchina) ed azioni (atti che la macchina può compiere in una determinata circostanza). OpenGL è paragonabile ad una macchina a stati perchè le sue funzioni sono di due tipi: generazione di primitive grafiche e modifica dello stato.

Pipeline di OpenGL. I dati di input vengono maneggiati a seconda dei valori dei parametri forniti ad OpenGL attraverso la pipeline. La rendering pipeline non è unica perchè ogni produttore ha libertà di implementarla come vuole guardando a costi ed a consumi. E' una pipeline smemorata: la rendering pipeline non ricorda mai il rendering precedente. La filosofia di OpenGL è di mostrate i dati di input e non di processarli, a quello ci pensa la CPU mentre alla pipeline spetta il compito di migliorare le prestazioni presentazionali.

Vertici e Punti. In OpenGL un oggetto 3D è descritto interamente tramite i suoi vertici e punti. La gestione dei vertici è affidata al programmatore che predispone una opportuna struttura di dati per gestirli. Il comando **glVertex*()** fornisce in input un insieme di vertici od un vettore di vertici (vedi appendice A). OpenGL tratta diversi tipi di dati in input, ogni gruppo di dati incomincia sempre con **glBegin()** e termina con **glEnd()**.

Struttura di una applicazione in OpenGL. Le applicazioni OpenGL hanno una struttura molto simile alle altre applicazioni che consiste di:

- ✓ **main()** che apre la finestra di rendering, definisce le funzioni di callback ed entra nel ciclo di rendering;
- ✓ **init()** che imposta tutte le variabili di stato (come i parametri dell'osservatore e gli attributi del rendering);
- ✓ **callbacks** che sono funzioni invocate in risposta agli eventi, come la funzione per il rendering e le funzioni per gestire l'imput dalla finestra.

Modello RGB. OpenGL utilizza il modello RGB per le informazioni di colore. Attraverso il rosso(R), il verde(G) e il blu(B) è possibile ottenere tutte le tinte compresa la luce bianca mediante *sintesi additiva*.

Shading. Lo **Shading** indica il processo di attribuzione del colore ad un poligono. OpenGL gestisce due tipi di shading: *Smooth Shading* che riempie i poligoni interpolando i colori dei vertici per determinare il colore dei pixel interni ed il *Flat Shading* che invece colora l'interno del poligono utilizzando il colore del primo vertice.

Viewing. Indipendentemente dalla posizione degli oggetti nella scena una delle scelte fondamentali da fare è l'impostazione della camera. Di default si utilizza la posizione ortografica, dove si assume che la camera sia posizionata ad una distanza infinita dalla scena.

CAPITOLO 3 ~ Programmazione Grafica

Sistemi di coordinate. In computer grafica siamo in grado di specificare qualunque sistema di

coordinate (device indipendent graphics). Fino ad ora abbiamo parlato di **world coordinates** (coordinate mondiali) anche espresse in float utilizzate per i vertici e di **screen coordinates** (coordinate dello schermo) solo intere utilizzate per i pixel.

Tracciare le linee. Le **linee** sono le più comuni primitive in grafica 2D e spesso anche la grafica 3D wireframe ha bisogno di utilizzare linee 2D. Il disegno della linea richiede un processo di discretizzazione dove si calcolano le coordinate intere dei pixel che sono sulla linea o nelle sue vicinanze. L'algoritmo di discretizzazione deve essere molto veloce ed efficiente perchè verrà invocato moltissime volte durante il processo di creazione ed aggiornamento dell'immagine, inoltre è suo compito creare linee soddisfacenti dal punto di vista visivo (le linee devono essere rette, terminare nettamente ed essere sempre dense uguale in ogni loro punto). Alcuni algoritmi noti di discretizzazione sono:

- ✓ **Algoritmo DDA.** Considera la risoluzione di una semplice equazione differenziale. È efficiente ma richiede addizioni in virgola mobile per ogni pixel.
- ✓ **Algoritmo di Bresenham.** Si basa sull'aritmetica intera. Si introduce una variabile decisionale che determina qualche pixel debba accendersi.

Il problema dell'aliasing. La discretizzazione può causare problemi di "seghettatura". Tale problema è detto **aliasing** e può essere risolto utilizzando più pixel per definire meglio la forma della figura, se non è possibile si può usare l'intensità del pixel come variabile per controllare l'aliasing, introducendo una sorta di pixel virtuale. Una linea antialiased ha una serie di pixel virtuali.

Sierpinski gasket. Il **triangolo di Sierpinski** è un frattale, un esempio base di un insieme auto-similiare generato da un pattern che si ripete allo stesso modo su scale diverse.

Volume della vista. Per definire il sistema di coordinate è necessario specificare un volume della vista dentro al quale costruire una scena, gli oggetti con coordinate esterne a questa vista verranno tagliati mediante clipping.

Matrix mode. OpenGL gestisce le informazioni concatenando diverse matrici insieme. Due modalità sono particolarmente utilizzate:

- ✓ **model-view**: per la gestione delle trasformazioni dei vertici;
- ✓ **projection**: per la gestione della camera.

La funzione glMatrixMode() permette di specificare la funzione desiderata.

Rimozione delle superfici nascoste. In grafica 3D è importante visualizzare le superfici rispettando la profondità dell'oggetto. OpenGL impiega una tecnica chiamata **hidden-surface** invocando lo z-buffer per decidere il corretto ordine dei pixel.

CAPITOLO 4 ~ Input ed interazione

X Window Input. Storicamente OpenGL nasce su piattaforme Unix con supporto per le reti. **X Window System** *è un modello client-server per architetture su rete*, dove il client era una applicazione OpenGL e ed il server era un server grafico adibito al rendering con dispositivi di puntamento e tastiera. Con il termine **indirect rendering** si indica una modalità in cui il rendering è eseguito con il supporto della rete: su una macchina locale si esegue un server che si occupa della raffigurazione in OpenGL e su una macchina remota di esegue l'applicazione responsabile della generazione e gestione dei dati da visualizzare. Sulla macchina locale il server cattura gli eventi e li invia al gestore dell'applicazione che in base alla propria logica genera i dati da visualizzare in OpenGL, allora viene generato un messaggio mandato al server attraverso una rete che lo smista.
Lo svantaggio di questa architettura è che per ogni frame i dati vanno mandati attraverso la rete con performance limitate dalle caratteristiche della rete, inoltre applicazioni che mantengono un grosso insieme di dati da visualizzare ad ogni frame sono svantaggiate.
Di default OpenGL utilizza la modalità **immediate mode** per il rendering così da non memorizzare i

dati (per il rendering di un nuovo frame si vedono rimandare i dati) con una certa lentezza se le scene da visualizzare sono complesse. In alternativa c'è la modalità **retained mode** per definire gli oggetti e mantenerli già pronti sulla memoria della scheda grafica.

Display List. Una display list *è un meccanismo per salvare una serie di comandi OpenGL all'interno della memoria della scheda grafica o su un server*, in essa si possono inserire sia dati per il cambiamento della scena sia per la sua definizione. Una display list può essere eseguita ogni istante evitando di trasferire i dati (come nel *immediate mode*). Le display list sono gerarchiche quindi posso chiamare una display list all'interno di un'altra display list. Hanno vantaggi e svantaggi:

- ✓ **Vantaggi.**
  - ✘ Permettono ad OpenGL di convertire i comandi in un formato più coneniente;
  - ✘ I produttori hardware solitamente supportano questi meccanismi per migliorare le prestazioni delle proprie schede grafiche.
- ✓ **Svantaggi.**
  - ✘ Sono strutture statiche;
  - ✘ Ridondanza di dati in memoria;
  - ✘ Scarsi risultati se la memoria è limitata;
  - ✘ Non tutti gli scenari possono trarne beneficio.

Modalità di input. I dispositivi di input contengono dei trigger utilizzati per inviare un segnale al sistema operativo. Quando avviene il trigger il dispositivo restituisce un segnale al sistema operativo. Le modalità di input sono **request mode** e **event mode**.

- ✓ **Request mode.** Il valore letto sul dispositivo non viene restituito fino a quando non scatta il trigger.
- ✓ **Event mode.** Molti sistemi possiedono diverse periferiche di input ognuna delle quali capace di generare un trigger. Per ogni trigger il dispositivo fornisce un numero identificativo ed un valore. I valori sono inseriti in una coda che sarà elaborata dall'applicazione ma l'applicazione è indipendente dagli elementi inseriti nella coda.

Callbacks. Rappresentano le interfacce per la programmazione *event driven*. Una funzione di callback è definita per ogni tipo di eventi che il sistema grafico riconosce, è l'utente che definisce qualche azione intraprendere nel caso del verificarsi di un dato evento.

Animazione. Possibile se all'interno della funzione callback display prima di effettuare il rendering puliamo il frame buffer con un glClear(). Il rendering del frame buffer è una operazione indipendente dalla visualizzazione del contenuto. Nel caso di animazioni potremmo vedere il refresh del contenuto.

Double Buffering. Invece di utilizzare un solo buffer RGB ne utilizziamo due (*double buffer*), Front Buffer adoperato solo per la visualizzazione e Back Buffer adoperato solo per il rendering. Al termine di ogni rendering i due buffer sono scambiati e la procedura si ripete.

CAPITOLO 5 ~ Sistemi di coordinate e frame

Elementi di base. La geometria è lo studio delle relazioni che intercorrono tra gli oggetti in uno spazio N-dimensionale. Nella grafica computerizzata siamo interessati ad oggetti che esistono in uno spazio tridimensionale. Gli oggetti di base ai quali siamo interessati solo:

- ✓ **Scalari.** Gli scalari sono definiti come i membri di insiemi che possono essere combinati tramite *addizione* e *moltiplicazione*. Obbediscono agli assiomi dell'*associatività*, *commutatività* ed *esistenza dell'inverso* (es: i numeri reali e complessi). Gli scalari hanno proprietà geometriche.
- ✓ **Vettori.** Dal punto di vista fisico, un vettore è una grandezza dotata di *modulo*, *direzione* e *verso* (es: forza, velocità). Un vettore è detto *libero* se esiste indipendentemente dal sistema di riferimento. Le operazioni definite tra i vettori comprendono: l'*inversione di un vettore*,

moltiplicazione di un vettore per uno scalare, la *somma di due vettori*.

- ✘ Spazio vettoriale. È una struttura algebrica formata dall'insieme dei vettori e dalle operazioni di *somma di vettori* e di *moltiplicazione per un numero reale*.
- ✘ Spazio affine. È una struttura collegata al concetto di spazio vettoriale che lega assieme punti e vettori. Intuitivamente è uno spazio vettoriale privo di un punto centrale privilegiato rispetto ad altri.

✓ **Punti.** Rappresenta una posizione nello spazio. Sono permesse operazioni tra punti e vettori: la *sottrazione tra due punti* fornisce un vettore e la *somma di un punto ed un vettore* fornisce un punto.

- ✘ Retta. Nello spazio euclideo la retta rappresenta un insieme di punti definita da una funzione matematica.

È a partire da questi oggetti che possiamo creare oggetti più complessi.

Sistemi di coordinate e frame. Nella grafica computerizzata siamo interessati allo spazio tridimensionale dove punti e vettori hanno un ruolo fondamentale. Dobbiamo rappresentarli entrambi. Si dice **indipendenza lineare** se in un insieme di vettori non è possibile rappresentarne uno in funzione degli altri. In uno spazio vettoriale il numero massimo di vettori linearmente indipendenti è fissato e definisce la **dimensione** dello spazio.

CAPITOLO 6 ~ Trasformazioni

Trasformazioni. La **trasformazione** è una operazione (o funzione) che permette di calcolare un nuovo punto o un nuovo vettore a partire da un punto o un vettore dati. Ne esistono tante diverse. Tra le tante ne scegliamo un gruppo con un certo numero di proprietà utili, nello specifico quelle che mantengono i segmenti di segmenti senza modificare quelli di partenza (sono dette anche *trasformazioni affini*); le trasformazioni si dicono *trasformazioni lineari* definite per *additività* e *omogeneità*. Le trasformazioni affini includono: rotazione, traslazione, scala e shear.

- ✓ **Traslazione.** La *traslazione* è una operazione che sposta un punto P(x,y) in una nuova posizione P'(x',y'). per definire questa operazione è necessario utilizzare un vettore che descriva lo spostamento.
- ✓ **Rotazione.** La *rotazione* è l'operazione che ruota un punto P(x,y) di un angolo dato attorno all'origine per ottenere un nuovo punto P'(x',y'). E' semplice descrivere la rotazione mediante la notazione delle *coordinate polari* che esprimono la posizione del punto attraverso la distanza del punto dall'origine degli assi e l'angolo che il segmento della distanza forma con l'asse x. Il verso positivo della rotazione è quello antiorario.
- ✓ **Scalatura.** Operazione che moltiplica un punto P(x,y) per due valori scalari per ottenere un nuovo punto. Se i due scalari sono uguali tra loro la scala è *uniforme*, altrimenti è definita *non-uniforme*.
- ✓ **Shear.** Consiste nel traslare i vertici della base ed in cima ed alla base di un oggetto in direzioni opposte.

Matrici di trasformazione. Una **matrice di trasformazione** trasforma un vettore (o un punto) in un altro, tale trasformazione è coerente con le *coordinate omogenee* degli oggetti. La matrice di riflessione è un caso particolare della matrice di scala.

Trasformazioni a corpo rigido. Le *trasformazioni a corpo rigido* sono costituite solo da rotazioni e traslazioni. Preservano angoli e lunghezze. Determinare una matrice di trasformazione a corpo rigido vuol dire comporre le matrici di traslazione di rotazione e di traslazione. Eseguire prima la rotazione e poi la traslazione è diverso che fare il contrario, quindi l'operazione non è commutabile.

Concatenazioni di trasformazioni. In genere si può esprimere una serie di operazioni di trasformazione con una *concatenazione di trasformazioni*, così da ottenere una modifica dell'oggetto complessiva (invece di calcolarne una alla volta). È possibile calcolare la matrice complessiva delle trasformazioni ed utilizzarla quando è opportuno.

Trasformazioni in OpenGL. Ogni trasformazione in OpenGL è definita da una matrice 4x4, il sistema gestisce fondamentalmente tre tipi di trasformazione: Modelview, Projection e Texture. Con il comando **glMatrixMode(GLenum *mode*)** OpenGL cambia il suo stato relativamente al tipo di trasformazione da definire, dove *mode* si può scegliere tra le tre modalità sopra specificate. Tutte le matrici definite dopo il comando glMatrixMode() saranno moltiplicate alla matrice corrente. Per annullare la matrice corrente si usa **glLoadIdentity()**, tramite una matrice identità si sovrascrive la matrice corrente. In ogni caso OpenGL mantiene sempre una **Current Trasformation Matrix** (CTM), con l'operazione di caricamento della matrice identità azzero la CTM; ogni matrice in input moltiplica la CTM con quella appena caricata e la CTM è applicata ad ogni vertice in input.

Trasformazioni ModelView. Le tre fondamentali matrici di trasformazione in OpenGL sono definite dai comandi glTranslate(), glRotate(), glScale(). In base ai parametri in ingresso OpenGL determina la matrice di trasformazione moltiplicandola con la CTM.

- ✓ **glTranslate[fd](type x, type y, type z)** genera una matrice di traslazione. Allora la CTM moltiplica una matrice di traslazione: CTMxT.
- ✓ **glRotate[fd](type angle, type x, type y, type z)** genera una matrice di rotazione dove *angle* indica l'angolo di rotazione in gradi.
- ✓ **glScale[fd](type x, type y, type z)** genera una matrice di scalatura.

Le operazioni di traslazione, rotazione e scalatura sono le più frequenti nella computer grafica ma non sono le uniche. Il comando **glMultMatrix(const type *m)** permette di definire una matrice diversa da quelle standard e di moltiplicarla per la CTM. OpenGL accetta matrici di float definite idealmente in un vettore m[16] da 0 a 15 quindi di 16 locazioni editabili manualmente.

CAPITOLO 7 ~ Camera e proiezioni

Push/Pop Matrix. Permettono di salvare la matrice corrente di trasformazione sullo stack, per poi moltiplicarla e maneggiarla. È possibile poi tornare sui propri passi estraendo la matrice di partenza dallo stack. **PushMatrix()** salva una copia della matrice corrente sullo stack (in testa od in cima allo stack), con **PopMatrix()** invece estraggo dallo stack la matrice salvata in precedenza (comunque quella che si trova in testa allo stack). *Non è necessario salvare tutte le trasformazioni, solo quelle che potrebbero servire successivamente*. Evito così di calcolare sempre l'inversa della matrice e non ho nessun problema di approssimazione nelle operazioni floating point.

Il Sistema Solare ed il Braccio Meccanico. Per prima cosa prendiamo per esempio la realizzazione di un modello planetario del Sistema Solare eliocentrico. Per disegnare la sfere utilizziamo una chiamata GLUT preesistente, la **glutWireSphere()**, poi per far ruotare le sfere utilizziamo glRotate*(). Per traslare le sfere uso **glTranslate*()**.
Pensiamo ora ad un modello di Braccio Meccanico. Il principio è identico a quello del modello del sistema planetaro. Abbiamo bisogno di due primitive per costruire il braccio.

Elementi base della visualizzazione. Gli elementi base della visualizzazione sono gli oggetti, l'osservatore, i raggi di proiezione incidenti verso un piano di proiezione che convergono verso il centro di proiezione, detta **COP**, che corrisponde alla posizione della lente della camera o all'occhio dell'osservatore. Sia nella visualizzazione classica che in quella della grafica computerizzata il COP può essere posizionato ovunque (nella visualizzazione classica ci sono vincoli fisici che tuttavia non esistono nella computer grafica per ovvi motivi): si parla allora di **vista prospettica** con COP in posizione finita e di **vista parallela** con COP in posizione infinita (dove di raggi di proiezioni, che qui chiamiamo *direzioni di proiezione*, sono virtualmente paralleli tra loro ma non per forza ortogonali al piano di proiezione). Non si conservano punti e distanze, che dipendono dalla prospettiva.

Proiezione planare geometrica. I raggi di proiezione sono linee che convergono nel centro di proiezione, la proiezione avviene su un piano e la proiezione preserva le linee ma mai gli angoli. Si tratta di un caso particolare della proiezione prospettica ma le API della computer grafica la tratta come un caso separato. Può essere:

- ✓ **Parallela.**
  - ✘ **Ortografica.** I raggi della proiezione sono perpendicolari al piano della proiezione. Il piano è parallelo ad una faccia dell'oggetto. Le forme sono preservate e può essere utilizzata per fare delle misurazioni, tuttavia non si ha una impressione realistica dell'oggetto.
  - ✘ **Assonometrica.** Possiamo vedere contemporaneamente più facce dello stesso oggetto. Il piano di proiezione ha una posizione ed un orientamento arbitrari ed i raggi di proiezione sono ancora perpendicolari al piano di proiezione.
  - ✘ **Obliqua.**
  - ✘ **Prospettica.** La dimensione dell'oggetto è proporzionale rispetto alla posizione degli oggetti.
- ✓ **Simmetrica.**

Il **punto di fuga** rappresenta in punto dove tutte le linee parallele convergono.

CAPITOLO 8 ~ Illuminazione

Trasformazioni di vista. Esistono passaggi tra la definizione degli oggetti nello spazio oggetti e la "trasformazione" di tali oggetti nello spazio mondo. Esistono comandi che ci permettono di posizionare la camera nella scena per modificare il campo di vista che vogliamo osservare. Si può passare da un sistema di riferimento all'altro, mediante matrici di trasformazione.
È detta **viewport transformation** la trasformazione che proietta ortograficamente il "canonical view volume" in un immagine sullo schermo di n_xxn_y. La **proiezione ortografica** mappa il volume che si vuole usare come volume di osservazione, l'idea è la stessa della *viewport transformation*. La **trasformazione della vista** comporta la traslazione del riferimento nello spazio. Nell'ordine, si esegue la trasformazione per inserire il modello nello spazio, la trasformazione della camera e la trasformazione della traslazione nello spazio.
La **perspective transform** richiede una operazione per la quale la proiezione del punto dipende dalla distanza dell'osservatore, bisogna ricorrere ad una elaborazione delle coordinate canoniche che permette di ottenere alla fine un termine adatto per attuare la divisione prospettica: trasformo il volume prospettico (tipicamente a forma di tronco di piramide) in un parallelepipedo i cui vertici saranno dati dai piani vicino e lontano, alto e basso, destra e sinistra. Tale trasformazione può essere convertita nel volume canonico.

Il modello di illuminazione. Il modello di illuminazione usato fin'ora in OpenGL era basato semplicemente su glColor3f(). Per dire che un oggetto possedeva un certo colore abbiamo assegnato all'oggetto lo stesso colore del pixel da cui parte la costruzione dell'oggetto. Ovviamente per ottenere effetti fotorealistici serva tener conto della superficie e del materiale di cui è fatto l'oggetto in esame. In questo modo dobbiamo modificare come la luce si rifrange sulla superficie caratterizzata dell'oggetto, tenendo conto della tipologia dell'illuminazione e del suo colore.
Nella grafica computerizzata un **modello di illuminazione** è utilizzato per modellare l'interazione tra una sorgente di luce e una superficie (in base al materiale esiste un corrispettivo ed adeguato modello di illuminazione). La fase di *shading* è il processo che determina il colore di un determinato pixel dell'immagine utilizzando un modello di illuminazione. Gli algoritmi di shading sono generalmente basati su algoritmi di riempimento e sull'interpolazione delle informazioni possedute per determinare quelle mancanti.

Interazione luce/materia. Per determinare il colore di un punto su si una superficie dobbiamo considerare l'interazione luce-materia. Sono da considerare le sorgenti di luce (tipo, colore, orientamento), le proprietà del materiale (tipo, colore, orientamento), la posizione dell'osservatore. Per i materiali è concesso usare il concetto di *normale alla superficie* per determinare l'orientamento di ogni punto dell'oggetto sulla superficie.

Simulare la luce. Simulare il comportamento della luce è un processo particolarmente complesso, infatti possono esistere molti tipi di interazione luce-materia. Per esempio, se un raggio di luce colpisce una superficie una parte viene assorbita e l'altra riflessa, magari su un altro piano che a sua

volta assorbe un po' di luce e ne riflette una parte sulla prima superficie. Il calcolo del comportamento della luce è affidato all'**equazione della radianza** (l'equazione lega ciò che vede l'osservatore con il modo di comportarsi del materiale e l'angolo di incidenza della luce). Tale equazione si semplifica introducendo una modernistica più semplice per affrettare il calcolo.
L'**equazione di rendering** descrive le infinite interazione luce-materia in termini di riflessione ed assorbimento della luce. Non si può risolvere analiticamente ed utilizzando tecniche numeriche è possibile trovare ottime approssimazioni. Tale equazione descrive anche l'illuminazione globale in che modo la luce interagisce con tutti gli oggetti della scena (in grado quindi di determinare ombre e interriflessioni multiple tra oggetti).

Illuminazione locale. In genere si cerca di individuare a partire dall'osservatore quali siano i possibili raggi che interagendo con l'oggetto possano colpire l'occhio dell'osservatore (o della camera). Si tracciano quindi i raggi sulle superfici che possono essere viste dall'osservatore, ed un base alle proprietà della scena, della luce e del materiale, si determina quale tipo di luce torna all'osservatore.
Una **superficie colorata** assorbe parte della luce visibile e restituisce il resto nell'ambiente sotto forma di *luce riflessa*. La luce che riceve l'osservatore è il prodotto di ogni componente RGB della luce sorgente per quella del materiale. Una **sorgente luminosa** può avere una forma qualunque ma modellare qualunque tipo di luce può diventare una operazione abbastanza complessa, teoricamente si dovrebbero considerare tutti i contributi provenienti da ogni punti disposto sulla superficie della luce. La luce ambiente è una luce che incide su ogni oggetto da tutte le direzioni. Le sorgenti di luce possono essere:

- ✓ **Puntiforme.** È caratterizzata da una posizione ed un colore. La luce è diffusa allo stesso modo in tutte le direzioni. Se la posizione è posta all'infinito otteniamo una luce direzionale. È necessario comunque definire un punto a partire dal quale esiste tale sorgente di luce. *Ogni tipo di luce si attenua man mano che ci si allontana dalla sorgente luminosa*. Un oggetto più vicino alla luce riceve una illuminazione più intensa rispetto ad un oggetto distante. Però questo tipo di attenuazione è troppo alta per gli oggetti vicini e troppo poco intensa per quelli lontani, dopotutto le sorgenti di luce reali non sono puntiformi. Una modifica a questo modello introduce l'*attenuazione quadratica*.
- ✓ **Spotlight.** È una sorgente di tipo faretto caratterizzata da una posizione, una direzione ed un angolo di apertura. Si può applicare a questo tipo di luce una *attenuazione angolare*.
- ✓ **D'ambiente.** Descrive una quantità di luce uniforme all'interno della scena, infatti non ha né una posizione né una direzione, ma tipicamente ha solo una intensità.

Tipi di superfici. Ne esistono molte. Ogni superficie diversa determina il modo in cui la luce si riflette su di essa.

- ✓ **Speculare.** Si comportano come gli specchi, riflettendo gran parte della luce in prossimità dell'angolo di riflessione.
- ✓ **Diffusa.** Riflettono la luce in tutte le possibile direzioni con uguale intensità.
- ✓ **Traslucida.** Trasmettono parte della luce all'interno del materiale attraverso il fenomeno della rifrazione. Questo tipo di superficie non esiste nel modello base che utilizziamo, anche se è imitabile ed ottenibile altrimenti.

Bidirectional Reflection Distribution Function. In natura non esistono superfici perfettamente speculari, diffusive o traslucide ma piuttosto combinazioni di queste. La funzione BRDF determina la quantità di luce in output in una data direzione considerando una quantità di luce in input in una data direzione. Nella computer grafica in tempo reale si tende ad usare varianti della BRDF semplificate per favorire la reattività delle applicazioni.

Modello di Phong. È un efficiente e fisicamente plausibile modello di simulazione luce/materia utile per determinare il colore di un punto sulla superficie. È implementato su tutte le schede grafiche: a partire da un punto si utilizzano quattro vettori di direzione della sorgente luminosa, direzione dello sguardo, normale alla superficie e vettore riflesso. Il modello di Phong combina linearmente tre tipi di interazione luce-materia: ambiente, diffusa e speculare. Ogni luce ha tre

componenti separate ed ogni materiale idem. Per ogni luce e ogni materiale le corrispondenti componenti interagiscono per realizzare il colore della scena finale.

Proprietà della luce e del materiale. Consideriamo la luce come composta da luce ambientale, *diffusa* e *speculare*. Ogni componente del materiale è combinata con le corrispondenti componenti della luce. Esiste anche un coefficiente di lucentezza che indica quanto una superficie è speculare.
Si considerano **superfici lambertiane** (tipo quella smerigliata) *che diffondono luce allo stesso modo in tutte le direzioni*. In natura sono superfici geometricamente irregolari e non esiste un angolo di riflessione privilegiato. La *legge di Lambert* descrive l'irraggiamento di una superficie irradiata da una sorgente puntiforme.
Molte superfici non sono né perfettamente diffuse né perfettamente speculari. I*n generale le superfici lisce hanno un punto di "lucentezza" a causa della riflessione della luce in direzione del vettore riflesso*. Se l'osservatore è posizionato nella stessa direzione del raggio riflesso si percepisce il classico "bagliore".

Metodi di Shading. Nella fase di shading si applica il modello di illuminazione per ottenere i colori dei pixel per tutte le posizioni sulla superficie. Si fa in due modi:

- ✓ attraverso dei metodi che calcolano le intensità solamente in alcuni punti della superficie ed ottengono le altre interpolando. È drasticamente il metodo più semplice ed immediato;
- ✓ con metodi che calcolano l'intensità ad ogni punto della superficie che sono anche computazionalmente più pesanti. Non è semplice calcolare le normali in ogni punto.

Quando si interpolano i colori si parla di Gouraud shading. Se si interpolano le normali si parla di Phone shading.

CAPITOLO 9 ~ Illuminazione in OpenGL

Modello di illuminazione di Phong. Per ogni sorgente luminosa e per ogni componente di colore, il modello di Phong (senza il termine dipendente dalla distanza), si scrive in una certa maniera. Per ciascuna componente di colore sommiamo i contributi di tutte le sorgenti. Il termine speculare richiede che sia calcolato il vettore della riflessione a partire da altri vettori per ogni vertice. **Blinn** suggerì di impiegare un'approssimazione coinvolgendo un vettore intermedio il cui calcolo e più efficiente. Il modello di Phong modificato dal suddetto vettore intermedio è detto modello di Blinn-Phong.

Abilitare la luce in OpenGL. Per abilitare il concetto luminoso in OpenGL è necessario definire normali per ogni superfici e per ciascun vertice di ogni oggetto. Serve poi definire la posizione di tutte le luci, definire il modello luminoso che indica il livello di luce ambientale globale e la posizione del punto di vista, definire le proprietà dei materiali ed abilitare lo shadin.
In OpenGL quando dobbiamo definire un oggetto dobbiamo definire le normali tramite una chiamata **glNormal*()** poiché la normale in OpenGL fa parte dello stato ed è modificabile. Verrà usata per calcolare le varie componenti dell'equazione di Blinn-Phong. Per calcoli corretti del coseno servono lunghezze unitarie delle normali. Ci sono trasformazioni spaziali che possono modificare la normale e può essere un problema se non si abilita la modificazione automatica della normale con *glEnable(GL_NORMALIZE)*.
Si possono avere normali diverse anche in uno stesso vertice, poiché ho una normale per ogni superficie che converge in quel dato vertice.
Il calcolo delle luci in OpenGL si abilita con *glEnable(GL_LIGHTING)*, una volta fatto questo si ignora automaticamente glColor() e si possono gestire fino a 8 sorgenti luminose. Posso scegliere i parametri del modello delle luci con **glLightModeli(parameter, GL_TRUE)**. La gestione delle luci è possibile con **glLight()**. Questo è il modello fisico ed è stato modificato il suo andamento quadratico aggiungendo un termine costante lineare. Per ciascuna sorgente luminosa possiamo impostare i valori RGBA per le componenti diffusa, speculare ed ambientale e la posizione.

Luce ambiente. La luce ambientale dipende dal colore delle sorgenti luminose. OpenGL permette

di definire un termine ambientale utilissima in fase di test.

Moving Light Sources. Le sorgenti luminose sono elementi geometrici la cui posizione è modificata dalla matrice model-view. In funzione della posizione o della direzione definita per una sorgente è possibile muovere la sorgente luminosa assieme all'oggetto, oppure tenere fisso uno e muovere l'altro, od ancora muovere entrambi in maniera indipendente.

Gestione dei materiali. I parametri dei materiali sono gestiti attraverso la funzione glMaterial().

Le normali nell'illuminazione. Le normali alle superfici giocano un ruolo fondamentale nel modello di illuminazione. Nel *flat shading* il colore della superficie è calcolare da un solo punto, nel *Gouraud shading* il colore della superficie è calcolato mediante interpolazione dei vertici, nel *Phong shading* il colore della superficie è calcolato per interpolazione delle normali ad ogni punto ma non è supportato in hardware da OpenGL.

- ✓ **Gouraud shading.** L'intensità della luce è calcolata per ogni vertice, e l'illuminazione interna è calcolata interpolando quella dei vertici. Se l'oggetto non è abbastanza tassellato si perde in parte o del tutto la componente speculare, ma è efficiente attraverso tecniche incrementali ed implementa l'illuminazione graduale.

CAPITOLO 10 ~ Clipping e Disegno di Linee

Approcci di un sistema grafico. In un sistema grafico l'applicazione fornisce in input un insieme di vertici e fornisce in output dei pixel all'interno del frame buffer. Il sistema grafico deve iterare attraverso un insieme di dati per ottenere l'immagine nel frame buffer.

- ✓ **Object-Oriented.** In questo approccio abbiamo un ciclo del tipo *for (each_obj) render obj*, quindi si tratta di un approccio tipico di una architettura grafica a pipeline. Oggi implementare questo processo hardware è relativamente semplice e poco costoso; lo svantaggio principale è che non gestisce gli effetti dell'illuminazione globale (si gestisce solo lo z-buffer e non si gestiscono gli effetti che coinvolgono più oggetti);
- ✓ **Image-Oriented.** In questo approccio abbiamo un ciclo del tipo *for (each_obj) assign_a_color (pixel)*, quindi per calcolare il colore di ogni pixel abbiamo bisogno di accedere ad una struttura dati che mantiene tutti gli oggetti. Tutti i dati devono essere sempre presenti in memoria, e possiamo gestire effetti di illuminazione globale perchè in ogni istante abbiamo tutta la geometria disponibile;

Task di una pipeline grafica. I blocchi fondamentali di una pipeline grafica sono rappresentabili mediante quattro *task*:

- ✓ **Modeling.** Fase in cui viene prodotta la geometria degli oggetti sotto forma di vertici e di relazioni tra vertici, in questa fase è possibile eliminare quella parte della geometria che non sarà visualizzata. Per eliminare la geometria specifica richiede una conoscenza precisa dei dati da manipolare;
- ✓ **Geometry Processing.** In questa fase si decide quali vertici verranno visualizzati e se ne determina il colore attraverso quattro fasi: proiezione (i vertici sono trasformati mediante la matrice di modelview), primitive assembly (i vertici sono raggruppati in oggetti), clipping (si determina quali oggetti cadono nel volume della vista) e shading (i vertici vengono colorati);
- ✓ **Rasterization.** In questa fase i vertici processati e colorati sono utilizzati per determinare i restanti pixel interni all'oggetto attraverso il processo di *rasterizzazione* o **scan conversion**. Nel caso di poligoni si determinano i pixel interni a partire dai vertici proiettati, nel caso di linee si determinano i pixel che congiungono i due vertici del piano;
- ✓ **Fragment Processing.** In questa fase si determina il colore da assegnare direttamente al frame buffer, si possono combinare diversi frammenti;

Clipping. Il clipping è un'operazione che serve ad individuare un nuovo volume di vista della finestra grafica, è il processo che se una primitiva o parte di una primitiva si trova all'interno del

volume di vista. Il clipping può applicarsi a linee, poligoni e primitive arbitrarie.

Clipping delle linee. Per quanto riguarda le **linee**, una linea può essere accettata, rifiutata o tagliata. Utilizzando un approccio di forza bruta dobbiamo calcolare l'intersezione di ogni linea con la finestra di clipping. Si tratta di un approccio non efficiente perchè non è altro che una divisione per intersezione. L'**algoritmo di Cohen-Sutherland** serve ad evitare di calcolare le intersezioni inutili poiché ogni intersezione richiede una divisione: la finestra viene estesa all'infinito per ottenere 9 aree distinte (nel caso 3D ho 27 regioni determinate da 6 bit), ad ogni regione viene assegnato un codice binario di 4 bit scelto con criterio. A seconda del valore di un dato bit capisco se la linea giace interamente o meno nella finestra di vista.
I *vantaggi* sono che per calcolare l'outcode sono necessarie solo operazioni di addizione e sottrazione, funziona bene se è necessario processare molte linee, l'algoritmo può essere facilmente esteso al 3D. Gli *svantaggi* sono che l'algoritmo va applicato ricorsivamente e una coppia di punti che esprime una linea verticale non può essere espressa in forma canonica.
Il **Liang-Barsky Clipping** considera una retta nella sua forma parametrica, allora è possibile distinguere i possibili casi osservando l'ordinamento dei valori per i quali la linea interseca le linee che definiscono la finestra di output. I *vantaggi* sono che è possibile accettare o rifiutare in modo semplice proprio come prima, è possibile estendere l'algoritmo al caso 3D e non dobbiamo usare l'algoritmo in modo ricorsivo.

Clipping dei poligoni. Per quanto riguarda i **poligoni**, è tutto più complesso. È necessario tener conto che non tutti i poligoni sono convessi, generare più di un poligono (a partire da un poligono concavo) da una operazione di clipping rende la gestione del clipping complessa. Per i poligoni convessi comunque non si cono problemi. Tramite la **tassellazione dei poligoni concavi** suddivido il poligono complesso in tanti piccoli poligoni convessi, e passare al clipping ciascuno di loro; devo utilizzare e scrivere però una operazione di tassellazione. Il clipping può essere visto anche come un processo che prende in input due vertici e può produrre un output come la coppia dei vertici della linea intersecata oppure nessun vertice (**black box**).

Pipeline di Clipping. Il clipping può essere visto come una pipeline in cui ogni stadio si occupa di verificare l'intersezione con un lato della finestra, per questo sono necessari quattro stadi indipendenti.

Clipping di figura complesse. Un *bounding box* è il più piccolo rettangolo che racchiude un poligono. Piuttosto che determinare il clipping di un poligono complesso, possiamo usare la bounding box allineata agli assi. È semplice così decidere se il poligono va accettato completamente o completamente rifiutato.

Rasterization. Tale processo determina i frammenti interno ad un poligono o di un segmento partendo dai vertici forniti in input (eventualmente dopo una fase di clipping). Ogni frammento ha informazioni circa il colore e la sua profondità (che serve per la rimozione delle superfici nascoste).

Rimozione di superfici nascoste. Un frammento diventa un pixel da scrivere nel frame buffer solo se si trova sopra un secondo pixel. Gli algoritmi per la rimozione delle superfici nascoste sono di due tipi:

- ✓ **Object-Space.** Utilizzando questo approccio confrontiamo i poligoni a coppia. Per ogni coppia di poligoni A e B possiamo avere quattro diverse situazioni: A occlude completamente B e visualizziamo A e contrario, A e B sono completamente visibili e li visualizziamo entrambi oppure A e leggermente sovrapposto a B e viceversa allora calcoliamo la parte visibile di entrambi. La cosa diventa complicata con troppo poligoni.
- ✓ **Image-Space.** Si applica nello spazio degli oggetti. Consideriamo un modello del raggio generato dall'osservatore e che passa attraverso un pixel. Ogni raggio viene intersecato con tutti i poligoni presenti e tra tutti quelli colpiti scegliamo quello più vicino all'osservatore. Il colore finale sarà quello del poligono colpito, e la complessità di questo sistema è proporzionale alla quantità di poligoni presenti.

Viene naturale utilizzare l'**algoritmo di approssimazione della sfera**, per sfruttare l'informazione

della profondità. In questa maniera sviluppiamo una approssimazione polinomiale di una sfera utilizzando una suddivisione ricorsiva a partire da un semplice tetraedro.

CAPITOLO 11 ~ Tecniche discrete

Buffer. Un **buffer** è una regione di memoria definita dalla sua *risoluzione spaziale* (nxm) e dalla sua *profondità* o *precisione* k, il numero di bit/pixel. Il OpenGL di buffer ce ne sono diversi con profondità variabili, alcuni dedicati alle operazioni di grafica (contengono informazioni relative al rendering). Oggi siccome i sistemi grafici consentono maggiore maneggevolezza è possibile gestire ed agire direttamente su questi buffer, un tempo tale capacità era preclusa. Ci sono operazioni previste che possono fare riferimento a maschere opportune, da eliminare con facilitò quando non è più necessario.
Concettualmente è possibile considerare tutta la memoria con un (grande) vettore bidimensionale di pixels. Possiamo quindi scrivere e leggere blocchi rettangolari di pixels con delle operazioni dette bit block transfer operation. Il frame buffer fa parte di questa memoria.
Trasferire informazioni in un buffer non è banale, per scrivere nel buffer è necessario leggere il pixel di destinazione prima di scriverlo. Per sovrascrivere e spostare generalmente blocchi di memoria, poiché si tratta di scrivere dei bit, è opportuno compiere delle operazioni di OR o XOR per scambiare le informazioni tra locazioni in maniera efficace. Il trasferimento così effettuato è conservativo.

Alpha-Blending. Normalmente ogni nuovo pixel sovrascrive il pixel precedente nel frame buffer. Sarebbe impossibile simulare la semi-trasparenza di un oggetto senza considerare la tecnica dell'alpha-blending, dove nell'informazione del colore inseriamo una quarta variabile che di default è sempre messa a 1 per indicare quanto trasparente è l'oggetto in questione.
In OpenGL, la quarta componente RGBA è detta **componente alpha** o **canale alpha**. La quantità alpha determina la trasparenza dei pixel dell'oggetto. Il valore di alpha è considerato a qualunque livello della pipeline.

Compositing. È una tecnica che permette di creare una immagine sintetica attraverso il rendering di molteplici elementi combinati insieme (mette insieme le diverse immagini per crearne una complessiva).

Formato immagini con alpha-blending. Esistono due formati di immagini che oltre al colore RGB includono anche il canale alpha: (TIFF) **.tiff** e (Targa) **.tga**. I programmi come Photoshop e Paintshop permettono di disegnare direttamente nel canale alpha.

The Pixel Pipeline. OpenGL ha una pipeline separata per i pixels, utilizzata per gestire le informazioni quando l'oggetto non è una figura semplice ma l'informazione di partenza non è data da vertici ma da pixel. I pixel non possono subire la stessa trasformazione geometrica subita dai vertici, allora ha una pipeline tutta sua, con delle operazioni dedicate di *lettura* e *scrittura*. **Writing Pixels** implica le seguenti operazioni: copiare i pixel dalla memoria del processore al frame buffer, *Format Conversion* se è necessario cambiare il formato delle informazioni da o in matrici di pixel, *Mapping* per mappare i pixel nella memoria ed associare i colori con le *Lookup Tables*, *Tests*. Il **Reading Pixels** concerne solo *Format Conversion*.

Raster Position. OpenGL conserva in memoria una "raster position" che parte dello stato della macchina OpenGL (equivalente ad un cursore interno in coordinate schermo), modificabile con la funzione **glRasterPos*()**. La raster position è una entità geometrica, passa per la pipeline geometrica, fornisce una posizione 2D in coordinate schermo ed indica dove la prossima primitiva raster verrà disegnata.

Buffer Selection. OpenGL può disegnare (write) o leggere (read) da uno qualsiasi dei color buffers. Il buffer di default è il *back buffer*. Il formato dei pixel nel frame buffer è diverso dal formato della memoria gestita direttamente dal processore perchè i due tipi di memoria hanno collocazioni diverse.

Bitmaps. OpenGL gestisce i pixel 1-bit (bitmaps) diversamente dai multi-bit (pixelmaps). Nel caso dei bitmaps sono l'equivalente mi maschere che determinano se il corrispondente pixel nel fram buffer sarà disegnato con il colore raster corrente. Ideale per il testo raster.

Pixel Maps. OpenGL permette di lavorare con matrici rettangolari di pixels chiamate pixel maps o immagini. I pixel sono organizzati in byte (8 bit), le tre funzioni disponibili sono: draw pixel (dalla memoria del processore al frame buffer), read pixel (dal frame buffer alla memoria del processore) e copy pixel (da frame buffer a frame buffer).
OpenGL non prevede un supporto per immagini comuni e bisogna "arrangiarsi". Bisogna cercare librerie adeguate o scrivere funzioni appropriate.

I limiti della modellazione geometrica. Le schede grafiche hanno la capacità di elaborare fino a decine di milioni di poligoni al secondo, ma tale numero non è sufficiente per rendere molti fenomeni quali nuvole, vegetazione, terreno e pelle. Se volessimo modellare una arancia dovremmo partire da un modello geometrico sferico e poi dovremmo sostituire la sfera con un modello più complesso. Dovremmo fotografare una vera arancia, digitalizzarla e sostituirla al modello sferico, questo processo è detto **texture mapping**. Il *texture mapping* impiega immagini per riempire i poligoni, *environment* impiega una immagine d'ambiente per riempire il poligono ed il *bump mapping* modifica i vettori dell'immagine durante il rendering.
Le operazioni di mapping sono implementate alla fine della pipeline di rendering, è molto efficiente perchè il numero dei poligoni è ridotto dopo l'operazione di clipping. Di tecniche di texture mapping ne esistono molte, e non sono semplici.

Sistemi di coordinate. Ne esistono molte.

- ✓ **Coordinate parametriche.** Potrebbero essere utilizzate per modellazione di curve e superfici;
- ✓ **Coordinate della texture.** Impiegate per identificare i punti nell'immagine che deve essere mappata;
- ✓ **Coordinate oggetto o mondo.** Concettualmente, dove il mapping dovrebbe aver luogo;
- ✓ **Window coordinates.** Dove l'immagine finale viene visualizzata.

Il problema di base è come determinare la funzione di mapping. Se consideriamo il mapping da coordinate texture ad un punto sulla superficie. Sembrerebbero necessarie tre funzioni ma noi vorremmo utilizzare la direzione opposta. Una soluzione al problema del mapping è di eseguire una prima operazione di mapping della texture su una semplice superficie intermedia (es: mapping su cilindro o sfera).

CAPITOLO 12 ~ Texture Mapping 1

Definizioni. Una **texture** è una immagine bitmap utilizzata per rappresentare il materiale di una superficie, che può essere una foto oppure può essere generata da un algoritmo (tramite *texture procedurale*). Il **texture mapping** è l'operazione di applicazione di una texture sulla superficie di un oggetto, in questo modo si aumenta in maniera considerevole il realismo di una scena senza aumentare la complessità geometrica. Un **texel** è l'abbreviazione di *texture element* per indicare un elemento della texture.
Per ogni triangolo all'interno della geometria si mappa un corrispondente regione all'interno della texture.

Texture Space. Una texture è definita all'interno di uno spazio piano (s,t) che prende il nome di **texture space**, dove la coordinata in basso a sinistra è (0.0, 0.0) e quella in alto a destra è (1.0, 1.0). Indipendentemente dalla dimensione della texture l'indirizzamento del singolo texel è normalizzato tra [0,1].
Il texture space è definito nel range [0,1] ma questo non significa che le coordinate di texture devono essere limitate a questo spazio. In OpenGL è possibile definire cosa accade quando indirizziamo una texture fuori da questo range.

Coordinate di texture. Per mappare una texture all'interno di un triangolo è necessario assegnare ad ogni vertice una **coordinata di texture**. Tale coordinata di texture è un punto (s,t) nel piano s,t ed ad ogni vertice del triangolo corrisponde un punto all'interno dell'immagine. Durante la fase di scanline le coordinate di texture sono interpolate, poiché per tutti i pixel interni al triangolo determiniamo per interpolazione la coordinata di texture, tramite una operazione di lookup leggiamo il texel corrispondente ed il texel ottenuto è mappato nel triangolo.

Interpolazione lineare. Ad ogni passo unitario all'interno dello screen space non corrisponde necessariamente un passo unitario nel world space. Non tutti i pixel coprono la stessa area nel world space. L'**interpolazione lineare** produce artefatti dovuti alla prospettiva. Concettualmente qualunque valore interpolato linearmente è potenzialmente non accurato. Il problema si risolve utilizzando un'interpolazione prospettica che tiene appunto conto della prospettiva ed OpenGL supporta la correzione prospettica che è efficace anche se non efficiente.

Filtering, Magnification e Minification. Ogni texture mappata su un poligono è normalmente deformata rispetto alle trasformazioni imposte alla geometria. La texture può essere allungata o accorciata e raramente un texel corrisponde ad un pixel nella rappresentazione grafica. Se applico per esempio una texture a mattoni su un poligono per farlo assomigliare ad un muro. Se mi avvicino al poligono può capitare di perdere il dettaglio perchè un texel copre più pixel, e se mi allontano invece un texel potrebbe essere più piccolo del pixel.
Il fenomeno della **magnification** si verifica quando un texel è più grande di un pixel, una texture piccola è applicata ad una superficie grande o la camera si avvicina troppo alla superficie mappata. Determino il valore della texture da applicare al pixel tramite diverse operazioni:

- ✓ **Point Sampling (GL_NEAREST)**. Si determina il pixel più vicino alla coordinata di texture. Questo produce un texel che copre diversi pixel.
- ✓ **Bilienar Sampling (GL_LINEAR)**. Determino con interpolazione lineare il valore da assegnare al pixel effettuando una media attorno ai 4 texel più vicini alla coordinata di texture.

Il fenomeno della **minification** si verifica quando un texel è più piccolo del pixel da mappare. Per determinare tale pixel posso:

- ✓ pensare alla soluzione da adottare che sarebbe di effettuare una media di tutti i texel che cadono nel pixel ma è troppo costosa per via degli accessi in memoria (interpolazione lineare);
- ✓ utilizzo soluzioni immediate come *Point Sampling* o *Bilineare Sampling*;
- ✓ utilizzo la **mipmapping**: è la soluzione migliore, si tratta di una tecnica che permette di avere più copie della texture originale ma a risoluzioni differenti. Risolve in maniera efficiente e con buoni risultati il problema della minification. Così l'immagine originale è affiancata da differenti versioni più piccole.

Basic Strategy. I passi necessari per usare la texture mapping in OpenGL sono:

- caricare il file di immagine o generare la texture periodica all'interno del codice;
- creare e definire un oggetto di tipo texture al quale associare l'immagine caricata;
- mappare la texture sul poligono.

Le texture però devono avere dimensioni in altezza e lunghezza pari a una potenza di 2 (32x32, 64x64, ecc). Non devono però essere necessariamente quadrate. Le texture occupano memoria sulla scheda video ed attraverso le estensioni sono possibili texture rettangolari.

Caricare/Generare la texture. Una texture può avere diverse fonti, può essere di diversi formati oppure è generata proceduralmente. Generalmente si utilizza una libreria per la gestione dei diversi formati grafici. OpenGL interpreta la texture in memoria come un array di byte e numerose sono le opzioni per definire come è impacchettata in memoria (*PixelStorei*).

Creare e definire una texture object. Una volta caricati i dati per la texture possiamo trasferire questi dati nella memoria grafica. Una texture object memorizza i dati di una texture e permette ad OpenGL di effettuare il texture mapping in qualunque momento. I passi necessari per

l'uso di una texture object sono:

- ✓ generare una texture;
- ✓ impostare lo stato corrente sulla texture appena create per definire le proprietà della texture;
- ✓ impostare le priorità della texture;
- ✓ Bind e Unbind la texture ogni qualvolta la texture deve essere mappata su di un poligono.

Automatic Texture-Coordinate Generation. Per oggetti complessi OpenGL permette di generare automaticamente le coordinate di texture. Non è un approccio sempre utilizzabile ma alle volte aiuta. Le coordinate di texture sono generate direttamente a partire dai vertici della geometria. Sono possibili tre approcci:

- ✓ **Object space**, dove la texture è attaccata all'oggetto;
- ✓ **Eye space**, dove l'oggetto si muove mentre la texture è fissa;
- ✓ **Sphere map**, che si basa sul vettore riflesso e permette di riflettere una texture.

Il mapping è calcolato a partire dall'equazione del piano.

CAPITOLO 13 ~ Texture Mapping 2

Sphere Mapping. Le coordinate sono calcolate considerando il vettore riflesso sulla normale del vertice del vettore dello sguardo. La direzione riflessa del vettore è usata come coordinata di texture. Si assume che la texture sia una texture panoramica a 360°.
I passi necessari per lo sphere mapping sono:

- ✓ effettuare il bind della texture che contiene la texture da riflettere;
- ✓ impostare il mapping sferico;
- ✓ abilitare la gestione delle coordinate di texture;
- ✓ disegnare gli oggetti fornendo le normali corrette per ogni vertice di ogni faccia;
- ✓ disabilitare la generazione delle coordinate.

Nella **Cube Enviroment Map** una *cube maps* è composta da 6 texture della stessa dimensione che rappresentano le facce di un cubo centrato nell'origine. Ogni faccia del cubo rappresenta una direzione lungo i tre assi. Per indicizzare un texel sulla cube map si utilizza un vettore direzionale che rappresenta una coordinata di texture 3D. Il vettore riflesso, utilizzato per accedere alla cube map, è calcolato utilizzando la direzione dello sguardo e la normale alla superficie.
La generazione della coordinata della cube map è effettuata dal calcolo (che ottiene una proiezione) per generare la coordinata 2D dal vettore riflesso 3D. Per accedere alla faccia di una texture abbiamo comunque bisogno di una coordinata 2D. La proiezione da 3D a 2D avviene usando una certa procedura: selezione della componente più grande in valore assoluto, la divisione delle restati componenti con la componente selezionata e l'accesso alla texture.

Creare una Cube Map in OpenGL. Le cube maps sono supportate da OpenGL a partire dalla versione 1.4 attraverso un nuovo tipo di texture chiamato GL_TEXTURE_CUBE_MAP. La nuova texture può essere usata con tutte le funzioni che prevedono una texture target all'interno dei parametri. È inoltre supportata una nuova modalità di generazione automatica delle texture. Con GL_REFLECTION_MAP_EXT impiego la stessa tecnica di calcolo descritto nella sphere map per determinare il vettore di riflessione, GL_NORMAL_MAP_EXT si impiega nel caso di rendering di scene con sorgenti a distanza infinita a riflessione diffusa.

Dynamic Cube Map Reflection. Per creare una cube map sono necessari i seguenti passi:

- – impostare il FOV della camera a 90° nella posizione dell'oggetto che deve riflettere l'ambiente;
- – puntare la camera lungo l'asse positivo delle x e renderizzare la scena;
- – ripetere il processo per la faccia negativa della x e di tutte le altre direzioni ruotando ogni volta la camera.

In questo modo generiamo dinamicamente un enviroment map che teoricamente andrebbe fatto per ogni oggetto ma se la scena è sufficientemente grande è necessaria una sola cube map. La risoluzione può essere inferiore a quella del frame buffer e può non essere generata ad ogni frame.

Skybox. È un metodo per creare un background e far apparire una scena più grande di quanto non sia. L'intera scena è rinchiusa all'interno di una **skybox** rappresentata mediante un cubo. Tutti gli oggetti non raggiungibili sono proiettati sulle facce del cubo creando l'illusione di oggetti 3D molto distanti.
Utilizzando una cube map possiamo contestualmente generare anche una skybox. È necessario disabilitare la generazione delle coordinate di texture automatica, guardando i vertiti all'interno appariranno in senso antiorario e quindi saranno visibili.

Multitexturing. Consente di applicare più di una texture su di un poligono attraverso le unità di texture (*texture unit*). Una **unità di texture** associa ad un frammento un texel di una texture e passa il risultato alla unità di texture. Applicazioni della multitexturing sono gli effetti di illuminazione, compositing ecc.
Ogni unità di texture combina il precedente colore del frammento con il colore della propria texture e passa il risultato alla prossima unità di texture (se attiva). Il colore finale dipende dai colori di tutte le texture che contribuiscono allo stesso frammento. Dal punto di vista pratico non ci sono novità, è necessario solo abilitare le texture e decidere come combinarle insieme, per fare ciò si utilizza la funzione **glActiveTexture(GLenum texUnit)**. Durante il passaggio dei vertici per ogni unità è necessario specificare le coordinate di texturing attraverso **glMultiTextCoord(GLenum texUnit, TYPE coords)**. Per disabilitare il multitexturing dobbiamo disabilitare tutte le unità di texture maggiori di 0.

CAPITOLO 14 ~ Curve e superfici 1

Fuga da Flatlandia. Fino ad ora abbiamo lavorato con primitive grafiche basate su linee e poligoni planari che si adattano bene all'hardware grafico e sono matematicamente semplici. Ma il mondo non è composto solo di entità “planari”, abbiamo bisogno di curve e di superfici curve e la loro implementazione in termini di operazioni di rendering potrebbe far ricorso a primitive di tipo planare come linee e poligoni.
Esistono molti modi per rappresentare le curve e le superfici:

- ✓ **Rappresentazione esplicita.** È la forma più familiare della rappresentazione delle curve 2D attraverso una formula matematica anche se non è possibile rappresentare tutte le curve ma permette delle estensioni al caso 3D.
- ✓ **Rappresentazione implicita.** Permette la realizzazione di curve bidimensionali in una maniera molto più robusta potendo disegnare così tutti i tipi di linee e circonferenze. Sono possibili espressioni che definiscono una superficie tridimensionale all'interno della quale l'intersezione di sue piani definisce una curva. In generale non è possibile esprimere i punti che soddisfano l'equazione in forma esplicita.

Superfici algebriche. Sono superfici quadratiche con al più 10 termini dove è possibile risolvere una intersezione con un raggio riducendo il problema alla soluzione di una equazione quadratica. Le **curve parametriche** sono equazioni separate per ciascuna rappresentazione spaziale.
Per scegliere le funzioni usualmente possiamo scegliere delle “buone” funzioni che non sono uniche per una data curva spaziale, delle quali possiamo interpolare o approssimare i dati, funzioni facili da valutare e da derivare e che permettano lo *smooth*.
Le **linee parametriche** possono definire una linea od un vettore che collega due punti distinti. Le **superfici parametriche** richiedono due parametri e pretendono le stesse proprietà delle curve.
Si usano spesso i *polinomi* perchè sono facili da calcolare e sono continui e differenziabili ovunque, basta prestare attenzione alla continuità nei punti di giunzione includendo la continuità delle derivate.

Altri tipi di curve e superfici. È possibile superare le limitazioni legate alla scelta dell'interpolazione con perdita di smooth o discontinuità delle derivate nei punti di giunzione. Abbiamo 4 condizioni (per le cubiche) che possiamo applicare a ciascun segmento. Impieghiamo allora condizioni diverse dell'interpolazione.

Continuità parametrica e geometrica. Possiamo richiedere che le derivate delle funzioni x, y e z siano continue nei punti di giunzione (= *continuità parametrica*). In alternativa possiamo richiedere che solo le tangenti alle curve risultanti siano continue (= *continuità geometrica*). L'ultima richiesta offre maggiore flessibilità perchè abbiamo solo due condizioni da soddisfare invece che tre nei punti di giunzione.

Higher Dimensional Approximations. Le tecniche sia per i polinomi interpolanti che per quelli di Hermite possono essere impiegati con polinomi di ordine maggiore. Nel caso interpolante, la matrice risultante diviene all'aumentare dell'ordine sempre più mal condizionata e le curve meno smooth presentano problemi di instabilità del calcolo. In entrambi i casi il lavoro per le operazioni di rendering aumenta sia per le curve che per le superfici.

L'idea di Bezier. Nella grafica e nel CAD non abbiamo informazioni sulle derivate delle nostre curve o superfici. Bezier suggerì che si sarebbero potuti utilizzare gli stessi 4 punti nel caso dell'interpolazione per approssimare le derivate della forma di Hermite.
Le proprietà dei polinomi di Bernstein assicurano che tutte le curve di Bezier rimangono all'interno del convex hull definito dai punti di controllo. Quindi anche se non interpoliamo tra i dati non possiamo andare troppo lontano.
Sebbene la forma di Bazier sia meglio dell'interpolazione, abbiamo che le derivate non sono continue nel punto di giunzione. Per migliorare potremmo andare agli ordii superiori di Bazier ma questo implica più tempo di lavoro e le derivate continuano a rimanere discontinue, oppure applichiamo condizioni differenti facendo attenzione a non aumentare l'ordine.

B-Spline. Dette Basis Splines, permettono di imporre condizioni di continuità addizionali sulle derivate per ciascuno segmento, sarà necessario un maggior costo computazionale di fattore x3 rispetto alle curve già viste. Si aggiunge un punto alla volta anziché tre. Per le superfici il costo sarà maggiore di un fattore x9.
È possibile generalizzare il discorso fino ad includere spline di qualsiasi grado. Dati e condizioni non devono essere forniti in punti equispaziati, è possibile ripetere i nodi ed è possibile imporre che la spline passi per determinati punti.

CAPITOLO 15 ~ Curve e superfici 2

B-Spline. Sono una tipologia di polinomi di grado 3 costruiti con l'idea di rendere la curva più “morbida” e bella. Le spline permettono di imporre condizioni di continuità addizionali sulle derivate per ciascun segmento. (vedi cap.14 par. “B-Spline”).
Cerchiamo di costruire delle curve speciali sulle quali imporre delle condizioni sulle derivate, le curve nell'intervallo tra un punto ed un altro hanno continuità ovunque a parte che sui punti di confine dove non è assicurato quanto detto sopra. Dobbiamo fare in modo che anche lì le curve siano continue: il segmento di curva che determiniamo è “compreso” tra i due punti, la curva complessiva si calcola per ogni punto di riferimento. La condizione di continuità nel punto di giunzione di due spline è che i loro valori per quel punto siano gli stessi, stesso discorso per le loro derivate prime.
Sono dette **blending function** le funzioni base delle B-Spline, tutte comprese tra 0 e 1 e la loro somma punto per punto da sempre 1 (si parla del valore delle funzioni). La curva inoltre è sempre contenuta entro il quadrilatero formato dai quattro punti di controllo.

B-Spline e funzioni di base. L'influenza dei punti di controllo è locale, se ne modifico uno l'effetto si propaga solo su una parte della curva (influenza solo 4 segmenti della curva, non tutti). È possibile definire una curva o una funzione generalizzata di spline che possiede una data espressione algebrica. Tale espressione ci permette di esprimere l'intera spline con una singola espressione. In pratica, dato un insieme di punti di controllo, posso costruire una funzione polinomiale a tratti su tutto l'intervallo come combinazione lineare delle funzioni di base.

General B-Spline. Il discorso è estendibile al caso in cui si voglia aumentare il grado del polinomio, definendo spline in cui i punti non sono equispaziati (anche se sarebbe più semplice averne per facilitarne il calcolo esplicito). Possiamo imporre anche se si tratti di curve interpolanti.

Esiste poi una generazione ricorsiva di spline la quale parte da una funzione che convola con se stessa fornisce una spline di ordine 1, aumentando il grado di convoluzioni passiamo progressivamente di grado in grado aumentando l'intervallo di appartenenza.

NURBS. Esistono spline introdotte per pesare tutti i punti di controllo con lo stesso peso, posso decidere magari che i punti centrali sono più importanti dei periferici oppure potrebbe essere utile usare le curve per poi poterle tracciare in una visione prospettica per correggere la prospettiva. Introduciamo una quarta variabile per descrivere la curva in coordinate omogenee. È utile usare curve espresse dal rapporto di polinomi di grado finito. Sono ottime per la visualizzazione prospettica.

Valutazione dei polinomi. Il metodo più semplice per il rendering di una curva polinomiale è di valutare il polinomio in un numero elevato di punti e di costruire la relativa poli-linea. Per le superfici potremmo formare una rete di triangoli o quadrilateri che approssima la superficie in oggetto. Il metodo di Horner per la valutazione dei polinomi richiede solo n moltiplicazioni.
Con la tecnica delle **differenze finite** utilizziamo valori equidistanti per cui la differenza ennesima del polinomio è costante. La differenza finita 0 è il valore del polinomio nel punto, le altre si ottengono in maniera ricorsiva.

Il metodo ricorsivo di deCasteljau. Possiamo impiegare la proprietà nota come "convex hull" delle curve di Bezier per ottenere un metodo ricorsivo efficiente che non richiede valutazioni di funzione impiegando solo valori nei punti di controllo. Si basa sull'idea che un qualsiasi polinomio o una qualsiasi sua parte è un polinomio di Bezier con punti di controllo scelti opportunamente.
Con un tot di punti determiniamo un polinomio cubico di Bezier e il suo convex hull. Possiamo ora considerare due polinomi metà del precedente (uno destro ed uno sinistro). Determiniamo allora un gruppo di punti di controllo per le due semicurve determinate, essendo a loro volta curve di Bezier, il loro convex hull è più vicino alla curva del precedente: tale proprietà è nota come **proprietà di riduzione della variazione** della curva di Bezier. Ripetendo l'operazione in maniera ricorsiva otteniamo una buona approssimazione.

CAPITOLO 16 ~ Billboard e Sistemi Particellari

Billboard. Tecnica per simulare ambienti o situazioni in cui ci sono oggetti che si ripresentano in maniera ripetitiva. Aumentare il numero di poligoni per un modelli 3D significa conferire realismo ad una scena. Tuttavia all'aumentare del numero di poligoni le performance ne risentono. In generale anche se l'hardware evolve è bene prestare sempre attenzione al numero di poligoni che si utilizzano. I poligoni che si risparmiano per un modello 3D possono essere usati per altri modelli. All'interno di una scena un *billboard* (o impostor) è un oggetto 2D con texture la cui caratteristica è quella di avere una faccia sempre rivolta verso una direzione, in generale verso la camera. La tecnica è molto utilizzata per ridurre la complessità geometrica di una scena senza influire più di tanto sul realismo del rendering.
Si usano i billboard per creare l'illusione di avere nella scena un oggetto 3D, la tecnica si estende anche agli oggetti che devono puntare verso una posizione target. Posso anche utilizzare una texture animata per dare l'illusione che l'oggetto 3D si muova (attenzione all'effetto *popping* causato quando la texture cambia bruscamente). In generale si può usare sempre ma il billboard è consigliato soprattutto in scene di massa con oggetti ripetuti.
La geometria utilizzata per il billboard è un semplice quadrilatero su cui è mappata una texture. L'orientamento e la posizione sono determinati attraverso una direzione ed un vettore. Nelle trasformazioni a corpo rigido abbiamo che una sequenza di rotazioni e traslazioni può essere espressa come una matrice di trasformazione, con una matrice simile si può definire un billboard.

- ✓ **Axial Billboard.** Un solo asse è orientato verso l'osservatore mentre gli altri due assi sono gli stessi del world space. La faccia del billboard è ruotata in modo da allinearsi sempre con la faccia dell'osservatore, se l'osservatore osserva la billboard dall'alto si percepisce il "trucco". Utilizzato frequentemente per rappresentare alberi.
- ✓ **World-Oriented Billboard.** Utilizzata nei sistemi particellari in sui un billboard è sempre rivolto verso l'osservatore indipendentemente dalla sua posizione.

- ✓ **Screen-Aligned Billboard.** Utilizzato per mostrare informazioni come testo in quanto sempre allineato con la viewport oppure per mostrare lensflare (riflessi di lente). La direzione della billboard è la normale negata della camera. Il vettore direzione alta è quello della camera, cosicchè i due vettori sono già perpendicolari.

Nebbia. Ogni oggetto tende a scomparire all'orizzonte mentre si allontana dall'osservatore. Come conseguenza il numero dei dettagli tende a diminuire o a scomparire in lontananza. Questo è il concetto che sta alla base della realizzazione della nebbia in OpenGL. Sono disponibili tre tipi di fog factor (GL_FOG_MODE) chiamati lineare (GL_LINEAR), esponenziale (GL_EXP) e *gaussiano* (GL_EXP2). Il metodo esponenziale è il più realistico.

Sistemi Particellari. Sono situazioni in cui gli oggetti presenti nella scena sono rappresentati tramite delle entità chiamate particelle con caratteristiche rappresentate da coordinate nello spazio, velocità ed altro. Ad ogni particella può essere associato un oggetto più complesso. Non hanno superfici lisce o definite, ma sono irregolari, complesse. Non sono oggetti rigidi ed i loro spostamenti non possono essere descritti dalle trasformazioni geometriche usuali. Si usano tali sistemi in simulazione di esplosioni, nuvole, pioggia, folle, stormi, scariche elettriche, fuochi.
Nei sistemi particellari gli oggetti non sono rappresentati come poligoni ma come nuvole di particelle che creano solo l'illusione del volume, le particelle formano un modello in continua evoluzione e non deterministico. L'oggetto modellato è ottenuto in maniera procedurale con parametri che controllano il comportamento ed è possibile gestire il corso della vita della particella.
Un **sistema particellare** è un insieme di punti 3D nello spazio che ha un certo ciclo di vita che detta la posizione e le caratteristiche di ogni punto nel tempo. Il comportamento è pseudocasuale.

Particelle Newtoniane. Un sistema di particelle è un insieme di particelle. Ciascuna particella è un punto materiale dotato di massa, possiede sei gradi di libertà e ciascuna particella ubbidisce alle leggi di Newton. La complessità del sistema dipende anche dalla complessità delle forze in gioco, cresce linearmente col numero di particelle interagenti.

- ✓ **Maglie.** Collegano ciascuna particella alle particelle più vicine, impiega il sistema spring-mass.
- ✓ **Forze semplici.** Considerano la forza sulla particella i-esima, la forza di attrito o di trascinamento.
- ✓ **Forze elastiche.** Si assume che ciascuna particella abbia una massa unitaria e sia collegata alla particella vicina con una molla. Si sfrutta la legge di Hooke. Un sistema molla-massa oscilla in eterno e se si vuole simulare lo smorzamento si possono aggiungere forze proporzionali alle velocità delle particelle.
- ✓ **Attrazione e repulsione.** Si simulano con particelle elettriche.

Suddivisione spaziale in scatole. È una tecnica di suddivisione spaziale, si divide lo spazio in regioni cubiche o a forma di parallelepipedo, le particelle possono interagire solo con le particelle presenti nella stessa scatola o nelle scatole vicine.

Linked Lists. Ciascuna particella tiene una lista delle particelle vicine, la struttura va aggiornata continuamente e si può ammortizzare il costo di questa operazione fornendo una struttura dati inizializzata.

Vincoli e urti. È facile in computer grafica ignorare le leggi della fisica, dopotutto le superfici sono virtuali. Dobbiamo individuare gli urti separatamente se vogliamo una soluzione esatta. Si possono approssimare come forze repulsive. Una volta individuato, un urto si calcola con la nuova direzione della particella. Si impiega il coefficiente di restituzione, la componente verticale cambia di segno.

CAPITOLO 17 ~ La programmazione della GPU (1)

Introduzione. Negli ultimi anni il più significativo progresso nella grafica *realtime* si è avuto con l'introduzione della pipeline programmabile. Oggi tutti i produttori di schede grafiche supportano le

pipeline programmabili attraverso le proprie GPU. Le librerie che supportano le pipeline programmabili sono numerose, come OpenGL e altre.
Prima del 2001 le schede grafiche erano dotate delle cosiddette **fixed pipeline**. Nella pipeline programmabile due componenti diventano completamente programmabili, i *vertex programs (shaders)* ed i *fragment programs (shaders)*. L'idea non è completamente nuova ma era già stata implementata all'interno di RenderMan (motore di rendering sviluppato dalla Pixar per i film in computer grafica). Il vantaggio della GPU programmabile è che non ci sono limiti alle possibilità del programmatore. Sono possibili effetti speciali di ogni tipo ed è possibile un controllo totale della GPU nel modo in cui i dati sono elaborati, ottimizzando anche il codice.

Elaborazione geometrica. In una pipeline tradizionale le informazioni sulla geometria sono fornite dal programma o da una display list o da un evaluator. I dati geometrici sono composti dai dati sui vertici e dal tipo della geometria.
I vertici sono trasformati attraverso la matrice modelview nelle coordinate dell'occhio (o della camera). Le normali sono trasformate utilizzando la trasposta inversa della matrice di modelview e le coordinate di texture sono generate automaticamente se è abilitato l'auto-texture.

Calcolo della luce. Considerando il modello di illuminazione di Phong senza attenuazione, tale modello è applicato ad ogni vertice generato per determinare la quantità di luce o di colore di ogni vertice. Questa fase si svolge nel vertex processor.

Primitive Assembly. I vertici processati vengono assemblati al passo successivo per formare gli oggetti come poligoni e forme varie. I vertici vengono quindi trasformati attraverso la matrice di proiezione. Il clipping elimina tutti i vertici non compresi nel volume della vista. Durante la fase di rasterizzazione le entità geometriche sono rasterizzate in frammenti, ogni frammento corrisponde ad un punto sulla griglia di interi perchè in pratica un frammento è un potenziale pixel.

Programmable Shaders. L'idea della programmabilità delle schede grafiche è di poter sostituire le funzioni fissate dai produttori con processori programmabili chiamati **shaders**. Oggi esistono in commercio shaders programmabili per vertici e frammenti. Se vogliamo usare gli shaders è necessario fornire tutte le funzioni necessarie a rimpiazzare quelle fornite di default dal produttore hardware. È necessaria una buona conoscenza della pipeline e dei dati che ogni shader prende in input.

- ✓ **RenderMan.** Sviluppato da Pixar, l'idea di base è che in genere il linguaggio prevedeva un renderizzatore che lavorava su un modello ed implementava le istruzioni con una struttura gerarchica che aveva un tot di nodi che prevedevano funzioni o parametri, combinando opportunamente tali nodi era facile sostituire un certo tipo di operazione una serie di parametri della scena grafica (*shanding trees*). L'output del modellatore è costituito dal modello geometrico con in più le informazioni necessarie all'operazione di rendering.

Storia breve sui linguaggi di shading per PC. Per i primi processori programmabili shader si utilizzava un linguaggio molto simile all'assembly. **OpenGL Shading Language** è un linguaggio di programmazione ad alto livello per la gestione degli shader di una GPU. Lo scopo è permettere un controllo più diretto ed immediato della pipeline grafica.
I *Data Types* sono basati sul linguaggio C ANSI con aggiunte di C++. Le tipologie di dato sono vettori multidimensionali. Non esistono *puntatori*, si possono però usare variabili di tipo *struttura*, se si devono passare parametri alle funzioni esistono dei *modificatori* di variabili che permettono di indicare se il valore è da prelevare od è dichiarato in un'altra funzione. Esistono *qualificatori* che hanno la stessa funzionalità che hanno anche in C, le variabili però possono modificarsi a seconda se fanno riferimento a vertici, frammenti che possono cambiare in qualsiasi punto dell'applicazione.

- ✓ **attibute.** Le variabili con questo qualificatore possono cambiare al più una volta per vertice all'interno del vertx shader;
- ✓ **uniform.** Le variabili con questo qualificatore rimangono costanti per l'intero lotto di primitive;
- ✓ **varying.** Le variabili con questo qualificatore sono impiegate per trasferire informazioni tra un vertex shader ed un fragment shader.

Gli *attributi* per i vertici verranno interpolati nella pipeline e poi calcolati come attributi dei frammenti. Le *variabili speciali* sono variabili collegate alla pipeline di OpenGL precedute da gl_, nel vertex shader posso accedere a tutte le qualità di ogni vertice modificandole. Alcune variabili hanno lo stesso nome ma range di utilizzo diverso.

CAPITOLO 18 ~ La programmazione della GPU (2)

Vertex Lightining with shaders. Uno degli usi più frequenti dei vertex shader è l'applicazione di modelli di illuminazione alternativi al modello di Phong. Quindi per primo esamineremo l'implementazione dei modelli di Phong e di Blinn-Phong per poi analizzare modelli alternativi. Il modello di Phong riporta una espressione valida per ogni componente RGB. Per poter operare correttamente con le luci bisogna scegliere il SdR in cui lavorare (es. eye frame), quindi sarà necessario determinare i vertici, le normali ecc in questo SdR. Il vertex shader diventa più complesso da calcolare, per il quale dobbiamo calcolare il contributo della luce (Blinn-Phong) ed altro.

Vertex Shader per Fragment Lighting. Se devo trasferire il calcolo da vertice a frammento ho bisogno delle stesse informazioni per il vertice, per cui la normale, la direzione della luce, la direzione dell'osservatore e le coordinate del vertice. Per passare da frammento a frammento devo lasciare che la pipeline interpoli da sola le informazioni.
Se volessi implementare il vero modello di Phong frammento per frammento dovrei calcolare i vettori riflessi. Il vero modello di Phong non è ottenibile in OpenGL, ma è utilizzabile tramite gli shaders. La differenza tra i modelli è apprezzabile, quando è possibile conviene utilizzare il modello di Phong.

Impiego delle Texture. Per utilizzare le texture è necessario poter accedere alle coordinate di texture per vertice. GLSL fornisce alcune variabili *attribute*, una per ciascuna unità di texture. Inoltre è fornito l'accesso alle matrici di texture per ciascuna unità di texture tramite un vettore *uniform*. Il compito del vertex shader è di calcolare le coordinate delle texture per il vertice ed assegnarle ad una variabile predefinita di OpenGL dove è indicata l'unità di texture interessata. La texture determinerà il colore del frammento. Si possono complicare le cose caricando texture in modo che diventi il colore relativo al frammento, si possono anche far interagire più texture tra loro pesandoli opportunamente o moltiplicandoli.
OpenGL permette di combinare il colore delle texture con il colore dei frammenti in diversi modi.

Passi in OpenGL per utilizzare gli shader. Se si vuole utilizzare le funzioni di OpenGL per realizzare gli effetti di shader, è necessario:

- ✔ leggere il codice sorgente dello shader;
- ✔ generare il programma oggetto;
- ✔ creare oggetti shader;
- ✔ collegare gli oggetti shader al programma oggetto;
- ✔ compilare gli shader;
- ✔ collegare ogni cosa insieme;
- ✔ selezionare il programma oggetto corrente;
- ✔ allineare le variabili *attribute* e *uniform* tra l'applicazione e gli shader.

Tipicamente gli shader sono presenti in un file sorgente in formato testo e devono essere letti dall'applicazione. Una volta letti in una stringa “null terminated” devono essere aggiunti all'oggetto programma e compilati.

Vertex Attribute. Una volta eseguite le operazioni di compilazione, link e selezione del programma, bisogna associare le variabili nello shader alle variabili del programma. Gli attributi dei vertici nello shader sono indicizzati nel programma principale tramite una tabella impostata durante l'operazione di link.

APPENDICE A ~ Funzioni di OpenGL

La sintassi di OpenGL. Ogni comando di OpenGL ha come prefisso **gl** (come anche le costanti simboliche, che però sono scritte in maiuscolo). Molti comandi hanno questo formato:

gl[comando][#][tipo][v]

dove **[comando]** indica il tipo di informazione/azione, **[#]** indica quanti dati passare (2,3 o 4), **[tipo]** indica il tipo di dato (int, float, double...) e **[v]** indica se passiamo un vettore di dati.
Per garantire la massima portabilità OpenGL definisce diversi tipi propri.

glVertex*()

La funzione passa come input i dati dei vertici di una figura. Per esempio, se passa due vertici di tipo float scriviamo *glVertex2f(2.0, 2.0)*.

glBegin(glEnum mode)
glEnd()

La funzione glBegin() indica l'inizio di una sequenza di dati in input, glEnum mode indica il tipo di figura che andrà disegnata mediante i dati espressi e glEnd() segna la fine della sequenza di dati.

glColor*()

La funzione imposta lo stato di OnpeGL relativamente al colore da usare per ogni vertice in input. Questa funzione segue sempre quella di dichiarazione di un vertice per specificare il suo colore.

glClearColor()

OpenGL permette di pulire il frame buffer con un colore precedentemente specificato.

glOrtho()

Permette di impostare il volume della vista nella proiezione ortografica. Se l'applicazione è in 2D possiamo usare la funzione glOrtho2d().

glViewport()

In OpenGL dobbiamo per forza utilizzare l'intera finestra per il rendering. Questa funzione permette di specificare quale parte della finestra vogliamo visualizzare.

glMatrixMode()

La funzione permette di selezionare quale modalità di trasformazione matriciale utilizzare.

GLuint glGenLists(Glsizei range)

Questo comando genera una o più display list. **Range** indica il numero di display list da allocare, l'**intero** restituito indica l'indice dal quale è stato generato il blocco consecutivo di display list.

glNewList(GLuint list, GLenum mode)
glEndList(void)

Una dispaly list appena creata è vuota, con questa funzione definisco l'inizio di un blocco di comandi da inserire: **list** è la display list da usare e **mode** è una costante simbolica. La funzione glEndList() termina il passaggio dei comandi nella display list.

GlCallList(GLuint list)
glListBase(GLuint base)
glCallLists(GLsizei n, GLenum type, const GLvoid *lists)

Una volta creata una display list può essere chiamata infinite volte, con questa funzione eseguo il blocco di comandi inseriti della display list **list**. Mediante le altre due funzioni posso eseguire diverse display list in successione.

GlDeleteLists(GLuint lists, GLsizei range)
GLboolean glIsList(GLuint list)

Con la prima funzione cancello le display list al termine dell'applicazione, come si dovrebbe sempre fare (cancello un **range** di display list a partire da **lists**). Con la seconda controllo se **list** è già stata eseguita oppure no.

glLight(GLenum light, GLenum pname, TYPE param)

La funzione permette di gestire le luci nella scena, dove **light** indica di quale luce parliamo, e **pname** è una tipologia di luce scelta tra un pool di alternative. **param** è il parametro associato alla caratteristica.

glMaterial(GLenum face, GLenum pname, TYPE param)

La funzione definisce le caratteristiche del materiale che costituisce l'oggetto, **face** indica la faccia dell'oggetto al quale applicare l'informazione, **pname** indica la caratteristica del materiale da modificare e **param** è un parametro associato alla caratteristica.

glShadeModel(GLenum mode)

La funzione definisce il modo con il quale determinare il colore interno dell'oggetto.

glBlendFunc(GLenum srcfactor, GLenum destfactor)

La funzione permette di specificare i fattori di blending della sorgente e della destinazione

glGenTextures(GLseizei n, GLuint *textureNames)

La funzione torna n textures dove **n** indica il numero di texture e **textureNames** è l'array contenente le texture appena create.

glBindTexture (GLenum target, GLuint textureName)

La funzione crea texture e indica ad OpenGL che texture usare in fase di mapping.

glTexImage2D (GLenum target, GLint level, GLint components, GLseizei width, GLseizei height, GLint border, GLenum format, GLenum type, const GLvoid *texel)

La funzione definisce una texture bidimensionale.

GlTexParameteri (target, par1, par2)

La funzione permette di impostare diversi parametri da associare alla texture corrente: magnfication, minification, mipmapping ed estensione del mapping space.

Target:
- GL_TEXTURE_2D

Par1:
- GL_TEXTURE_WRAP_S
- GL_TEXTURE_WRAP_T
- GL_TEXTURE_MAG_FILTER
- GL_TEXTURE_MIN_FILTER

Par2:
- GL_CLAMP
- GL_REPEAT

- GL_NEAREST
- GL_LINEAR

glTexCoord{1,2,3,4}{sifd} (TYPE coords)

La funzione permette la mappatura manuale e non della texure su ogni vertice del poligono.

glHint()

Abilita la correzione prospettica se il risultato non è soddisfacente.

glMap1f(type, u_min, u_max, stride, order, pointer_to_array)

La funzione imposta un valutatore, con **type** che indica che cosa vogliamo valutare, **u_min** e **u_max** che indicano i valori minimi e massimi di u, **stride** che indica la separazione tra i punti, **order** che indica il grado+1 del polinomio e **pointer_to_array** che punta ai dati.

glEvalCoord1f(u)

La funzione permette che tutti i valutatori abilitati siano calcolati per il valore specificato per u (che non devono essere equispaziati).

glFog (GLenum pname, GLfloat param, const GLfloat *params)

La funzione permette di attivare tutti i parametri relativi alla nebbia.

APPENDICE B ~ Funzioni di GLUT

glutInit(&argc, argv)

La funzione gestisce l'input da riga di comando per inizializzare l'applicazione.

glutInitDisplayMode(mode)

La funzione imposta le proprietà del frame buffer all'interno della finestra di rendering.

glutWindowSize(pixel, pixel)

La funzione specifica la dimensione della finestra di rendering in pixel.

glutWindowPosition(pixel, pixel)

La funzione specifica la posizione della finestra di rendering a partire dall'angolo in alto a sinistra.

glutCreateWindow(“nomefinestra”)

Apre la finestra di rendering.

void glutDisplayFunc(void (*func)(void))

Lancia la funzione di callback per il display. Invocata ogni volta che è necessario un *refresh*: quando la finestra appare per la prima volta, quando è ridimensionata, quando torna in primo piano o quando il refresh è manuale da parte dell'utente. Ogni programma che usa GLUT deve avere una o più callbak di refresh.

glutMainLoop()

Entra nel ciclo di rendering e porta l'applicazione in un loop infinito.

void glutReshapeFunc(void (*func)(int width, int heigth))

Width è la nuova larghezza della finestra e **height** la nuova altezza. Si invoca questa funzione quando la finestra cambia dimensioni, di default se manca una funzione simile GLUT mantiene il viewpoint coincidente con tutta la finestra.

void glutKeyboardFunc(void (*func)(unsiegned char key, int x, int y))

Invocata quando un utente preme un tasto, **key** riporta il codice ASCII relativo al tasto per il quale deve scattare la funzione, **x** e **y** sono le posizioni orizzontale e verticale del mouse.

void glutMouseFunc(void (*func)(int button, int state, int x, int y))

Come prima solo che si parla del mouse e non della tastiera. **Button** indica il tasto del mouse premuto, **state** se è premuto/rilasciato o trascinato, **x** e **y** le posizioni orizzontale e verticale del cursore.

void glutIdleFunc(void (*func)(void))

È invocata da GLUT ad ogni ciclo di rendering (quindi praticamente sempre). È utilizzata per eseguire compiti in background o per sostenere animazioni nel caso si generino altri eventi all'interno della finestra.

glutPostRedisplay()

Alcuni eventi possono chiedere l'esecuzione della display callback, allora con questa funzione imposto un flag per indicare che voglio invocare il refresh. Al termine del ciclo se il flag è impostato GLUT invocherà il refresh.

glutSolidTeapot(val)

La funzione disegna una teiera (@.@).