

Dev.

La rivista che ti insegna a programmare

DATABASE

Note di Data Warehouse:
Le Dimensioni di analisi

JAVA

Nome in codice bshd5

JAVA

Creare documenti PDF
on the fly in Java

SICUREZZA

Secure Tunneling
tra servers insicuri

Provati per voi

GSPDA Jasper S20

Microsoft Office 2007
Beta 2

Programmo Subito

Invio e ricezione di messaggi SMS
tramite il Compact Framework 2.0

Ottimizzazione in C++
Anonymous Web Surfing

JSR184

Utilizzare il 3D in J2ME



• **Phonon - KDE**
verso il **multimedia**

• **In chiave di Linux,**
come realizzare
un **distro Live USB**



GRUPPO EDITORIALE
INFOMEDIA
THE SOFTWARE SIDE OF YOUR MIND

ISSN 1124-5468



9 771124 546002 6 0143

di Roberto Casarini
rcasarini@infomedia.it

Tutti al mare... tutti al mare a mostrare il palmare...

Durante una cena estiva, mi sono trovato a parlare, con persone più o meno "integrate" nel mondo dell'informatica, di come si evolve l'approccio dell'informatico agli strumenti legati alla sua "passione professionale" durante le vacanze estive. Ne è uscita una divertente analisi, corredata dal racconto di varie situazioni vissute dagli interlocutori, di come sia possibile al giorno d'oggi essere a stretto contatto con un mondo parallelo (l'informatica?) in qualsiasi luogo (o quasi) del mondo reale. La presenza di questo atteggiamento, che ormai accomuna sempre più persone di ogni sesso, età, stile di vita e censo si manifesta nella coscienza dei taluni di appassionati suddetti anche per la scelta dell'abbigliamento: t-shirt con simboli o messaggi a noi conosciuti, t-shirt dei veri LUG sparsi in tutta Italia.

È stato divertente quindi convenire sulle caratteristiche atipiche che identificano le persone che rientrano nella "evoluita specie" degli informatici, dato che non si può non notare la tendenza di questi individui ad assumere comportamenti inequivocabili riconoscibili non necessariamente dalla presenza di tecnologia o libri o riviste del settore.

Coloro che in spiaggia stazionano sotto l'ombrellone con riviste e libri (chi di voi non ha letto DEV in spiaggia?) o chi ha tentato l'impresa più ardua di portarsi il proprio notebook sotto il sole rischiando il pericolo di acqua e sabbia. Non sono mancati nelle affollate spiagge i palmari che non temono queste insidie in quanto più compatti e pratici.

Coloro che in un rifugio di montagna si appartano su un tavolo di legno come soffocati dalla silenziosa lentezza della natura alla spasmodica ricerca di un brandello di rete.

Coloro che contemplando il panorama dall'altro delle mura di un castello medievale che stanno visitando non possono far a meno di considerare con inevitabile stupore che ai tempi in cui la visita lo ha calato un'informazione per arrivare da un paese ad un altro anche poco distante in linea d'aria impiegava, se tutto andava bene, almeno qualche ora.

I più evoluti, in una pausa tra le attività di vacanza e le attività informatiche non disdegnano, trovandosi a vedere l'orizzonte, riflettere su come per i primi navigatori del mondo "reale" non fosse comprensibile la vastità geografica del mare, come adesso del resto non sono pienamente comprensibili le potenzialità dell'informatica, nella quale comunque ci si avventura con la passione dettata da propositi di progresso comune.

Tutte queste situazioni possono sembrare estreme ad un occhio esterno, talvolta lo sono, anche se non vi nego che tutte le volte che devo fare le valigie il posto per il mio fido portatile con i vari "gadget" c'è sempre.

Il rinnovamento tecnologico dell'intero settore alberghiero già da diversi anni permette connessioni Wi-Fi e anche l'industria dei trasporti sta agevolando sempre più le nostre richieste. Teoricamente potremmo svolgere le principali attività lavorative anche lontano dai nostri uffici, non tanto per evadere dalla quotidianità dell'ufficio ma per cercare di reperire determinate mancanze dovute al troppo virtuale che ci circonda. Anche per gli informatici alcune società hanno intrapreso la strada del ritiro lavorativo dove appunto si cerca la massima produttività in un contesto naturale dove si possa raggiungere la giusta concentrazione e creatività.

L'informatico in vacanza si confronta con la realtà. E da questa esperienza nascono interessanti intersezioni tra presente, passato e futuro sulle quali sarebbe bene riflettere in un contesto che più ci aggrada (e non solo in vacanza...).

Roberto Casarini



Lo speciale di questo numero:

JSR184 - Utilizzare il 3D in J2ME

- 12 **JSR184 - Utilizzare il 3D in J2ME**
di Mauro Pesci
- 17 **JSR184 - Utilizzare il 3D in J2ME, la modalità immediate mode**
di Mauro Pesci
- 22 **JSR184 - Utilizzare il 3D in J2ME, luci materiali e...**
di Mauro Pesci
- 29 **Giochi per cellulare, la nuova età dell'oro del "garagegaming"**
di Maurizio Terzo



8

Programmazione

- 32 **Database - Note di Data Warehouse:**
Le Dimensioni di analisi - ultima parte
di Massimo Cenci
- 37 **Java - Nome in codice bshd5**
di Francesco Tinti
- 41 **Java - Creare documenti PDF on the fly in Java**
di Carlo Lollo
- 47 **Sicurezza - Secure Tunneling tra servers insicuri**
di Massimo Nardone

Rubriche

- 11 **Algomatich del doctor Brom - Salvare capra, cavoli e tutto il resto**
di Maurizio Codogno
- 53 **Reportage - I tre giorni di Visual Studio 2005 Team System**
di Paolo De Nictolis

In Prova

GSPDA Jasper S20

di Stefano Sanna

8

Microsoft Office 2007 Beta 2

di Paolo De Nictolis

90

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

programma_subito@informedia.it
e linux@informedia.it

I listati e i programmi di utilità che corredano gli articoli sono disponibili gratuitamente dal sito <http://www.informedia.it/DEV> oppure da [ftp://ftp.informedia.it/pub/DEV/Listati/](http://ftp.informedia.it/pub/DEV/Listati/)



Programmo Subito

Numero 95

- 58 **Livello 1** Invio e ricezione di messaggi SMS tramite il Compact Framework 2.0 di Emanuele Mattei
- 63 **Livello 2** C# & PHP: file sharing via http - ultima parte di Francesco De Angelis
- 68 **Livello 2** Ottimizzazione in C++ di Massimiliano Scarano
- 71 **Livello 1** Anonymous Web Surfing - ultima parte di Adriano Dorato e Carlo Lollo

Linux Subito

Numero 75

- 76 **Livello 1** Phonon - KDE verso il multimedia di Fabrizio Angius
- 80 **Livello 2** IDE, fare senza di Alessio Saltarin
- 85 **Livello 3** In chiave di Linux - parte 2 di Matteo Lucarelli

Ai lettori

Developer's Tools a cura di Lorenzo Vandoni
Multimedia Conversion Library

46

Planet Open Source a cura di Alessio Saltarin
#4 - FLOSS?

62

DEV in DIRICTO a cura di Massimo Farina
Mobile number portability: quali sono le regole che la governano?

70

Java Me World a cura di Stefano Sanna
La libertà di essere Beta

75

Dev Developing Software Solutions Numero 143
Settembre 2006 Anno XIV

Direttore Responsabile
Marialetizia Mari (mmari@infomedia.it)

Coordinamento Editoriale
Roberto Casarini (rcasarini@infomedia.it)

Collaboratori
Fabrizio Angius, Massimo Cenci, Maurizio Codogno, Francesco De Angelis, Paolo De Nictolis, Adriano Dorato, Massimo Farina, Carlo Lollo, Matteo Lucarelli, Emanuele Mattei, Massimo Nardone, Mauro Pesci, Alessio Saltarin, Stefano Sanna, Massimiliano Scarano, Maurizio Tero, Francesco Tinti, Lorenzo Vandoni

Grafica e Impaginazione
Valentina Mucci (grafica@gruppoinfomedia.it)

GRUPPO EDITORIALE
INFOMEDIA

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

Direzione
Natale Fino (nfino@infomedia.it)

Divisione Libri
Lina Vanni (lvanni@infomedia.it)

Marketing & Advertising
Tel. 0587 73 64 60
marketing@infomedia.it

Supporto Sistemistico
webmaster@infomedia.it

Amministrazione
Sara Mattei (comunicazione@infomedia.it)

Segreteria
Enrica Nasti (enr@infomedia.it)

Stampa
Betagraf Spa - Fano di Argelato (BO)

Distribuzione
Parrini & C. (Roma)

Ufficio Abbonamenti
Tel. 0587 73 64 60
Fax 0587 73 22 32
abbonamenti@infomedia.it
<http://online.infomedia.it>

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

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

"DEV, Developing Software & Solutions" è una rivista del Gruppo Editoriale Infomedia S.r.l.
Divisione e Amministrazione
Via Valdara P. 116 - Pontacco
Registrazione presso il Tribunale di Pisa
n. 21 del 25.11.1999
Consenso pubblicitario inferiore al 45%

JSR184 Utilizzare il 3D in J2ME

Si stanno aprendo le porte ad un nuovo mondo sulla programmazione di giochi per i cellulari. In questo viaggio conosceremo le tecniche essenziali per sfruttare al meglio queste nuove potenzialità

di Mauro Pesci > mpesci@infomedia.it

Giocare con i cellulari è diventato un passatempo molto comune. Il numero dei giochi disponibili cresce in misura esponenziale mese dopo mese, contemporaneamente alla crescita del parco dei dispositivi che supportano Java e alla continua evoluzione delle loro caratteristiche hardware. Fino ad ora, le prestazioni dell'hardware di tali dispositivi hanno permesso, quindi, favorito la creazione di giochi che ricalcavano l'ostile di diversi anni fa. Si trattava esclusivamente di programmi con grafica ed impostazione bidimensionale composti da oggetti e sprite disegnati pixel per pixel.

Il passo successivo sarà, ovviamente, quello di conquistare l'universo del 3D. Anche se buona parte dei cellulari oggi in commercio possiede l'hardware dedicato necessario all'accelerazione delle operazioni grafiche 3D, è ovvio che tali dispositivi si collocano in una fascia di prestazioni ben al di sotto delle, ormai famosissime, console portatili come il *Playstation Portable* o il *Nintendo DS*.

Nonostante tutto, però, c'è motivo di credere che nell'immediato futuro assisteremo ad una notevole diffusione di giochi 3D per cellulari, considerando anche la velocità con la quale vengono migliorate le caratteristiche dei telefoni in commercio.

Per questo sono state create le *Mobile 3D Graphics API*, chiamate per brevità *M3G* e definite con il nome di *JSR-184*. Si tratta di uno sforzo per stabilire *API 3D* standard per dispositivi che supportano *J2ME*.

Chiunque ha avuto esperienze di programmazione con le *API* di *J2ME* avrà certamente imparato a convivere con le loro limitazioni. Queste sono dovute dalla necessità di creare standard adatti ad abbracciare una gamma più ampia possibile di dispositivi che possiedono limitatissime caratteristiche hardware.

Anche le *JSR-184*, purtroppo, devono rispettare questa esigenza. Di conseguenza, disporre di tutte le funzionalità fruibili in altri contesti come le centinaia di funzioni *Open-*

GL o *DirectX*, rappresenta, almeno per il momento, un sogno ancora irrealizzabile.

Nonostante ciò il programmatore può contare sulla disponibilità di un insieme completo di classi utili per le trasformazioni dei vertici, la gestione delle luci, il supporto per il texture mapping, gli sprite 3D, la gestione della trasparenza, dei materiali, dell'effetto nebbia e delle telecamere.

Come vedremo, però, se da un lato tale standard ingloba solo quanto è essenziale per poter ottenere una animazione 3D accettabile, dall'altro risulta molto semplice e, a volte immediato, sfruttarne le funzionalità. La **Figura 1** mostra un esempio di applicazione 3D scaricata dal sito www.forum.nokia.com.



FIGURA 1

Modalità di utilizzo

M3G può essere divisa nettamente in due parti: ossia modalità *immediate* e modalità *retained*. In modalità *immediate* ogni oggetto 3D che compone una scena viene elaborato e disegnato individualmente. In pratica questa modalità può essere considerata come una via di accesso a basso livello alle funzioni 3D della *API*, allo stesso modo di come avviene per le *OpenGL*.

Il programmatore, quindi, ha la responsabilità di impostare manualmente i vertici degli oggetti, effettuare le trasformazioni, attivare le luci, caricare le texture e così via.

In modalità *retained*, invece, è possibile utilizzare un tipo di file in formato M3G, nel quale si possono immagazzinare tutti i dati per descrivere un intero mondo composto da oggetti 3D, telecamere, luci e animazioni.

Per visualizzare una scena animata in un'applicazione, utilizzando i file M3G, è sufficiente una manciata di linee di codice, per questo risulta molto comoda ed immediata da utilizzare.

Questo aspetto ne esalta ovviamente la comodità di utilizzo. Essendo, però, nostra intenzione esplorare le funzionalità che stanno alla base della creazione delle scene 3D, in questo appuntamento, ci occuperemo esclusivamente della modalità *immediate* rimandando ad un altro articolo la trattazione di questo interessante argomento.

Le basi

In entrambi i casi, comunque, per poter accedere alle funzionalità di M3G è necessario importare il seguente *package*:

```
javax.microedition.m3g.*
```

Nel **Riquadro 1** sono elencate tutte le classi presenti all'interno del suddetto *package*. Noterete subito il loro numero particolarmente ridotto.

RIQUADRO 1

Le componenti cruciali che intervengono alla creazione di un videogioco in Game Maker

AnimationController	Mesh
AnimationTrack	MorphingMesh
Appearance	Node
Background	Object3D
Camera	PolygonMode
CompositingMode	RayIntersection
Fog	SkeletalMesh
Graphics3D	Sphere3D
Group	Texture2D
Image3D	Transform
IndexBuffer	Transformable
KeyframeSequence	TriangleStripArray
Light	VertexBuffer
Loader	VertexBuffer
Material	World

In aggiunta, oltre ad importare il solito

```
javax.microedition.lcdui.*
```

le nostre applicazioni devono estendere le classe *Canvas* o anche *GameCanvas*. Tali passaggi si rendono necessari, poiché, per poter visualizzare le scene sottoposte a *rendering* è necessario che M3G possa lavorare direttamente con l'oggetto *Graphics*. Quest'ultimo, come molti già sanno, ha accesso diretto alle funzioni di basso livello che gestiscono il contesto grafico del dispositivo.

Così, ogni volta che si renderà necessario disegnare il risultato di un rendering, comunemente nel caso dell'esecuzione del metodo *paint*, dovremo indicare l'istanza dell'oggetto *Graphics* da utilizzare.

Prepariamo, quindi, lo scheletro di una classe generica

per poter avere una visione più chiara di come lavorerà il codice

```
import javax.microedition.lcdui.*;
import javax.microedition.m3g.*;
public class Applicazione3D extends Canvas {
    public Applicazione3D() {
        Init();
    }
    protected void Init() {
        // Qui verrà inserito il codice di inizializzazione
    }
    protected void paint(Graphics g) {
        // Qui viene inserito il codice di visualizzazione
    }
}
```

Quando tutto è pronto la prima cosa da fare è creare un oggetto di tipo *Graphics3D*. Scriveremo la seguente dichiarazione appena dopo quella della classe *GameCanvas*

```
Graphics3D g3d;
```

Per utilizzare l'oggetto dobbiamo recuperare l'istanza del *singleton* associato alla nostra applicazione tramite

```
g3d = Graphics3D.getInstance();
```

La classe *Graphics3D*, come abbiamo detto, è un *singleton* che può essere agganciato al contesto grafico del dispositivo. Tutte le operazioni di *rendering* vengono eseguite con la chiamata al metodo *render()* di questa classe.

Per capire come utilizzarla possiamo osservare lo spezzone di codice inserito nel metodo *paint()* della nostra classe.

```
protected void paint(Graphics g) {
    g3d.bindTarget(g);
    g3d.clear(null);
    // Qui viene inserito il codice che disegna la scena
    g3d.releaseTarget();
}
```

Il meccanismo di funzionamento, come si può vedere, è estremamente semplice. In termini pratici la prima riga di codice

```
g3d.bindTarget(g);
```

non fa altro che rilasciare all'oggetto *Graphics3D* il riferimento alla classe *Graphics* passato come argomento dal sistema.

I più esperti e attenti avranno sicuramente notato che è possibile agganciare l'oggetto *Graphics3D* a qualunque immagine di tipo mutabile per ottenere rendering su immagini in memoria anziché direttamente nel display.

Esiste anche un'altra definizione di questo metodo la quale permette di abilitare/disabilitare il *depth buffer* e specificare un *hint*, ossia un suggerimento riguardo alla qualità dell'output renderizzato.

Infatti, in determinate situazioni può rivelarsi più importante

la qualità della velocità di resa delle immagini. Per questo MJG prevede tre opzioni: *antialiasing*, il *dithering* e il *true color rendering*.

È necessario tenere presente, che tali opzioni non sono altro che semplici "suggerimenti" e, per questo, su alcune implementazioni possono essere tranquillamente trascurate.

Dopo che il contesto grafico è stato agganciato, l'istruzione seguente

```
g3d.clear(null);
```

non fa altro che pulire il contenuto del contesto grafico appena agganciato.

A questo punto è possibile effettuare tutte le operazioni necessarie a creare la scena che deve essere renderizzata; si possono disegnare e trasformare solidi, aggiungere luci, creare e muovere telecamere e così via.

Chiude il processo di disegno il metodo

```
g3d.releaseTarget();
```

che rilascia il contesto grafico.

Prima della chiamata *bindTarget* e dopo la *releaseTarget* è possibile utilizzare normalmente l'oggetto *Graphics* come in una classica applicazione 2D.

In questo modo senza la necessità di ulteriori operazioni possiamo unire facilmente la grafica 2D con quella 3D, aggiungere del testo o, addirittura, rielaborare l'immagine prodotta.

Utilizzare la grafica 2D prima della *bindTarget* può non servire a molto dato che tutte le operazioni di disegno vengono annullate dalla chiamata *clear(null)*. Potremmo pensare di eliminare questa chiamata, cancellare manualmente lo schermo ed inserire un'immagine di sfondo, ma questa operazione può essere eseguita, come vedremo anche da MJG.

A questo proposito aggiungiamo le due funzioni *Draw3D()* e *Draw2D()* alla classe e nel metodo *paint()*, in questo modo, inseriamo le chiamate:

```
protected void paint(Graphics g) {
    g3d.bindTarget(g);
    g3d.clear(null);

    Draw3D();

    g3d.releaseTarget();

    Draw2D();
}
```

Lo scheletro dell'applicazione

Il **Listato1** ed il **Listato2** presentano, rispettivamente, il codice contenuto nei file *templateMIDlet* e *templateCanvas* che è possibile scaricare dal sito FTP.

Anche se questa applicazione non visualizza praticamente nulla, essa rappresenta il punto di partenza, il codice è già pronto e compilabile.

LISTATO 1

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
public class templateMIDlet extends MIDlet
{
    templateGameCanvas template2DCanvas;
    private boolean init = false;
    private boolean pausa = false;
    public templateMIDlet(){
        Instances midlet = this;

    }

    public void startApp()
    {
        if (init == false) init();
        Display.getDisplay(this).setCurrent(template2DCanvas);
        try
        {
            if (pausa == false)
            {
                template2DCanvas.start();
            }
            else
            {
                template2DCanvas.restart();
            }
            pausa = false;
        }
        catch(Exception e)
        {
            gameFramework.utile.Error.Show(e);
        }
    }

    public void pauseApp()
    {
        template2DCanvas.pause();
        pausa = true;
    }

    public void destroyApp(boolean unconditional)
    {
        template2DCanvas = null;
        System.gc();
        notifyDestroyed();
    }

    private synchronized void init()
    {
        if (!init)
        {
            try
            {
                template2DCanvas = new templateGameCanvas(this);
                init = true;
            }
            catch (Exception e)
            {
                init = false;
            }
        }
    }
}
```

LISTATO 2

```

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
import javax.microedition.media.*;
import javax.microedition.m3g.*;
import java.io.*;
import java.util.*;

public class templateCanvas extends GameCanvas implements Runnable
{
    public static Graphics    g2d;
    public static Graphics3D  g3d;
    templateMIDlet    istanza_midlet;

    public static final int    MILLISEC_PER_FRAME = 10;
    public static int    LARGHEZZA_DISPLAY;
    public static int    ALTEZZA_DISPLAY;

    private volatile        Thread aniThread = null;
    Thread                dThread;
    long                startTime;
    long                endTime;

    private boolean        graphicsSet = false;

    private boolean        keyPressed = false;
    private boolean        keyReleased = false;
    private int            previousKeyPressed = 0;
    private int            keyStates;

    // Costruttore della classe Canvas
    templateCanvas (templateMIDlet _templateMIDlet)
    {
        super(true);

        // Memorizzazione del riferimento alla
        // classe MIDlet
        istanza_midlet = _templateMIDlet;

        // Impostazione della modalità 1/4 a schermo pieno
        setFullScreenMode(true);

        g3d = Graphics3D.getInstance();
    }

    // Metodo richiamato dalla classe
    // MIDlet per lanciare l'applicazione
    public void start()
    {
        // Inizializzazione
        init();

        // Creazione e avvio di un nuovo
        // thread
        aniThread = new Thread(this);
        aniThread.start();

        // Lancio dell'applicazione
        run();
    }

    public void init() {
        }

        // Richiamato dalla classe MIDlet per
        // ripristinare l'esecuzione dopo
        // un'interruzione
        public void restart()
        {
            aniThread.notify();
        }

        // Richiamato dalla classe MIDlet per
        // sospendere l'esecuzione del
        // programma
        public void pause()
        {
            try
            {
                aniThread.wait();
            }
            catch (Exception e)
            {
            }
        }

        // Richiamato dalla classe MIDlet per
        // terminare l'esecuzione del programma
        public void stop()
        {
            aniThread = null;
        }

        public void exit()
        {
            stop();
            istanza_midlet.destroyApp(true);
        }

    public void run()
    {
        dThread = Thread.currentThread();

        try
        {
            while (dThread == aniThread)
            {
                startTime = System.currentTimeMillis();

                if (isShown())
                {
                    keyboard();
                    frame();

                    repaint();
                    serviceRepaints();

                    endTime = System.currentTimeMillis() - startTime;

                    if (endTime < MILLISEC_PER_FRAME)
                    {

```

LISTATO 2

```

        synchronized(this)
        {
            wait(MILLS_PER_FRAME - endTime);
        }
    }
}
}
catch (InterruptedException ex)
{
    // Errore
}
}

void frame()
{
    // Qui viene inserito il codice che effettua
    // le operazioni periodiche di gestione dell'applicazione
}

void keyboard()
{
    keyStates = getKeyStates();

    if (!keyPressed)
    {
        keyPressed = true;
    }

    if ((keyStates & GameCanvas.LEFT_PRESSED) != 0)
    {
    }
    else if ((keyStates & GameCanvas.RIGHT_PRESSED) != 0)
    {
    }
    else if ((keyStates & GameCanvas.FIRE_PRESSED) != 0)
    {
    }
    else if ((keyStates & GameCanvas.UP_PRESSED) != 0)
    {
    }
}

}
else if ((keyStates & GameCanvas.DOWN_PRESSED) != 0)
{
}
else if (keyStates == 0)
{
    keyPressed = false;
}
else
{
}
}

public void paint(Graphics g)
{
    if (graphicsSet == false)
    {
        graphicsSet = true;
        g2d = g;

        // Recupero delle dimensioni del display
        LARGHEZZA_DISPLAY = getWidth();
        ALTEZZA_DISPLAY = getHeight();
    }

    g2d.bindTarget(g2d);
    g2d.clear(null);

    Draw3D();

    g2d.releaseTarget();

    Draw2D();
}

private void Draw3D() {
}

private void Draw2D() {
}
}

```

Se utilizzate NetBeans per la programmazione troverete utili i progetti pronti, contenuti nella stessa cartella dei sorgenti.

Conclusioni

Fino a qui è tutto abbastanza semplice ed immediato. Abbiamo visto come partire per la realizzazione del 3D e che non necessita di procedure particolarmente lunghe, tediose o criptiche disseminate di parametri difficili da ricordare. Dal prossimo articolo, però, inizieremo a vedere argomenti facilmente comprensibili a chi ha già una certa padronanza dei concetti propri della programmazione 3D, come le operazioni con le matrici, con i vertici, con le trasformazioni.

Bibliografia

[1] Sun Corporation, "Mobile 3D Graphics API Specification", www.sun.com

Mauro Pesci

Lavora come libero professionista nel settore dello sviluppo del software, in particolare modo, si sta occupando della realizzazione di alcuni videogiochi commerciali per telefonini commerciali distribuiti tramite il portale di H3G..

JSR184

Utilizzare il 3D in J2ME la modalità immediate mode

Gli aspetti principali della programmazione grafica 3D sono la gestione dei vertici, dei poligoni e la loro rappresentazione a video. Svilupperemo un piccolo esempio pratico per comprendere i meccanismi utilizzati da M3G

di **Mauro Pesci** > mpesci@infomedia.it

Con questo articolo studieremo esempi pratici di programmazione 3D in immediate mode. Vedremo come creare, visualizzare e muovere oggetti tridimensionali, applicheremo effetti di luce e texture per una migliore realistica rappresentazione.

Poiché non rientra, nello scopo di questo speciale trattare i concetti basilari del 3D, come le operazioni matematiche sulle matrici, le trasformazioni e così via, consiglio a coloro che non hanno esperienze specifiche di fare riferimento agli articoli pubblicati su queste pagine come riportato nella bibliografia. È importante anche fare riferimento alla documentazione delle JSR184, associata a questi articoli che potete trovare sul sito FTP, dato che per questioni di spazio non saranno presentati tutti i metodi, o le loro varianti, incluse nelle specifiche API.

Utilizzo dei vertici

L'elemento che sta alla base di tutta l'architettura dei motori 3D è il vertice. Ogni vertice, espresso da una terna di coordinate, può essere traslato e ruotato nello spazio. Può avere diverse proprietà (colore, trasparenza, normali ecc.) le quali concorrono a determinare l'aspetto finale dei poligoni. Gli effetti di gouraud shading e di illuminazione, infatti vengono disegnati prendendo come riferimento i valori delle proprietà dei vertici.

M3G mette a disposizione due classi, per una gestione completa dei vertici, la `VertexBuffer` e la `VertexArray`. Quest'ultima ha una utilità polivalente in quanto può essere utilizzata per memorizzare indipendentemente i vari tipi di informazioni relative ai vertici.

La **Figura 1** mostra come il `VertexArray` è utilizzato rispettivamente per contenere le coordinate dei vertici, le normali, i colori e le coordinate delle texture.

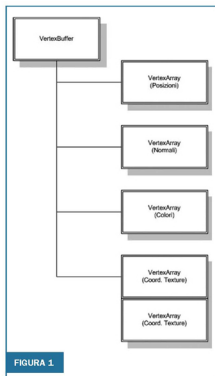


FIGURA 1

Ogni riferimento ai VertexArray, relativi ad uno stesso oggetto, è memorizzato nel VertexBuffer il quale si comporta, praticamente, da contenitore.

Per maggiore chiarezza procediamo con un esempio classico e semplice al contenuto, ossia la creazione di un cubo.

Dichiariamo, perciò, un array di byte nel quale immagazziniamo le coordinate spaziali del nostro cubo

```
private static final byte[] POSIZIONI_VERTICI = {
    -1,-1, 1,1,-1, 1,-1, 1, 1, 1, 1, 1,
    -1,-1,-1, 1,-1,-1, 1,-1, 1,-1, 1,-1
};
```

dichiariamo anche una variabile di classe di tipo VertexBuffer

```
private VertexBuffer datiVerticiCubo;
```

Nel metodo init() della classe strutturata nel precedente articolo scriviamo queste righe di codice

```
datiVerticiCubo = new VertexBuffer();
VertexArray posizioniVertici = new VertexArray(POSIZIONI_VERTICI.length / 3, 1);
posizioniVertici.set(0, POSIZIONI_VERTICI.length / 3, POSIZIONI_VERTICI);
datiVerticiCubo.setPositions(posizioniVertici, 1.0f, null);
```

La prima si commenta da sé, ossia crea un nuovo oggetto VertexBuffer. Poi, come secondo passo, è necessario creare un array di vertici da memorizzare nel buffer. Per questo motivo, l'istruzione successiva crea un oggetto VertexArray.

Come primo parametro esso richiede il numero dei vertici di cui è composto l'oggetto che stiamo creando. Nel nostro caso, dato che ogni vertice è definito da tre byte uno per ogni coordinata, passiamo il valore della dimensione dell'array POSIZIONI_VERTICI diviso 3.

Tramite il secondo e terzo parametro indichiamo il numero di componenti di ogni vertice e la dimensione in byte di ognuna. Ogni vertice del nostro cubo, infatti, è formato da 3 coordinate (x, y e z) le quali, abbiamo stabilito, che rientrano nel campo dei valori di un tipo di dato byte. Di conseguenza abbiamo impostato un numero di componenti pari a 3 di 1 byte ognuna.

Procederemo, poi, trasferendo i valori dei vertici contenuti nell'array POSIZIONI_VERTICI, tramite il metodo set (l'unico) della classe VertexArray che presenta queste dichiarazioni

```
void (int firstVertex, int numVertices, byte[] values)
void (int firstVertex, int numVertices, short[] values)
```

Dato che durante la costruzione l'oggetto ha allocato tutta la memoria necessaria a contenere le definizioni dei vertici, il parametro firstVertex serve, in pratica, ad indicare da quale dei vertici allocati inizia la scrittura. numVertices richiede quanti vertici vogliamo prelevare dall'array indicato nel terzo parametro.

La differenziazione tra le due dichiarazioni del metodo dipende dalla dimensione in byte di ogni componente, in

accordo a quanto passato al costruttore dell'oggetto tramite il parametro componentSize.

Qualora, ad esempio, il valore dei vertici sia superiore all'intervallo da -128 a 127 diventa necessario che l'array dei vertici venga dichiarato come short[].

L'ultima istruzione, del blocco di codice appena visto, così definita

```
datiVerticiCubo.set(positions, float scale, float[] bias)
```

inserisce il riferimento del VertexArray, contenente la definizione dei vertici del cubo, nel VertexBuffer. Il parametro scale permette di indicare un fattore di ridimensionamento dell'oggetto, compreso tra 0.0f e 1.0f (il valore 1.0f lascia inalterate le dimensioni dell'oggetto), mentre bias richiede una terna di costanti (X, Y, Z) per da sommare ai vertici per una traslazione spaziale. Non avendo necessità di sfruttare il bias, lo impostiamo a null.

Impostare i poligoni

A questo punto abbiamo impostato solo le posizioni dei vertici del nostro cubo. Ovviamente per poter essere disegnato, a video come un cubo, si rende necessario congiungere tra loro i vertici per formare una figura solida. Sappiamo già che ogni solido è costituito da più poligoni e che ogni poligono può essere scomposto in più triangoli. M3G per questioni di ottimizzazione e semplicità sui calcoli lavora solo con questo tipo di figura.

Utilizzeremo, quindi, la classe TriangleStripArray, la quale non è altro che una concatenazione di triangoli.

Con questo sistema dopo aver specificato i vertici del primo triangolo, per disegnare i successivi è sufficiente semplicemente indicare un solo vertice. Gli altri due vertici di ogni nuovo triangolo vengono presi dagli ultimi specificati per triangolo precedente.

In questa maniera viene anche ridotto il numero di informazioni necessarie per disegnare un oggetto. In Figura2 è proposto un esempio per chiarire questo concetto.

Sappiamo che un cubo è formato da 8 vertici e, per poterlo coprire con dei triangoli, avremmo bisogno di 6 vertici per faccia, ossia, $6 \times 6 = 36$ vertici. Con il triangle-strip riusciamo a ridurre il tutto a 4 (per la prima faccia) + (5 facce * 2 triangoli per faccia) = $3 + 11 = 14$ vertici.

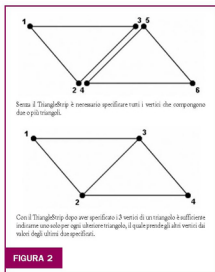
È ovvio che quando si deve disegnare un oggetto molto complesso il risparmio influisce notevolmente anche sulle prestazioni e sulla memoria necessaria per memorizzarne la definizione.

Creiamo, allora, un nuovo array contenente gli indici dei vertici da collegare

```
private static int[] INDICI_TRIANGOLI = {
    0, 1, 2, 3, 7, 1, 5, 4, 7, 6, 2, 4, 0, 1
};
```

è da notare che il valore di ogni vertice fa riferimento all'ordine indicato nel VertexArray delle posizioni.

A questo punto dobbiamo creare un nuovo oggetto TriangleStripArray. Dichiariamo la variabile di classe



```
private TriangleStripArray triangoliCubo;
```

ed inseriamo questa linea in fondo al metodo Init()

```
triangoliCubo = new TriangleStripArray(INDICI_TRISANGOLO, new int[] {INDICI_TRISANGOLO.length});
```

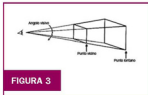
Le telecamere

Una telecamera definisce il punto di vista dell'osservatore in termini di posizione ed orientamento. È una classe molto utile poiché ad ogni trasformazione effettuata su di essa tutti gli oggetti che compongono la scena vengono trasformati di conseguenza. Diventa così molto semplice far viaggiare l'osservatore all'interno di un mondo virtuale.

In una scena possono anche coesistere più telecamere; in questo modo è possibile tenere aggiornati diversi punti di osservazione ed attivarli all'occorrenza.

Se provassimo ad eseguire il programma a questo punto vedremmo un messaggio d'errore causato dalla assenza "sulla scena" di una telecamera. La classe Graphics3D, infatti, non è in grado di determinare un punto di osservazione dal quale basare tutti i calcoli e le operazioni di disegno. Sempre in fondo al metodo Init() bisogna creare una nuova istanza dell'oggetto camera

```
Camera telecamera = new Camera();
```



Un'operazione molto importante nella procedura di impostazione di una telecamera è la scelta del tipo di proiezione da utilizzare. Tale scelta può ricadere sulla proiezione prospettica o parallela. La seconda, solitamente poco utilizzata se non nelle applicazioni CAD, produce un output "piatto" dove la distanza dei vertici dall'osservatore non influisce sul risultato.

Nel caso della proiezione prospettica, invece, viene simulato l'effetto naturale alla quale è affetto l'occhio umano. Oggetti, delle stesse dimensioni, distanti dall'osservatore appaiono più piccoli di quelli vicini.

Riproducendo questo fenomeno è possibile rendere l'impressione della profondità sui display che possono visualizzare solo immagini bidimensionali.

Vediamo, quindi, come impostare questo tipo di proiezione sulla telecamera. Innanzitutto calcoliamo il rapporto tra larghezza ed altezza del display del dispositivo

```
float aspectRatio = (float) GetWidth() / (float) GetHeight();
```

Questo valore serve a riproporzionare l'output. Non è desiderabile, infatti, che qualora un'applicazione venga eseguita su un cellulare che possiede un display di dimensioni diverse da quelle per il quale è stato progettato, le immagini risultino distorte o stirate.

Dopo che impostiamo il volume di taglio con la riga

```
telecamera.setPerspective(30.0f, aspectRatio, 1.0f, 1000.0f);
```

Il volume di taglio, mostrato in **Figura3**, non è altro che la porzione spazio visibile dall'osservatore. Il primo parametro passato rappresenta l'ampiezza dell'angolo visuale, mentre gli ultimi due indicano il punto più vicino visibile ed il più lontano. Tutti i vertici al di fuori del tronco di piramide che viene ad originarsi non saranno disegnati. Questo è un altro meccanismo utilizzato per snellire le operazioni di disegno evitando di perdere tempo utile nell'elaborare oggetti che, per la loro posizione, non sono visibili all'osservatore.

Come ultimo passo provvediamo a collegare la telecamera appena creata all'oggetto Graphics3D tramite

```
g3d.setCamera(telecamera, null);
```

Per fare in modo che la API processi tutti i dati inseriti per il cubo, nel metodo Draw3D, scriviamo questa riga

```
g3d.render(dat/VerticiCubo, triangoliCubo, new Appearance(), null);
```

A questo punto possiamo fare riferimento al **Listato1** per vedere chiaramente tutte le modifiche apportate al metodo Init() della nostra classe.

Abbiamo già accennato al metodo render() della classe Graphics3D. Passandogli, come parametri, un VertexBuffer, il riferimento al relativo TriangleStripArray, un oggetto Appearance, di cui parleremo nel prossimo articolo, provvederà ad effettuare tutti

i passi della pipeline necessari ad ottenere un'immagine bidimensionale direttamente sul display.

Qualora volessimo visualizzare una scena composta da più oggetti dovremmo lanciare ripetutamente il metodo con VertexBuffer e TriangleStripArray caricati ogni volta con valori diversi.

Qualcuno potrebbe essere tentato di creare un mega VertexBuffer contenente i dati di tutti gli oggetti da disegnare. Questo procedimento può essere attuato solo nei confronti di insiemi di oggetti fissi poiché, come vedremo nel prossimo paragrafo, le trasformazioni possono essere applicate solo al momento della chiamata al render(). Per questo motivo da una stessa operazione di trasformazione vengono affetti tutti i vertici del buffer.

LISTATO 1

```
public void Init() {

    // Creo una nuova istanza dell'oggetto VertexBuffer
    datiVerticiCubo = new VertexBuffer();

    // Creo un nuovo VertexArray per memorizzare i
    // le posizioni dei vertici del cubo
    VertexArray posizioniVertici = new VertexArray(POSIZIONI_
                                                    VERTICI.length / 3, 3, 1);

    // Memorizzo le posizioni dei vertici nell'array
    posizioniVertici.set(0, POSIZIONI_VERTICI.length / 3, POSIZIONI_
                                                                VERTICI);

    // Passo il riferimento dell'VertexArray al VertexBuffer
    datiVerticiCubo.setPositions(posizioniVertici, 1.0f, null);

    // Creo una nuova istanza dell'oggetto TriangleStripArray
    triangoliCubo = new TriangleStripArray(INDICI_TRIANGOLI, new int[]
                                           {INDICI_TRIANGOLI.length});

    // Creo una nuova istanza dell'oggetto Camera
    Camera telecamera = new Camera();

    // Calcolo il valore dell'aspectRatio
    float aspectRatio = (float) LARGHEZZA_DISPLAY / (float) ALTEZZA_
                                                                DISPLAY;

    // Imposto la correzione prospettica
    telecamera.setPerspective(30.0f, aspectRatio, 1.0f, 1000.0f);

    // Creo un oggetto trasformazione
    Transform trasformazioneTelecamera = new Transform();

    // Imposto una traslazione
    trasformazioneTelecamera.postTranslate(0.0f, 0.0f, 10.0f);

    // Attivo la telecamera
    g3d.setCamera(telecamera, trasformazioneTelecamera);

    initDone = true;
}
```

Le trasformazioni

Per completare la panoramica sulle funzionalità di base per lavorare con la grafica 3D è rimasto un ultimo argomento. Se provassimo a lanciare l'applicazione fino a qui scritta otterremmo "stranamente" una schermata completamente bianca.

Ciò avviene perché la telecamera creata è posizionata sull'origine, di conseguenza risulta come se l'osservatore fosse vicinissimo all'oggetto nella scena.

È necessario applicare una trasformazione. Le trasformazioni sono operazioni di traslazione, rotazione o ridimensionamento effettuate su di un oggetto. Per vedere il cubo, quindi, possiamo allontanare la telecamera dal cubo, ossia possiamo traslarla sull'asse Z.

Dato che anche le trasformazioni sono implementate tramite oggetti, ne creiamo subito uno con

```
Transform trasformazioneTelecamera = new Transform();
```

dopodiché impostiamo la traslazione necessaria tramite

```
trasformazioneTelecamera.postTranslate(0.0f, 0.0f, 10.0f);
```

Abbiamo spostato la telecamera indietro lungo l'asse Z di 10 unità. A questo punto sostituiamo la riga precedentemente scritta

```
g3d.setCamera(telecamera, null);
```

con

```
g3d.setCamera(telecamera, trasformazioneTelecamera);
```

per associare la trasformazione alla telecamera.

Eseguendo il programma, come mostrato in **Figura 4**, finalmente possiamo ammirare l'oggetto.



FIGURA 4



FIGURA 5

Un po' di animazione

Ciaspettavamo di vedere un cubo, ma sul display è presente solamente un rettangolo. Questo avviene poiché stiamo guardando direttamente verso una delle sue facce. Proviamo ad applicare una trasformazione anche all'oggetto in modo da creare un'animazione.

Aggiungiamo una variabile di classe

```
Transform trasformazioneCubo = new Transform();
```

mentre nel metodo `frame()` della classe inseriamo questa riga di codice

```
trasformazioneTelecamera.postRotate(1.0f, 1.0f, 1.0f, 1.0f);
```

questo comando applica una rotazione di un grado su tutti gli assi.

Rimane ora, da sostituire la vecchia riga

```
g3d.render(datVerticiCubo, triangoliCubo, new Appearance(), null);
```

con la nuova

```
g3d.render(datVerticiCubo, triangoliCubo, new Appearance(), trasformazioneCubo);
```

Abbiamo, quindi, associato la trasformazione al cubo; ad ogni fotogramma verrà effettuata una ulteriore rotazione di un grado generando così un'animazione progressiva.

La **Figura 5** chiarisce il funzionamento di queste istruzioni.

Conclusioni

Fino ad ora abbiamo studiato gran parte delle nozioni più importanti per essere in grado di sfruttare al meglio e velocemente le caratteristiche delle JSR-184. Con un po' di impegno e fantasia potrete creare oggetti solidi ed interi mondi da esplorare per i vostri videogiochi o per qualunque altra applicazione vogliate far girare sui vostri cellulari.

Rimangono ancora alcuni argomenti molto interessanti come gli effetti di colorazione, il texturing e gli sprite 3D che tratteremo nel prossimo articolo.

Per questo mi auguro che la lettura si sia rivelata fin qui interessante e stimolante.

Bibliografia

- [1] Sun Corporation, "Mobile 3D Graphics API Specification", www.sun.com
- [2] DEV n°132 anno XIII Settembre 2005, "Utilizzare le OpenGL" Mauro Pesci

JSR184

Utilizzare il 3D in J2ME, luci materiali e ...

Siamo arrivati a toccare l'argomento più interessante di questa serie, qualcosa che ci permetterà di arricchire di colori e luci gli oggetti 3D sui nostri cellulari. Con un minimo sforzo è possibile ottenere risultati davvero soddisfacenti.

di Mauro Pesci > mpesci@infomedia.it

I migliori videogiochi che utilizzano il 3D danno sfoggio delle più evolute tecniche di texturing, di proiezione dell'illuminazione o degli effetti speciali come gestione delle particelle e così via.

Come abbiamo già accennato in precedenza il set di funzionalità disponibili in M3G non è molto fornito, ma sufficiente a creare scene ed animazioni piuttosto complesse e realistiche.

Non so se sui tutti i dispositivi attualmente in circolazione è possibile far girare applicazioni che sfruttano tutte queste caratteristiche allo stesso momento. Spesso è necessario scendere a compromessi con la mini-macchina cercando di risparmiare calcoli e memoria dove possibile.

Un "trucco" molto utilizzato, solo per fare un esempio, è quello di colorare direttamente sulle texture gli effetti di una immaginaria sorgente di luce. Questo permette di evitare i calcoli onerosi necessari per la proiezione dell'illuminazione.

Non è escluso, però, che nel prossimo futuro entreranno in commercio cellulari con chipset di accelerazione hardware molto potenti. Si sta parlando, infatti, di introdurre una versione light delle OpenGL che è già supportata dagli N-GAGE della Nokia su piattaforma Symbian.

Aggiungere un po' di colore

Nell'esempio dell'articolo precedente abbiamo visto un cubo completamente bianco ruotare all'interno del display. La colorazione "piatta" è determinata dall'assenza,

nel *VortexBuffer* del cubo, di informazioni relative al colore di ogni vertice.

Come anche le OpenGL, M3G può gestire le interpolazioni di colore tra vertici

Organizziamo un array come il seguente

```
private static final byte[] COLOR1_VERTICES = {
    0, (byte)255, 0, 0, (byte)255, (byte)255,
    (byte)255, 0, 0, (byte)255, 0, (byte)255,
    (byte)255, (byte)255, 0, (byte)255, (byte)255, (byte)255,
    0, 0, (byte)128, 0, 0, (byte)255
};
```

nel quale sono stati impostati i colori, in formato RGB, che ogni vertice dovrà assumere. Notare che sono stati scritti solo 8 colori (8 x 3 componenti = 24 byte) perché 8 sono i vertici del cubo che deve essere colorato.

Aggiungiamo le seguenti variabili di classe

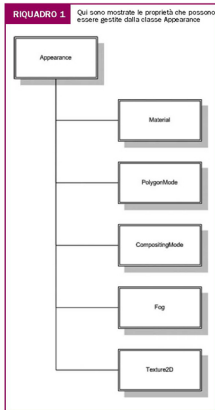
```
private Appearance apparenzaCubo;
```

la quale dichiara una nuova variabile di tipo `Appearance`. Questa classe definisce, in pratica, l'aspetto che avrà l'oggetto disegnato dal metodo `Render()`. Come mostrato dal **Riquadro 1**, questa classe mantiene le informazioni relative a varie proprietà dell'aspetto di un oggetto.

Memorizza istanze degli oggetti

- Material il quali viene utilizzato, come vedremo più avanti, per gestire l'illuminazione
- Fog che indica come deve sfumare l'oggetto in corrispondenza dell'effetto nebbia e in base alla densità della stessa.
- Texture2D che contiene informazioni riguardo a come applicare le texture all'oggetto
- PolygonMode che andiamo subito a studiare

infatti dichiariamo la variabile



`private PolygonMode modalitaPoligono;`

di tipo `PolygonMode`, una classe tramite la quale si possono impostare le modalità di disegno dei poligoni. Tali modalità sono riferite

- Al tipo di Culling da utilizzare. È possibile scegliere tra `CULL_BACK`, `CULL_FRONT`, `CULL_NONE`.
- Al tipo di Winding, `WINDING_CCW` o `WINDING_CW`
- Al tipo di Shading, scelto tra `SHADE_FLAT` e `SHADE_SMOOTH`

Spieghiamo brevemente il significato di ognuno di essi. Un poligono, quando viene disegnato nello spazio, possiede non una ma due facce. Ognuna delle due facce di un poligono può possedere diverse proprietà in termini di colore, texture ecc. Ovviamente a seconda della posizione del punto di osservazione, sarà possibile vedere una delle due facce.

Spesso però una delle facce che compongono un oggetto può non essere mai visibile; pensate all'esempio di una sfera.

Per gestire l'illuminazione è necessario impostare le proprietà del materiale che compone un poligono

Le facce interne non saranno mai visibili.

Selezionando il tipo di culling si dice alla API quale faccia "non" disegnare, accelerando così il processo di rendering.

Un problema è, però, distinguere quale sia, di un poligono, la faccia anteriore da quella posteriore. La convenzione utilizzata è quella di discriminare l'ordine in cui vengono specificati i vertici.

Se viene utilizzato un winding `CW` allora i vertici indicati seguendo il senso orario rappresentano la faccia anteriore; il contrario accade se è utilizzato il winding `CCW`.

In ultimo il tipo di Shading seleziona la modalità di riempimento, con il colore, dei triangoli. L'opzione `SHADE_SMOOTH` genera una interpolazione dei valori di colore tra i vertici. Associando opportunamente i colori ai vertici di un triangolo si possono così ottenere splendidi effetti di sfumature intermedie tra questi.

Impostando, invece, `SHADE_FLAT` per il riempimento dei triangoli verrà utilizzato il colore dell'ultimo vertice.

Appena prima la dichiarazione della telecamera, all'interno del solito metodo `Init()`, scriviamo le seguenti righe di codice

```

appearanceCubo = new Appearance();
modalitaPoligono = new PolygonMode();
  
```

LISTATO 1

Il listato completo delle modifiche per attivare i colori dei vertici

```
public void Init() {

    // Creo una nuova istanza dell'oggetto VertexBuffer
    datiVerticeCubo = new VertexBuffer();

    // Creo un nuovo VertexArray per memorizzare i
    // le posizioni dei vertici del cubo
    VertexArray posizioniVertici = new VertexArray(POSIZIONE_
                                                    VERTICI.length / 3, 3, 1);

    // Memorizzo le posizioni dei vertici nell'array
    posizioniVertici.set(0, POSIZIONE_VERTICI.length / 3, POSIZIONE_
                                                                VERTICI);

    // Passo il riferimento del VertexArray al VertexBuffer
    datiVerticeCubo.setPositional(posizioniVertici, 1.0f, null);

    // Creo una nuova istanza dell'oggetto TriangleStripArray
    triangoliCubo = new TriangleStripArray(INDICI_TRIANGOLI.length, new int[]
                                                    {INDICI_TRIANGOLI.length});

    // Creo una nuova istanza dell'oggetto Camera
    Camera telecamera = new Camera();

    // Calcolo il valore dell'aspectRatio
    float aspectRatio = (float) LARGHEZZA_DISPLAY / (float) ALTEZZA_
                                                                DISPLAY;

    // Creo una nuova istanza dell'oggetto Appearance
    apparenzaCubo = new Appearance();

    // Creo una nuova istanza dell'oggetto PolygonMode
    modalitaPoligono = new PolygonMode();

    apparenzaCubo.setPolygonMode(modalitaPoligono);

    // Creo un VertexArray per memorizzare i colori dei vertici
    VertexArray coloriVertici = new VertexArray(COLORI_VERTICI.length /
                                                    3, 3, 1);

    // Memorizzo i colori dei vertici
    coloriVertici.set(0, COLORI_VERTICI.length / 3, COLORI_VERTICI);

    // Inserisco il VertexArray nel VertexBuffer
    datiVerticeCubo.setColors(coloriVertici);

    // Imposto la correzione prospettiva
    telecamera.setPerspective(30.0f, aspectRatio, 1.0f, 1000.0f);

    // Creo un oggetto trasformazione
    Transform trasformazioneTelecamera = new Transform();

    // Imposto una traslazione
    trasformazioneTelecamera.postTranslate(0.0f, 0.0f, 10.0f);

    // Attivo la telecamera
    g3d.setCamera(telecamera, trasformazioneTelecamera);

    initDone = true;
}
```



```
VertexArray coloriVertici =
new VertexArray(COLORI_
VERTICI.length / 3, 3, 1);
coloriVertici.set(0, COLORI_
VERTICI.length / 3, COLORI_
VERTICI);
datiVerticeCubo.setColors(coloriV
ertici);
```

a parte la creazione degli oggetti *Appearance* e *PolygonMode*, è stato creato un nuovo *VertexArray* per contenere i dati dei colori il quale è stato inserito nel *VertexBuffer* tramite il metodo *SetColors()*. Per completare le modifiche al codice precedente è necessario cambiare la

vecchia riga

```
g3d.render(datiVerticeCubo, triangoliCubo, new Appearance(),
          trasformazioneCubo);
```

all'interno del metodo *Draw3D()*, con la

```
g3d.render(datiVerticeCubo, triangoliCubo, apparenzaCubo, trasformazioneCubo);
```

dove passiamo il nostro oggetto *Appearance* con le impostazioni di default.

Per impostazione di default è attiva l'opzione *SHADE_SMOOTH*; lanciamo, quindi, il programma per vedere i cambiamenti effettuati. In **Figura 1** ne è mostrato l'output, mentre il **Listato 1** presenta i cambiamenti effettuati nel metodo *Init()*.

Le luci

Per ricreare il fenomeno naturale dell'illuminazione, *M3G* dispone anche di classi per la gestione di vari tipi di fonte di luce, ossia

- *luce d'ambiente*; un tipo di illuminazione che non ha una sorgente precisa, ma proviene da tutte le direzioni
- *luce unidirezionale*; la quale è costituita da raggi di luce paralleli, come se provenissero da una distanza infinita. Diversamente alla precedente, questo tipo di illuminazione, proviene da una direzione ben precisa.
- *spotlight*; un tipo di luce che proviene da un punto nello spazio può possedere una certa ampiezza angolare e disegna un fascio sull'oggetto che colpisce. Paragonate questa alle luci che sono utilizzate nei teatri per illuminare una piccola porzione di palco.

Attivare una fonte di luce è molto semplice. Dopo aver creato i dati delle normali

```
private static final byte[] NORMALI_VERTICI = {
0, 0, 127, 0, 0, 127, 0, 0, 127, 0, 0, 127, // fronte
0, 0, -128, 0, 0, -128, 0, 0, -128, 0, 0, -128, // retro
127, 0, 0, 127, 0, 0, 127, 0, 0, 127, 0, 0, // destra
-128, 0, 0, -128, 0, 0, -128, 0, 0, -128, 0, 0, // sinistra
0, 127, 0, 0, 127, 0, 0, 127, 0, 0, 127, 0, // alto
0, -128, 0, 0, -128, 0, 0, -128, 0, 0, -128, 0, // basso
};
```

ed impostato le normali nel `VertexBuffer` con il solito sistema

```
VertexArray normalCubo = new VertexArray(NORMALI_VERTICI.length / 3, 3, 1);
normalCubo.set(0, NORMALI_VERTICI.length / 3, NORMALI_VERTICI);
datoVerticiCubo.setNormale(normalCubo);
```

si crea l'oggetto `Light`

```
Light luce = new Light();
```

poi una trasformazione da applicare alla fonte di illuminazione

```
Transform trasformazioneLuce = new Transform();
```

arretriamo la fonte di qualche unità per distanziarla dall'oggetto

```
trasformazioneLuce.postTranslate(0.0f, 0.0f, 5.0f);
```

impostiamo il tipo di luce da creare

```
luce.setMode(Light.OMNI);
```

e l'intensità della stessa

```
luce.setIntensity(3.0f);
```

alla fine segnaliamo all'oggetto `Graphics3D` di azzerare eventuali luci ed impostare quella appena creata per effettuare le operazioni di rendering.

```
g3d.resetLights();
g3d.addLight(luce, trasformazioneLuce);
```

Per sperimentare le altre impostazioni provate a cambiare il tipo di luce con i valori `Light.DIRECTIONAL`, `Light.SPOT` e `Light.AMBIENT`.

Nel caso utilizzate lo `SPOT` è necessario definire anche l'apertura angolare che il fascio di luce dovrà avere con il metodo `setSpotAngle()`

```
luce.setSpotAngle(30.0f);
```

I materiali

Quando si usa l'illuminazione per simulare il fenomeno fisico bisogna anche tenere in considerazione che, come in

natura, il colore apparente di ogni oggetto è determinato dall'assorbimento/riflessione delle radiazioni luminose dei materiali.

Per vedere quindi gli effetti di una sorgente di luce è necessario associare al `VertexBuffer` le proprietà del materiale che lo compone.

Il texture-mapping è la tecnica tramite la quale vengono utilizzate le immagini per riempire i poligoni

Sono diversi i parametri che si possono scegliere

- Riflessione d'ambiente; la luce riflessa di una illuminazione d'ambiente
- Riflessione diffusa; luce che è riflessa ugualmente in tutte le direzioni
- Riflessione speculare; è l'effetto che crea un oggetto con una superficie lucida
- Emissività; ossia la possibilità dell'oggetto di emettere luce

Per impostare un materiale si segue questo procedimento

```
Material materialeCubo = new Material();
```

poi dopo aver creato l'oggetto si usa il metodo `setColor()` in questo modo



FIGURA 2

Ora viene gestita l'illuminazione



FIGURA 3

È applicata una texture di 64x64 pixel al cubo

LISTATO 2 Il listato completo delle modifiche per l'illuminazione

```
public void Init() {
    // Attivo la telecamera
    g3d.setCamera(telecamera, trasformazioneTelecamera);

    // Creo una nuova istanza dell'oggetto VertexBuffer
    datiVerticiCubo = new VertexBuffer();

    // Creo un nuovo VertexArray per memorizzare le normali
    VertexArray normalCubo = new VertexArray(NORMALI_VERTICI.length / 3, 3, 1);

    // Memorizzo le posizioni dei vertici nell'array
    normalCubo.set(0, NORMALI_VERTICI.length / 3, NORMALI_VERTICI);

    // Memorizzo le posizioni dei vertici nell'array
    posizioniVertici.set(0, POSIZIONE_VERTICI.length / 3, POSIZIONE_VERTICI);

    // Passo il riferimento del VertexArray al VertexBuffer
    datiVerticiCubo.setPosition(posizioniVertici, 1.0f, null);

    // Imposto i valori del materiale
    materialeCubo.setColor(Material.AMBIENT, 0x00FF0000);

    // Memorizzo l'oggetto nell'Appearance
    apparenzaCubo.setMaterial(materialeCubo);

    // Creo una nuova istanza dell'oggetto Camera
    Camera telecamera = new Camera();

    // Creo un nuovo oggetto Light
    Light luce = new Light();

    // Calcolo il valore dell'aspectRatio
    float aspectRatio = (float) LARGHEZZA_DISPLAY / ((float) ALTEZZA_DISPLAY);

    // Imposto il tipo di fonte d'illuminazione
    luce.setMode(Light.DIRECTIONAL);

    // Creo una nuova istanza dell'oggetto Appearance
    apparenzaCubo = new Appearance();

    // Imposto l'intensità della luce
    luce.setIntensity(5.0f);

    // Creo una nuova istanza dell'oggetto PolygonMode
    modalitaPoligono = new PolygonMode();

    // Creo una nuova trasformazione per la luce
    Transform trasformazioneLuce = new Transform();

    // Imposto la modalità di disegno dei poligoni
    apparenzaCubo.setPolygonMode(modalitaPoligono);

    // Imposto il valore di traslazione
    trasformazioneLuce.postTranslate(0.0f, 0.0f, 3.0f);

    // Imposto la correzione prospettica
    telecamera.setPerspective(30.0f, aspectRatio, 1.0f, 1000.0f);

    // Azzerò tutte le luci
    g3d.resetLights();

    // Creo un oggetto trasformazione
    Transform trasformazioneTelecamera = new Transform();

    // Aggiungo la luce alla scena compresa di trasformazione
    g3d.addLight(luce, trasformazioneLuce);

    // Imposto una traslazione
    trasformazioneTelecamera.postTranslate(0.0f, 0.0f, 10.0f);

    initDone = true;
}
```

materialeCubo.setColor(Material.DIFFUSE, 0x00FF0000);

in pratica il primo parametro può assumere uno dei seguenti valori *AMBIENT*, *DIFFUSE*, *EMISSIVE* e *SPECULAR*. Nel caso di *SPECULAR* dobbiamo indicare anche l'indice di riflessività tramite il metodo *setShininess()*.

Le impostazioni del materiale vanno passate all'oggetto *Appearance*

apparenzaCubo.setMaterial(materialeCubo);

Ora è possibile vedere i risultati delle modifiche, mostrati in **Figura 2**, lanciando l'applicazione. Il **Listato 2** riporta il codice completo del metodo *Init()*.

Il texture mapping

Abbiamo appena visto alcune tecniche per modificare l'aspetto degli oggetti. Questi metodi però producono risultati un po' artificiali. Potremmo aggiungere maggiore realismo utilizzando delle immagini per riempire i poligoni.

Il procedimento per specificare le coordinate sulle *texture*

LISTATO 3 Il listato completo delle modifiche per la texture mapping

```

public void Init() {
    // Creo una nuova istanza dell'oggetto VertexBuffer
    datiVerticiCubo = new VertexBuffer();

    // Creo un nuovo VertexArray per memorizzare i
    // le posizioni dei vertici del cubo
    VertexArray posizioniVertici = new VertexArray(POSIZIONE_VERTICI.length
                                                    / 3, 3, 1);

    // Memorizzo le posizioni dei vertici nell'array
    posizioniVertici.set(0, POSIZIONE_VERTICI.length / 3, POSIZIONE_VERTICI);

    // Passo il riferimento dell'VertexArray al VertexBuffer
    datiVerticiCubo.setPosition(posizioniVertici, 1.0f, null);

    // Creo una nuova istanza dell'oggetto TriangleStripArray
    triangoliCubo = new TriangleStripArray(INDICI_TRIANGLEOLI, LUNGHEZZE_TRIANGLEOLI);

    // Creo una nuova istanza dell'oggetto Camera
    Camera telecamera = new Camera();

    // Calcolo il valore dell'aspectRatio
    float aspectRatio = (float) LARGHEZZA_DISPLAY / (float) ALTEZZA_DISPLAY;

    // Creo una nuova istanza dell'oggetto Appearance
    apparenzaCubo = new Appearance();

    // Creo una nuova istanza dell'oggetto PolygonMode
    modalitaPoligono = new PolygonMode();

    // Imposto la modalità di disegno dei poligoni
    apparenzaCubo.setPolygonMode(modalitaPoligono);

    // Imposto la correzione prospettica
    telecamera.setPerspective(30.0f, aspectRatio, 1.0f, 1000.0f);

    // Creo un oggetto trasformazione
    Transform trasformazioneTelecamera = new Transform();

    // Imposto una traslazione
    trasformazioneTelecamera.postTranslate(0.0f, 0.0f, 10.0f);

    // Attivo la telecamera
    g3d.setCamera(telecamera, trasformazioneTelecamera);

    // Creo un nuovo VertexArray per memorizzare le normali
    VertexArray normaliCubo = new VertexArray(NORMALI_VERTICI.length /
                                                3, 3, 1);

    // Memorizzo le normali nel VertexArray
    normaliCubo.set(0, NORMALI_VERTICI.length / 3, NORMALI_VERTICI);

    // Creo un nuovo oggetto Light
    Light luce = new Light();

    // Imposto il tipo di fonte d'illuminazione
    luce.setMode(Light.ON);

    // Imposto l'intensità della luce
    luce.setIntensity(5.0f);

    // Creo una nuova trasformazione per la luce
    Transform trasformazioneLuce = new Transform();

    // Imposto il valore di traslazione
    trasformazioneLuce.postTranslate(0.0f, 0.0f, 3.0f);

    // Azzerò tutte le luci
    g3d.resetLights();

    // Aggiungo la luce alla scena compresa di trasformazione
    g3d.addLight(luce, trasformazioneLuce);

    VertexArray datiTexture = new VertexArray(COORDINATE_TEXTURE.length
                                                / 2, 2, 1);

    datiTexture.set(0, COORDINATE_TEXTURE.length / 2, COORDINATE_TEXTURE);

    datiVerticiCubo.setTexCoords(0, datiTexture, 1.0f, null);

    try {
        Image2D imgTexture = (Image2D) Loader.load(FILE_TEXTURE)[0];
        Texture2D textureCubo = new Texture2D(imgTexture);
        textureCubo.setWrapping(Texture2D.REPEAT);
        apparenzaCubo.setTexture(0, textureCubo);
    } catch (Exception e) {
        // Gestione dell'errore
    }

    initDone = true;
}

```

è il solito

```
private static final byte[] COORDINATE_TEXTURE = {
```

```

0, 1, 1, 1, 0, 0, 1, 0, // fronte
0, 1, 1, 1, 0, 0, 1, 0, // retro
0, 1, 1, 1, 0, 0, 1, 0, // destra
0, 1, 1, 1, 0, 0, 1, 0, // sinistra

```

```

0, 1, 1, 1, 0, 0, 1, 0, // alto
0, 1, 1, 1, 0, 0, 1, 0, // basso
};

```

ovviamente si deve fare riferimento ad un file di immagine che forma la *texture*

```
private static final String FILE_TEXTURE = "/texture.png");
```

è inutile ricordare di creare il *VertexArray*

```
VertexArray datTexture = new VertexArray(COORDINATE_TEXTURE.length / 2, 2, 1);
```

memorizzare i dati dell'array

```
datTexture.set(0, COORDINATE_TEXTURE.length / 2, COORDINATE_TEXTURE);
```

ed inserire le coordinate nella nuova texture

```
datVerticiCubo.setTexCoord(0, datTexture, 2.0f, null);
```

A questo punto carichiamo in memoria l'immagine "texture.png"

```
try
{
    Image2D imgTexture = (Image2D) Loader.load(FILE_TEXTURE)[0];
}
catch (Exception e)
{
    // Gestione dell'errore
}
```

e se il caricamento dell'immagine va a buon fine passiamo alla creazione della texture

```
Texture2D textureCubo = new Texture2D(imgTexture);
```

Quando si "attacca" una texture su di un poligono, normalmente, si vuole che in qualche modo vengano preservati i valori di colore generati dall'illuminazione. Quest'ultima, infatti, non lavora direttamente sulla texture, ma su un colore assegnato al poligono (o all'oggetto).

Il risultato dell'illuminazione di dato colore viene poi sommato all'immagine della texture secondo varie funzioni, come accade del resto per le OpenGL.

Il colore dell'oggetto è impostato tramite il metodo

```
datVerticiCubo.setDefaultColor(0x00FFFFFF);
```

si associa la texture all'Appearance con

```
apparenzaCubo.setTexture(0, textureCubo);
```

dove 0 rappresenta il numero della texture, La funzione di blending è attivata tramite *setBlending()* della classe *Texture2D*

```
textureCubo.setBlending(Texture2D.FUNC_DECAL);
```

La **Figura 3** mostra il cubo coperto dalla texture creata, mentre il **Listato 3** il codice completo a scopo di riferimento.

Conclusioni

Ovviamente lo speciale che avete letto non ha la pretesa di essere una guida completa alle pur semplici classi della API, ma sono certo che da questo punto siete in grado di muovervi da soli prendendo come riferimento la sola documentazione javadoc, scaricabile dal sito FTP

Si raccomanda di fare riferimento alla documentazione Javadoc contenuta sul sito FTP per maggiori dettagli sulle classi di M3G

Nell'immediato futuro avremo sicuramente la possibilità di approfondire la trattazione tramite esempi più completi e complessi, come la realizzazione di un vero videogioco 3D.

Spero anche che l'argomento presentato sia ragione di interesse e stimolo per dare vita ad idee interessanti nel mondo dei videogiochi per cellulari. A questo scopo invito tutti i lettori a contattarci per qualunque consiglio e idea vogliate sottoporci.

Bibliografia

- [1] Sun Corporation, "Mobile 3D Graphics API Specification", www.sun.com
- [2] DEV n°132 anno XIII Settembre 2005, "Utilizzare le OpenGL"

Download del progetto

Scarica il progetto completo di questo articolo all'indirizzo:
<ftp://ftp.infomedia.it/pub/DEV/Listati/Dev143/SP3.zip>

Mauro Pesci

Lavora come libero professionista nel settore dello sviluppo del software, in particolar modo, si sta occupando della realizzazione di alcuni videogiochi commerciali per telefonini distribuiti tramite il portale di H3G.