Java3D – Quick start.

Un rapidissimo test di funzionamento (Java 6 + Java3D 1.5).

Scaricate dal sito di java3d lo zip. Create una cartella a piacere per il tutorial. In quella cartella copiate i file

```
j3dcore-d3d.dll
j3dcore-ogl-cg.dll
j3dcore-ogl.dll
j3dcore.jar
j3dutils.jar
vecmath.jar
```

Nella stessa cartella create un file java di nome Main.java contenente il codice che segue:

```
package it.quickjava3d;

import java.awt.*;
import java.awt.event.*;
import javax.vecmath.*;
import javax.media.j3d.*;
import com.sun.j3d.utils.universe.SimpleUniverse;

public class Main {
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() { public void run() {
			startSample();
		}});
	}

	private static void startSample() {
		Canvas3D canvas = new Canvas3D(
			SimpleUniverse.getPreferredConfiguration());
		SimpleUniverse universe = new SimpleUniverse(canvas);
		canvas.setPreferredSize(new Dimension(400, 400));
		Frame window = new Frame("Java3D Sample");
		window.addWindowListener(jvmKiller);
		window.add(canvas);
		window.pack();
		window.setVisible(true);
	}

	private static WindowListener jvmKiller = new WindowAdapter() {
		public void windowClosing(WindowEvent e) {
			System.exit(0);
		}
	};
}
```

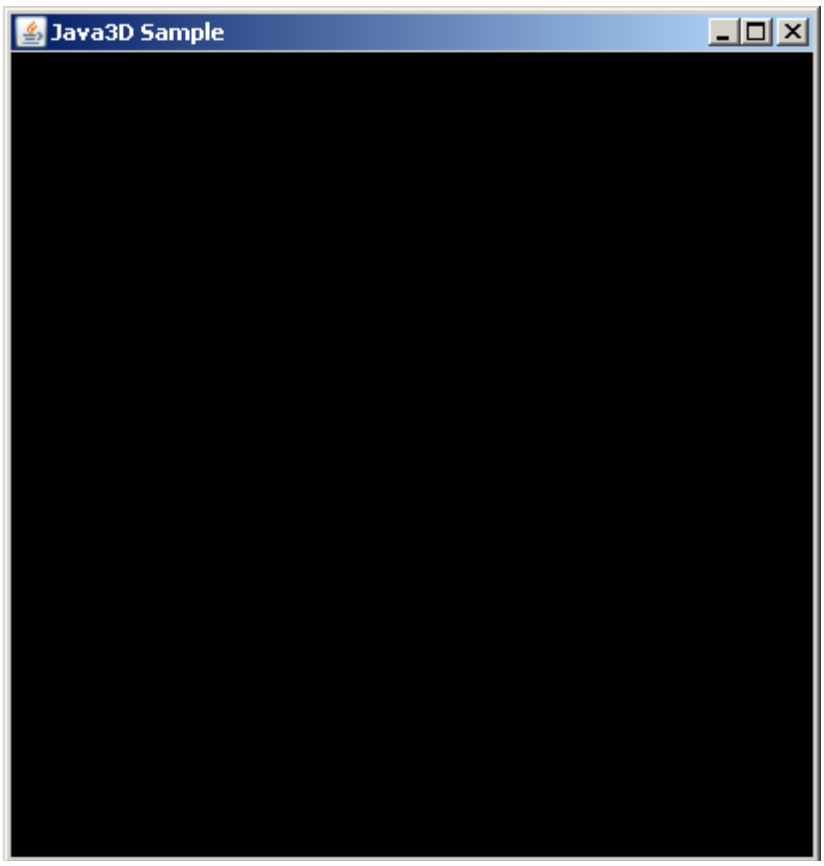
Compilate con:

```
javac -d . *.java
```

Eseguite il programma risultante con il comando:

```
java it.quickjava3d.Main
```

Il risultato è questo bel finestrone nero.



Se la compilazione o l'esecuzione dovessero rilasciare qualche eccezione di tipo “non trovo la classe X” avete un problema di classpath che, in Java 6, non dovreste avere. Provate a risolvere specificando il classpath sia in compilazione:

```
javac -d . -cp . *.java
```

che in esecuzione:

```
java -cp . it.quickjava3d.Main
```

L'ultima spiaggia è procedere all'installazione vera e propria di Java3D. Prendete i file citati all'inizio della sezione e spostate i jar nella cartella “ext” del JRE e del JDK, le dll nella cartella “lib”, sempre del JRE e del JDK. Lo stesso vale per Linux, OsX e Solaris (salvo estensioni).

Canvas3D.

Canvas3D è il componente AWT in cui è proiettata la rappresentazione dell'universo 3D. Esiste una controparter Swing, JCanvas3D, che al momento è sperimentale. Il fatto che Canvas3D sia un componente pesante comporta dei problemi nel caso in cui lo si voglia usare come parte di un'interfaccia Swing. Per via della diversa gestione dell'ordine Z, i componenti AWT si sovrappongono ai componenti Swing anche quando questi ultimi siano collocati di fronte ai primi. Per lo scopo di questo tutorial la questione non è di grande interesse. Quanto scritto nel main su visto coincide anche con tutto il codice AWT/Swing necessario all'applicazione. Il resto sarà Java3D. Volendo sfruttare il 3D in un'applicazione Swing o si usa JCanvas3D oppure si adottano degli accorgimenti ad hoc: è sufficiente ricordare che il problema di convivenza tra AWT e Swing, tra componenti pesanti e componenti leggeri, è limitato alla sovrapposizione dei primi ai secondi. O si evita questa sovrapposizione distribuendo opportunamente i componenti o, nel caso in cui questo non sia possibile, si usano componenti AWT al posto di quelli Swing per le parti sovrapposte.

L'inizializzazione di un Canvas3D richiede un insieme di parametri grafici. Nel codice della classe Main abbiamo usato come parametro il prodotto dell'invocazione:

```
SimpleUniverse.getPreferredConfiguration();
```

Questo metodo rileva la configurazione grafica del sistema di esecuzione e stabilisce i parametri ottimali per la creazione di un Canvas3D.

Sempre nella classe su riportata, dopo la creazione del Canvas3D abbiamo creato un SimpleUniverse. Il costruttore di SimpleUniverse prende come argomento un Canvas3D. Quello che fa SimpleUniverse è creare un insieme standard di oggetti necessari e sufficienti ad un uso tipico di Java3D. In particolare, SimpleUniverse crea una piattaforma visiva trasformabile e un albero di proiezione dei contenuti 3D. La piattaforma visiva è il punto a cui è agganciata la "telecamera", la vista sul mondo tridimensionale. L'albero di proiezione è la struttura dati in cui sono inseriti i vari pezzi dell'universo 3D. Il motore di proiezione di Java3D analizza periodicamente questo albero e, secondo la posizione e le caratteristiche della "vista", connessa alla piattaforma visiva, proietta il risultato di questa analisi nel Canvas3D.

Dove lo metto, dove lo metto.

SimpleUniverse consente di connettere un oggetto Java3D all'albero di proiezione attraverso il metodo

```
addBranchGraph(BranchGroup g);
```

Proviamo ad usare questo metodo per aggiungere qualcosa alla nostra scena 3d. Modifichiamo la classe Main aggiungendo la linea in rosso:

```
package it.quickjava3d;

import java.awt.*;
import java.awt.event.*;
import javax.vecmath.*;
import javax.media.j3d.*;
import com.sun.j3d.utils.universe.SimpleUniverse;

public class Main {
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() { public void run() {
			startSample();
		}});
	}

	private static void startSample() {
		Canvas3D canvas = new Canvas3D(
			SimpleUniverse.getPreferredConfiguration());
		SimpleUniverse universe = new SimpleUniverse(canvas);
		new MyWorld().create(universe);
		canvas.setPreferredSize(new Dimension(400, 400));
		Frame window = new Frame("Java3D Sample");
		window.addWindowListener(jvmKiller);
		window.add(canvas);
		window.pack();
		window.setVisible(true);
	}

	private static WindowListener jvmKiller = new WindowAdapter() {
		public void windowClosing(WindowEvent e) {
			System.exit(0);
		}
```

```
	};
}
```

Ora creiamo una classe MyWorld e dotiamola di un metodo create(SimpleUniverse u).

```
package it.quickjava3d;

import javax.vecmath.*;
import javax.media.j3d.*;
import com.sun.j3d.utils.universe.*;

public class MyWorld {

	public void create(SimpleUniverse universe) {
		...scrivere qui il codice che presenteremo in seguito...
	}
}
```

All'interno di questo metodo scriveremo il necessario per colorare lo sfondo della scena 3D. Creiamo un BranchGroup.

```
BranchGroup universeBranch = new BranchGroup();
```

Nel nostro esempio usiamo un BranchGroup perchè il metodo addBranchGraph di SimpleUniverse richiede un BranchGroup. Occorre comunque una breve descrizione di cosa sia un BranchGroup.

La maggior parte degli elementi presenti in Java3D sono nodi, nel senso di nodo di un albero. Esiste un'apposita classe Node a testimonianza di questo. I nodi sono pezzi dell'albero di proiezione. Alcuni di questi nodi possono contenere altri nodi. Questi nodi, in Java3D, sono dei gruppi. Anche qui, c'è un'apposita classe Group. Ci sono molti tipi di Node e molti tipi di Group. Ognuno ha la sua specialità. Quella di BranchGroup è la connessione dinamica.

Tutti i Gruppi, in quanto tali, possono subire l'inserimento di figli quando l'albero di proiezione è “morto”. L'albero di proiezione è “morto” finchè non viene dato in pasto al motore di proiezione. Quando l'albero è nella disponibilità del motore di proiezione, esso è definito “vivo”. Per traslato, un nodo connesso ad un albero di proiezione “vivo” è considerato anch'esso “vivo”. Un nodo connesso ad un albero di proiezione “morto” o non connesso affatto è considerato “morto”.

Un BranchGroup, opportunamento configurato, può subire l'inserimento di figli anche quando sia vivo.

Ora creiamo lo sfondo. Anche lo sfondo è un nodo.

```
Background bg = new Background(0.5f, 0.5f, 0.5f);
```

Per creare lo sfondo abbiamo usato il costruttore di Background che richiede come argomenti tre valori float. Questi valori sono i componenti rosso, verde e blu del colore di riempimento. I valori di questa tripletta RGB possono variare da zero a uno. La tripletta (1, 1, 1) definisce il colore bianco, il suo opposto (0, 0, 0) è nero. Su abbiamo usato un grigio.

Aggiungiamo il nodo Background al nodo BranchGroup:

```
universeBranch.addChild(bg);
```

Non sempre è sufficiente aggiungere un nodo ad un gruppo connesso (o che sarà connesso, nel nostro caso) ad un albero di proiezione perchè il nodo abbia effetto. Alcuni nodi, in effetti, richiedono particolari impostazioni aggiuntive. Background è uno di questi. Affinchè un

Background sia “visibile” occorre indicargli quale volume dovrà essere influenzato dal suo effetto.

Java3D offre una categoria di oggetti ad hoc per la definizione di volumi, Bounds. Tra i diversi Bounds disponibili abbiamo un volume sferico, BoundingSphere. Creando un BoundingSphere e passandolo al Background come area di applicazione lo sfondo avrà modo di influenzare l'universo virtuale.

```
BoundingSphere infinity = new BoundingSphere(
      new Point3d(),
      Double.POSITIVE_INFINITY);
```

Il volume su creato è una sfera di raggio infinito. Applicato al Background:

```
bg.setApplicationBounds(infinity);
```

causa l'effettività del nostro sfondo in ogni punto dell'universo. I termini “volume di applicazione” o “volume di influenza” in Java3D significano che il tal nodo produrrà i suoi effetti quando il centro della piattaforma visiva si troverà all'interno di quel volume.

Da ultimo, colleghiamo il nodo BranchGroup all'albero di proiezione.

```
universe.addBranchGraph(universeBranch);
```

Ricapitolando, così appare la classe MyWorld:

```
package it.quickjava3d;

import javax.vecmath.*;
import javax.media.j3d.*;
import com.sun.j3d.utils.universe.*;

public class MyWorld {

      public void create(SimpleUniverse universe) {
            BranchGroup universeBranch = new BranchGroup();
            Background bg = new Background(0.5f, 0.5f, 0.5f);
            universeBranch.addChild(bg);
            BoundingSphere infinity = new BoundingSphere(
                  new Point3d(),
                  Double.POSITIVE_INFINITY);
            bg.setApplicationBounds(infinity);
            universe.addBranchGraph(universeBranch);
      }
}
```

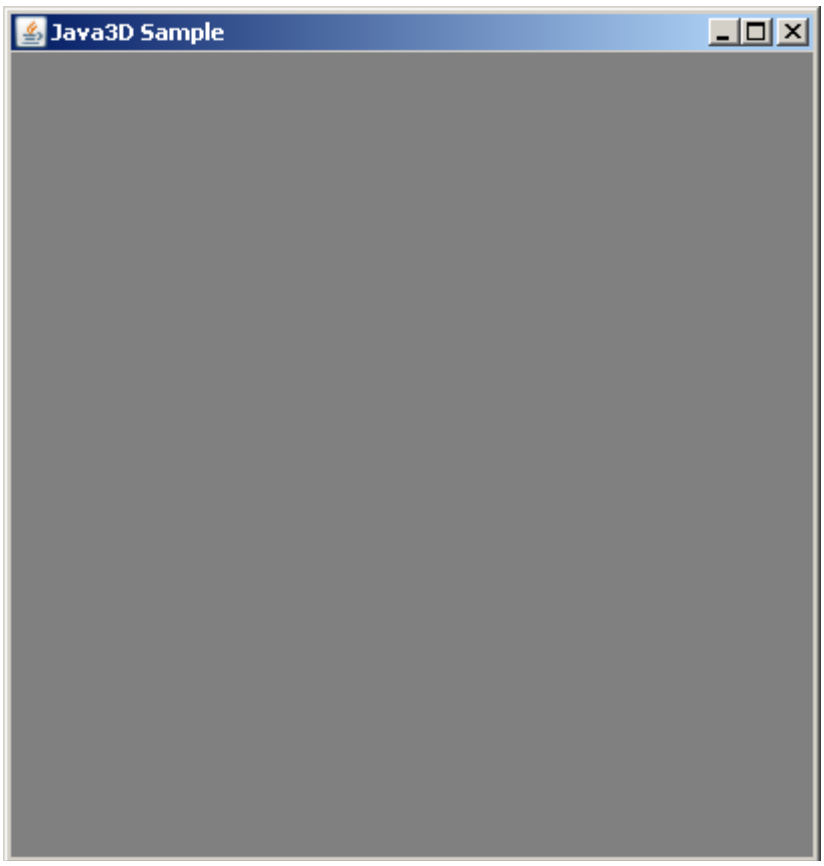
Compilando:

```
javac -d . *.java
```

ed eseguendo:

```
java it.quickjava3d.Main
```

apparirà sullo schermo un bel finestrone grigio:



Ora un esperimento. In MyWorld.java, spostiamo la linea

```
universe.addBranchGraph(universeBranch);
```

dall'ultima posizione in seconda fila:

```
	public void create(SimpleUniverse universe) {
		BranchGroup universeBranch = new BranchGroup();
		universe.addBranchGraph(universeBranch);
		Background bg = new Background(0.5f, 0.5f, 0.5f);
```

In pratica anziché aggiungere il nostro BranchGroup all'albero di proiezione dopo aver aggiungo il Background al BranchGroup, prima aggiungiamo il BranchGroup e poi aggiungiamo il Background. Ricompilando e tentando di eseguire otteniamo questa eccezione:

```
Exception in thread "AWT-EventQueue-0"
javax.media.j3d.RestrictedAccessException
: Group: only a BranchGroup node may be added
        at javax.media.j3d.Group.addChild(Group.java:265)
        at it.quickjava3d.MyWorld.create(MyWorld.java:15)
        at it.quickjava3d.Main.startSample(Main.java:20)
        at it.quickjava3d.Main.access$000(Main.java:9)
        at it.quickjava3d.Main$1.run(Main.java:12)
        at java.awt.event.InvocationEvent.dispatch(Unknown Source)
        at java.awt.EventQueue.dispatchEvent(Unknown Source)
        at java.awt.EventDispatchThread.pumpOneEventForFilters(Unknown Source)
        at java.awt.EventDispatchThread.pumpEventsForFilter(Unknown Source)
        at java.awt.EventDispatchThread.pumpEventsForHierarchy(Unknown Source)
        at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
        at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
        at java.awt.EventDispatchThread.run(Unknown Source)
```

Quello che è successo è questo. Il BranchGroup diventa vivo nel momento in cui è connesso all'albero di proiezione. A questo punto noi aggiungiamo al BranchGroup un nodo, il Background.

Java3D si ferma e ci dice che se un gruppo è vivo, in quanto connesso all'albero di proiezione, allora non possiamo aggiungergli altri nodi a meno che questi nodi non siano dei BranchGroup.

Sempre nell'ottica di voler inserire il nostro Background in un nodo "vivo", com'è il nostro BranchGroup universeBranch per via della linea universe.addBranchGraph(universeBranch) che precede la linea universeBranch.addChild(bg), seguendo la traccia dell'errore potremmo pensare che la soluzione stia nell'inserire il Background in un BranchGroup e poi inserire questo BranchGroup in universeBranch. Così facendo rispetteremmo l'indicazione secondo cui solo un BranchGroup può essere aggiunto ad un gruppo vivo. Proviamo:

```
	public void create(SimpleUniverse universe) {
		BranchGroup universeBranch = new BranchGroup();
		universe.addBranchGraph(universeBranch);

		Background bg = new Background(0.5f, 0.5f, 0.5f);
		BoundingSphere infinity = new BoundingSphere(
			new Point3d(),
			Double.POSITIVE_INFINITY);
		bg.setApplicationBounds(infinity);

		BranchGroup insertionGroup = new BranchGroup();
		insertionGroup.addChild(bg);

		universeBranch.addChild(insertionGroup);
	}
```

E otteniamo una diversa eccezione:

```
Exception in thread "AWT-EventQueue-0"
javax.media.j3d.CapabilityNotSetException
: Group: no capability to append children
        at javax.media.j3d.Group.addChild(Group.java:268)
        at it.quickjava3d.MyWorld.create(MyWorld.java:24)
        at it.quickjava3d.Main.startSample(Main.java:20)
        at it.quickjava3d.Main.access$000(Main.java:9)
        at it.quickjava3d.Main$1.run(Main.java:12)
        at java.awt.event.InvocationEvent.dispatch(Unknown Source)
        at java.awt.EventQueue.dispatchEvent(Unknown Source)
        at java.awt.EventDispatchThread.pumpOneEventForFilters(Unknown Source)
        at java.awt.EventDispatchThread.pumpEventsForFilter(Unknown Source)
        at java.awt.EventDispatchThread.pumpEventsForHierarchy(Unknown Source)
        at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
        at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
        at java.awt.EventDispatchThread.run(Unknown Source)
```

Per quanto possa sembrare bizzarro, qui sta il punto. Capacità non impostata. Ogni nodo ha delle particolari abilità. Quella di BranchGroup è, come detto, la connessione dinamica. Le abilità dei nodi sono, in un certo senso, latenti. I nodi "speciali" possono usare le loro particolari abilità solo da morti. Nel momento in cui essi diventano vivi, l'uso di quelle stesse capacità richiede una dichiarazione preventiva che suona come "sfrutterò le mie abilità particolari anche quando sarò vivo". Una ragionevole supposizione vuole che questo doppio binario sia usato da Java3D per ottimizzare la proiezione. I nodi che pur potendo non dichiarano di voler usare le proprie abilità in vita sono con ogni probabilità trattati in modo più efficiente di quelli che possono e dichiarano di voler usare quelle stesse abilità.

Per noi questo significa che se vogliamo poter aggiungere un BranchGroup ad un altro BranchGroup quando quest'ultimo è in vita dobbiamo specificare che il BranchGroup ricevente userà anche in vita la sua capacità di subire l'inserimento di figli.

```
	public void create(SimpleUniverse universe) {
		BranchGroup universeBranch = new BranchGroup();
		universeBranch.setCapability(BranchGroup.ALLOW_CHILDREN_EXTEND);
```

```
        universe.addBranchGraph(universeBranch);

        Background bg = new Background(0.5f, 0.5f, 0.5f);
        BoundingSphere infinity = new BoundingSphere(
              new Point3d(),
              Double.POSITIVE_INFINITY);
        bg.setApplicationBounds(infinity);

        BranchGroup insertionGroup = new BranchGroup();
        insertionGroup.addChild(bg);

        universeBranch.addChild(insertionGroup);
    }
```

La necessità di specificare espressamente le capacità di cui un certo nodo farà uso in vita ricorre per ogni nodo. Occorre quindi tenere sempre presente che se si desidera far uso di una particolare abilità di un nodo anche quando questo nodo sarà vivo allora sarà sempre necessario specificare questo fatto attraverso una dichiarazione d'uso della relativa capacità.

Un lavoro fatto col cubo.

Azzeriamo il contenuto del metodo create(SimpleUniverse) in MyWorld. Quello che segue è il contenuto di un file cubo.obj

```
# Created with Anim8or 0.95
# Object "object01":
# ComRec:
g mesh01
# No. points 8:
v -0.50000 -0.50000 -0.50000
v -0.50000 -0.50000 0.50000
v -0.50000 0.50000 -0.50000
v -0.50000 0.50000 0.50000
v 0.50000 -0.50000 -0.50000
v 0.50000 -0.50000 0.50000
v 0.50000 0.50000 -0.50000
v 0.50000 0.50000 0.50000

# No. normals 30:
vn 0 0 -1
vn 0 0 1
vn -1 0 0
vn 1 0 0
vn 0 1 0
vn 0 -1 0
vn 0 0 -1
vn 0 -1 0
vn -1 0 0
vn 0 0 1
vn 0 -1 0
vn -1 0 0
vn 0 0 -1
vn 0 1 0
vn -1 0 0
vn 0 0 1
vn 0 1 0
vn -1 0 0
vn 0 0 -1
vn 0 -1 0
vn 1 0 0
vn 0 0 1
vn 0 -1 0
vn 1 0 0
vn 0 0 -1
vn 0 1 0
```

```
vn 1 0 0
vn 0 0 1
vn 0 1 0
vn 1 0 0

# No. texture coordinates 8:
vt 0 0
vt 0 0
vt 0 1
vt 0 1
vt 1 0
vt 1 0
vt 1 1
vt 1 1

# No. faces 6:
f 3/3/13 7/7/25 5/5/19 1/1/7
f 6/6/22 8/8/28 4/4/16 2/2/10
f 2/2/12 4/4/18 3/3/15 1/1/9
f 7/7/27 8/8/30 6/6/24 5/5/21
f 4/4/17 8/8/29 7/7/26 3/3/14
f 5/5/20 6/6/23 2/2/11 1/1/8
```

Il testo termina con una linea vuota. E' un cubo di lato unitario centrato rispetto all'origine degli assi. Prendete questo testo e salvatelo in un file di nome “cubo.obj”, nella cartella del tutorial.

Ora caricheremo questo cubo nell'universo Java3D e vedremo... un bel niente. Scopriremo il perchè e vi porremo rimedio.

Per caricare singoli oggetti o interi ambiente 3d prodotti con strumenti quali ArtOfIllusion, Anim8or o i magnifici Maya, 3DStudio Max, Lightwave eccetera, che costano una sacco di sfrugullioni e chissa come ce li hanno tutti quanti, si possono usare i Loader. I loader sono degli utili accessori di Java3D che, come dice il nome, caricano. Java3D contiene un Loader predefinito che carica oggetti Alias Wavefront, da file in formato obj del genere su riportato. Per usare le capacità di questo loader importiamo due package:

```
import com.sun.j3d.loaders.*;
import com.sun.j3d.loaders.objectfile.*;
```

Il primo è quello del “framework”, il secondo è una specifica definizione per i file obj. Di caricatori ce ne sono a bizzeffe, per ogni formato possibile, immaginabile e desiderabile. Per caricare il cubo precedentemente salvato nel file cubo.obj, occorrono poche righe. La prima dichiara un Loader e lo indirizza ad un loader ObjectFile.

```
Loader loader = new ObjectFile();
```

La seconda dichiara uno “Scene”. Scene è un contenitore per gli oggetti che un caricatore produce a seguito dell'analisi del file caricato.

```
Scene scene;
```

Dopodichè si passa al caricamento effettivo che assume vesti note.

```
try {
      scene = loader.load(getClass().getResource(“/cubo.obj”));
} catch(Exception ex) {
      throw new RuntimeException(ex);
}
```

Ora la “scena” contiene tutti gli elementi che il caricatore ha estratto dal file. Tali elementi sono

stati tradotti in nodi Java3D. Scene non è di per sé un nodo ma contiene un BranchGroup a cui è connesso ogni altro nodo caricato.

```
BranchGroup sceneGroup = scene.getSceneGroup();
```

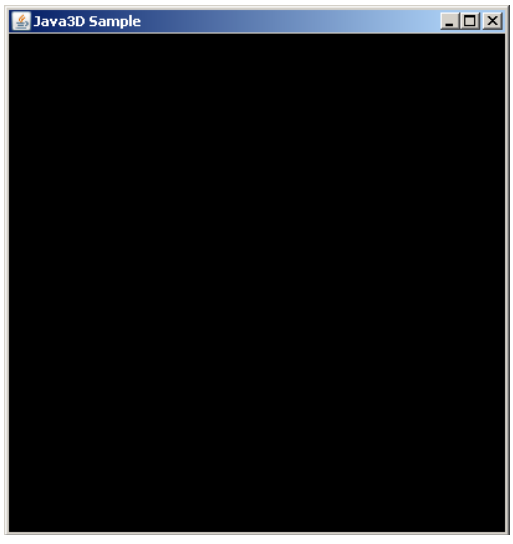
Possiamo usare questo sceneGroup come argomento del metodo addBranchGraph di SimpleUniverse, inserendo pertanto il cubo nel nostro universo Java3D.

```
universe.addBranchGraph(sceneGroup);
```

La classe MyWorld appare quindi riscritta:

```
package it.quickjava3d;

import javax.vecmath.*;
import javax.media.j3d.*;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.loaders.objectfile.*;
import com.sun.j3d.loaders.*;

public class MyWorld {

	public void create(SimpleUniverse universe) {
		Loader loader = new ObjectFile();
		Scene scene;
		try {
			scene = loader.load(getClass().getResource("/cubo.obj"));
		} catch(Exception ex) {
			throw new RuntimeException(ex);
		}
		BranchGroup sceneGroup = scene.getSceneGroup();
		universe.addBranchGraph(sceneGroup);
	}
}
```

Compilando ed eseguendo appare una vecchia conoscenza:



Il finestrone nero. Il cubo c'è, ma non si vede. Per un po' di ragioni. La prima è che la scena manca di illuminazione. La seconda è che il cubo non illuminato appare nero. La terza è che non c'è un colore di sfondo, il che significa che è usato il nero e quindi il cubo non si distingue dallo sfondo. La quarta e ultima è che la telecamera è collocata al centro degli assi. E' dentro al cubo. Normalmente se la telecamera si trova dentro ad un solido il solido non appare a meno che la geometria di quel solido non specifichi il disegno di entrambe le superfici delle sue facce. Una questione che a noi non interessa.

Risolviamo questi problemi uno alla volta. Aggiungiamo un colore di sfondo. Sappiamo come fare: serve un background. Aggiungiamo dopo l'ultima riga di codice del metodo create in MyWorld:

la creazione di un Background, bianco:

```
Background bg = new Background(1, 1, 1);
```

la creazione di un volume sferico, assegnato al background come volume di applicazione:

```
BoundingSphere infinity = new BoundingSphere(
      new Point3d(), Double.POSITIVE_INFINITY);
bg.setApplicationBounds(infinity);
```

la creazione di un BranchGroup:

```
BranchGroup root = new BranchGroup();
```

in cui inseriamo lo sfondo:

```
root.addChild(bg);
```

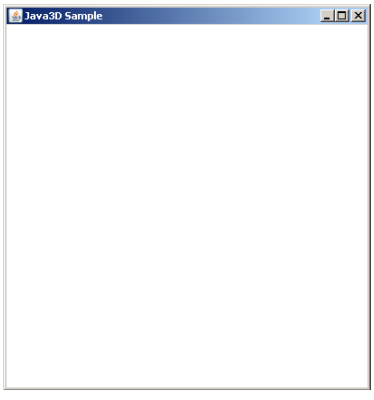
e che infine connettiamo all'albero di proiezione di “universe”:

```
universe.addBranchGraph(root);
```

Il metodo create di MyWorld sarà quindi:

```
      public void create(SimpleUniverse universe) {
            Loader loader = new ObjectFile();
            Scene scene = null;
            try {
                  scene = loader.load(getClass().getResource("/cubo.obj"));
            } catch(Exception ex) {
                  throw new RuntimeException(ex);
            }
            BranchGroup sceneGroup = scene.getSceneGroup();
            universe.addBranchGraph(sceneGroup);

            Background bg = new Background(1, 1, 1);
            BoundingSphere infinity = new BoundingSphere(
                  new Point3d(), Double.POSITIVE_INFINITY);
            bg.setApplicationBounds(infinity);
            BranchGroup root = new BranchGroup();
            root.addChild(bg);
            universe.addBranchGraph(root);
      }
```

La parte verde scuro è quella che abbiamo appena aggiunto. Compilando ed eseguendo otteniamo una finestra bianca.



Niente cubo, ancora. L'impostazione di uno sfondo bianco risolve il problema della fusione tra il cubo, nero, e lo sfondo, nero. Ora dobbiamo solo spostare la telecamera in modo tale che guardi non da dentro il cubo verso l'esterno ma dall'esterno verso il cubo. Vedremo in seguito l'altra soluzione: spostare il cubo più avanti. Iniziamo con la telecamera.

La telecamera, che in verità si chiama View, vista, è connessa ad una piattaforma di visualizzazione, ViewingPlatform. Tra questa piattaforma e la vista c'è uno strato intermedio che, con poca fantasia, è stato chiamo piattaforma visiva, ViewPlatform. La piattaforma visiva è connessa alla piattaforma di visualizzazione tramite un gruppo di trasformazione, che è un nodo di tipo TransformGroup.

La peculiarità di un gruppo di trasformazione sta nella sua capacità di trasformare lo spazio di coordinate. Meglio sarebbe dire che un gruppo di trasformazione definisce un sottospazio di coordinate che è il prodotto dell'applicazione di una trasformazione al suo superspazio. Ma cerchiamo di essere pratici: un TransformGroup consente di spostare, scalare o proiettare i nodi che contiene. Importante: causa la trasformazione del suo contenuto. Per spostare un gruppo di trasformazione bisogna aggiungerlo ad un altro gruppo di trasformazione.

Per prima cosa otteniamo il gruppo di trasformazione della vista.

```
ViewingPlatform viewingPlatform = universe.getViewingPlatform();
TransformGroup viewPlatformTransformGroup =
      viewingPlatform.getViewPlatformTransform();
```

Ora applichiamo uno spostamento al contenuto di questo gruppo. Possiamo farlo in due modi. Uno è uno spostamento assoluto e si può fare così.

Creiamo un vettore che contiene la posizione in cui vogliamo collocare la vista/telecamera:

```
Vector3d translation = new Vector3d(0, 0, 2);
```

Creiamo una trasformazione tridimensionale o Transform3D. Un Transform3D contiene i valori che un TransformGroup usa per alterare il sistema di coordinate in cui si trova e generare il sottospazio a cui abbiamo accennato prima.

```
Transform3D newTransform = new Transform3D();
```

Tutti i valori di un nuovo Transform3D sono “impostati a zero” (non in senso numerico: numericamente sono i valori di una trasformazione identica). E' come dire che questi dati, applicati ad un gruppo di trasformazione, non modificano di una virgola lo spazio di coordinate a cui appartiene il TransformGroup. Modifichiamo questi dati in modo tale che essi rappresentino la trasformazione necessaria per uno spostamento in direzione 0, 0, 1 di 2 unità (questo il significato del vettore precedentemente creato).

```
newTransform.setTranslation(translation);
```

Infine rifiliamo questi dati al gruppo di trasformazione:

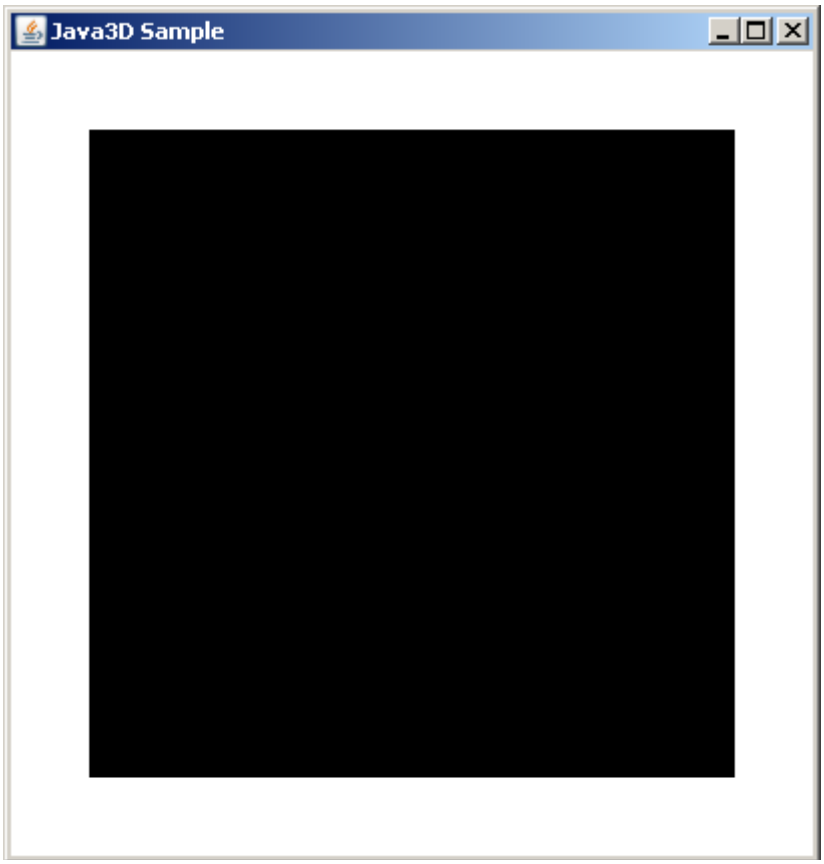
```
viewPlatformTransformGroup.setTransform(newTransform);
```

Il risultato è che la vista, nodo appartenete al gruppo di trasformazione viewPlatform-TransformGroup, risulterà spostata in direzione 0, 0, 1, di 2 unità. Il metodo create(SimpleUniverse) è ora così fatto:

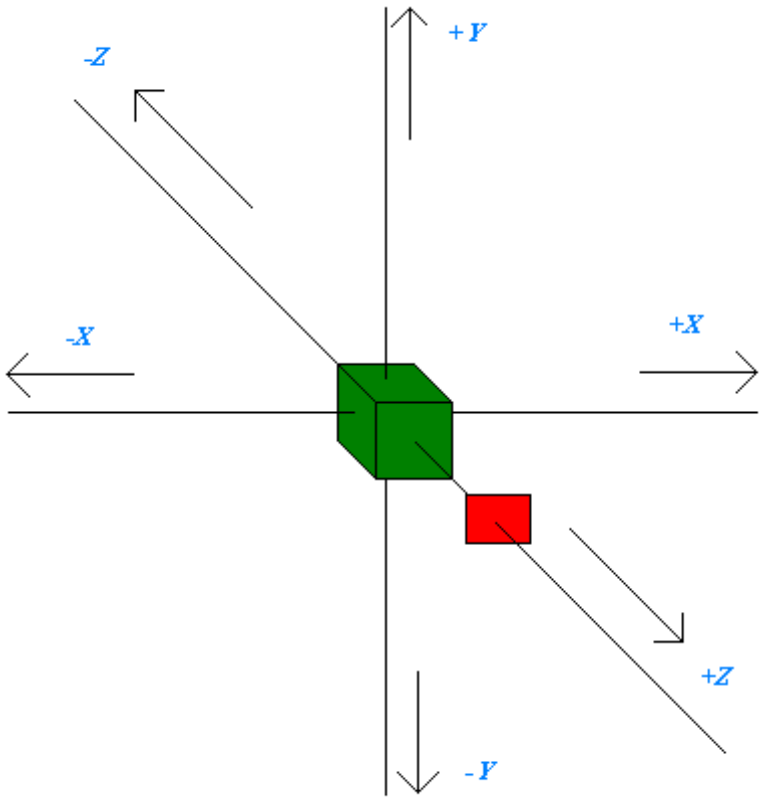
```
      public void create(SimpleUniverse universe) {
            Loader loader = new ObjectFile();
```

```
        Scene scene = null;
        try {
              scene = loader.load(getClass().getResource("/cubo.obj"));
        } catch(Exception ex) {
              throw new RuntimeException(ex);
        }
        BranchGroup sceneGroup = scene.getSceneGroup();
        universe.addBranchGraph(sceneGroup);

        Background bg = new Background(1, 1, 1);
        BoundingSphere infinity = new BoundingSphere(
              new Point3d(), Double.POSITIVE_INFINITY);
        bg.setApplicationBounds(infinity);
        BranchGroup root = new BranchGroup();
        root.addChild(bg);
        universe.addBranchGraph(root);

        ViewingPlatform viewingPlatform = universe.getViewingPlatform();
        TransformGroup viewPlatformTransformGroup =
              viewingPlatform.getViewPlatformTransform();

        Vector3d translation = new Vector3d(0, 0, 2);
        Transform3D newTransform = new Transform3D();
        newTransform.setTranslation(translation);
        viewPlatformTransformGroup.setTransform(newTransform);
  }
```

Compilando ed eseguendo otteniamo un finestrone bianco con la centro un quadratone nero.



Che significa spostamento in direzione (0, 0, 1) di 2 unità?. L'effetto l'abbiamo visto. La telecamera si è “tirata indietro”. Un po' più precisamente, la telecamera si è spostata verso chi guarda il monitor. Esattamente, la telecamera si è spostata nella direzione “+Z” indicata nello schema che segue:



In verde è rappresentato il nostro cubo. Il rosso la posizione della telecamera. La telecamera guarda verso il cubo. Il sistema di coordinate di Java3D è destrorso. Vi risparmio la ragione per gentilezza verso le articolazioni delle dita della vostra mano destra.

Abbiamo visto un modo di “spostare le cose”. In pratica abbiamo creato una nuova trasformazione e l'abbiamo applicata scartando quella vecchia. Si può fare diversamente. Possiamo prendere la trasformazione in atto e combinarla con una nuova in modo tale da ottenere uno spostamento relativo al precedente spazio di coordinate.

Prendiamo da un altro verso. Immaginate che, per effetto di una precedente trasformazione, la nostra telecamera fosse stata leggermente ruotata lungo l'asse Y. “Sbirciava” l'oggetto con la coda dell'occhio. Lo spostamento applicato nel modo su visto avrebbe azzerato quella rotazione.

La conservazione delle trasformazioni precedenti si ottiene premoltiplicando o postmoltiplicando i dati di due trasformazioni, quella che si vuole applicare e quella attualmente usata dal gruppo di trasformazione. Vediamo come.

Partiamo dalla stessa operazione di prima: ricaviamo il gruppo di trasformazione della vista:

```
ViewingPlatform viewingPlatform = universe.getViewingPlatform();
TransformGroup viewPlatformTransformGroup =
      viewingPlatform.getViewPlatformTransform();
```

Creiamo un Trasform3D:

```
Transform3D oldTransform = new Transform3D();
```

Immettiamo in questo Transform3D i dati di trasformazione correntemente usati dal gruppo di trasformazione della vista:

```
viewPlatformTransformGroup.getTransform(oldTransform);
```

Ora oldTransform contiene i valori che il TransformGroup viewPlatformTransformGroup sta applicando. Procediamo come prima, creando un vettore per lo spostamento:

```
Vector3d translation = new Vector3d(0, 0, 2);
```

Creiamo un nuovo Transform3D:

```
Transform3D newTransform = new Transform3D();
```

e impostiamo su di esso i valori dello spostamento:

```
newTransform.setTranslation(translation);
```

Ora moltiplichiamo il "vecchio" Transform3D con il "nuovo":

```
oldTransform.mul(newTransform);
```

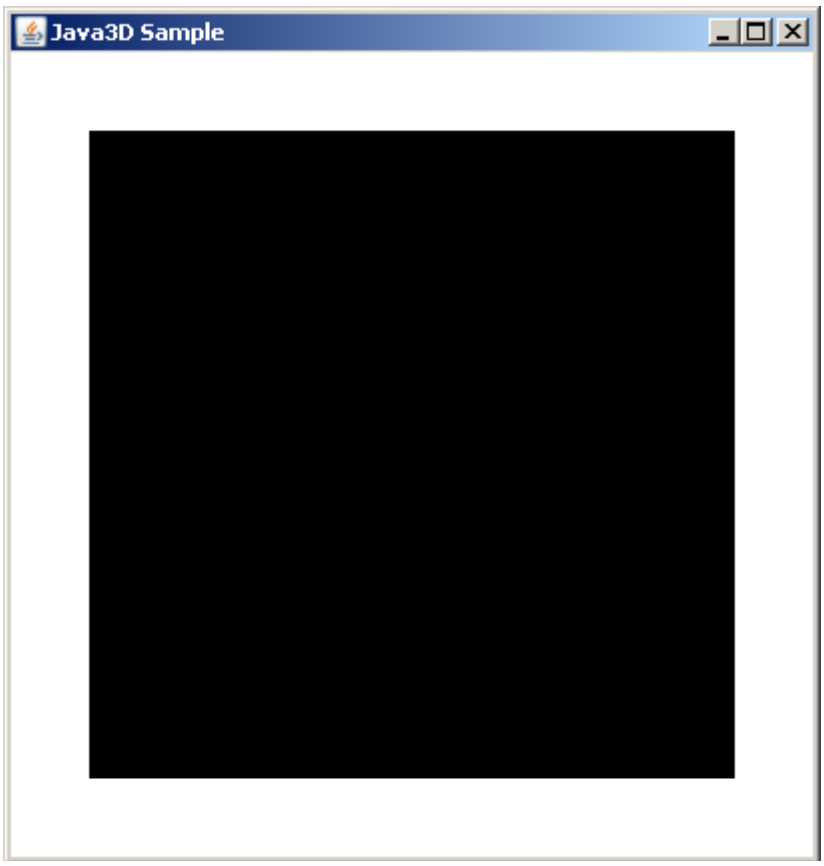
Infine applichiamo nuovamente il "vecchio" Transform3D al gruppo di trasformazione della vista.

```
viewPlatformTransformGroup.setTransform(oldTransform);
```

Il codice completo del metodo create(SimpleUniverse) è:

```
	public void create(SimpleUniverse universe) {
		Loader loader = new ObjectFile();
		Scene scene = null;
		try {
			scene = loader.load(getClass().getResource("/cubo.obj"));
		} catch(Exception ex) {
			throw new RuntimeException(ex);
		}
		BranchGroup sceneGroup = scene.getSceneGroup();
		universe.addBranchGraph(sceneGroup);

		Background bg = new Background(1, 1, 1);
		BoundingSphere infinity = new BoundingSphere(
			new Point3d(), Double.POSITIVE_INFINITY);
		bg.setApplicationBounds(infinity);
		BranchGroup root = new BranchGroup();
		root.addChild(bg);
		universe.addBranchGraph(root);

		ViewingPlatform viewingPlatform = universe.getViewingPlatform();
		TransformGroup viewPlatformTransformGroup =
			viewingPlatform.getViewPlatformTransform();

		Transform3D oldTransform = new Transform3D();
		viewPlatformTransformGroup.getTransform(oldTransform);

		Vector3d translation = new Vector3d(0, 0, 2);
		Transform3D newTransform = new Transform3D();
		newTransform.setTranslation(translation);

		oldTransform.mul(newTransform);

		viewPlatformTransformGroup.setTransform(oldTransform);
	}
```

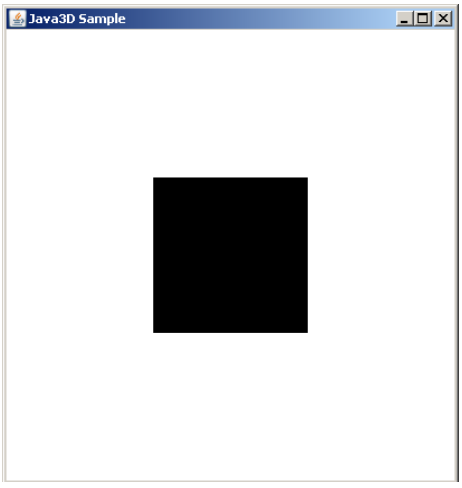
Il risultato a video è lo stesso di prima:



Per osservare il “cumulo” delle trasformazioni possiamo provare a moltiplicare due volte il vecchio gruppo di trasformazione per il nuovo prima di applicarlo:

```
newTransform.setTranslation(translation);

oldTransform.mul(newTransform);
oldTransform.mul(newTransform);

viewPlatformTransformGroup.setTransform(oldTransform);
```

L'effetto risultate è uno spostamento di 2 unità in direzione Z+ seguito da uno spostamento di altre 2 unità nella stessa direzione, per un totale di 4 unità. Il quadrato nero appare di conseguenza più distante:



Questa faccenda della moltiplicazione o, per meglio dire, il cumulo delle trasformazioni, ha una sua

importanza nel momento in cui si desideri applicare una trasformazione che cumuli rotazioni e spostamenti.

Abbiamo scelto di spostare la vista. Ora proviamo a tenerla ferma e a spostare il cubo di due unità in direzione (0, 0, -1), lontano dall'osservatore dello schermo.

Partiamo dall'avvenuto caricamento del cubo.

```
public void create(SimpleUniverse universe) {
      Loader loader = new ObjectFile();
      Scene scene = null;
      try {
            scene = loader.load(getClass().getResource("/cubo.obj"));
      } catch(Exception ex) {
            throw new RuntimeException(ex);
      }
      BranchGroup sceneGroup = scene.getSceneGroup();

      /* qui scriveremo il nuovo codice */

      Background bg = new Background(1, 1, 1);
      BoundingSphere infinity = new BoundingSphere(
            new Point3d(), Double.POSITIVE_INFINITY);
      bg.setApplicationBounds(infinity);
      BranchGroup root = new BranchGroup();
      root.addChild(bg);
      universe.addBranchGraph(root);
}
```

Sappiamo che per spostare qualcosa dobbiamo inserirlo in un gruppo di trasformazione. Dunque creiamo questo gruppo di trasformazione e aggiungiamo ad esso il BranchGroup che contiene il cubo:

```
TransformGroup cubeTransformGroup = new TransformGroup();
cubeTransformGroup.addChild(sceneGroup);
```

Ora creiamo un vettore per lo spostamento di due unità in direzione -Z (ci allontaniamo dall'origine):

```
Vector3d translation = new Vector3d(0, 0, -2);
```

Creiamo un Transform3D:

```
Transform3D cubeTransform = new Transform3D();
```

e applichiamo ad esso i valori dello spostamento:

```
cubeTransform.setTranslation(translation);
```

Assegnamo i nostri dati di trasformazione al gruppo di trasformazione del cubo:

```
cubeTransformGroup.setTransform(cubeTransform);
```

Ora dobbiamo inserire questo nodo TransformGroup nella scena 3D. Per farlo abbiamo bisogno di un BranchGroup. Potremmo usare il BranchGroup che abbiamo creato più avanti per il nodo Background: ricordiamo che la trasformazione applicata dal TransformGroup “inizia” dai figli di quel TransformGroup e non si applica ai nodi “fratelli” (cioè agli altri nodi immediatamente appartenenti al gruppo genitore del TransformGroup). Dopo la linea:

```
root.addChild(bg);
```

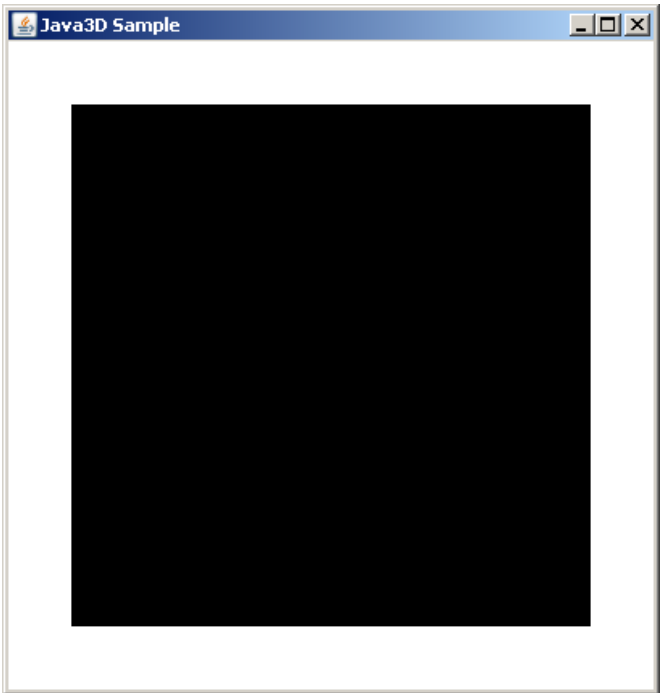
aggiungiamo

```
root.addChild(cubeTransformGroup);
```

Il metodo create(SimpleUniverse) appare quindi:

```
	public void create(SimpleUniverse universe) {
		Loader loader = new ObjectFile();
		Scene scene = null;
		try {
			scene = loader.load(getClass().getResource("/cubo.obj"));
		} catch(Exception ex) {
			throw new RuntimeException(ex);
		}
		BranchGroup sceneGroup = scene.getSceneGroup();

		TransformGroup cubeTransformGroup = new TransformGroup();
		cubeTransformGroup.addChild(sceneGroup);
		Vector3d translation = new Vector3d(0, 0, -2);
		Transform3D cubeTransform = new Transform3D();
		cubeTransform.setTranslation(translation);
		cubeTransformGroup.setTransform(cubeTransform);

		Background bg = new Background(1, 1, 1);
		BoundingSphere infinity = new BoundingSphere(
			new Point3d(), Double.POSITIVE_INFINITY);
		bg.setApplicationBounds(infinity);
		BranchGroup root = new BranchGroup();
		root.addChild(bg);

		root.addChild(cubeTransformGroup);

		universe.addBranchGraph(root);
	}
```

Il risultato visivo è sempre lo stesso.



Giramenti.

Facciamo girare il nostro cubo su sé stesso. Per questo genere di animazioni Java3D mette a disposizione la classe Behavior. Gli oggetti di tipo Behavior sono dei nodi. Come tali vengono inseriti nell'albero di proiezione. La particolarità dei Behavior è che essi possiedono un metodo, processStimulus, che Java3D invoca quando si verifichi una certa condizione. La condizione che deve verificarsi è stabilita dal Behavior stesso. Come Background, anche Behavior ha una regione di attivazione. Quando la telecamera si trova in quella regione, il Behavior è attivo e il motore di proiezione esegue il suo metodo processStimulus. Quando la telecamera è fuori dalla regione di attivazione, il Behavior non è attivo. Behavior è una classe astratta. Iniziamo col definirne una versione concreta.

```
package it.quickjava3d;

import javax.vecmath.*;
import javax.media.j3d.*;

public class CubeRotator extends Behavior {

}
```

I metodi da definire sono due. Il metodo initialize(), come suggerisce il nome, si occupa dell'inizializzazione del nodo. Una cosa che occorre fare nel metodo initialize è predisporre la condizione di prima attivazione del Behavior. Nel caso di animazioni continue, come quella che vogliamo realizzare, la condizione di attivazione è “ad ogni passaggio del ciclo di rendering”. Questa condizione è espressa da uno dei molti oggetti WakeupCriterion disponibili per i Behavior e, precisamente, da un WakeupOnElapsedFrames creato con un valore zero come parametro. Creiamo quindi un campo in CubeRotator, di tipo WakeupOnElapsedFrames:

```
private final WakeupOnElapsedFrames WAKEUP = new WakeupOnElapsedFrames(0);
```

Lo creo come campo perchè mi servirà sia nel metodo initialize che nel metodo processStimulus e la condizione di attivazione del Behavior sarà sempre la stessa.

Sottolineo ancora cosa sia questo WAKEUP. I Behavior sono comportamenti, computazioni, calcoli, pensatela come più vi piace, attivati cioè eseguiti da Java3D quando si verifichi una certa condizione. La condizione di attivazione per un'animazione continua è “ogni volta che devi disegnare”, rivolto al motore di rendering. Questa condizione si esprime con un'istanza di WakeupOnElapsedFrames. Vediamo come questa condizione è usata dal Behavior in questa seconda implementazione parziale di CubeRotator:

```
package it.quickjava3d;

import javax.vecmath.*;
import javax.media.j3d.*;

public class CubeRotator extends Behavior {
	private final WakeupOnElapsedFrames WAKEUP = new WakeupOnElapsedFrames(0);

	public void initialize() {
		wakeupOn(WAKEUP);
	}

	public void processStimulus(java.util.Enumeration criteria) {
		/* codice che specifica cosa farà questo behavior e infine, di
		nuovo... */
		wakeupOn(WAKEUP);
	}
}
```

Java3D eseguirà una prima volta il metodo initialize. Qui CubeRotator dirà: voglio essere eseguito quando terminerà il frame corrente. Questo causa l'esecuzione del metodo processStimulus al termine del frame corrente. L'ultima istruzione del metodo processStimulus dice ancora: voglio essere eseguito quanto terminerà il frame corrente. Questa condizione sarà esaminata dal motore di Java3D quando sarà il momento di eseguire la proiezione del frame successivo. Java3D rileverà che CubeRotator ha richiesto di essere eseguito al termine di quel frame, sarà invocato il suo metodo processStimulus, che chiederà ancora l'esecuzione eccetera eccetera fino alla fine dei tempi.

Veniamo alla nostra rotazione. Si tratta di un moto angolare uniforme vale a dire che facciamo girare il nostro cubo di un tot per unità di tempo. Prendiamo come unità di tempo il millisecondo e come “tot” un seimillesimo di radiante. Esprimiamo questo valore con una costante, anch'essa campo terminale della classe CubeRotator.

```
package it.quickjava3d;

import javax.vecmath.*;
import javax.media.j3d.*;

public class CubeRotator extends Behavior {
	private final WakeupOnElapsedFrames WAKEUP = new WakeupOnElapsedFrames(0);
	private final double ANGULAR_VELOCITY = 2 * Math.PI / 6000;

	public void initialize() {
		wakeupOn(WAKEUP);
	}

	public void processStimulus(java.util.Enumeration criteria) {
		/* codice che specifica cosa farà questo behavior e infine, di
		nuovo... */
		wakeupOn(WAKEUP);
	}
}
```

Usiamo i radianti perchè le rotazioni in un Transform3D devono essere espresse in radianti. Possiamo anche usare i gradienti ma poi dovrebbe trasformarli in radianti prima di applicarli. Tanto vale fare confusione una volta sola.

Noi vogliamo far ruotare un cubo ma sappiamo che una rotazione in Java3D è una trasformazione applicata da un TransformGroup. Il nostro CubeRotator il cubo non lo vede neppure: vedrà invece un TransformGroup. Questo TransformGroup arriverà dall'esterno. Usiamo a questo scopo un terzo campo che assumerà un valore in costruzione:

```
package it.quickjava3d;

import javax.vecmath.*;
import javax.media.j3d.*;

public class CubeRotator extends Behavior {
	private final WakeupOnElapsedFrames WAKEUP = new WakeupOnElapsedFrames(0);
	private final double ANGULAR_VELOCITY = 2 * Math.PI / 6000;
	private final TransformGroup TARGET;

	public CubeRotator(TransformGroup target) {
		TARGET = target;
	}

	public void initialize() {
		wakeupOn(WAKEUP);
	}

	public void processStimulus(java.util.Enumeration criteria) {
		/* codice che specifica cosa farà questo behavior e infine, di
		nuovo... */
		wakeupOn(WAKEUP);
```

```
	}
}
```

Per applicare una trasformazione al gruppo TARGET dovremo creare un Transform3D a cui assegnare i dati di trasformazione attuali, creare un secondo Transform3D a cui applicare il “tot” di rotazione, moltiplicare il primo Transform3D per il secondo e poi riapplicarlo a TARGET. Onde evitare di usare l'operatore new in un metodo che sarà invocato ...mila volte al secondo, creiamo altri due campi, stavolta di tipo Transform3D. Uno ci servirà per trasferire i dati di trasformazione correnti, l'altro per immagazzinare la variazione di rotazione da applicare.

```
package it.quickjava3d;

import javax.vecmath.*;
import javax.media.j3d.*;

public class CubeRotator extends Behavior {
	private final WakeupOnElapsedFrames WAKEUP = new WakeupOnElapsedFrames(0);
	private final double ANGULAR_VELOCITY = 2 * Math.PI / 6000;
	private final TransformGroup TARGET;
	private final Transform3D TARGET_TRANSFORM = new Transform3D();
	private final Transform3D MUL_TRANSFORM = new Transform3D();

	public CubeRotator(TransformGroup target) {
		TARGET = target;
	}

	public void initialize() {
		wakeupOn(WAKEUP);
	}

	public void processStimulus(java.util.Enumeration criteria) {
		/* codice che specifica cosa farà questo behavior e infine, di
		nuovo... */
		wakeupOn(WAKEUP);
	}
}
```

Prima di passare al codice contenuto in processStimulus, creiamo una funzione di temporizzazione. Questa funzione ci serve per determinare quanti millisecondi sono trascorsi dall'ultimo spostamento. Il tempo trascorso, moltiplicato per la velocità angolare, ci dirà di quanti radianti dovremo far girare il gruppo di trasformazione in questo frame. Per la funzione di temporizzazione abbiamo bisogno di un valore tampone che conservi il tempo del passaggio precedente nel ciclo di rendering. Altro campo, stavolta non terminale:

```
package it.quickjava3d;

import javax.vecmath.*;
import javax.media.j3d.*;

public class CubeRotator extends Behavior {
	private final WakeupOnElapsedFrames WAKEUP = new WakeupOnElapsedFrames(0);
	private final double ANGULAR_VELOCITY = 2 * Math.PI / 6000;
	private final TransformGroup TARGET;
	private final Transform3D TARGET_TRANSFORM = new Transform3D();
	private final Transform3D MUL_TRANSFORM = new Transform3D();
	private long then;

	public CubeRotator(TransformGroup target) {
		TARGET = target;
	}

	public void initialize() {
		wakeupOn(WAKEUP);
	}

	public void processStimulus(java.util.Enumeration criteria) {
		/* codice che specifica cosa farà questo behavior e infine, di
```

```
            nuovo... */
            wakeupOn(WAKEUP);
      }
}
```

La funzione di temporizzazione calcola semplicemente quanto tempo è trascorso tra due invocazioni della stessa. La inseriamo come metodo privato e la invocheremo da processStimulus.

```
      private long computeElapsedTime() {
            long now = System.nanoTime();
            if(then == 0) {
                  then = now;
            }
            long dTime = now - then;
            then = now;
            return dTime / 1000000; //nano to millis
      }
```

A questo punto è fatta. Nel metodo processStimulus invochiamo computeElapsedTime e otteniamo i millisecondi trascorsi dall'ultima invocazione:

```
            long elapsedTime = computeElapsedTime();
```

Poi moltiplichiamo questo tempo per la velocità angolare ottenendo lo spostamento angolare:

```
            double displacement = ANGULAR_VELOCITY * elapsedTime;
```

Ricaviamo dal TransformGroup TARGET i dati di trasformazione attuali:

```
            TARGET.getTransform(TARGET_TRANSFORM);
```

Prendiamo il Transform3D MUL_TRANSFORM e lo impostiamo ai valori di una rotazione angolare di displacement radianti:

```
            MUL_TRANSFORM.rotY(displacement);
```

Da notare che il metodo rotY (come rotX e rotZ) azzera i valori di trasformazione precedentemente contenuti nel Transform3D (quindi non può essere applicato a TARGET_TRANSFORM altrimenti perderemmo la posizione e la rotazione precedente).

Ora accumuliamo lo spostamento corrispondente a questo frame con le trasformazioni precedentemente applicate al gruppo TARGET, moltiplicando i “vecchi” dati con la rotazione corrente:

```
            TARGET_TRANSFORM.mul(MUL_TRANSFORM);
```

passiamo questi dati a TARGET:

```
            TARGET.setTransform(TARGET_TRANSFORM);
```

e chiudiamo il metodo processStimulus con la richiesta di attivazione già esaminata:

```
            wakeupOn(WAKEUP);
```

La classe CubeRotator interamente costruita è come segue:

```
package it.quickjava3d;

import javax.vecmath.*;
```

```
import javax.media.j3d.*;

public class CubeRotator extends Behavior {
	private final WakeupOnElapsedFrames WAKEUP = new WakeupOnElapsedFrames(0);
	private final double ANGULAR_VELOCITY = 2 * Math.PI / 6000;
	private final TransformGroup TARGET;
	private final Transform3D TARGET_TRANSFORM = new Transform3D();
	private final Transform3D MUL_TRANSFORM = new Transform3D();
	private long then;

	public CubeRotator(TransformGroup target) {
		TARGET = target;
	}

	public void initialize() {
		wakeupOn(WAKEUP);
	}

	public void processStimulus(java.util.Enumeration criteria) {
		long elapsedTime = computeElapsedTime();
		double displacement = ANGULAR_VELOCITY * elapsedTime;

		TARGET.getTransform(TARGET_TRANSFORM);
		MUL_TRANSFORM.rotY(displacement);
		TARGET_TRANSFORM.mul(MUL_TRANSFORM);
		TARGET.setTransform(TARGET_TRANSFORM);

		wakeupOn(WAKEUP);
	}

	private long computeElapsedTime() {
		long now = System.nanoTime();
		if(then == 0) {
			then = now;
		}
		long dTime = now - then;
		then = now;
		return dTime / 1000000; //nano to millis
	}
}
```

Per vedere CubeRotator all'opera torniamo alla classe MyWorld. Ci sono quattro cose da fare nel metodo create(SimpleUniverse). La prima è creare un nodo CubeRotator passandogli come argomento in costruzione il TrasformGroup a cui è connesso il cubo.

```
CubeRotator rotator = new CubeRotator(cubeTransformGroup);
```

La seconda è assegnare a CubeRotator un volume di attivazione. Sfruttando il volume sferico usato per il nodo Background diremo:

```
rotator.setSchedulingBounds(infinity);
```

La terza è aggiungere il nodo CubeRotator all'albero di proiezione. Ancora, usiamo il BranchGroup “root”, creato per il nodo Background:

```
root.addChild(rotator);
```

L'ultima è impostare due capacità per il gruppo di trasformazione del cubo che sarà manipolato dal Behavior. Ricordate quella faccenda della distinzione tra comportamento speciale “da morto” e “da vivo”? La specialità di TransformGroup, cioè la trasformazione, è soggetta alla stessa regola della connessione dinamica di BranchGroup: se si vuole usare quando il nodo è “vivo” occorre dichiararlo con il metodo setCapability. Per un gruppo di trasformazione, la segnalazione d'uso in vita delle capacità di trasformazione è indicata con ALLOW_TRANSFORM_READ e ALLOW_TRANSFORM_WRITE. La prima consente l'estrazione dei dati di trasformazione. La

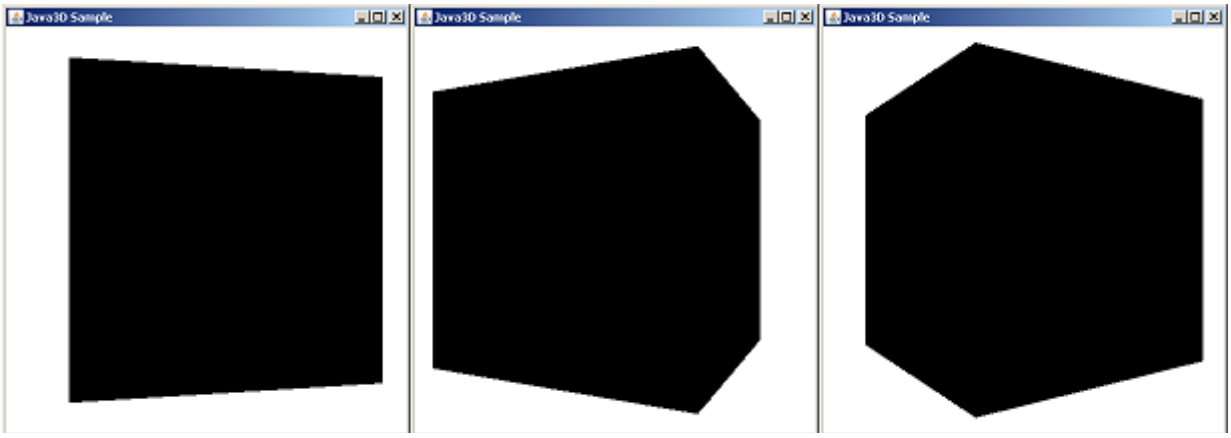
seconda consente l'immissione di nuovi dati di trasformazione. In create(SimpleUniverse) inseriremo le linee:

```
cubeTransformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
cubeTransformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
```

Ricapitolando, il metodo create(SimpleUniverse) in MyWorld diventa:

```
	public void create(SimpleUniverse universe) {
		Loader loader = new ObjectFile();
		Scene scene = null;
		try {
			scene = loader.load(getClass().getResource("/cubo.obj"));
		} catch(Exception ex) {
			throw new RuntimeException(ex);
		}
		BranchGroup sceneGroup = scene.getSceneGroup();

		TransformGroup cubeTransformGroup = new TransformGroup();

		cubeTransformGroup.setCapability(
			TransformGroup.ALLOW_TRANSFORM_READ);
		cubeTransformGroup.setCapability(
			TransformGroup.ALLOW_TRANSFORM_WRITE);
		CubeRotator rotator = new CubeRotator(cubeTransformGroup);

		cubeTransformGroup.addChild(sceneGroup);
		Vector3d translation = new Vector3d(0, 0, -2);
		Transform3D cubeTransform = new Transform3D();
		cubeTransform.setTranslation(translation);
		cubeTransformGroup.setTransform(cubeTransform);

		Background bg = new Background(1, 1, 1);
		BoundingSphere infinity = new BoundingSphere(
			new Point3d(), Double.POSITIVE_INFINITY);
		bg.setApplicationBounds(infinity);
		BranchGroup root = new BranchGroup();
		root.addChild(bg);

		root.addChild(cubeTransformGroup);

		rotator.setSchedulingBounds(infinity);
		root.addChild(rotator);

		universe.addBranchGraph(root);
	}
```

Abbiamo così applicato la nostra rotazione. Se si potesse rendere l'idea di una rotazione 3D animata con un'immagine statica non staremmo qui a parlare di Java3D. Comunque, qui sotto c'è un riassunto:



Ebbene, abbiamo ruotato il cubo. In verità non è stata una rotazione ma una rototraslazione, rispetto all'origine, o una rotazione del cubo intorno all'asse z = -2. Infatti, se osservate il codice di create(SimpleUniverse) o ricordate i passaggi precedenti alla rotazione, noterete come il cubo subisca uno spostamento dall'origine al punto (0, 0, -2) proprio nel metodo create(SimpleUniverse):

```
CubeRotator rotator = new CubeRotator(cubeTransformGroup);

cubeTransformGroup.addChild(sceneGroup);
Vector3d translation = new Vector3d(0, 0, -2);
Transform3D cubeTransform = new Transform3D();
cubeTransform.setTranslation(translation);
cubeTransformGroup.setTransform(cubeTransform);

Background bg = new Background(1, 1, 1);
```

Questo spostamento è preservato dalla concatenazione di trasformazioni realizzata attraverso la moltiplicazione di un Transform3D con un altro.