# *JAVA 3D – TRASFORMAZIONI*

Corso di Immagini e Multimedialità
Lezione 3.2: JAVA3D – Trasformazioni

Cristian.virgili@uniud.it

# *Come si devono fare gli esercizi*

Ogni esercizio contenuto nella scheda deve avere la seguente documentazione:

Traccia dell'esercizio;
**Scenegraph** della scena, rappresentato usando la notazione adottata durante le lezioni e le esercitazioni di laboratorio;
**Codice commentato**. Non è necessario tutto il codice, ma le parti più significative in relazione a quanto richiesto nell'esercizio.
Motivazioni sul modo in cui è stato svolto l'esercizio. Ad esempio, giustificare perché è stata scelta una determinata struttura per lo scenegraph...
Schermate con i risultati ottenuti;
Conclusioni ed eventuali osservazioni.

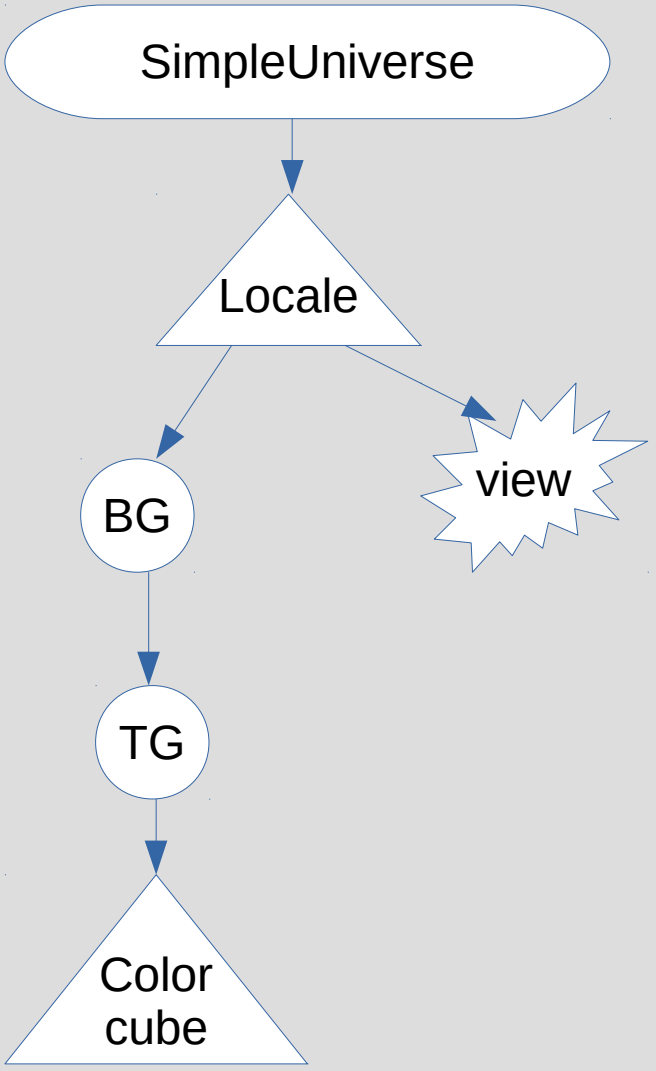
Inviare anche il codice **FUNZIONANTE**

# Esercizio 1

Costruire una scena composta da un singolo cubo
a cui applicare separatamente:

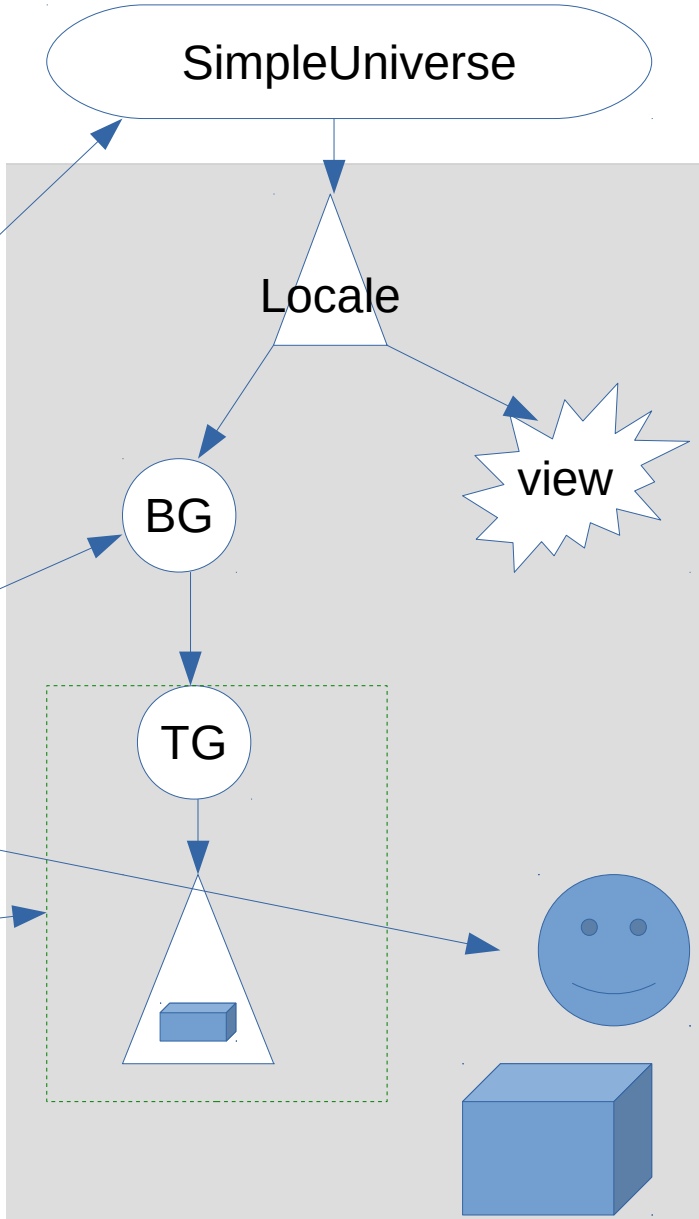
- rotazione
- scalatura
- traslazione

# Esercizio 1
# Disegniamo lo scene graph



# STRUTTURA GENERALE PER GLI ESEMPI

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.GraphicsConfiguration;
import javax.media.j3d.BranchGroup;
import javax.media.j3d.Canvas3D;
import javax.media.j3d.Transform3D;
import javax.media.j3d.TransformGroup;
import javax.vecmath.Vector3d;

import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.ColorCube;
import com.sun.j3d.utils.universe.SimpleUniverse;


public class HelloJava3DRotazione extends Applet {


    public HelloJava3DRotazione() {
        setLayout(new BorderLayout());
        Transform3D t = new Transform3D();
        GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();
        Canvas3D canvas3D = new Canvas3D(config);
        add("Center", canvas3D);
        BranchGroup scene = createSceneGraph();
        scene.compile();

        SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
        simpleU.getViewingPlatform().setNominalViewingTransform();
        simpleU.addBranchGraph(scene);
    }

     // Funzione che crea il sottografo
     public BranchGroup createSceneGraph() {
       BranchGroup node = new BranchGroup();
       TransformGroup TG = createSubGraph(); //funzione implementata successivamente
       node.addChild(TG); //aggiunge l'oggetto TG come figlio del BrachGruop
      return node;
     }



    public static void main(String[] args) {
        new MainFrame(new HelloJava3DRotazione(), 1024, 768);
    }
```



# *Variazione della dimensione del ColorCube*

- Esercizio: modificare la dimensione del cubo
- Per variare le dimensioni del cubo è sufficiente modificare il parametro dimCube passato al costruttore del ColorCube: TG.addChild(new ColorCube(dimCube))

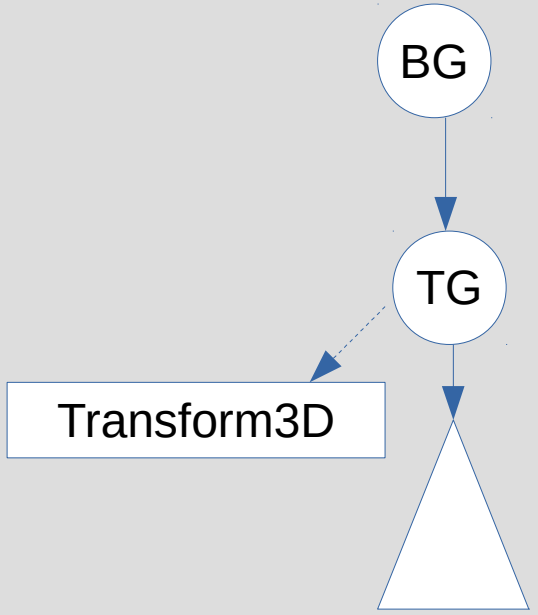
# *APPLICAZIONE DELLE TRASFORMAZIONI TRASLAZIONE*

L'obiettivo ora è quello di spostare il cubo in una posizione che sia differente da quella dell'origine.

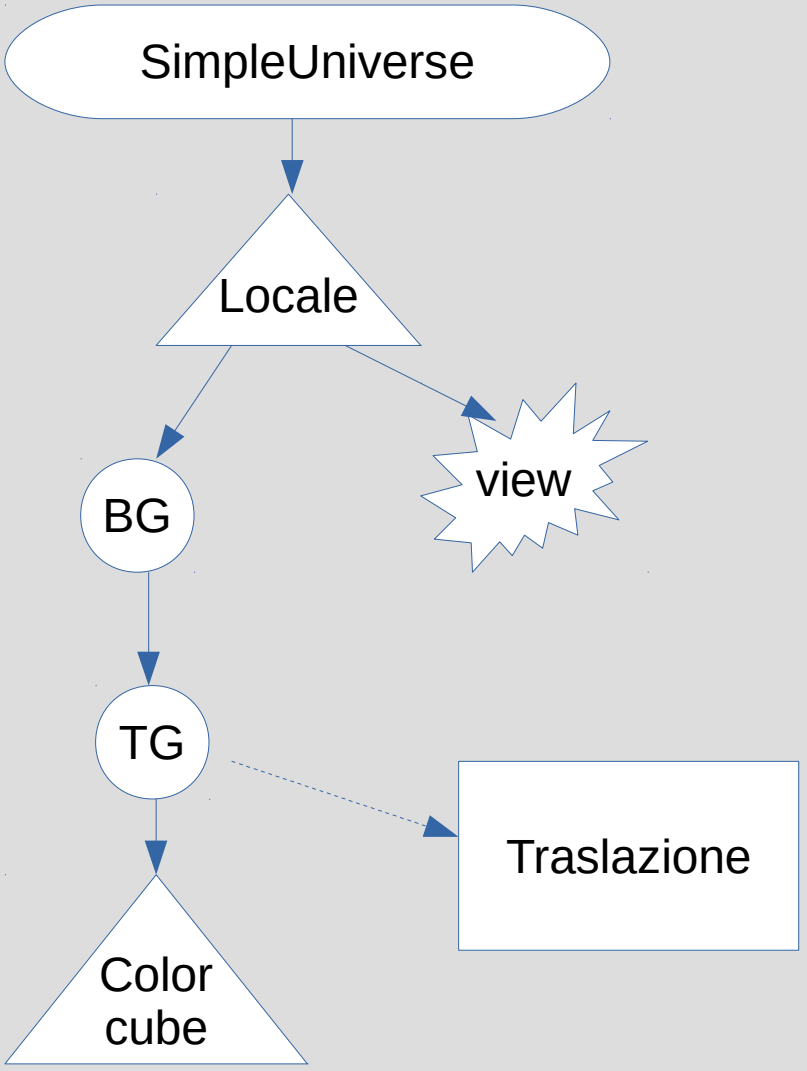
- TransfomGroup
- Transform3D
- La traslazione è specificata usando il metodo setTranslation(Vector3d vector) il valore passato come parametro è un Vector3d che definisce con precisione double i valori della traslazione per ognuna delle tre coordinate x,y,z (la d posta dopo i valori rappresenta il fatto che stiamo lavorando con double).
- Come ultimo passo, la trasformazione contenuta nell'oggetto Transfomr3D deve essere assegnata all'oggetto TransfomGroup utilizzando il metodo setTransform(Transform3D td).

# APPLICAZIONE DELLE TRASFORMAZIONI
# TRASLAZIONE

```
public TransformGroup createSubGraph(){
  TransformGroup transform = new TransformGroup(); // crea oggetto TG
  Transform3D td3 = new Transform3D(); // creo oggetto per la trasformazione
  td3.setTranslation(new Vector3d(-0.3d, 0.4d, 0.2d)); // def. traslazione
  transform.setTransform(td3); // assegno a transform la trasformazione
  transform.addChild(new ColorCube(0.3));  //aggiungo al TG come figlio il cubo
 return transform;
 }
```




# Esercizio 1 - Traslazione
# Disegniamo lo scene graph



# APPLICAZIONE DELLE TRASFORMAZIONI
# TRASLAZIONE - Esercizio 3.1 a

Esercizi:

- Fare le traslazioni sui singoli assi (x,y,z)

- Cosa succede quando mantenendo fisso il punto di visualizzazione e traslo:

  - Per valori positivi della x
  - Per valori negativi della x

  - Per valori positivi della y
  - Per valori negativi della y

  - Per valori positivi della z
  - Per valori negativi della z

  - Fare una traslazione in modo che il cubo sia visto in basso a dx e risulti più piccolo.

# APPLICAZIONE DELLE TRASFORMAZIONI
## Rotazione

L'obiettivo è quello di ruotare il cubo intorno all'asse x, all'asse y o all'asse z e di vedere il comportamento di quest'ultimo a seconda dell'angolo di rotazione che si utilizza.

```
public TransformGroup createSubGraph(){

      TransformGroup transform = new TransformGroup(); // crea oggetto TG
      Transform3D td3 = new Transform3D(); // creo oggetto per la trasformazione
      td3.rotX(Math.PI*0.4d); // def. rotazione su x
      transform.setTransform(td3); // assegno a transform la trasformazione
      transform.addChild(new ColorCube(0.3)); //aggiungo al TG come figlio il cubo
      return transform;
  }



      td3.rotY(Math.PI*0.4d);// def. rotazione su y
      td3.rotZ(Math.PI*0.4d);// def. rotazione su z
```

# APPLICAZIONE DELLE TRASFORMAZIONI
## Rotazione- - Esercizio 3.1 b

Esercizi da fare per **RELAZIONE FINALE**:

- Fare le rotazioni sui singoli assi (x,y,z)

- Cosa succede quando mantenendo fisso il punto di visualizzazione e ruoto:

  - Per valori positivi della x (provare con 90°,180°,...)
  - Per valori negativi della x

  - Per valori positivi della y
  - Per valori negativi della y

  - Per valori positivi della z
  - Per valori negativi della z

# APPLICAZIONE DELLE TRASFORMAZIONI
## Scaling

L'obiettivo ora è quello di scalare le dimensioni del cubo

```
public TransformGroup createSubGraphScale(){
      TransformGroup transform = new TransformGroup(); // crea oggetto TG
      Transform3D td3 = new Transform3D(); // creo oggetto per la trasformazione
      td3.setScale(new Vector3d(3.0d,1.5d, 1.5d)); //def. scaling
      transform.setTransform(td3); // assegno a transform la trasformazione
      transform.addChild(new ColorCube(0.3));  //aggiungo al TG come figlio il cubo
      return transform;
}
```

# APPLICAZIONE DELLE TRASFORMAZIONI
## Scaling- - Esercizio 3.1 c

Esercizi:

- Fare le scalature sui singoli assi (x,y,z)

- Cosa succede quando mantenendo fisso il punto di visualizzazione e scalo:

  - Per valori positivi della x
  - Per valori negativi della x

  - Per valori positivi della y
  - Per valori negativi della y

  - Per valori positivi della z
  - Per valori negativi della z

# *FATE BENE GLI ESERCIZI*

- Valuteremo:
  - Lo SceneGraph (completi o indicate le parti mancanti)
  - Il codice sorgente: struttura, riusabilità, commenti
  - Il risultato finale: “schermate”
  - Aggiunte: tutto quello che aggiungete sarà tenuto in considerazione

# *APPLICAZIONE DELLE TRASFORMAZIONI COMBINAZIONE*

E' possibile combinare tra di loro diverse trasformazioni; questo può essere fatto semplicemente utilizzando il metodo

mul(Transform3D 3d).

Il vantaggio nell'utilizzare la combinazione di trasformazioni rispetto all'applicare separatamente più trasformazioni singole è legato ad un problema di efficienza.

# *APPLICAZIONE DELLE TRASFORMAZIONI COMBINAZIONE*

**Traslazione e Rotazione:**

per realizzare bisogna creare due oggetti Transform3D necessari per definire le trasformazioni.

Esercizio: combiniamo due trasformazioni prima un  rotazione di 45° su **x** e poi una traslazione sempre su **x** di -0.4.

# APPLICAZIONE DELLE TRASFORMAZIONI COMBINAZIONE

```
public TransformGroup createSubGraph (){
        TransformGroup transform = new TransformGroup(); // crea oggetto TG
        Transform3D td3 = new Transform3D(); // creo oggetto per la trasf.
        Transform3D temp1 = new Transform3D();// creo oggetto per la trasf.
        td3.rotX(Math.PI/4); //def. rotazione su x
        temp1.setTranslation(new Vector3d(-0.4d, 0.0d, 0.0d)); //def. traslazione
        td3.mul(temp1); // moltiplico i due operatori prima temp1 poi td3
        transform.setTransform(td3); // assegno a transform la trasformazione
        //aggiungo al TG come figlio il cubo
        transform.addChild(new ColorCube(0.3));
       return transform;
  }
```

# APPLICAZIONE DELLE TRASFORMAZIONI COMBINAZIONE

Combinare una Scalatura ed una Traslazione.

```
public TransformGroup createSubGraphScaleTranslation (){
      TransformGroup transform = new TransformGroup(); // crea oggetto TG
      Transform3D td3 = new Transform3D(); // creo oggetto per la trasformazione
      Transform3D temp1 = new Transform3D();// creo oggetto per la trasformazione
      td3.setScale(new Vector3d(1d, 1d, 2d)); //def. scaling su x
      temp1.setTranslation(new Vector3d(-0.3d, 0.0d, 0.0d)); // def. traslazione
      td3.mul(temp1); // moltiplico i due operatori prima temp1 poi td3
      transform.setTransform(td3); // assegno a transform la trasformazione
      transform.addChild(new ColorCube(0.3));  //aggiungo al TG come figlio il cubo
     return transform;
}
```

# APPLICAZIONE DELLE TRASFORMAZIONI COMBINAZIONE

Le operazioni di trasformazioni combinate sono il risultato di una moltiplicazione di matrici

Verificare: il prodotto tra matrici è commutativa?

TA = T1 * T2 , TB = T2 * T1, TA = TB(?)

Provare ad esempio:
- Rotazione y (180°) e poi Traslazione x (-0.5)
- Traslazione x(-0.5) e poi Rotazione y (180°)

Cosa succede se invece faccio lo stesso tipo di trasformazione?

# *APPLICAZIONE DELLE TRASFORMAZIONI COMBINAZIONE*

**CONCLUSIONI**

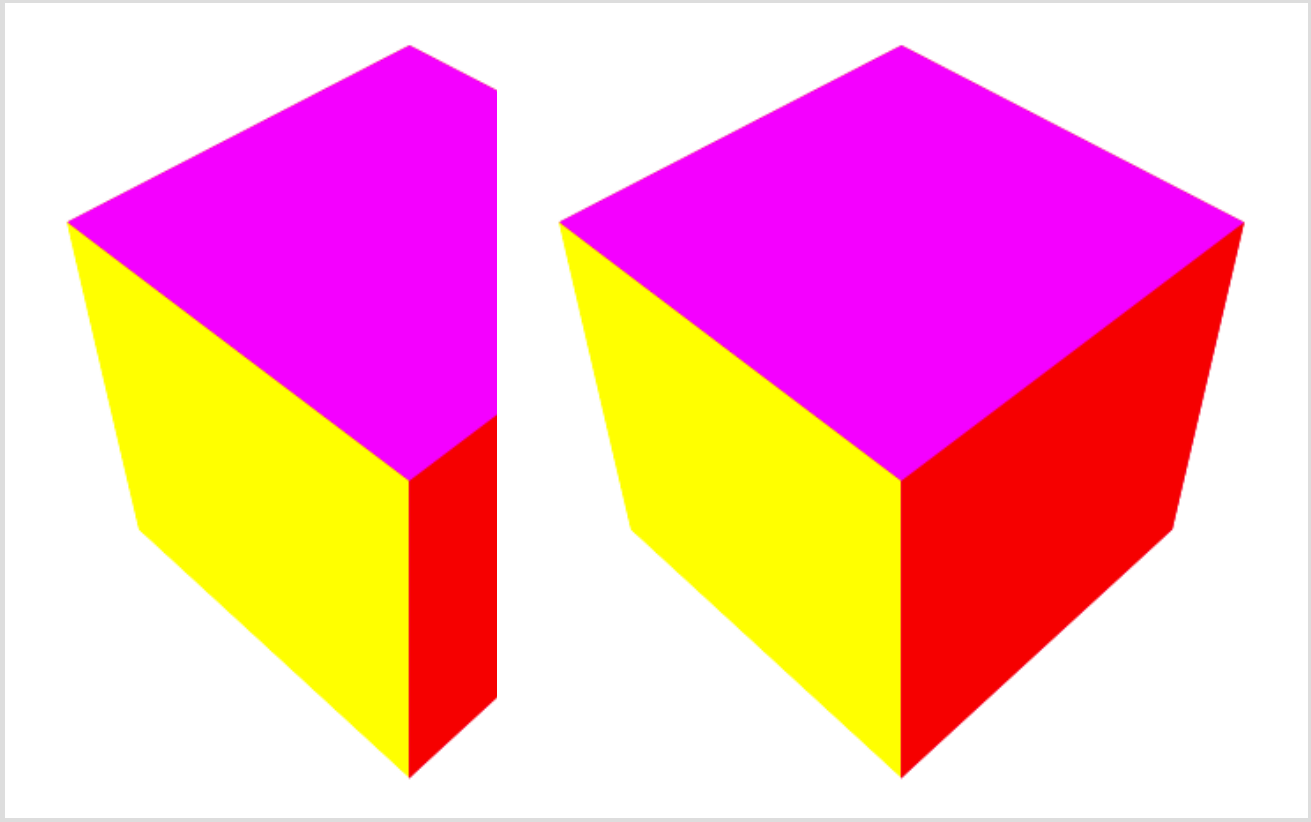
La combinazione di trasformazioni è più efficiente che applicare le trasformazioni singolarmente;

La combinazione di trasformazioni non gode della proprietà commutativa se le trasformazioni sono diverse;

La combinazione di trasformazioni gode della proprietà commutativa se le trasformazioni sono dello stesso tipo;

E’ possibile combinare un numero arbitrario *n* di trasformazioni.

# AGGIUNTA DI UN CUBO ALLA SCENA

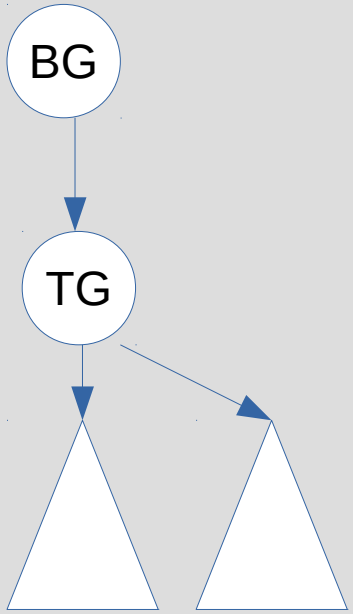


Cristian Virgili e Vito Roberto

# AGGIUNTA DI UN CUBO ALLA SCENA

## Soluzione 1

```
// Funzione che crea il sottografo
public BranchGroup createSceneGraph() {
  BranchGroup BG = new BranchGroup();
  TransformGroup TG = new TransformGroup();
  TG.addChild(new ColorCube(0.4));
  TG.addChild(new ColorCube(0.4));
  BG.addChild(TG);
 return BG;
}
```

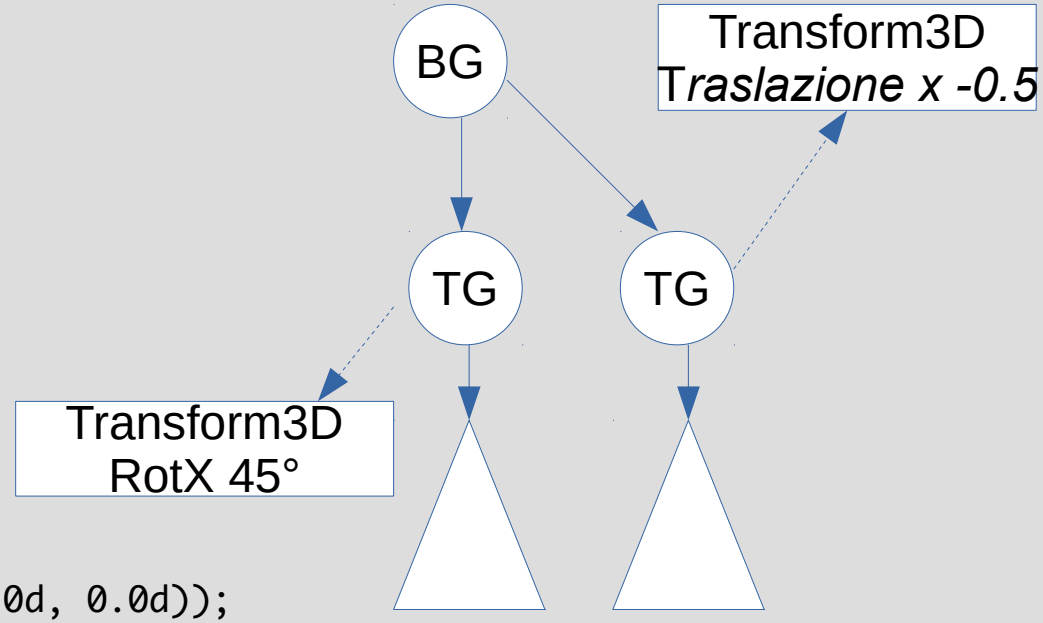



Risultato?
Se voglio averli diversi?

# AGGIUNTA DI UN CUBO ALLA SCENA

## Soluzione 2

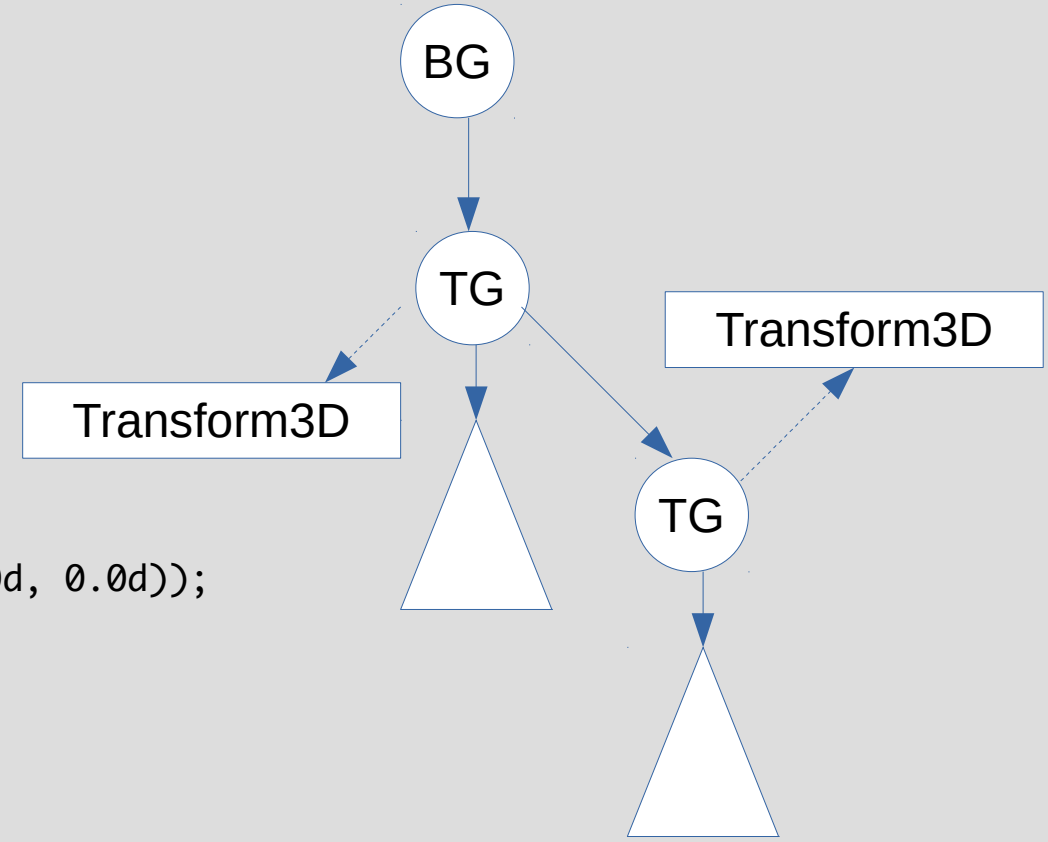
```
// Funzione che crea il sottografo
public BranchGroup createSceneGraph() {
  BranchGroup BG = new BranchGroup();
  //Primo nodo
  TransformGroup TG = new TransformGroup();
  Transform3D t3d = new Transform3D();
  t3d.rotX(Math.PI/4);
  TG.setTransform(t3d);
  TG.addChild(new ColorCube(0.4));

  //Secondo nodo
  TransformGroup TG2 = new TransformGroup();
  Transform3D t3d2 = new Transform3D();
  t3d2.setTranslation(new Vector3d(-0.5d, 0.0d, 0.0d));
  TG2.setTransform(t3d2);
  TG2.addChild(new ColorCube(0.3));

  BG.addChild(TG);
  BG.addChild(TG2);
 return BG;
}
```



# AGGIUNTA DI UN CUBO ALLA SCENA

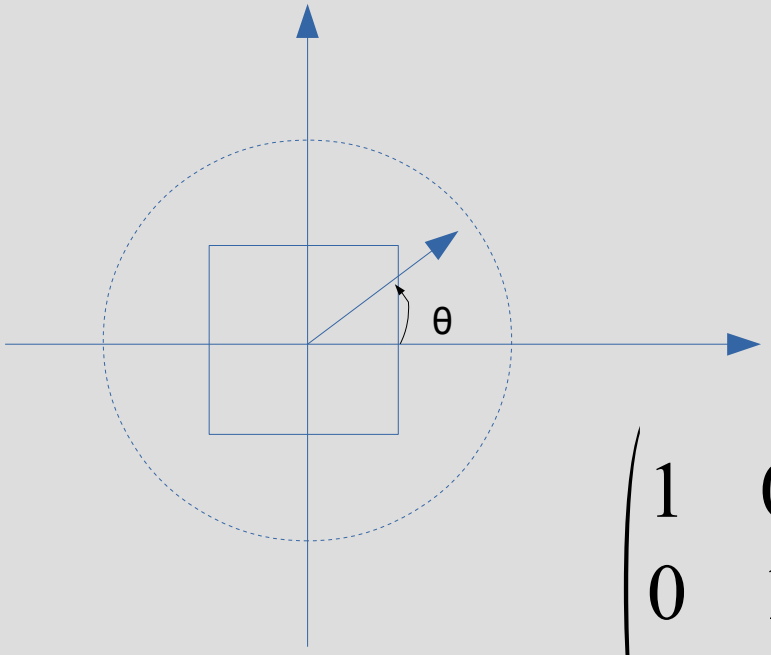
## Soluzione 3

```
// Funzione che crea il sottografo
public BranchGroup createSceneGraph() {
  BranchGroup BG = new BranchGroup();
  //Primo nodo
  TransformGroup TG = new TransformGroup();
  Transform3D t3d = new Transform3D();
  t3d.rotX(Math.PI/4);
  TG.setTransform(t3d);
  TG.addChild(new ColorCube(0.4));

  //Secondo nodo
  TransformGroup TG2 = new TransformGroup();
  Transform3D t3d2 = new Transform3D();
  t3d2.setTranslation(new Vector3d(-0.5d, 0.0d, 0.0d));
  TG2.setTransform(t3d2);
  TG2.addChild(new ColorCube(0.3));
  TG.addChild(TG2);
  BG.addChild(TG);

 return BG;
}
```



# Esercizio - Relazione finale
# Esercizio 3.2

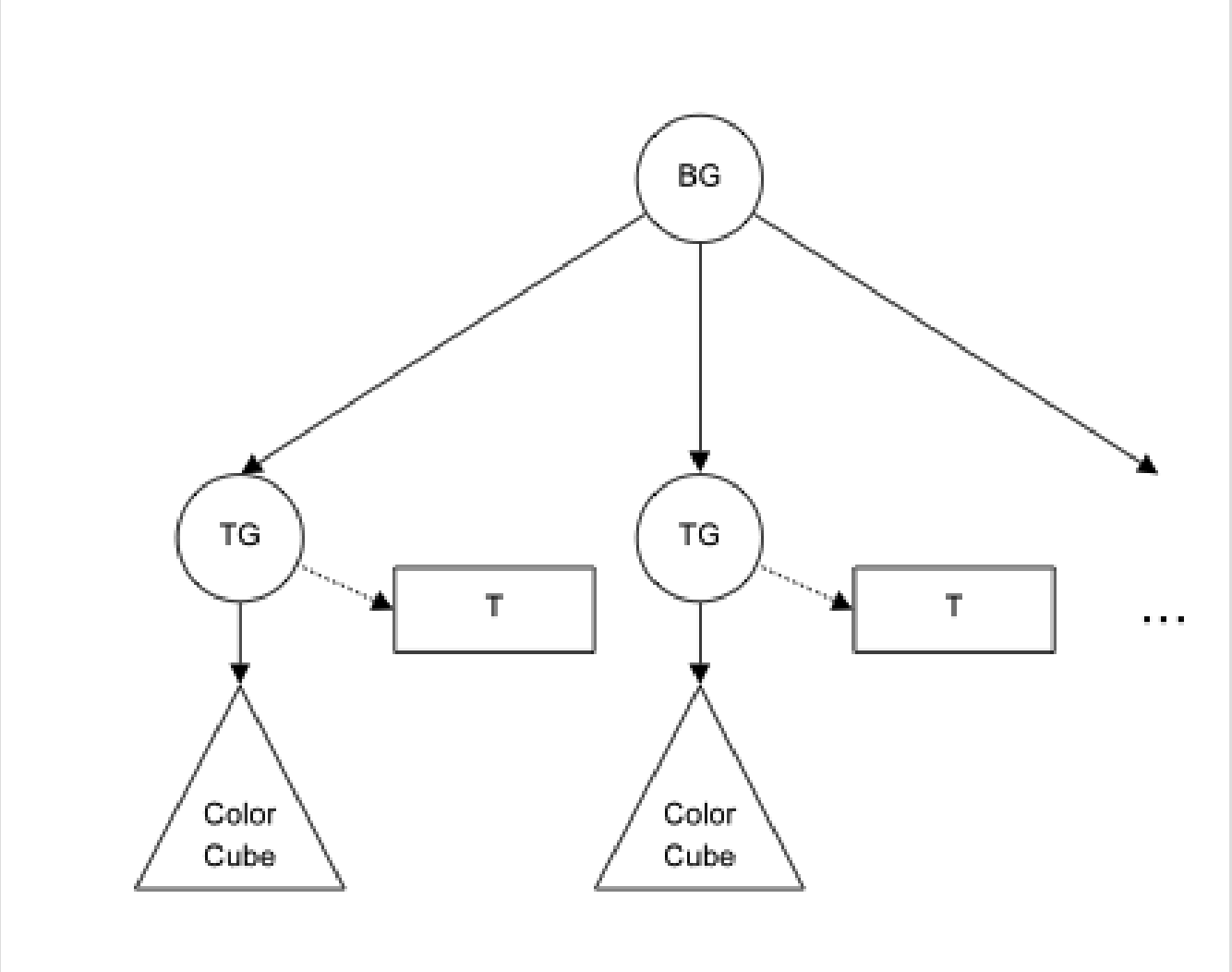
Creare una scena con un numero arbitrario di cubi (diversi) disposti a cerchio



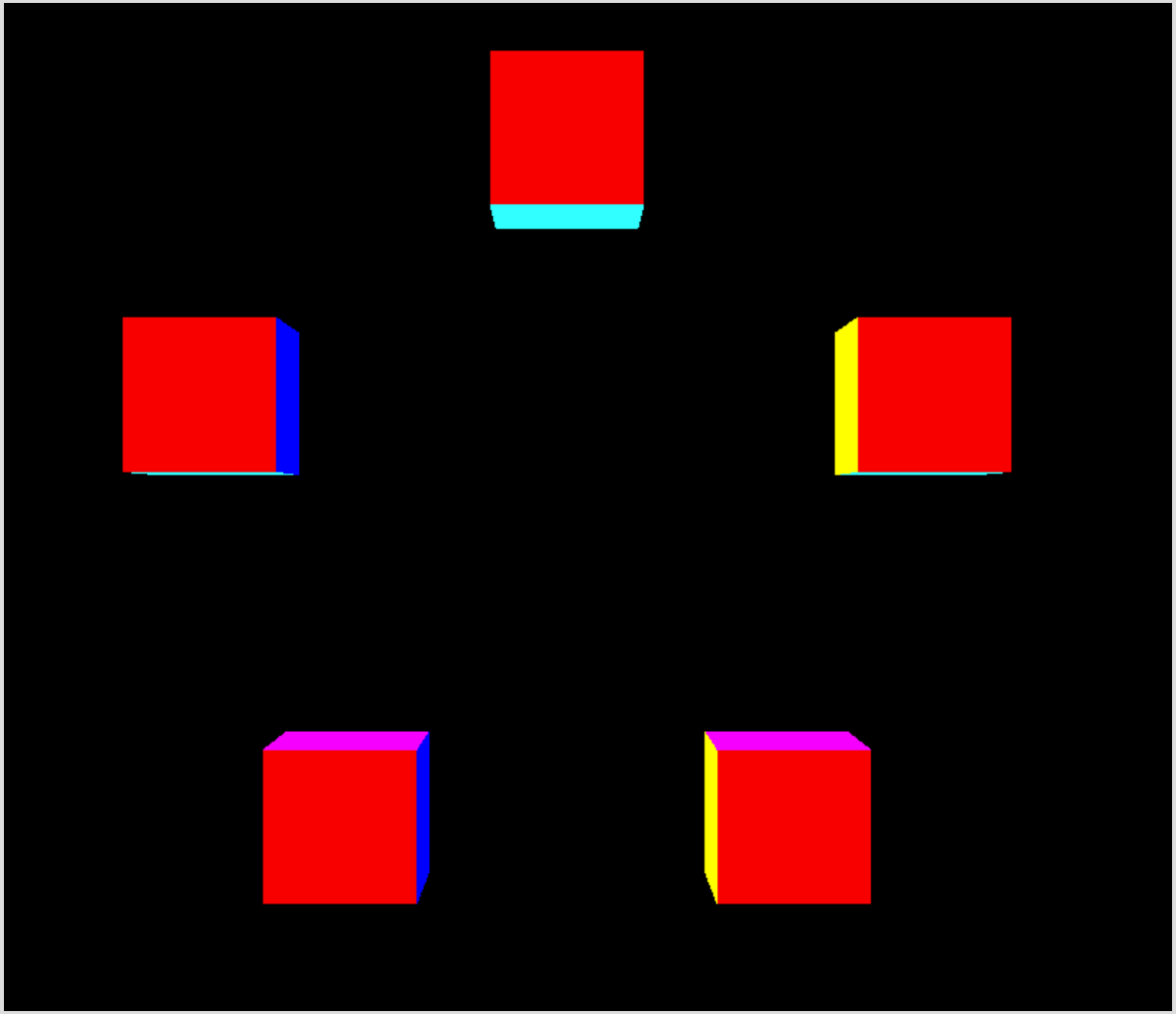

$$T = \begin{pmatrix} 1 & 0 & \cos(\theta) \\ 0 & 1 & \sin(\theta) \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & \cos(\theta) \\ 0 & 1 & \sin(\theta) \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + \cos(\theta) \\ y + \sin(\theta) \\ 1 \end{pmatrix}$$

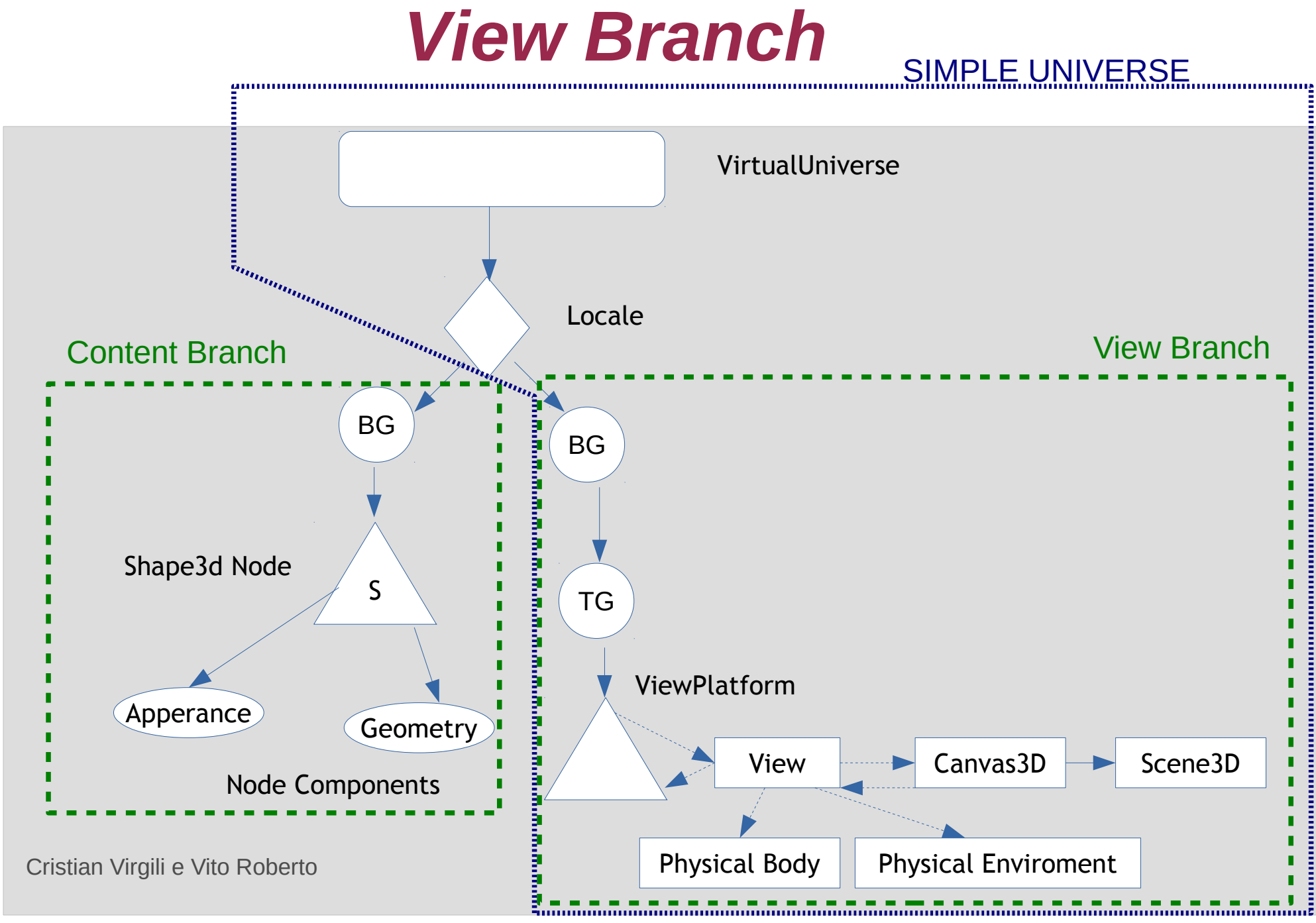
# Serie di cubi



# *Risultato per n = 5*



# View Branch


SIMPLE UNIVERSE
VirtualUniverse
Locale
Content Branch
View Branch
BG
BG
Shape3d Node
S
TG
Apperance
Geometry
ViewPlatform
Node Components
View
Canvas3D
Scene3D
Physical Body
Physical Enviroment

# *View Branch*

Obiettivo del View Branch:

-Implementare il modello dell'osservatore.

- L'implementazione è separata e indipendente da quella degli oggetti 3D (Content Branch).

# *View Branch*

Il View Branch è composto da:

- ViewPlatform:
indica da dove viene vista la scena (posizione e orientamento).

- View:
indica come viene vista la scena (proiezione, antialiasing, etc.)

- Canvas3D:
il componente in cui viene visualizzata la proiezione della scena.

- Screen3D:
proprietà fisiche dello schermo su cui si proietta.

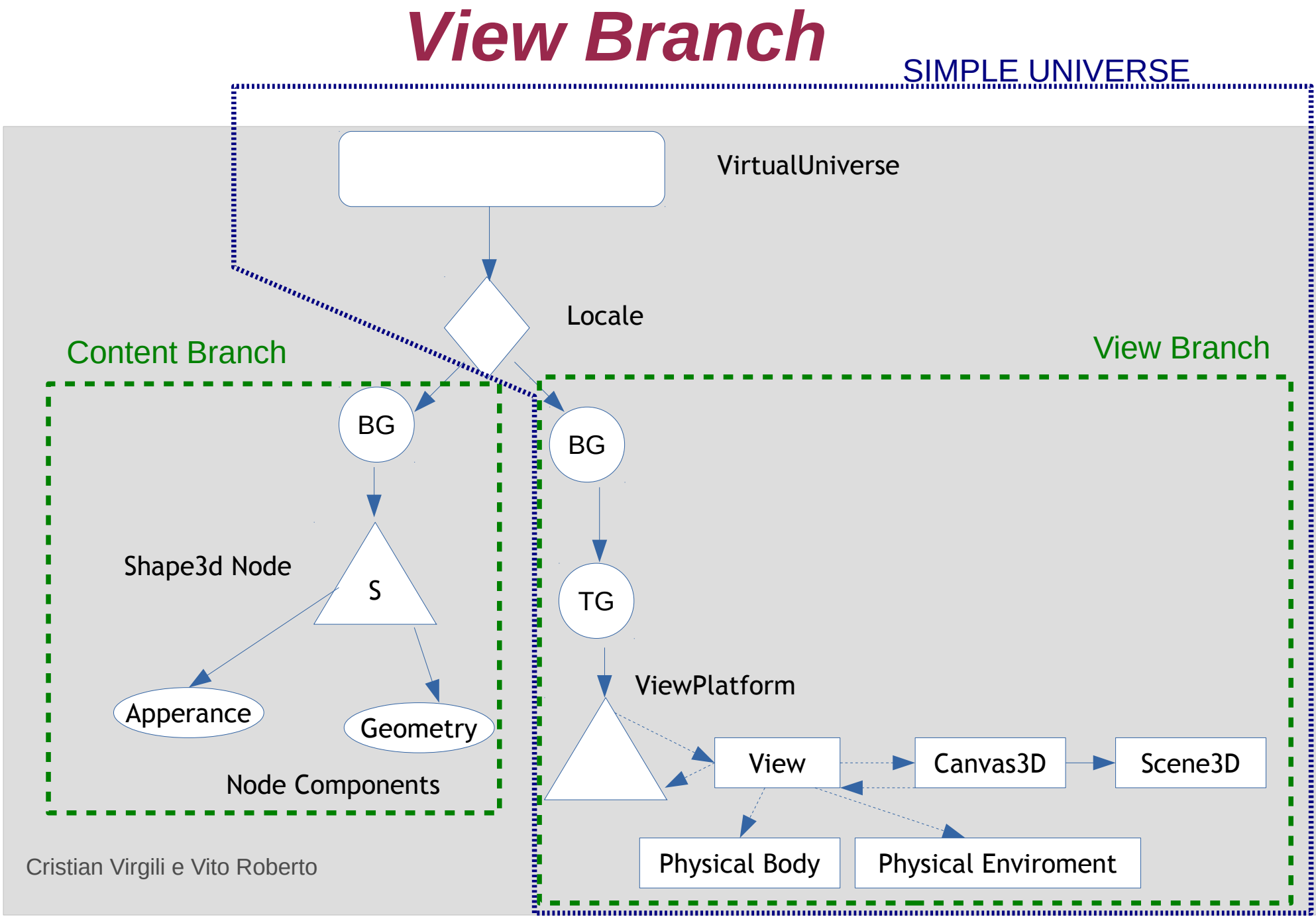
# View Branch

Il View Branch è composto da:

- PhisicalBody:
  Indica le specifiche della testa dell'osservatore.

- PhisicalEnviroment:
  Sistemti di input, audio, ecc.

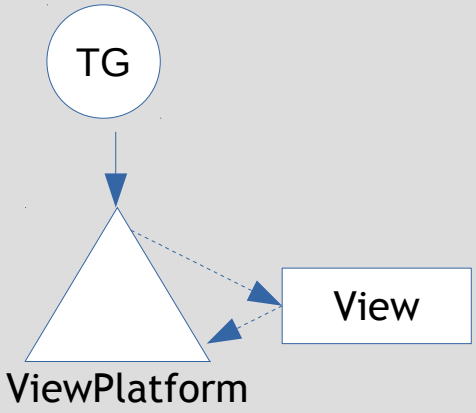
# View Branch


SIMPLE UNIVERSE
VirtualUniverse
Locale
Content Branch
View Branch
BG
BG
Shape3d Node
S
TG
Apperance
Geometry
ViewPlatform
Node Components
View
Canvas3D
Scene3D
Physical Body
Physical Enviroment

# *View Branch*
# *View Platform*

Per ogni *VirtualUniverse* ci possono essere più *ViewPlatform*.

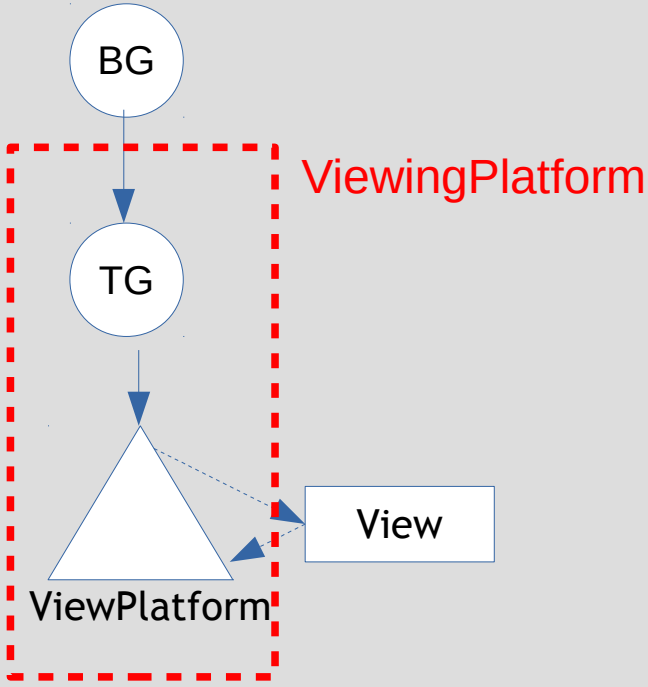
- Ogni *View* può essere collegato ad un unico *ViewPlatform*.



# View Branch
# ViewingPlatform

È un contenitore che raggruppa:
- un ViewPlatform;

- un TransformGroup.



# *View Branch*
# *Come interagire?*

Il SimpleUniverse crea automaticamente i nodi, locale, BG, ecc.

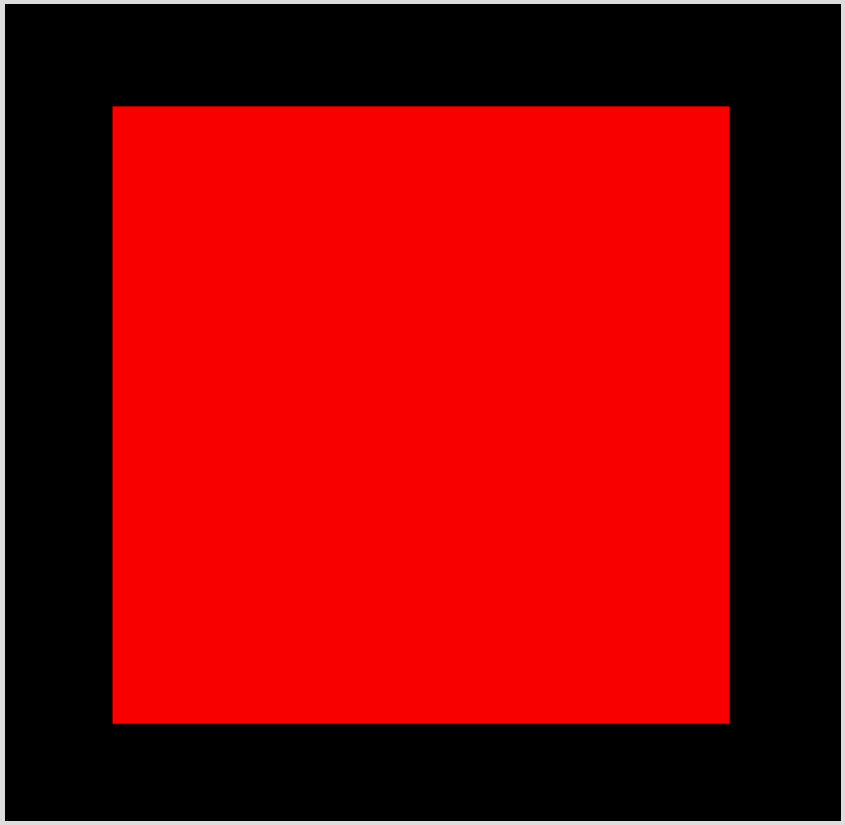
È possibile accedere ai singoli componenti del View Branch.

È possibile alterare i parametri secondo le necessità.

# View Branch
# Visualizzazione base

```
public BaseView() {
                setLayout(new BorderLayout()); //layout manager del container
                //trova la miglior configurazione grafica per il sistema
                GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();
                // Canvas3: si occupa del rendering 3D on-screen e off-screen
                Canvas3D canvas3D = new Canvas3D(config);
                add("Center", canvas3D);
                BranchGroup scene = createSceneGraph();    // creazione del sottografo principale
                scene.compile();
                //Creazione del SimpleUniverse
                SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
                simpleU.getViewingPlatform().setNominalViewingTransform();
                simpleU.addBranchGraph(scene);
          }
/**
 * Funzione che crea il sottografo
 * @return il BranchGroup da aggiungere al SimpleUniverse
 */

 public BranchGroup createSceneGraph() {
   BranchGroup BG = new BranchGroup();
   TransformGroup transform = new TransformGroup(); // crea oggetto TG
     transform.addChild(new ColorCube(0.3));  //aggiungo al TG come figlio il cubo
   BG.addChild(transform); //aggiunge l'oggetto TG come figlio del BrachGruop
  return BG;
 }
```

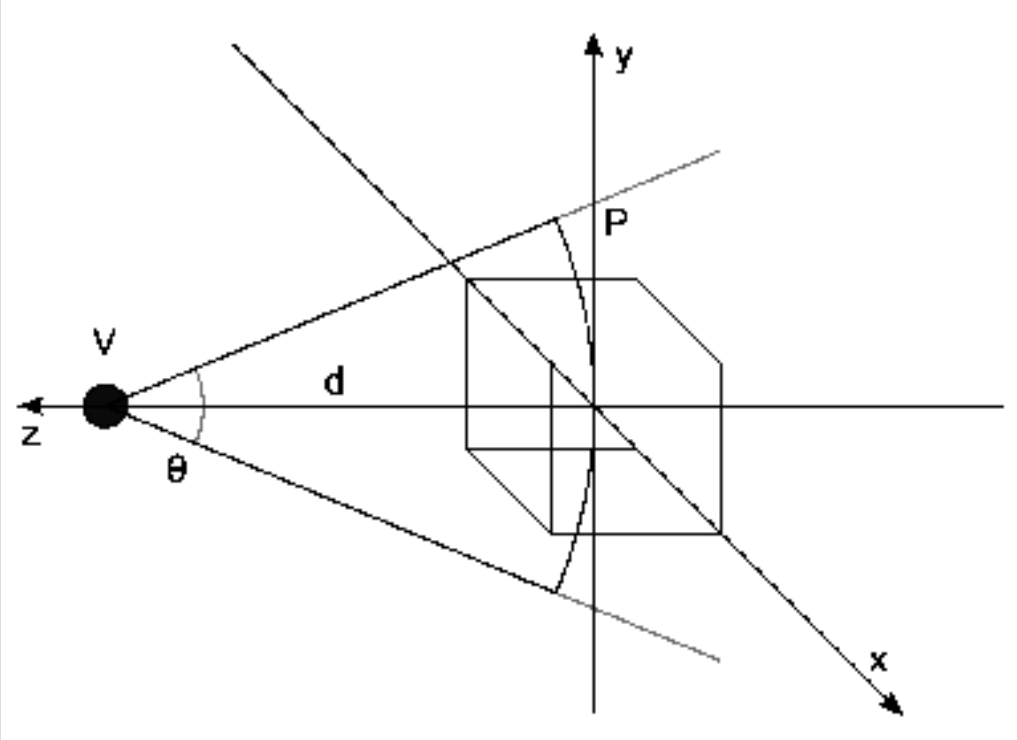
# View Branch
# Visualizzazione base



Cristian Virgili e Vito Roberto

# View Branch
# Visualizzazione base

$$V = (0, \quad 0, \quad d) \qquad P = \left(0, \quad d * \tan\left(\frac{\theta}{2}\right), \quad 0\right)$$




Il campo visivo di default è di π/4 radianti.

# View Branch
# Visualizzazione base

```
Il posizionamento utilizzato fin'ora:
simpleU.getViewingPlatform().setNominalViewingTransform ( ) ;

Equivale al seguente codice:

Ampiezza del campo visivo attuale.
View view = simpleUniverse.getViewer().getView();
double fieldOfView = view.getFieldOfView(); // 0.25*Math.PI

// Trasformazione per traslare l 'osservatore lungo l 'asse z
// in modo da poter inquadrare un quadrato 2x2 sul
// piano xy e centrato nell'origine.
Transform3D viewTransform = new Transform3D ( ) ;
double distance = 1.0/Math.tan(fieldOfView/2.0);
viewTransform.setTranslation(new Vector3d(0.0, 0.0, distance));


//Attivazione della trasformazione appena ricavata .
ViewingPlatform vp = simpleUniverse.getViewingPlatform();
TransformGroup vtg = vp.getViewPlatformTransform ( ) ;
vtg.setTransform (viewTransform ) ;
```

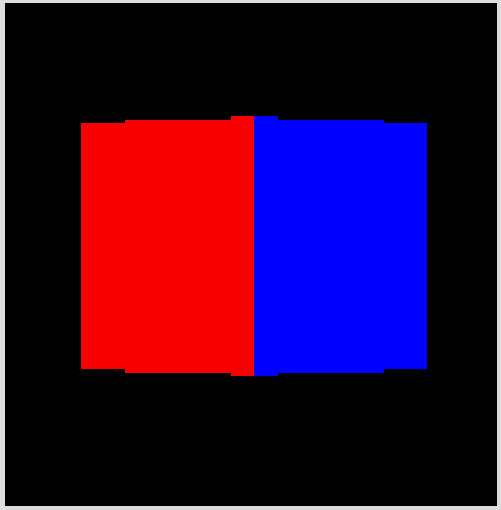
# View Branch
# Esercizio

Partendo da una scena composta da un singolo ColorCube centrato nell'origine, ruotare la telecamera sull'asse Y di PI/4
e traslarla di (0,0,10) unità.

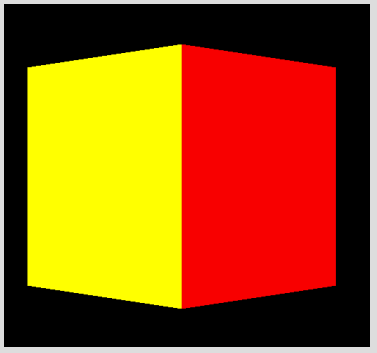
```
Transform3D t3d = new Transform3D();
t3d.setTranslation(new Vector3d(0,0,10));
Transform3D t3d2 = new Transform3D();
t3d2.rotY(Math.PI/4);
t3d2.mul(t3d);
TransformGroup vtg = simpleU.getViewingPlatform().getViewPlatformTransform();
vtg.setTransform(t3d2);
```

# View Branch
# Esercizio

Risultato:



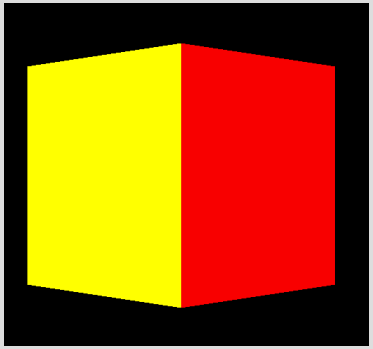
E' corretto?
Cosa sarebbe successo se avessimo fatto ruotare il cubo sull'asse Y?



# View Branch
# Esercizio

Partendo da una scena composta da un singolo ColorCube centrato nell'origine, ruotare la telecamera sull'asse Y di PI/4 e traslarla di (0,0,10).
La visualizzazione deve essere la stessa nel caso avessi fatto ruotare il cubo sulla Y

```
Transform3D t3d = new Transform3D();
t3d.setTranslation(new Vector3d(0,0,10));
Transform3D t3d2 = new Transform3D();
t3d2.rotY(Math.PI/4);
t3d2.invert();
t3d2.mul(t3d);
TransformGroup vtg = simpleU.getViewingPlatform().getViewPlatformTransform();
vtg.setTransform(t3d2);
```



# View Branch

Applicare una trasformazione all'osservatore è visivamente equivalente ad applicare la trasformazione inversa a tutta la scena.

Intuitivamente, quando si applicano le trasformazioni di traslazione o rotazione, all'osservatore ci aspettiamo il risultato corretto.

Quando si combinano più trasformazioni si deve ricordare che:

$$\left(A \cdot B\right)^{-1} \neq A^{-1} \cdot B^{-1}$$
$$\left(A \cdot B\right)^{-1} = B^{-1} \cdot A^{-1}$$

# View Branch
# lookAt()

Tramite combinazioni di Transform3D possiamo posizionare l'osservatore a piacimento.
Transform3D fornisce un modo più semplice nel caso specifico in cui vogliamo puntare verso un oggetto:

```
public void lookAt ( Point3d eye, Point3d center, Vector3d up)
```

- **eye**:  coordinate dell'osservatore.
- **center**: coordinate del punto verso cui guardare.
- **up**: direzione in cui si trova l'alto.

*N.B.: la trasformazione fornita da lookAt() è l'inversa di quella necessaria per posizionare il ViewPlatform.*

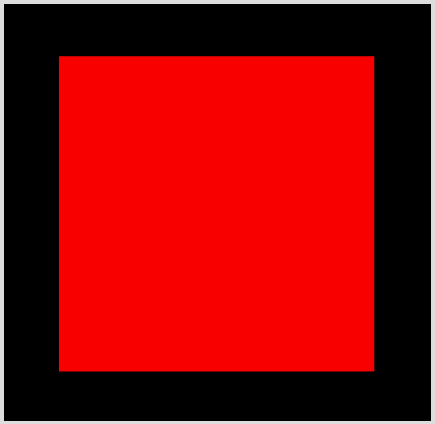
# Posizionamento lookAt()

Il posizionamento assoluto:

```
viewTransform.setTranslation(new Vector3d(0.0, 0.0, distance));
```

equivale al seguente codice:

```
viewTransform.lookAt ( new Point3d (0.0 ,    0.0 ,   distance ) ,
                       new Point3d (0.0, 0.0, 0.0),
                       new Vector3d(0.0 , 1.0, 0.0));
viewTransform.invert( ) ;
```



# Verifica

Possiamo verificare in due modi:

- controllando visivamente (anche da prospettive che permettano di capire l'orientamento).

- verificando la matrice di trasformazione rispetto a casi noti.

Un modo semplice di verificare la matrice è:

System.out.print(viewTransform ) ;

In tutti i casi precedenti il risultato è:

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \frac{1}{\left(\tan\left(\frac{\pi}{8}\right)\right)} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

# *Esercizio in classe*

Creare una scena con un ColorCube e posizionare l'osservatore sotto l'origine rivolto verso l'alto (  ).

Suggerimenti:

- per ottenere la ViewingPlatform:
  ViewingPlatform vp = simpleUniverse.getViewingPlatform();
  TransformGroup vtg = vp.getViewPlatformTransform ( ) ;

- per leggere la trasformazione corrente:
  vtg.getTransform (trans);

- per impostare una nuova trasformazione:
  vtg.setTransform (trans);

# Esercizio 1 – Relazione finale
# Esercizio 3.3

Utilizzare lookAt() per mostrare una scena da diversi punti di vista.

Trovare trasformazioni da applicare al ViewPlatform per cui si hanno:
- 1 punto di fuga;
- 2 punti di fuga;
- 3 punti di fuga.