

Tesi - Tutorial OpenGL ES

Tutorial OpenGL ES di [SàgaShiftyblow](http://www.opengles.altervista.org) (Falivene Elisabetta) www.opengles.altervista.org

Versione finale (Ultimo aggiornamento 17 Settembre 2006)

Indice

Capitolo 1: Introduzione a OpenGL® ES

- 1.1 Introduzione
- 1.2 OpenGL®: Cenni storici
- 1.3 Cos'è OpenGL® ES?
- 1.4 Dispositivi Embedded
- 1.5 Sistemi operativi "Embedded"

Capitolo 2: Primi passi

- 2.1 Introduzione
- 2.2 Un esempio di codice: Creare una finestra
- 2.3 Mantenere le proporzioni della finestra
- 2.4 Utilizzare la tastiera e il mouse

Capitolo 3: Il disegno

- 3.1 Introduzione
- 3.2 Disegno e frame
- 3.3 Double buffering
- 3.4 Coordinate
- 3.5 Colori
- 3.6 Effetti di colore: Blending
- 3.7 Un esempio di codice: visualizzazione 2D

Capitolo 4: Disegnare in tre dimensioni

- 4.1 Introduzione
- 4.2 Proiezioni
- 4.3 Profondità
- 4.4 Occultamento delle facce nascoste
- 4.5 Trasformazioni geometriche: rotazione, traslazione, ridimensionamento
- 4.6 Animazione
- 4.7 Illuminazione e proprietà di materiale
- 4.8 Un esempio di codice: visualizzazione 3D

Capitolo 5: Effetti avanzati

- 5.1 Introduzione
- 5.2 Texture Mapping
- 5.3 Proprietà delle texture e Mipmap
- 5.4 Trasparenza
- 5.5 Nebbia
- 5.6 Un esempio di codice: rendere la scena realistica

Appendici

A. Preparare l'ambiente di lavoro

- A.1 Introduzione
- A.2 Programmare in OpenGL® ES
- A.3 Preparare l'ambiente di lavoro
- A.4 Creare un nuovo progetto in Microsoft Embedded C++

B. Caricare un'immagine: Windows Bitmap

C. Strumenti utili

- C.1 Dominio dei tipi di dato
- C.2 Suffissi per funzioni a seconda del tipo degli argomenti
- C.3 Precedenza e associatività degli operatori

C.4 Sequenze di escape
C.5 Codici Ascii
C.6 Codici ASCII multilingua
C.7 Codici di riconoscimento dei tasti
C.8 Codici di tastiera

Ringraziamenti

Bibliografia

◀◀ Capitolo 1: Introduzione a OpenGL® ES ▶▶

◀◀ 1.1 Introduzione ▶▶

Benvenuti in questo corso!

Questo testo è pensato per dare una possibilità a tutti coloro che vogliano cimentarsi nell'impresa 3D, con un particolare occhio per gli italiani che, troppo poco spesso, come si è potuto avere esperienza, riescono a reperire materiale didattico di livello avanzato nella propria lingua nativa.

Il corso è strutturato in capitoli e paragrafi, pensati per guidare passo passo lo studente nell'acquisire i concetti della programmazione OpenGL® ES. Ogni capitolo è composto da paragrafi "di studio", in cui i concetti vengono chiariti ed analizzati in profondità, e da paragrafi "d'esempio", in cui quanto è stato visto viene applicato in esempi semplici e completi. Essi sono costruiti in maniera tale da prevedere un semplice sistema di gestione della tastiera che permetta, su pressione di tasti, di modificare in run-time le variabili più tipiche, mostrando così, sul campo, differenze significative tra le varie possibilità offerte, laddove ritenuto necessario ad aumentare la comprensione degli argomenti. Ogni riga di codice, spesso separatamente, è spiegata con dovizia ed attenzione, perchè nulla sfugga alla piena comprensione delle meccaniche utilizzate. La presenza abbondante di esempi, immagini dimostrativi e solleciti ad effettuare prove, accompagna lo studente per tutto il corso, facendo in modo che non costituisca solo parte passiva nell'insegnamento, bensì attiva e partecipe, poiché, prendendo in prestito le parole di un antico saggio, "Ascolto e dimentico, vedo e ricordo, faccio e capisco" (Confucio)

Il corso prevede, come prerequisito, una conoscenza di base della programmazione in linguaggio C e C++ e della grafica 3D, ma, dove fattibile, è pensato per supporre il meno possibile la presenza di conoscenze precedenti. Un esempio ne sono i riquadri marroni "Nota bene", posti a lato di alcuni argomenti, che hanno proprio lo scopo di sopperire alla mancanza di alcuni concetti di base che potrebbero, da utente a utente, non essere conosciuti o anche solamente non essere ben chiari.

Come già detto, il corso si propone di guidare lo studente dal principio, lasciando meno possibile al caso. In questo primo capitolo, ci si occuperà, quindi, di spiegare cosa sia OpenGL® ES. Fatta una breve introduzione storica ed amministrativa sulla libreria grafica in esame, ci si occuperà di dare una breve panoramica sui dispositivi embedded e relativi sistemi operativi. Nel secondo capitolo si inizierà, invece, a lavorare sulla programmazione vera e propria, soffermandosi sulla base costituita dal problema della creazione di una semplice finestra.

Nel capitolo terzo si passerà ad occuparsi del disegno vero e proprio, con tutte le implicazioni del caso. Assimilato il disegno in due dimensioni, quindi, nel capitolo quarto, si analizzerà il disegno in tre dimensioni. Infine, nel quinto capitolo, ci si occuperà di argomenti più avanzati, come il texturing e l'effetto nebbia.

Sono previste varie appendici finali, ognuna di carattere del tutto singolare. L'appendice A, ad esempio, introduce il discente al problema, più specifico, di preparare il "banco da lavoro", cioè di reperire gli strumenti necessari alla programmazione ed imparare il loro utilizzo. Il corso prevede infatti molti esempi e si è reputato opportuno fornire al lettore strumenti specifici che permettano di testarli. Non potendo presentare tutti i sistemi disponibili è stato scelto un ben preciso ambiente di programmazione e ci si è occupati di creare una piccola, ma completa, guida alla sua installazione ed uso, anche per poter mostrare un esempio pratico che sia linea guida nella gestione dell'analogo problema con diversi ambienti.

La scelta di dividere questa parte dal resto del corso è stata fatta per mantenere una politica di genericità, che non favorisca la preparazione al lavoro con un singolo specifico ambiente, scelto dittatorialmente dall'autrice, bensì mostri il generico funzionamento dei meccanismi della libreria in esame, del tutto al di fuori del contesto dell'ambiente di programmazione, che non è argomento né obiettivo specifico del percorso didattico proposto.

Al corso si affianca un sito di riferimento <http://www.opengles.altervista.org>, dove è possibile reperire, in ogni momento, la versione più aggiornata del corso, contattare

l'autore per chiarimenti e segnalazioni, oppure lasciare messaggi nel forum online per scambiare pareri, domande e risposte con gli altri studenti coinvolti.
Buono studio!

«« 1.2 OpenGL®: Cenni storici »»

I primi passi verso uno standard grafico furono fatti negli anni ottanta, quando organizzazioni, appartenenti a diverse nazioni, cominciarono a cooperare per sviluppare uno standard che fosse generalmente accettato nell'ambito della grafica per computer. Il primo obiettivo era quello della **portabilità**, ovvero la costruzione di funzioni grafiche standard, facili da trasportare da un sistema hardware all'altro e da un sistema operativo all'altro. Da questo sforzo nacque, nel 1977, **GKS** (Graphical Kernel System) che fu accettato come standard ((GKS)-ISO 7942) dall'organizzazione ISO nel 1986 (International Standard Organization) e da varie altre organizzazioni nazionali che si occupano di standard, come l'ANSI (American National Standard Institute). GKS era pensato, inizialmente, solo per la grafica 2D, ma l'estensione in grado di gestire il caso 3D non tardò ad aggiungersi al sistema originale.

Ad esso seguì **PHIGS** (Programmer Hierarchical Interactive Graphical Standard), un'estensione dello stesso GKS che prevedeva nuove capacità relative alla modellazione gerarchica, alla gestione dei colori e alla manipolazione delle immagini. Fu accettato ufficialmente come standard nel 1989. L'impegno di standardizzazione si faceva, quindi, sempre più significativo, e con esso arrivò anche OpenGL®.

Tra il 1984 ed il 1988, infatti, la Silicon Graphics mise in commercio una workstation grafica chiamata **SGI**, la quale conteneva un insieme di routine native detto **GL** (Graphic Library). Con il successo della workstation anche GL andò affermandosi, fino a diventare uno standard de facto. Il passo da workstation ad altri sistemi grafici fu breve e così nacque **OpenGL®**, come versione indipendente dalla piattaforma di GL negli anni '90 (l'accettazione come standard iniziò nel 1992), ed in seguito fu arricchita dal pacchetto **GLU** (GL Utility), che provvede funzioni di alto livello, come quelle per la creazione e gestione delle finestre.

OpenGL® è costituito da un insieme di specifiche indipendenti da un linguaggio di programmazione in particolare, il che, nel tempo, ha permesso di costruire varie sovrastrutture in grado di permettere l'uso delle funzioni grafiche in diversi linguaggi di programmazione, come il C/C++, l'Ada o il Fortran.

L'impegno su OpenGL® non si è mai spento, il che è ampiamente dimostrato non solo dall'aggiornamento continuo effettuato dall'**Architecture Review Board**, il consorzio di rappresentanti di varie compagnie e organizzazioni, che tuttora si occupa di migliorare la libreria grafica, ma anche e soprattutto dal nuovo prodotto, **OpenGL® ES** (Embedded System), che estende la libreria all'utilizzo sui dispositivi mobili.

«« 1.3 Cos'è OpenGL® ES? »»

OpenGL® ES è un API multi piattaforma, royalty-free con accesso completo a funzioni di grafica bidimensionale e tridimensionale sui sistemi *embedded*, categoria che include dispositivi mobili, veicoli ed elettrodomestici. E' un sottoinsieme ben definito di funzioni dell'API OpenGL® e rappresenta un'interfaccia flessibile di basso livello tra il software e gli acceleratori grafici.

I vantaggi di una tale soluzione sono molteplici:

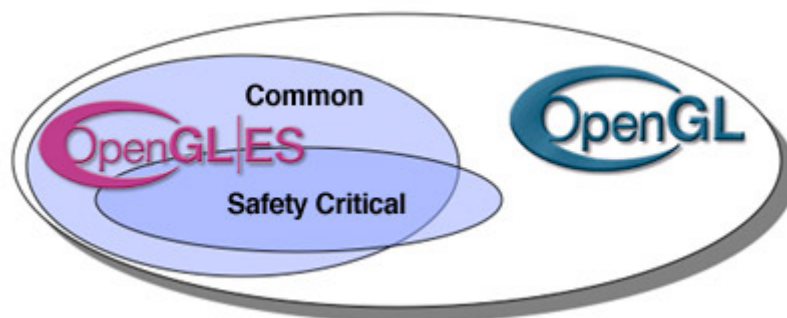
- **Standard e royalty-free.** Ognuno può ottenere la specifica OpenGL® ES ed implementare prodotti basati su questa API. Avendo un ampio supporto da parte di moltissimi produttori Hardware e Software, essa rappresenta uno standard aperto e multi piattaforma.
- **Poco spazio occupato in memoria e basso consumo energetico.** Lo spazio *embedded* varia da PDA (Personal Digital Assistant) con processore a 400Mhz e 64MB di RAM a cellulari con processore a 50MHz e 1 MB RAM. OpenGL® ES è stata sviluppata per adattarsi a queste differenze, richiedendo un'occupazione minima in memoria, un traffico dati/istruzioni minimo e supporto per operazioni sia intere sia

in virgola mobile. Nelle sue prime versioni non è stata assunta la presenza di processori in virgola mobile (*Floating Point Unit, FPU*).

- **Estensibile e in evoluzione.** OpenGL® ES permette alle innovazioni hardware di essere accessibili dall'API attraverso il meccanismo delle estensioni. Trattasi propriamente di parti che estendano le potenzialità del nucleo originale. Quando tali estensioni diventino largamente accettate, cominciano ad essere prese in considerazione per l'inclusione nel nucleo dello standard OpenGL®. Questo processo permette all'API di evolvere in maniera controllata. Le estensioni sono divisibili in due categorie: quelle già totalmente integrate nella definizione principale (core addition) e quella che sono propriamente estensioni (profile extensions). Le core addition non utilizzano i suffissi delle estensioni e non hanno bisogno d'inizializzazione, mentre le profile extensions mantengono i propri suffissi e devono essere inizializzate poiché non ancora parti integranti del nucleo.
- **Facile da usare.** Essendo basata su OpenGL®, OpenGL® ES si presenta ben strutturata e con un design intuitivo e logico.

Le specifiche OpenGL® ES includono la definizione di differenti *profili*. Ogni profilo è un sottoinsieme delle specifiche OpenGL® con l'aggiunta di alcune estensioni addizionali create appositamente. I profili attuali condividono lo stesso processo di elaborazione e la stessa struttura dei comandi. Ogni definizione di profilo comporta un header file ed una libreria a collegamento dinamico (DLL, cioè Dynamic Linked Library) distinti che definiscono i comandi nel profilo. Correntemente, la specifica contiene in totale due definizioni di profili:

- **Common Profile.** Per l'uso comune.
- **Safety Critical Profile.** Per l'uso in casi al limite della sicurezza, dove un errore, di qualsiasi genere e tipo, comprometterebbe sistemi con alte responsabilità, come può essere, ad esempio, la console di pilotaggio di un aereo di linea.



Esistono, o sono comunque in progettazione, diverse versioni di OpenGL® ES. OpenGL® ES 1.1 enfatizza l'accelerazione hardware per le API, mentre OpenGL® ES 1.0 si focalizza sull'abilitazione delle implementazione puramente software. OpenGL® ES 1.1 è completamente compatibile all'indietro con 1.0, permettendo così il facile adattamento delle applicazioni tra le due versioni dell'API.

Le specifiche OpenGL® ES sono state sviluppate dal gruppo Khronos, una proficua unione di moltissime società, che può essere consultata [qui](#).



mesi o anni, e che dispongono di limitate capacità. Per questo, è stata grande l'attenzione nell'evitare sprechi di memoria ed energia, il che si incarna in tecniche avanzate di ottimizzazione della gestione della memoria volatile (RAM) e di quella fissa (Hard-disk), nonché in sistemi atti a prevenire il consumo indiscriminato d'energia, come può essere l'atto di mettere in stand-by i dispositi non utilizzati correntemente.

Palm OS è un sistema sviluppato dalla Palm Computing, una sussidiaria della 3COM. I dispositivi basati su Palm OS sono prodotti dalla Palm e da altre società quali Symbol, IBM, HandSpring, Acer, Sony, Kyocera. L'ultima versione del sistema operativo è Palm OS 5 (Garnet), che supporta microprocessori a 32 Bit ARM della Motorola, Intel e Texas Instruments. A differenza delle versioni precedenti, Palm OS 5 è multi thread, e possiede il supporto nativo per il sistema di comunicazione Wi-Fi, un sistema di crittografia a 128 Bit ed un supporto per schermi ad alta risoluzione (320 x 320 pixel). Inoltre, fruisce di un miglior supporto multimediale per video e audio. Si sono susseguite, nel tempo, diverse versioni, di cui l'ultima è la 5.4.5 che migliora il supporto per dispositivi bluetooth e anche il supporto per schermi di risoluzione maggiore. PalmOS 5 supporta, seppur in maniera limitata, il Multitasking.

Windows CE, nato nel 1996, è anch'esso uno tra i sistemi di maggior utilizzo. Si tratta di un sistema che tenta di emulare più possibile il Windows classico, in accordo alla filosofia Microsoft che si propone di far sì che l'utente veda il dispositivo portatile come non altro che un PC miniaturizzato, a differenza di Palm che, dal canto suo, cerca invece di renderlo differente e complementare al PC. Le interfacce utente sono molto semplici da utilizzare e sono ottimizzate per i dispositivi mobili. Il sistema, infatti, seppur tenti di mantenersi, a livello di interfaccia, simile a Windows classico, possiede un proprio kernel, cioè è un sistema operativo a sè, che non deve essere scambiato per una "riduzione" di Windows classico.

Dalla filosofia appena citata, derivano le caratteristiche di uno dei principali dispositivi Microsoft che utilizza Windows CE, Pocket PC, che utilizzeremo come piattaforma hardware d'esempio nel corso.

Rispetto a Palm OS, Windows CE richiede molta più memoria e processore, e ciò si riflette anche sul costo dei dispositivi. Il Casio E-200, ad esempio, dispone di un minimo di 64 MB di RAM e un processore da 206 Mhz. Il dispositivo Palm comune possiede, invece, 8 MB di RAM con un processore economico a 33 Mhz.

Tra gli altri sistemi operativi citiamo **BlackBerry** di Research In Motion e il sistema Open Source **GNU/Linux**.

◀◀ Capitolo 2: Primi passi ▶▶

◀◀ 2.1 Introduzione ▶▶

Il primo passo per creare un'applicazione grafica è creare una finestra che la contenga. Vedremo come realizzare una semplice finestra, gestirne i piccoli problemi ed infine provare le prime semplici interazioni tra l'utente e la finestra, tramite l'uso di strumenti quali la tastiera ed il mouse.

◀◀ 2.2 Un esempio di codice: Creare una finestra ▶▶

Questo paragrafo sarà, virtualmente, diviso in due parti. Nella prima descriveremo un codice "semplificato", tramite il quale arriveremo al risultato senza entrare troppo nel merito ed utilizzando funzioni di alto livello, semplici e intuitive. Nella seconda parte vedremo un codice più complesso ma molto più approfondito, per lo studente che volesse entrare più nel merito e gestire la propria applicazione fin nei minimi particolari.

Partiamo quindi dalla base, cioè vedremo come far visualizzare una semplice finestra senza contenuto.

Useremo, anche in seguito, 2 librerie per le nostre applicazioni OpenGL® ES. La prima è la libreria libGLES_CM.lib, cioè la libreria principale di OpenGL® ES. La seconda è la libreria ug.lib. Per quanto riguarda quest'ultima, si tratta di una libreria specifica di

Vincent che permette di creare finestre e gestire varie funzioni in modo simile al pacchetto GLUT di OpenGL®. Questo pacchetto, appartenente ad OpenGL® classico, provvede molte funzioni di alto livello per la gestione del 3D. Non lo tratteremo in quest'ambito poichè la versione ES a tutt'oggi non prevede ancora un pacchetto simile.

Solitamente e' possibile collegare queste librerie al programma modificando delle opzioni di progetto nell'ambiente utilizzato, ma lo stesso risultato si può ottenere anche, più facilmente, utilizzando la direttiva *pragma*.

Per collegare una libreria è necessario utilizzare un comando con la sottostante sintassi:

```
#pragma comment(lib, "NOME DELLA LIBRERIA")
```

NB: Ogni implementazione di C o C++ supporta caratteristiche uniche legate alla macchina su cui si lavora o al sistema operativo. Per esempio, alcuni programmi necessitano un controllo preciso sulle aree di memoria in cui vengono memorizzati i dati oppure la gestione completa del come alcune funzioni ricevano parametri. Le direttive **pragma** offrono una strada per gestire caratteristiche specifiche della macchina in uso o del sistema operativo, il tutto mantenendo completa compatibilità con i meccanismi di C e C++. Essendo tali, le direttive pragma sono differenti per ogni compilatore.

Codice
<pre>#pragma comment(lib, "libGLES_CM.lib") #pragma comment(lib, "ug.lib")</pre>

Il file necessario per l'uso di OpenGL® ES è *GLES/gl.h*. Altre funzioni possono essere trovate nel file *GLES/egl.h*.

La libreria **Vincent** comprende il file *ug.h*, che è necessario per il funzionamento di tutte le funzioni con prefisso *ug*. Esso include il *GLES/egl.h* è quindi è l'unico che è necessario includere. EGL è ciò che crea una connessione tra OpenGL® ES e il sistema operativo a finestre sottostante, di modo da permetterne la gestione.

Codice
<pre>#include "ug.h"</pre>

Il codice di inizializzazione si troverà nella funzione sottostante: **Inizializza**. In essa o nelle prime righe della funzione **main** è possibile la dichiarazione delle variabili.

Codice
<pre>void Inizializza (){ }</pre>

La funzione **Disegna**, qui dichiarata, è quella che viene richiamata per il disegno di ogni frame, quindi tutto il codice relativo al disegno va inserito in essa. Necessita un parametro in input di tipo *UGWindow*, che rappresenta la finestra di programma.

NB: Per chiarire come OpenGL® ES mostra la grafica a schermo bisogna aver chiaro il concetto di **frame**. Un film non è altro che una sequenza di immagini che vengono fatte scorrere velocemente sullo schermo del cinema, dando l'illusione del

movimento di ciò che rappresentano. Il modo in cui vengono visualizzate immagini e animazioni in OpenGL® ES è lo stesso. Ogni immagine della sequenza viene chiamata frame ed **FPS** (Frame Per Second) viene detto il numero di frame mostrati a schermo ogni secondo. Per ognuno di questi frame è necessario programmare nel dettaglio cosa in esso debba venire mostrato.

Codice

```
void Disegna (UGWindow Finestra) { }
```

Passiamo quindi alla funzione principale, il **main**.

La prima variabile necessaria è la *UGCtx*. L'*UGCtx* è un gestore (handle) che gestisce il motore principale (UG) di Vincent. La funzione **ugInit** inizializza il motore e restituisce un gestore che può essere assegnato alla variabile *UGCtx*.

Codice

```
int main()
{
    UGCtx HandleMotoreUg= ugInit();
```

Il prossimo passo è creare una finestra tramite la funzione **ugCreateWindow**. Come mostrato precedentemente nella funzione **Disegna**, una variabile di tipo *UGWindow* viene usata per conservare il gestore di una finestra OpenGL® ES.

La funzione accetta come parametri:

- UGCtx *ug*: Il gestore precedentemente dichiarato
- const *char* **config*: Specifica varie opzioni, tra le quali l'abilitazione di determinati buffer. Ci occuperemo meglio di ciò nei prossimi capitoli. Per il momento passiamo una stringa vuota.
- const *char* **title*: Specifica quale testo vada mostrato nella barra del titolo della finestra.
- int *width* & int *height*: Specifica, rispettivamente, larghezza e altezza della finestra.
- int *x* & int *y*: Specifica le coordinate alle quali deve trovarsi l'angolo in alto a sinistra della finestra al momento dell'apparizione, cioè, più intuitivamente, identifica la posizione della finestra.

Codice

```
UGWindow Finestra= ugCreateWindow(HandleMotoreUg, "", "OpenGL ES", 250, 250, 100, 100);
```

Richiamiamo quindi l'**Inizializza** precedentemente dichiarato, di modo da effettuare le dovute inizializzazioni.

Codice

Inizializza ();

Successivamente all'inizializzazione, è necessario specificare cosa vada disegnato nella finestra. Ciò avviene richiamando la funzione di disegno, precedentemente dichiarata, tramite la **ugDisplayFunc**, che prende in ingresso la funzione suddetta e il gestore della finestra.

Codice

ugDisplayFunc (Finestra, Disegna);

Per impedire al programma di terminare immediatamente dopo la conclusione del **main**, abbiamo bisogno di un qualche tipo di ciclo che lo tenga attivo. Esso viene chiamato *Main Loop*. Questo ciclo continua a rimanere attivo senza mai interrompersi, gestendo i messaggi del programma, o dell'utente, al loro arrivo. Lo richiamiamo tramite la funzione **ugMainLoop** che richiede un solo parametro: il gestore della finestra.

Il restituire il valore zero al termine dell'esecuzione del **main** è dovuto solo ad una convenzione che, in generale, le funzione C rispettano: restituire il valore zero significa aver portato correttamente a termine l'esecuzione della funzione, restituire -1 o un qualsiasi altro valore significa che si è verificato un errore che a volte è identificabile tramite il valore restituito.

Codice

ugMainLoop (HandleMotoreUg); return 0; }
--

Eseguendo il programma si potrà avere l'impressione che non succeda nulla, questo perchè non è stato ancora specificato cosa disegnare nella finestra creata e quindi essa eredita come contenuto quello della schermata precedente.



Fig 2.1 Risultato finale. La finestra creata ha ereditato il contenuto della schermata precedente.

E' possibile creare la stessa finestra anche **non** utilizzando la funzione **ugCreateWindow**, bensì sfruttando i più classici mezzi provvisti dalla libreria. Il processo è molto più lungo e complesso, ma lo riporteremo per approfondimento dei meccanismi di creazione di una finestra in OpenGL® ES, che, naturalmente, vanno ben oltre il richiamo di una semplice funzione di libreria.

In questo caso il codice si presenterà nel seguente modo.

Anzitutto richiamiamo le varie librerie necessarie includendo quella per gestire le chiamate di sistema di Windows.

Codice
<pre>#pragma comment(lib, "libGLES_CM.lib") #pragma comment(lib, "ug.lib") #include "ug.h" #include "windows.h"</pre>

Poiché stiamo scrivendo software dipendente dal dispositivo (**PDA**) che lo utilizzerà dobbiamo preoccuparci delle sue intrinseche limitazioni. Facciamo una piccola osservazione che può essere utile ad ottimizzare la gestione dei numeri in virgola mobile.

I PDA non hanno un'unità **FPU** (Floating Point Unit, cioè l'unità di calcolo che gestisce i numeri in virgola mobile) quindi tutte le operazioni in virgola mobile vengono simulate tramite la **CPU** (Central Processing Unit, cioè il processore principale). Per gestire i numeri con la virgola, il PDA utilizza i reali nel formato a virgola fissa. Per gestire un numero a virgola fissa abbiamo bisogno soltanto di un intero della stessa dimensione (in bytes) di un normale intero. I primi 16 bit rappresenteranno la parte intera del numero e gli ultimi 16 la parte reale, cioè dopo la virgola. Ciò significa una minore precisione, ma è sicuramente meglio di simulare l'FPU con la CPU. Per convertire un numero intero in un numero a virgola fissa basta spostare i suoi bit a sinistra, come fa la funzione **FixedFromInt**. OpenGL® ES provvede tutto un insieme di funzioni che lavorano con i numeri a virgola fissa (*Glfixed*), che sono disponibile tramite l'estensione *OES_fixed_point extension*.

Codice
<pre>#define PRECISION 16 #define ONE (1 << PRECISION) #define ZERO 0</pre>

Soffermiamoci un momento sui concetti di Device e Rendering Context.

Il **Device Context** (DC) è una struttura dati mantenuta internamente dal sistema grafico, dove per "internamente" intendiamo che all'utente non è possibile accedervi direttamente. Ogni DC è associato ad un particolare dispositivo di visualizzazione e nel caso di uno schermo è, solitamente, associato ad una finestra. Le funzioni di disegno necessitano di riferirsi ad un DC, il quale determina come esse si comportino: ad esempio, quale colore vada usato per il disegno, quale font per il testo, quale area può essere sfruttata per il disegno nella finestra attuale.

Il **Rendering Context** (RC) è una struttura molto simile al DC, con la differenza che il DC contiene informazione pertinenti ai componenti grafici del sistema operativo, mentre l'RC contiene informazioni pertinenti a OpenGL® ES. Ad ogni chiamata al GDI (Graphics Display Interface, un set di API grafiche utilizzate per la renderizzazione di grafica 2D) corrisponde un DC. Ad ogni DC corrisponde un RC. Ogni programma in OpenGL® ES è collegato ad un RC. Esso è ciò che collega le chiamate di OpenGL® ES al DC.

Definiamo, quindi, i gestori globali. Useremo *hInst* per conservare l'istanza della nostra applicazione, *hWnd* per la nostra finestra, *hDC* per il device context relativo alla finestra. Solitamente, il Rendering Context in OpenGL® ES viene chiamato *hRC*. Per far sì che il nostro programma disegni in una finestra, avremo bisogno di creare un Device Context, che viene chiamato *hDC*, ed ha il compito di connettere la finestra alla GDI.

Codice
<pre>HINSTANCE hInst; HWND hWnd; HDC hDC;</pre>

Definiamo poi le variabili necessarie per identificare rispettivamente lo schermo, la superficie di disegno e il Rendering Context.

Codice
<pre>EGLDisplay glesDisplay; EGLSurface glesSurface; EGLContext glesContext;</pre>

Diamo poi un nome all'applicazione. definendo così anche la didascalia della finestra.

Codice
<pre>TCHAR szAppName[] = L"OpenGL ES";</pre>

Definiamo i prototipi delle varie funzioni che utilizzeremo.

La ragione per cui dobbiamo farlo, ad esempio per **WndProc**, è che la funzione **CreateGLWindow**, che utilizzeremo nel seguito, si riferisce a **WndProc**, che verrà definita dopo **CreateGLWindow** nel codice. In C per l'accesso a procedure o sezioni di codice definite dopo la funzione che vi vuole accedere, è necessario almeno dichiarare la sezione/funzione in causa nella sezione di dichiarazioni del programma.

Codice
<pre>inline GLfixed FixedFromInt(int Valore) {return Valore << PRECISION;}; int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, LPTSTR lpCmdLine,int nCmdShow); LRESULT CALLBACK WndProc(HWND hWnd, UINT Messaggio, WPARAM wParam, LPARAM lParam);</pre>

Definiamo poi le funzioni, precedentemente viste, **Disegna** e **Inizializza**, nonché una funzione **Clear** che si occuperà di gestire le risorse rimaste inutilizzate, liberandole per un successivo utilizzo.

Occupiamoci per prima cosa delle inizializzazioni, cioè della funzione **Inizializza**.

Codice

```

BOOL Inizializza ()
{
    EGLConfig Configurazioni [10];
    EGLint ConfigurazioniCorrette;

```

configAttribs è una lista di interi che contiene il formato desiderato del frame buffer. Impostiamo quindi i dati per avere un frame buffer con colori a 24 bit e 16 bit di z-buffer. Vogliamo anche avere un window buffer.

NB: Un **buffer** (cuscinetto) è un'area particolare della memoria in cui l'informazione rimane in attesa di essere smistata verso un dispositivo o un applicativo che deve elaborarla. Un buffer viene, ad esempio, utilizzato per operazioni del tipo copia/incolla.

Codice

```

const EGLint configAttribs[] =
{
    EGL_RED_SIZE, 8,
    EGL_BLUE_SIZE, 8,
    EGL_ALPHA_SIZE, EGL_DONT_CARE,
    EGL_DEPTH_SIZE, 16,
    EGL_STENCIL_SIZE, EGL_DONT_CARE,
    EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
    EGL_NONE, EGL_NONE
};

```

Recuperiamo il Device Context ed eseguiamo, quindi, la richiesta necessaria per avere la disponibilità dello schermo.

Codice

```

hDC = GetWindowDC(hWnd);
glesDisplay = eglGetDisplay(hDC);

```

Inizializziamo lo schermo.

Codice

```

if(!eglInitialize(glesDisplay, NULL, NULL))
    return FALSE;

```

Impostiamo la configurazione del frame buffer migliore per le nostre esigenze. Vogliamo un massimo di 10 configurazioni.

Codice

```

if(!eglChooseConfig(glesDisplay, configAttribs, &Configurazioni[0], 10, &
ConfigurazioniCorrette))
    return FALSE;

```

Se non c'è una configurazione abbastanza buona per noi, allora chiudiamo il programma.

Codice
<pre>if (ConfigurazioniCorrette < 1) return FALSE;</pre>

Utilizziamo ora la funzione **eglCreateWindowSurface** per creare una superficie per *EGL* e restituirne il gestore. Ogni Rendering Context creato con una *EGLConfig* compatibile può essere usato per disegnare su detta superficie.

Codice
<pre>glesSurface = eglCreateWindowSurface(glesDisplay, Configurazioni[0], hWnd, configAttribs); if(!glesSurface) return FALSE;</pre>

Creiamo quindi il Rendering context e controlliamo che sia stato creato correttamente, altrimenti usciamo dal programma.

Codice
<pre>glesContext = eglCreateContext(glesDisplay,Configurazioni[0],0,configAttribs); if(!glesContext) return FALSE;</pre>

Attiviamo il contesto per il disegno.

Codice
<pre>eglMakeCurrent(glesDisplay, glesSurface, glesSurface, glesContext);</pre>

Per ora fermiamoci qui.

Codice
<pre>return TRUE; }</pre>

Stabiliamo ora cosa disegnare nella finestra. Per adesso la lasceremo vuota.

Codice
<pre>void Disegna { }</pre>

Definiamo la funzione che si occupa di rilasciare le risorse a fine lavoro.

Codice
<pre>void Clear() {</pre>

Se è stato assegnato l'uso dello schermo, lo liberiamo e facciamo lo stesso con i contesti creati.

Codice
<pre>if(glesDisplay) { eglMakeCurrent(glesDisplay, NULL, NULL, NULL); if(glesContext) eglDestroyContext(glesDisplay, glesContext); if(glesSurface) eglDestroySurface(glesDisplay, glesSurface); eglTerminate(glesDisplay); }</pre>

Distruggiamo infine la finestra e deregistriamo la sua classe. Ciò ci permette di distruggere correttamente la finestra, e quindi riaprirne un'altra senza ricevere il messaggio d'errore *"Windows Class already registered"*.

Codice
<pre>DestroyWindow(hWnd); UnregisterClass(szAppName, hInst); }</pre>

Passiamo ora alla funzione principale, in cui creeremo la finestra di disegno, inizializzeremo OpenGL® ES e infine, eseguito il programma, rilasceremo tutte le risorse utilizzate.

Codice
<pre>int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int nCmdShow){</pre>

Creiamo dunque la variabile necessaria per il message loop, ovvero il ciclo in cui ci si occuperà di ricevere e gestire i messaggi.

NB: Chiamiamo **evento** un segnale dato in input al (o output dal) programma, che può essere il click su un tasto del mouse, la pressione di una tasto sulla tastiera, lo scattare di un timer etc etc. Quando accade un evento, il sistema operativo mette, nella coda dedicata ai messaggi della finestra alla quale l'evento è relativo, un **messaggio**. Esso viene, cioè, messo in attesa di essere elaborato. Si hanno ora due scelte: o lasciar gestire il messaggio dal sistema operativo oppure farlo gestire dalla propria applicazione.

Codice
MSG msg;

Creiamo una struttura che contenga alcuni dei valori di inizializzazione della nostra finestra.

Codice
WNDCLASS WClass;

Inizializziamo le variabili locali.

Codice
hInst = hInstance; bool done = FALSE;

Occupiamoci quindi di far sì che non vi sia mai più di una singola istanza della nostra applicazione. Per prima cosa, assicuriamoci del fatto che la finestra che vogliamo creare non esista già, e in quel caso, invece di crearne una nuova, semplicemente mettiamo in primo piano quella esistente.

Per impostare la selezione sulla finestra eventualmente trovata, utilizzeremo la funzione **SetForegroundWindow**. Il valore `"/ 0x00000001"` viene usato per portare la finestra in primo piano e selezionarla.

Codice
<pre>if(hWnd = FindWindow(szAppName, szAppName)) { SetForegroundWindow((HWND)((ULONG) hWnd 0x00000001)); return 0; }</pre>

Affermiamo ora alcune proprietà della finestra tramite la variabile *WClass*.

Forziamo il ridisegno della finestra in caso di ridimensionamento, sia orizzontalmente che verticalmente.

Codice
WClass.style = CS_HREDRAW CS_VREDRAW;

Definiamo un puntatore di funzione, cioè una variabile che indichi quale funzione debba essere richiamata dal sistema operativo nel caso in cui venga generato un determinato messaggio.

Codice
WClass.lpfnWndProc = (WNDPROC) WndProc;

Vogliamo poi che non siano tenuti in considerazione dati provenienti da altre finestre, quindi mettiamo a zero i due campi *cbClsExtra* e *cbWndExtra*.

Codice
<pre>WClass.cbClsExtra = 0; WClass.cbWndExtra = 0;</pre>

Specifichiamo l'istanza.

Codice
<pre>WClass.hInstance = hInstance;</pre>

Dotiamo la nostra finestra di un'icona che, nel nostro caso, sarà quella standard.

Codice
<pre>WClass.hIcon = LoadIcon(hInstance, NULL)</pre>

Anche per il mouse desideriamo utilizzare la freccia standard.

Codice
<pre>WClass.hCursor = 0;</pre>

Il colore di sfondo non ha importanza per ora.

Codice
<pre>WClass.hbrBackground = 0;</pre>

Non vogliamo inserire un menù.

Codice
<pre>WClass.lpszMenuName = NULL;</pre>

Inseriamo quindi il nome della classe. Non necessariamente esso deve coincidere con il nome dell'applicazione, ma di norma è così.

Codice
<pre>WClass.lpszClassName = szAppName;</pre>

Prima di creare la finestra, registriamo la classe della nuova finestra.

Codice
<pre>if(!RegisterClass(&WClass)) return FALSE;</pre>

Creiamo, infine, la finestra tramite la funzione **CreateWindow**. Essa prende vari parametri:

Codice
<pre>hWnd=CreateWindow(szAppName, //Nome della classe szAppName, //Stringa della didascalia WS_VISIBLE, //Stile della finestra CW_USEDEFAULT,CW_USEDEFAULT, //Posizione iniziale [x,y] CW_USEDEFAULT, CW_USEDEFAULT, //Larghezza e altezza NULL, NULL, //Finestra che la contiene e gestore del menu hInst, NULL); //Gestore dell'istanza e valore personalizzato //da passare al messaggio WM_CREATE</pre>

Controlliamo che la finestra sia stata correttamente creata. In questo caso, *hWnd* conterrà il gestore della finestra. Se la finestra non è stata generata, facciamo sì che il programma termini. Lo stesso accade se l'inizializzazione di OpenGL® ES non è stata eseguita correttamente.

Codice
<pre>if(!hWnd) return FALSE; if(!Inizializza ()) return FALSE;</pre>

Quindi mettiamo la finestra in primo piano, la selezioniamo ed aggiorniamo.

Codice
<pre>ShowWindow(hWnd, nCmdShow); UpdateWindow(hWnd);</pre>

Chiamiamo ora quindi il **message loop**, nel quale eventuali messaggi entranti verranno riconosciuti e gestiti. Il ciclo continuerà fino a che la variabile *done* sia uguale a False.

Codice
<pre>while(!done) {</pre>

Anzitutto controlliamo se ci siano messaggi della finestra in attesa. Usando la funzione **PeekMessage**, possiamo controllare se ci sono messaggi senza fermare il nostro programma.

Codice
<pre>if(PeekMessage(&msg,NULL,0,0,PM_REMOVE)) //se c'è un messaggio in attesa</pre>

```
{
```

Controlliamo ora se è stato lanciato un messaggio di uscita. Nel caso si tratti proprio di esso (*WM_QUIT*) allora impostiamo la variabile *done* a True di modo da far terminare il message loop e chiudere il programma.

Codice

```
if(msg.Messaggio == WM_QUIT)
    done=TRUE;
    else
    {
```

Se il messaggio non è un messaggio di uscita, interpretiamo il messaggio e lo inviamo, in modo che **WndProc** o Windows possano occuparsene.

Codice

```
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    else
```

Se non ci sono messaggi disegniamo la nostra scena OpenGL® ES.

Codice

```
        Disegna();
    }
```

In caso di uscita dal message loop rilasciamo le risorse e terminiamo il programma.

Codice

```
    Clear();
    return 0;
}
```

Occupiamoci quindi di tutti i messaggi della finestra. Quando abbiamo registrato la classe della finestra, abbiamo fatto in modo di far saltare il programma a questa sezione di codice per occuparsi dei messaggi della finestra.

Codice

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT Messaggio, WPARAM wParam,
LPARAM lParam)
{
```

La variabile *messaggio* contiene il nome del messaggio di cui vogliamo occuparci. Preoccupiamoci quindi ora di identificare i messaggi e gestirli.

Codice
<pre>switch (Messaggio) {</pre>

Nel caso si riceva un messaggio di disegno...

Codice
<pre>case WM_PAINT: ValidateRect(hWnd,NULL); //Serve per evitare altri messaggi WM_PAINT return 0;</pre>

Nel caso si riceva un messaggio di uscita mandiamo un messaggio di uscita e chiudiamo il programma.

NB: Nell'uso dell'emulatore PocketPc2003 è consigliato provvedere sempre un metodo di **uscita** dal programma. La chiusura forzata tramite la X in alto a destra della finestra PocketPc, infatti, non rilascia le risorse, e impedisce una successiva compilazione del programma, poichè l'eseguibile sull'emulatore non verrà copiato, in quanto ritenuto ancora in uso. Ciò costringe l'utente a chiudere l'emulatore e doverlo rieseguire, nonché copiare di nuovo la dll in esso ad ogni compilazione.

Codice
<pre>case WM_DESTROY: PostQuitMessage(0); return 0; };</pre>

Esistono molti altri messaggi che è possibile gestire, ad esempio:

WM_ACTIVATE: Indica la richiesta di controllo dell'attività della finestra (ad esempio, se è minimizzata o meno).

WM_SYSCOMMAND: Indica che è stata fatta una chiamata di sistema. La variabile *wParam* (richiesta in input da **WndProc**) conterrà informazioni atte a identificare la chiamata. Ad esempio, il valore **SC_SCREENSAVE** indica che sta partendo uno screensaver, mentre **SC_MONITORPOWER**, indica, invece, che il video sta provando ad entrare nella modalità di risparmio energetico.

WM_KEYDOWN: Se un tasto viene premuto, è possibile identificarlo controllando il valore di *wParam*. Un modo corretto di gestire questo messaggio è adottare un array di 256 elementi. Ogni tasto sulla tastiera può essere rappresentato da un numero da 0-255. Ogniqualvolta un tasto venga premuto impostiamo a TRUE l'elemento corrispondente nell'array. Ad esempio,

quando premiamo il tasto che rappresenta il numero 40, `array[40]` verrà impostato a `TRUE`. Quando esso viene rilasciato, diventerà `FALSE`. Ciò permette di gestire la pressione contemporanea dei tasti.

WM_KEYUP: Come sopra, nel caso in cui un tasto sia stato rilasciato.

WM_SIZE: Indica che la finestra è stata ridimensionata. Per sapere le nuove dimensioni della finestra leggeremo i valori *HIWORD* e *LOWORD* di *lParam*.

Tutti i messaggi di cui non ci occupiamo saranno passati a **DefWindowProc** in modo che il sistema operativo se ne possa occupare.

Codice
<pre>return DefWindowProc(hWnd, Messaggio, wParam, lParam); }</pre>

Il risultato finale è lo stesso della versione precedente.

Ora abbiamo le conoscenze necessarie per creare una semplice finestra OpenGL® ES. Eseguito il programma sembrerà non sia successo nulla, questo perché non abbiamo ancora specificato cosa disegnare nella finestra creata. Si noterà, inoltre, come, premendo il tasto OK nell'angolo in alto a destra, il programma continui a girare. Nei prossimi paragrafi verrà spiegato come uscire dal programma utilizzando la tastiera o il mouse.

« 2.3 Mantenere le proporzioni della finestra »

Esiste la possibilità che la finestra creata venga ridimensionata dall'apparire di una nuova finestra sullo schermo. Ad esempio, richiamando la tastiera del simulatore PocketPc, essa si posizionerà nella parte bassa dello schermo, occupando parte dello spazio della finestra. Il risultato è che la nostra finestra verrà spostata verso l'alto con tutto il suo contenuto. Tale condizione non è desiderabile poiché impedisce la corretta visualizzazione dei contenuti della finestra. Mostriamo in figura un esempio del problema riscontrato nel caso del disegno di una semplice figura su schermo. Rimandiamo maggiori informazioni sulla creazione dell'esempio al seguito.

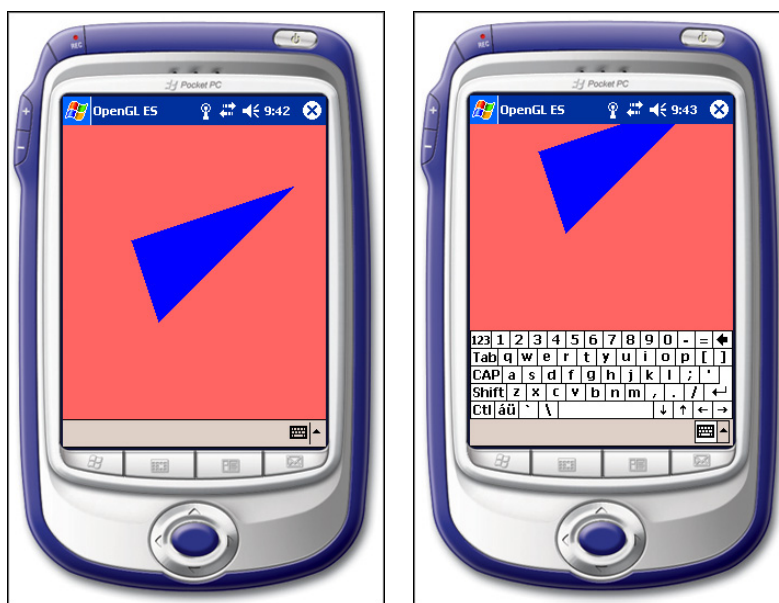


Fig 2.2 Un esempio di come l'immagine viene tagliata dall'apparire della tastiera.

Anzitutto, creiamo una funzione **Aggiorna** che si occupi di aggiornare il contenuto della finestra in caso di ridimensionamento. Essa avrà come parametri la finestra da aggiornare, la sua larghezza e la sua altezza. In essa dovranno, naturalmente, essere previste anche le trasformazioni di vista, come vedremo più nel dettaglio nel seguito.

Per garantire un corretto aggiornamento utilizzeremo la funzione **glViewport**. Essa ha lo scopo di specificare posizione e dimensione dello spazio riservato al disegno (viewport) nella finestra da noi creata. Questi i suoi parametri:

- GLint *x*: posizione x dell'angolo in basso a sinistra dello spazio di disegno desiderato
- GLint *y*: posizione y dell'angolo in basso a sinistra dello spazio di disegno desiderato
- GLsizei *width*: Larghezza dello spazio di disegno desiderato
- GLsizei *height*: Altezza dello spazio di disegno desiderato

Esempio

```
void Aggiorna(UGWindow Finestra, int Larghezza, int Altezza)
{
    [...]
    glViewport(0, 0, Larghezza, Altezza);
    [...]
}
```

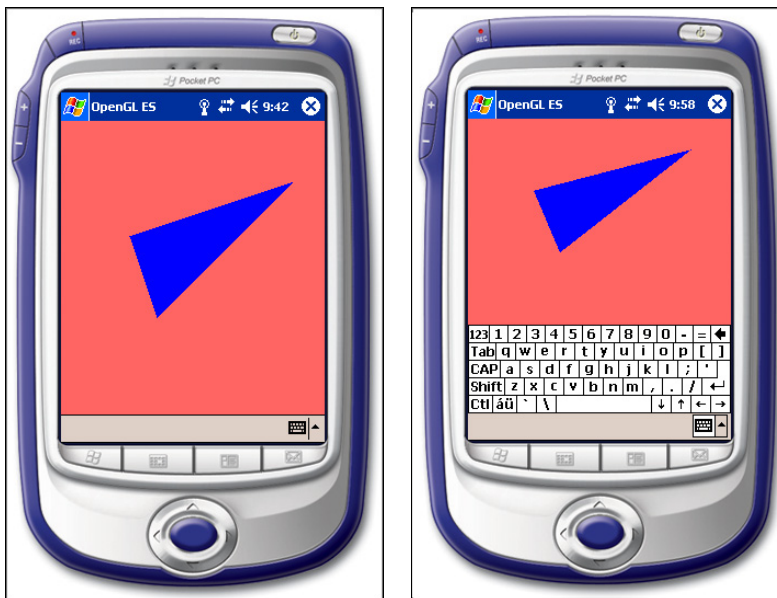


Fig 2.2 Risultato dopo l'aggiunta dell'aggiornamento.

2.4 Utilizzare la tastiera e il mouse

Creata una finestra, ci poniamo il problema di interagire con essa per le operazioni basilari come la chiusura della stessa. Vedremo, quindi, come eseguire questa operazione utilizzando il mouse e la tastiera.

Per prima cosa, dovremo creare delle funzioni che gestiscano gli input ricevuti rispettivamente da mouse e tastiera. Dette funzioni necessiteranno come parametri, nel caso della tastiera, di:

- la finestra corrente, rappresentata da una variabile di tipo *UGwindow*
- il tasto premuto, rappresentato da un intero
- le coordinate *x* e *y* del puntatore del mouse al momento della pressione del tasto

Esempio

```
void Tastiera(UGWindow Finestra, int TastoPremuto, int x, int y)
```

```
{
```

Verifichiamo quindi quale tasto sia stato effettivamente premuto.

Esempio

```
switch(TastoPremuto)
```

Possiamo, quindi, confrontare la variabile *tastoPremuto* con le rappresentazioni dei vari caratteri di tastiera in modo da riconoscere quale sia il tasto che è stato effettivamente premuto. Ad esempio, nel caso sia stato premuto il tasto *e* o il tasto *Fine* reagiamo con l'uscita dal programma.

Esempio

```
{
    case 'e' : PostQuitMessage(0); break;
    case UG_KEY_END : PostQuitMessage(0); break;
}
```

E' infatti possibile riconoscere, oltre ai normali caratteri alfanumerici, tutti i tasti speciali come le frecce o i tasti funzione, i quali sono rappresentati da particolari codici:

Flag	Descrizione
UG_KEY_F1 - UG_KEY_F2	Tasti funzione da F1 a F12
UG_KEY_LEFT	Freccia sinistra
UG_KEY_RIGHT	Freccia destra
UG_KEY_UP	Freccia su
UG_KEY_DOWN	Freccia giu
UG_KEY_PAGE_UP	Tasto Pagina su
UG_KEY_PAGE_DOWN	Tasto Pagina giu
UG_KEY_HOME	Tasto Home
UG_KEY_END	Tasto Fine
UG_KEY_INSERT	Tasto Ins

Nel caso del mouse, dovremo creare un'analoga funzione che necessiterà come parametri:

- la finestra corrente, rappresentata da una variabile di tipo *UGwindow*
- il tasto premuto, rappresentato da un intero oppure da *UG_BUT_LEFT*, *UG_BUT_MIDDLE* o *UG_BUT_RIGHT*, che rappresentano rispettivamente, il tasto destro, centrale e sinistro del mouse
- lo stato del tasto, rappresentato da un intero o da *UG_BUT_DOWN* o *UG_BUT_UP*, che corrispondono rispettivamente agli stati di rilasciato e premuto
- le coordinate *x* e *y* del puntatore del mouse al momento della pressione del tasto

Ad esempio, facciamo in modo che alla pressione di un qualsiasi tasto si ottenga l'uscita dal programma.

Esempio

```
void Mouse(UGWindow Finestra, int Bottone, int Stato, int x, int y)
```

```
{
    PostQuitMessage(0);
}
```

Definite le funzioni, va comunicato al sistema di OpenGL® quale esse siano. Per fare questo utilizziamo le funzioni **ugKeyboardFunc** e **ugPointerFunc** allo stesso modo in cui abbiamo utilizzato la funzione **ugDisplayFunc**. Queste funzioni associano le funzioni che gestiscono tastiera e mouse al gestore della finestra.

Esempio

```
ugKeyboardFunc(Finestra, Tastiera);
ugPointerFunc(Finestra, Mouse);
```

E' possibile ottenere lo stesso risultato sfruttando il sistema dei messaggi. Ogni qualvolta accada un evento, come già visto, la finestra riceve dei messaggi dal sistema operativo, che avevamo imparato a gestire, nel precedente capitolo, tramite la funzione **WndProc**. Vediamo, quindi, quali sono i messaggi che vengono inviati nel caso di interazione con il mouse e la tastiera.

Nel caso del mouse riconosciamo i seguenti:

Messaggio	Descrizione
WM_LBUTTONDOWN	Pressione del tasto sinistro
WM_LBUTTONUP	Rilascio del tasto sinistro
WM_CONTEXTMENU	Pressione del tasto destro. Solitamente una finestra risponde a questo messaggio con la visualizzazione di un menu a tendina che contiene delle scorciatoie, tramite l'uso della funzione TrackPopupMenu
WM_RBUTTONDOWN	Pressione del tasto destro
WM_RBUTTONUP	Rilascio del tasto destro
WM_MBUTTONDOWN	Pressione del tasto di centro
WM_MBUTTONUP	Rilascio del tasto di centro
WM_MOUSEWHEEL	Pressione della rotella
WM_MOUSEMOVE	Movimento del cursore sulla finestra

Ogni messaggio relativo agli eventi del mouse porta con sé due parametri: *lparam* e *wparam*. Il primo di essi, *lparam*, contiene la posizione del cursore al momento dell'evento, la quale è recuperabile nei suoi parametri x e y tramite il seguente codice:

Esempio

```
xPos = LOWORD(lParam);
yPos = HIWORD(lParam);
```


Le posizioni x e y sono da intendersi relative all'area della finestra, cioè prendendo come origine del sistema cartesiano l'angolo in alto a sinistra della finestra corrente.

Il parametro *wparam* invece contiene dei flag che indicano la condizione degli altri stati del mouse e dei tasti CTRL e SHIFT, al momento dell'evento. Ciò ha lo scopo di permettere il riconoscimento di determinate combinazioni di tasti. Riportiamo la tabella contenente i vari valori assumibili da *wparam*:

NB: In informatica per **flag** (bandierina) si intende, solitamente, una variabile che può assumere solo due valori (vero/falso, on/off, 1/0, acceso/spento) e che segnala, con il suo stato, se un dato evento si è verificato oppure no, o se il sistema si trova in un certo stato o no. In OpenGL® spesso ritroviamo l'uso di flag, interni al linguaggio, che identificano l'uso di un certo metodo di sfumatura del colore, o una certa proprietà di illuminazione, etc..

Valore	Descrizione
MK_CONTROL	Pressione del tasto CTRL
MK_LBUTTON	Pressione del tasto sinistro del mouse
MK_MBUTTON	Pressione del tasto centrale del mouse
MK_RBUTTON	Pressione del tasto destro del mouse
MK_SHIFT	Pressione del tasto SHIFT

Nel caso della tastiera il discorso è più complesso. La pressione e il rilascio, rispettivamente, di un tasto generano i messaggi WM_KEYDOWN/WM_SYSKEYDOWN e WM_KEYUP/WM_SYSKEYUP. Il prefisso SYS identifica la pressione di una combinazione di pressione di un tasto e in contemporanea del tasto **ALT**. Questa distinzione deriva dal fatto che la pressione dell'ALT ha come risultato che l'evento venga gestito prima dal sistema che da qualsiasi altra applicazione, di modo da permettere il richiamo di funzioni di sistema primarie come nel caso della pressione della combinazione ALT+CTRL+CANC che permette l'eventuale chiusura forzata, o comunque la gestione, di un programma in stato di errore. A questo punto, è ovvio come riconoscere gli input provenienti dalla tastiera. Quando si riceve un messaggio di tipo WM_KEYDOWN è sufficiente esaminare il codice che lo accompagna per determinare quale tasto sia stato premuto. In particolare, come avveniva anche nel caso mouse, il messaggio WM_KEYDOWN è accompagnato da due parametri: *wparam*, il quale contiene il codice del tasto premuto, ed *lparam*, che contiene le seguenti informazioni.

Numero di bit	Descrizione
0–15	Specifica quante volte il tasto sia stato premuto. Quando un tasto viene tenuto premuto vengono inviati vari messaggi WM_KEYDOWN. In questo modo essi vengono contati, il che dà un modo per capire quanto a lungo un tasto sia stato premuto.
16-23	Specifica il codice di tastiera. I codici dipendono dal sistema operativo. Nell' Appendice C.7 sono riportati i codici nel caso di Window CE.
29	Il valore di questo bit è 1 se assieme al tasto è stato premuto il tasto ALT e 0 in caso contrario.
30	Il valore di questo bit specifica lo stato precedente del tasto: assume valore 1 se prima dell'arrivo del messaggio il tasto era premuto e 0 se era rilasciato.
31	Il valore di questo bit specifica lo stato attuale: assume valore 1 se il tasto è

stato rilasciato, 0 altrimenti.

All'atto pratico, all'interno della funzione **WndProc** dovremmo aggiungere un codice di questo tipo:

Esempio

```
case WM_KEYDOWN:
    switch (wParam)
    {
        case VK_HOME:
            // Inserire il codice da eseguire su pressione del tasto HOME
            break;

        case VK_END:
            // Inserire il codice da eseguire su pressione del tasto END
            break;

        default:
            // Inserire il codice di gestione di tutti gli altri casi
            break;
    }
```

Questo schema andrà ripetuto per ogni tasto che si voglia riconoscere. Riportiamo la tabella dei codici dei vari tasti nell'[Appendice C.7](#).

«« Capitolo 3: Il disegno »»

«« 3.1 Introduzione »»

Ora che abbiamo chiaro come realizzare una finestra e gestirla, vedremo come realizzare il disegno vero e proprio. Occupiamoci per ora del caso 2D.

Vedremo come OpenGL® gestisce la pipeline grafica, ovvero come si giunge dall'interpretazione del codice al disegno su schermo, nonché come ottenere le migliori prestazioni, tramite tecniche semplici quanto geniali, come può essere quella del double buffering. Osserveremo, poi, come impostare proprietà sensibili del disegno quali il colore, la dimensione, etc.. A questo proposito, un'osservazione da tenere sempre presente in seguito: OpenGL® ES, come OpenGL®, è a tutti gli effetti una macchina a stati. Una macchina di questo tipo è caratterizzata dal possedere diversi possibili stati: una volta che si entri in uno di essi, vi si rimane fino a quando non si passi ad un altro. OpenGL® ES adotta la stessa politica: ad esempio, la scelta di utilizzare un determinato colore implica l'entrata in uno stato, il che si traduce nel fatto che l'applicazione del colore venga eseguita su tutte le forme geometriche che verranno disegnate da quel momento in poi, fino a quando non si specifichi un nuovo colore, cioè fino a che non si cambi stato. Ciò varrà, in generale, per ogni definizione di proprietà, come nella trasparenza, l'effetto nebbia, l'illuminazione, le caratteristiche del materiale e così via.

«« 3.2 Disegno e frame »»

Prima di addentrarci in discorsi più complessi, analizziamo le basi del disegno su computer e quindi il sistema del disegno in frame, che abbiamo in parte già affrontato, e le sue implicazioni e problemi.

Come già osservato, un **frame** rappresenta una singola

NB: Un'immagine può essere

immagine in quel flusso continuo che altri non è che l'animazione. Eseguendo uno qualsiasi dei codici proposti nei precedenti capitoli è possibile notare come, nella finestra creata, venga visualizzato ciò che si poteva vedere prima della sua creazione nel punto esatto in cui si trova attualmente. Questo avveniva perchè non era stato ancora definito cosa disegnarvi (neanche a livello di un semplice colore di sfondo) e quindi essa non conteneva nulla di nuovo, che potesse sovrascrivere il contenuto delle schermate precedenti. Al momento di definire un disegno sull'area della finestra, sarà necessario avere il controllo dell'immagine visualizzata, ed averlo in ogni singolo momento. Ciò porta con sé il concetto di refresh ed i primi problemi.

rappresentata in diversi modi. Il più semplice è vederla come una matrice di piccoli quadrati (**pixel**), ognuno dei quali può assumere diversi colori. E' questa la visualizzazione più elementare, detta a **mappa di byte** o **byte-map**. In C/C++ essa può essere immagazzinata in forma di array bidimensionale (matrice) il cui *i*-esimo elemento sia un numero che rappresenta il colore del quadratino *i*-esimo (**pixel**).

Intendiamo con **refresh** l'atto con cui il contenuto dello schermo/finestra viene aggiornato in ogni unità di tempo per permetterne la visualizzazione continua e non statica. Consideriamo, ad esempio, lo schermo di un classico pc casalingo: ciò che è possibile vedervi, sebbene sembri un'immagine unica e fissa, in movimento dove e se necessario, non lo è di per sé. Per permettere la visualizzazione corretta di tutti i cambiamenti in modo fluido, lo schermo viene aggiornato un certo numero di volte al secondo (**FPS** = Frame Per Second) punto per punto, a partire dal primo dell'angolo in alto a sinistra (almeno nei sistemi detti *raster*, che sono i più comuni). Da notare, come il ridisegno continuo sia necessario, non solo per visualizzare elementi nuovi in movimento, bensì anche per eliminare vecchi elementi. Si pensi alla chiusura di una finestra in ambiente Windows: se non vi fosse questo aggiornamento, l'immagine della finestra rimarrebbe impressa sullo schermo nonostante non debba più esserlo, in quanto chiusa. La fluidità viene assicurata da un alto valore di FPS, ma non solo. Una gestione di questo tipo può dare luogo, infatti, a fenomeni di "sfarfallio", in cui l'occhio riesca a notare l'aggiornamento mentre viene eseguito, a causa del fatto che il disegno viene effettuato punto per punto.

Una tecnica che permette la risoluzione di questo problema è la tecnica detta del **double buffering**.

« 3.3 Double buffering »

Questa tecnica è ottima per la fluidità delle applicazioni di animazione. Come già visto, il disegno successivo di due frame provoca un fenomeno di sfarfallio. Per risolvere il problema, utilizzeremo due buffer sovrapposti su cui disegnare: uno solo dei due verrà visualizzato, mentre l'altro rimarrà nascosto. Il disegno punto per punto avverrà sul buffer nascosto, fino a che l'informazione dell'immagine prossima alla visualizzazione non sia stata completamente e correttamente scritta. A questo punto i buffer verranno scambiati, di modo da visualizzare sempre quello su cui il disegno è completo.

Ricordiamo che per buffer (cuscinetto) non intendiamo altro che un'area di memoria dedicata, in cui l'informazione rimane in attesa di essere smistata verso un applicativo che deve elaborarla.

Vediamone un esempio, mostrando gli essenziali tratti di codice necessari.

Dichiariamo, per prima cosa, il colore di sfondo della nostra finestra nella sezione delle inizializzazioni, tramite la funzione **glClearColor**. Il colore scelto sovrascriverà l'immagine della schermata precedente all'esecuzione del programma, che potevamo visualizzare sulla nostra finestra negli esempi dati nei precedenti capitoli.

La funzione **glClearColor/glClearColorx** accetta come parametri 4 valori variabili nell'intervallo [0,1], nello stile RGBA che descriveremo meglio in seguito:

- **GLclampf/GLclampx red**: Specifica la concentrazione di rosso nel colore di sfondo.

- GLclampf/GLclampx *green*: Specifica la concentrazione di verde nel colore di sfondo.
- GLclampf/GLclampx *blue*: Specifica la concentrazione di blu nel colore di sfondo.
- GLclampf/GLclampx *alpha*: Specifica la trasparenza del colore di sfondo.

Il colore specificato dalla **glClearColor** sarà quello che verrà utilizzato dalla funzione **glClear** per riempire il *color buffer*. OpenGL® ES prevede diversi buffer. Approfondiremo questo discorso in seguito.

Esempio	
	<pre>void Inizializza() { [...] glClearColor (0.87f, 0.87f, 0.87f, 0.0f); [...] }</pre>

A questo punto, andrà richiamata la funzione, già accennata, **glClear**, che disporremo all'interno della funzione di disegno.

La funzione accetta come parametro:

- GLbitfield *mask*: Specifica il buffer da riportare al colore di sfondo. Sono accettati i flag GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, GL_STENCIL_BUFFER_BIT.

Esempio	
	<pre>void Disegna(UGWindow Finestra) { [...] glClear(GL_COLOR_BUFFER_BIT); [...] }</pre>

A questo punto, va notata una questione tecnica: quando si intende eseguire un disegno in OpenGL® ES, i comandi dati vengono via via messi in coda per l'esecuzione in varie locazioni, come possono essere buffer di rete o acceleratori grafici. Per avere la sicurezza che vengano tutti eseguiti prima possibile, richiameremo la funzione **glFlush**, che ha proprio lo scopo di far eseguire tutti i comandi ancora in coda. A **glFlush** si accompagna la funzione **glFinish**, che ha lo stesso scopo, con la differenza che **glFinish** blocca l'esecuzione fino a che tutti i comandi non siano stati realmente eseguiti. **glFlush**, invece, garantisce l'esecuzione di essi il più presto possibile e in un tempo finito, ma non lascia il programma in attesa fino al reale svuotamento dei buffer dei comandi.

Esempio	
	<pre>glFlush();</pre>

A questo punto scambiamo i buffer. La funzione da utilizzare è la **ugSwapBuffers** che ha per parametro la finestra su cui vogliamo lavorare.

Esempio	
	<pre>ugSwapBuffers(Finestra); }</pre>

Siamo ora in grado di gestire il double buffering. Come esercizio, lasciamo al lettore l'esecuzione di una prova di disegno senza l'utilizzo di detta tecnica, di modo da notare la differenza provvista dal suo utilizzo. Un esempio di codice più completo verrà mostrato alla fine di questo capitolo.

''' 3.4 Coordinate '''

Prima di parlare di disegno in se, è importante capire attraverso quale processo una scena arriva ad essere visualizzata sullo schermo.

In generale, nella rappresentazione di una scena si passa attraverso vari sistemi di coordinate prima di ottenere la rappresentazione su schermo 2D finale.

Anzitutto, ogni singolo oggetto presente nella scena viene inserito in un proprio sistema di riferimento cartesiano, dove vengono ricavate le coordinate che, singolarmente, rappresentano la sua forma e le sue proporzioni. Esse vengono dette **coordinate locali** (local coordinates). In seguito, si passa alle **coordinate di mondo** (world coordinates), cioè ogni oggetto viene inserito nella scena, opportunamente posizionato e direzionato, insieme a tutti gli altri oggetti presenti sulla scena e se ne ricavano le coordinate globali.

Immaginiamo, ad esempio, di dover disegnare un albero, definendone, una per una, ogni singola componente: i rami, le foglie, il tronco. Per prima cosa, definiremo le coordinate locali, mettendo ogni singola componente in un sistema di riferimento a se. Ne è naturale conseguenza che oggetti identici (es. le foglie) potranno avere un'unica rappresentazione in coordinate locali. Fatto ciò, passeremo alle coordinate di mondo, mettendo ogni componente al proprio posto di modo da formare sulla scena l'albero completo di ogni sua parte.

A questo punto, le coordinate di mondo vengono sottoposte a diverse routine che si occupano di convertirle alle **coordinate di vista** (viewing coordinates), coordinate cioè che tengano in considerazione la posizione e l'orientamento di un'ipotetica telecamera da cui si stia osservando la scena. Questo processo viene comunemente detto **viewing pipeline**. A esso segue il calcolo delle **coordinate di proiezione** (projection coordinates), ovvero di quelle necessarie a rappresentare la scena 3D su di uno schermo piatto (2D), ed infine si passa alle **coordinate normalizzate** (normalized coordinates), ovvero si normalizzano i valori di modo tale da essere compresi nell'intervallo tra 0 e 1 (o tra 1 e -1 a seconda del sistema) in modo tale da aggirare problemi dovuti al dominio di valori assumibili dalle coordinate nel sistema corrente.

L'ultimo passo da fare è definire il piano di **clipping**, ovvero il taglio che vogliamo visualizzare della scena, ed infine convertire i valori nella forma richiesta dal dispositivo di output grafico per poter quindi definire le **coordinate su schermo** (device coordinates), cioè quelle che indicheranno propriamente dove l'immagine andrà visualizzata sullo schermo disponibile in tempo di run-time.

I pacchetti grafici permettono di controllare il posizionamento, all'interno della finestra di programma, anche utilizzando un'altra "finestra" detta **viewport**. Gli oggetti all'interno della finestra di clipping vengono riportati sulla viewport ed è quest'ultima ad essere posizionata appropriatamente all'interno della finestra principale. La finestra di clipping contiene, cioè, *cosa* vogliamo vedere della scena, mentre la viewport indica *dove* lo vogliamo vedere sullo schermo. Nella finestra principale è possibile visualizzare più di una viewport, così da rappresentarvi più di una scena.

''' 3.5 Colori '''

Come ultima questione, prima di passare ad un vero esempio pratico, discutiamo il sistema di colorazione.

A differenza delle OpenGL®, OpenGL® ES non prevede il paradigma begin/end. La versione classica, infatti, si basava sul definire una forma e le sue proprietà, come il colore, in un blocco di codice racchiuso tra le funzioni **glBegin** e **glEnd**. Il codice sarebbe apparso, cioè, in questo modo:

	Esempio
	<pre> glBegin(GL_TRIANGLES); // Definizione del metodo di unione dei vertici definiti glColor3f(1.0, 0.0, 0.0); // Impostazione del colore glVertex3f(0.25, 0.65, 0.0); // Definizione dei vertici da unire glVertex3f(0.35, 0.35, 0.0); // Segue la definizione dei vari vertici [...] glEnd(); </pre>

Nella versione ES, il paradigma begin/end ha lasciato il passo al più pratico ed ottimale utilizzo degli **array**. Come vedremo, infatti, impostare il colore significa modificare l'**array dei colori** (color array), come definire i vertici di una forma significa modificare l'**array dei vertici** (vertex array) e così via.

Il sistema di rappresentazione del colore utilizzato in OpenGL® ES, come nella quasi totalità dei programmi, è il formato **RGBA**. Ogni colore viene definito tramite quattro valori numerici, variabili in un intervallo finito. Nel nostro caso l'intervallo è tra 0 e 1. I primi tre valori rappresentano le concentrazioni dei colori fondamentali cioè, rispettivamente, di rosso (Red), verde (Green) e blu (Blue), mentre il quarto rappresenta la trasparenza (Alpha). Di qui l'acronimo RGBA. Più un valore di concentrazione è alto, cioè prossimo al limite superiore dell'intervallo, più alta è la concentrazione della rispettiva componente di colore. Ad esempio, essendo il nero il risultato della concentrazione nulla dei colori primari, i valori di RGBA in quel caso sarebbero tutti nulli. Analogamente, per il caso del bianco sarebbero tutti pari a 1.

NB: Molti programmi di manipolazione grafica utilizzano il sistema RGBA, esternandone i valori numerici, come accade nel caso di Adobe Photoshop o Macromedia Flash, ad esempio. Un consiglio pratico per ottenere precisamente il colore desiderato è quello di utilizzare questi programmi per sceglierlo, grazie alle avanzate tavolozze grafiche messe da essi a disposizione e poi ricavare i valori delle concentrazioni RGBA. Solitamente detti programmi utilizzano una scala da 0 a 255 invece che da 0 a 1, quindi andrà effettuata l'opportuna proporzione.

In OpenGL® classico, in realtà, esistevano due formati di rappresentazione del colore, di cui uno era l'RGBA e l'altro la **mappa indicizzata dei colori** (index-color). Detto sistema si basava sul concetto di tavola di colori. Una tavola di colori è costituita da un array contenente valori decimali o esadecimali che, convertiti in forma binaria e divisi in tre sottostringhe di ugual dimensione, rappresentano il livello d'intensità di ognuno dei tre emettitori di elettroni che si trovano in ogni monitor RGB. Ogni emettitore è in pratica il controllore di uno dei tre colori fondamentali: la concentrazione di uno dei tre detti colori è proporzionale al livello di intensità dell'emettitore stesso. Ogni pixel (punto sullo schermo), quindi, assume un colore a seconda dell'intensità con cui viene colpito. Solitamente una tavola di colori contiene 256 valori a 24 bit: ogni blocco di 8 bit rappresenta uno dei valori d'intensità. Ciò significa poter lavorare con una tavolozza di circa 17 milioni di colori (cioè 2^{24} , il numero di combinazioni possibili in un sistema binario disponendo di 24 cifre). Detto sistema **non è supportato** in OpenGL® ES, poiché ritenuto ridonante ed obsoleto e, quindi, non eleggibile a introduzione nel sistema ES, che ha come primaria necessità la minimalità e ottimizzazione.

E' possibile tramite le funzioni di OpenGL® ES, definire non solo il colore, ma dare un effetto di **sfumatura** (shading), ottenibile tramite l'uso della funzione **glShadeModel**. Essa fa uso di 2 parametri, in forma di flag, che specificano il tipo di sfumatura che si desidera usare. Si tratta di GL_FLAT e GL_SMOOTH. Per default, viene utilizzato GL_FLAT. GL_FLAT fa sì che il colore della forma sia unico, in particolare l'ultimo specificato. GL_SMOOTH abilita lo "smooth shading", cioè il colore di ogni vertice e quello della primitiva vengono dati dall'interpolazione dei valori del colore di tutti i vertici della primitiva.

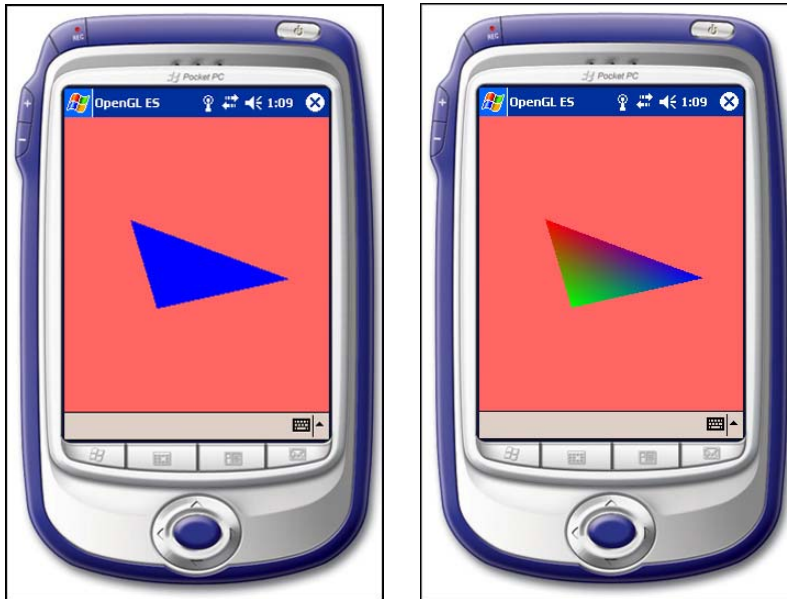


Fig 3.1 Visualizzazione del colore, rispettivamente, in `GL_FLAT` e `GL_SMOOTH`.

☛ 3.6 Effetti di colore: Blending ☛

Un effetto interessante quanto facile da implementare è quello del blending. Utilizzando il sistema di colore RGBA, è possibile ricavare il valore finale del colore di un pixel come la "somma" tra quello attuale (in letteratura chiamato destinazione) ed un eventuale valore entrante (chiamato sorgente) che gli si voglia sovrapporre. Il blending è di default disabilitato e va quindi abilitato.

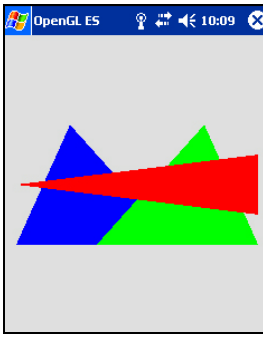
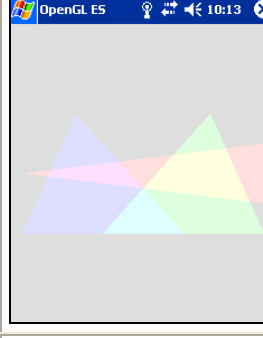
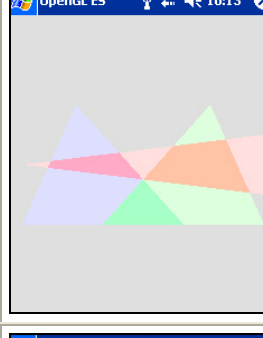
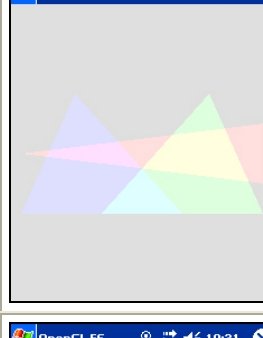
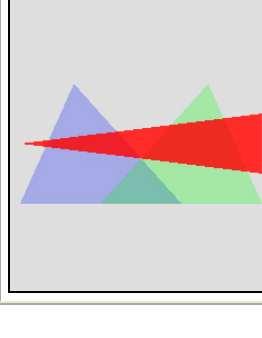
	Esempio
	<code>glEnable(GL_BLEND);</code>

La funzione utilizzata per creare questo effetto è la **glBlendFunc**, tramite la quale è possibile specificare come mescolare i colori. I metodi previsti vengono espressi in forma di 11 flag che possono essere specificati nei due parametri della funzione.

- `GLenum sfactor`: Specifica come i valori di rosso, verde, blu ed alpha del colore sorgente vengono calcolati. Accetta come valori nove flag: `GL_ZERO`, `GL_ONE`, `GL_DST_COLOR`, `GL_ONE_MINUS_DST_COLOR`, `GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`, `GL_DST_ALPHA`, `GL_ONE_MINUS_DST_ALPHA`, e `GL_SRC_ALPHA_SATURATE`.
- `GLenum dfactor`: Specifica come i valori di rosso, verde, blu ed alpha del colore destinazione vengono calcolati. Accetta come valori otto flag: `GL_ZERO`, `GL_ONE`, `GL_SRC_COLOR`, `GL_ONE_MINUS_SRC_COLOR`, `GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`, `GL_DST_ALPHA`, e `GL_ONE_MINUS_DST_ALPHA`.

	Esempio
	<code>glBlendFunc (GL_ONE, GL_ZERO);</code>

Il modo migliore e più semplice di mostrare l'utilizzo pratico di questo effetto, tecnicamente tutt'altro che semplice, è mostrarne degli esempi. Vediamone i più tipici, lasciando al lettore il piacere di sperimentare tutte le possibili combinazioni.

Parametri	Descrizione	Immagine
(GL_ONE, GL_ZERO)	Metodo di default. Fa sì che il colore destinazione non venga usato. Semplicemente il valore di colore finale è quello del colore sorgente.	
(GL_ONE, GL_ONE)	In questo metodo il colore destinazione ed il sorgente vengono "sommati". L'effetto di questa operazione è simile a quello di un mescolamento reale di colori: un misto di rosso e verde forma il giallo e così via.	
(GL_ONE, GL_ONE_MINUS_DST_ALPHA)	Incorpora il valore di alpha del colore destinazione. Ciò crea un lieve miglioramento della trasparenza, ma il rosso si mescola al blu rimanendo ancora rosso.	
(GL_SRC_ALPHA, GL_ONE)	Crea una forma ancora migliore di trasparenza. Le forme ora appaiono tutte trasparenti.	
(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)	In questo caso, le forme meno trasparenti sono più visibili. La forma rossa è più opaca perché ha un valore di alpha più alto rispetto alle altre, pari a 0.8. Il blu, invece, ad esempio, è più trasparente poiché ha un valore di alpha di 0.25.	

Per completezza e onde fornire mezzi di approfondimento, riporteremo un'analisi più tecnica dell'effetto del blending.

Ognuno dei flag appena visti specifica 4 valori, rispettivamente, relativi ai quattro componenti RGBA: rosso, verde, blu ed alpha. Nel seguito, ci riferiamo ai componenti del colore sorgente e destinazione come (Rs , Gs , Bs , As) e (Rd , Gd , Bd , Ad). Si presuppone che questi valori siano interi e compresi nell'intervallo tra zero e (kR , kG , kB , kA) con

$$k = 2^m - 1$$

e (mR , mG , mB , mA) il numero dei bitplane rosso, verde, blu ed alpha.

Siano inoltre (sR , sG , sB , sA) e (dR , dG , dB , dA) i fattori di blending per sorgente e destinazione rispettivamente, ed (fR , fG , fB , fA) gli stessi valori, per cui non sia specificato se si parli della sorgente o della destinazione. Questi valori sono compresi nell'intervallo [0,1].

Per determinare il valore RGBA del pixel attuale, in applicazione del blending, il sistema usa le seguenti equazioni:

$$R_d = \min (kR , R_s sR + R_d dR)$$

$$G_d = \min (kG , G_s sG + G_d dG)$$

$$B_d = \min (kB , B_s sB + B_d dB)$$

$$A_d = \min (kA , A_s sA + A_d dA)$$

Sebbene queste equazioni sembrano così precise, l'aritmetica del blending, come ammesso dalla stessa documentazione ufficiale, non è specificata in maniera chiara poiché, la documentazione asserisce, esso "lavora con valori di colore imprecisi". Tuttavia, è garantito che un fattore di blending pari a 1 non modifica il suo moltiplicando, mentre un valore di 0 riduce il moltiplicando a zero. In questo modo, ad esempio, se *sfactor* è pari a

GL_SRC_ALPHA e *dfactor* è pari a **GL_ONE_MINUS_SRC_ALPHA**, nonchè As è pari a kA, le equazioni suddette si riducono a:

$$R_d = R_s$$

$$G_d = G_s$$

$$B_d = B_s$$

$$A_d = A_s$$

Flag	Fattori di blending (fR , fG , fB , fA)
GL_ZERO	(0, 0, 0, 0)
GL_ONE	(1, 1, 1, 1)
GL_SRC_COLOR	(Rs / kR , Gs / kG , Bs / kB , As / kA)
GL_ONE_MINUS_SRC_COLOR	(1, 1, 1, 1) - (Rs / kR , Gs / kG , Bs / kB , As / kA)
GL_DST_COLOR	(Rd / kR , Gd / kG , Bd / kB , Ad / kA)
GL_ONE_MINUS_DST_COLOR	(1, 1, 1, 1) - (Rd / kR , Gd / kG , Bd / kB , Ad / kA)
GL_SRC_ALPHA	(As / kA , As / kA , As / kA , As / kA)
GL_ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (As / kA , As / kA , As / kA , As / kA)
GL_DST_ALPHA	Ad / kA , Ad / kA , Ad / kA , Ad / kA)
GL_ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (Ad / kA , Ad / kA , Ad / kA , Ad / kA)
GL_SRC_ALPHA_SATURATE	(i, i, i, 1)

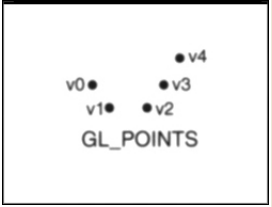
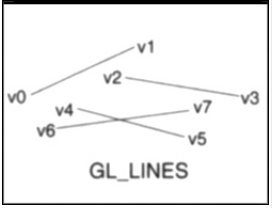
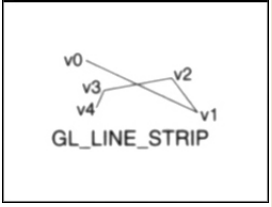
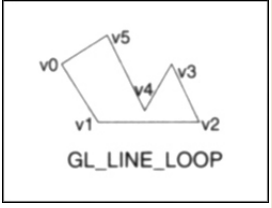
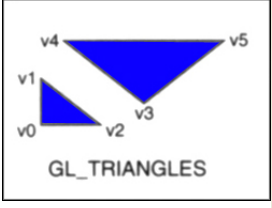
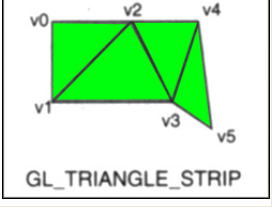
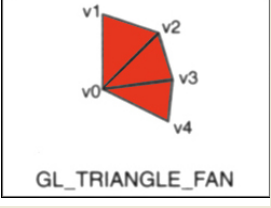
$$* i = \min (A_s , kA - A_d) / kA$$

3.7 Un esempio di codice: visualizzazione 2D

Da premettere al disegno 2D in OpenGL® ES, è il fatto che esso **non** prevede la gestione propria del caso 2D. Trattandosi di un sistema pensato principalmente per la gestione del caso 3D, il 2D viene gestito come un caso particolare, cioè quello in cui i valori sull'asse delle z (cioè la profondità) sono sempre pari a zero. Fatta questa premessa, passiamo a mostrare il codice di qualche semplice esempio di ciò che è stato affrontato nei precedenti paragrafi.

L'esempio riportato in questo paragrafo prevede il disegno di una semplice forma sullo schermo e la sua colorazione. Verrà implementato il sistema del double buffering.

OpenGL® ES prevede il disegno di forme geometriche tramite la definizione e quindi l'unione dei vertici delle stesse. Ciò significa che per disegnare un triangolo ne definiremo i tre vertici e poi il modo in cui andranno uniti. Per ogni metodo d'unione esiste un flag che lo identifica. Vediamone i principali:

Flag identificativo	Descrizione	Immagine
GL_POINTS	Un punto per ogni vertice	
GL_LINES	Viene disegnata una linea per ogni coppia di vertici data	
GL_LINE_STRIP	Dopo il primo vertice viene disegnata una linea che unisce ogni vertice successivo al precedente.	
GL_LINE_LOOP	Come GL_LINE_STRIP, tranne per l'unione tra il primo e l'ultimo vertice della catena.	
GL_TRIANGLES	Per ogni tripletta di vertici viene disegnato un triangolo che li unisce.	
GL_TRIANGLE_STRIP	Dopo i primi 2 vertici, ogni vertice successivo utilizza i due precedenti per disegnare un triangolo.	
GL_TRIANGLE_FAN	Dopo i primi due vertici, ogni vertice successivo usa il precedente ed il primo per disegnare un triangolo. Viene usato per creare forme coniche.	

Diversamente dalle OpenGL® classiche, non sono più supportati QUADS, QUAD_STRIP e POLYGON poiché i risultati da essi provvisti sono comunque ottenibili utilizzando gli altri sistemi supportati. Ad, esempio un quadrato può essere ottenuto attraverso l'unione di due triangoli. L'eliminazione, dalle disponibilità della libreria, di questi flag dipende da scelte progettuali, volte a depennare, quanto più possibile, le ridondanze in un linguaggio che si rivolge a dispositivi embedded, dispositivi dotati, per loro natura, di limitate capacità computazionali e di memorizzazione.

Esistono funzioni come la **glPointSize** e **glLineWidth** che permettono la gestione della dimensione di punti e linee. Di default entrambi i valori di grandezza sono posti a 1.

Passiamo quindi al codice di disegno di un triangolo. Anzitutto, specifichiamo i vertici del triangolo che vogliamo visualizzare in forma di un **array di vertici**. Ogni punto viene rappresentato come una tripletta di valori di tipo float che ne rappresentano, rispettivamente, le coordinate sugli assi cartesiani x,y e z. L'asse x rappresenta la posizione orizzontale, l'asse y quella verticale ed, infine, l'asse z rappresenta la profondità. L'asse z è tale che il suo asse positivo è rivolto verso l'osservatore, quindi più alto è il valore di coordinata z più l'oggetto sarà posto in direzione dell'esterno del dispositivo di visualizzazione. A differenza delle OpenGL® classiche, la versione ES prevede che le coordinate dei vertici possano essere definite solo nei tipi *float*, *short* e *byte*. Per tale motivo, è stato aggiunto il suffisso 'f' ai valori numerici forniti, suffisso che indica la conversione a valori di tipo float.

Utilizzeremo la primitiva GL_TRIANGLE.

Definiamo, inoltre, analogamente, l'**array dei colori**. Faremo in modo che ogni vertice abbia un colore differente, per poter vedere l'effetto creato dall'interpolazione della sfumatura di tipo GL_SMOOTH. Creiamo inoltre una variabile *Sfumatura* che tenga traccia dal fatto che la forma sia sfumata in GL_FLAT o meno.

Codice
<pre>#pragma comment(lib, "libGLES_CM.lib") #pragma comment(lib, "ug.lib") #include "ug.h" GLfloat Triangolo[] = { 0.40f, 0.7f, 0.0f, 0.20f, 0.3f, 0.0f, 0.80f, 0.3f, 0.0f }; GLfloat Colori[] = { 1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f }; bool Sfumatura = false;</pre>

Inizializziamo il colore di sfondo come visto precedentemente.

Codice
<pre>void Inizializza() { glClearColor (0.87f, 0.87f, 0.87f, 0.0f);</pre>

Esistono molti altri array oltre quello del colore e quello dei vertici. Ognuno di essi può essere disabilitato per poter risparmiare risorse per la gestione degli altri. Per questo motivo, di default, sono tutti disabilitati, il che implica che al momento di doverne utilizzare uno, sarà necessario abilitarlo. A questo scopo, esiste la funzione **glEnableClientState** che prende come parametro l'array da abilitare sotto forma di flag. Abilitiamo ora l'array dei vertici e quello dei colori. A **glEnableClientState** si accompagna naturalmente la funzione **glDisableClientState** per la disabilitazione.

Codice
<pre>glEnableClientState(GL_COLOR_ARRAY); glEnableClientState(GL_VERTEX_ARRAY);</pre>

Inizializziamo quindi, tramite la funzione **glColorPointer**, l'array dei colori. Essa prevede 4 parametri:

- **GLint size:** Specifica il numero dei valori con cui è definito il colore per ogni vertice. Nel nostro caso, abbiamo definito ogni colore utilizzando 4 valori. Avremmo potuto anche farlo con 3 valori, ignorando l'Alpha.
- **GLenum type:** Specifica quale tipo di dato viene utilizzato per i valori dell'array. Ad esempio, **GL_BYTE**, **GL_FLOAT**, e così via.
- **GLsizei stride:** Specifica l'offset tra colori consecutivi, cioè quanti valori in più ci sono tra il colore corrente e il prossimo. Potremmo infatti, ad esempio, utilizzare lo stesso array per definire i dati dei vertici e quelli dei colori. Poniamo a 0 il valore poiché questo non è il nostro caso.
- **const GLvoid *pointer:** Specifica la locazione di memoria del primo valore dell'array nella forma del puntatore ad essa.

Codice
<pre>glColorPointer(4, GL_FLOAT, 0, Colori);</pre>

Definiamo quindi la sfumatura. Per ora, impostiamo la **GL_FLAT**.

Codice
<pre>glShadeModel(GL_FLAT);</pre>

Fatto ciò passiamo a stabilire il tipo di proiezione che, per ora, sarà ortografica. Discuteremo meglio il discorso delle proiezioni nel prossimo capitolo, in [4.2](#). OpenGL® ES prevede l'uso di diversi tipi di matrici. Ad esempio:

- **GL_PROJECTION:** viene usata per effettuare le proiezioni
- **GL_MODELVIEW:** viene usata nel ricavo delle coordinate di vista

La funzione **glMatrixMode** si occupa di cambiare la matrice corrente. Eventuali comandi che cambino il contenuto di una matrice, come una moltiplicazione o l'impostazione a matrice identità, agiscono sulla matrice corrente al momento del richiamo del comando. Ciò rientra nella filosofia del sistema a stati di OpenGL® ES. Impostiamola a **GL_PROJECTION**, cioè alla matrice che contiene i valori che definiscono la proiezione, cosicché sia possibile impostarli a proprio piacimento.

Codice

glMatrixMode(GL_PROJECTION);

Inizializziamo, quindi, la matrice all'identità. Detta matrice ha le stesse proprietà dell'uno nella moltiplicazione: moltiplicarla per un'altra matrice restituisce la matrice moltiplicatavi.

Codice

glLoadIdentity();

Fatto ciò, passiamo a definire la proiezione ortografica.

Codice

glOrthof(0.0f, 1.0f, 0.0f, 1.0f, -1.0f, 1.0f);

Definiamo, infine, il viewport con la funzione **glViewport**. Essa accetta 4 parametri:

- GLfloat *x*: Posizione x dell'angolo in basso a sinistra della sottofinestra di viewport
- GLfloat *y*: Posizione y dell'angolo in basso a sinistra della sottofinestra di viewport
- GLfloat *width*: Larghezza della sottofinestra di viewport
- GLfloat *height*: Altezza della sottofinestra di viewport

Nel caso in cui il viewport non venga esplicitamente definito, di default i parametri sono impostati in questa maniera: *x*=0, *y*=0, nonché *width* ed *height* pari alla dimensione della finestra. Per non distorcere l'immagine, è necessario che *width* ed *height* siano nello stesso rapporto della larghezza ed altezza della finestra sul piano.

Codice

glViewport(0, 0, 250, 250);

Impostata la vista, passiamo al disegno del triangolo.

In OpenGL® ES, le primitive vengono visualizzate utilizzando gli **array di vertici**. E' necessario indicare quali vertici utilizzare, il che si ottiene grazie alla funzione

glVertexPointer. Essa prende 4 parametri:

- GLint *size*: Specifica il numero delle coordinate per vertice. Nel nostro caso, abbiamo definito ogni vertice utilizzando 3 valori.
- GLenum *type*: Specifica quale tipo di dato viene utilizzato per i valori dell'array. Ad esempio, GL_BYTE, GL_FLOAT, e così via.
- GLsizei *stride*: Specifica l'offset tra vertici consecutivi cioè quanti valori in più ci sono tra la fine del vertice corrente e l'inizio del prossimo. Poichè non ne sono stati previsti poniamo questo valore a 0.
- const GLvoid **pointer*: Specifica la locazione di memoria del primo valore nell'array, nella forma del puntatore ad essa.

Codice

glVertexPointer(3, GL_FLOAT, 0, Triangolo);
--

Termina qui la preparazione del disegno. Ora la forma è pronta per la visualizzazione. Come in precedenza, inizializziamo ciò che deve essere visualizzato nella finestra al colore di sfondo.

Codice
<pre>void Disegna(UGWindow Finestra) { glClear(GL_COLOR_BUFFER_BIT);</pre>

Per disegnare la primitiva utilizzando l'insieme corrente di array, utilizziamo la funzione **glDrawArrays**. I parametri sono:

- GLenum *mode*: Specifica quali primitiva disegnare. In questo caso GL_TRIANGLES.
- GLint *first*: Specifica l'indice di partenza dell'array, cioè quanti vertici saltare nell'array prima di cominciare a leggerlo. Vogliamo iniziare dal principio, quindi poniamo questo valore a 0.
- GLsizei *count*: Specifica quanti vertici leggere. Per ora ne abbiamo definiti solo 3.

Codice
<pre>glDrawArrays(GL_TRIANGLES, 0, 3);</pre>

Concludiamo la funzione di disegno con il rilascio delle risorse e lo scambio dei buffer (per il double buffering).

Codice
<pre>glFlush(); ugSwapBuffers(Finestra); }</pre>

Per poter apprezzare la differenza tra i due diversi metodi di sfumatura del colore, inseriamo nella funziona di tastiera una parte di codice che farà in modo che su pressione del tasto destro del mouse l'utente possa cambiare il tipo di sfumatura da GL_SMOOTH a GL_FLAT e viceversa.

Codice
<pre>void Tastiera(UGWindow Finestra, int tastoPremuto, int x, int y) { switch(tastoPremuto) { case 'e' : PostQuitMessage(0); break; case 's': Sfumatura = !Sfumatura; glShadeModel(Sfumatura? GL_SMOOTH : GL_FLAT);</pre>

Fatto ciò, sarà necessario aggiornare l'immagine su schermo, mostrando un nuovo frame che rifletta il cambiamento di metodo di sfumatura. Questo è compito della funzione **ugPostRedisplay**, la quale prende come parametro il gestore della finestra che vogliamo aggiornare.

Codice
<pre> ugPostRedisplay(Finestra); break; } } </pre>

In ultimo, diamo il codice per la funzione di aggiornamento, che, tra l'altro, modifica lo spazio di disegno nel caso in cui venga modificata la dimensione della finestra, come visto in [2.6](#).

Codice
<pre> void Aggiorna(UGWindow Finestra, int Larghezza, int Altezza){ glMatrixMode(GL_PROJECTION); glLoadIdentity(); glViewport(0, 0, Larghezza, Altezza); glOrthof(0.0f, 1.0f, 0.0f, 1.0f, -1.0f, 1.0f); glMatrixMode(GL_MODELVIEW); glLoadIdentity(); } </pre>

Infine, definiamo, come al solito la funzione **main**.

Codice
<pre> int main() { UGCtx ug = ugInit(); UGWindow Finestra = ugCreateWindow(ug, "", "OpenGL ES", 250, 250, 100, 100); Inizializza(); ugDisplayFunc(Finestra, Disegna); ugKeyboardFunc(Finestra, Tastiera); ugReshapeFunc(Finestra, Aggiorna); ugMainLoop(ug); return 0; } </pre>

Il risultato finale è il disegno di un semplice triangolo a sfondo rosso, con la possibilità di visualizzarlo in GL_FLAT oppure GL_SMOOTH.

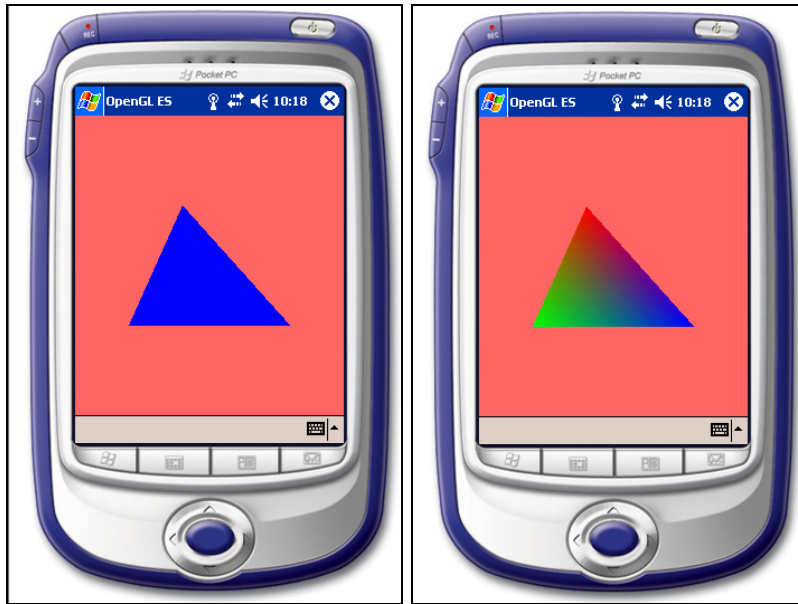


Fig 3.2 Risultato finale, rispettivamente, in *GL_FLAT* e *GL_SMOOTH*

Abbiamo con ciò mostrato come può essere creata e visualizzata una forma in due dimensioni. Passiamo, quindi, al caso 3D.

☛☛ Capitolo 4: Disegnare in tre dimensioni ☛☛

☛☛ 4.1 Introduzione ☛☛

Finalmente, possiamo passare al caso 3D. Vedremo il concetto di profondità e di proiezione, con tutte le sue implicazioni, quali, ad esempio, la necessità di nascondere le facce posteriori di un poligono. Ci occuperemo poi di capire come applicare al nostro solido trasformazioni geometriche come rotazioni, traslazioni, etc.. In seguito, vedremo come animarlo e come aggiungere illuminazione alla nostra scena.

☛☛ 4.2 Proiezioni ☛☛

Prima di passare al disegno, come fatto nel caso bidimensionale, occupiamoci di discutere le problematiche proprie del caso 3D.

Come già osservato, esistono insiemi di routine il cui scopo è permettere la visualizzazione di una scena in tre dimensioni su di uno schermo piatto ovvero in due dimensioni. Introduciamo, quindi, il concetto di proiezione.

Definiamo **proiezione** il processo necessario alla visualizzazione 2D di una scena 3D.

Chiamiamo **piano di vista** il piano su cui vogliamo visualizzarla. Possiamo pensare ad esso come alla superficie di una foto su cui viene rappresentata l'immagine della scena, una volta fotografata. Nel caso della visualizzazione su schermo, tramite OpenGL® ES, possiamo scegliere diverse tecniche di proiezione.

Un modo per ottenere la descrizione di un oggetto solido su un piano di vista è quello di proiettare i punti della superficie dell'oggetto su linee parallele. Questo metodo viene detto **proiezione parallela** od **ortografica**.

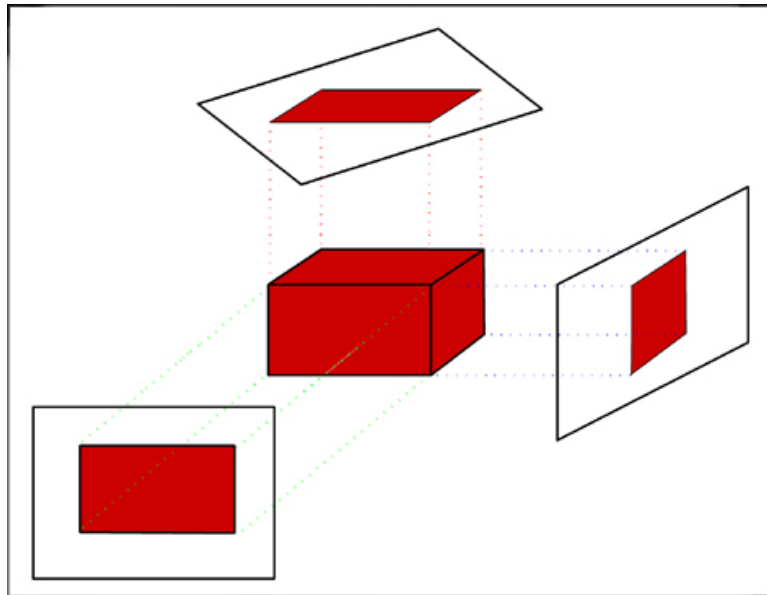


Fig 4.1 Proiezione parallela frontale, laterale e superiore di un parallelepipedo

La proiezione parallela può essere realizzata grazie alla funzione **glOrthof**, che richiede 6 parametri, che identificano il "volume" di vista:

- GLfloat *left* & GLfloat *right*: Specifica i valori per i piani di taglio (clipping) di destra e sinistra
- GLfloat *bottom* & GLfloat *top*: Specifica i valori per i piani di taglio superiore e inferiore
- GLfloat *near* & GLfloat *far*: Specifica la profondità di disegno attraverso i piani di taglio anteriore, cioè il piano di vista, e posteriore. Ogni oggetto al di fuori di essa non verrà disegnato.

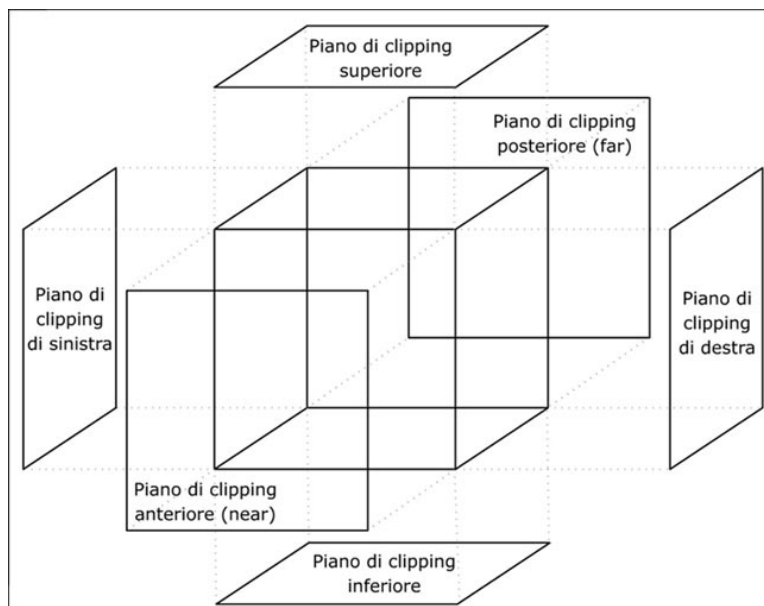


Fig 4.2: In pratica, i sei parametri definiscono una "scatola" entro cui viene visualizzato l'oggetto. Tutto ciò che fuoriesce dalla scatola non viene visualizzato.

I parametri appena discussi definiscono, in pratica, la parte di scena che verrà visualizzata sulla nostra finestra.

Un'altro metodo è quello di proiettare i punti sul piano di vista attraverso cammini convergenti. Questo processo, detto **proiezione prospettica**, fa sì che oggetti più lontani dal piano di vista (cioè dall'osservatore) rispetto ad altri vengano visualizzati più piccoli rispetto a quest'ultimi.

Una scena visualizzata con questo tipo di proiezione risulta sicuramente più realistica, poiché è questo il modo in cui i nostri occhi percepiscono il mondo.



Fig 4.3 Proiezione prospettica: Le linee parallele (bordi della lama) convergono verso un unico punto (che viene chiamato punto di fuga). La mano che impugna la spada appare più grande della punta della lama poiché più vicina al piano di vista, cioè all'osservatore.

La proiezione prospettica può essere realizzata grazie alla funzione **gluPerspectivef**, che richiede 4 parametri:

- GLfloat *fovy*: Angolo di apertura della piramide di vista
- GLfloat *aspect*: Deformazione larghezza/altezza della base del tronco di piramide (in generale la si fa coincidere con quella della viewport)
- GLfloat *znear* & GLfloat *zfar*: distanza delle due basi del tronco di piramide dalla telecamera

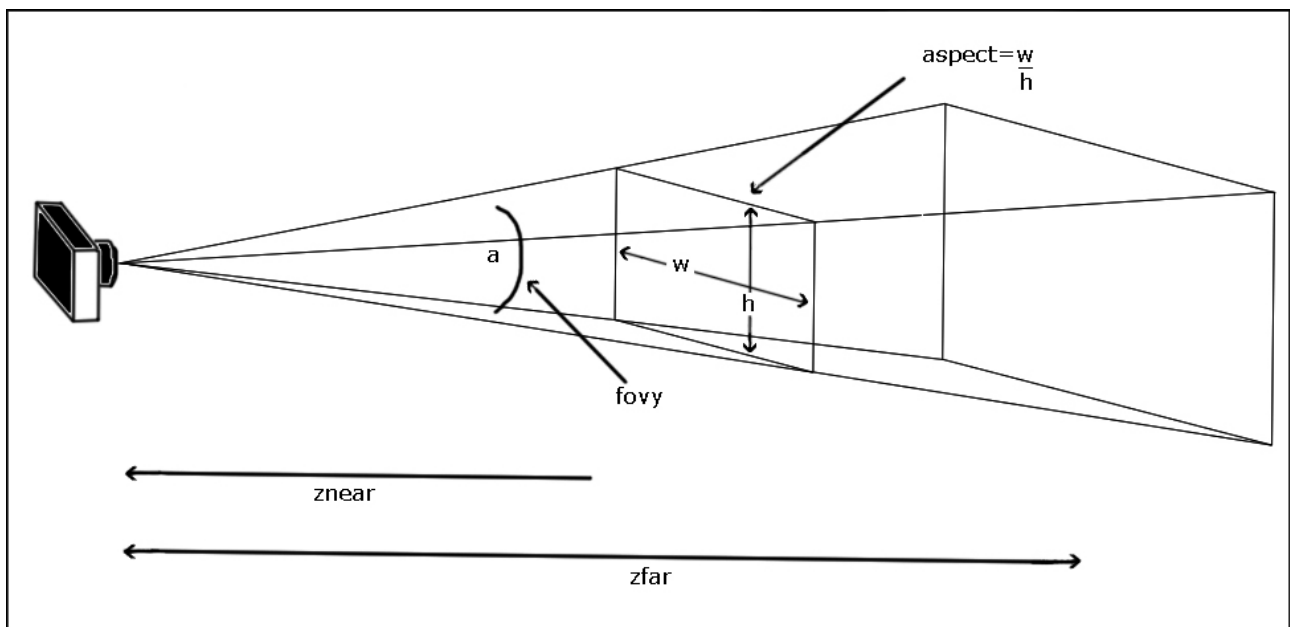


Fig 4.4: Proiezione prospettica: piramide di vista

Come prima, i parametri descritti non fanno altro che definire la “scatola” di taglio, come mostrata per il caso della proiezione ortogonale. In questo caso però, per ovvi motivi, la scatola diventa un tronco di piramide e, di conseguenza, va riveduto il metodo utilizzato per identificarla.

4.3 Profondità

La prima peculiarità di una forma 3D, rispetto al 2D, è la presenza della **profondità**, ovvero di valori diversi da zero sull'asse cartesiano delle z . Tale presenza porta con sé un ovvio problema: due oggetti la cui visualizzazione si sovrapponga nella schermata finale, la quale rappresenta la scena dal punto di vista scelto, devono essere rappresentati in maniera corretta rispetto alla profondità cui si trovano. Più semplicemente, oggetti posti "dietro" altri devono essere nascosti da quelli che vi si trovano davanti. Sebbene nella realtà un tale discorso sia ovvio, in OpenGL® ES diventa più problematico, poiché le forme rappresentate sulla scena, senza adeguato controllo, vengono disegnate l'una sopra all'altra nell'ordine in cui vengono incontrate nel codice, cioè quella rappresentata "davanti" sarà semplicemente l'ultima effettivamente disegnata.

Il problema viene risolto dal cosiddetto **test di profondità** (depth test). Nel caso vi siano punti di forme diverse che si sovrappongono, viene controllato quale di essi si trovi più vicino in profondità e sarà esso l'unico ad essere visualizzato.

Ciò si ottiene tramite l'uso di un buffer detto **buffer della profondità** (depth buffer). Esso contiene un valore per ogni punto sullo schermo, che viene rappresentato normalizzato, come compreso tra 0 e 1, dove lo zero rappresenta la profondità del piano di vista (near clipping plane) e l'uno quella del piano di taglio posteriore (far clipping plane). Tale valore rappresenta, in pratica, la distanza tra gli oggetti visualizzati e l'osservatore. Il test di profondità fa sì che per ogni punto (x,y) della scena, vengano confrontati i valori di profondità dei vari punti degli oggetti presenti che si sovrappongono nel punto (x,y) . Il punto che sarà visualizzato è quello che possiede il più basso valore di profondità, cioè quello più "vicino" all'osservatore. Esplicitiamo l'algoritmo utilizzato, per migliore comprensione ed approfondimento.

Algoritmo per il test di profondità

Siano BuffProf e BuffScena, rispettivamente il buffer della profondità e quello che dovrà contenere il contenuto della scena finale.

- 1) Inizializzare i valori dei buffer per ogni elemento (x,y) :
BuffProf (x,y) = 1.0 //profondità del piano di taglio posteriore
BuffScena (x,y) = colore di sfondo
- 2) Analizzare ogni poligono sulla scena, uno per volta.
 - Per ogni posizione (x,y) e quindi per il corrispondente pixel del poligono in analisi, calcolare il valore di profondità z .
 - Se $z < \text{BuffProf}(x,y)$, calcolare il valore di colore del pixel in quella posizione e impostare:
BuffProf = z
BuffScena = colore (x,y)

Quando tutte le superfici siano state analizzate BuffProf conterrà i valori di profondità delle superfici visibili e BuffScena conterrà i corrispondenti valori di colore per dette superfici.

Vediamo ora le funzioni che realizzano quanto descritto.

Come tutti i buffer, il buffer di profondità andrà abilitato, tramite la funzione **glEnable**. In ogni momento se ne può decidere la disabilitazione, tramite l'analoga funzione **glDisable**. Di default, il test è disabilitato.

In realtà, andrebbe anche passato il valore GL_TRUE alla funzione **glDepthMask**, per far sì che il buffer possa essere aggiornato per la scrittura, ma detto valore viene impostato già per default. Si può comunque proteggerlo da scrittura passando GL_FALSE come parametro alla suddetta funzione.

Questa proprietà è utile, ad esempio, in caso si abbia lo stesso complesso sfondo con le visualizzazioni di diversi oggetti. Lasciato lo sfondo nel buffer della profondità, potremo

disabilitarvi la scrittura e passare al disegno degli oggetti dinnanzi a esso. Ciò ci permette di generare una serie di schermate con diversi oggetti sullo stesso sfondo, o con gli stessi oggetti in diverse posizioni, come avviene per ogni sequenza di animazione.

Un'altro modo di utilizzare questo parametro è quello di sfruttarlo per effetti di trasparenza. Infatti, nel suddetto caso, vorremo salvare nel buffer della profondità solo i valori degli oggetti opachi per il test di visibilità. Quindi dovremmo disabilitare il test di profondità nel caso si disegni una superficie trasparente.

	Esempio
	<code>glEnable(GL_DEPTH_TEST); // abilitazione</code> <code>glDisable(GL_DEPTH_TEST); // disabilitazione</code>

Il test di profondità, come si è potuto vedere, viene realizzato in modo tale che più basso è il valore di profondità più vicina è la coordinata all'osservatore. In realtà questo è solo il caso di default. Possiamo modificare tale visione, utilizzando la funzione **glDepthFunc**, che specifica come debba essere eseguito il test. In alcune applicazioni, sotto determinate condizioni, tale cambiamento porta un buon risparmio di risorse. I metodi disponibili vengono identificati dai flag sottostanti.

Flag	Descrizione
GL_NEVER	Il valore non passa mai il test
GL_LESS	Il valore passa il test se il valore di profondità in arrivo è minore del valore memorizzato
GL_EQUAL	Il valore passa il test se il valore di profondità in arrivo è uguale al valore memorizzato
GL_LEQUAL	Il valore passa il test se il valore di profondità in arrivo è minore o uguale al valore memorizzato
GL_GREATER	Il valore passa il test se il valore di profondità in arrivo è maggiore del valore memorizzato
GL_NOTEQUAL	Il valore passa il test se il valore di profondità in arrivo è diverso dal valore memorizzato
GL_GEQUAL	Il valore passa il test se il valore di profondità in arrivo è maggiore o uguale del valore memorizzato
GL_ALWAYS	Il valore passa sempre il test

Affinchè il test di profondità sia abilitato al confronto dei valori, andrà inizializzato il buffer di profondità utilizzando la funzione **glClearDepthf**, con parametro il valore con cui inizializzare tutti i valori del buffer. Se passiamo il valore 1 tutte le primitive appariranno a schermo. Nel caso si scelgano i confronti con GL_GREATER o GL_GEQUAL, il valore va impostato a 0.

	Esempio
	<code>glClearDepthf(1.0f);</code>

E' necessario, quindi, richiamare la funzione **glClear**, onde inizializzare il buffer della profondità al valore passato alla **glClearDepthf**. Per farlo è necessario passare come parametro il valore GL_DEPTH_BUFFER_BIT. E' possibile anche passare più di un flag, utilizzando il separatore '|', come possiamo vedere nell'esempio di codice sottostante.

	Esempio
--	----------------

```
glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
```

Avevamo osservato in precedenza come il parametro *config* della funzione **ugCreateWindow** potesse essere utilizzato per abilitare determinati buffer. In caso si utilizzi questa funzione per visualizzare la finestra, perché il buffer della profondità possa funzionare correttamente è necessario che tale parametro sia pari al valore `UG_DEPTH`.

Esempio

```
UGWindow Finestra = ugCreateWindow(ug, "UG_DEPTH", "OpenGL ES", 250, 250, 100, 100);
```

4.4 Occultamento delle facce nascoste

Un'altra problematica portata dalla profondità è quella delle facce nascoste.

Un cubo è composto da 6 facce, 6 superfici delimitate da segmenti di linee di lunghezza finita. Delle 6 facce, a seconda delle impostazioni del punto di vista, solo alcune saranno realmente visibili. Nonostante ciò, esse vengono trattate, computazionalmente parlando, esattamente allo stesso modo delle altre. E' ragionevole, quindi, pensare che sia uno spreco computazionale, utilizzare tempo e calcoli per il disegno e le trasformazioni che riguardano le facce non visibili. Il discorso va esteso a tutti i tipi di poligoni, e viene risolto dalla tecnica di **occultamento delle facce nascoste** (backface culling).

Come per il test di profondità, anche questo metodo è già implementato in OpenGL® ES. Basterà abilitarlo e impostarlo a dovere a seconda delle esigenze. Di default, è disabilitato.

Esempio

```
glEnable(GL_CULL_FACE); //abilitazione  
glDisable(GL_CULL_FACE); //disabilitazione
```

Il riconoscimento di quali siano le facce posteriori viene effettuato in maniera molto semplice e intuitiva. La proiezione di un poligono su schermo ha **orientamento orario** (clockwise winding) se un immaginario oggetto, che percorra il sentiero che parte dal suo primo vertice e prosegua sul secondo, sul terzo e così via fino all'ultimo e torni al primo, si muova in direzione oraria rispetto all'interno del poligono stesso. Analogo è il caso dell'orientamento antiorario. La funzione **glFrontFace** specifica quali tra i poligoni con orientamento orario e antiorario, siano considerati "anteriori" rispetto a quelli con orientamento opposto. Il parametro `GL_CCW` (counter-clockwise winding) imposta come anteriori i poligoni con orientamento antiorario, mentre `GL_CW` (clockwise winding) opera il contrario. La scelta di default è `GL_CCW`.

Esempio

```
glFrontFace (GL_CCW);
```

Un'altra funzione utile è **glCullFace**, la quale specifica quali facce debbano essere "culled", cioè non visualizzate. Tra i valori che gli possono essere passati come parametri ci sono: `GL_FRONT`, `GL_BACK` e `GL_FRONT_AND_BACK`, i quali, rispettivamente, indicano le facce anteriori, le facce posteriori, ed entrambe quest'ultime. Intuitivamente, di default, si ha `GL_BACK`.

	Esempio
	<code>glCullFace (GL_BACK);</code>

4.5 Trasformazioni geometriche: rotazione, traslazione, ridimensionamento

Nel caso 3D, come anche nel 2D, oltre a disegnare nuove forme, potrà essere necessario applicare ad esse delle trasformazioni geometriche. I tre tipi di base di trasformazione sono:

- Rotazione: ruota la forma attorno ai tre assi cartesiani
- Traslazione: sposta la forma su uno dei tre assi a scelta
- Ridimensionamento: ridimensiona la forma modificandone la scala

I creatori di OpenGL® hanno previsto una gestione delle matrici per le operazioni di traslazione, rotazione e ridimensionamento, molto semplice. Questa si aggiunge alle molte ragioni per cui OpenGL® è tanto popolare. Prima di vedere le trasformazioni in sè, è bene fare una breve premessa a proposito dei meccanismi matematici su cui si muove il discorso. Riporteremo una breve, esplicativa e simpatica introduzione scritta del professor Uche Akotaobi per la University of Southern California.

" Cos'è una matrice?

E' semplicemente un metodo per descrivere un insieme di trasformazioni n-dimensionalì in un unico conveniente pacchetto.

Ciò è. E' tutto ciò che sono, e tutto ciò per cui vengono utilizzate.

Ma, mio dio, una volta odiavo quelle cose! Ogni volta che le vedevo su un sito web, tornavo subito indietro, come se quel sito contenesse qualcosa di terrificante. Tempo fa giurai che non le avrei mai utilizzate nei miei motori grafici, solo per dimostrare, dispettosamente, che era possibile vivere senza di loro!

Bene, certamente, questo è possibile. Non c'è nulla che tu possa fare con una matrice di trasformazione che tu non possa fare algebricamente, e ci sono cose, al contrario, che puoi fare algebricamente ma non tramite le matrici (la distorsione di corpi non rigidi mi viene in mente...ma non è questo il punto). Naturalmente, ciò non significa che le matrici siano malvagie -- al tempo ero giovane e tignoso.. Il punto è che, di fatto, queste cose rendono la vita più semplice al programmatore 3D.

In ogni caso, questa è una matrice 4x4

$$\begin{pmatrix} A1 & A2 & A3 & A4 \\ B1 & B2 & B3 & B4 \\ C1 & C2 & C3 & C4 \\ D1 & D2 & D3 & D4 \end{pmatrix}$$

Una matrice è un array bidimensionale di dati, dove ogni riga o colonna consiste di uno o più valori numerici. Le operazioni aritmetiche che possono essere eseguite con le matrici includono l'addizione, la sottrazione, la moltiplicazione e la divisione. La dimensione di una matrice è definita in termini di numero di righe e colonne. Una matrice con M righe ed N colonne si dice matrice MxN e può essere rappresentata con un array di dimensioni MxN. Possiamo utilizzare il sistema numerico per trasformare oggetti geometrici 3D traslandoli, ruotandoli e ridimensionandoli. "

Ognuna di queste trasformazioni si ottiene lavorando sulla matrice GL_MODELVIEW, che andrà, quindi, anzitutto selezionata come matrice corrente.

Le trasformazioni vengono operate tramite le tre funzioni **glRotatef**, **glTranslatef** e **glScalef**, dove la *f* in suffisso indica che il tipo dei loro parametri dovrà essere *float*. Potrebbe essere sostituita con una *x* nel caso si voglia utilizzare variabili di tipo *GLfloat*. Molte funzioni in OpenGL® funzionano in questa maniera. A volte, è permesso anche inserire una *v* alla fine per

indicare che verrà utilizzato un array come parametro. Maggiori dettagli su questo discorso possono essere trovati nell'[Appendice C.2](#).

Nell'operare una trasformazione, è importante racchiudere il codice nelle funzioni **glPushMatrix** e **glPopMatrix**, perchè altre forme disegnate in seguito non vengano influenzate dalla trasformazione corrente. Infatti, le trasformazioni lavorano sulla matrice GL_MODELVIEW, che contiene le coordinate dell'intero sistema. Le glPushMatrix e glPopMatrix garantiscono che a fine trasformazione venga ripristinato il sistema di coordinate non ruotato, traslato o ridimensionato.

Come si può intuire, ogni trasformazione operata nel blocco glPushMatrix-glPopMatrix influenza la matrice GL_MODELVIEW, e quindi le coordinate dell'intero sistema. Ciò vuol dire che è possibile applicare più di un tipo di trasformazione contemporaneamente. E' importante perciò tenere presente che l'ordine con cui le trasformazioni vengono applicate, l'ordine, cioè, con cui sono chiamate nel codice, non è affatto indifferente.

Ruotare, traslare o ridimensionare una forma, significa operare una moltiplicazione tra la matrice che la rappresenta in forma di coordinate numeriche e una matrice particolare, della stessa dimensione, che contiene valori differenti a seconda del tipo di trasformazione desiderata. Il fatto che l'ordine non sia indifferente, quindi, deriva dal fatto che la moltiplicazione tra matrici non è indifferente rispetto all'ordine dei moltiplicandi.

Ad esempio, traslando la forma verso sinistra e poi ruotandola, la vedremo ruotata sull'asse orizzontale, mentre, ruotando e poi traslando, vedremo la forma ruotata muoversi diagonalmente rispetto all'origine.

Analizziamo, quindi, le funzioni che operano le trasformazioni.

La funzione **glTranslatef** (/glTranslatex) accetta come parametri:

- GLfloat (/GLfixed) x: Specifica la coordinata sull'asse delle x del vettore di traslazione
- GLfloat (/GLfixed) y: Specifica la coordinata sull'asse delle y del vettore di traslazione
- GLfloat (/GLfixed) z: Specifica la coordinata sull'asse delle z del vettore di traslazione

Per vettore di traslazione si intende un vettore la cui direzione, verso e modulo specificano direzione, verso e modulo della traslazione. Tale vettore viene espresso in forma di una coordinata tridimensionale che identifica un punto nello spazio cartesiano. Sarà possibile ricostruire il vettore unendo l'origine del sistema con il punto in questione.

NB: Un **vettore** è uno strumento spesso utilizzato in matematica. Nel sistema cartesiano esso viene rappresentato come una freccia. Si tratta di un segmento caratterizzato da: **direzione**, cioè la retta su cui giace il segmento, **verso**, cioè uno dei due possibili versi su detta retta, **modulo** o intensità, cioè la lunghezza del segmento, ed, infine, **punto di applicazione**, cioè il punto di inizio, quello che precede tutti gli altri punti del segmento.

Esempio

```
glTranslatef(0.25f, 0.5f, 0.0f);
```

La glTranslate produce una traslazione di (x, y, z). La matrice corrente viene moltiplicata per la matrice di traslazione, e il prodotto ottenuto la rimpiazza, come se avessimo richiamato la funzione glMultMatrix con argomento la matrice di traslazione:

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Se la matrice corrente è la GL MODELVIEW o la GL PROJECTION, tutti gli oggetti disegnati dopo la chiamata a `glTranslate` vengono traslati. Di qui la necessità dell'uso di `glPushMatrix` e `glPopMatrix` per salvare e ripristinare le coordinate prive della traslazione.

La funzione **glScalef** (/ **glScalex**) accetta come parametri:

- GLfloat (/GLfixed) *x*: Specifica il fattore di ridimensionamento sull'asse delle *x*
- GLfloat (/GLfixed) *y*: Specifica il fattore di ridimensionamento sull'asse delle *y*
- GLfloat (/GLfixed) *z*: Specifica il fattore di ridimensionamento sull'asse delle *z*

	Esempio
	glScalef (0.5f, 0.5f, 0.5f);

La `glScalef` produce un ridimensionamento di (*x*, *y*, *z*). La matrice corrente viene moltiplicata per la matrice di ridimensionamento, e il prodotto ottenuto la rimpiazza, come se avessimo richiamato la funzione `glMultMatrix` con argomento la matrice di ridimensionamento:

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Se la matrice corrente è la GL MODELVIEW o la GL PROJECTION, tutti gli oggetti disegnati dopo la chiamata a `glScalef` vengono ridimensionati. Di qui la necessità dell'uso di `glPushMatrix` e `glPopMatrix` per salvare e ripristinare le coordinate prive del ridimensionamento. La documentazione ufficiale segnala un piccolo bug del sistema OpenGL®: se i fattori di ridimensionamento sono diversi da uno e l'illuminazione è attiva a volte si potrebbero avere dei problemi nella sua visualizzazione. In tal caso, basti abilitare la normalizzazione automatica delle normali chiamando la `glEnable` con argomento `GL_NORMALIZE`.

La funzione **glRotatef** (/ **glRotatex**) accetta come parametri:

- GLfloat (/GLfixed) *angle*: Specifica l'angolo di rotazione, in forma di gradi
- GLfloat (/GLfixed) *x*: Coordinata sull'asse delle *x* di un vettore
- GLfloat (/GLfixed) *y*: Coordinata sull'asse delle *y* di un vettore
- GLfloat (/GLfixed) *z*: Coordinata sull'asse delle *z* di un vettore

	Esempio
	glRotatef (xrot, 1.0f, 0.0f, 0.0f);

La funzione `glRotatef` produce una rotazione di gradi pari al parametro *angle* attorno al vettore (*x*, *y*, *z*). La matrice corrente viene moltiplicata per la matrice di rotazione, e il prodotto ottenuto la rimpiazza, come se avessimo richiamato la funzione `glMultMatrix` con argomento la matrice di rotazione:

$$\begin{pmatrix} x^2(1-c) + c & xy(1-c) - zs & xz(1-c) + ys & 0 \\ xy(1-c) + zs & y^2(1-c) + c & yz(1-c) - xs & 0 \\ xz(1-c) - ys & yz(1-c) + xs & z^2(1-c) + c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

dove $c = \cos(\text{angle})$, $s = \sin(\text{angle})$, e $|(x, y, z)| = 1$, (altrimenti GL provvede a normalizzare questo vettore)

Un valore di uno su di uno degli ultimi tre parametri e di 0 sugli altri due è normalmente usato per specificare un asse come vettore attorno cui effettuare la rotazione. Se la matrice è la GL_MODELVIEW o la GL_PROJECTION, tutti gli oggetti disegnati dopo la chiamata di `glRotate` ne vengono influenzati.

La rotazione segue la regola della mano destra, cioè se il vettore (x, y, z) punta verso l'utente la rotazione sarà in senso antiorario.

Si può pensare ad esempio alla rotazione intorno all'asse delle x come all'inclinarsi su e giù di una macchina su una collina, guardandola da davanti. Mentre alla rotazione sull'asse delle y come alla rotazione di una trivella, mentre state trapanando un pezzo di legno. La rotazione sulle z è come la rotazione di un ventilatore guardando dal davanti le pale.

Un'ultima precisazione: immaginiamo abbiate notato come le matrici abbiamo quattro e non soltanto 3 righe. Questo deriva dal fatto che si immaginano le coordinate dei punti date in forma di **coordinate omogenee**. Per completezza chiariamo anche questa nozione.

In molti concetti geometrici è utile ed esemplificativo poter utilizzare il concetto di infinito. Si pensi al disegno di una curva che va al limite all'infinito, ad esempio. Nelle coordinate euclidee classiche questo concetto non è, però, contemplato. A questo scopo esistono le coordinate omogenee. Consideriamo due numeri a e w e calcoliamo il valore del rapporto a/w . Se a rimane fissato e w varia, ci possiamo accorgere di come tanto più w sia piccolo tanto più a/w diventi grande. Al tendere di w a zero, a rigor di logica, il rapporto dovrà tendere all'infinito. Quindi, per catturare il concetto di infinito utilizzeremo per ogni valore di coordinata un rapporto a/w , cioè rimpiazzeremo ognuna delle coordinate (x, y, z) di un punto nello spazio con dei valori $x/w, y/w$ e z/w . Un punto in coordinate omogenee può essere rappresentato tramite non più tre, quindi, bensì quattro valori distinti (x, y, z, w) . Ovvio è che ad ogni punto (x, y, z) corrisponde un punto in coordinate omogenee $(x, y, z, 1)$. Di qui il motivo della quarta riga delle nostre matrici. Come vedremo in seguito, ci sono sistemi in OpenGL® ES che utilizzano appieno le proprietà delle coordinate omogenee. Ad esempio, si utilizzano per definire la proprietà di una luce di proiettare all'infinito.

4.6 Animazione

Come abbiamo già visto, visualizzare un'animazione vuol dire semplicemente mostrare velocemente in sequenza una serie di immagini minimamente differenti tra loro. Vediamo come realizzare una semplice animazione in OpenGL® ES.

Immaginiamo di voler animare un semplice cubo, facendolo ruotare su se stesso. Per permettere l'esecuzione di questo tipo di animazione avremo bisogno di una funzione detta *Idle*, che venga richiamata dal main loop quando non vi siano altri comandi da eseguire, allo scopo di far continuare all'infinito la rotazione.

La funzione **Idle** accetta come parametro il gestore della finestra. Per far ruotare il cubo, la funzione incrementerà l'angolo della rotazione. Dopo la modifica di detti valori, l'immagine a schermo dovrà essere aggiornata, il che verrà eseguito tramite la funzione

ugPostRedisplay.

Esempio

```
void Idle(UGWindow Finestra)
```

```
{
    xrot += 2.0f;
    yrot += 3.0f;

    ugPostRedisplay(Finestra);
}
```

L'ultimo passo da fare, come di consueto, è notificare al motore UG quale funzione Idle intendiamo utilizzare, il che viene realizzato dalla funzione **ugIdleFunc**. Essa prende due parametri: il primo è il gestore del motore UG ed il secondo la funzione Idle. Come per le `ugKeyboardFunc` e `ugMouseFunc`, poniamo questa funzione nel main, poco prima della funzione `ugMainLoop`.

Esempio

```
ugIdleFunc(ug, Idle);
```

L'esempio completo verrà fornito in [4.8](#).

4.7 Illuminazione e proprietà di materiale

Una parte importante, nel disegno di una scena, è recitata dall'illuminazione. Vediamo, quindi, come OpenGL® ES permette la gestione delle fonti di luce.

Anzitutto, vediamo cosa accade al disegno di ogni singolo punto quando l'illuminazione è abilitata o disabilitata.

A **luci disabilitate**, ogni vertice, di ognuno dei solidi disegnati nella scena, ha un proprio colore (assegnatogli tramite la **glColor4f**), ed ogni punto, facente parte di ogni solido, ha colore ottenuto interpolando quello dei vertici. Ne segue che una primitiva ha un unico colore uniforme se i suoi vertici hanno tutti lo stesso colore. Quindi, nel caso 3D, se tutte le facce di un solido hanno stesso colore, il solido appare come un'unica macchia di colore senza effetto di tridimensionalità.

A **luce abilitata**, ogni punto viene colorato con un colore che dipende dalla luce che lo colpisce e dalle proprietà di quest'ultima, nonché dalla capacità di risposta alla luce della primitiva di cui fa parte il punto, cioè le proprietà del materiale di cui idealmente è composta; ed con un'intensità che dipende dai due insiemi di proprietà appena citati, ed eventualmente dall'inclinazione con cui la luce lo colpisce, cioè l'angolo tra la direzione della luce e la direzione della normale alla primitiva in quel punto. Una normale è un vettore perpendicolare ad una superficie, cioè con angolazione di 90 gradi. Il sistema non calcola le normali automaticamente: è quindi necessario specificare una normale per ogni poligono che venga disegnato per far sì che venga influenzato dalla luce. Ad esempio, nel caso della luce diffusa, l'intensità dell'illuminazione dipende dall'angolo formato dalla direzione della luce con la normale alla superficie.

OpenGL® ES prevede, come per OpenGL® classico, quattro tipi di illuminazione :

- **Ambientale**: La luce non proviene da una direzione ben definita. Quando la luce raggiunge una superficie essa viene riflessa in tutte le direzioni. Illumina uniformemente un oggetto.
- **Diffusa**: La luce proviene da una direzione ben precisa. Per il resto è simile a quella ambientale, nel senso che viene riflessa in tutte le direzioni.
- **Speculare**: La luce è direzionale ma viene riflessa verso una particolare direzione. Come avviene per un riflesso su di una superficie reale, riflesso che è conosciuto come riflesso speculare.
- **Emissiva**: La luce proviene da un oggetto specifico. Esso emette una certa quantità di luce, ma non riflette tutte le superfici.

Sono previste, solitamente, fino a 8 fonti di luce, identificabili ognuna con il flag, rispettivamente, di GL_LIGHT0 ... GL_LIGHT7.

Come già detto, le luci hanno varie proprietà che possono essere impostate. Una delle più importanti è il modo in cui una sorgente proietta la luce a seconda della sua posizione ed orientamento. Essa, infatti, può proiettare luce:

- **All'infinito:** La sorgente è posizionata in un determinato punto ed è composta di raggi luminosi paralleli che illuminano l'intera scena.
- **Da un punto verso l'intera scena:** La sorgente è posizionata in un punto ed è composta di raggi che dal punto si irradiano in tutte le direzioni.
- **Da un punto verso una determinata area (spot light):** La sorgente è posizionata in un punto ben preciso ed è composta di raggi che, a partire da questo punto, si irradiano in un cono con un determinato angolo di apertura.

Come per le luci, anche le proprietà del materiale sono molte ed impostabili. L'idealizzazione del dotare una primitiva di un materiale, deriva dal fatto che diversi materiali riflettono in maniera differente la luce. Il concetto di base è che una pallina di vetro, rifletta la luce in maniera diversa da una di legno. Fatte queste premesse, possiamo passare a vedere un pò di codice.

Di nuovo, adottare il sistema di illuminazione necessita di abilitazione. La funzione necessaria è la **glEnable** con argomento GL_LIGHTING. E' necessario abilitare anche un'eventuale fonte di luce, con i flag GL_LIGHT0.. GL_LIGHT7.

	Esempio
	<pre>glEnable(GL_LIGHTING); // Abilitazione glEnable(GL_LIGHT0); glDisable(GL_LIGHTING); // Disabilitazione glDisable(GL_LIGHT0);</pre>

Quando l'illuminazione viene attivata il modello di colorazione deve essere GL_SMOOTH, quindi è necessario il comando **glShadeModel** con argomento GL_SMOOTH.

E', inoltre, necessario definire le normali. Ciò si ottiene tramite la funzione **glNormal3f/glNormal3x** che accetta come parametro:

- GLfloat (/GLfixed) *nx*: Coordinata sull'asse delle x della nuova normale
- GLfloat (/GLfixed) *ny*: Coordinata sull'asse delle y della nuova normale
- GLfloat (/GLfixed) *nz*: Coordinata sull'asse delle z della nuova normale

Il valore di default è (0,0,1).

In seguito, è possibile specificare con i comandi **glLightf/glLightx** alcuni parametro della luce. La funzione accetta come parametri:

- GLenum *light*: Rappresenta la fonte di luce, con i flag GL_LIGHT0.. GL_LIGHT7. In diverse implementazioni è possibile siano previste più di otto fonti. Il valore può essere ricavato dal flag GL_MAX_LIGHT.
- GLenum *pname*: Specifica un parametro, in un unico valore, della fonte di luce. Sono accettati come parametri dieci flag di cui spiegheremo la natura a breve.
- GLfloat (/GLfixed) *param*: Specifica il valore/i a cui dovrà essere impostato il parametro identificato dal flag dichiarato nel parametro *pname*.

La funzione è disponibile anche nelle versioni **glLightfv/glLightxv** la cui unica differenza è nel parametro:

- const GLfloat * (/const GLfloat *) *params*: Specifica un puntatore al valore/i a cui dovrà essere impostato il parametro identificato dal flag dichiarato nel parametro *pname*.

Analizziamo ora i dieci possibili parametri *pname*:

- **GL_AMBIENT:** *Params* prevede 4 valori fixed oppure float che specificano l'intensità RGBA della luce ambientale cioè, quindi, il suo colore. L'intensità di default è (0, 0, 0, 1).
- **GL_DIFFUSE:** Come per GL_AMBIENT, ma nel caso della luce diffusa. Il valore di default per GL_LIGHT0 è pari a (1, 1, 1, 1). Per le altre è (0, 0, 0, 0).
- **GL_SPECULAR:** Come per GL_DIFFUSE, ma nel caso della luce speculare.
- **GL_POSITION:** *Params* prevede 4 valori fixed oppure float che specificano la posizione della luce in coordinate omogenee. La posizione viene trasformata dalla matrice GL_MODELVIEW quando glLight viene richiamata, e viene poi memorizzata nelle coordinate di vista. Se la quarta componente della posizione è 0, la luce viene trattata come una fonte direzionale. Le illuminazioni diffusa e speculare considerano la direzione della luce, ma non la sua posizione, e l'attenuazione è disabilitata. Altrimenti, esse sono basate sulla collocazione attuale della luce in coordinate di vista e l'attenuazione è abilitata. La posizione di default è (0,0,1,0). In questo modo, la luce di default viene considerata direzionale, parallela e in direzione dell'asse z.
- **GL_SPOT_DIRECTION:** *Params* prevede 3 valori fixed oppure float, che specificano la direzione della luce (all'infinito) in coordinate omogenee. La direzione di spot viene trasformata dall'inversa della matrice GL_MODELVIEW quando glLight viene richiamata (come avveniva nel caso delle normali), e viene poi memorizzata nelle coordinate di vista. Il valore è significativo solo quando GL_SPOT_CUTOFF è diverso da 180. Il valore di default è (0, 0, -1).
- **GL_SPOT_EXPONENT:** *Params* prevede 1 valore fixed oppure float, che specifica l'intensità della distribuzione della luce. Il valore è compreso nell'intervallo [0, 128]. L'effettiva intensità della luce si attenua con il coseno dell'angolo tra la direzione della luce e la retta tra la direzione della luce e il vertice illuminato, elevato all'esponente di spot. In tal modo, più grande è l'esponente di spot più la fonte di luce risulta concentrata, indipendentemente dall'angolo di spot cutoff. Il valore di default è 0, il che porta a una distribuzione uniforme di luce.
- **GL_SPOT_CUTOFF:** *Params* prevede 1 valore fixed oppure float, che specifica il massimo angolo del cono di luce. Sono accettati i valori nell'intervallo tra [0, 90] più il valore speciale 180. Se l'angolo tra la direzione della luce e la retta tra la direzione della luce e il vertice illuminato è più grande dell'angolo di spot cutoff allora la luce è completamente mascherata. Altrimenti, la sua intensità è controllata dall'esponente di spot e il fattore di attenuazione. Il valore di default è 180, che porta a una distribuzione uniforme di luce.
- **GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION:** *Params* prevede 1 valore fixed oppure float, che specifica uno dei tre fattori di attenuazione della luce. Sono accettati solo valori non negativi. Se la luce è posizionale, invece che direzionale, la sua intensità si attenua con la somma reciproca del fattore costante, del fattore lineare moltiplicato per la distanza tra la luce e il vertice illuminato, e il fattore quadratico moltiplicato per la radice della stessa distanza. Il valore di default è (1,0,0) cioè assenza di attenuazione.

Come si può notare i primi tre parametri (GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR) definiscono le caratteristiche della luce (ambientale, diffusa, speculare), mentre le seguenti definiscono come essa proietta la luce rispetto alla sua posizione, orientamento, intensità, etc..

Vediamo quindi come definire le proprietà di **materiale**. La funzione addetta allo scopo è la **glMaterialf/glMaterialx**. Essa accetta i seguenti parametri:

- **GLenum *face*:** Specifica quale faccia/e devono essere toccate dalla modifica del parametro *pname*. Di default è GL_FRONT_AND_BACK
- **GLenum *pname*:** Specifica il parametro di materiale che si voglia impostare. Sono accettati come parametri sei flag di cui spiegheremo la natura a breve.
- **GLfloat (/GLfixed) **param*:** Specifica il valore/i a cui dovrà essere impostato il parametro identificato dal flag dichiarato nel parametro *pname*.

La funzione è disponibile anche nelle versioni **glMaterialfv/glMaterialxv** le cui differenze sono nei parametri:

- `GLenum pname`: Specifica il parametro di materiale che si voglia impostare. Sono accettati come parametri sei flag di cui spiegheremo la natura a breve.
- `const GLfloat * (/const GLfloat *) params`: Specifica un puntatore al valore/i a cui dovrà essere impostato il parametro identificato dal flag dichiarato nel parametro `pname`.

Analizziamo ora i possibili sei parametri `pname`:

- **GL_AMBIENT**: *Params* prevede 4 valori fixed oppure float che specificano la riflessione RGBA del materiale, nel caso della luce ambientale. Il valore di default è (0.2, 0.2, 0.2, 1.0).
- **GL_DIFFUSE**: *Params* prevede 4 valori fixed oppure float che specificano la riflessione RGBA del materiale, nel caso della luce diffusa. Il valore di default è (0.8, 0.8, 0.8, 1.0).
- **GL_SPECULAR**: *Params* prevede 4 valori fixed oppure float che specificano la riflessione speculare RGBA. Il valore di default è (0, 0, 0, 1).
- **GL_EMISSION**: *Params* prevede 4 valori fixed oppure float che specificano la riflessione RGBA dell'intensità della luce emessa del materiale. Il valore di default è (0, 0, 0, 1).
- **GL_SHININESS**: *Params* prevede 4 valori fixed oppure float che specificano l'esponente speculare RGBA del materiale. Sono accettati solo valori compresi nell'intervallo [0, 128]. Il valore di default è 0.
- **GL_AMBIENT_AND_DIFFUSE**: Equivale a chiamare la `glMaterial` due volte con argomento `pname` la prima volta `GL_AMBIENT` e la successiva `GL_DIFFUSE`.

4.8 Un esempio di codice: visualizzazione 3D

Vediamo un esempio che riassume quanto visto finora.

In esso disegneremo un cubo, visualizzato in proiezione prospettica. Attiveremo il test di profondità, nasconderemo le facce posteriori e lo animeremo, facendolo ruotare su se stesso. Doteremo la scena di illuminazione ambientale e direzionale, nonché imposteremo le proprietà di materiale per il solido creato. Provvederemo, inoltre, il programma di una serie di funzioni di tastiera che permettano facilmente la modifica delle opzioni più topiche: la presenza o meno dell'illuminazione e delle luci, il modello di sfumatura flat o smooth, e persino un piccolo sistema di interazione che permetta di fermare la rotazione automatica e far sì che l'utente possa ruotare il cubo sui tre assi manualmente.

Come al solito, carichiamo le librerie necessarie

Codice
<pre>#pragma comment(lib, "libGLES_CM.lib") #pragma comment(lib, "ug.lib") #include "ug.h"</pre>

Impostiamo delle variabili che mantengano i valori di rotazione sui vari assi del nostro poligono. Come esempio di trasformazione geometrica, infatti, eseguiremo delle rotazioni.

Codice
<pre>float Rotazione_AsseX = 0.0f; float Rotazione_AsseY = 0.0f; float Rotazione_AsseZ = 0.0f;</pre>

Utilizzeremo la variabile *autoRotazione* per fornire all'utente la scelta di far ruotare automaticamente il poligono o ruotarlo manualmente. La variabile assumerà il valore 1 se la rotazione è automatica o 0 se è manuale. Vedremo meglio come ottenere ciò nella funzione di gestione della tastiera.

Codice
int autoRotazione=1;

Come per *autoRotazione*, anche la variabile *Sfumatura* verrà utilizzata per fornire una scelta all'utente: quella di utilizzare il modello di sfumatura *flat* o lo *smooth*, di modo da permettere di notare facilmente le differenze tra i due sistemi, all'atto dell'aggiunta dell'illuminazione.

Codice
bool Sfumatura = false;

Costruire un poligono significa fornire al sistema di disegno le coordinate di ogni suo vertice, ed il modo con il quale detti vertici vadano uniti. Costruiamo un cubo per ora. L'introduzione della variabile *val* ha l'unico scopo di rendere più semplice il cambiamento dei valori dei vari vertici, che, in questo caso, assumono valori che differiscono solo per il segno. Il cubo verrà disegnato tracciando per ogni faccia due triangoli che, uniti, formino un quadrato.

Ricordiamo che questo modo di tracciare i quadrati fa parte di una scelta progettuale di OpenGL® ES, volta a eliminare il più possibile le ridondanze in un linguaggio che si propone come rivolto a dispositivi embedded, che, per loro natura, dispongono di limitate capacità di computazione e di memorizzazione.

Codice
<pre>float val=0.5f; GLfloat Cubo[] = { // Faccia frontale -val, -val, val, val, -val, val, -val, val, val, val, val, val, // Faccia di retro -val, -val, -val, -val, val, -val, val, -val, -val, val, val, -val, // Faccia laterale sinistra -val, -val, val, -val, val, val, -val, -val, -val, -val, val, -val, // Faccia laterale destra val, -val, -val, val, val, -val, val, -val, val, val, val, val, // Base (alto) -val, val, val, val, val, val, -val, val, -val, val, val, -val,</pre>

```
// Base (basso)
-val, -val, val,
-val, -val, -val,
val, -val, val,
val, -val, -val,
};
```

Creiamo due luci d'esempio, la prima ambientale, la seconda direzionale. L'uso degli array permette una facile modifica dei vari parametri.

Renderemo possibile all'utente, in seguito, su pressione del tasto 0 od 1 della tastiera di abilitare o disabilitare le luci, rispettivamente 0 od 1. Su pressione del tasto 2 renderemo possibile anche l'abilitazione/disabilitazione dell'illuminazione. Nelle immagini mostriamo gli effetti ottenibili.



Fig 4.5 Effetti dell'illuminazione. Partendo dall'alto a sinistra abbiamo il caso dell'illuminazione disabilitata, e a susseguire i vari casi possibili con l'illuminazione abilitata, cioè con luci spente, con luce direzionale ed ambientale accese, con luce ambientale accesa e direzionale spenta, con luce direzionale accesa ed ambientale spenta.

Codice	
GLfloat AmbientLuce0[]	= { 0.9f, 0.9f, 0.9f, 1.0f };
GLfloat DiffuseLuce0[]	= { 0.0f, 0.0f, 0.0f, 1.0f };
GLfloat SpecularLuce0[]	= { 1.0f, 1.0f, 1.0f, 0.0f };
GLfloat PositionLuce1[]	= { 200.0f, 0.0f, 0.0f, 1.0f};
GLfloat DirectionLuce1[]	= { -200.0f, 0.0f, 0.0f, 1.0f };
GLfloat CutoffLuce1[]	= { 10.0f};
GLfloat AmbientLuce1[]	= { 0.1f, 0.1f, 0.1f, 1.0f };
GLfloat DiffuseLuce1[]	= { 0.1f, 0.1f, 0.1f, 1.0f };
GLfloat SpecularLuce1[]	= { 0.1f, 0.1f, 0.1f, 1.0f };

Definiamo quindi il materiale del poligono, come spiegato nel precedente paragrafo.

Codice	
GLfloat AmbientMat[]	= { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat DiffuseMat[]	= { 1.0f, 0.0f, 0.0f, 1.0f };
GLfloat SpecularMat[]	= { 1.0f, 0.0f, 0.0f, 1.0f };
GLfloat ShineMat[]	= { 40.0f };

Occupiamoci ora delle necessarie impostazioni iniziali, come sempre, all'interno della funzione **Inizializza**.

Codice	
	void Inizializza () {

Definiamo il colore di sfondo e il modello di sfumatura.

Codice	
	glEnableClientState (GL_COLOR_ARRAY);
	glClearDepthf (1.0f);
	glClearColor (0.87f, 0.87f, 0.87f, 0.0f);
	glShadeModel (GL_SMOOTH);

Abilitiamo l'illuminazione e le due luci, impostando per ognuna di esse i vari parametri. Facciamo lo stesso per quanto riguarda il materiale.

Codice	
	glEnable (GL_LIGHTING);
	glEnable (GL_LIGHT0);
	glLightfv (GL_LIGHT0, GL_AMBIENT, AmbientLuce0);
	glLightfv (GL_LIGHT0, GL_DIFFUSE, DiffuseLuce0);
	glLightfv (GL_LIGHT0, GL_SPECULAR, SpecularLuce0);
	glEnable (GL_LIGHT1);


```

glLightfv(GL_LIGHT1, GL_POSITION, PositionLuce1);
glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, DirectionLuce1);
glLightfv(GL_LIGHT1, GL_SPOT_CUTOFF, CutoffLuce1);
glLightfv(GL_LIGHT1, GL_AMBIENT, AmbientLuce1);
glLightfv(GL_LIGHT1, GL_DIFFUSE, DiffuseLuce1);
glLightfv(GL_LIGHT1, GL_SPECULAR, SpecularLuce1);

glEnable(GL_COLOR_MATERIAL);
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, AmbientMat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, DiffuseMat);
glMaterialfv(GL_FRONT, GL_SPECULAR, SpecularMat);
glMaterialfv(GL_FRONT, GL_SHININESS, ShineMat);

```

Abilitiamo il test di profondità, come spiegato in [4.2](#).

Codice

```

glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LEQUAL);

```

Abilitiamo il vertex array e quindi facciamo puntare all'array che contiene i vertici dei triangoli che andranno a formare il nostro cubo.

Codice

```

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, Cubo);

```

Nascondiamo le facce nascoste, come visto in [4.3](#).

Codice

```

glEnable(GL_CULL_FACE);
glFrontFace (GL_CCW);
glCullFace (GL_BACK);
}

```

Passiamo ora al disegno vero e proprio, impostando la prospettiva. Per muovere la posizione della telecamera (cioè il nostro punto di vista) dobbiamo modificare la *projection matrix*. Le cose si complicano. Per renderci più semplice la vita, utilizziamo la funzione **gluLookAtf**. Essa accetta 9 parametri, in forma di 3 coordinate o vettori:

- GLfloat (*eyex, eyey, eyez*): Posizione della telecamera nella scena
- GLfloat (*centerx, centery, centerz*): Punto verso cui la telecamera è puntata
- GLfloat (*upx, upy, upz*): Vettore normale, che di solito è posto a (0,1,0)

Il codice sottostante pone la telecamera lontana dall'origine e che guarda verso di essa.

Codice

```

void Disegna(UGWindow Finestra)

```

```
{
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glLoadIdentity();

    gluLookAtf(
        0.0f, 0.0f, 3.0f,
        0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f);
}
```

Passiamo ora a ruotare la nostra forma, definendone i valori di rotazione sui vari assi.

Codice

```
glRotatef(Rotazione_AsseX, 1.0f, 0.0f, 0.0f);
glRotatef(Rotazione_AsseY, 0.0f, 1.0f, 0.0f);
glRotatef(Rotazione_AsseZ, 0.0f, 0.0f, 1.0f);
```

Disegniamo quindi il cubo, colorandone ogni faccia grazie alla funzione **glColor4f**. Precedentemente abbiamo accennato alle normali. Esse devono essere perpendicolari alle superfici. Quindi, ad esempio, la superficie frontale avrà un vettore normale (0,0,1), mentre quella di retro (0,0,-1). La lunghezza dei vettori è pari a 1. Le normali vengono specificate chiamando la funzione **glNormal3f** prima di disegnare la primitiva. Questa funzione prende come parametri 3 valori float che identificano il vettore della normale.

Codice

```
// Faccia frontale e retro
glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
glNormal3f(0.0f, 0.0f, 1.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
glNormal3f(0.0f, 0.0f, -1.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 4, 4);

// Faccia laterale sinistra e destra
glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
glNormal3f(-1.0f, 0.0f, 0.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 8, 4);

glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
glNormal3f(1.0f, 0.0f, 0.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 12, 4);

// Basi (alto e basso)
glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
glNormal3f(0.0f, 1.0f, 0.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 16, 4);

glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
glNormal3f(0.0f, -1.0f, 0.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 20, 4);
```

Chiudiamo la funzione come al solito.

Codice
<pre>glFlush(); ugSwapBuffers(Finestra); }</pre>

La funzione di aggiornamento cambia, rispetto a quanto visto in precedenza, solo per l'utilizzo della proiezione prospettica invece che dell'ortogonale. Utilizziamo la funzione **ugluPerspectivef**, come visto in [4.2](#).

Codice
<pre>void Aggiorna(UGWindow Finestra, int Larghezza, int Altezza){ glMatrixMode(GL_PROJECTION); glLoadIdentity(); glViewport(0, 0, Larghezza, Altezza); ugluPerspectivef(45.0f, 1.0f * Larghezza / Altezza, 1.0f, 100.0f); glMatrixMode(GL_MODELVIEW); glLoadIdentity(); }</pre>

Finalmente, la nostra prima animazione! Impostiamo la funzione **Idle**, che, ricordiamo, viene richiamata dal main loop quando non vi siano altri comandi da eseguire, per far ruotare, di un tot per ogni riaggiornamento, il cubo sui vari assi. Verrà visualizzata così una sequenza di immagini in cui il nostro cubo sarà ruotato sempre di più, cioè verrà visualizzata l'animazione della rotazione.

In seguito, chiamiamo il ridisegno.

Codice
<pre>void Idle(UGWindow Finestra) { if(autoRotazione==1){ Rotazione_AsseX += 1.0f; Rotazione_AsseY += 3.0f; Rotazione_AsseZ += 1.0f; } ugPostRedisplay(Finestra); }</pre>

Come ultima cosa, vediamo come dare all'utente qualche possibilità di interazione. Il seguente codice predispone varie funzioni alla pressione di determinati tasti della tastiera durante l'esecuzione. Vediamole.

Tasto premuto	Effetto
e	Uscita dal programma
0/1	Disabilitazione su pressione ed abilitazione su successiva

	pressione delle luci, rispettivamente, 0 ed 1
2	Disabilitazione dell'illuminazione
a	Disabilitazione su pressione ed abilitazione su successiva pressione della rotazione automatica
x/y/z	Rotazione manuale del poligono sugli assi, rispettivamente, x/y/z di 2 gradi per ogni pressione
s	Passaggio da modello di sfumatura smooth a flat su pressione e viceversa alla successiva

Codice

```
void Tastiera(UGWindow Finestra, int tastoPremuto, int x, int y)
{
    switch(tastoPremuto)
    {
        case 'e' :
            PostQuitMessage(0);
            break;

        case '0' :
            if (glIsEnabled(GL_LIGHT0)){
                glDisable(GL_LIGHT0);
            } else {
                glEnable(GL_LIGHT0);
            }
            break;

        case '1' :
            if (glIsEnabled(GL_LIGHT1)){
                glDisable(GL_LIGHT1);
            } else { if (!glIsEnabled(GL_LIGHT1))
                glEnable(GL_LIGHT1);
            }
            break;

        case '2' :
            if (glIsEnabled(GL_LIGHTING)){
                glDisable(GL_LIGHTING);
                glDisable(GL_LIGHT0);
            } else {
                glEnable(GL_LIGHTING);
                glEnable(GL_LIGHT0);
            }
            break;

        case 'x' :
            Rotazione_AsseX += 2.0;
            break;

        case 'y' :
            Rotazione_AsseY += 2.0;
            break;

        case 'z' :
            Rotazione_AsseZ += 2.0;
            break;
    }
}
```

```

    case 'a' :
        if (autoRotazione==0){
            autoRotazione=1;
        } else {
            autoRotazione=0;
        }
    break;

    case 's':
        Sfumatura = !Sfumatura;
        glShadeModel(Sfumatura? GL_SMOOTH : GL_FLAT);
        ugPostRedisplay(Finestra);
    break;
}
}

```

Il main rimane il solito, con l'aggiunta dell'impostazione della funzione **Idle**.

Codice

```

int main()
{
    UGctx ug = ugInit();
    UGWindow Finestra = ugCreateWindow(ug, "UG_DEPTH", "OpenGL ES", 250,
    250, 100, 100);

    Inizializza();

    ugDisplayFunc(Finestra, Disegna);
    ugKeyboardFunc(Finestra, Tastiera);
    ugReshapeFunc(Finestra, Aggiorna);
    ugIdleFunc(ug, Idle);

    ugMainLoop(ug);
    return 0;
}

```

Il codice è impostato in maniera tale da rendere semplice la modifica. Si invita lo studente ad eseguirlo più volte, modificando i valori delle proprietà della luce e del materiale, di modo da capire più a fondo le differenze che intercorrono tra di esse.

◀◀ Capitolo 5: Effetti avanzati ▶▶

◀◀ 5.1 Introduzione ▶▶

Una volta digerite le basi, sarà il momento di passare al lavoro vero e proprio. Non più triangoli colorati o cubi rotanti al centro dello schermo, bensì paesaggi sconfinati, spiagge assolate e metropoli frenetiche. Disegnare una "scena" è ben diverso da quanto affrontato finora: implica il dover rendere effetti che abbiano lo scopo di riprodurre, il più fedelmente possibile, la realtà. Inizieremo descrivendo il texture mapping, cioè la tecnica necessaria ad applicare immagini alle nostre forme solide. Esamineremo quindi almeno un paio di effetti che contribuiscono al fotorealismo: la trasparenza e la nebbia.

◀◀ 5.2 Texture Mapping ▶▶

Immaginiamo di aver costruito una forma 3D che rappresenti un cubo. Per quanto un cubo di un unico colore possa essere interessante, all'interno della scena che intendiamo costruire non vorremo un semplice cubo, bensì una scatola, un dado o un mattone. Il problema che si pone, quindi, è come disegnare sulle facce del nostro oggetto per dargli le caratteristiche di un oggetto reale. Un modo, tra i più semplici, di raggiungere questo obiettivo è applicare alla superficie della forma una **texture** (Texture Mapping).

La "texture" non è altro che un'immagine, che viene sovrapposta alle facce di una forma solida seguendone i contorni e venendo replicata se necessario: ad esempio, immaginiamo di voler disegnare una scatola di legno. Una volta disegnata la forma di un cubo, sovrapporremo una texture che rappresenti il lato della scatola sulle facce del cubo.

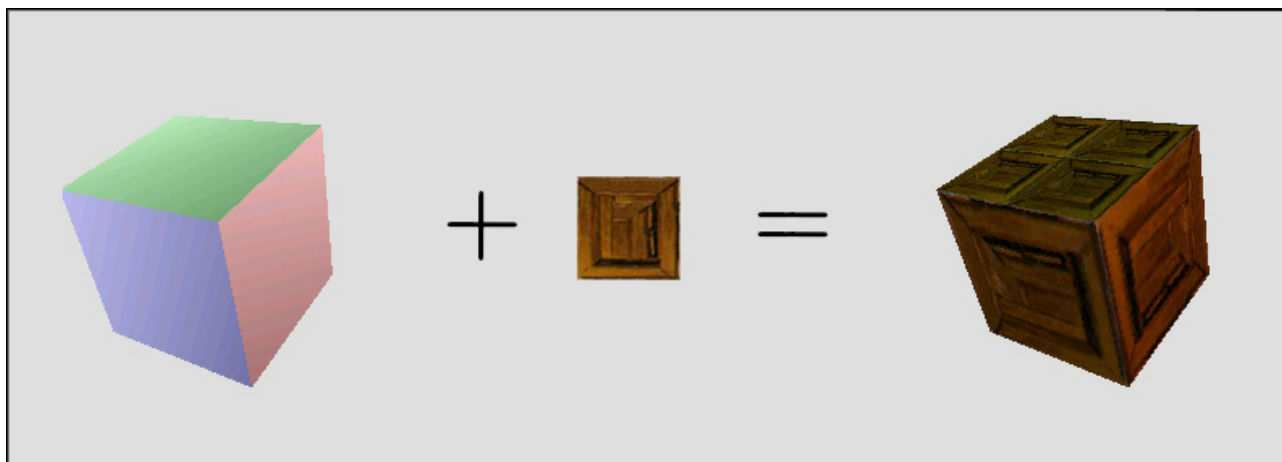


Fig 5.1 Esempio di applicazione di una texture.

Il passo forse più laborioso dell'applicazione di una texture è il caricamento dell'immagine dalla memoria fissa. Ovvio è che, ai fini di operare questo processo, è necessario decidere previamente quale tipo di formato di immagine si desidera utilizzare ed, inoltre, conoscerne la natura nel dettaglio. Infatti, per il caricamento sarà necessario analizzare ogni bit del file immagine e destinarlo al proprio ruolo. Rimandiamo all'[Appendice B](#) per la spiegazione di un esempio di codice di caricamento di un'immagine di tipo Windows Bitmap, di modo da mostrare nello specifico in cosa consiste il processo.

Vediamo come applicare una texture al cubo trattato nel precedente capitolo. Un esempio più completo verrà dato nel codice di fine capitolo.

Ogni texture possiede, anzitutto, uno specifico identificatore, rappresentato da un intero senza segno. Sarà quindi necessario creare un array grande abbastanza da contenere tutti gli eventuali identificatori necessari.

Esempio

```
static GLuint texture[1];
```

Va dunque definito dove apparirà la texture sul poligono in questione. Per fare ciò, utilizzeremo le **coordinate di texture**, riposte, ordinatamente, in un array ad esse dedicato. Per coordinate di texture intendiamo delle coppie di valori variabili, di solito, nell'intervallo $[0,1]$, dove il valore $(0,0)$ rappresenti l'angolo in basso a sinistra della texture e $(1,1)$ quello in alto a destra. Pensando al nostro cubo, guardandolo dal davanti, il primo vertice che si incontra è quello in basso a sinistra e quindi utilizzeremo il valore $(0,0)$ per il primo vertice, e via dicendo...

Esempio

```

GLfloat CoordinateTexture[] = {
    // Faccia frontale
    0.0f, 0.0f,
    2.0f, 0.0f,
    0.0f, 2.0f,
    2.0f, 2.0f,
    // Faccia di retro
    1.0f, 0.0f,
    1.0f, 1.0f,
    0.0f, 0.0f,
    0.0f, 1.0f
    // Faccia laterale sinistra
    1.0f, 0.0f,
    1.0f, 1.0f,
    0.0f, 0.0f,
    0.0f, 1.0f,
    // Faccia laterale destra
    1.0f, 0.0f,
    1.0f, 1.0f,
    0.0f, 0.0f,
    0.0f, 1.0f,
    // Base (alto)
    0.0f, 0.0f,
    1.0f, 0.0f,
    0.0f, 1.0f,
    1.0f, 1.0f,
    // Base (basso)
    1.0f, 0.0f,
    1.0f, 1.0f,
    0.0f, 0.0f,
    0.0f, 1.0f,
};

```

Nella funzione **Inizializza**, abilitiamo il Texture Mapping.

Esempio

```

void Inizializza(){
    [...]
    glEnable(GL_TEXTURE_2D);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    [...]
}

```

Carichiamo quindi l'immagine, richiamando la funzione il cui codice viene descritto in dettaglio nell'[Appendice B](#). Se il caricamento non dovesse avvenire correttamente, restituiamo un messaggio d'errore.

Esempio

```

unsigned char *Img=NULL;
Img=CaricaImmagine("img.bmp");

```

Indichiamo dove recuperare le coordinate di texture.

Esempio
<code>glTexCoordPointer(2, GL_FLOAT, 0, CoordinateTexture);</code>

Generiamo quindi un identificatore per la texture (**Texture Name**) tramite la funzione specifica **glGenTextures**. Essa prende come parametri:

- Il numero di identificatori da generare
- Il puntatore ad un array che li possa contenere

Esempio
<code>glGenTextures(1, texture);</code>

Per impostare la texture corrente, utilizziamo la funzione **glBindTexture**, che prende come parametri **GL_TEXTURE_2D** ed il Texture Name che identifica quella che si desidera selezionare. Ciò comporta che, caricandone più di una, sia possibile cambiare la texture corrente durante l'esecuzione semplicemente richiamando la **glBindTexture** con parametro un diverso Texture Name. Nel codice di fine capitolo mostreremo questa possibilità nel dettaglio.

Esempio
<code>glBindTexture(GL_TEXTURE_2D, texture[0]);</code>

Per impostare altre proprietà della texture, possiamo utilizzare la funzione **glTexParameterf/glTexParameteri**. Questa funzione permette di definire varie proprietà. La vedremo in dettaglio nel prossimo paragrafo.

Esempio
<code>glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);</code>

Altre proprietà sono definibili tramite la funzione **glTexImage2D**. Anche in questo caso, i dettagli sono rimandati al prossimo paragrafo.

Esempio
<code>glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, info_Header.biWidth, info_Header.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, Img);</code>

Pronta la texture, è possibile liberare la memoria allocata alla memorizzazione dell'immagine poichè OpenGL® ES dispone ormai dei suoi dati.

Esempio
<pre>delete[] Img; [...]</pre>



Fig 5.2 Esempi di applicazione di texture.

5.3 Proprietà delle texture e Mipmap

Sono molte le proprietà che possono essere impostate al momento di definire una texture. Tra di esse vi è l'utilizzo delle mipmap. Analizzeremo il tutto, esaminando le due principali funzioni atte a definire dette proprietà: **glTexParameterf** e **glTexImage2D**. Prima di passare a ciò, soffermiamoci sul concetto di mipmap, dato che nel seguito ci troveremo ad utilizzarlo.

Una **mipmap** è un insieme ordinato di array che rappresentano la stessa immagine con dimensioni sempre più piccole. Se la texture ha dimensione $2n \times 2m$ ci saranno $\max(n,m)+1$ mipmap (con $\max(m,n)$ la funzione che restituisce il massimo tra i due parametri m ed n). La prima mipmap è la texture originale con dimensione $2n \times 2m$. Ogni mipmap seguente ha dimensione $2^{k-1} \times 2^{l-1}$ dove $2^k \times 2^l$ sono le dimensioni della mipmap precedente finché $k = 0$ o $l = 0$. Quindi le ultime mipmap avranno dimensione $1 \times 2^{l-1}$ o $2^{k-1} \times 1$ fino all'ultima, che ha dimensione 1×1 .

Vengono definite utilizzando la funzione **glTexImage2D** con l'argomento che indica il livello di dettaglio, impostato all'ordine della mipmap. Il livello 0 è la texture originale, il livello massimo (n,m) è la texture finale 1×1 .

Torniamo ora alla nostra funzione **glTexParameterf/glTexParametersi**. Il suo ruolo è molto simile, per filosofia, alle funzioni di impostazione delle proprietà dell'illuminazione e dei materiali. Ricordiamo che disegnare un pixel significa definirne, in un valore numerico, il colore e, quindi, aggiungere una texture significa dare ai pixel delle nostre forme 3D un colore definito dal disegno della texture. Quando ci si riferisca a "valori di texture" lo si farà tenendo presente questa visione della cosa.

La funzione **glTexParameterf/glTexParametersi** definisce come vada gestita l'applicazione della texture per grandezze diverse da quella originale. Prende tre parametri:

- **GLenum *target***: Specifica la natura della texture, deve essere **GL_TEXTURE_2D**. OpenGL® ES non supporta texture 1D o 3D.
- **GLenum *pname***: Specifica la proprietà della texture che si voglia impostare. Sono accettati come parametri 4 flag di cui spiegheremo la natura a breve.
- **GLfloat (/GLint) *params***: Specifica il valore/i a cui dovrà essere impostato il parametro identificato dal flag dichiarato nel parametro *pname*.

Analizziamo ora i possibili parametri *pname*:

- **GL_TEXTURE_MIN_FILTER**: La funzione di rimpicciolimento di una texture è utilizzata quando un'immagine viene scelta come texture per un'area più piccola della propria dimensione. Ci sono 6 funzioni di questo tipo: due utilizzano il più vicino o i più vicini quattro elementi di texture per calcolare la texture risultante. Le altre utilizzano le mipmap. Vediamo le funzioni possibili, cioè i possibili valore assegnabili al parametro *params*:
 - **GL_NEAREST**: Restituisce il valore dell'elemento di texture più vicino al centro del pixel cui stiamo aggiungendo correntemente una texture.
 - **GL_LINEAR**: Restituisce la media pesata dei quattro elementi di texture più vicini al centro del pixel cui stiamo aggiungendo correntemente una texture. Può includere bordi, a seconda del valore di **GL_TEXTURE_WRAP_S** e **GL_TEXTURE_WRAP_T**.
 - **GL_NEAREST_MIPMAP_NEAREST**: Sceglie la mipmap che si avvicina di più alla dimensione del pixel cui stiamo aggiungendo correntemente una texture ed usa il criterio **GL_NEAREST** per produrre un valore di texture.
 - **GL_LINEAR_MIPMAP_NEAREST**: Sceglie la mipmap che si avvicina di più alla dimensione del pixel cui stiamo aggiungendo correntemente una texture ed usa il criterio **GL_LINEAR** per produrre un valore di texture.
 - **GL_NEAREST_MIPMAP_LINEAR**: Sceglie due mipmap che si avvicinano di più alla dimensione del pixel cui stiamo aggiungendo correntemente una texture ed usa il criterio **GL_NEAREST** per produrre un valore di texture per ognuna delle mipmap. La texture finale è la media pesata di questi due valori.
 - **GL_LINEAR_MIPMAP_LINEAR**: Sceglie due mipmap che si avvicinano di più alla dimensione del pixel cui stiamo aggiungendo correntemente una texture ed usa il criterio **GL_NEAREST** per produrre un valore di texture per ognuna delle mipmap. La texture finale è la media pesata di questi due valori.

Poichè vengono gestiti molti elementi di texture nel processo di rimpicciolimento, le prestazioni potrebbero non sempre essere ottime. Le funzioni **GL_NEAREST** e **GL_LINEAR**, che gestiscono al massimo da uno a quattro texture, risultano più veloci delle altre quattro funzioni, che però presentano prestazioni grafiche migliori per ovvi motivi. E' quindi necessario vagliare, caso per caso, il compromesso più consono alla situazione.

Il valore di default di **GL_TEXTURE_MIN_FILTER** è **GL_NEAREST_MIPMAP_LINEAR**.

- **GL_TEXTURE_MAG_FILTER**: La funzione di ingrandimento di una texture è utilizzata quando un'immagine viene scelta come texture per un'area più grande o uguale alla propria dimensione. Ci sono 6 funzioni di questo tipo: due utilizzano il più vicino o i più vicini quattro elementi di texture per calcolare la texture risultante. Le altre usano le

mipmap. Vediamo le funzioni possibili, cioè i possibili valore assegnabili al parametro *params*:

- **GL_NEAREST**: Restituisce il valore dell'elemento di texture più vicino al centro del pixel cui stiamo aggiungendo correntemente una texture.
- **GL_LINEAR**: Restituisce la media pesata dei quattro elementi di texture più vicini al centro del pixel cui stiamo aggiungendo correntemente una texture. Può includere bordi, a seconda del valore di GL_TEXTURE_WRAP_S e GL_TEXTURE_WRAP_T.

In questo caso, non troviamo l'uso delle mipmap perchè, come spiegato, vi sono mipmap più piccole dell'immagine ma non più grandi. GL_NEAREST è generalmente più veloce di GL_LINEAR, ma può produrre immagini con angoli meno smussati. Il valore di default di GL_TEXTURE_MAG_FILTER è GL_LINEAR.

Ricordiamo che l'applicazione di texture è una tecnica che applica un'immagine alla superficie di un oggetto con coordinate (s,t) definibili dall'utente. Le GL_TEXTURE_WRAP_X (con X=S oppure X=T) definiscono, in generale, come le texture debbano essere drappeggiate sulla superficie degli oggetti.

- **GL_TEXTURE_WRAP_S**: Imposta il drappeggio per la coordinate di texture S a GL_CLAMP o GL_REPEAT.
 - **GL_CLAMP**: Fa sì che la coordinata S possa assumere valori nell'intervallo [0,1] ed è utile per evitare che l'immagine venga ripetuta.
 - **GL_REPEAT**: Fa sì che la parte intera della coordinata S sia ignorata. L'utilizzare unicamente la parte frazionaria fa sì che la texture si ripeta.

Gli elementi di un eventuale texture di bordo sono accessibili solo se il drappeggio è impostato a GL_CLAMP.

Il valore di default di GL_TEXTURE_WRAP_S è GL_REPEAT.

- **GL_TEXTURE_WRAP_T**: Imposta il drappeggio per la coordinate di texture T a GL_CLAMP or GL_REPEAT.
Il valore di default di GL_TEXTURE_WRAP_T è GL_REPEAT.

Esiste una seconda versione di glTexParameterf: **glTexParameterfv/glTexParameteriv**.

Differisce dalla prima versione solo per un tratto: l'impostazione di un ulteriore parametro.

- **GL_TEXTURE_BORDER_COLOR**: Imposta un colore per il bordo. *Params* può contenere 4 valori che esprimono il colore scelto in formato RGBA. Il valore di default è (0, 0, 0, 0).

	Esempio
	<pre>glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST); glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST); glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP); glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);</pre>

Vediamo ora la seconda funzione promessa: **glTexImage2D**.

La funzione accetta come parametri:

- GLenum *target*: Specifica la natura della texture, deve essere GL_TEXTURE_2D. OpenGL® ES non supporta texture 1D o 3D.
- GLint *level*: Specifica il livello di dettaglio. Il livello 0 rappresenta l'immagine base, mentre il livello N la riduzione mipmap ennesima.
- GLint *components*: Specifica il numero di componenti di colore nella texture. Può assumere i valori 1,2,3 oppure 4.
- GLsizei *width*: Specifica la larghezza dell'immagine texture. Deve essere pari a 2n+2 (bordo) per qualche intero n. Può essere ricavata dal BITMAPINFOHEADER.

- `GLsizei height`: Specifica l'altezza dell'immagine texture. Deve essere pari a $2n+2$ (bordo) per qualche intero n . Può essere ricavata dal `BITMAPINFOHEADER`.
- `GLint border`: Specifica la larghezza del bordo. Può assumere i valori 0 oppure 1.
- `GLenum format`: Specifica il formato dei dati dei pixel. Sono accettati 8 valori che discuteremo in dettaglio nel seguito.
- `GLenum type`: Specifica il tipo dei dati dei pixel. Sono accettati 8 valori: **`GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT` e `GL_FLOAT`**.
- `const GLvoid *pixels`: Specifica un puntatore in memoria ai dati dell'immagine.

In generale, i dati dell'immagine vengono letti a partire da *pixel* come una sequenza di valori del tipo indicato da *type*. Questi valori vengono raggruppati in insiemi da 1,2,3 o 4 valori a seconda di *format*. Un'immagine può avere un massimo di 4 valori per elemento di texture, a seconda di *components*. Un'immagine texture con 1 componente utilizza soltanto la componente rossa del colore RGBA estratto da *pixel*. Una con 2 componenti utilizza solo le componenti di rosso e trasparenza. Una con 3 utilizza solo le componenti di rosso, blu e verde. Una con 4 utilizza tutti e quattro i valori RGBA.

Analizziamo, ora, gli 8 valori assumibili dal parametro *format*.

- **`GL_RED`**: Ogni elemento rappresenta una diversa concentrazione di rosso. Il valore viene convertito in formato float e il valore RGBA del colore del pixel corrispondente viene ottenuto mettendo a 0.0f i valori di verde e blu, nonché 1.0f per la trasparenza.
- **`GL_GREEN`**: Ogni elemento rappresenta una diversa concentrazione di verde. Il valore viene convertito in formato float e il valore RGBA del colore del pixel corrispondente viene ottenuto mettendo a 0.0f i valori di rosso e blu, nonché 1.0f per la trasparenza.
- **`GL_BLUE`**: Ogni elemento rappresenta una diversa concentrazione di blu. Il valore viene convertito in formato float e il valore RGBA del colore del pixel corrispondente viene ottenuto mettendo a 0.0f i valori di rosso e verde, nonché 1.0f per la trasparenza.
- **`GL_ALPHA`**: Ogni elemento rappresenta una diversa concentrazione di trasparenza. Il valore viene convertito in formato float e il valore RGBA del colore del pixel corrispondente viene ottenuto mettendo a 0.0f i valori di rosso, blu e verde.
- **`GL_RGB`**: Ogni elemento rappresenta una tripla nel formato RGB. I valori vengono convertiti in formato float e il valore RGBA del colore del pixel corrispondente viene ottenuto mettendo a 1.0f il valore per la trasparenza.
- **`GL_RGBA`**: Ogni elemento rappresenta un elemento nel formato RGBA. I valori vengono convertiti in formato float.
- **`GL_LUMINANCE`**: Ogni elemento rappresenta una diversa concentrazione di luminosità. Il valore viene convertito in formato float e il valore RGBA del colore del pixel corrispondente viene ottenuto replicando il valore di luminosità tre volte, impostando così i valori di rosso, blu e verde, e mettendo a 1.0f il valore di trasparenza.
- **`GL_LUMINANCE_ALPHA`**: Ogni elemento rappresenta una coppia di diverse concentrazioni di luminosità e trasparenza. I valori vengono convertiti in formato float e il valore RGBA del colore del pixel corrispondente viene ottenuto replicando il valore di luminosità tre volte, impostando così i valori di rosso, blu e verde, e mettendo il valore di trasparenza a quello corrispondente nella coppia di valori dati.

	Esempio
	<pre>glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, info_Header.biWidth, info_Header.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, Img);</pre>

Se si desidera utilizzare solo una parte dell'immagine, può essere utilizzata la funzione **`glTexSubImage2D`** che ha gli stessi parametri della precedente tranne altri 2: `GLint xoffset` e `GLint yoffset`, tramite i quali poter specificare fin dove utilizzare l'immagine sull'asse delle x e delle y.

5.4 Trasparenza

Un'effetto interessante e facile da ottenere è quello della trasparenza. Precedentemente, nel paragrafo 3.6, abbiamo potuto apprezzare gli effetti del blending. Ora utilizzeremo gli stessi strumenti per ottenere un effetto di trasparenza sulla nostra forma, stavolta, solida.

Il primo passo è abilitare il blending nella funzione di inizializzazione.

	Esempio
	<pre>void Inizializza() { [...] glEnable(GL_BLEND); }</pre>

L'effetto di trasparenza si otterrà semplicemente utilizzando la combinazione di parametri (GL_SRC_ALPHA, GL_ONE).

	Esempio
	<pre>glBlendFunc(GL_SRC_ALPHA, GL_ONE);</pre>

Onde godere appieno dell'effetto, tutte le facce della nostra forma andranno disegnati, dove ciò significa includere nella visualizzazione le facce posteriori che, inserendo la trasparenza, diventano visibili. Disabiliteremo quindi il test di profondità.

	Esempio
	<pre>glDisable(GL_DEPTH_TEST); [...]</pre>

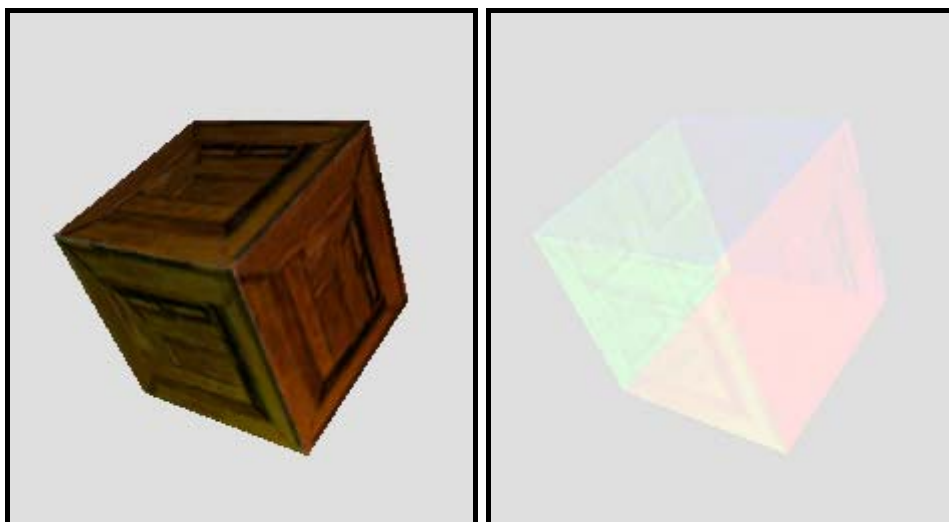


Fig 5.3 Trasparenza. Le facce sono state colorate sotto la texture per maggior visibilità dell'effetto.

☞ 5.5 Nebbia ☞

Un altro effetto, atto a rendere più reale la scena, è la nebbia. Oltre a dare un tocco di realistica alla scena, un effetto di questo genere può risultare notevolmente utile anche a scopi pratici. Immaginiamo di voler rappresentare nella nostra scena un altopiano con

vegetazione rada al punto giusto da avere la visuale libera per chilometri oltre il punto di vista. Naturalmente disegnare tutto ciò che è visibile a qualsiasi distanza sarebbe estremamente costoso, computazionalmente parlando. Inoltre, non è mai strettamente necessario disegnare fino a questo livello di dettaglio. Come si può osservare in molti giochi attuali, il metodo utilizzato per ovviare questo problema è fare oculato uso della nebbia, che può così coprire il paesaggio oltre una certa distanza.

OpenGL® ES fornisce metodi di alto livello al fine di rendere questo effetto. Vediamo quali.

La funzione preposta a questo compito è la **glFogf/glFogfv**. Essa accetta come parametri :

- **Glenum pname** : Specifica una proprietà della nebbia, espressa come flag. Accetta i seguenti valori : **GL_FOG_MODE**, **GL_FOG_DENSITY**, **GL_FOG_START**, **GL_FOG_END**, e **GL_FOG_INDEX**.
- **GLfloat(/const GLfloat*) param** : Indica il valore a cui la proprietà specificata in *pname* deve essere impostata.

Nella versione **glFogfv** il parametro *pname* accetta anche il flag **GL_FOG_COLOR**, il cui valore, definito in *param*, deve essere costituito da un array di 4 valori.

Prima di analizzare nel dettaglio i flag visti, premettiamo che in OpenGL® ES è possibile definire tre tipi di nebbia, identificati da altrettanti flag.

Flag	Descrizione dell'effetto
GL_EXP	La forma più semplice di nebbia. Un oggetto che si muova non rende l'impressione reale del muoversi all'interno di un banco di nebbia. Si tratta di un semplice effetto di distorsione dell'intera scena.
GL_EXP2	Versione più avanzata dell'effetto precedente. L'effetto del muoversi all'interno del banco di nebbia viene reso. Tuttavia la nebbia non appare totalmente realistica poiché la linea di confine tra gli spazi pieni di nebbia e gli altri è molto netta.
GL_LINEAR	La nebbia più realistica. Gli oggetti vi si muovono da dentro a fuori e viceversa nel modo più realistico possibile.

Sebbene il GL_LINEAR sia il miglior tipo di nebbia, ciò non vuol dire che vada sempre utilizzato. Ovviamente, migliore è l'effetto nebbia più lento sarà il programma, quindi è necessaria una scelta ben ponderata.

Analizziamo, ora più in profondità, i 6 possibili valori del parametro *pname* della funzione **glFog**.

- **GL_FOG_MODE**: *params* accetta come valori **GL_LINEAR**, **GL_EXP**, e **GL_EXP2**, definendo così il tipo di nebbia scelta. Di default il valore è **GL_EXP**.
- **GL_FOG_DENSITY**: *Params* accetta un valore non negativo, intero o float, che specifica la densità della nebbia. Più il valore è alto più la nebbia sarà densa. Il valore di default è 1.0.
- **GL_FOG_START**: *params* accetta un valore intero o float, che specifica il punto di inizio della nebbia, come distanza dall'osservatore. Non vi è nebbia prima di questo punto. Il valore di default è 0.0.
- **GL_FOG_END**: *params* accetta un valore intero o float, che specifica il punto di fine della nebbia, come distanza dall'osservatore. Non vi è nebbia oltre questo punto. Il valore di default è 1.0.
- **GL_FOG_COLOR**: *params* accetta un array di quattro valori interi o float, che specificano il colore della nebbia. Il valore di default è (0,0,0,0).

A questo punto possiamo passare a vedere qualche esempio di codice.

Anzitutto, come sempre, la nebbia va abilitata.

	Esempio
	<code>glEnable(GL_FOG);</code> <code>glDisable(GL_FOG) ;</code>

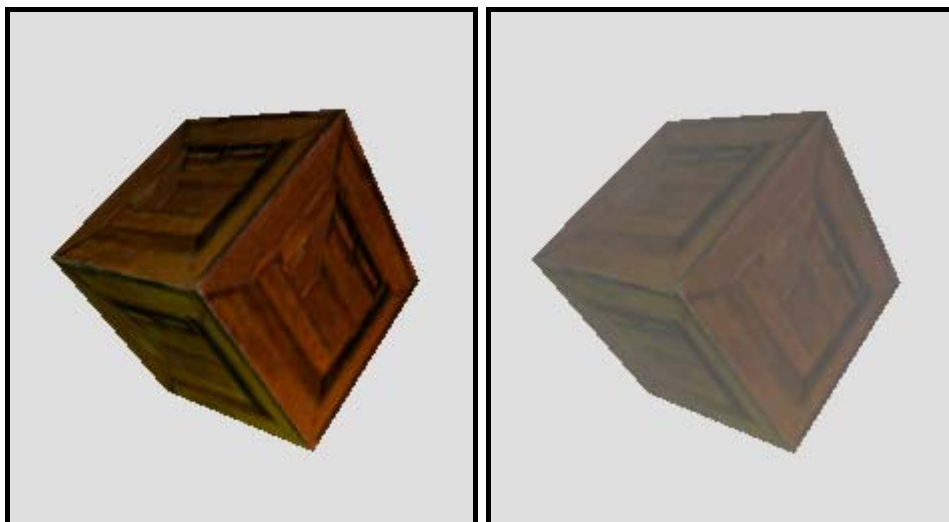
Definiamo, quindi la nebbia e la sua proprietà tramite la funzione `glFogf/glFogfv`.

	Esempio
	<code>glFogf(GL_FOG_MODE, GL_EXP);</code> <code>glFogfv(GL_FOG_COLOR, ArrayCheDefinisceIlColoreDellaNebbia);</code> <code>glFogf(GL_FOG_DENSITY, 0.35f);</code> <code>glFogf(GL_FOG_START, 1.0f);</code> <code>glFogf(GL_FOG_END, 5.0f);</code>

Un'altra funzione, non esclusiva della nebbia, ma atta a definirne una proprietà, è **glHint**. Questa funzione permette di specificare quanto sia importante per noi il rapporto aspetto/resa. Essa accetta due parametri :

- GLenum *target* : può accettare diversi valori. Per ora ci limiteremo a `GL_FOG_HINT`. Impostarne il valore significa definire con quale **accuratezza** la nebbia debba essere resa.
- GLenum *mode* : Definisce il valore del parametro *target*. Può assumere tre valori: `GL_DONT_CARE`, `GL_FASTEST` o `GL_NICEST`. `GL_NICEST` indica che si preferisce curare maggiormente la qualità dell'immagine finale, `GL_FASTEST` indica che si preferisce curare di più l'efficienza e quindi la velocità, `GL_DONT_CARE` indica che non si hanno particolari preferenze. Il valore di default è `GL_DONT_CARE`.

	Esempio
	<code>glHint(GL_FOG_HINT, GL_DONT_CARE);</code>



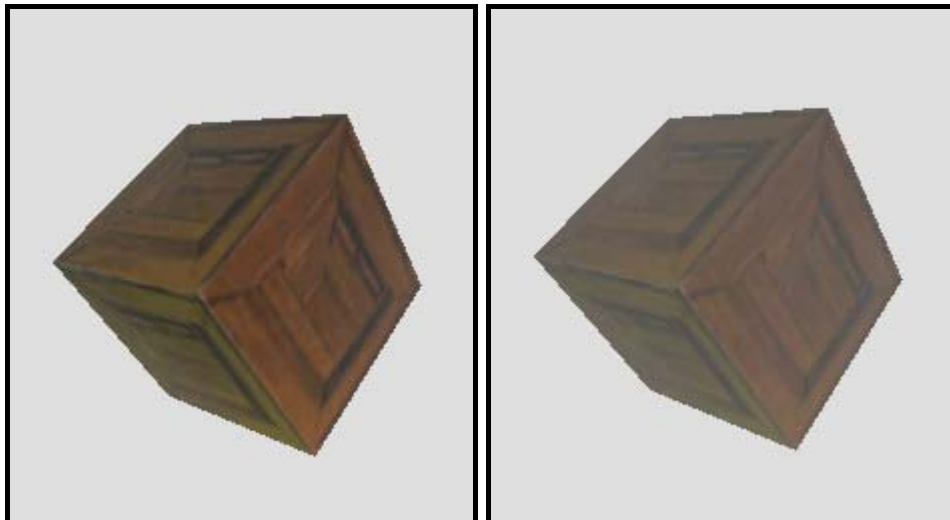


Fig 5.4 Nebbia. Partendo dal più in alto a sinistra troviamo i casi di nebbia disabilitata, nebbia GL_EXP, nebbia GL_EXP2, nebbia GL_LINEAR

5.6 Un esempio di codice: rendere la scena realistica

Vedremo, in questo esempio di fine capitolo, come rendere la nostra scena più realistica, applicando gli effetti visti finora.

L'esempio mostrato si basa su quello presentato nel paragrafo 4.8: la scena, precedentemente realizzata, viene qui arricchita degli effetti spiegati nel capitolo attuale. Si rimanda quindi a detto paragrafo per spiegazioni più complete del codice già ivi analizzato.

Come al solito, carichiamo le librerie necessarie.

Codice
<pre>#pragma comment(lib, "libGLES_CM.lib") #pragma comment(lib, "ug.lib") #include "ug.h"</pre>

Dichiariamo le nostre variabili.

Codice
<pre>float Rotazione_AsseX = 0.0f; float Rotazione_AsseY = 0.0f; float Rotazione_AsseZ = 0.0f; float val=0.5f; int autoRotazione=1; bool Sfumatura = false;</pre>

Dichiariamo le variabili necessarie alla gestione dell'effetto nebbia. L'array *ColoreNebbia* contiene il colore scelto per la nebbia, mentre l'array *TipoNebbia* contiene i tre flag che ne identificano i vari tipi: verrà utilizzato per creare una semplice funzione di tastiera che permetta all'utente di intercambiarli durante l'esecuzione, onde poterne apprezzare le differenze. La variabile *tmpNebbia* ha semplice scopo di contatore, necessario a scorrere gli elementi dell'array.

Codice
<pre>float ColoreNebbia[] = { 0.5f, 0.5f, 0.5f, 1.0f }; float TipoNebbia[] = { GL_EXP, GL_EXP2, GL_LINEAR }; int tmpNebbia = 0;</pre>

Si rimanda all'[Appendice B](#) per la spiegazione dettagliata delle procedure necessarie al caricamento di un'immagine. Per ora basti sapere che la variabile *info_Header* ha scopi collegati a detta procedura. La sua dichiarazione in questo punto è dovuta al fatto che, più avanti, verrà utilizzata da funzioni esterne a quella relativa al caricamento vero e proprio, e quindi deve essere dichiarata globalmente.

Codice
<pre>BITMAPINFOHEADER info_Header;</pre>

L'array *texture* conterrà gli eventuali texture name necessari. Per ora prevediamone quattro. La variabile *tmpTexture* ha semplice scopo di contatore, necessario a scorrere gli elementi dell'array.

Codice
<pre>static GLuint texture[4]; int tmpTexture=-1;</pre>

Definiamo i vertici del nostro cubo e le coordinate di texture relative.

Codice
<pre>GLfloat Cubo[] = { // Faccia frontale -val, -val, val, val, -val, val, -val, val, val, val, val, val, // Faccia di retro -val, -val, -val, -val, val, -val, val, -val, -val, val, val, -val, // Faccia laterale sinistra -val, -val, val, -val, val, val, -val, -val, -val, -val, val, -val, // Faccia laterale destra val, -val, -val, val, val, -val, val, -val, val, val, val, val, // Base (alto) -val, val, val,</pre>

```

        val, val, val,
        -val, val, -val,
        val, val, -val,
        // Base (basso)
        -val, -val, val,
        -val, -val, -val,
        val, -val, val,
        val, -val, -val,
    };

    GLfloat CoordinateTexture[] = {
        // Faccia laterale sinistra
        1.0f, 0.0f,
        1.0f, 1.0f,
        0.0f, 0.0f,
        0.0f, 1.0f,
        // Faccia laterale destra
        1.0f, 0.0f,
        1.0f, 1.0f,
        0.0f, 0.0f,
        0.0f, 1.0f,
        // Base (alto)
        0.0f, 0.0f,
        1.0f, 0.0f,
        0.0f, 1.0f,
        1.0f, 1.0f,
        // Base (basso)
        1.0f, 0.0f,
        1.0f, 1.0f,
        0.0f, 0.0f,
        0.0f, 1.0f,
        // Faccia frontale
        0.0f, 0.0f,
        2.0f, 0.0f,
        0.0f, 2.0f,
        2.0f, 2.0f,
        // Faccia di retro
        1.0f, 0.0f,
        1.0f, 1.0f,
        0.0f, 0.0f,
        0.0f, 1.0f
    };

```

Definiamo i valori delle proprietà dell'illuminazione e del materiale.

Codice

```

GLfloat AmbientLuce0[]      = { 0.9f, 0.9f, 0.9f, 1.0f }; //colore della luce
GLfloat DiffuseLuce0[]      = { 0.0f, 0.0f, 0.0f, 1.0f };
GLfloat SpecularLuce0[]    = { 1.0f, 1.0f, 1.0f, 0.0f };

GLfloat AmbientMat[]        = { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat DiffuseMat[] = { 1.0f, 0.0f, 0.0f, 1.0f };
GLfloat SpecularMat[]       = { 1.0f, 0.0f, 0.0f, 1.0f };
GLfloat ShineMat[]          = { 40.0f };

```

Si riporta, per gusto di completezza, il codice necessario al caricamento dell'immagine di texture. Si rimanda all'[Appendice B](#) per informazioni dettagliate sul suo funzionamento.

	Codice
	<pre> unsigned char *CaricaImmagine(char *NomeDelFile){ FILE *FileImmagine; BITMAPFILEHEADER file_Header; unsigned char *Immagine = NULL; unsigned char tmp_daBGRaRGB; TCHAR Percorso[256]; char PercorsoImmagine[256]; GetModuleFileName(NULL, Percorso, 256); TCHAR *posizBarra = wcsrchr(Percorso, '\\'); *(posizBarra + 1) = '\\0'; wcstombs(PercorsoImmagine, Percorso, 256); strcat(PercorsoImmagine, NomeDelFile); FileImmagine = fopen(PercorsoImmagine,"rb"); fread(&file_Header,sizeof(BITMAPFILEHEADER),1,FileImmagine); if (file_Header.bfType != 0x4D42) { MessageBox(NULL, L"Formato immagine non corretto: richiesto formato bmp", L"Errore", MB_OK); fclose(FileImmagine); return NULL; } fread(&info_Header,sizeof(BITMAPINFOHEADER),1,FileImmagine); fseek(FileImmagine,file_Header.bfOffBits,SEEK_SET); Immagine = new unsigned char[info_Header.biSizeImage]; fread(Immagine,1,info_Header.biSizeImage,FileImmagine); for (unsigned int i = 0; i < info_Header.biSizeImage; i+=3) { tmp_daBGRaRGB = Immagine[i]; Immagine[i] = Immagine[i+2]; Immagine[i+2] = tmp_daBGRaRGB; } fclose(FileImmagine); return Immagine; } </pre>

Passiamo alle inizializzazioni. Anzitutto definiamo le proprietà della nebbia e ne abilitiamo l'utilizzo.

	Codice
	<pre> void Inizializza(){ glFogf(GL_FOG_MODE, GL_EXP); } </pre>

```

glFogfv(GL_FOG_COLOR, ColoreNebbia);
glFogf(GL_FOG_DENSITY, 0.40f);
glFogf(GL_FOG_START, 1.0f);
glFogf(GL_FOG_END, 5.0f);
glHint(GL_FOG_HINT, GL_DONT_CARE);
glEnable(GL_FOG);

```

La variabile *Img* verrà usata per contenere i dati dell'immagine di texture una volta caricata dal filesystem.

Codice

```

unsigned char *Img=NULL;

```

Effettuiamo le inizializzazioni di routine.

Codice

```

glEnableClientState(GL_COLOR_ARRAY);
glClearDepthf(1.0f);
glClearColor (0.87f, 0.87f, 0.87f, 0.0f);
glShadeModel(GL_SMOOTH);

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightfv(GL_LIGHT0, GL_AMBIENT, AmbientLuce0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, DiffuseLuce0);
glLightfv(GL_LIGHT0, GL_SPECULAR, SpecularLuce0);

glEnable(GL_COLOR_MATERIAL);
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, AmbientMat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, DiffuseMat);
glMaterialfv(GL_FRONT, GL_SPECULAR, SpecularMat);
glMaterialfv(GL_FRONT, GL_SHININESS, ShineMat);

glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LEQUAL);

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, Cubo);

glEnable(GL_CULL_FACE);
glFrontFace (GL_CCW);
glCullFace (GL_BACK);

```

Indichiamo dove recuperare le coordinate di texture.

Codice

```

glTexCoordPointer(2, GL_FLOAT, 0, CoordinateTexture);

```

Carichiamo l'immagine di texture, tramite la funzione personalizzata che abbiamo creato. Nel caso non sia stato possibile farlo, visualizziamo un messaggio d'errore.

Codice
<pre> Img=CaricaImmagine("img.bmp"); if (!Img) { MessageBox(NULL, L"Errore durante il caricamento dell'immagine", L"Errore", MB_OK); } </pre>

Generiamo i texture name e riponiamoli nell'array creato per questo scopo.

Codice
<pre> glGenTextures(4, texture); </pre>

Leghiamo la texture ad uno dei texture name generati e definiamone le proprietà. Infine, conclusa la necessità di conservarla, cancelliamo l'immagine dalla variabile che la conteneva, di modo da ottimizzare la funzione.

Codice
<pre> glBindTexture(GL_TEXTURE_2D, texture[0]); glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST); glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST); glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, info_Header.biWidth, info_Header.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, Img); </pre>

Infine, conclusa la necessità di conservarla, cancelliamo l'immagine dalla variabile che la conteneva, di modo da ottimizzare la funzione.

Codice
<pre> delete[] Img; </pre>

Ripetiamo il processo più volte modificando le proprietà di texture. Più avanti permetteremo, su pressione di un tasto della tastiera, di intercambiare le texture prodotte, per mostrare le differenze tra le varie proprietà impostabili sulla stessa texture.

Il lettore noti come il caricamento avvenga più volte con parametro lo stesso nome di immagine. Ciò, naturalmente, non è strettamente necessario, ma viene previsto nel codice per dare la struttura che permetterà un'importante modifica: cambiando i nomi delle immagini da caricare sarà possibile caricare immagini differenti, ed intercambiarle tramite tastiera. L'intercambio avverrà in maniera molto semplice. La funzione che, a tutti gli effetti, definisce quale sia la texture attuale è la **glBindTexture**, quindi sarà sufficiente richiamarla per scegliere la texture da visualizzare.

Un'ultima nota tecnica: le immagini da caricare, onde permetterne il ritrovamento da parte dell'eseguibile, devono essere copiate nella cartella dell'emulatore/dispositivo che contiene detto eseguibile.

	Codice
	<pre> Img=CaricaImmagine("img.bmp"); if (!Img) { MessageBox(NULL, L"Errore durante il caricamento dell'immagine", L"Errore", MB_OK); } glBindTexture(GL_TEXTURE_2D, texture[1]); glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, info_Header.biWidth, info_Header.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, Img); delete[] Img; Img=CaricaImmagine("img.bmp"); if (!Img) { MessageBox(NULL, L"Errore durante il caricamento dell'immagine", L"Errore", MB_OK); } glBindTexture(GL_TEXTURE_2D, texture[2]); glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT); glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE); glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, info_Header.biWidth, info_Header.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, Img); delete[] Img; Img=CaricaImmagine("img.bmp"); if (!Img) { MessageBox(NULL, L"Errore durante il caricamento dell'immagine", L"Errore", MB_OK); } glBindTexture(GL_TEXTURE_2D, texture[3]); glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST); glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); glTexParameterf(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE); glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, info_Header.biWidth, info_Header.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, Img); delete[] Img; } </pre>

Passiamo al disegno, la definizione delle funzioni di aggiornamento, **idle** e **main**: nulla cambia rispetto a quanto già visto in [4.8](#).

	Codice
	<pre> void Disegna(UGWindow Finestra) { glClear(GL_COLOR_BUFFER_BIT GL_DEPTH_BUFFER_BIT); </pre>

```

glLoadIdentity();

ugluLookAtf(0.0f, 0.0f, 3.0f,
              0.0f, 0.0f, 0.0f,
              0.0f, 1.0f, 0.0f);

glRotatef(Rotazione_AsseX, 1.0f, 0.0f, 0.0f);
glRotatef(Rotazione_AsseY, 0.0f, 1.0f, 0.0f);
glRotatef(Rotazione_AsseZ, 0.0f, 0.0f, 1.0f);

// Faccia frontale e retro
glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
glNormal3f(0.0f, 0.0f, 1.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
glNormal3f(0.0f, 0.0f, -1.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 4, 4);

// Faccia laterale sinistra e destra
glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
glNormal3f(-1.0f, 0.0f, 0.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 8, 4);

glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
glNormal3f(1.0f, 0.0f, 0.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 12, 4);

// Basi (alto e basso)
glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
glNormal3f(0.0f, 1.0f, 0.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 16, 4);

glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
glNormal3f(0.0f, -1.0f, 0.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 20, 4);

glFlush();
ugSwapBuffers(Finestra);
}

void Aggiorna(UGWindow Finestra, int Larghezza, int Altezza){

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    glViewport(0, 0, Larghezza, Altezza);
    ugluPerspectivef(45.0f, 1.0f * Larghezza / Altezza, 1.0f, 100.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void Idle(UGWindow Finestra)
{
    if(autoRotazione==1){

```

```

        Rotazione_AsseX += 1.0f;
        Rotazione_AsseY += 3.0f;
        Rotazione_AsseZ += 1.0f;
    }
    ugPostRedisplay(Finestra);
}

int main()
{
    UGCtx ug = ugInit();

    UGWindow Finestra = ugCreateWindow(ug, "UG_DEPTH", "OpenGL ES", 250,
    250, 100, 100);

    Inizializza();

    ugDisplayFunc(Finestra, Disegna);
    ugKeyboardFunc(Finestra, Tastiera);
    ugReshapeFunc(Finestra, Aggiorna);
    ugIdleFunc(ug, Idle);

    ugMainLoop(ug);
    return 0;
}

```

Vediamo, infine, le aggiunte fatte alle possibilità di interazione di tastiera data all'utente.

Tasto premuto	Effetto
t	Possibilità di intercambiare le 4 texture caricate, premendo più volte il tasto. Alla quinta pressione il texture mapping viene disabilitato
n	Disabilitazione su pressione ed abilitazione su successiva pressione dell'effetto nebbia
o	Possibilità di intercambiare i tre tipi di nebbia disponibili, premendo più volte il tasto

Codice

```

void Tastiera(UGWindow Finestra, int tastoPremuto, int x, int y)
{
    switch(tastoPremuto)
    {
        case 'e' :
            PostQuitMessage(0);
            break;

        case '0' :
            if (glIsEnabled(GL_LIGHT0)){
                glDisable(GL_LIGHT0);
            } else {
                glEnable(GL_LIGHT0);
            }
            break;
    }
}

```



```

case 'l' :
    if (glIsEnabled(GL_LIGHTING)){
        glDisable(GL_LIGHTING);
        glDisable(GL_LIGHT0);
    } else {
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
    }
    break;

case 't' :
    if (tmpTexture<5)
        tmpTexture +=1;
    else
        tmpTexture=0;

    glBindTexture(GL_TEXTURE_2D, texture[tmpTexture]);

    if(tmpTexture==5)
        glDisable(GL_TEXTURE_2D);
    else
        glEnable(GL_TEXTURE_2D);
    break;

case 'x' :
    Rotazione_AsseX += 2.0;
    break;

case 'y' :
    Rotazione_AsseY += 2.0;
    break;

case 'z' :
    Rotazione_AsseZ += 2.0;
    break;

case 'a' :
    if (autoRotazione==0){
        autoRotazione=1;
    } else {
        autoRotazione=0;
    }
    break;

case 's':
    Sfumatura = !Sfumatura;
    glShadeModel(Sfumatura? GL_SMOOTH : GL_FLAT);
    ugPostRedisplay(Finestra);
    break;

case 'n' :
    if (glIsEnabled(GL_FOG)){
        glDisable(GL_FOG);
    } else {
        glEnable(GL_FOG);
    }
    break;

```

```

case 'o' :
    ++tmpNebbia %= 3;
    glFogf(GL_FOG_MODE, TipoNebbia[tmpNebbia]);
break;

```

Analizziamo a parte il codice richiamato su pressione del tasto "b", poiché in esso viene ottenuto l'effetto di trasparenza. Premendo il tasto è possibile abilitare, o disabilitare su ulteriore pressione, la trasparenza del nostro cubo:

- Nel caso il blending sia disabilitato, lo si abilita e si seleziona la funzione di blending atta a rendere l'effetto di trasparenza. Per questioni di apprezzamento immediato dell'effetto, legate soltanto alle impostazioni dell'esempio attuale, disabilitiamo la luce zero e la nebbia. Come spiegato nel paragrafo relativo, è necessario disabilitare il face culling e il test di profondità, perché siano visibili le facce posteriori, ora in vista poiché le anteriori sono state rese trasparenti.
- In caso contrario, per eliminare la trasparenza, disabilitiamo il blending. E renderemo nuovamente attivi la luce zero, il face culling, il test di profondità e l'effetto nebbia.

Codice

```

case 'b' :
    if (glIsEnabled(GL_BLEND)){
        glDisable(GL_BLEND);
        glEnable(GL_LIGHT0);
        glEnable(GL_CULL_FACE);
        glEnable(GL_DEPTH_TEST);
        glEnable(GL_FOG);
    } else {
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE);
        glDisable(GL_LIGHT0);
        glDisable(GL_CULL_FACE);
        glDisable(GL_DEPTH_TEST);
        glDisable(GL_FOG);
    }
}

```

☺☺ Appendici ☹☹

☺☺ A. Preparare l'ambiente di lavoro ☹☹

☺☺ A.1 Introduzione ☹☹

Il corso è strutturato in modo tale da contenere vari esempi. Proponiamo in questa appendice un ambiente di lavoro e la sua preparazione in vista dell'inizio della programmazione vera e propria, per permettere allo studente di testare con più facilità gli esempi presentati. Ci concentreremo su di un unico ambiente, per semplicità, che è stato scelto tra gli altri poiché considerato di facile uso, reperibilità e attendibilità.

Il lettore sia ben consapevole del fatto che le procedure proposte in questa appendice costituiscono una linea guida, un esempio di come vada affrontato il problema dell'approntare un ambiente di lavoro. Non possono per loro natura ambire a più di questo. Essendo, infatti, procedimenti molto specifici, implicano un'intrinseca mancanza di flessibilità: il loro funzionamento è, cioè, assicurato solo e soltanto nel caso essi vengano seguiti alla lettera. La presenza di un differente sistema operativo, o di una patch più avanzata dei programmi, può minarne in parte l'efficacia. Si consiglia quindi, dove possibile, di seguire passo passo quanto descritto, o altrimenti, in caso di malfunzionamenti, di riferirsi ai manuali di riferimento dei rispettivi programmi utilizzati.

☺☺ A.2 Programmare in OpenGL® ES ☹☹

Esistono vari ambienti di programmazione e vari dispositivi atti alla programmazione e l'uso di OpenGL® ES. Per semplicità, nel seguito si adotterà un unico ambiente di programmazione: Microsoft Embedded Visual C++, con l'appoggio del simulatore PocketPc 2003, con sistema operativo sottostante Windows XP. Ciò non significa che il codice presentato non possa essere utilizzato se non si utilizzino questi strumenti, ovviamente. Le uniche differenze riscontrabili si trovano nell'impostazione dell'ambiente di sviluppo.

Ad esempio, per lavorare su Smartphone invece che su Pocket Pc sarà necessario solo scaricare il giusto compilatore. (NB: entrambi i sistemi non sono dotati di un processore x86 compatibile) Quindi, oltre all'ambiente di sviluppo, sarà necessario MS Smartphone 2003 SDK. Questo software può essere scaricato [qui](#), ed è persino possibile richiedere il dvd che lo contenga direttamente alla [Microsoft](#).

☺☺ A.3 Preparare l'ambiente di lavoro ☹☹

Ci sono vari ambienti con cui poter lavorare ma, come già affermato, ci concentreremo su Microsoft Embedded Visual C++. Questi sono i link dove poter scaricare il materiale necessario:

1. [Embedded Visual C++ 4.0](#)
2. [Embedded Visual C++ 4.0 Service Pack 4](#)
3. [Pocket PC 2003 SDK](#)
4. [Vincent Mobile 3D Rendering Library](#)

I primi tre link conducono a file muniti di classica installazione modello Windows. Per il quarto è necessaria invece una certa attenzione nel distribuire i file forniti nelle corrette locazioni.

Una volta scompattato il file scaricato, sarà necessario copiare alcuni file nella *directory* \ *Programmi* \ *Windows CE Tools* \ *wce420* \ *POCKET PC 2003* \ *Include* (con \ *Programmi* \ *Windows CE Tools* la *directory* di installazione di Pocket Pc, cui d'ora in poi ci riferiremo come \ *pocketpc* \). Nelle sub-*directory* *Lib* e *Include*, vi sono altre due *directory* *Armv4* e *Emulator*. Queste *directory* vengono usate a turno a seconda di determinati parametri di compilazione impostabili dall'utente. Si intenda nel seguito per "*dist*" la *directory* che contiene file di Vincent. Infatti, essa non porta lo stesso nome per tutte le distribuzioni. E' necessario attenersi ai seguenti passi:

Copiare il contenuto della directory `\dist\include` in `\pocketpc\Include\Armv4` e in `\pocketpc\Include\Emulator`.

Copiare il contenuto della directory `\dist\bin\emu\Debug` in `\pocketpc\Lib\Emulator` con l'eccezione del file `libGLES_CM.dll`.

Copiare il contenuto della directory `\dist\bin\arm\Release` in `\pocketpc\lib\Armv4` con l'eccezione del file `libGLES_CM.dll`.

Infine, copiare il contenuto della directory `\dist\include` in `\Programmi\Microsoft eMbedded C++ 4.0\EVC\Include`.

Copiati i file sarà necessario fare un altro paio di passi prima di poter scrivere un qualsiasi programma. Anzitutto, lanciato Microsoft Embedded C++, vanno impostate le directory di Include. Selezionato *Tools > Options > Directories*, fare doppio click su uno degli spazi vuoti nella finestra delle directory e poi selezionare le directory `\Programmi\Microsoft eMbedded C++ 4.0\EVC\Include` e `\Programmi\Microsoft eMbedded C++ 4.0\EVC\Include\GLES`.

Una volta fatto questo, per il funzionamento dell'emulatore di Pocket Pc 2003 sarà necessario seguire un altro breve procedimento. In modalità emulatore Pocket Pc 2003 necessita l'appoggio su una rete lan. Eviteremo il problema installando una rete fittizia. Ciò viene fatto tramite l'installazione della *Scheda Microsoft Loopback*. Riportiamo le istruzioni per la sua installazione sotto Windows XP. Per altre versioni di Windows il procedimento è analogo, tranne piccoli cambiamenti dipendenti dalla distribuzione.

Per installare la Scheda Microsoft Loopback sotto Microsoft Windows XP:

1. Nel Pannello di controllo, fare doppio click su **Installazione Hardware**, e poi su **Avanti**.
2. Selezionare **Sì, l'hardware è già stato collegato**, e poi **Avanti**.
3. Nella lista **Hardware installato**, scorrere fino all'ultima opzione e fare click su **Aggiungi nuova periferica hardware**
4. Selezionare **Installa l'hardware selezionato manualmente da un elenco (Utente esperto)**, e poi **Avanti**.
5. Nella lista **Tipi di hardware comuni**, fare click su **Schede di rete**, e poi su **Avanti**.
6. Nella lista **Produttore**, fare click su **Microsoft**.
7. Nella lista **Scheda di rete**, fare click su **Scheda Microsoft Loopback**, e poi su **Avanti**.
8. Fare click su **Avanti**, and poi su **Fine**.

Fatto ciò, dovremo configurare il *Platform Manager* di Microsoft Embedded C++ in modo da utilizzare l'IP della connessione fittizia appena creata.

Aperto Microsoft Embedded C++, fare click su *Tools -> Configure Platform Manager*, poi scegliere l'emulatore utilizzato (POCKET PC 2003 emulator oppure SMARTPHONE 2003 emulator) e fare click su *Properties*.

Sulla voce *Transport* selezionare *TCP/IP Transport for Windows CE* e da *Configure* configurarlo per utilizzare un indirizzo fisso (*Fixed address*).

Prendere nota dell'IP che viene visualizzato nella tendina accanto alla voce *Use Fixed Address*. Questo IP andrà impostato come quello utilizzato dalla connessione fittizia.

Come ultima cosa, nella finestra dove si trovava la voce *Transport*, nella tendina *Emulator startup* scegliere *Emulator Startup Server* e in *Configure* selezionare nella parte *Communication* la voce *NAT (outgoing only)*.

Fatto ciò, in Windows, da *Pannello di Controllo* -> *Connessioni di rete* -> *Connessione alla rete locale*, selezionare, facendo click destro, le *Proprietà della connessione*. Selezionare poi *Protocollo Internet (TCP/IP)* e *Proprietà*. Selezionato *Utilizza il seguente IP* riempire il campo *Indirizzo IP* con l'indirizzo che si trovava nel *Platform Manager* e la *Subnet Mask* con 255.255.0.0.

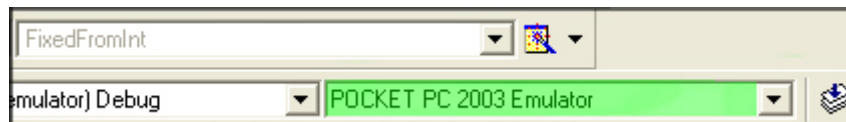
Ora siamo pronti per scrivere il nostro primo programma!

« A.4 Creare un nuovo progetto in Microsoft Embedded C++ »

Il primo passo per creare un programma è creare un "progetto".

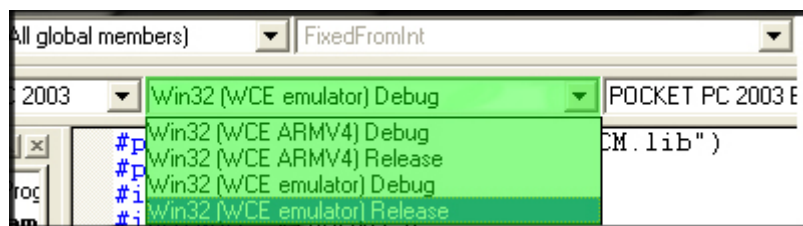
Fare click su *File* e poi su *New*. Selezionare *WCP POCKET PC 2003 Application* e inserire il nome del proprio progetto nella casella di testo *Project Name*, quindi fare click su *OK*. Nella finestra che appare, selezionare *An empty project* e premere *Finish*.

Verrà aperto il progetto, i cui file potranno essere trovati nella directory che porta il nome scelto per il progetto e che risiede nella directory */Microsoft eMbedded C++ 4.0/Common/EVC/MyProjects/*. Selezionare nella tendina, sopra la finestra del codice, *POCKET PC 2003 Emulator*.



Premere poi *F7* o *Build > Build TestProject.exe*. Apparso l'emulatore, fare click su *Emulator > Folder Sharing* e specificare una directory sul proprio hard disk. Il contenuto di questa directory sarà reso disponibile nella directory *StorageCard* dell'emulatore.

Un'ultima cosa da ricordare: è necessario selezionare la versione per cui si voglia compilare. Va selezionata *Win32 (WCE emulator) Debug* nel caso si voglia testare il programma sull'emulatore e *Win32 (WCE ARMV4) Release* nel caso lo si voglia fare sul Pocket Pc reale.



Nel caso si voglia testare il proprio codice su un reale Pocket Pc è necessario copiarvi il file */dist/bin/emu/Debug/libGLES_CM.dll*.

Anche nel caso si voglia testare il programma sull'emulatore è necessario copiarvi il suddetto file. Sfortunatamente, non esiste un modo automatico di fare ciò, quindi bisogna copiare il file ogniqualvolta si faccia partire l'emulatore. Per fare questo, selezionare *Tools > Remote File Viewer*, poi selezionare nella finestra che compare *Pocket Pc 2003 Emulator* e premere *OK*. Selezionare la cartella *Windows* e fare doppio click, quindi premere il bottone *Export file to device* (freccia gialla che punta verso l'alto) e selezionare il file */dist/bin/emu/Debug/libGLES_CM.dll*. La libreria verrà così caricata sull'emulatore.

Creato il progetto, è possibile cominciare a scriverne il codice. Selezionando dal menù *File -> New* sarà possibile creare un nuovo file di codice da aggiungere al progetto creato. Selezionare, quindi, nella finestra che verrà aperta, la voce *C++ Source File* e includerlo nel progetto corrente selezionandolo dalla tendina *Add To project*. In ultimo, sarà necessario assegnargli un nome, nella casella di testo *File name*, e procedere alla scrittura del codice. Se si possiede già un file di codice basti selezionare la voce *Project -> Add to project -> Files* per aggiungerlo al proprio progetto.

Per compilarlo ed eseguirlo scegliere dal menù *Build -> Rebuild all*. In assenza di errori, l'emulatore Pocket Pc 2003 si avvierà automaticamente ed al suo interno sarà caricato l'eseguibile del codice appena compilato. Dalla finestra dell'emulatore, selezionare dunque dal menù *Start* la voce *Programmi* e quindi *File Explorer*, un programma molto simile al suo analogo della versione Windows classica. Di default, la cartella selezionata sarà *My Documents*. Premiamo allora sulla voce *My documents* per aprire un menù a tendina da cui selezioneremo *My device*, cioè la cartella in cui il nostro eseguibile è stato caricato. Un click sull'eseguibile e sarà possibile vedere il risultato del proprio lavoro.

« B. Caricare un'immagine: WindowsBitmap »

Il passo forse più laborioso dell'applicazione di una texture è il caricamento dell'immagine. Vediamo in dettaglio un esempio di funzione che abbia proprio il fine di caricare una texture (in generale un'immagine), in questo caso di tipo Windows Bitmap, cioè il formato dati utilizzato per la rappresentazione di immagini sui sistemi operativi Microsoft Windows.

Ovvio è che, ai fini di operare questo processo, è necessario decidere previamente quale tipo di formato immagine si desidera utilizzare e conoscerne la natura nel dettaglio. Come si vedrà, infatti, per il caricamento sarà necessario analizzare ogni bit del file immagine e destinarlo al proprio ruolo.

NB: Ai fini dell'esempio, è accettabile una qualsiasi immagine bitmap 64x64 a 24 bit, ma per renderla compatibile al formato scelto, si consiglia di crearla o almeno salvarla con il classico "Paint" di Windows. Programmi come Photoshop, infatti, utilizzano per il salvataggio in bitmap un formato diverso dal Window Bitmap.

Prima di iniziare citiamo la struttura di un file bitmap standard. Esso è costituito, nell'ordine, da:

- **File Header:** I primi bit dell'immagine contengono un insieme di informazioni relative al file: la dimensione in byte, il tipo, e l'offset tra l'inizio del file e la posizione in esso dei bit dell'immagine (mappa dei pixel). La struttura di tipo `BITMAPFILEHEADER` è pensata proprio allo scopo di contenerli.

<i>BITMAPFILEHEADER (dimensione: 14 byte)</i>			
<i>Offset</i>	<i>Tipo</i>	<i>Nome</i>	<i>Contenuto</i>
0	WORD	bfType	Tipo, identifica il formato
2	DWORD	bfSize	Dimensione del file in byte
6	DWORD	bfReserved	0
10	DWORD	bfOffBits	Offset del primo byte della mappa dei pixel a partire dall'inizio del file

- **Information Header:** Altro insieme di informazioni, stavolta relative all'immagine. Contiene l'altezza, la larghezza, i bit per pixel dell'immagine (cioè la risoluzione) e il numero di colori utilizzati. In questo caso, utilizzeremo la struttura `BITMAPINFOHEADER`.
 - Modello di colore: Nelle ultime versioni del formato, cioè la 4 e la 5, il blocco informazioni è stato ampliato con strutture che consentono di definire, eventualmente, modelli di colore personalizzati. In genere sono poco comuni.
 - Tavolozza: Questo array fa corrispondere un colore ad ogni indice che può essere assegnato ad un pixel. Nella tavolozza ogni colore è rappresentato da una struttura di 4 byte (RGBQUAD), uno ciascuno per i componenti rosso, verde e blu più un byte non utilizzato. Nel caso di immagini con 16, 24 o 32 colori, questa tabella di colori non è necessaria perché il colore dei pixel non è

indicizzato, bensì codificato direttamente nelle sue componenti cromatiche, cioè i quattro valori RGBA.

<i>BITMAPINFOHEADER (dimensione: 40 byte)</i>			
<i>Offset</i>	<i>Tipo</i>	<i>Nome</i>	<i>Contenuto</i>
0	DWORD	biSize	Dimensione in byte del blocco d'informazioni
4	LONG	biWidth	Larghezza dell'immagine in pixel
8	LONG	biHeight	Altezza dell'immagine in pixel: quando il valore è positivo la mappa dei pixel incomincia dalla riga di pixel più in basso e finisce con quella più in alto (variante più comune), viceversa nel caso il valore sia negativo
12	WORD	biPlanes	Sempre 1
14	WORD	biBitCount	Profondità di colore dell'immagine in bit per pixel (1, 4, 8, 16, 24 o 32) In caso di 1, 4 o 8 bit per pixel i colori sono indicizzati. I valori 16 e 32 sono poco comuni
16	DWORD	biCompression	Compressione della mappa dei pixel
20	DWORD	biSizeImage	Dimensione in byte della mappa dei pixel
24	LONG	biXPelsPerMeter	Risoluzione orizzontale del dispositivo di output in pixel per metro; 0 se la risoluzione non è specificata
28	LONG	biYPelsPerMeter	Risoluzione verticale del dispositivo di output in pixel per metro; 0 se la risoluzione non è specificata
32	DWORD	biClrUsed	Numero di corrispondenze effettivamente utilizzate nella tavolozza dei colori; 0 indica il numero massimo (16 o 256) Per profondità maggiori di 8 bit per pixel la tavolozza non è normalmente necessaria, ma quando c'è può essere usata dal sistema o da alcuni programmi per ottimizzare la rappresentazione dell'immagine.
36	DWORD	biClrImportant	Numero di colori utilizzati nell'immagine

- **Mappa dei Pixel:** Corpo vero e proprio della bitmap, dove ad ogni pixel viene fatto corrispondere un colore sotto forma di indice nella tavolozza, oppure nelle sue componenti cromatiche, cioè i quattro valori RGBA.

La funzione che creeremo richiederà come parametro il nome del file che si desidera richiamare. Creiamo, anzitutto, delle variabili d'appoggio:

- *FileImmagine:* Puntatore ad una struttura di tipo FILE che ha scopo di supporto nel processo di apertura del file
- *file_Header:* Variabile di tipo BITMAPFILEHEADER, che ha lo scopo di contenere il File Header
- *Immagine:* Array che conterrà i dati dell'immagine. Ricordiamo che i dati di un'immagine vengono rappresentati tramite dei valori interi senza segno.
- *tmp_daBGRaRGB:* Variabile temporanea, il cui scopo spiegheremo in dettaglio nel seguito
- *Percorso:* Variabile che conterrà il percorso della directory corrente
- *PercorsoImmagine:* Variabile che conterrà il percorso finale dell'immagine, comprensivo di directory

Codice
<pre> unsigned char *CaricaImmagine(char *NomeDelFile){ FILE *FileImmagine; BITMAPFILEHEADER file_Header; unsigned char *Immagine = NULL; unsigned char tmp_daBGRaRGB; TCHAR Percorso[256]; char PercorsoImmagine[256]; </pre>

La prima cosa da fare è determinare la directory corrente, nella quale supponiamo sia l'immagine della nostra texture. Approfittiamo di ciò per ricordare che, al momento dell'esecuzione, l'immagine scelta andrà caricata nel dispositivo/emulatore nella directory dove si trova l'eseguibile del programma.

Utilizziamo la funzione **GetModuleFileName** per recuperare la directory corrente. Detta funzione restituisce il percorso della directory in cui è contenuto l'eseguibile del programma che la richiama, con in coda il nome del programma stesso.

La funzione accetta come parametri:

- Il modulo in uso. Il valore NULL indica il corrente.
- TCHAR *path*: La variabile in cui memorizzare il percorso trovato
- int *num*: Il massimo numero di caratteri del percorso che possono essere caricati

Codice
<pre> GetModuleFileName(NULL, Percorso, 256); </pre>

Il percorso a noi necessario è quello della directory corrente senza avere in coda il nome del programma. Dovremo quindi eliminarlo. Per farlo, cercheremo all'interno della stringa ottenuta il primo carattere "\" utilizzando la funzione **wcsrchr**, dopodichè basterà mettere un carattere NULL prima della posizione trovata, poichè, al momento della lettura, solo la parte precedente al NULL verrà considerata.

Codice
<pre> TCHAR *posizBarra = wcsrchr(Percorso, '\\'); *(posizBarra + 1) = '\0'; </pre>

La directory corrente ottenuta viene espressa, per la natura della funzione GetModuleFileName, in caratteri di tipo **wide**. Le funzione che utilizzeremo in seguito, invece, necessiteranno che il percorso sia espresso in caratteri di tipo **multibyte** (TCHAR e char). Procediamo quindi alla conversione tramite la funzione **wcstombs**, che prende come parametri:

- char *storeMultibyte*: Variabile in cui conservare i caratteri multibyte
- TCHAR *WideString*: La stringa di caratteri da convertire
- int *num*: Il massimo numero di caratteri che possono essere analizzati

Codice
<pre> wcstombs(PercorsoImmagine, Percorso, 256); </pre>

Infine, determiniamo il percorso finale, allegando, a quello della directory corrente, il nome del file bitmap.

Codice
<code>strcat(PercorsoImmagine, NomeDelFile);</code>

Apriamo, quindi, il file in modalità binaria.

Codice
<code>FileImmagine = fopen(PercorsoImmagine,"rb");</code>

Mettiamo il File Header nella struttura BITMAPFILEHEADER.

Codice
<code>fread(&file_Header,sizeof(BITMAPFILEHEADER),1,FileImmagine);</code>

Un file di tipo Windows Bitmap ha un ID di *0x4D42*. Questo valore è memorizzato nella variabile *bftype* della struttura BITMAPFILEHEADER. Se l'ID trovato è diverso vuol dire che il file non è un Windows Bitmap e quindi fermeremo il caricamento, mandando un messaggio d'errore. Si noti come la L prima della stringa d'errore serva, in ambiente Windows, allo scopo della conversione in caratteri wide, cioè nel tipo di caratteri utilizzati da Pocket Pc.

Codice
<pre> if (file_Header.bftype != 0x4D42) { MessageBox(NULL, L"Formato immagine non corretto: richiesto formato bmp", L"Errore", MB_OK); fclose(FileImmagine); return NULL; } </pre>

Mettiamo l'Info Header nella struttura BITMAPINFOHEADER.

Codice
<code>fread(&info_Header,sizeof(BITMAPINFOHEADER),1,FileImmagine);</code>

La variabile *bfOffBits* della struttura BITMAPINFOHEADER specifica il numero di bit dell'immagine. Grazie a questa informazione, possiamo muovere il puntatore dall'inizio del file (SEEK_SET) all'inizio dei dati dell'immagine.

Codice
<code>fseek(FileImmagine,file_Header.bfOffBits,SEEK_SET);</code>

La variabile *biSizeImage* della struttura BITMAPINFOHEADER contiene la dimensione dell'immagine. Conoscerla ci permette di allocare la memoria necessaria a contenere l'immagine.

Codice
<code>Immagine = new unsigned char[info_Header.biSizeImage];</code>

Fatto ciò, carichiamo l'immagine bit per bit.

Codice
<code>fread(Immagine,1,info_Header.biSizeImage,FileImmagine);</code>

Rimane un'osservazione. Il formato prevede la memorizzazione dell'immagine in formato BGR, anziché RGB. Per portare i dati al formato da noi voluto, dovremo quindi scambiare il primo ed il terzo valore per ogni tripletta.

Codice
<pre> for (unsigned int i = 0; i < info_Header.biSizeImage; i += 3) { tmp_daBGRaRGB = Immagine[i]; Immagine[i] = Immagine[i+2]; Immagine[i+2] = tmp_daBGRaRGB; } </pre>

Infine, chiudiamo il file e restituiamo il puntatore ai dati dell'immagine.

Codice
<pre> fclose(FileImmagine); return Immagine; } </pre>

Un'ultima, fondamentale, nota: onde permetterne il caricamento, si ricordi di caricare l'immagine scelta, con il corretto nome, nella cartella dell'emulatore/dispositivo in cui si trovi l'eseguibile di programma.

☐☐ C. Strumenti utili ☐☐

C.1 Dominio dei tipi di dato

C/C++ riconoscono i tipi mostrati nella tabella.

Nome del tipo	Byte	Altro nome	Dominio dei valori
int	*	signed, signed int	Dipendente dal sistema

unsigned int	*	unsigned	Dipendente dal sistema
__int8	1	char, signed char	-128 to 127
__int16	2	short, short int, signed short int	-32,768 to 32,767
__int32	4	signed, signed int	-2,147,483,648 to 2,147,483,647
__int64	8	nessuno	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
char	1	signed char	-128 to 127
unsigned char	1	nessuno	0 to 255
short	2	short int, signed short int	-32,768 to 32,767
unsigned short	2	unsigned short int	0 to 65,535
long	4	long int, signed long int	-2,147,483,648 to 2,147,483,647
unsigned long	4	unsigned long int	0 to 4,294,967,295
enum	*	nessuno	Come int
float	4	nessuno	3.4E +/- 38 (7 cifre)
double	8	nessuno	1.7E +/- 308 (15 cifre)
long double	10	nessuno	1.2E +/- 4932 (19 cifre)

Il tipo *long double* (80-bit, 10-byte di precisione) viene mappato direttamente nel *double* (64-bit, 8-byte di precisione) sotto Windows NT, Windows 98, and Windows 95.

Signed e *unsigned* sono modificatori che possono essere usati con qualsiasi tipo. Il tipo *char* è *signed* per default, ma specificando */J* (opzione di compilazione) lo si può far diventare *unsigned* per default.

I tipi *int* e *unsigned int* hanno la grandezza di una parola. Essa, di preciso, ha grandezza di 2 byte (come *short* e *unsigned short*) in MS-DOS e nelle versioni 16 bit di Windows, e 4 byte nei sistemi operativi a 32-bit. In ogni caso, al portabilità del codice non dovrebbe dipendere dalla grandezza del tipo.

C.2 Suffissi per funzioni a seconda del tipo degli argomenti

Molte funzioni OpenGL® si presentano in varie versioni, la cui differenza sta nel suffisso. Solitamente questo suffisso indica il tipo degli argomenti della funzione. Citiamo i più comuni nella tabella.

Suffisso	Tipo
b	Intero a 8-bit
ub	Intero senza segno a 8-bit
s	Intero a 16-bit
us	Intero senza segno a 16-bit
i	Intero a 32-bit
ui	Intero senza segno a 32-bit

f	Float a 32-bit
d	Float a 64 bit
X	Fixed point

C.3 Precedenza e associatività degli operatori

La tavola mostra gli operatori di C/C++ nonchè le proprietà di precedenza e associatività. La precedenza più alta si trova nella parte più alta della tabella.

Simbolo	Nome o significato	Associatività
	<i>Precedenza più alta</i>	
++	Post-incremento	Da sinistra a destra
--	Post-decremento	
()	Chiamata di funzione	
[]	Elemento di array	
->	Puntatore a un membro di una struttura	
.	Struttura o membro di unione	
++	Pre-incremento	Da destra a sinistra
--	Pre-decremento	
!	NOT Logico	
~	Bitwise NOT	
-	Meno unario	
+	Più unario	
&	Indirizzo	
*	Indirection*	
sizeof	Grandezza in byte	
new	Alloca memoria al programma	
delete	Dealloca memoria al programma	
(type)	Conversione di tipo (es. (float)int i)	
.*	Puntatore a membro (Oggetti)	Da sinistra a destra
->*	Puntatore a membro (Puntatori)	
*	Moltiplicazione	Da sinistra a destra
/	Divisione	
%	Resto di divisione intera	
+	Somma	Da sinistra a destra
-	Sottrazione	
<<	Shift sinistro	Da sinistra a destra
>>	Shift destro	

<	Minore	Da sinistra a destra
<=	Minore o uguale	
>	Maggiore	
>=	Maggiore o uguale	
==	Uguale	Da sinistra a destra
!=	Diverso	
&	Bitwise AND	Da sinistra a destra
^	Bitwise esclusivo OR	Da sinistra a destra
	Bitwise OR	Da sinistra a destra
&&	AND logico	Da sinistra a destra
	OR logico	Da sinistra a destra
? :	Condizione if then	Da sinistra a destra
=	Assegnamento	Da sinistra a destra
*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =	Assegnamento composto	
,	Virgola	Da sinistra a destra
	Precedenza più bassa	

* Per "indirection" si intende la capacità di riferirsi ad un oggetto utilizzando un nome, un contenitore o altro, piuttosto che il valore dell'oggetto stesso. L'esempio più semplice ne è l'atto di manipolare un valore attraverso il suo indirizzo in memoria, come avviene nel caso si acceda ad una variabile attraverso il suo puntatore.

C.4 Sequenze di escape

Le sequenze di *escape* hanno lo scopo di permettere l'uso di caratteri speciali. Le riportiamo per intero.

Sequenza	Nome
\a	Segnale sonoro di allarme
\b	Spazio
\f	Formfeed*
\n	Nuova linea
\r	Carriage return**
\t	Tabulazione orizzontale
\v	Tabulazione verticale
\?	Punto interrogativo letterale
\'	Virgoletta singola
\"	Virgoletta doppia
\\	Backslash
\ddd	Carattere ASCII in notazione ottale

\xdd	Carattere ASCII in notazione esadecimale
\0	Carattere NULL

* Cambio di pagina: Segnale per la stampante di smettere la stampa sul foglio attuale e ricominciare dal foglio seguente

** Ritorno di carrello: Posiziona il puntatore di scrittura all'inizio della linea seguente.

Nella rappresentazione esadecimale dei caratteri gli zero iniziali vengono ignorati dal compilatore. Esso stabilisce la fine del carattere di *escape* esadecimale quando incontra o il primo carattere non esadecimale o quando incontra più di due caratteri esadecimali, non inclusi gli zero iniziali. Nel secondo caso, riporta un errore e ignora tutti i caratteri dopo il secondo.

C.5 Codici Ascii

La tavola dei codici ASCII (*American Standards Committee for Information Interchange*) contiene i valori decimali e esadecimali dell'insieme esteso dei caratteri ASCII. Tale insieme contiene l'insieme dei caratteri ASCII e 128 caratteri per la grafica e il disegno di linee, spesso chiamati "Insieme dei caratteri IBM® "

Ctrl	Dec	Hex	Char	Code	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
^@	0	00		NUL	32	20	sp	64	40	@	96	60	`
^A	1	01	␣	SOH	33	21	!	65	41	A	97	61	a
^B	2	02	␢	SIX	34	22	"	66	42	B	98	62	b
^C	3	03	♥	EIX	35	23	#	67	43	C	99	63	c
^D	4	04	♦	EOI	36	24	\$	68	44	D	100	64	d
^E	5	05	♣	ENQ	37	25	%	69	45	E	101	65	e
^F	6	06	♠	ACK	38	26	&	70	46	F	102	66	f
^G	7	07	•	BEL	39	27	'	71	47	G	103	67	g
^H	8	08	◼	BS	40	28	(72	48	H	104	68	h
^I	9	09	○	HI	41	29)	73	49	I	105	69	i
^J	10	0A	⓪	LF	42	2A	*	74	4A	J	106	6A	j
^K	11	0B	♂	VI	43	2B	+	75	4B	K	107	6B	k
^L	12	0C	♀	FF	44	2C	,	76	4C	L	108	6C	l
^M	13	0D	℣	CR	45	2D	-	77	4D	M	109	6D	m
^N	14	0E	℣	SO	46	2E	.	78	4E	N	110	6E	n
^O	15	0F	※	SI	47	2F	/	79	4F	O	111	6F	o
^P	16	10	▶	SLE	48	30	0	80	50	P	112	70	p
^Q	17	11	◀	CS1	49	31	1	81	51	Q	113	71	q
^R	18	12	↕	DC2	50	32	2	82	52	R	114	72	r
^S	19	13	!!	DC3	51	33	3	83	53	S	115	73	s
^T	20	14	¶	DC4	52	34	4	84	54	T	116	74	t
^U	21	15	§	NAK	53	35	5	85	55	U	117	75	u
^V	22	16	▬	SYN	54	36	6	86	56	V	118	76	v
^W	23	17	‡	EIB	55	37	7	87	57	W	119	77	w
^X	24	18	↑	CAN	56	38	8	88	58	X	120	78	x
^Y	25	19	↓	EM	57	39	9	89	59	Y	121	79	y
^Z	26	1A	→	SIB	58	3A	:	90	5A	Z	122	7A	z
^[27	1B	←	ESC	59	3B	;	91	5B	[123	7B	{
^\	28	1C	└	FS	60	3C	<	92	5C	\	124	7C	
^]	29	1D	⦿	GS	61	3D	=	93	5D]	125	7D	}
^^	30	1E	▲	RS	62	3E	>	94	5E	^	126	7E	~
^_	31	1F	▼	US	63	3F	?	95	5F	_	127	7F	Δ [†]

† ASCII code 127 has the code DEL. Under MS-DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL+BKSP key.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Œ	160	A0	Š	192	C0	Ł	224	E0	Ɔ
129	81	Ü	161	A1	Í	193	C1	Ł	225	E1	Ɔ
130	82	é	162	A2	ó	194	C2	Ł	226	E2	Ɔ
131	83	â	163	A3	ú	195	C3	Ł	227	E3	Ɔ
132	84	ä	164	A4	ñ	196	C4	Ł	228	E4	Ɔ
133	85	ä	165	A5	Ñ	197	C5	Ł	229	E5	Ɔ
134	86	ä	166	A6	ä	198	C6	Ł	230	E6	Ɔ
135	87	g	167	A7	g	199	C7	Ł	231	E7	Ɔ
136	88	è	168	A8	è	200	C8	Ł	232	E8	Ɔ
137	89	è	169	A9	è	201	C9	Ł	233	E9	Ɔ
138	8A	è	170	AA	è	202	CA	Ł	234	EA	Ɔ
139	8B	ï	171	AB	½	203	CB	Ł	235	EB	Ɔ
140	8C	î	172	AC	¼	204	CC	Ł	236	EC	Ɔ
141	8D	ì	173	AD	ì	205	CD	Ł	237	ED	Ɔ
142	8E	ï	174	AE	«	206	CE	Ł	238	EE	Ɔ
143	8F	ä	175	AF	»	207	CF	Ł	239	EF	Ɔ
144	90	é	176	B0	⌘	208	D0	Ł	240	F0	Ɔ
145	91	æ	177	B1	⌘	209	D1	Ł	241	F1	Ɔ
146	92	æ	178	B2	⌘	210	D2	Ł	242	F2	Ɔ
147	93	ô	179	B3	⌘	211	D3	Ł	243	F3	Ɔ
148	94	ô	180	B4	⌘	212	D4	Ł	244	F4	Ɔ
149	95	ô	181	B5	⌘	213	D5	Ł	245	F5	Ɔ
150	96	û	182	B6	⌘	214	D6	Ł	246	F6	Ɔ
151	97	û	183	B7	⌘	215	D7	Ł	247	F7	Ɔ
152	98	ü	184	B8	⌘	216	D8	Ł	248	F8	Ɔ
153	99	ö	185	B9	⌘	217	D9	Ł	249	F9	Ɔ
154	9A	ü	186	BA	⌘	218	DA	Ł	250	FA	Ɔ
155	9B	ç	187	BB	⌘	219	DB	Ł	251	FB	Ɔ
156	9C	ç	188	BC	⌘	220	DC	Ł	252	FC	Ɔ
157	9D	¥	189	BD	⌘	221	DD	Ł	253	FD	Ɔ
158	9E	Ŕ	190	BE	⌘	222	DE	Ł	254	FE	Ɔ
159	9F	f	191	BF	⌘	223	DF	Ł	255	FF	Ɔ

C.6 Codici ASCII multilingua

Ci sono alcune varianti dell'insieme di caratteri, chiamate *"code pages"*. I sistemi venduti in alcuni stati europei usano l'insieme dei caratteri multilingua conosciuto come Code Page 850, che contiene meno simboli grafici e più lettere accentate e caratteri speciali. Per completezza lo riportiamo.

0	32	64	96	128	160	192	224
1 ☐	33 !	65 A	97 a	129 ü	161 í	193 ı	225 ß
2 ☐	34 "	66 B	98 b	130 é	162 ó	194 T	226 ô
3 ♥	35 #	67 C	99 c	131 â	163 ú	195 †	227 ò
4 ♦	36 \$	68 D	100 d	132 ä	164 ñ	196 —	228 õ
5 ♣	37 %	69 E	101 e	133 à	165 ñ	197 †	229 õ
6 ♠	38 &	70 F	102 f	134 ã	166 º	198 ã	230 µ
7 •	39 ’	71 G	103 g	135 ç	167 º	199 ã	231 þ
8 ■	40 (72 H	104 h	136 ê	168 ğ	200 ı	232 þ
9 ◊	41)	73 I	105 i	137 ë	169 ☐	201 ĩ	233 ú
10 ◊	42 *	74 J	106 j	138 è	170 ı	202 ı	234 û
11 ♂	43 +	75 K	107 k	139 ï	171 ½	203 ĩ	235 ù
12 ♀	44 ,	76 L	108 l	140 î	172 ¼	204 ĩ	236 ú
13 ♪	45 _	77 M	109 m	141 ì	173 ì	205 =	237 ý
14 ♪	46 .	78 N	110 n	142 Ä	174 «	206 ĩ	238 -
15 ✱	47 /	79 O	111 o	143 Å	175 »	207 ☐	239 ´
16 ►	48 0	80 P	112 p	144 É	176 ☐	208 ð	240 -
17 ◄	49 1	81 Q	113 q	145 æ	177 ☐	209 ð	241 ±
18 ⇕	50 2	82 R	114 r	146 ff	178 ☐	210 Ê	242 =
19 !!	51 3	83 S	115 s	147 ô	179	211 Ê	243 ¾
20 ¶	52 4	84 T	116 t	148 ö	180 †	212 È	244 ¶
21 ☐	53 5	85 U	117 u	149 ò	181 Á	213 ´	245 ☐
22 —	54 6	86 V	118 v	150 û	182 Â	214 Í	246 ÷
23 ⇕	55 7	87 W	119 w	151 ù	183 À	215 Î	247 ~
24 ↑	56 8	88 X	120 x	152 Ü	184 ☐	216 Ï	248 °
25 ↓	57 9	89 Y	121 y	153 ö	185 ĩ	217 J	249 ..
26 →	58 :	90 Z	122 z	154 Ü	186	218 Γ	250 ´
27 ←	59 ;	91 [123 {	155 ø	187 ĩ	219 ■	251 ´
28 ⊥	60 <	92 \	124 !	156 f	188 ı	220 ■	252 ³
29 ⇕	61 =	93]	125 }	157 Ø	189 Ç	221 i	253 ²
30 ▲	62 >	94 ^	126 ~	158 ×	190 ¥	222 ì	254 ■
31 ▼	63 ?	95 _	127 Δ	159 f	191 ı	223 ■	255

C.7 Codici di riconoscimento dei tasti

Costante simbolica	Valore esadecimale	Equivalente su mouse o tastiera
VK_LBUTTON	01	Tasto sinistro del mouse
VK_RBUTTON	02	Tasto destro del mouse
VK_CANCEL	03	Control-break processing
VK_MBUTTON	04	Tasto di centro del mouse
—	05–07	Non definito

VK_BACK	08	Tasto BACKSPACE
VK_TAB	09	Tasto TAB
—	0A–0B	Non definito
VK_CLEAR	0C	Tasto CLEAR
VK_RETURN	0D	Tasto ENTER
—	0E–0F	Non definito
VK_SHIFT	10	Tasto SHIFT
VK_CONTROL	11	Tasto CTRL
VK_MENU	12	Tasto ALT
VK_PAUSE	13	Tasto PAUSE
VK_CAPITAL	14	Tasto CAPS LOCK
—	15–19	Riservato ai sistemi che utilizzano i Kanji
—	1A	Non definito
VK_ESCAPE	1B	Tasto ESC
—	1C–1F	Riservato ai sistemi che utilizzano i Kanji
VK_SPACE	20	Tasto SPACEBAR
VK_PRIOR	21	Tasto PAGE UP
VK_NEXT	22	Tasto PAGE DOWN
VK_END	23	Tasto END
VK_HOME	24	Tasto HOME
VK_LEFT	25	Tasto LEFT ARROW
VK_UP	26	Tasto UP ARROW
VK_RIGHT	27	Tasto RIGHT ARROW
VK_DOWN	28	Tasto DOWN ARROW
VK_SELECT	29	Tasto SELECT
	2A	Specifico dell'OEM
VK_EXECUTE	2B	Tasto EXECUTE
VK_SNAPSHOT	2C	Tasto PRINT SCREEN
VK_INSERT	2D	Tasto INS
VK_DELETE	2E	Tasto DEL
VK_HELP	2F	Tasto HELP
	3A–40	Non definito
VK_LWIN	5B	Tasto sinistro di Window su una Microsoft Natural Keyboard
VK_RWIN	5C	Tasto destro di Window su una Microsoft Natural Keyboard
VK_APPS	5D	Tasto applicazione di Window su una Microsoft Natural Keyboard
	5E–5F	Non definito
VK_NUMPAD0	60	Tasto 0 sul tastierino numerico

VK_NUMPAD1	61	Tasto 1 sul tastierino numerico
VK_NUMPAD2	62	Tasto 2 sul tastierino numerico
VK_NUMPAD3	63	Tasto 3 sul tastierino numerico
VK_NUMPAD4	64	Tasto 4 sul tastierino numerico
VK_NUMPAD5	65	Tasto 5 sul tastierino numerico
VK_NUMPAD6	66	Tasto 6 sul tastierino numerico
VK_NUMPAD7	67	Tasto 7 sul tastierino numerico
VK_NUMPAD8	68	Tasto 8 sul tastierino numerico
VK_NUMPAD9	69	Tasto 9 sul tastierino numerico
VK_MULTIPLY	6A	Tasto Moltiplicazione
VK_ADD	6B	Tasto Somma
VK_SEPARATOR	6C	Tasto Separatore
VK_SUBTRACT	6D	Tasto Sottrazione
VK_DECIMAL	6E	Tasto Decimale
VK_DIVIDE	6F	Tasto Divisione
VK_F1	70	Tasto F1
VK_F2	71	Tasto F2
VK_F3	72	Tasto F3
VK_F4	73	Tasto F4
VK_F5	74	Tasto F5
VK_F6	75	Tasto F6
VK_F7	76	Tasto F7
VK_F8	77	Tasto F8
VK_F9	78	Tasto F9
VK_F10	79	Tasto F10
VK_F11	7A	Tasto F11
VK_F12	7B	Tasto F12
VK_F13	7C	Tasto F13
VK_F14	7D	Tasto F14
VK_F15	7E	Tasto F15
VK_F16	7F	Tasto F16
VK_F17	80H	Tasto F17
VK_F18	81H	Tasto F18
VK_F19	82H	Tasto F19
VK_F20	83H	Tasto F20
VK_F21	84H	Tasto F21
VK_F22	85H	Tasto F22
VK_F23	86H	Tasto F23
VK_F24	87H	Tasto F24

	88–8F	Non assegnato
VK_NUMLOCK	90	Tasto NUM LOCK
VK_SCROLL	91	Tasto SCROLL LOCK
VK_LSHIFT	0xA0	Tasto SHIFT sinistro
VK_RSHIFT	0xA1	Tasto SHIFT destro
VK_LCONTROL	0xA2	Tasto CTRL sinistro
VK_RCONTROL	0xA3	Tasto CTRL destro
VK_LMENU	0xA4	Tasto ALT sinistro
VK_RMENU	0xA5	Tasto ALT destro
	BA–C0	Specifico dell'OEM; riservato. Vedi tabelle seguenti.
	C1–DA	Non assegnato
	DB–E2	Specifico dell'OEM; riservato. Vedi tabelle seguenti.
	E3 – E4	Specifico dell'OEM
	E5	Non assegnato
	E6	Specifico dell'OEM
	E7–E8	Non assegnato
	E9–F5	Specifico dell'OEM
VK_ATTN	F6	Tasto ATTN
VK_CRSEL	F7	Tasto CRSEL
VK_EXSEL	F8	Tasto EXSEL
VK_EREOF	F9	Tasto Erase EOF
VK_PLAY	FA	Tasto PLAY
VK_ZOOM	FB	Tasto ZOOM
VK_NONAME	FC	Riservato per usi futuri
VK_PA1	FD	Tasto PA1
VK_OEM_CLEAR	FE	Tasto CLEAR

L'OEM dovrebbe avere note speciali a proposito dei tasti a uso OEM: 2A, DB–E4, E6, and E9–F5. In aggiunta agli assegnamenti dei tasti VK nella tabella precedente. Microsoft ha assegnato i seguenti tasti OEM.

Costante simbolica	Valore Esadecimale	Equivalente su mouse o tastiera
VK_OEM_SCROLL	0x91	Nessuno
VK_OEM_1	0xBA	" ; : " per gli Stati Uniti
VK_OEM_PLUS	0xBB	" + " per ogni stato/regione
VK_OEM_COMMA	0xBC	" , " per ogni stato/regione
VK_OEM_MINUS	0xBD	" - " per ogni stato/regione
VK_OEM_PERIOD	0xBE	" . " per ogni stato/regione
VK_OEM_2	0xBF	" / ? " per gli Stati Uniti
VK_OEM_3	0xC0	" ` ~ " per gli Stati Uniti

VK_OEM_4	0xDB	"[" per gli Stati Uniti
VK_OEM_5	0xDC	"\" per gli Stati Uniti
VK_OEM_6	0xDD	"]]" per gli Stati Uniti
VK_OEM_7	0xDE	"'" per gli Stati Uniti
VK_OEM_8	0xDF	Nessuno
VK_OEM_AX	0xE1	Tasto AX sulle tastiere AX giapponesi
VK_OEM_102	0xE2	"<>" or "\ " sulle tastiere RT a 102 tasti

Per gli IME (*East Asian Input Method Editors*) sono rispettate anche le seguenti convenzioni.

Costante simbolica	Valore Esadecimale	Descrizione
VK_DBE_ALPHA NUMERIC	0x0f0	Cambia la modalità ad alfanumerica
VK_DBE_KATAKANA	0x0f1	Cambia la modalità a Katakana
VK_DBE_HIRAGANA	0x0f2	Cambia la modalità a Hiragana
VK_DBE_SBCSC HAR	0x0f3	Cambia la modalità a caratteri a byte singolo
VK_DBE_DBCSC HAR	0x0f4	Cambia la modalità a caratteri a byte doppio
VK_DBE_ROMAN	0x0f5	Cambia la modalità a caratteri romani
VK_DBE_NOROMAN	0x0f6	Cambia la modalità a caratteri non romani
VK_DBE_ENTER WORDREGISTER MODE	0x0f7	Attiva la finestra di dialogo Word Register
VK_DBE_ENTER IMECONFIGMODE	0x0f8	Attiva la finestra di dialogo per il settaggio di un ambiente IME
VK_DBE_FLUSH STRING	0x0f9	Cancella le stringhe non determinate senza determinarle
VK_DBE_CODE INPUT	0x0fa	Cambia la modalità a input di codice
VK_DBE_NOCODE INPUT	0x0fb	Cambia la modalità a input non di codice

Gli OEM non dovrebbero utilizzare porzioni non assegnate delle tavole dei codici poichè saranno sicuramente assegnate in futuro. In caso di necessità, è più consigliabile riutilizzare i codici già esistenti sovrascrivendoli.

C.8 Codici di tastiera

Alcuni tasti, come i tasti funzione F1..F12, i tasti cursore e le combinazioni ALT+tasto non hanno un proprio codice ASCII. Quando un tasto viene premuto, un microprocessore interno alla tastiera genera un "codice di tastiera esteso" di 2 byte. Il primo (meno significativo) byte contiene il codice ASCII. Il secondo (più significativo) contiene un "codice di tastiera", un codice unico generato dalla tastiera quando un tasto viene premuto o rilasciato. Poichè la

tavola estesa dei codici di tastiera è più vasta della tavola dei codici ASCII, i programmi possono utilizzarla per identificare tasti che non possiedono un proprio codice ASCII.

Key	Scan Code		ASCII or Extended			ASCII or Extended with SHIFT			ASCII or Extended with CTRL			ASCII or Extended with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
ESC	1	01	27	1B	ESC	27	1B	ESC	27	1B	ESC	1	01	NUL §
1!	2	02	49	31	1	33	21	!				120	78	NUL
2@	3	03	50	32	2	64	40	@	3	03	NUL	121	79	NUL
3#	4	04	51	33	3	35	23	#				122	7A	NUL
4\$	5	05	52	34	4	36	24	\$				123	7B	NUL
5%	6	06	53	35	5	37	25	%				124	7C	NUL
6^	7	07	54	36	6	94	5E	^	30	1E	RS	125	7D	NUL
7&	8	08	55	37	7	38	26	&				126	7E	NUL
8*	9	09	56	38	8	42	2A	*				127	7F	NUL
9(10	0A	57	39	9	40	28	(128	80	NUL
0)	11	0B	48	30	0	41	29)				129	81	NUL
-_	12	0C	45	2D	-	95	5F	_	31	1F	US	130	82	NUL
=+	13	0D	61	3D	=	43	2B	+				131	83	NUL
EKSP	14	0E	8	08		8	08		127	7F		14	0E	NUL §
IAB	15	0F	9	09		15	0F	NUL	148	94	NUL §	15	A5	NUL §
Q	16	10	113	71	q	81	51	Q	17	11	DC1	16	10	NUL
W	17	11	119	77	w	87	57	W	23	17	E1B	17	11	NUL
E	18	12	101	65	e	69	45	E	5	05	ENQ	18	12	NUL
R	19	13	114	72	r	82	52	R	18	12	DC2	19	13	NUL
I	20	14	116	74	i	84	54	I	20	14	SO	20	14	NUL
Y	21	15	121	79	y	89	59	Y	25	19	EM	21	15	NUL
U	22	16	117	75	u	85	55	U	21	15	NAK	22	16	NUL
I	23	17	105	69	i	73	49	I	9	09	IAB	23	17	NUL
O	24	18	111	6F	o	79	4F	O	15	0F	SI	24	18	NUL
P	25	19	112	70	p	80	50	P	16	10	DLE	25	19	NUL
[{	26	1A	91	5B	[123	7B	{	27	1B	ESC	26	1A	NUL §
]}	27	1B	93	5D]	125	7D	}	29	1D	GS	27	1B	NUL §
ENTER	28	1C	13	0D	CR	13	0D	CR	10	0A	LF	28	1C	NUL §
ENTER	28	1C	13	0D	CR	13	0D	CR	10	0A	LF	166	A6	NUL §
LC IRL	29	1D												
RC IRL	29	1D												
A	30	1E	97	61	a	65	41	A	1	01	SOH	30	1E	NUL
S	31	1F	115	73	s	83	53	S	19	13	DC3	31	1F	NUL
D	32	20	100	64	d	68	44	D	4	04	EOI	32	20	NUL
F	33	21	102	66	f	70	46	F	6	06	ACK	33	21	NUL
G	34	22	103	67	g	71	47	G	7	07	BEL	34	22	NUL
H	35	23	104	68	h	72	48	H	8	08	BS	35	23	NUL
J	36	24	106	6A	j	74	4A	J	10	0A	LF	36	24	NUL
K	37	25	107	6B	k	75	4B	K	11	0B	VI	37	25	NUL
L	38	26	108	6C	l	76	4C	L	12	0C	FF	38	26	NUL
::	39	27	59	3B	:	58	3A	:				39	27	NUL §
""	40	28	39	27	"	34	22	"				40	28	NUL §
~	41	29	96	60	~	126	7E	~				41	29	NUL §
L SHIFT	42	2A												
\	43	2B	92	5C	\	124	7C	 	28	1C	FS			
Z	44	2C	122	7A	z	90	5A	Z	26	1A	SUB	44	2C	NUL
X	45	2D	120	78	x	88	58	X	24	18	CAN	45	2D	NUL
C	46	2E	99	63	c	67	43	C	3	03	EIX	46	2E	NUL
V	47	2F	118	76	v	86	56	V	22	16	SYN	47	2F	NUL
B	48	30	98	62	b	66	42	B	2	02	SIX	48	30	NUL
N	49	31	110	6E	n	78	4E	N	14	0E	SO	49	31	NUL
M	50	32	109	6D	m	77	4D	M	13	0D	CR	50	32	NUL
,<	51	33	44	2C	,	60	3C	<				51	33	NUL §
.>	52	34	46	2E	.	62	3E	>				52	34	NUL §

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
/?	53	35	47	2F	/	63	3F	?				53	34	NUL§
GRAY #	53	35	47	2F	/	63	3F	?	149	95	NUL	164	A5	NUL
R SHIF	54	36												
*PRISC	55	37	42	2A	*	PRISC		↑↑	16	10				
L ALI	56	38												
R ALI#	56	38												
SPACE	57	39	32	20	SPC	32	20	SPC	32	20	SPC	32	20	SPC
CAPS	58	3A												
F1	59	3B	59	3B	NUL	84	54	NUL	94	5E	NUL	104	68	NUL
F2	60	3C	60	3C	NUL	85	55	NUL	95	5F	NUL	105	69	NUL
F3	61	3D	61	3D	NUL	86	56	NUL	96	60	NUL	106	6A	NUL
F4	62	3E	62	3E	NUL	87	57	NUL	97	61	NUL	107	6B	NUL
F5	63	3F	63	3F	NUL	88	58	NUL	98	62	NUL	108	6C	NUL
F6	64	40	64	40	NUL	89	59	NUL	99	63	NUL	109	6D	NUL
F7	65	41	65	41	NUL	90	5A	NUL	100	64	NUL	110	6E	NUL
F8	66	42	66	42	NUL	91	5B	NUL	101	65	NUL	111	6F	NUL
F9	67	43	67	43	NUL	92	5C	NUL	102	66	NUL	112	70	NUL
F10	68	44	68	44	NUL	93	5D	NUL	103	67	NUL	113	71	NUL
F11#	87	57	133	85	E0	135	87	E0	137	89	E0	139	8B	E0
F12#	88	58	134	86	E0	136	88	E0	138	8A	E0	140	8C	E0
NUM	69	45												
SCROLL	70	46												
HOME	71	47	71	47	NUL	55	37	7	119	77	NUL			
HOME#	71	47	71	47	E0	71	47	E0	119	77	E0	151	97	NUL
UP	72	48	72	48	NUL	56	38	8	141	8D	NUL§			
UP#	72	48	72	48	E0	72	48	E0	141	8D	E0	152	98	NUL
PGUP	73	49	73	49	NUL	57	39	9	132	84	NUL			
PGUP#	73	49	73	49	E0	73	49	E0	132	84	E0	153	99	NUL
GRAY-	74	4A				45	2D	-						
LEF	75	4B	75	4B	NUL	52	34	4	115	73	NUL			
LEF#	75	4B	75	4B	E0	75	4B	E0	115	73	E0	155	9B	NUL
CENIER	76	4C				53	35	5						
RIGH	77	4D	77	4D	NUL	54	36	6	116	74	NUL			
RIGH#	77	4D	77	4D	E0	77	4D	E0	116	74	E0	157	9D	NUL
GRAY+	78	4E				43	2B	+						
END	79	4F	79	4F	NUL	49	31	1	117	75	NUL			
END#	79	4F	79	4F	E0	79	4F	E0	117	75	E0	159	9F	NUL
DOWN	80	50	80	50	NUL	50	32	2	145	91	NUL§			
DOWN#	80	50	80	50	E0	80	50	E0	145	91	E0	160	A0	NUL
PGDN	81	51	81	51	NUL	51	33	3	118	76	NUL			
PGDN#	81	51	81	51	E0	81	51	E0	118	76	E0	161	A1	NUL
INS	82	52	82	52	NUL	48	30	0	146	92	NUL§			
INS#	82	52	82	52	E0	82	52	E0	146	92	E0	162	A2	NUL
DEL	83	53	83	53	NUL	46	2E	.	147	93	NUL§			
DEL#	83	53	83	53	E0	83	53	E0	147	93	E0	163	A3	NUL

† Extended codes return 0 (NUL) or E0 (decimal 224) as the initial character. This is a signal that a second (extended) code is available in the keystroke buffer.

§ These key combinations are only recognized on extended keyboards.

£ These keys are only available on extended keyboards. Most are in the Cursor/Control cluster. If the raw scan code is read from the keyboard port (60h), it appears as two bytes (E0h) followed by the normal scan code. However, when the keypad ENTER and / keys are read through the BIOS interrupt 16h, only E0h is seen since the interrupt only gives one-byte scan codes.

†† Under MS-DOS, SHIFT + PRINT SC causes interrupt 5, which prints the screen unless an interrupt handler has been defined to replace the default interrupt 5 handler.

Ringraziamenti

E' più difficile di quanto possa sembrare stendere una pagina di ringraziamenti. Nella mia vita, come spero in quella di ognuno di voi, ci sono state moltissime persone che hanno fatto tanto per me, senza le quali non sarei mai arrivata dove sono. Non ve ne abbiate a male se vedrete il vostro nome prima o dopo altri, la matematica vuole che ci sia un primo e un secondo, ma nel mio cuore siete tutti al primo posto!

Ringrazio mio zio Gianni: il Basic, il C scritto a mano sullo sfondo blu a righe bianche dell'edit del Dos, la pallina che rimbalzava su pareti che erano linee bianche e che ti avevo costretto a trasformare in un rudimentale ping pong, la fisica di base, le lezioni di matematica, risolvere mille equazioni.. Senza di te non avrei seguito la via dell'informatica, ma soprattutto non avrei mai imparato la curiosità che vuole arrivare all'inizio ed al motivo di tutte le cose, fino al più piccolo perchè.

Ringrazio mio padre che mi regalò il mio primo Olivetti per spronarmi, come altre mille volte ha fatto nella mia vita.

Ringrazio mia madre e le sue piccole attenzioni che dai per scontato fino a quando riesci a capire quanto siano belle ed importanti quelle piccole cose: ricordarti l'ombrello dimenticato o chiederti un giorno dopo come va un mal di testa di cui ti eri dimenticata tu stessa.

Insomma, ringrazio tutta la mia famiglia! Vi cito tutti e non se ne parla più: zia Sabina e le sue lasagne, zia Fifi, zio Giocchino, zia Antonella, zio Marino, zia Laura, zia Tosca, zio Ennio, zio Massimo, zio Marco, zia Patrizia, zio Tonino, i miei nonni Bettina e Antonio, Gemma e Giovanni, i miei cugini Gianluca, Cristian, Massimo, Antonio, Fabrizio, Valerio e i più piccoli Francesco e Marina.

Ringrazio i miei amici soprattutto per ciò che ha salvato la mia sanità mentale in questi sette anni di università: la loro presenza e le partite a D&D! Diego "Donovan Dhrazun" Fonseca, Valeria "Gellront" Montano, Sara "Airin" Baiocchi, Danilo "Kuff" Guidi, Stella "Non giocherò mai e poi mai" Di Credico, David "Shimmallister" Condemi, Daniele "Frollo" De Lorenzo, Danilo "Deepswin" Rizzo, Macro, Riccardo, Davide Longo. Anche i non giocatori naturalmente: Nicodemo Bonofiglio, Daniele Di Mico e Sergio Di Mico, e tutti gli infiniti compagni di università: Massimo "Lucifero", Pietro, Ciccio, Massimo "Gorecki", Alessandro, Andrei, Antonello e Riccardo e il loro sito di cartoline, Angelo, Beef, Luca "Virtuapsicologia" Angeletti, Manuel..siete troppi per citarvi tutti!

E se la via giusta è il "giusto mezzo" è doveroso citare anche quelle persone con cui non sono più in contatto, ma che hanno avuto troppa parte nei miei 27 anni per non ricordarli in questa sede, ringrazio anche loro: Roberta Botticelli, Daniele Cruciani, Marco Massara, Emiliano Di Girolamo.

Ringrazio i miei professori, dalle elementari all'università, dalla dolce Suor Lia al mitico Rogora con i suoi questionari d'analisi a risposta multipla, per avermi trasmesso l'amore per la cultura, la parola scritta e i numeri.

Primi tra tutti il mio relatore Prof.Luigi Cinque, il suo assistente Dott.Alessio Malizia e il Prof.Stefano Levialdi, che mi hanno condotto per mano attraverso questo lavoro di tesi.

Ringrazio i miei cinque "esperti", che, investiti del ruolo ufficiale di valutatore, con grandissima pazienza e disponibilità, si sono presi la briga di leggere la mia tesi e rispondere al mio questionario: Simone Rozzi, Daniele Marini, Marco Schaerf, Antonio Monteleone, Angelo Moriconi.

Un sincero grazie!

Bibliografia

- [1] Donald Hearn M. Pauline Baker "Computer graphics in OpenGL®" 3rd Edition, Pearson Prentice Hall 2004
- [2] OpenGL® Architecture Review Board Editor: Dave Shreiner, "OpenGL® Reference Manual Fourth Edition" anche conosciuto come "Bluebook", Addison Wesley 1992
- [3] OpenGL® Architecture Review Board Editor: Dave Shreiner - Mason Woo - Jackie Neder - Tom Davis, "OpenGL® Programming Guide Fourth Edition" anche conosciuto come "Redbook", Addison Wesley 1992
- [4] OpenGL® ES 1.0 Reference Manual Version 1.0, Editor: Claude Knaus, December 19, 2003 http://www.khronos.org/opengles/documentation/opengles1_0/opengles_1_0_manual.pdf
- [5] OpenGL® ES Native Platform Graphics Interface (EGL Version 1.1.02), Editor: Jon Leech, November 10, 2004
- [6] Mark Segal - Kurt Akeley, The OpenGL® Graphic System: A specification (Version 1.3), 2001 <http://www.opengl.org/documentation/specs/version1.3/glspec13.pdf>
- [7] Mark Segal - Kurt Akeley, The OpenGL® Graphic System: A specification (Version 1.5), 2003 <http://www.opengl.org/documentation/specs/version1.5/glspec15.pdf>
- [9] OpenGL® ES Common/Common-Lite Profile Specification Version 1.0.02 (Annotated), Editor: David Blythe http://www.khronos.org/cgi-bin/fetch/fetch.cgi?opengles_spec_1_0
- [10] OpenGL® ES Common/Common-Lite Profile Specification Version 1.0.07 (Annotated), Editor: David Blythe - Aaftab Munshi http://www.khronos.org/cgi-bin/fetch/fetch.cgi?opengles_spec_1_1
- [11] OpenGL® ES Common Profile Specification 2.0 Version 1.06 (Annotated), Editor: Aaftab Munshi http://www.khronos.org/cgi-bin/fetch/fetch.cgi?opengles_spec_2_0
- [12] OpenGL® ES 1.1 Reference Manual, http://www.khronos.org/opengles/documentation/opengles1_1/gl_egl_ref_1_1_20041110/index.html
- [13] OpenGL® ES Safety Critical Profile Specification Version 1.0 (Annotated), Editor Chris Hall - Claude Knaus http://www.khronos.org/cgi-bin/fetch/fetch.cgi?opengles_sc_spec_1_0
- [14] OpenGL® ES Safety Critical Philosophy, Editor : Claude Knaus, 2004 http://www.khronos.org/opengles/sc/documentation/es_sc_philosophy.pdf
- [15] Dave Astle - Dave Durnil, "OpenGL® ES Game Development" First edition, Course Technology PTR 2004
- [16] Tom McReynolds - Dave Blythe, "Advanced Graphics Programming Using OpenGL®", Morgan Kaufmann 2005
- [17] Sito dei famosi tutorial su OpenGL® prodotti dal Nehe: <http://nehe.gamedev.net/>
- [18] Wikipedia l'enciclopedia mondiale: <http://www.wikipedia.org/>
- [19] Sito del gruppo Khronos: <http://www.khronos.org/>
- [18] Sito del progetto Vincent: <http://ogl-es.sourceforge.net/index.htm>
- [19] Sito del gruppo Hybrid: <http://www.hybrid.fi/main/esframework/index.php>
- [20] Sito sulle realtà aumentate: http://studierstube.org/handheld_ar/
- [21] Alan Dix Janet Finlay Gregory Abowd Russel Beale "Human-Computer Interaction" , Pearson Prentice Hall 1993 www.hiraeth.com/books/hci
- [22] H. Rex Hartson, D. Hix, "Advances in Human-Computer Interaction", edits. Ablex Pub. corp., Norwood, New Jersey, USA, 1992
- [23] Ben Shneiderman, "Designing the User Interface" 3rd Edition, Addison-Wesley, 1999
- [24] Jef Raskin, Human Interfaces, Addison-Wesley, 2004
- [25] Alan Dix - Janet Finlay - Gregory Abowd - Russell Beale, Interazione Uomo-Macchina, McGraw-Hill, Milano, 2004
- [26] Jakob Nielsen, "Designing Web Usability", MacMillan Computer Publishing, 2000 <http://www.aw.com/DTUI>
- [27] Jacob Nielsen, "Usability Engineering", Academic Press, London, 1993 <http://www.useit.com>
- [28] Sito bibliografico totale sulla Interazione Uomo-Macchina <http://www.hcibib.org/>
- [29] Jakob Nielsen, "Ten usability Heuristics", articolo reperibile in http://www.useit.com/papers/heuristics/heuristic_list.html
- [30] Molich & Nielsen, "Improving a human-computer dialogue, Communications", 1990

- [31] Nielsen & Molich, "Heuristic evaluation of user interfaces", Proc. ACM CHI'90 Conf. (Seattle, WA, 1-5 April)
- [32] Nielsen, "Enhancing the explanatory power of usability heuristics" Proc. ACM CHI'94 Conf. (Boston, MA, April 24-28)
- [33] Nielsen, "Heuristic evaluation". In Nielsen, J., and Mack, R.L. (Eds.), Usability Inspection Methods, John Wiley & Sons, New York, NY.