

La rivista che ti insegna a programmare

Utilizzare le OpenGL

.NET

Sviluppare applicazioni per Windows XP Media Center editing

DB

Microsoft SQL Server: un sistema sicuro?

ASP

Trasferire una sessione da ASP ad ASP .NET

SICUREZZA

Buffer overflow e dintorni

LINUX

C# e mono: unit test

Provati per voi
Netscape 8

Editor PHP a confronto: Zend Studio 4.0 e PHP Designer 2005



GRUPPO EDITORIALE
INFOMEDIA
THE SOFTWARE SIDE OF YOUR MIND

Sped. abb. post. 45% articolo 2 comma 20/b legge 662/96 - Filiale di Bologna



- Linux:** • Costruire una propria distribuzione live
- PHP:** • Emulazione di stored procedure con PHP
- Linux:** • Sviluppiamo il nostro progetto Open Source

di Roberto Casarini
rcasarini@infomedia.it

“Giocare” con le OpenGL

Ricordo come da bambino passassi ore ed ore di fronte al computer alle prese con i miei videogiochi preferiti nel tentativo di scoprirne trucchi e segreti per una rapida vittoria. Non potevo immaginare che solo qualche anno più tardi avrei saputo barare spudoratamente avvalendomi delle mie conoscenze informatiche, ma questa è un'altra storia...

Chi non ricorda la prima versione di Arkanoid? Oppure il mitico Tetris? E stiamo parlando di due dimensioni a 16 colori! La grafica di quegli anni, non paragonabile a quella odierna, sebbene poco realistica era pur sempre coinvolgente.

Per quanto mi riguarda alla passione per i videogame si è aggiunta con il tempo la passione per la programmazione e la grafica 3D.

Quanti tra voi avrebbero voluto implementare una variante del proprio gioco preferito, curandone tutti i dettagli trascurati? Io ero uno di questi.

Dietro lo sviluppo degli attuali videogame ci sono molti aspetti complessi.

Se all'epoca era possibile realizzare autonomamente un videogioco elementare in 2D, oggi risulterebbe una impresa ardua cercare di realizzare un prodotto al passo con i tempi. Le software house specializzate in videogiochi sono aziende molto articolate, dove a fianco dei programmatori sono presenti professionalità diversificate e necessarie al processo di sviluppo, tra le quali matematici, ingegneri fisici, grafici, addetti al suono, fino ad arrivare ai doppiatori o agli psicologi: un vero cast da film.

Per programmare oggi un videogioco, la soluzione migliore è senza dubbio avvalersi di strumenti mirati, come *3d Game Studio* (a cui dedicheremo spazio nei prossimi numeri di DEV).

Lo speciale di questo mese, invece, consentirà a chi ha una passione per la grafica e i videogiochi di poter iniziare ad addentrarsi nel mondo della grafica 3D. Per farlo, abbiamo scelto come interfaccia software le librerie OpenGL, che rappresentano un veloce ed efficiente strato di comunicazione con l'hardware grafico.

Il termine “open” talvolta erroneamente associato all'open source, testimonia l'intenzione degli autori di creare uno standard aperto in grado di favorire la diffusione della libreria verso altri sistemi operativi, al contrario delle DirectX che sono cablate su Windows.

Prima di congedarmi, e lasciarvi ad una immersione nel mondo 3D sintetico, un cenno al mini sondaggio del mese scorso “Quale linguaggio di programmazione usate maggiormente e quale vorreste approfondire, impiegare o conoscere?” è d'obbligo.

Numerose e-mail di risposta sono giunte in redazione, e pur sapendo di non poter ritenere il campione statisticamente significativo (ci vorrebbero decine e decine di migliaia di risposte per un campione rappresentativo dell'intero panorama italiano), sintetizzare il risultato ci sembra comunque opportuno.

Come prevedibile, VB è risultato il linguaggio più impiegato e conosciuto sul posto di lavoro. A seguire, e con mio grande stupore non proprio a ruota, un po' meno utilizzati risultano essere linguaggi come Java e quelli per lo sviluppo Web. Insomma una realtà in cui il connubio tra Basic e desktop continua a rappresentare una buona parte dello sviluppo software nostrano.

Roberto Casarini



Lo speciale di questo numero: Utilizzare le OpenGL

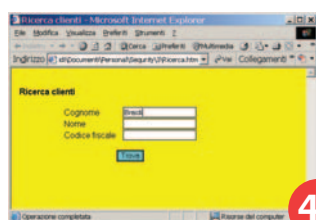
- 17 Utilizzare le OpenGL (PARTE 1) di Mauro Pesci
- 21 Utilizzare le OpenGL (PARTE 2) di Mauro Pesci
- 25 Utilizzare le OpenGL (PARTE 3) di Mauro Pesci
- 32 Utilizzare le OpenGL (PARTE 4) di Mauro Pesci

Programmazione

- 38 ASP .NET –Trasferire una sessione da ASP ad ASP .NET di Gian Maria Ricci
- 42 Cellulari –Il protocollo AT+ dei telefoni cellulari. La classe PhonebookManager in azione (PARTE 18) di Vito Vessia
- 49 Database –Microsoft SQL Server: un sistema sicuro? di Lorenzo Braidì
- 53 C# –Utilizzare le API di shell in C# di Gian Maria Ricci
- 56 .NET – Sviluppare applicazioni per Windows XP Media Center Edition di Lorenzo Vandoni
- 59 Sicurezza –Buffer overflow e dintorni (PARTE 4) di Gianluca Insolubile
- 65 Web –Emulazione di stored procedures con PHP di Max Arcidiacono
- 69 Visual Objects – VO e XML (PARTE 2) di Fabio Cresta

Rubriche

- 6 Spazio Libri –a cura di Marco Iannaccone
- 8 Extended Mode –Worms 4 Mayhem a cura di Stefano Arcidiacono
- 10 Developer's Tools –a cura di Roberto Casarini
- 98 Horizon –Le tre leggi –di Luigi Morelli



Sei agli inizi? Non riesci a risolvere i primi problemi? Niente paura! Iscriviti alle mailing-list Infomedia!

programmo_subito@infomedia.it
e linux-it@infomedia.it

I listati e i programmi di utilità che corredano gli articoli sono prelevabili gratuitamente dal sito <http://www.infomedia.it/DEV> oppure da <ftp://ftp.infomedia.it/pub/DEV/Listati/>



Programmo Subito

Numero 85

74 **Livello 2** Rappresentare i propri dati con Qt – di Fabrizio Angius

79 **Livello 2** Generare numeri casuali – di Giuseppe Cornacchia

Linux Subito

Numero 64

81 **Livello 2** Open source in pratica (PARTE 3) – di Alessio Saltarin

86 **Livello 3** Costruire una propria distribuzione live (PARTE 2)
di Corrado Ignoti

92 **Livello 1** C# e mono: unit test (PARTE 1) – di Gianluigi Spagnuolo

In Prova

Editor PHP a confronto:
Zend Studio 4.0 e PHP Designer 2005

di Paolo De Nictolis

11

Netscape 8.0.1

di Mauro Iannaccone

15

Ai lettori

Doktor Morb a cura di Luigi Morelli
Denti del giudizio

9

J2ME World a cura di Stefano Sanna
J2ME e sistemi di sicurezza: una esperienza

41

Italian Free a cura di Gianluca Masina
Mnz Cad

48

E-ducational a cura di Luigi Morelli
Base 5

52

Free Reading a cura di Lorenzo Vandoni
Aspect-Oriented Programming

55

www.infomedia.it - The Software Side Of Your Mind

Dev.

Developing
Software
Solutions

Numero 132
Settembre 2005
Anno XIII

Direttore Responsabile

Marialetizia Mari (mmari@infomedia.it)

Coordinamento Editoriale

Roberto Casarini (rcasarini@infomedia.it)

Collaboratori

Fabrizio Angius, Stefano Arcidiacono,
Max Arcidiacono
Lorenzo Braidì, Giuseppe Cornacchia
Fabio Cresta, Paolo De Nictolis
Marco Iannaccone, Corrado Ignoti
Gianluca Insolubile, Gianluca Masina
Luigi Morelli, Mauro Pesci
Gian Maria Ricci, Alessio Saltarin
Stefano Sanna, Gianluigi Spagnuolo
Lorenzo Vandoni, Vito Vessia

Grafica e Impaginazione

Manola Greco (mgreco@infomedia.it)

**GRUPPO EDITORIALE
INFOMEDIA**

Gruppo Editoriale Infomedia srl
Via Valdera P.116 56038 Ponsacco (Pi)
Tel 0587 73 64 60 (r.a.)
Fax 0587 73 22 32
www.infomedia.it - red_dev@infomedia.it

Direzione

Natale Fino (nfino@infomedia.it)

Divisione Libri

Lisa Vanni (book@infomedia.it)

Marketing & Advertising

Tel. 0587 73 64 60
marketing@infomedia.it

Supporto Sistemistico

webmaster@infomedia.it

Amministrazione

Sara Mattei (amministrazione@infomedia.it)

Segreteria

Enrica Nassi (info@infomedia.it)

Stampa

Betagraf Spa - Fano di Argelato (BO)

Distribuzione

Parrini & C. (Roma)

Ufficio Abbonamenti

Tel. 0587 73 64 60
Fax 0587 73 22 32
abbonamenti@infomedia.it
online@infomedia.it

Si prega di inviare i comunicati stampa e
gli inviti stampa per la redazione all'indi-
irizzo: comunicatistampa@infomedia.it

La riproduzione con qualsiasi mezzo di
illustrazioni e di articoli pubblicati sulla rivista,
nonché la loro traduzione, è riservata e non
può avvenire senza espressa autorizzazione
dell'editore. I manoscritti e le illustrazioni
inviati alla redazione non saranno restituiti
anche se non pubblicati e l'editore non si
assume alcuna responsabilità per il caso che
si tratti di esemplari unici.

"DEV. Developing Software & Solutions"
è una rivista del Gruppo Editoriale Infomedia S.r.l.
Direzione e Amministrazione
Via Valdera P., 116 - Ponsacco
Registrazione presso il Tribunale di Pisa
n. 21 del 25.11.1999

Contenuto pubblicitario inferiore al 45%

Utilizzare le OpenGL

Imparare a programmare le OpenGL non è un'impresa. È proprio la sua semplicità d'uso che ne ha fatto la libreria grafica più utilizzata contando su implementazioni in Windows, Linux, Mac ed alcuni dispositivi portatili. Lo scopo di questo speciale è fornire un'introduzione alle caratteristiche fondamentali della libreria.

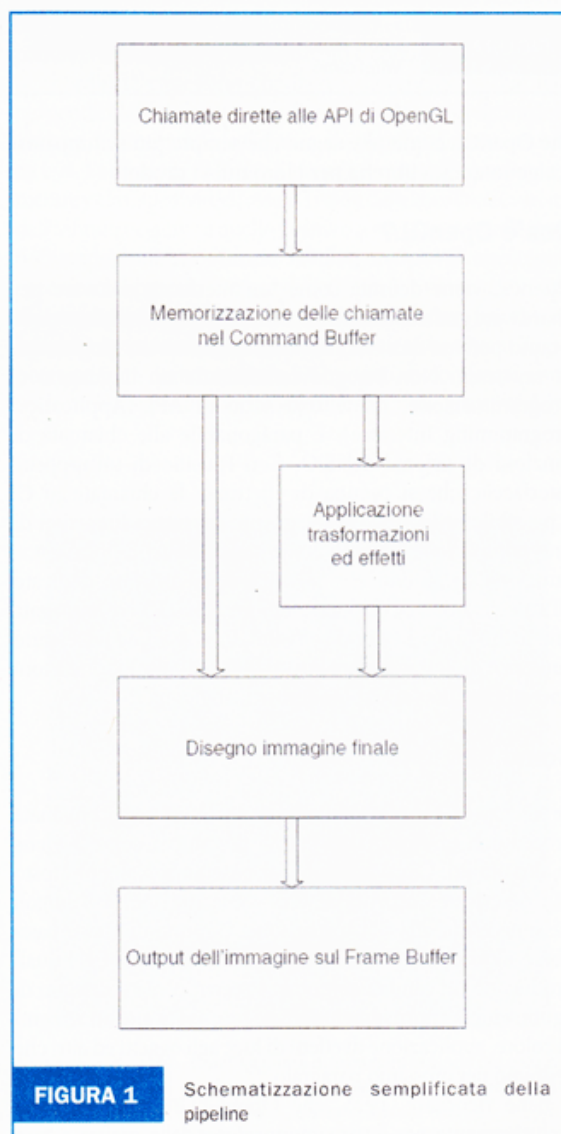
di Mauro Pesci > mpesci@infomedia.it

Con l'avanzamento tecnologico, per lo sviluppo dei videogiochi vengono utilizzate schede acceleratrici sempre più potenti. Ma come domare tutta questa potenza per utilizzarla per i nostri scopi?

Numerose librerie permettono di utilizzare le caratteristiche offerte dall'hardware dei nostri computer, tra queste spiccano i nomi dei colossi quali Direct3D e OpenGL. L'idea di questo speciale è fornire una presentazione completa e ricca di esempi delle funzionalità di base di OpenGL illustrandone le reali capacità e la semplicità d'uso.

Un po' di storia

Originariamente chiamata IRIS GL non era che una libreria per la programmazione della grafica 2D. Estesa in seguito verso il nuovo mondo del 3D venne impiegata sulle macchine proprietarie dell'azienda che disponevano di vari tipi di hardware dedicato, per esempio, per l'esecuzione dei calcoli ultra veloci delle trasformazioni matriciali o per la gestione del display. Negli anni, Silicon Graphics, ha sentito la necessità di agevolare l'utilizzo della IRIS GL su altre macchine favorendo la portabilità della libreria. Nasce così la prima versione di OpenGL nel Luglio 1992. Il termine "open" rappresentava, appunto, l'intenzione di creare uno standard aperto in grado di favorire la diffusione della libreria verso altri sistemi operativi. Quando l'hardware ad alte prestazioni rappresentava un lusso che poche aziende potevano permettersi OpenGL veniva impiegata per applicazioni CAD, ma dopo la conseguente diminuzione dei costi, le schede acceleratrici 3D divennero uno standard alla portata di tutti. Si aprirono così le porte a tantissime altre applicazioni della libreria specialmente al mondo dei videogiochi. Con il tempo gli sviluppatori furono sempre più attratti da OpenGL per via della sua semplicità e della facilità con cui è possibile portare il codice su altre piattaforme tanto che, attualmente, esistono in commercio un'infinità di prodotti che le utilizzano, molta documentazione ed esempi. La maggior parte delle schede video moderne implementano a livello hardware, sia le direttive Direct3D



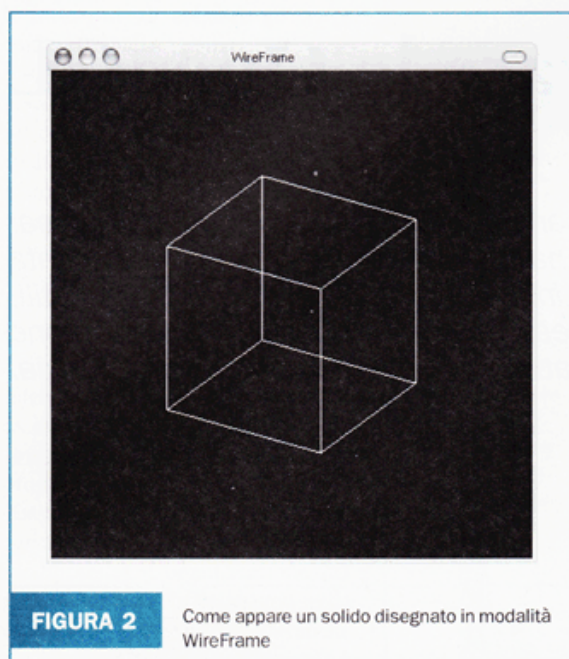


FIGURA 2 Come appare un solido disegnato in modalità WireFrame

che OpenGL e questo è segno che sempre più sviluppatori scelgono questa libreria per i loro nuovi prodotti.

Cos'è OpenGL?

OpenGL viene definita come "un'interfaccia software per l'hardware grafico" e costituisce una libreria portabile, veloce e tanto potente da avvicinarsi alla qualità di un programma di ray-trace. Non bisogna considerarla un linguaggio di programmazione, ma è una serie di API (Application Programming Interface). È paragonabile alle chiamate di funzioni di sistema del S.O. Con l'ausilio di un'apposita interfaccia, che si occupa di effettuare le chiamate in C, è possibile utilizzarle, praticamente, in tutti i linguaggi di programmazione esistenti come Visual Basic, Delphi, C#. Funziona su sistemi che possiedono l'hardware dedicato all'accelerazione delle operazioni grafiche 3D ed impiegata pressoché in ogni campo software dove si rende necessario l'utilizzo di tali funzionalità, dai CAD, alle applicazioni scientifiche, fino ai più moderni videogiochi.

Come funziona

Le API OpenGL sono caratterizzate da un sistema di chiamate procedurali piuttosto che di tipo descrittivo come avviene in altri ambiti. Ciò significa che il programmatore, piuttosto che descrivere l'aspetto di una scena, ne indica i singoli passi necessari alla sua costruzione. Ognuno di questi passi non è altro che una chiamata alle funzioni OpenGL i quali producono un cambiamento nella scena, come il disegno di figure semplici normalmente chiamate *primitive*, cambiamenti di colore, applicazione di effetti di luce agli oggetti ed altri che vedremo nel prossimo paragrafo.

È bene ricordare, però, che OpenGL è stata concepita esclusivamente per la programmazione della grafica e non

possiede alcuna funzionalità che permetta di gestire l'interfaccia grafica del sistema operativo, o anche i dispositivi di I/O come i Joystick o l'audio. Il problema può essere aggirato affiancando, nelle proprie applicazioni, altre librerie dedicate DirectX comprese. Ci chiediamo ora come vengano gestiti i "passi" ai quali è stato dato accenno poco fa. Entra in gioco la famosa *pipeline* ossia un meccanismo di elaborazione, a stadi successivi, di tutti i comandi impartiti.

La **Figura 1** mostra uno schema semplificato dei moduli della *pipeline*. Si può notare come i dati relativi ai vertici, alla trasformazioni, all'illuminazione e così via sono originariamente memorizzati all'interno di un buffer fino al lancio di uno speciale comando di svuotamento o *flush*.

Quando ciò avviene tutte le operazioni impostate sono eseguite in un solo colpo e l'immagine finale viene prodotta. Una tale scelta architettonica è stata adottata al fine di ottimizzare i processi di comunicazione con l'hardware. In ogni stadio della *pipeline* sono eseguiti complessi calcoli matematici atti a trasformare i dati "grezzi" immessi dal programmatore. Non deve nascere alcun motivo di apprensione in chi teme di dover rispolverare i testi di matematica. Programmare le OpenGL non significa necessariamente penetrare nei meandri della computer-grafica e presto ve ne renderete conto. A differenza di quanto avviene per le Direct3D, infatti, basteranno poche righe di codice per ottenere risultati sorprendenti.

Un panoramica degli effetti

Cerchiamo di dare un'idea delle caratteristiche essenziali di OpenGL, quelle funzionalità – o *effetti* – necessari alla costruzione delle nostre scene dalle più schematiche alle più realistiche. Il primo termine da conoscere è *Rendering* che indica l'azione di trasformare la descrizione geometrica di un

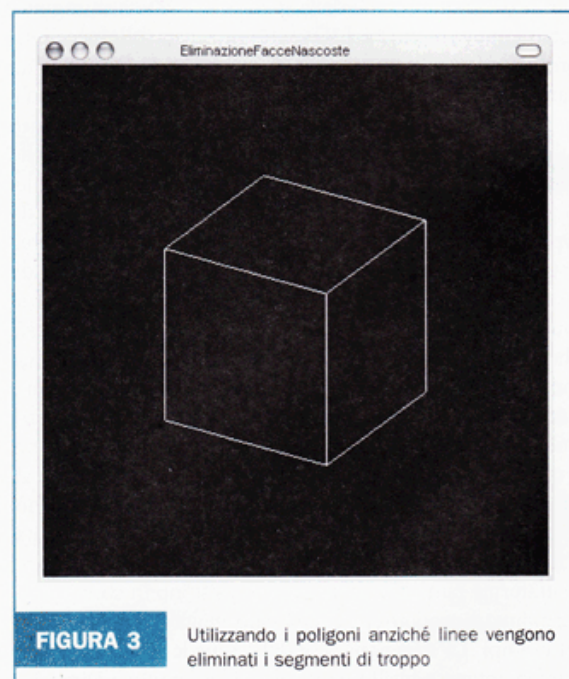


FIGURA 3 Utilizzando i poligoni anziché linee vengono eliminati i segmenti di troppo

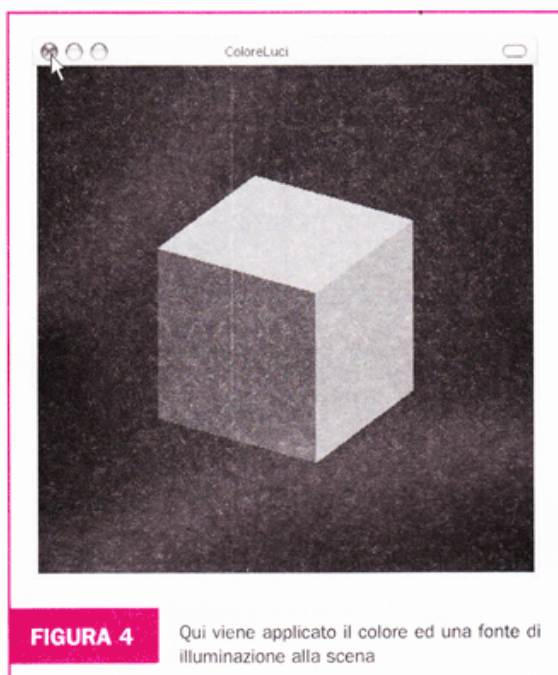


FIGURA 4 Qui viene applicato il colore ed una fonte di illuminazione alla scena

oggetto tridimensionale in un'immagine 2D. Per tridimensionale o 3D si intende un oggetto rappresentato tramite le misure di altezza, larghezza e profondità. Purtroppo però sappiamo che i display esistenti possono visualizzare solamente immagini che possiedono le sole due dimensioni di altezza e larghezza. Come è possibile, allora, ottenere l'illusione della profondità? Semplicemente operando come hanno fatto da secoli i pittori i quali pur avendo a disposizione solo una tela o un foglio di carta, tramite lo studio della *prospettiva*, hanno dipinto scene che possiedono un alto livello di realismo. Osservando l'immagine in **Figura 2** ci rendiamo subito conto come essa sia sufficiente per dare una certa illusione della terza dimensione. Purtroppo, però, un oggetto così rappresentato può causare problemi di percezione. Sapete, infatti, stabilire se la faccia anteriore del cubo è orientata verso l'alto o il basso?

Per ovviare a questo inconveniente bisogna eliminare le linee che non sono necessarie. In **Figura 3** è mostrato lo stesso oggetto trattato tramite la procedura di rimozione delle facce nascoste o *hidden surfaces removal*. Notate come ora il cubo appare solido. È possibile andare oltre il semplice disegno di linee. Aggiungendo del colore ed un'adeguata fonte di illuminazione l'oggetto in **Figura 4** guadagna un'ulteriore livello di realismo. Infine anziché colorare le facce dell'oggetto con una tinta unica, OpenGL, permette di associare, ad ognuna di esse, un'immagine o *texture*. Possiamo, così, fare in modo che il nostro oggetto dia l'impressione di essere costituito da un materiale. La **Figura 5** mostra un'immagine di una scena dove sono state utilizzate diverse *texture*. Esistono altri effetti e sono *Fogging*, *Blending* ed *Antialiasing*. L'effetto *Fog* o nebbia simula il fenomeno atmosferico che conosciamo bene, il quale fa sì che gli oggetti siano offuscati in funzione della loro distanza dal punto di osservazione. *Blending* significa mescolanza o sovrapposizione di più immagini. Applicando questo effetto agli oggetti di una scena possiamo fare in modo che essi

risultino trasparenti. Le immagini visualizzate sullo schermo sono formate da punti discreti perciò quando si disegnano, ad esempio, figure geometriche tramite linee esse appariranno come dentellate. L'*Antialiasing* serve a sfumare leggermente i contorni delle linee dotandole di un'apparenza più delicata.

Principi di programmazione 3D

Per descrivere gli oggetti in tre dimensioni è necessario disporre di un sistema di riferimento. Esso deve permettere di specificare la posizione, le misure ecc. Disegnare un punto sullo schermo significa specificare la posizione in termini di riga e colonna e queste informazioni prendono il nome di *coordinate*. Prendendo in considerazione il monitor del computer con una risoluzione di 800x600 il suo punto centrale è rappresentato dalla coppia di coordinate (400,300), cioè 400 pixel a destra e 300 pixel in basso rispetto all'angolo superiore sinistro dello schermo. È necessario ora introdurre un nuovo concetto, quello di *sistema di coordinate*. Un *sistema di coordinate* è un metodo di riferimento che permette di identificare univocamente un dato punto in un piano o nello spazio. Il più utilizzato e conosciuto è il sistema di coordinate cartesiano che basa i riferimenti su due rette perpendicolari chiamate asse X e Y e dove ogni misurazione avviene determinando la distanza, di un punto preso in esame, da essi. La distanza dall'asse X prende il nome di *coordinata x*, mentre y è la rispettiva dell'asse Y. I valori delle coordinate variano dall'infinito negativo a quello positivo come mostrato in **Figura 6**. Quando si utilizzano funzioni di disegno OpenGL è necessario specificare il proprio sistema di coordinate e come proiettarlo sul dispositivo fisico. Le finestre, infatti, sono fisicamente misurate in termini di *pixel*, ed è per questo che OpenGL, per poter operare correttamente, ha bisogno di conoscere come tradurre le coppie di coordinate cartesiane nel corrispondente

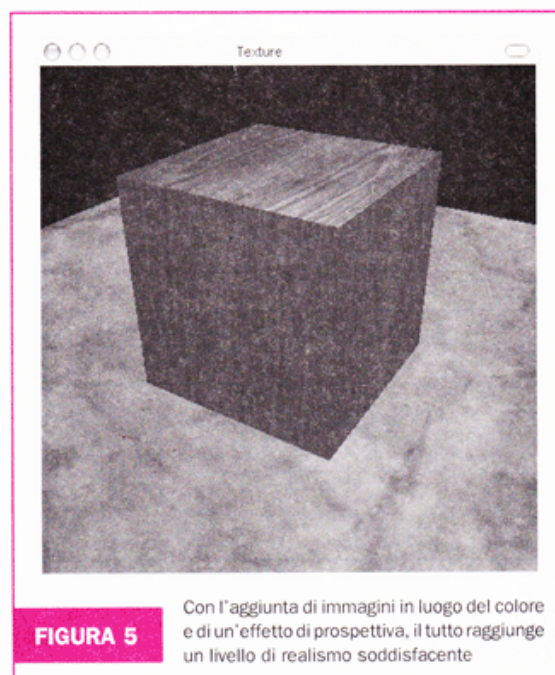


FIGURA 5 Con l'aggiunta di immagini in luogo del colore e di un'effetto di prospettiva, il tutto raggiunge un livello di realismo soddisfacente

sistema di coordinate dello schermo. Otteniamo questo risultato delineando l'area che viene occupata dalla finestra sul piano cartesiano, la quale prende il nome di *clipping-area*. La **Figura 7** mostra come una *clipping-area* rappresenta, in definitiva, l'intervallo di coordinate x e y che sono visibili all'interno della finestra. Non è necessario però che la *clipping-area* sia di dimensioni pari a quelle della finestra. Se ne abbiamo specificato una che va dai punti 0 e 100 per ambedue le coordinate su di una finestra di 200 x 200 pixel, l'output sarà ridimensionato e scalato per riempirla opportunamente. In questo modo ogni incremento di 1 unità nel piano cartesiano corrisponderà a 2 pixel sulla finestra. Spesso, però, si ha la necessità di non visualizzare l'output sull'intera finestra ed entra così in gioco la *viewport*. Dimensionando opportunamente la *viewport* è possibile rimpicciolire l'output oppure visualizzare solo un particolare di esso. Estendiamo ora il concetto di sistema di coordinate al mondo 3D aggiungendo un terzo asse chiamato Z e posizionato in direzione dell'osservatore. Per questo sistema valgono gli stessi principi esposti poco fa tranne per il fatto che le coordinate vengono espresse per mezzo della terna (x,y,z) .

Dove reperire materiale sul Web

Esiste un'infinità di materiale su Internet; numerosissimi siti presentano sezioni piene di tutorial, download dei file sorgenti, link e molto altro per cui vale la pena dare un'occhiata in giro per approfondimenti. Comunque, sia gli esperti che i principianti fanno riferimento ai seguenti siti in particolare:

- www.opengl.org: il sito ufficiale di OpenGL contenente praticamente tutte le novità di questo mondo oltre a svariate documentazioni, applicazioni, giochi di esempio e forum
- nehe.gamedev.net: un sito eccellente che presenta, tra le altre

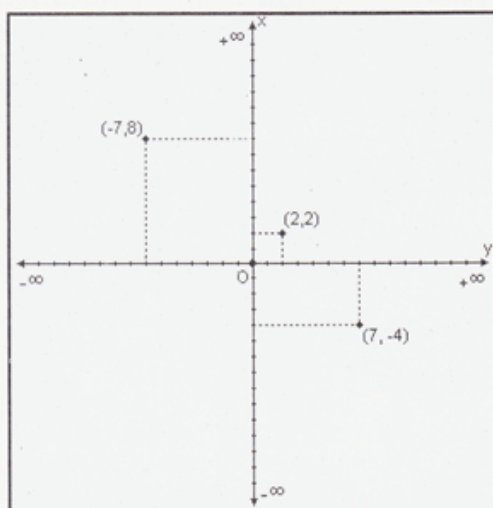


FIGURA 6 Un esempio di sistema di coordinate cartesiano

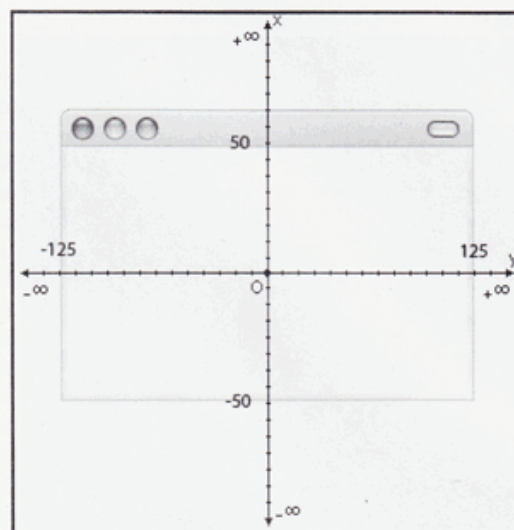


FIGURA 7

Notate come una *clipping-area* rappresenta gli intervalli di coordinate che sono visibili all'interno della finestra

cose, una serie di tutorial che coprono tutti gli aspetti della programmazione OpenGL ed alcuni template per utilizzarle in svariati linguaggi di programmazione come C#, Delphi

- www.gamedev.net: dedicato alla programmazione dei videogiochi in genere

Conclusioni

In questo primo articolo, molto teorico, abbiamo esposto in maniera concisa quanto c'è da sapere per iniziare la nostra avventura nello straordinario mondo della grafica 3D in tempo reale. Proseguiremo, nel prossimo articolo, con l'introduzione della libreria GLUT la quale permette di creare, con poche istruzioni, applicazioni complete e portabili per tutte le piattaforme Windows, Linux e Mac OS. Sul sito FTP potrete anche trovare i programmi che hanno generato le immagini qui presenti, compresi i pacchetti di installazione della libreria GLUT necessari alla compilazione dei sorgenti e all'esecuzione dei programmi.

Bibliografia

- [1] Richard S.Wright jr. & Benjamin Lipchak, "OpenGL SuperBible – Third Edition", SAMS, 2005, ISBN 0-672-32601-9

Mauro Pesci

Si occupa di Analisi e Sviluppo del software in vari ambienti e piattaforme. Ha lavorato per diverse aziende nel settore, mentre ora è impegnato in un progetto presso la HP di Roma. Attualmente studia Informatica presso l'università di Tor Vergata (Roma).

Utilizzare le OpenGL

Prima di analizzare l'utilizzo di OpenGL, descriviamo la libreria GLUT, che fa risparmiare molto tempo evitando di affrontare aspetti relativi alla programmazione specifica per un particolare sistema operativo.

di Mauro Pesci > mpesci@infomedia.it

Come si è detto, OpenGL è una sofisticata API per la creazione di grafica 3D. Gli oltre 300 comandi disponibili permettono di generare scene molto complesse e realistiche. È normale quindi stupirsi di fronte alla totale assenza di funzioni dedicate alla creazione di finestre o alla gestione dello schermo. Non è presente nemmeno un sistema per ricevere comandi da tastiera, né per rilevare l'attività del mouse o utilizzare l'audio. La motivazione di una scelta simile deriva dalla volontà dei creatori della libreria di renderla il più possibile indipendente dalla piattaforma utilizzata. L'aggiunta di una sola funzione per l'apertura delle finestre, infatti, comporterebbe un'implementazione basata su chiamate dirette alle API del sistema operativo ospite, il che risulta in conflitto con l'idea originale.

Alcune specifiche preliminari

La prima regola che gli sviluppatori di OpenGL hanno seguito per mantenere un alto livello di portabilità è stata quella di creare uno standard relativo alla nomenclatura delle funzioni e alla dichiarazione dei tipi di dato. Per le prime viene utilizzato un sistema che permette di identificare a colpo d'occhio la libreria alla quale una funzione appartiene, il comando, il numero degli argomenti che richiede e il tipo di dato utilizzato. In **Figura 1** è mostrato un esempio di funzione dove è visibile il suffisso `gl` il quale indica che la funzione appartiene alla libreria OpenGL, il nome del comando `Vertex` il numero degli argomenti pari a 3, e la `f` finale che sta per float, il tipo di parametri che deve avere la funzione. È importante tener presente un altro fattore quando si usano le funzioni OpenGL; la maggior parte dei compilatori C++ considera i valori numerici di tipo `double`, di conseguenza è necessario indicare con il simbolo `f` ogni parametro passato ad una funzione che accetta dati di tipo float, come nell'esempio:

```
glColor4f(0.0f, 1.0f, 0.0f);  
glVertex3f(0.0f, 10.0f, 0.0f);
```

Quando si scrive un qualunque programma vengono normalmente utilizzati tipi di dato che sono definiti

dal compilatore, ma che in qualche modo sono legati all'architettura della macchina. Non possiamo, infatti, essere sicuri che un dato di tipo `float` o `int` del C sia trattato o abbia lo stesso significato su piattaforme diverse. Questo conduce, ovviamente, a problemi di portabilità del codice; di conseguenza OpenGL è stata dotata di un proprio insieme di tipi di dato da poter utilizzare con la sicurezza che questi saranno interpretati alla stessa maniera su piattaforme distinte. Nel **Riquadro 1** è presente un elenco di tutti i tipi definiti, il loro corrispondente in C, la rappresentazione interna standard e i suffissi, identici a quelli utilizzati nella nomenclatura delle funzioni. Come si può vedere anche per i tipi di dato viene utilizzata una struttura dei nomi standard, cioè il prefisso `GL` per distinguerli dai tipi definiti dal C. Il significato della maggior parte di loro è chiaro e noto, ma ne sono stati aggiunti altri molto utili. Innanzitutto il tipo `GLbitfield` è utile per quelle variabili che devono contenere campi di bit, `GLenum` serve per le variabili enumerate, mentre `GLclamp`, con suffisso `f o d`, ne rappresenta una in virgola mobile a singola o doppia precisione i cui valori possono variare solo tra 0.0 e 1.0. Ciò avviene poiché molte funzioni accettano valori che variano in questo intervallo. Per esempio la funzione `glColor3f` vuole, come parametri, i livelli RGB di un colore, ma questo caso l'intensità della singola componente è definita nell'intervallo 0.0 e 1.0 non tra 0 e 255 come siamo abituati.



FIGURA 1

Ecco un esempio di nomenclatura delle funzioni OpenGL.

La libreria GLUT

Per facilitare l'apprendimento delle OpenGL ed accelerare i processi di sviluppo venne creata la libreria ausiliaria AUX, ma a causa di alcuni problemi di implementazione e compatibilità fu ben presto sostituita da GLUT, scritta dalla stessa Silicon Graphics. La sigla GLUT sta per *OpenGL utility toolkit*, da non confondere con *GLU - OpenGL utility library*. GLUT è una libreria multiplatforma che contiene molte funzioni per la creazione delle finestre, di menu pop-up e permette la gestione della tastiera, del joystick e del video in modalità *full screen*. Anche gli sviluppatori senza esperienza di programmazione della GUI in un particolare sistema operativo sono, così, in grado di realizzare programmi completi, funzionali e completamente portabili. Come la vecchia AUX, però, non è in grado di far fronte alle esigenze di sviluppo di applicazioni commerciali, ma è eccellente a livello didattico, per testare le nostre idee o per creare programmi di presentazione. Per queste ragioni sarà utilizzata in tutto il corso di questo speciale. I riferimenti per scaricare GLUT possono essere trovati direttamente dal sito www.opengl.org.

Il nostro primo programma con GLUT

Possiamo finalmente, dopo tante parole introduttive, iniziare ad utilizzare concretamente queste librerie, perciò studiamo come costruire un eseguibile completo che apre una finestra vuota. Questo codice può essere considerato come un template da utilizzare in tutti i programmi che realizzeremo. Osservando il codice nel **Listato 1** analizziamo, quindi, ogni funzione. Iniziamo dal `main()` dove troviamo la chiamata seguente

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
```

contenente un parametro composto da diverse flag. La prima, `GLUT_RGBA`, dice a GLUT di inizializzare una finestra che supporti una modalità video a 32 bit – la sigla RGBA sta per Red, Green, Blue ed Alpha dove Alpha rappresenta il canale di trasparenza. `GLUT_DOUBLE` chiede che vengano utilizzati due buffer per la memorizzazione delle immagini finali. Questo meccanismo noto con il nome di *double-buffering* permette di utilizzare un'area di memoria nascosta dove hanno luogo le operazioni di disegno dei vari oggetti, rimanendo dietro le quinte. Alla fine l'immagine viene scambiata in un colpo solo con quella visibile sulla finestra evitando, così, il fastidioso e noto effetto di tremolio dell'immagine. L'impostazione opposta è la `GLUT_SINGLE`; in questo caso ogni comando di disegno provocherà un cambiamento immediato sull'immagine visualizzata nella finestra. Per effettuare lo scambio di buffer appena menzionato deve essere lanciata la funzione `glutSwapBuffers()` che vedremo tra poco. A seguire troviamo la chiamata

```
glutInitWindowSize(640, 480);
```

la quale non fa altro che comandare a GLUT di impostare una dimensione di 640 x 480 pixel per la finestra che successivamente verrà creata. Nella documentazione di GLUT sono menzionate anche funzioni per posizionare e

LISTATO 1

Il codice necessario per strutturare un'applicazione di base con GLUT. Il risultato dell'esecuzione di questo programma è l'apertura di una finestra vuota

```
void SetupRC()
{
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
}

void ChangeSize(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
}

void RenderScene()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glutSwapBuffers();
}

void main()
{
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowSize(640, 480);
    glutCreateWindow("Il primo programma con GLUT");

    SetupRC();

    glutReshapeFunc(ChangeSize);
    glutDisplayFunc(RenderScene);
    glutMainLoop();
}
```

spostare la finestra sullo schermo o aprirne una in modalità a schermo intero. È arrivato, finalmente, il momento di creare la finestra e ciò può essere ottenuto tramite la chiamata

```
glutCreateWindow("Il primo programma con GLUT");
```

che setta anche il testo visualizzato come titolo.

L'ultima riga di codice

```
glutMainLoop();
```

fa in modo che GLUT dia inizio ad un ciclo Idle che "tiene in vita" il programma fino al click sul pulsante di chiusura della finestra. Fin qui, le istruzioni viste sono, da sole, sufficienti al corretto funzionamento di un programma minimo, che non svolge alcuna operazione, ma si limita a visualizzare una finestra vuota con sfondo nero. Nelle nostre applicazioni, presentazioni o videogiochi abbiamo bisogno essenzialmente che:

- All'avvio dell'applicazione esista una procedura di inizializzazione dell'ambiente
- Ad intervalli regolari, il programma, disegni sulla finestra il fotogramma opportuno
- Che essa reagisca agli stimoli di ridimensionamento da parte dell'utente

La prima e la seconda condizione sono piuttosto ovvie e saranno realizzate dalle funzioni `SetupRC()` e `RenderScene()`. La terza, invece, significa che ogni volta un utente ridimensiona o riduce a icona la finestra il programma deve, in qualche modo, modificare le dimensioni dell'immagine visualizzata. Tale operazione sarà svolta dalla `ChangeSize()`. Osserviamo che non abbiamo a disposizione alcun metodo per conoscere a priori come gestire gli eventi di sistema. GLUT ancora una volta ci viene in aiuto risolvendo anche questo problema tramite alcuni comandi che prendono come parametro una nostra funzione e la collegano ad un particolare evento di sistema. Così, ogni volta che avviene un ridimensionamento verrà lanciata la funzione passata come argomento a

```
glutReshapeFunc((void (GLUTCALLBACK *func)(int width, int height));
```

Nel nostro caso avremo

```
glutReshapeFunc(ChangeSize);
```

dove la funzione `ChangeSize()` dovrà accettare due interi in luogo dei valori di larghezza e altezza assunti dalla finestra dopo il ridimensionamento. In pratica, come si vede nel listato, la funzione viene dichiarata in questa maniera

```
void ChangeSize(int w, int h);
```

Per i meno esperti puntualizziamo che i nomi della funzione e dei parametri non sono significativi e potrete cambiarli a vostro piacimento. La funzione di disegno, che ha il compito di produrre la scena, viene passata alla

```
glutDisplayFunc(void (GLUTCALLBACK *func)(void));
```

Nel nostro listato troviamo infatti

RIQUADRO 1

Funzioni che accettano *format string*

Tipo di dato	Rappresentazione interna	Tipo C	Suffisso letterale C
GLbyte	intero a 8bit	signed char	b
GLshort	intero a 16bit	short	s
GLint, GLsizei	intero a 32bit	long	l
GLfloat, GLclampf	virgola mobile a 32bit	float	f
GLdouble, GLclampd	virgola mobile a 64bit	double	d
GLubyte, GLboolean	intero senza segno a 8bit	unsigned char	ub
GLushort	intero senza segno a 16bit	unsigned short	us
GLuint, GLenum, GLbitfield	intero senza segno a 32bit	unsigned long	ul

LISTATO 2

La funzione che si occupa di regolare correttamente l'aspetto dell'output quando avviene un ridimensionamento della finestra

```
void ChangeSize(GLsizei w, GLsizei h)
{
    GLfloat aspectRatio;

    if (h == 0) h = 1;

    aspectRatio = (GLfloat) w / (GLfloat) h;

    glViewport(0, 0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if (w <= h)
        glOrtho(CV_SINISTRA, CV_DESTRA, CV_ALTO / aspectRatio, CV_BASSO / aspectRatio, CV_VICINO, CV_LONTANO);
    else
        glOrtho(CV_SINISTRA * aspectRatio, CV_DESTRA * aspectRatio, CV_ALTO, CV_BASSO, CV_VICINO, CV_LONTANO);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

```
glutDisplayFunc(RenderScene);
```

con `RenderScene()` dichiarata senza parametri. La funzione `glClearColor()` presente all'interno di `SetupRC()` indica a OpenGL quale colore deve essere utilizzato per la cancellazione del contenuto del buffer. Abbiamo scelto il colore nero impostando i valori delle componenti RGB a 0.0f. Nella funzione `RenderScene()` attuale troviamo due istruzioni, la prima delle quali,

```
glClear(GL_COLOR_BUFFER_BIT);
```

serve a cancellare il contenuto del *color buffer* quello, cioè, che contiene l'immagine renderizzata. In questa maniera abbiamo preparato il buffer ad essere riempito da un altro fotogramma della scena. Dopo aver effettuato tutte le operazioni di disegno si rende necessario visualizzare l'immagine prodotta copiando il contenuto del *color buffer* sull'area visibile della finestra e ciò può essere ottenuto richiamando la funzione `glutSwapBuffers()`. Questa funzione, internamente, richiama la `glFlush()` che determina l'elaborazione di tutte le istruzioni immagazzinate precedentemente nel *Command Buffer*, scatenando i processi di *rendering*.

Gestire la viewport

In quasi tutte le applicazioni l'utente ha la possibilità di ridimensionare la finestra dove il programma è in esecuzione. È nostra intenzione scrivere programmi che visualizzino l'output corretto in tutte le condizioni, sia quando l'utente ingrandisce la finestra a tutto

LISTATO 3

Come utilizzare il timer messo a disposizione da GLUT per eseguire le azioni di rendering ad intervalli regolari

```
void Timer(int v)
{
    glutPostRedisplay();
    glutTimerFunc(1, Timer, 1);
}

void main()
{
    ...
    ...

    glutTimerFunc(1, Timer, 1);

    ...
    ...
}
```

schermo sia quando la rimpiccolisce, l'output dovrà risultare opportunamente scalato. Nel **Listato 2** viene presentata la nuova versione della funzione `ChangeSize()`. La prima operazione effettuata è la chiamata a

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

I parametri `x` e `y` indicano la posizione dell'angolo in basso a destra, mentre `width` e `height` la larghezza e l'altezza dell'area, nella quale OpenGL produce l'output, all'interno della finestra. Solitamente, per far coincidere tale area con l'intera finestra, i parametri `x` e `y` sono impostati a 0 mentre `width` ed `height` assumono il valore dei parametri `w` e `h` passati da GLUT come di seguito

```
glViewport(0, 0, w, h);
```

Dopodichè bisogna impostare il *volume di clipping*. Nel precedente articolo si è parlato di *area di clipping* relativamente alle immagini a due dimensioni. Aggiungendo una ulteriore dimensione non ha più senso di parlare di aree; definiamo, così, il volume all'interno del quale vengono rappresentati gli oggetti della scena tramite la funzione

```
glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
        GLdouble near, GLdouble far);
```

I parametri `left` e `right` definiscono l'intervallo di coordinate lungo l'asse delle `x` entro il quale vengono visualizzate le immagini. `Bottom` e `top` definiscono lo stesso tipo di intervallo ma sull'asse delle `y`, mentre `near` e `far` sull'asse delle `z`. `glOrtho` indica anche, a OpenGL, di utilizzare un sistema di proiezioni di tipo ortografico dove, cioè, non esiste l'effetto di profondità tipico della prospettiva. Tutte le linee che compongono un oggetto presentano le stesse dimensioni sia che esse si trovino vicino all'osservatore oppure a grande distanza. Le altre istruzioni verranno discusse nei prossimi articoli e per il momento basta sapere che esse informano OpenGL su quali matrici applicare le impostazioni indicate.

Infine può accadere che, ridimensionando la finestra dell'applicazione, il rapporto tra larghezza ed altezza cambi in modo non desiderato deformando così, l'immagine finale. Calcolando però il rapporto tra le dimensioni della finestra in base ai valori passati alla funzione `ChangeSize()` siamo in grado di rendere l'output immune da eventuali distorsioni con questi comandi

```
aspectRatio = (GLfloat) w / (GLfloat) h;
```

```
if (w >= h)
    glOrtho(-100.0, 100.0, -100.0 / aspectRatio, 100.0 / aspectRatio, 1.0, -1.0);
else
    glOrtho(-100.0 * aspectRatio, 100 * aspectRatio, -100, 100, 1.0, -1.0);
```

Creare una semplice animazione

Difficilmente vorremo scrivere programmi che generano immagini statiche ed è per questo che impareremo a sfruttare un'altra caratteristica di GLUT che assolve degnamente a questo scopo. Esiste, infatti, un comando che ci permette di specificare quale nostra funzione lanciare ad ogni intervallo di tempo stabilito e presenta questa sintassi

```
void glutTimerFunc(unsigned int msec, void (*func)(int value), int value);
```

Il primo parametro `msec` specifica il numero di millisecondi che GLUT deve attendere prima di lanciare la funzione indicata nel parametro `(*func)`, mentre, `value`, permette di passare a questa un valore. Il **Listato 3** ne porta un esempio.

Qui la funzione `Timer` viene richiamata dopo 1 millisecondo provocando il ridisegno dell'intera scena tramite la funzione `RenderScene()` e un'ulteriore attivazione del timer.

Conclusioni

Questo articolo ha coperto tutte le nozioni di base necessarie all'utilizzo di GLUT per la realizzazione delle nostre applicazioni di test. Chi volesse approfondire l'argomento ed imparare ad utilizzare il Joystick o il mouse nei propri programmi, creare scritte 3D utilizzando i font di sistema e tanto altro, può fare riferimento alla documentazione presente sul sito FTP.

Bibliografia

- [1] Richard S.Wright jr. & Benjamin Lipchak, "OpenGL SuperBible - Third Edition", SAMS, 2005, ISBN 0-672-32601-9

Mauro Pesci

Si occupa di Analisi e Sviluppo del software in vari ambienti e piattaforme. Ha lavorato per diverse aziende nel settore, mentre ora è impegnato in un progetto presso la HP di Roma. Attualmente studia Informatica presso l'università di Tor Vergata (Roma).

Utilizzare le OpenGL

Siamo finalmente pronti per iniziare questa avventura nel mondo della grafica 3D. Inizieremo dall'uso delle figure più semplici fino ad arrivare alla trasformazione geometrica degli oggetti.

di Mauro Pesci > mpesci@infomedia.it

Ora che abbiamo a disposizione un modello di codice da utilizzare per scrivere velocemente nuovi programmi possiamo addentrarci nel mondo del 3D e fare la conoscenza delle funzionalità basilari OpenGL. Consiglio vivamente di studiare e compilare i sorgenti forniti a corredo degli articoli raccomandandovi di modificarli per ottenere risultati più complessi ma soprattutto per prendere familiarità con ogni singola istruzione.

Lavorare con le primitive

Le scene ed in particolare gli oggetti che si muovono in esse sono costituite da una composizione di figure bidimensionali elementari chiamate *primitive*. Possiamo paragonare, queste ultime, ai mattoncini i quali, opportunamente combinati ed aggregati tra loro, danno origine a tutte le costruzioni più complesse immaginabili.

Prendiamo il cubo come oggetto di esempio. Esso è costituito da sei facce, ognuna delle quali potrebbe essere disegnata da una primitiva di tipo quadrilatero, oppure, suddividendo il quadrilatero in due parti uguali, da due triangoli.

OpenGL può lavorare con diverse figure primitive, dai semplici punti ai poligoni; in totale ne esistono 10 tipologie diverse, riassunte nel **Riquadro 1**.

Ma quale sistema viene adottato per disegnare tali figure nello spazio? Il "minimo comun denominatore", per tutte le primitive, è il *Vertice*, un punto definito nello spazio rappresentato per mezzo delle sue coordinate x, y, z. La funzione OpenGL più utilizzata per specificare un vertice è la

```
glVertex3f(GLfloat x, GLfloat y, GLfloat z);
```

Di conseguenza, una *primitiva*, non è altro che un'interpretazione di una lista di vertici.

Ora che sappiamo quindi come indicare le coordinate dei punti nello spazio siamo in grado di prendere in esame ognuna delle tipologie di primitive.

Da ricordare che il sistema di coordinate della finestra è diverso

da quello utilizzato da OpenGL, per questo i rispettivi assi delle y hanno i versi invertiti; così mentre nel primo caso i valori positivi sono orientati verso il basso, nel secondo partono dal basso e vanno verso l'alto.

Punti

Diamo uno sguardo al seguente codice

```
glBegin(GL_POINTS);  
glVertex3f(0.0f, 0.0f, 0.0f);  
glEnd();
```

Osserviamo che le chiamate alle funzioni che descrivono i vertici devono essere racchiuse tra la coppia `glBegin()` e `glEnd()` e, settando il parametro opportuno, OpenGL

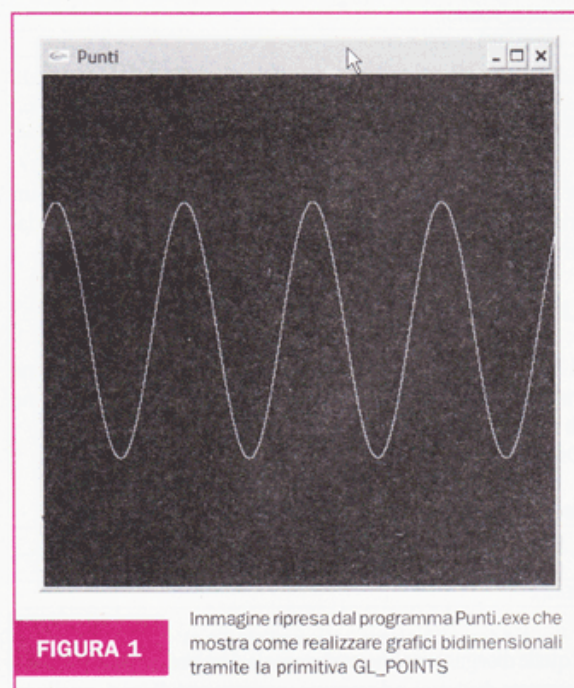


FIGURA 1

Immagine ripresa dal programma Punti.exe che mostra come realizzare grafici bidimensionali tramite la primitiva GL_POINTS

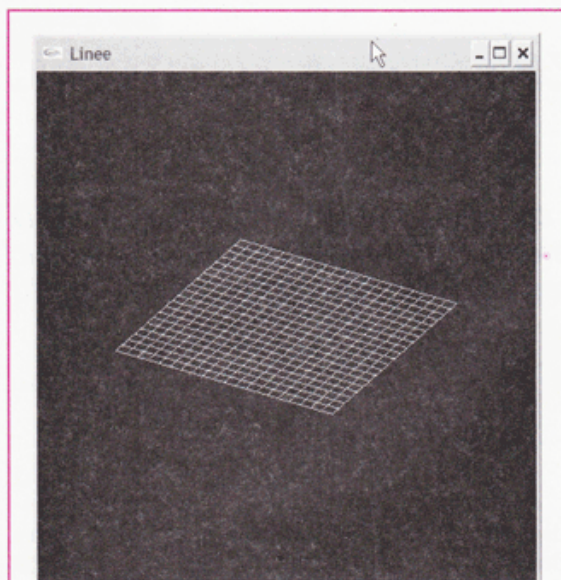


FIGURA 2 Disegno di un semplice piano tramite le linee

li interpreterà disegnando la figura voluta. In questo caso abbiamo passato `GL_POINTS` per disegnare semplici punti. Potreste essere indotti a pensare che nel caso si debba disegnare più di una figura sia necessaria una coppia `glBegin/glEnd` per ognuna. Ciò non farebbe altro che produrre notevoli rallentamenti durante l'esecuzione del programma. OpenGL, infatti, proprio in virtù del parametro passato alla funzione `glBegin` si rende conto di quanti vertici sono necessari per descrivere la figura. Scrivendo infatti

```
glBegin(GL_POINTS);
glVertex3f(0.0f, 0.0f, 0.0f);
glVertex3f(10.0f, 0.0f, 0.0f);
glVertex3f(5.0f, 10.0f, 0.0f);
glEnd();
```

verranno disegnati correttamente tre punti, ma non un triangolo. In **Figura 1**, è riportato l'output prodotto dal programma di esempio Punti.exe.

Linee

Il livello successivo ai singoli punti è quello di capire come visualizzare le linee. Specificando due vertici, della posizione di inizio e di fine, sarà disegnata una linea. Osservate il codice seguente

```
glBegin(GL_LINES);
glVertex3f(-10.0f, 0.0f, 0.0f);
glVertex3f(10.0f, 0.0f, 0.0f);
glEnd();
```

il quale disegna una linea orizzontale. Se è presente un numero scorretto di vertici, ad esempio una quantità dispari, quello in

più sarà semplicemente ignorato. Spesso, disegnando figure tramite linee contigue, potremmo trovarci di fronte a lunghe sequenze di vertici dove molti di essi sono semplicemente duplicati. Per disegnare un quadrato con questo sistema ad esempio saremmo costretti a procedere in questo modo

```
glBegin(GL_LINES);
glVertex3f(-10.0f, 0.0f, 0.0f); // Lato 1 punto 1
glVertex3f(10.0f, 0.0f, 0.0f); // Lato 1 punto 2
glVertex3f(10.0f, 0.0f, 0.0f); // Lato 2 punto 1
glVertex3f(10.0f, 10.0f, 0.0f); // Lato 2 punto 2
glVertex3f(-10.0f, 10.0f, 0.0f); // Lato 3 punto 1
glVertex3f(-10.0f, 10.0f, 0.0f); // Lato 3 punto 2
glVertex3f(-10.0f, 10.0f, 0.0f); // Lato 4 punto 1
glVertex3f(-10.0f, 0.0f, 0.0f); // Lato 4 punto 2
glEnd();
```

Notate, abbiamo indicato ben 8 vertici, il doppio di quelli veramente necessari.

Utilizzando il parametro `GL_LINE_STRIP` di `glBegin()` non è più necessario specificare i vertici in comune a due diverse linee. Dovremo solamente fornire la lista di essi e OpenGL li collegherà disegnando una classica linea spezzata. Osservate nel codice seguente come, a parte il primo, tutti gli altri vertici sono concatenati tra loro

```
glBegin(GL_LINE_STRIP);
glVertex3f(-10.0f, 0.0f, 0.0f); // Lato 1 punto 1
glVertex3f(10.0f, 0.0f, 0.0f); // Lato 1 punto 2 - Lato 2 punto 1
glVertex3f(10.0f, 10.0f, 0.0f); // Lato 2 punto 2 - Lato 3 punto 1
glVertex3f(-10.0f, 10.0f, 0.0f); // Lato 3 punto 2 - Lato 4 punto 1
glVertex3f(-10.0f, 0.0f, 0.0f); // Lato 4 punto 2
glEnd();
```

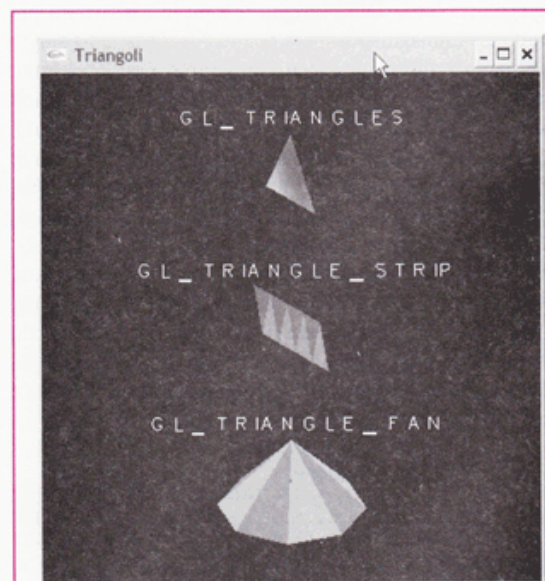
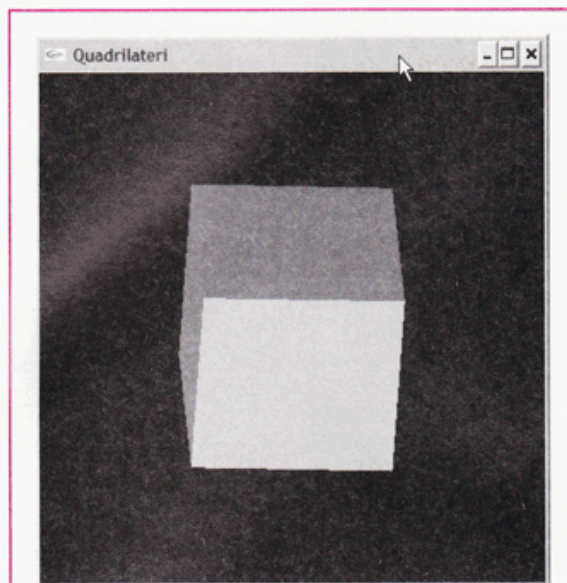


FIGURA 3 Riuniti in questa immagine vediamo un esempio per ognuno dei tre tipi di primitiva `GL_TRIANGLESXXX`



Qui è stato realizzato un cubo utilizzando i quadrilateri. In particolare viene utilizzata la primitiva GL_QUADS per le facce superiore ed inferiore, mentre GL_QUAD_STRIP per quelle laterali

FIGURA 4

Ora sono stati necessari solo 5 vertici. Un bel risparmio di tempo e dati che devono essere trasferiti dalla memoria centrale del computer a quella della scheda acceleratrice. Possiamo, però, fare di meglio; avete notato che il primo e l'ultimo vertice sono identici?

Utilizzando GL_LINE_LOOP, infatti, che si comporta similmente al precedente, viene disegnata una linea tra l'ultimo vertice specificato ed il primo. In termini di codice si ha

```
glBegin(GL_LINE_LOOP);
glVertex3f(-10.0f, 0.0f, 0.0f); // Lato 1 punto 1
// Lato 1 punto 2 - Lato 2 punto 1
glVertex3f(10.0f, 0.0f, 0.0f);
// Lato 2 punto 2 - Lato 3 punto 1
glVertex3f(10.0f, 10.0f, 0.0f);
// Lato 3 punto 2 - Lato 4 punto
glVertex3f(-10.0f, 10.0f, 0.0f);
glEnd();
```

Un esempio di utilizzo delle linee è visibile in **Figura 2** prodotta dal programma Linee.exe

Triangoli

Con le linee è possibile disegnare qualunque tipo di oggetto 3D. È immediato, però, osservare che le figure create con questa tecnica non risultano solide ma presentano un aspetto che viene definito con il termine *wire-frame*, ossia a "fil di ferro". Per questo utilizziamo i poligoni, cioè, figure chiuse che possono essere riempite o meno da un colore selezionato. Il primo poligono che prendiamo in considerazione è il

triangolo, il quale, costituito da soli tre lati, è il più semplice esistente. Da notare anche che OpenGL è particolarmente ottimizzata per lavorare con i triangoli, di conseguenza questa scelta solitamente è la più adottata per costruire i solidi.

Il parametro GL_TRIANGLES disegna, ovviamente, un triangolo collegando tre vertici tra loro come nell'esempio

```
glBegin(GL_TRIANGLES);
glVertex3f(-10.0f, 0.0f, 0.0f);
glVertex3f(0.0f, 10.0f, 0.0f);
glVertex3f(10.0f, 0.0f, 0.0f);
glEnd();
```

Lavorando in uno spazio tridimensionale bisogna fare particolare attenzione al fatto che ogni poligono può essere visto da diverse angolazioni. Per questo si può dire che esso possiede, in realtà, due facce, quella anteriore e quella posteriore. Ma come stabilire quale è l'una e quale l'altra? Il metodo adottato consiste nel distinguere l'ordine con il quale vengono specificati i vertici del poligono. Così la differenza nasce dall'ordine di lettura dei vertici secondo che essi siano indicati in senso orario - *Clockwise winding* - o antiorario - *Counterclockwise winding*.

Per default OpenGL considera i vertici orientati in senso antiorario come la faccia anteriore del poligono. È possibile impostare, tramite la funzione

```
glFrontFace(GL_CW);
```

tali vertici come la faccia posteriore.

L'importanza di poter effettuare questa distinzione sarà chiara più avanti in questo articolo quando si parlerà di rimozione delle facce nascoste.

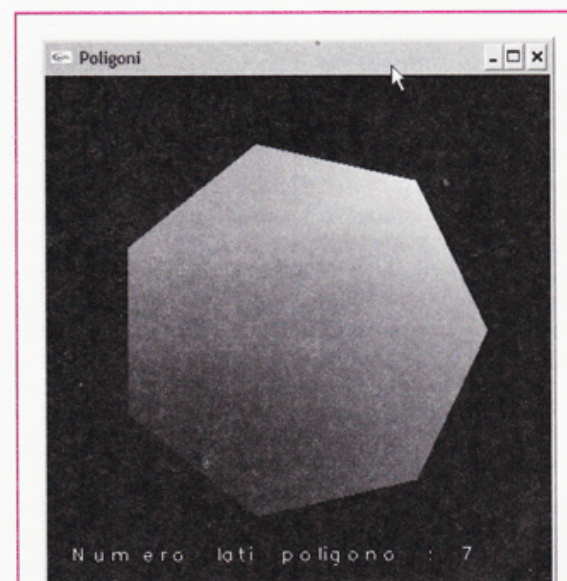


FIGURA 5

In questa immagine possiamo osservare un semplice esempio di primitiva GL_POLYGON

Dal **Riquadro 1** possiamo notare che esistono altri parametri da passare alla `glBegin()` per disegnare triangoli. Troviamo, infatti, `GL_TRIANGLE_STRIP` e `GL_TRIANGLE_FAN`.

Le primitive sono figure geometriche bidimensionali, come punti, linee o poligoni utilizzate per creare oggetti solidi più complessi

Con il primo dei due si possono generare catene di triangoli adiacenti le quali, allo stesso modo che per le linee, fanno risparmiare tempo e quantità di dati trasferiti. Il meccanismo di costruzione della catena consiste nel considerare gli ultimi due vertici specificati come i primi due del triangolo successivo. In questo modo il prossimo vertice sarà preso come quello finale del triangolo che deve essere disegnato.

Osserviamo un esempio

```
glBegin(GL_TRIANGLE_STRIP);
glVertex3f(-10.0f, 0.0f, 0.0f); // Triangolo 1 Vertice 1
// Triangolo 1 Vertice 2 - Triangolo 2 Vertice 1
glVertex3f(0.0f, 10.0f, 0.0f);
// Triangolo 1 Vertice 3 - Triangolo 2 Vertice 2
glVertex3f(10.0f, 0.0f, 0.0f);
glVertex3f(30.0f, 10.0f, 0.0f); // Triangolo 2 Vertice 3
```

```
glEnd();
```

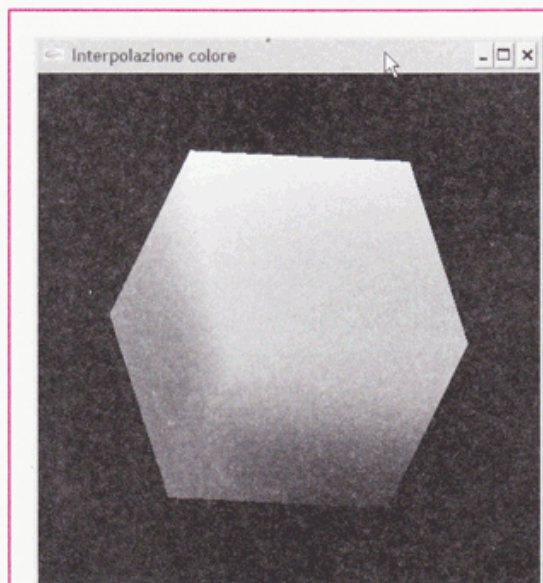


FIGURA 6

Un esempio visivo della capacità di interpolare i colori tra i vari vertici

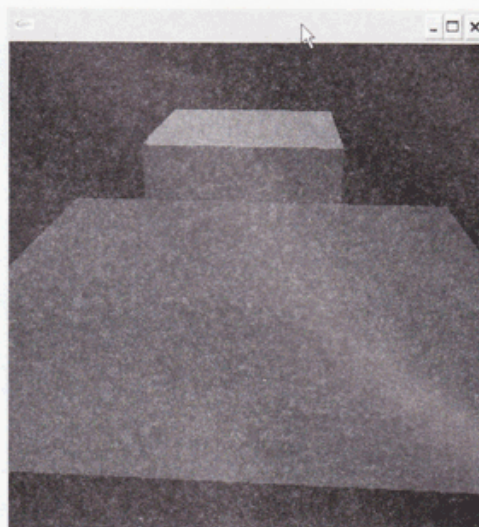


FIGURA 7

L'immagine mostra una situazione dove non è utilizzato il controllo di profondità per gli oggetti

Il `GL_TRIANGLE_FAN` produce un gruppo di triangoli collegati ad un punto centrale. In questo caso il primo vertice specificato appena dopo `glBegin()` sarà considerato come centrale. Ogni altro vertice viene collegato al precedente per formare il nuovo triangolo, come nel codice seguente

```
glBegin(GL_TRIANGLE_FAN);
glVertex3f(-10.0f, 0.0f, 0.0f); // Vertice centrale
glVertex3f(0.0f, 10.0f, 0.0f); // Triangolo 1 Vertice 2
// Triangolo 1 Vertice 3 - Triangolo 2 Vertice 2
glVertex3f(10.0f, 0.0f, 0.0f);
// Triangolo 2 Vertice 3 - Triangolo 3 Vertice 2
glVertex3f(30.0f, 10.0f, 0.0f);
...
glEnd();
```

Il programma di esempio `Triangoli.exe` il cui output è presente in **Figura 3** mostra esempi vari sui triangoli.

Quadrilateri

Ormai dovremmo aver capito il funzionamento del meccanismo che regola la creazione delle primitive. Possiamo, quindi, permetterci di spendere poche parole in merito ai quadrilateri i quali sono semplicemente una lista di quattro vertici, conosciuti da OpenGL come `GL_QUADS`.

L'unico accorgimento da seguire è che tali vertici devono giacere sullo stesso piano.

Come per i triangoli è possibile creare delle catene di quadrilateri tramite `GL_QUAD_STRIP`, dove ognuno di essi è indicato dagli ultimi due vertici del quadrilatero precedente uniti ad altri due nuovi vertici. Chiariamo il tutto con il solito esempio, questa volta costruendo un cubo

```
glBegin(GL_QUADS);

// Faccia superiore
glVertex3f(-10.0f, 10.0f, 10.0f);
glVertex3f(-10.0f, 10.0f, -10.0f);
glVertex3f(10.0f, 10.0f, -10.0f);
glVertex3f(10.0f, 10.0f, 10.0f);

// Faccia inferiore
glVertex3f(-10.0f, -10.0f, 10.0f);
glVertex3f(-10.0f, -10.0f, -10.0f);
glVertex3f(10.0f, -10.0f, -10.0f);
glVertex3f(10.0f, -10.0f, 10.0f);

glEnd();

// Facce laterali
glBegin(GL_QUAD_STRIP);

glVertex3f(-10.0f, -10.0f, 10.0f); // Quad 1 - Vertice 1
glVertex3f(-10.0f, 10.0f, 10.0f); // Quad 1 - Vertice 2
glVertex3f(10.0f, -10.0f, 10.0f); // Quad 1 - Vertice 3
glVertex3f(10.0f, 10.0f, 10.0f); // Quad 1 - Vertice 4

glVertex3f(10.0f, -10.0f, -10.0f); // Quad 2 - Vertice 3
glVertex3f(10.0f, 10.0f, -10.0f); // Quad 2 - Vertice 4

glVertex3f(-10.0f, -10.0f, -10.0f); // Quad 3 - Vertice 3
glVertex3f(-10.0f, 10.0f, -10.0f); // Quad 3 - Vertice 4

glVertex3f(-10.0f, -10.0f, 10.0f); // Quad 4 - Vertice 3
glVertex3f(-10.0f, 10.0f, 10.0f); // Quad 4 - Vertice 4

glEnd();
```

In Figura 4 un esempio dell'output del programma Quadrilateri.exe.

Poligoni

L'ultima primitiva conosciuta da OpenGL è `GL_POLYGON` che è una figura composta da un numero qualunque di vertici. Le restrizioni o regole di costruzione dei poligoni sono due. La prima vuole, come per i quadrilateri, che tutti i vertici del poligono devono giacere sullo stesso piano, mentre la seconda consiste nel fatto che un poligono deve essere obbligatoriamente convesso, ovvero non deve risultare in nessun caso che tracciando una linea qualsiasi questa interseca il poligono in più di due punti.

Il programma Poligoni.exe produce l'output di Figura 5.

Eliminazione delle facce nascoste

Come già detto, visualizzando un poligono nello spazio, accade che OpenGL debba disegnare entrambe le facce, quella anteriore e quella posteriore.

Nell'ultimo esempio presentato abbiamo costruito un solido, ossia, un oggetto dotato di tre dimensioni, come composizione di primitive a due dimensioni.

Sarete d'accordo sul fatto che ogni poligono possiede una faccia, che trovandosi "dentro" al cubo, non verrà visualizzata in alcun caso. Settando questa flag

RIQUADRO 1

Questo è l'elenco completo delle primitive

<code>GL_POINTS</code>	Semplici punti
<code>GL_LINES</code>	Linee
<code>GL_LINE_STRIP</code>	Sequenza di linee
<code>GL_LINE_LOOP</code>	Sequenza chiusa di linee
<code>GL_TRIANGLES</code>	Triangoli
<code>GL_TRIANGLE_STRIP</code>	Sequenza di triangoli
<code>GL_TRIANGLE_FAN</code>	Sequenza di triangoli a "ventaglio"
<code>GL_QUADS</code>	Quadrilateri
<code>GL_QUAD_STRIP</code>	Sequenza di quadrilateri
<code>GL_POLYGON</code>	Poligoni

```
glEnable(GL_CULL_FACE);
```

e lanciando la funzione

```
glCullFace(GL_BACK);
```

otterremo un notevole risparmio di tempo chiedendo ad OpenGL di non disegnare tutte le facce posteriori dei poligoni.

La `glCullFace()` accetta i parametri `GL_BACK`, `GL_FRONT` e `GL_FRONT_AND_BACK` anche se credo che quest'ultimo sia utile in pochissimi casi.

In qualunque momento, anche all'interno delle chiamate `glBegin/glEnd`, possiamo cambiare le impostazioni oppure disabilitare questa funzionalità tramite

```
glDisable(GL_CULL_FACE)
```

Osservate l'esempio FacceNascoste.exe per chiarire ulteriori dubbi.

Il colore

Abbiamo già incontrato la funzione `glColor` che permette di settare il colore di disegno corrente. Tramite questa si possono realizzare bellissimi effetti sfruttando una caratteristica molto interessante di OpenGL; la *Shading* o sfumatura.

Ad ogni vertice di una primitiva, infatti, che sia una linea o un poligono complesso possiamo associargli un proprio colore. La figura sarà disegnata e riempita con una sfumatura di colori interpolati tra quello di un vertice ed un altro.

La Figura 6 mostra un cubo formato da quadrilateri colorati con questa tecnica. Per default, l'interpolazione del colore è abilitata, ma è comunque possibile cambiarla in ogni momento con

```
glShadeModel(GL_FLAT)
```

In questa maniera, durante il disegno del poligono, sarà presa in considerazione solo l'ultima occorrenza della funzione `glColor`. Con la chiamata

```
glShadeModel(GL_SMOOTH)
```

si ritorna allo stato normale.

Test di profondità

Osservate la **Figura 7**, cosa notate di strano? Il piano verde sul quale dovrebbe essere appoggiato il cubo rosso compare davanti ad esso rendendo l'immagine irrealistica e confusa. Questo avviene perché nel codice abbiamo chiesto di disegnare prima il cubo e solo dopo il piano. OpenGL non fa altro che disegnare le figure una dopo l'altra sovrascrivendo l'immagine sottostante.

Il controllo di profondità permette di eliminare questo problema e, tramite particolari test, riesce a stabilire l'effettiva profondità di un oggetto e la sua reale collocazione rispetto agli altri, all'interno della scena.

Per attivarlo sono necessarie alcune operazioni. Innanzitutto bisogna chiedere a GLUT di riservare memoria per il buffer necessario, aggiungendo la flag `GLUT_DEPTH` alla funzione di inizializzazione del display in questo modo

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH)
```

successivamente dobbiamo fare in modo di svuotare tale buffer prima di iniziare a disegnare un fotogramma. Per questo scopo usufruiamo della chiamata `glutClear()` già presente nella funzione `RenderScene()`, la quale dovrà presentare questa forma.

```
glutClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
```

infine deve essere abilitato il controllo della profondità. Può essere fatto ovunque si renda necessario, ma per i nostri scopi va più che bene nella funzione `SetupRC()` dove inseriremo la riga

```
glEnable(GL_DEPTH_TEST)
```

Le trasformazioni

Quando pensiamo alla grafica 3D dovremmo sapere che in realtà essa non è costituita da oggetti tridimensionali, ma tutto ciò che vediamo sullo schermo non è altro che un'immagine bidimensionale. Esiste, quindi, un processo che trasforma i dati degli oggetti 3D nella loro rappresentazione 2D chiamato *proiezione*.

Ci sono anche altri tipi di trasformazioni che producono effetti diversi; ad esempio è possibile ruotare, spostare o ridimensionare gli oggetti. Ogni tipologia di trasformazione geometrica è applicata alla scena in momenti diversi, da quando vengono specificati i vertici fino a che essi non appaiono sullo schermo.

In prima istanza viene applicata una trasformazione alla vista – *Viewing Transformations* – utilizzata per determinare il punto di osservazione della scena. Il comando OpenGL per impostare tale punto è

```
gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz)
```

I parametri di questa funzione sono intuitivi, infatti, `eye` indicano il punto dove è posizionato l'osservatore, le

coordinate `center` rappresentano il punto che sta di fronte all'osservatore, mentre `up` è un vettore puntato verso l'alto dalla prospettiva dell'osservatore.

Chiunque nutra qualche perplessità nei confronti dell'ultimo parametro, può pensare che non è sufficiente stabilire la posizione di un osservatore e la direzione verso la quale sta guardando; si rivela necessario anche conoscere l'inclinazione della sua testa.

Senza questa informazione, infatti, non abbiamo modo di sapere se esso sta guardando, magari, con la testa sottosopra oppure inclinata di 45 gradi.

Il motivo per cui viene applicata prima delle altre risiede nel fatto che lo spostamento del punto di osservazione determina un cambiamento nel sistema di coordinate di lavoro.

Ogni altra trasformazione dovrà tenere conto delle modifiche apportate da questa.

Esempio, vogliamo visualizzare un oggetto centrato alle coordinate (0,0,0) da una certa distanza, scriviamo quindi

```
gluLookAt(0.0f, 50.0f, 50.0f, // Posizione del punto di osservazione
0.0f, 0.0f, 0.0f, // Punto verso il quale "volgiamo lo sguardo"
0.0f, 1.0f, 0.0f); // Vettore che punta verso l'alto, perpendicolarmente al piano
```

I triangoli sono figure per le quali OpenGL possiede una gestione ottimizzata. È consigliabile, quindi, utilizzarli il più possibile nella creazione degli oggetti

L'esempio `PuntoOsservazione.exe` mostra come utilizzare questa trasformazione per girare intorno alla scena osservando il centro di essa.

Il secondo tipo di trasformazioni che vengono applicate sono le *trasformazioni di modellazione* – *Modeling Transformations* – utilizzate prevalentemente per spostare gli oggetti nello spazio, ruotarli attorno agli assi o ridimensionarli.

Iniziando dalle traslazioni la funzione dedicata è la

```
glTranslatef(GLfloat x, GLfloat y, GLfloat z)
```

che genera lo spostamento dell'oggetto della misura indicata per ogni asse.

Abbiamo poi, per le rotazioni

```
glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z)
```

la quale ruota l'oggetto di un angolo pari ad `angle` espresso in gradi.

Per indicare che un dato asse deve essere ruotato, il valore da passare ai parametri `x`, `y` o `z` è `1.0f`.

Nel caso più semplice una rotazione avviene intorno ad un singolo asse, ma è possibile realizzare rotazioni che coinvolgono tutti gli assi. Nell'esempio,

```
glRotatef(45, 0.0f, 0.0f, 1.0f)
```

un oggetto risulterebbe ruotato secondo l'asse Z. Nell'ultimo caso, il ridimensionamento agisce aumentando o diminuendo la dimensione degli oggetti espandendo i vertici, lungo i tre assi.

Ogni parametro della funzione

```
glScalef(GLfloat x, GLfloat y, GLfloat z)
```

rappresenta il fattore moltiplicativo delle dimensioni dell'oggetto. Un valore pari a 1.0f lascia inalterate le dimensioni lungo l'asse, un valore maggiore provoca un aumento, mentre un valore tendente a zero determina il rimpicciolimento.

Scrivendo

```
glScalef(1.0f, 4.0f, 1.0f)
```

un qualunque oggetto risulterebbe allungato verticalmente, ma manterrebbe inalterate le altre dimensioni.

È bene prestare particolare attenzione al fatto che le trasformazioni descritte non vengono applicate al singolo oggetto, ma all'intero sistema di riferimento, di conseguenza ogni altra operazione sarà condizionata dalle nuove impostazioni.

L'inversione dell'ordine di due trasformazioni, rispettivamente una traslazione ed una rotazione produce effetti diversi, come avviene in questo caso

```
glTranslatef(10.0f, 0.0f, 0.0f);
```

```
glRotatef(45, 0.0f, 0.0f, 1.0f);
```

```
...
```

```
disegnaUnCubo();
```

oppure

```
glRotatef(45, 0.0f, 0.0f, 1.0f);
```

```
glTranslatef(10.0f, 0.0f, 0.0f);
```

```
...
```

```
disegnaUnCubo();
```

Nel primo caso abbiamo traslato il cubo di 10 unità sull'asse X con una successiva rotazione di 45 gradi sull'asse Z. Nel secondo è avvenuto il contrario, cioè, è stata prima effettuata una rotazione di 45 gradi sull'asse Z poi una traslazione sull'asse X risultante.

Notate la differenza?

Passiamo al prossimo tipo di trasformazioni: le trasformazioni delle proiezioni.

Questa tipologia di trasformazioni serve a definire quali parti di spazio 3D risultano visibili all'osservatore e a specificare in che modo la scena 3D sarà proiettata sullo schermo.

Esistono due tipi di proiezione

- **Ortagonale**, di cui abbiamo già parlato nel precedente articolo, che non discerne alcuna differenza sulla distanza degli oggetti. Essi, infatti, avranno la stessa dimensione sia che si trovino vicino al punto di osservazione oppure no. Abbiamo anche detto che questo tipo di proiezione viene principalmente utilizzato nelle applicazioni CAD oppure, nel caso dei videogiochi, per creare menu o immagini 2D.
- **Prospettica**, la quale genera immagini che rispecchiano il fenomeno ottico dove gli oggetti uguali e distanti tra loro sembrano avere dimensioni diverse, oppure dove linee parallele, allontanandosi dall'osservatore, sembrano convergere.

La prospettiva viene attivata con la funzione

```
gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar)
```

Diamo una breve spiegazione dei parametri. Il primo e più criptico, *fovy*, è l'angolo del campo visivo in direzione verticale, *aspect* è il solito *aspect ratio* ossia il rapporto tra larghezza della finestra e la sua altezza.

Infine *zNear* e *zFar* sono rispettivamente il limite visibile più vicino all'osservatore e il più lontano. Una funzione *ChangeSize()* tipica dovrà assumere la forma illustrata dal Listato 1.

L'ultimo tipo di trasformazioni è quello che avviene alla *viewport*, quando l'immagine finale è mappata fisicamente sulla finestra. Comunque l'utente non ha bisogno di occuparsi di questo aspetto.

Conclusioni

Dopo la lettura di questo articolo avrete acquisito familiarità con le funzioni di disegno di OpenGL. Potrete ora creare da soli un programma completamente funzionante che visualizza vari tipi di oggetti solidi e non.

Nel prossimo articolo termineremo questo percorso studiando metodi per creare scene più complesse ed utilizzare caratteristiche avanzate come la *Texture mapping*.

Bibliografia

- [1] Richard S.Wright jr. & Benjamin Lipchak, "OpenGL SuperBible – Third Edition", SAMS, 2005, ISBN 0-672-32601-9

Mauro Pesci

Si occupa di Analisi e Sviluppo del software in vari ambienti e piattaforme. Ha lavorato per diverse aziende nel settore, mentre ora è impegnato in un progetto presso la HP di Roma. Attualmente studia Informatica presso l'università di Tor Vergata (Roma).

Utilizzare le OpenGL

L'ultimo passo del nostro percorso; conoscere le tecniche di gestione delle fonti luminose ed applicare agli oggetti immagini 2D per conferirgli un aspetto più realistico

di Mauro Pesci > mpesci@infomedia.it

Le funzionalità più avanzate di OpenGL si occupano, per lo più, della simulazione di fenomeni naturali come la luce e i materiali. In questo modo, tramite il continuo sviluppo, si tenta di dotare la libreria di caratteristiche tali da realizzare scene sempre più vicine alla realtà.

Facciamo un po' di luce

Disegnando primitive o creando oggetti solidi, fino a questo momento, siamo in grado di associargli un determinato colore RGB. Nel mondo reale, però, non vediamo le cose semplicemente in funzione al loro colore assoluto. Tutti sapranno che esso non è altro che un'interpretazione dei nostri organi visivi di una particolare gamma di radiazioni, cioè, la luce. Solitamente scaturisce da una fonte ben definita ed il colore degli oggetti varia in funzione della posizione rispetto ad essa.

OpenGL riesce ad approssimare le condizioni del mondo reale tramite l'utilizzo di tre tipi di illuminazione:

- **La luce di ambiente**, la quale è una luminosità che non proviene da una direzione in particolare pur possedendo una fonte. Quando viene creata una luce di ambiente tutti i poligoni di una scena sono illuminati alla stessa maniera.
- **La luce diffusa**. Rappresenta quel tipo di luce che viene riflessa in maniera uniforme. In questo modo, una superficie, diviene più brillante quando i raggi emessi dalla fonte la colpiscono perpendicolarmente.
- **La luce speculare**. Simile alla precedente a meno del fatto che, in questo caso, i raggi provenienti dalla fonte sono riflessi verso una particolare direzione. Questo effetto genera, solitamente un punto luminoso sulla superficie riflettente, come può fare la luce solare sulla carrozzeria di un'autovettura.

La luce però è solo una parte dell'equazione. Sappiamo, infatti, che colore e brillantezza di un oggetto dipendono anche dal **materiale** che lo costituisce e da come, questo, reagisce alle

radiazioni luminose. Utilizzando l'illuminazione, quindi, non parleremo più del colore di un poligono in termini di livelli RGB, ma delle modalità in cui esso riflette o assorbe determinate lunghezze d'onda. Potremmo creare una pallina blu dall'apparenza metallica oppure un cubo opaco e più simile alla plastica semplicemente impostando le proprietà dei materiali che li costituiscono.

Vediamo ora come unire il tutto. Per attivare l'uso della luce in OpenGL è sufficiente la chiamata

```
glEnable(GL_LIGHTING)
```

la quale abilita il calcolo del colore di ogni vertice in base al materiale e alla posizione della fonte di luce. Ovviamente in assenza di quest'ultima la scena rimarrebbe scura.

Per prima cosa è necessario settare i tre tipi di illuminazione tramite la funzione

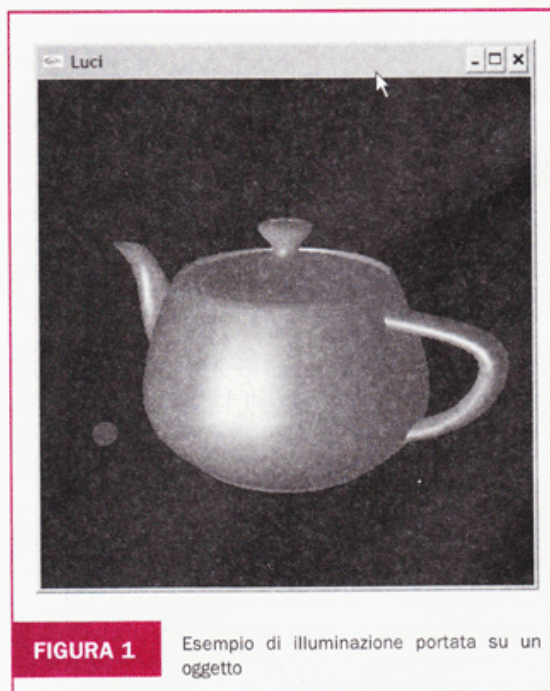


FIGURA 1

Esempio di illuminazione portata su un oggetto

```
void glLightfv(GLenum light, GLenum pname, const GLfloat *param)
```

Il primo parametro della funzione identifica la fonte di luce alla quale applicare le impostazioni. OpenGL ne possiede 8 e variano da `GL_LIGHT0` a `GL_LIGHT7`.

Il secondo parametro chiede che venga specificata una tra le tre tipologie di luce, cioè

- `GL_AMBIENT` per impostare i valori della luce di ambiente
- `GL_DIFFUSE` per impostare i valori della luce diffusa
- `GL_SPECULAR` per impostare i valori della luce speculare

oppure la posizione della luce tramite `GL_POSITION`. Infine, la funzione, accetta un array di 4 float contenente le impostazioni della sorgente, dove i primi tre rappresentano l'intensità delle componenti variabili tra 0.0f e 1.0f, il quarto se impostato a 1.0f fa in modo che i raggi luminosi si diffondano in tutte le direzioni, mentre se 0.0f essa sarà come se provenisse da una distanza infinita. In quest'ultimo modo i raggi sono elaborati come paralleli tra loro. Per poter utilizzare la fonte impostata bisogna attivarla tramite la chiamata a

```
glEnable(GL_LIGHT0);
```

Ora è necessario fare in modo che i poligoni possiedano un materiale per poter reagire all'illuminazione. Abilitiamo la gestione dei materiali

```
glEnable(GL_COLOR_MATERIAL);
```

dopodiché informiamo OpenGL sulle proprietà delle facce dei poligoni con la funzione

```
void glColorMaterial(GLenum face, GLenum mode);
```

Dove il primo parametro indica a quale faccia applicare l'impostazione, i valori possibili sono `GL_FRONT`, `GL_BACK` per le facce anteriori e posteriori, mentre `GL_FRONT_AND_BACK` lo fa su ambedue. Il secondo parametro accetta i valori `GL_EMISSION`, `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_AMBIENT_AND_DIFFUSE`. Questi dicono che il materiale deve seguire le impostazioni di colore della fonte luminosa indicata. Fatto ciò non ci resta che rendere il materiale speculare con la funzione

```
void glMaterialfv(GLenum face, GLenum pname, const GLfloat *params)
```

chiamata in questa forma

```
GLfloat riflettivitàSpeculare[] = {1.0f, 1.0f, 1.0f, 1.0f};
```

```
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, riflettivitàSpeculare);
```

ed impostare il suo livello di riflettività con

```
glMateriali(GL_FRONT, GL_SHININESS, 128);
```



FIGURA 2

Sulla scena sono presenti due oggetti, la teiera e il piano orizzontale, ognuno associato ad una propria texture

Più il valore dell'ultimo parametro è alto – al massimo può assumere 128 – più l'effetto speculare è "a fuoco", cioè il punto generato è maggiormente definito e concentrato. Vediamo un esempio pratico, il **Listato 2** riporta una possibile inizializzazione della fonte luminosa `GL_LIGHT0` e del materiale in un generico programma.

Tra i programmi di esempio c'è `Luci.exe` che illustra in maniera pratica l'applicazione delle regole fino ad ora apprese, un'immagine ripresa dall'esempio è la **Figura 1**.

Ora è tutto pronto e OpenGL è in grado di gestire propriamente la sorgente di luce impostata.

Non abbiamo però inserito alcun oggetto nella scena. Anche se lo avessimo fatto il risultato non sarebbe stato quello voluto.

Questo perché non è stato introdotto un ulteriore concetto, quello di *normale*. Una normale non è altro che il vettore perpendicolare al piano su cui giace un poligono. Sono utilizzate per determinare l'angolo di inclinazione di questo rispetto alla sorgente luminosa.

È evidente che se tale angolo è uguale a zero significa che la luce è posta perpendicolarmente alla figura, di conseguenza il poligono risulterà maggiormente illuminato. Diversamente, quando l'angolo aumenta, la quantità di luce riflessa diminuisce.

Non esistono funzioni specifiche per il calcolo delle normali di un poligono e, senza soffermarci troppo sull'argomento, vi consiglio di utilizzare la libreria `gltools` che troverete sul sito FTP.

Questa mette a disposizione la

```
glGetNormalVector(GLTVector vP1, GLTVector vP2, GLTVector vP3,  
GLTVector vNormal)
```

LISTATO 1

Come impostare ed utilizzare la correzione prospettica nelle immagini

```
// Chiamata ad ogni ridimensionamento della finestra
// compresa la prima volta che questa viene creata
void ChangeSize(GLsizei w, GLsizei h)
{
    GLfloat aspectRatio;

    // Nel caso in cui la finestra venga ridotta
    // a icona produrrebbe un errore di divisione
    // per zero nella prossima istruzione
    if (h == 0) h = 1;

    // Calcolo il rapporto tra larghezza e altezza
    // della finestra
    aspectRatio = (GLfloat) w / (GLfloat) h;

    // Imposta la dimensione della viewport
    glViewport(0, 0, w, h);

    // Resetto la matrice di trasformazione delle proiezioni
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Imposto le dimensioni del volume di clipping
    gluPerspective(45.0f, aspectRatio, 10.0f, 300.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

Le variabili di tipo `GLTVector` sono array di *float* che contengono le coordinate di vertici del poligono. Il parametro `vNormal` è il risultato del calcolo e deve essere passato alla funzione

```
glNormal3fv(const GLfloat *v);
```

la quale accetta un'array di *float* in vece del vettore che rappresenta la normale. Osservate il **Listato 3** per avere un'idea più chiara di come implementare la gestione corretta delle normali. Noterete anche la funzione

```
glNewList(1, GL_COMPILE);
```

questa istruzione fa in modo che OpenGL compili tutti i comandi che incontra fino alla successiva chiamata

```
glEndList();
```

in modo da eseguirli più velocemente. Ogni volta che è necessario eseguire la lista basta chiamare la funzione

```
glCallList(1);
```

Le Texture

Il riempimento tramite un semplice colore spesso non è sufficiente a soddisfare le nostre esigenze. Una importante caratteristica posseduta da ogni libreria di grafica 3D che si rispetti è la possibilità di applicare ai poligoni un'immagine di riempimento, la cosiddetta *texture mapping*.

Potremo creare scene molto convincenti fornendo agli oggetti una parvenza quasi reale. Abbiamo già visto, nel primo articolo, un esempio di *texture mapping*.

La prima cosa che dobbiamo fare per utilizzare le texture in un programma è caricare le immagini e a questo scopo ci assiste la

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

L'enorme quantità di parametri serve a fornire tutte le indicazioni necessarie per interpretare correttamente i dati puntati da *data*. Vediamone il significato

- *target* deve essere impostato a `GL_TEXTURE_2D`
- *level* impostato a 0 serve a specificare il livello di mipmap. Non tratteremo questo argomento.
- *internalformat* specifica il formato che i dati devono assumere quando sono caricati. Può assumere i valori `GL_ALPHA`, `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA`, `GL_RGB`, `GL_RGBA`. Solitamente, questo parametro, è impostato a `GL_RGB` e `GL_RGBA`.
- *width*, *height* sono le dimensioni dell'immagine. È molto importante sapere che l'immagine può avere misure che seguono le potenze di 2 ossia 4x2, 64x128, ..., 4096x1024 e così via
- *border* serve a creare un bordo per l'immagine. Impostare a 0.
- *format* indica il contenuto dei dati dell'immagine ossia il formato dei dati dell'immagine. Esistono molti valori che può assumere, ma quelli più comuni sono `GL_RGB` e `GL_RGBA`.
- *type* è il tipo di dato utilizzato per i pixel. Per le immagini a 24 o 32 bit l'impostazione corretta è `GL_UNSIGNED_BYTE`
- *data* è l'array che contiene i dati dell'immagine

Come era ovvio aspettarsi OpenGL non possiede funzionalità per caricare le immagini. Utilizziamo la funzione contenuta nel file `LoadTGA.c`. Altri parametri devono essere impostati prima di utilizzare una texture. In primo luogo dobbiamo definire cosa accade quando la essa viene ridimensionata, cioè, quando l'oggetto viene a trovarsi molto vicino all'osservatore oppure lontanissimo. Scrivendo

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

faremo in modo che sia utilizzato un filtro che sfuma leggermente i bordi dei pixel ingranditi evitando l'antiestetica seghettatura.

Anche la gestione delle texture deve essere abilitata con

```
glEnable(GL_TEXTURE_2D);
```

Una volta caricata l'immagine bisogna mapparla sul poligono. Ciò deve essere fatto durante l'impostazione dei vertici tramite

```
glTexCoord2f(GLfloat s, GLfloat t)
```

Ogni coordinata è associata al vertice seguente e può variare tra 0.0 e 1.0 qualunque siano le dimensioni dell'immagine. Il codice seguente associa una generica immagine ad un triangolo

```
glBegin(GL_TRIANGLE);
glTexCoord2f(0.0f, 0.0f);
glVertex3f(0.0f, 0.0f, 0.0f);
glTexCoord2f(0.5f, 1.0f);
glVertex3f(5.0f, 10.0f, 0.0f);
glTexCoord2f(1.0f, 0.0f);
glVertex3f(10.0f, 0.0f, 0.0f);
glEnd();
```

In **Figura 2** potete vedere l'output del programma Texture.exe.

Ormai dovrete aver acquisito familiarità con i nostri programmi, quindi, studiarne il codice non dovrebbe risultare un'impresa così difficile.

L'effetto nebbia

Un effetto molto semplice da utilizzare con le OpenGL è la nebbia – *fogging* – il quale viene applicato al termine di tutte le operazioni inerenti al colore. Esso determina il mescolamento del colore della geometria presente in scena con quello della nebbia. Si possono anche variare i valori di densità e le equazioni che generano un effetto diverso in base alla distanza degli oggetti dal punto di osservazione. Per usare l'effetto è sufficiente seguire questa procedura:

```
// Inizializzare un array di quattro GLfloat
// contenente il colore della nebbia
GLfloat coloreNebbia[] = {0.7f, 0.7f, 0.7f, 1.0f};

// Abilitare l'effetto
glEnable(GL_FOG);

// Impostare il colore della nebbia
glFogfv(GL_FOG_COLOR, coloreNebbia);

// Settare la distanza, dall'osservatore, di
// inizio del banco di nebbia
glFogf(GL_FOG_START, 5.0f);

// Settare la distanza dove questo finisce
glFogf(GL_FOG_END, 30.0f);

// Indica l'equazione di addensamento
```

LISTATO 2

Questo frammento di codice mostra un esempio di utilizzo delle fonti di luce OpenGL

```
// Array per le impostazioni delle luci
GLfloat luceAmbiente[] = {0.1f, 0.1f, 0.1f, 1.0f};
GLfloat luceDiffusa[] = {0.45f, 0.2f, 0.4f, 1.0f};
GLfloat luceSpeculare[] = {1.0f, 1.0f, 0.4f, 1.0f};

GLfloat posizioneLight0[] = {0.0f, 0.0f, 0.0f, 1.0f};

// Array di impostazione della riflettività speculare
// degli oggetti
GLfloat riflettivitàSpeculare[] = {1.0f, 1.0f, 1.0f, 1.0f};

// Gestisce tutte le impostazioni iniziali dell'applicazione
void SetupRC()
{
    // Cambia il colore di cancellazione
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    // Attiva il controllo della profondità
    glEnable(GL_DEPTH_TEST);

    // Attivo l'interpolazione del colore
    glShadeModel(GL_SMOOTH);

    // Attivo l'uso della luce
    glEnable(GL_LIGHTING);

    // Imposto le componenti ed attivo la luce 0
    glLightfv(GL_LIGHT0, GL_AMBIENT, luceAmbiente);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, luceDiffusa);
    glLightfv(GL_LIGHT0, GL_SPECULAR, luceSpeculare);
    glLightfv(GL_LIGHT0, GL_POSITION, posizioneLight0);

    // Attivo la fonte 0
    glEnable(GL_LIGHT0);

    // Abilito l'uso dei materiali
    glEnable(GL_COLOR_MATERIAL);

    // Imposto le proprietà del materiale in modo da
    // rispettare le impostazioni della fonte luminosa
    glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);

    // Imposto i parametri di riflettività
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, riflettivitàSpeculare);

    // Imposto il livello di riflettività
    glMateriali(GL_FRONT_AND_BACK, GL_SHININESS, 32);
}

// della nebbia
glFogi(GL_FOG_MODE, GL_LINEAR);

// Settare la densità della nebbia
glFogf(GL_FOG_DENSITY, 0.5f);
```

Questo è quanto. L'unica osservazione può essere fatta per l'ultima istruzione la quale permette di cambiare la funzione che calcola l'addensamento della nebbia in funzione alla

LISTATO 3

La funzione che crea alcuni oggetti ed imposta le normali corrette

```

// Creo i muri memorizzandoli in una lista
void DisegnaMuri()
{
    GLint i, j;

    // Creo una nuova lista
    glNewList(1, GL_COMPILE);

    // Imposto il colore del muro sottostante
    glColor3f(1.0f, 1.0f, 1.0f);

    // Muro sottostante
    glBegin(GL_QUADS);

    for (i = -MEZZO_MURO; i < MEZZO_MURO - PASSO; i += PASSO)
    {
        for (j = -MEZZO_MURO; j < MEZZO_MURO - PASSO; j += PASSO)
        {
            // Creo un array di vettori contenente
            // le coordinate dei vertici del muro
            GLTVector3 points[4] = { {j, 0, i+PASSO}, {j+PASSO, 0, i+PASSO},
            {j+PASSO, 0, i}, {j, 0, i} };

            // Calcolo la normale di tre dei quattro punti
            glGetNormalVector(points[0], points[1], points[2], vNormal);

            // Imposto la normale
            glNormal3fv(vNormal);

            // Imposto i vertici in base ai valori
            // dell'array di vettori
            glVertex3fv(points[0]);
            glVertex3fv(points[1]);
            glVertex3fv(points[2]);
            glVertex3fv(points[3]);
        }
    }

    glEnd();

    // Laterale DX
    glBegin(GL_QUADS);

    for (i = 0; i < DIMENSIONE_MURO; i += PASSO)
    {
        for (j = -MEZZO_MURO; j < MEZZO_MURO - PASSO; j += PASSO)
        {
            // Creo un array di vettori contenente
            // le coordinate dei vertici del muro
            GLTVector3 points[4] = { {MEZZO_MURO-PASSO, i, j+PASSO},
            {MEZZO_MURO-PASSO, i+PASSO, j+PASSO}, {MEZZO_MURO-PASSO,
            i+PASSO, j}, {MEZZO_MURO-PASSO, i, j} };

            // Calcolo la normale di tre dei quattro punti
            glGetNormalVector(points[0], points[1], points[2], vNormal);

            // Imposto la normale
            glNormal3fv(vNormal);

            // Imposto i vertici in base ai valori
            // dell'array di vettori
            glVertex3fv(points[0]);
            glVertex3fv(points[1]);
            glVertex3fv(points[2]);
            glVertex3fv(points[3]);
        }
    }

    glEnd();

    // Chiudo la lista
    glEndList();
}

```

distanza dall'osservatore. Possiamo, infatti, specificare altri parametri oltre a `GL_LINEAR`, come `GL_EXP` e `GL_EXP2`. Scegliendo un'equazione di tipo esponenziale otterremo un effetto di addensamento più marcato solo alle grandi distanze.

Il blending

Sappiamo, ormai, che le immagini vengono disegnate nel *color buffer* per sovrapposizione. Potremmo essere interessati ad attivare la funzionalità

```
glEnable(GL_BLEND);
```

facendo in modo che le nuove figure in procinto di essere disegnate all'interno del color buffer siano, invece, combinate con quelle esistenti. Esistono vari modi di combinare il colore di due immagini ed ognuno può generare un particolare effetto, come, ad esempio, la trasparenza.

Per poter procedere è indispensabile conoscere la terminologia comunemente utilizzata. Il colore già presente all'interno del color buffer è chiamato *destinazione*, mentre quello che sta per essere scritto è il *sorgente*. Per stabilire come saranno combinati i due colori viene utilizzata la seguente equazione

$$C_i = (C_s * S) + (C_d * D)$$

dove C_i è il colore finale, C_s è il sorgente, C_d quello di destinazione. S e D sono i fattori di blending del colore sorgente e destinazione. Tali fattori sono richiesti dalla

```
glBlendFunc(GLenum S, GLenum D);
```

Il **Riquadro 1** riassume i valori possibili per questi parametri. Vediamo un esempio classico. Questa è l'impostazione più comune per generare un la trasparenza

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Ecco cosa succede se sul color buffer è presente un rosso {1.0f, 0.0f, 0.0f, 0.0f}, mentre il colore che sta per essere disegnato è un blu con queste impostazioni {0.0f, 0.0f, 1.0f, 0.5f}. Abbiamo che

$$\begin{aligned} C_d &= \{1.0f, 0.0f, 0.0f, 1.0f\} \\ C_s &= \{0.0f, 0.0f, 1.0f, 0.5f\} \\ S &= \text{livello alpha del sorgente} = 0.5f \\ D &= 1 - 0.5f = 0.5f \end{aligned}$$

L'equazione assume la forma

$$C_i = (C_s * 0.5f) + (C_d * 0.5f) = \{0.5, 0.0f, 0.0f, 0.0f\} + \{0.0f, 0.0f, 0.5f, 0.5f\} = \{0.5f, 0.0f, 0.5f, 0.5f\}$$

generando un viola scuro.

RIQUADRO 1 Schema che illustra l'elenco dei fattori di blending

Funzione	Fattore Blend RGB	Fattore Blend Alpha
GL_ZERO	(0,0,0)	0
GL_ONE	(1,1,1)	1
GL_SRC_COLOR	(Rs, Gs, Bs)	As
GL_ONE_MINUS_SRC_COLOR	(1,1,1) - (Rs, Gs, Bs)	1 - As
GL_DST_COLOR	(Rd, Gd, Bd)	Ad
GL_ONE_MINUS_DST_COLOR	(1,1,1) - (Rd, Gd, Bd)	1 - Ad
GL_SRC_ALPHA	(As, As, As)	As
GL_ONE_MINUS_SRC_ALPHA	(1,1,1) - (As, As, As)	1 - As
GL_DST_ALPHA	(Ad, Ad, Ad)	Ad
GL_ONE_MINUS_DST_ALPHA	(1,1,1) - (Ad, Ad, Ad)	1 - Ad
GL_CONSTANT_COLOR	(Rc, Gc, Bc)	Ac
GL_ONE_MINUS_CONSTANT_COLOR	(1,1,1) - (Rc, Gc, Bc)	1 - Ac
GL_CONSTANT_ALPHA	(Ac, Ac, Ac)	Ac
GL_ONE_MINUS_CONSTANT_ALPHA	(1,1,1) - (Ac, Ac, Ac)	1 - Ac
GL_SRC_ALPHA_SATURATE	(f,f,f)	1

Conclusioni

Spero di essere stato abbastanza chiaro ed esauriente dato che affrontare un argomento così complesso non sarebbe sufficiente un libro intero.

È molto importante, quindi, fare riferimento alle specifiche di OpenGL che potrete trovare sul sito FTP per approfondire la conoscenza delle funzioni e parametri da esse accettati. Esistono tante funzionalità, infatti, che non ho avuto la possibilità di mostrare per questioni di spazio.

Forse ritornerò a trattare temi riguardanti la programmazione di OpenGL, ma dipende anche da voi. Scrivete e partecipate con critiche, domande e consigli.

Fateci sapere se il nostro lavoro risulta per voi utile o, in qualche modo, interessante.

Bibliografia

- [1] Richard S. Wright jr. & Benjamin Lipchak, "OpenGL SuperBible - Third Edition", SAMS, 2005, ISBN 0-672-32601-9

Mauro Pesci

Si occupa di Analisi e Sviluppo del software in vari ambienti e piattaforme. Ha lavorato per diverse aziende nel settore, mentre ora è impegnato in un progetto presso la HP di Roma. Attualmente studia Informatica presso l'università di Tor Vergata (Roma).