JAVA 3D

Corso di Immagini e Multimedialità
Lezione 3.4: JAVA3D – Oggetti 3D

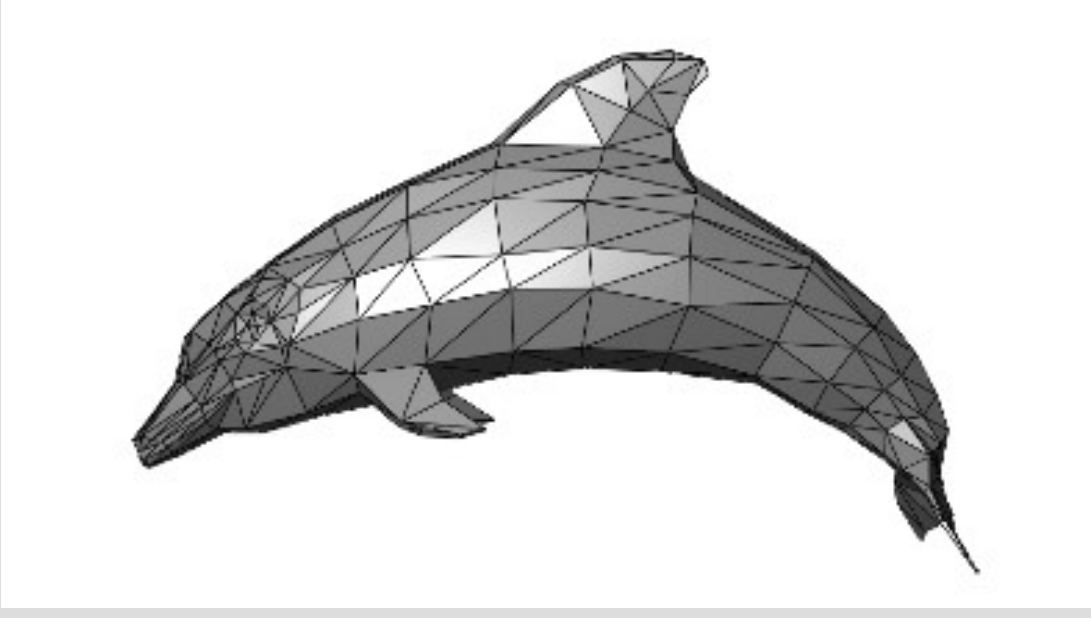
cristian.virgili@uniud.it

OGGETTI 3D

Ogni oggetto 3D è rappresentato da un insieme di poligoni.

- Ogni poligono è rappresentato da un insieme di triangoli.

- Ogni triangolo è identificato da una terna di vertici.



Le classi delle primitive geometriche

Esistono tre modi per creare nuove geometrie:

- usare le classi di utility per box, coni, cilindri e sfere

- specificare le coordinate dei vertici per punti, linee e superfici

- usare i loader di geometrie (che non studieremo)

Primitive geometriche

Le classe ColorCube

Gli esempi che abbiamo fatto usano tutti la classe ColorCube

Usando ColorCube non è necessario specificare
forma e colore

E' una classe semplice, ma con essa non si può andare
molto lontano

Shape3D (1)

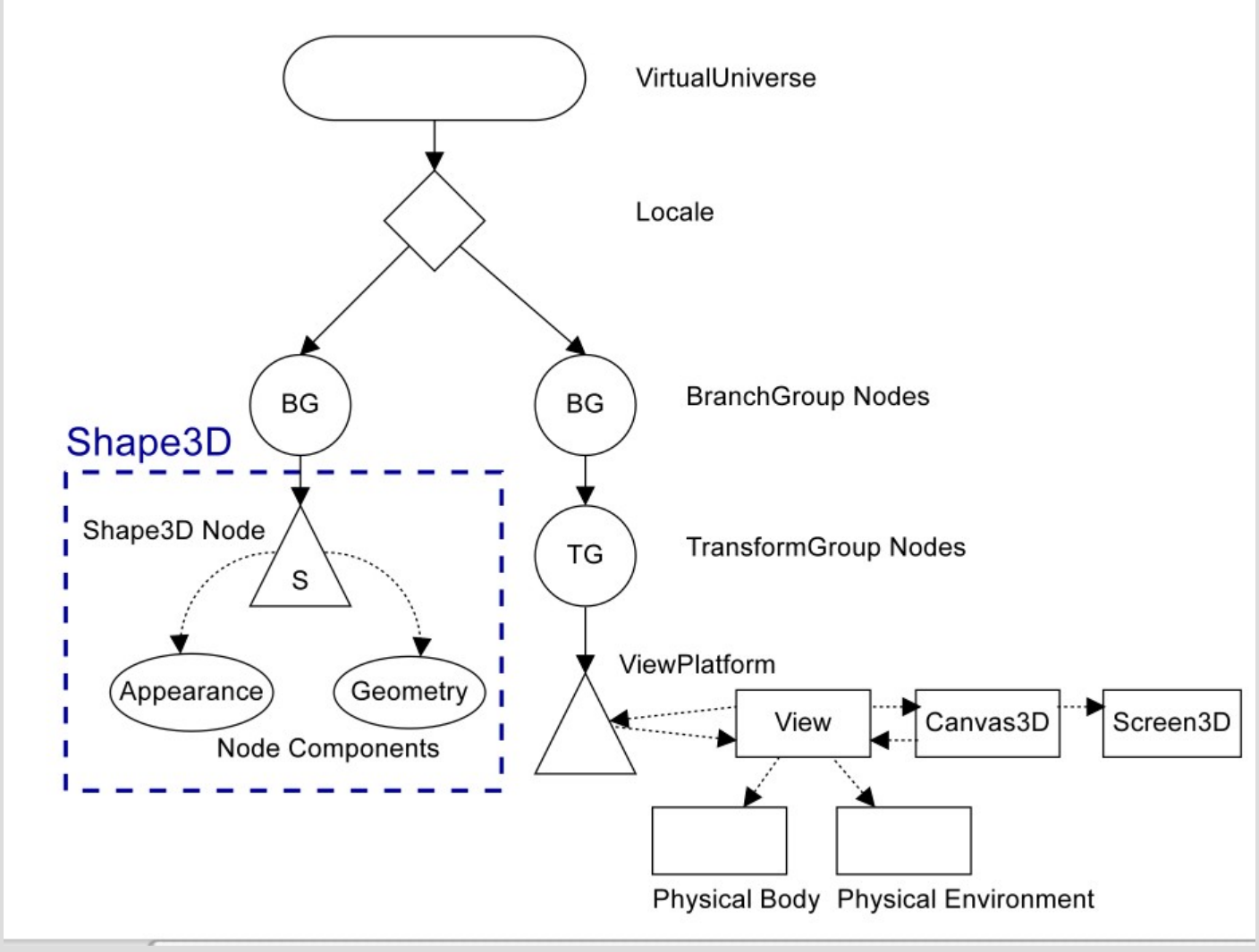
Un nodo *Shape3D* definisce un oggetto visuale

- è una sottoclasse di *Leaf*, quindi può essere solo una foglia in uno scene graph e non contiene informazioni su forma o colore di un oggetto

- tali informazioni sono conservate in oggetti *NodeComponent* collegati allo *Shape3D*

Uno *Shape3D* può riferirsi ad un componente *Geometry* ed ad un componente *Appearance*

Shape3D (2)


VirtualUniverse
Locale
BG
BG
BranchGroup Nodes
Shape3D
Shape3D Node
S
TG
TransformGroup Nodes
Appearance
Geometry
Node Components
ViewPlatform
View
Canvas3D
Screen3D
Physical Body
Physical Environment

Shape3D (3)

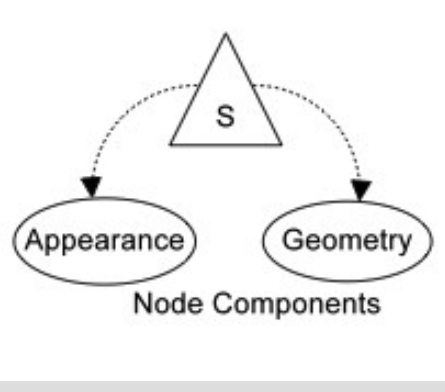
- shape3D è una foglia.

- Il suo aspetto è descritto da due NodeComponent.

    - **Appearance**: per descrivere l'aspetto (colore, materiale, etc.).

    - **Geometry**: per descrivere la forma (linee, superfici, etc.).

- I **NodeComponent** sono legati a Shape3D da riferimenti. Non sono figli.



Shape3D (4)

Per definire un oggetto visuale bastano:

uno **Shape3D** ed un componente **Geometry**
- inoltre uno Shape3D può fare riferimento anche ad un componente *Appearance*

- Finché uno Shape3D non è né vivo né compilato, possono essere cambiati i riferimenti ad i suoi componenti

- Se lo Shape3D è vivo o compilato i suoi riferimenti possono essere cambiati solo se le rispettive **capability** sono state impostate

Shape3D (5)

Esempi di capability per lo *Shape3D* sono:

- ALLOW_GEOMETRY_READ
- ALLOW_GEOMETRY_WRITE
- ALLOW_APPEARANCE_READ
- ALLOW_APPEARANCE_WRITE
- ALLOW_COLLISION_BOUNDS_READ
- ALLOW_COLLISION_BOUNDS_WRITE.

Definire nuovi oggetti visuali

Se un particolare oggetto visuale (un albero, una casa, un avatar, ecc.) deve apparire più volte in un universo virtuale conviene definirne una classe per poterlo creare più volte con facilità

Ci sono molti modi per progettare una classe di un oggetto visuale

Vediamone uno...

Esempio base derivazione Shape3D

```
abstract class VisualObject extends Shape3D {

        protected Geometry geometry;
        protected Appearance appearance;

// Impostazione       dei NodeComponent .
public VisualObject() {
        geometry=createGeometry();
        appearance=createAppearance();
        setGeometry(geometry) ;
        setAppearance(appearance) ;
}
// Metodo per creare la geometria.
protected abstract Geometry createGeometry();

// Metodo per creare l'aspetto .
private Appearance     createAppearance ( )  {
        Appearance app = new Appearance ( ) ;

        //POLYGON_LINE - the polygon is rendered as lines drawn between consecutive vertices.
        //CULL_BACK - culls all front-facing polygons. The default.
        app.setPolygonAttributes(new PolygonAttributes(PolygonAttributes.POLYGON_LINE,
                                                       PolygonAttributes.CULL_BACK,0)
        );
        return app;
}
```

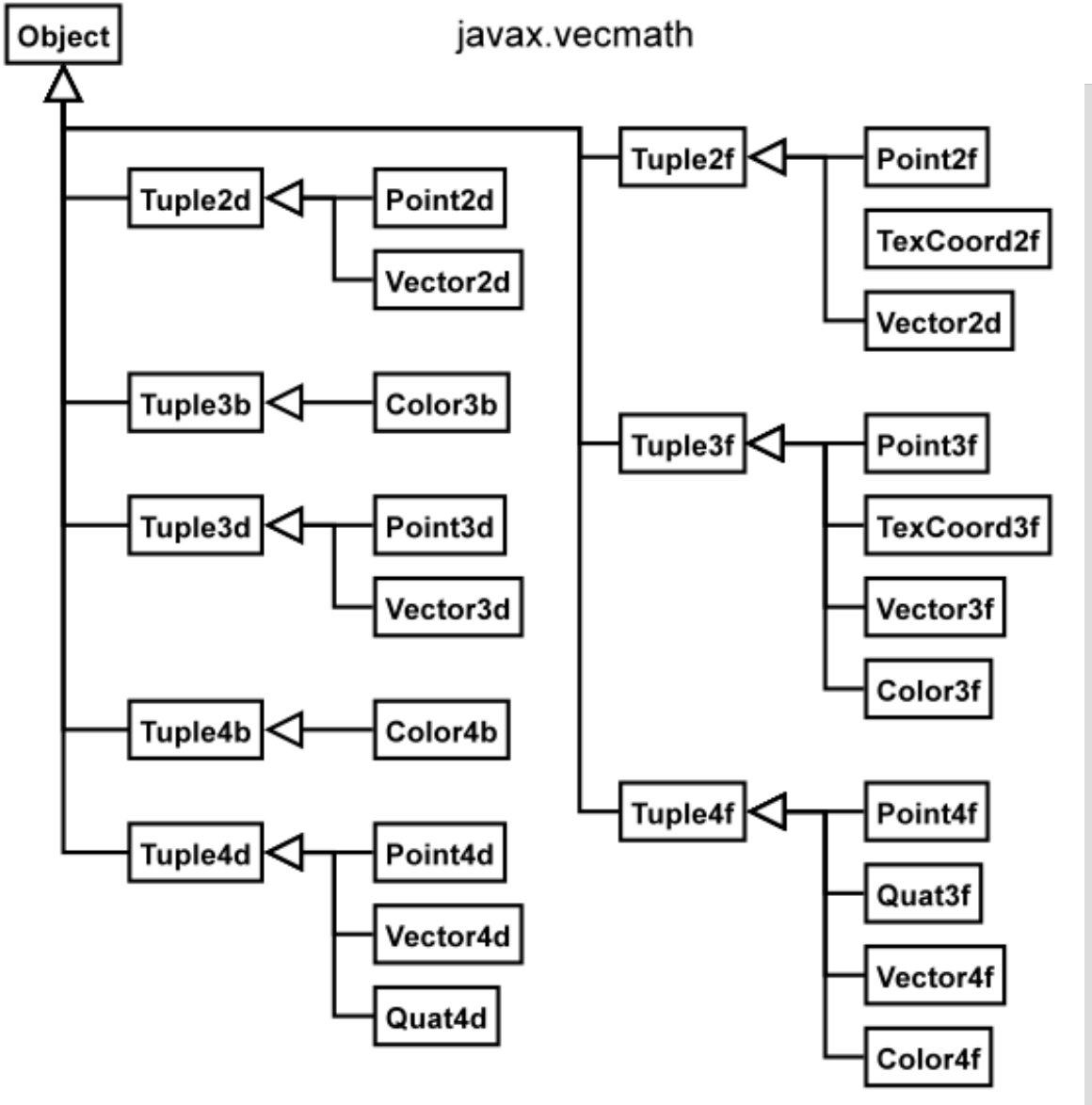
Le classi matematiche

- Per la creazione di oggetti visuali è necessario usare la classe *Geometry* od una sua sottoclasse

- Le sottoclassi di *Geometry* descrivono primitive basate sui vertici come punti, linee e poligoni

- Per poter studiare tali classi bisogna prima introdurre alcune classi matematiche: *Point**, *Color**, *Vector** e *TexCoord**

Tuple


javax.vecmath
Object
Tuple2d
Point2d
Vector2d
Tuple3b
Color3b
Tuple3d
Point3d
Vector3d
Tuple4b
Color4b
Tuple4d
Point4d
Vector4d
Quat4d
Tuple2f
Point2f
TexCoord2f
Vector2d
Tuple3f
Point3f
TexCoord3f
Vector3f
Color3f
Tuple4f
Point4f
Quat3f
Vector4f
Color4f

Le classi matematiche

Ogni vertice di un oggetto visuale può specificare fino a quattro oggetti matematici rappresentanti:

- coordinate (*Point**)

- colori (*Color**)

- superfici normali (*Vector**)

-coordinate di texture (*TexCoord**)

Le classi matematiche

Le coordinate (*Point**) sono necessarie per posizionare ogni vertice e non possono essere omesse; gli altri dati sono opzionali

Ad esempio:
- si può definire un colore (*Color**) ad ogni vertice
- per abilitare le risposte alle luci sono necessarie le superfici normali (*Vector**)
- per il texture mapping sono necessarie le coordinate di texture (*TexCoord**)

*Point**

Rappresenta un punto nello spazio. Può rappresentare:

- un vertice

- la posizione di una sorgente luminosa

- la posizione di una sorgente sonora

- altri dati posizionali

*Color**

Rappresenta un colore per un vertice, una proprietà del materiale, una nebbia, ecc.

Rappresenta un colore in uno di questi spazi:

- (*Color3**)RGB (red, green, blue):
  gamma dei colori visualizzabili

- (*Color4**)RGBA (red, green, blue, alpha):
  colori con trasparenza.

Vector

Rappresenta una direzione:

- un asse di rotazione

- la normale di una superficie

- la direzione di un raggio luminoso

- una velocità

- etc.

*TextCoord**

Gli oggetti *TexCoord* rappresentano le coordinate di una texture ad un vertice

Esistono 2 classi di *TexCoord**

- *TexCoord2f*: conserva le coordinate di una coppia (s,t)
- *TexCoord3f*: conserva le coordinate di una tripa (s,t,r)

Definire nuovi oggetti visuali (2)

Utilizzare lo *Shape3D* come base per la creazione di nuovi oggetti visuali rende tale lavoro più semplice

Un oggetto visuale può essere utilizzato esattamente come è stata utilizzata precedentemente la classe *ColorCube*

Una volta creato un oggetto visuale può essere inserito in qualsiasi gruppo dello scene graph esattamente come il *ColorCube*

Definire nuovi oggetti visuali (3)

Un altro modo per creare nuovi oggetti visuali è definire una classe contenitore non derivata dalle classi Java3D

In questo approccio l'oggetto visuale dovrà:
- contenere un oggetto (root) *Node* o *Shape3D*

- definire almeno un metodo che restituisce un riferimento all'oggetto root

Questa tecnica è leggermente più difficile, ma offre maggiore flessibilità

Definire nuovi oggetti visuali (4)

Un ulteriore modo per creare oggetti visuali è quello usato dalle classi *Box*, *Cone*, *Cylinder e Sphere* (package **com.sun.j3d.utils.geometry**)

Ogni classe estende la classe *Primitive* che (a sua volta) estende *Group*

Le classi Geometry (1)

In Computer Graphics, tutto (da un semplice triangolo fino ad un jumbo jet) è modellato con dati “vertex-based”

In Java3D, ogni oggetto *Shape3D* dovrebbe chiamare il suo metodo *setGeometry()* per referenziare una o più *Geometry*

Per essere più precisi *Geometry* è una superclasse astratta e gli oggetti referenziati sono istanze di sue sottoclassi

Le classi Geometry (2)

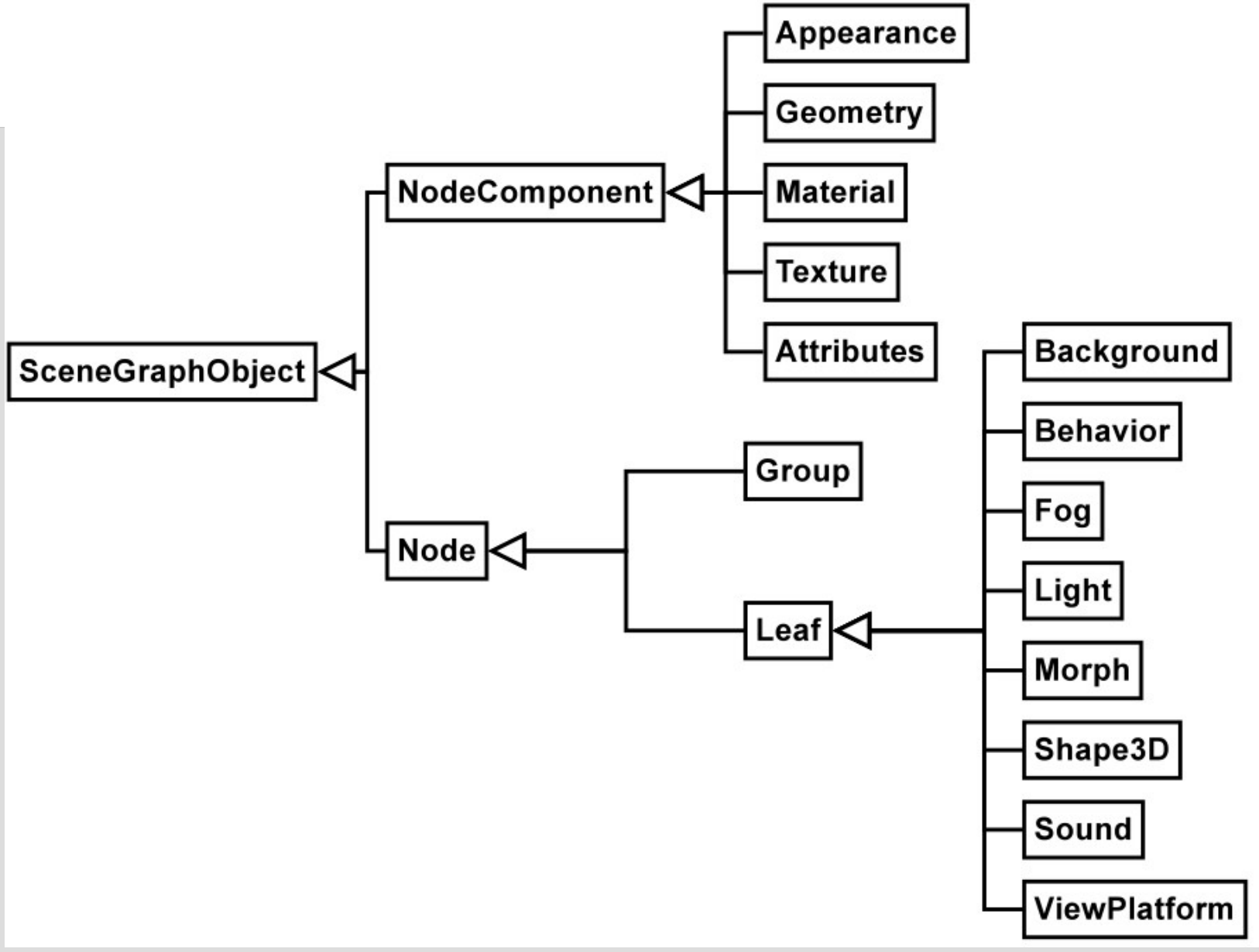
Le sottoclassi di *Geometry* possono essere suddivise in tre categorie:

- "non-indexed vertex-based" (ogni volta che un oggetto è renderizzato, i suoi vertici possono essere usati solo una volta)

- "indexed vertex-based" (ogni volta che un oggetto è renderizzato, i suoi vertici possono essere usati più volte)

-altri oggetti (le classi *Raster*, *Text3D* e *CompressedGeometry*)

Gerarchia di Geometry



Geometry Array (1)

Le sottoclassi di *Geometry* posso essere usate per specificare punti, linee e poligoni pieni (triangoli e quadrilateri)

Queste primitive "*vertex-based*" sono sottoclassi della classe *GeometryArray*

Come dice il nome stesso la classe *GeometryArray* conserva i suoi dati in degli array

Geometry Array (2)

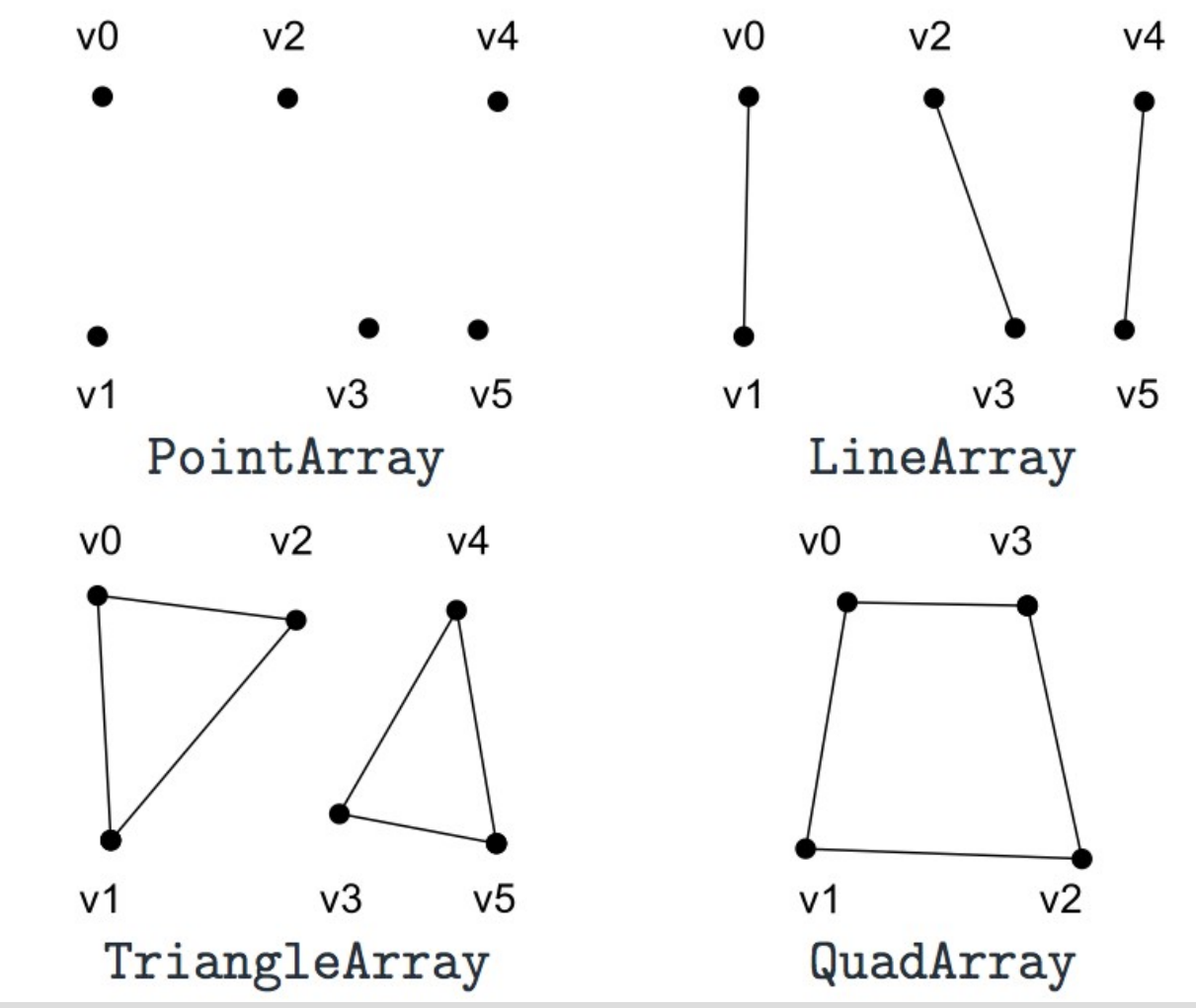
- Per esempio, se un oggetto *GeometryArray* deve specificare un triangolo, deve essere utilizzato un array di tre elementi: uno per ogni vertice

- Ogni elemento di questo array conserva le coordinate del suo vertice (che possono essere definite ad esempio con un oggetto *Point**)

- In aggiunta possono essere definiti altri tre array per memorizzare colore, superficie normale e coordinate delle texture

GeometryArray (1)

Classe derivata da Geometry per la definizione di array di elementi geometrici. È la base di:



Costruttore di Geometry Array

Le classi derivate da *GeometryArray* forniscono un costruttore così definito:

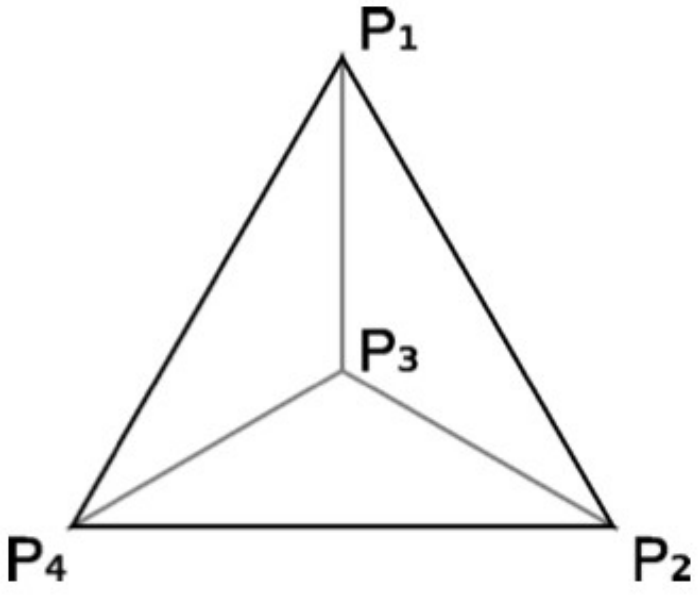
GeometryArray (int vertexCount ,
int vertexFormat)

- *vertexCount*: numero di vertici necessari.
- *vertexFormat*: combinazione di flag per descrivere i dati forniti, per ora ci interessa solo *Geometry.COORDINATES*.

Metodi di *GeometryArray*:

```
public void setCoordinates( int index,
                            double[] coordinates);

public void setCoordinates( int index,
                            point3f[] coordinates);
```

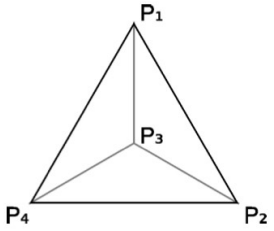
Esempio di Tetraedo



$$P_1 = (1, 1, 1)$$
$$P_2 = (-1, -1, 1)$$
$$P_3 = (-1, 1, -1)$$
$$P_4 = (1, -1, -1)$$

```
Partite da VisualObject
USIAMO:

TriangleArray       triangles ;
```

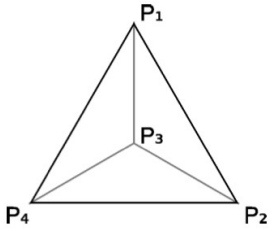
Esempio di Tetraedro



$$P_1 = (1, 1, 1)$$
$$P_2 = (-1, -1, 1)$$
$$P_3 = (-1, 1, -1)$$
$$P_4 = (1, -1, -1)$$

```
public class Tetrahedron extends VisualObject{

      private static final Point3d P1 = new Point3d( 1.0,   1.0, 1.0);
      private static final Point3d P2 = new Point3d(-1.0, -1.0, 1.0);
      private static final Point3d P3 = new Point3d(-1.0,   1.0, -1.0);
      private static final Point3d P4 = new Point3d ( 1.0 , -1.0, -1.0);

      private static      final  Point3d [ ]   faces  =      {
            P1,P3,P2,
            P1,P2,P4,
            P2,P3,P4,
            P1,P4,P3
      };

      protected Geometry createGeometry ( )    {
            TriangleArray triangles ;

//TriangleArray.COORDINATES mi dice che i triangoli sono formati da un ARRAY DI
COORDINATE
            triangles = new TriangleArray(faces.length,
                                          TriangleArray.COORDINATES);
            triangles.setCoordinates(0,faces );
            return triangles ;
      }
```

Esempio di Tetraedro



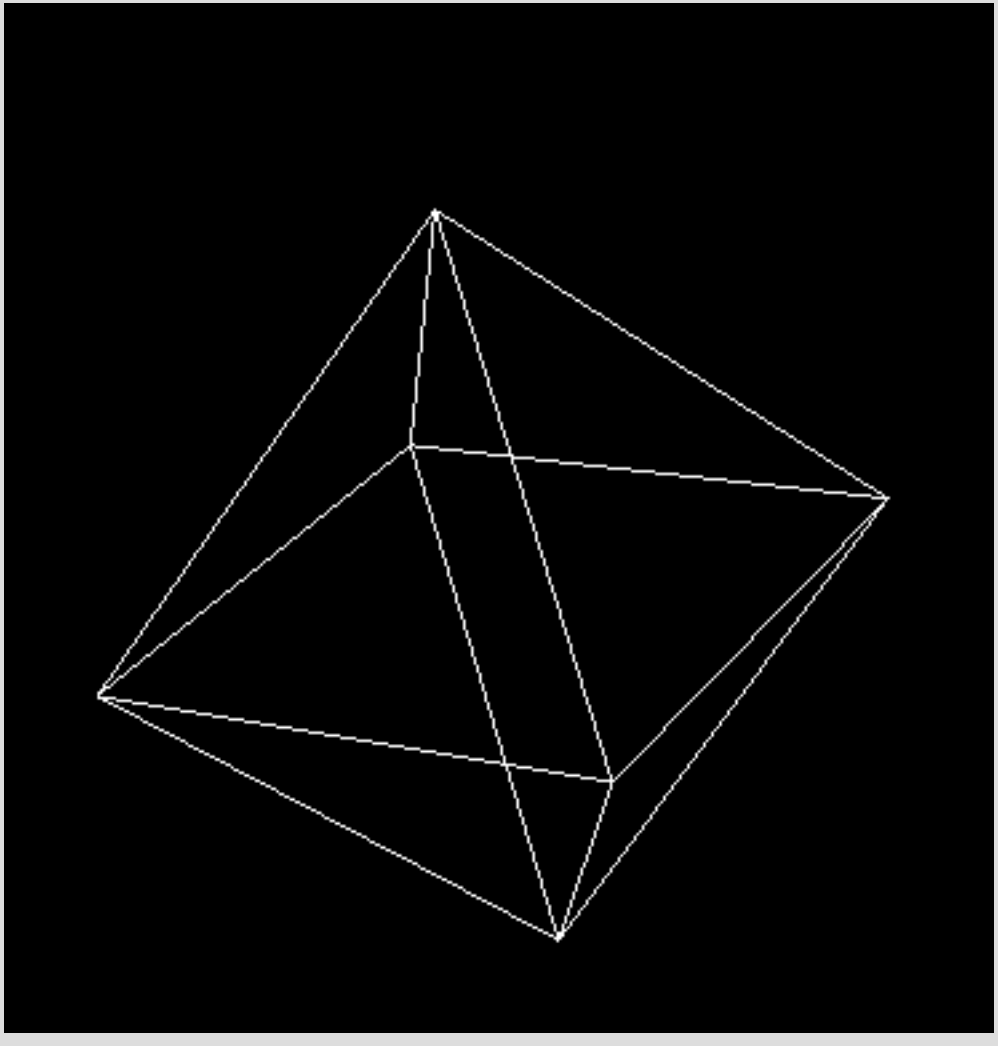
$$P_1 = (1, 1, 1)$$
$$P_2 = (-1, -1, 1)$$
$$P_3 = (-1, 1, -1)$$
$$P_4 = (1, -1, -1)$$

Ora che avete un nuovo oggetto inseritelo nel mondo virtuale...

FATTO? Animiamo la scena...

```
public BranchGroup createSceneGraph() {

                    BranchGroup objRoot = new BranchGroup();
                    Transform3D rotate = new Transform3D();
                    rotate.rotX(Math.PI/4.0d);
                    TransformGroup objRotate = new TransformGroup(rotate);
                    //ROTAZIONE
                    TransformGroup objSpin = new TransformGroup();
                    objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
                    objRoot.addChild(objRotate);
                    objRotate.addChild(objSpin);
                    objSpin.addChild(new ColorCube(0.3));
                    Transform3D xAxis = new Transform3D();
                    xAxis.rotZ(-Math.PI/4.0f);
                    Alpha rotationAlpha2 = new Alpha(-1, 2000);
                    RotationInterpolator rotatorX  = new RotationInterpolator(rotationAlpha2,
                                    objSpin, xAxis, 0.0f,
                                    (float) Math.PI*2.0f);
                    BoundingSphere bounds = new BoundingSphere();
                    rotatorX.setSchedulingBounds(bounds);
                    objSpin.addChild(rotatorX);
                    return objRoot;
Cristian Virgili    }
```

Esercizio: disegnare un ottaedro (octahedron)



Sottoclassi di GeometryArray

Le sottoclassi di *GeometryArray* non permettono il riuso dei vertici

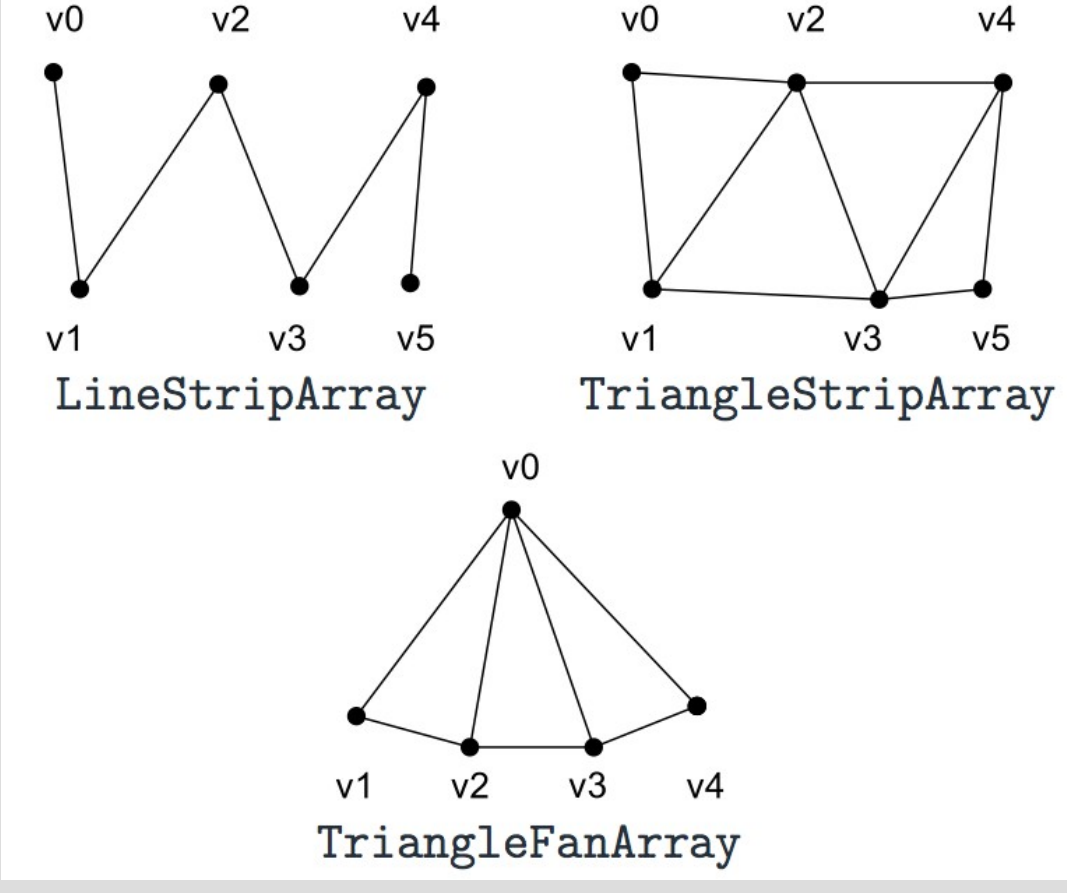
Alcune configurazioni geometriche invitano al riuso dei vertici

Il riuso permettere anche migliori risultati di rendering e performance

In questi casi conviene usare le sottoclassi di *GeometryStripArray*

GeometryStripArray (1)

La figura mostra alcuni esempi delle sottoclassi di *GeometryStripArray*
Notare il diverso riutilizzo dei vertici che ogni classe mette in atto



GeometryStripArray (2)

Il costruttore di *GeometryStripArray* ha un terzo parametro oltre al numero di vertici ed al vertex format

Il terzo parametro è un array di “numero di vertici per strip (striscia)”

Tale parametro rappresenta il numero di vertici che ogni singola striscia della geometria contiene

Costruttore di GeometryStripArray

Le classi derivate da *GeometryStripArray* forniscono sempre un costruttore così definito:

GeometryStripArray (int vertexCount ,
int vertexFormat ,
int[ ] stripVertexCounts)

- *vertexCount*: numero di vertici necessari.

- *vertexFormat*: combinazione di flag per descrivere i dati forniti. Per ora ci interessa solo *GeometryArray.COORDINATES*.

- *stripVertexCounts*: array di conteggio del numero vertici per le diverse strisce. La grandezza di questo array è il numero di strisce separate. La somma degli elementi di questa matrice è il numero totale di vertici validi visualizzati.

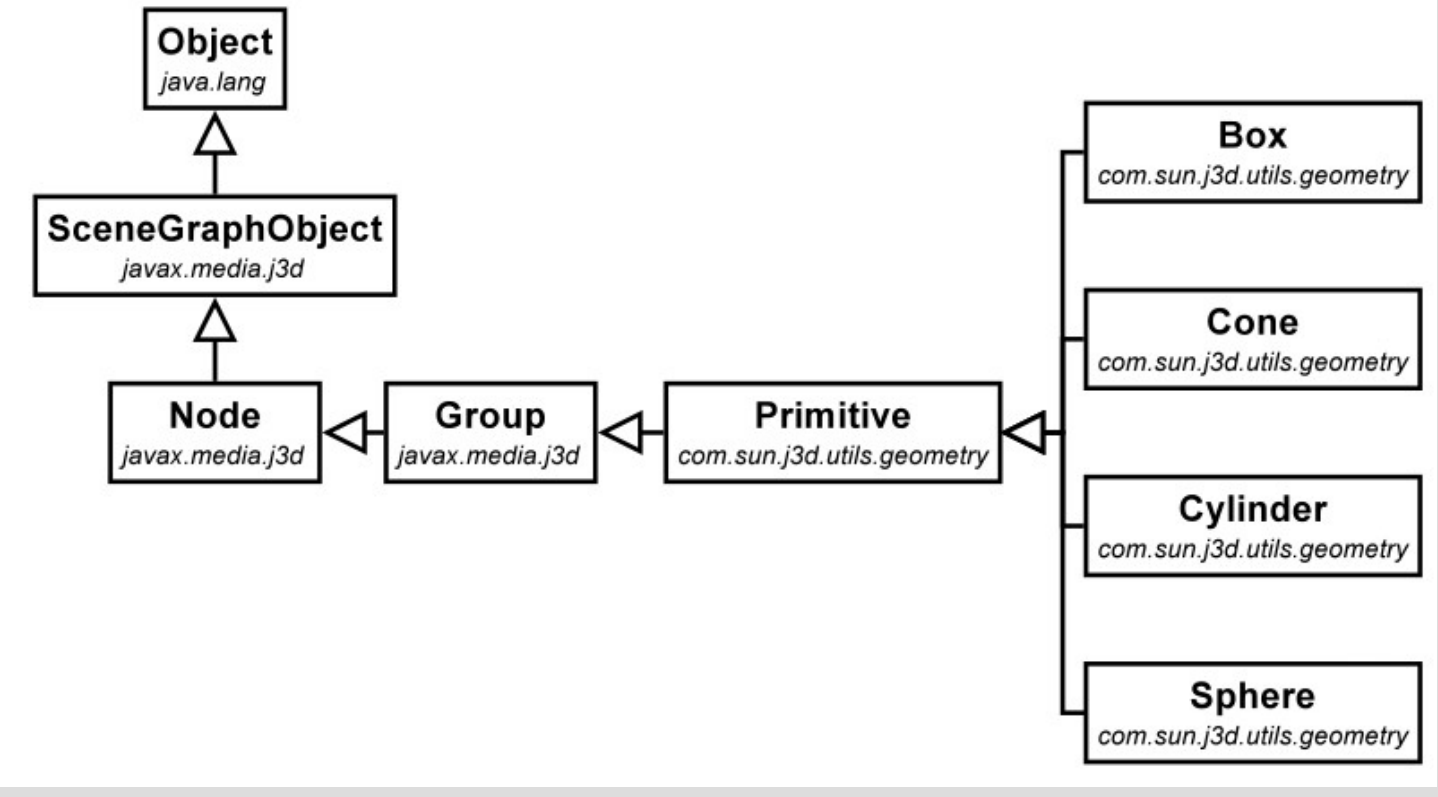
Esempio: il cilindro

```
class MyCylinder extends Shape3D {
public static final       float TOP = 1.0f;
        public static final float BOTTOM = -1.0f;

        protected Point3f v[] = null;
        protected TriangleStripArray triangleStrip = null;
        protected PolygonAttributes polyAttrbutes = new PolygonAttributes ( ) ;
        protected Appearance appearance = new Appearance ( ) ;

public MyCylinder( int    steps ) {
                v = new Point3f[(steps+1)*2];

                for(int i=0; i<steps; i++) {
                        double angle = 2.0*Math.PI*(double)i/(double)steps;
                        float x = (float)Math.sin(angle);
                        float y = (float)Math.cos(angle);
                        v[i*2+0] = new Point3f(x, y, BOTTOM);
                        v[i*2+1] = new Point3f(x,y,TOP);
                }
                v[steps*2+0] = new Point3f(0.0f , 1.0f , BOTTOM);
                v[steps*2+1] = new Point3f(0.0f , 1.0f , TOP);
                int [ ]  stripCounts ={( steps+1)*2};
                triangleStrip = new TriangleStripArray((steps+1)*2,
                        GeometryArray.COORDINATES,stripCounts );
                triangleStrip.setCoordinates(0,v);
                setGeometry( triangleStrip );
                //Impostazione aspetto wireframe .
                polyAttrbutes.setPolygonMode(PolygonAttributes.POLYGON_LINE ) ;
                polyAttrbutes.setCullFace(PolygonAttributes.CULL_NONE ) ;
                appearance.setPolygonAttributes (polyAttrbutes) ;
                setAppearance ( appearance ) ;
        }
```

Primitive (1)

-È un insieme di oggetti che descrivono delle geometrie.

- Sono prive di aspetto.



Primitive (2)
BOX

Costruisce una parallelepipedo.
Le dimensioni di default sono: 2×2×2.

Costruttore di Box:

- Box()

- Box(float xdim, float ydim, float zdim,
  Appearance appearance)

Primitive (3)
CONE

Costruisce un cono.
Il raggio di default è 1, con altezza 2 e 15 suddivisioni sull'asse x.

Costruttore di Cone:
- Cone()

- Cone(float radius, float height)

-Cone(float radius, float height, int primflags, int xdivision, int ydivision, Appearance ap)

Primitive (4)
CILINDER

Costruisce un cilindro.

Il raggio di default è 1, con altezza 2 e 15 suddivisioni sull'asse x.

Costruttore di *Cylinder*:
- *Cylinder*()
- *Cylinder*(**float** radius, **float** height)
- *Cylinder*(**float** radius, **float** height, **int** primflags,
  int xdivision, **int** ydivision, Appearance ap)

```
Cylinder c = new Cylinder();
c.setAppearance(app);
```

Primitive (5)
SPHERE

Costruisce una sfera.

Il raggio di default è 1, con 15 suddivisioni sugli assi.

Costruttore di Sphere:
- Sphere()

- Sphere(float radius)

- Sphere(float radius, int primflags, int divisions,
  Appearance ap)

Le classi utility per le geometrie

- Le classi di utility per le geometrie **NON** definiscono il colore o l'aspetto di un oggetto
    - tali informazioni derivano dal nodo *Appearance*
    - senza un riferimento ad un nodo *Appearance* l'oggetto potrebbe non essere visualizzato o essere visualizzato bianco

- la classe *Primitive* definisce molti valori comuni alle classi *Box*, *Cone*, *Cylinder e Sphere (*ad esempio definisce il numero di poligoni utilizzati per rappresentare le superfici)

OGGETTI COMPOSTI

Abbiamo visto più modi per ottenere un oggetto 3D:

- Basandosi su *Shape3D* per creare una foglia:
  - Costruzione di una foglia creando gli opportuni elementi geometrici.
  - Definizione di un classe derivata che estenda Shape3D.

- Basandosi su *Group* per creare oggetti composti che includano più foglie:
  - Creazione di un gruppo e inserimento degli elementi che costituiscono l'oggetto,
  - Definizione di una classe derivata che implementi Group.

L'estensione di *Shape3D* e *Group* garantisce la migliore riutilizzabilità del codice.

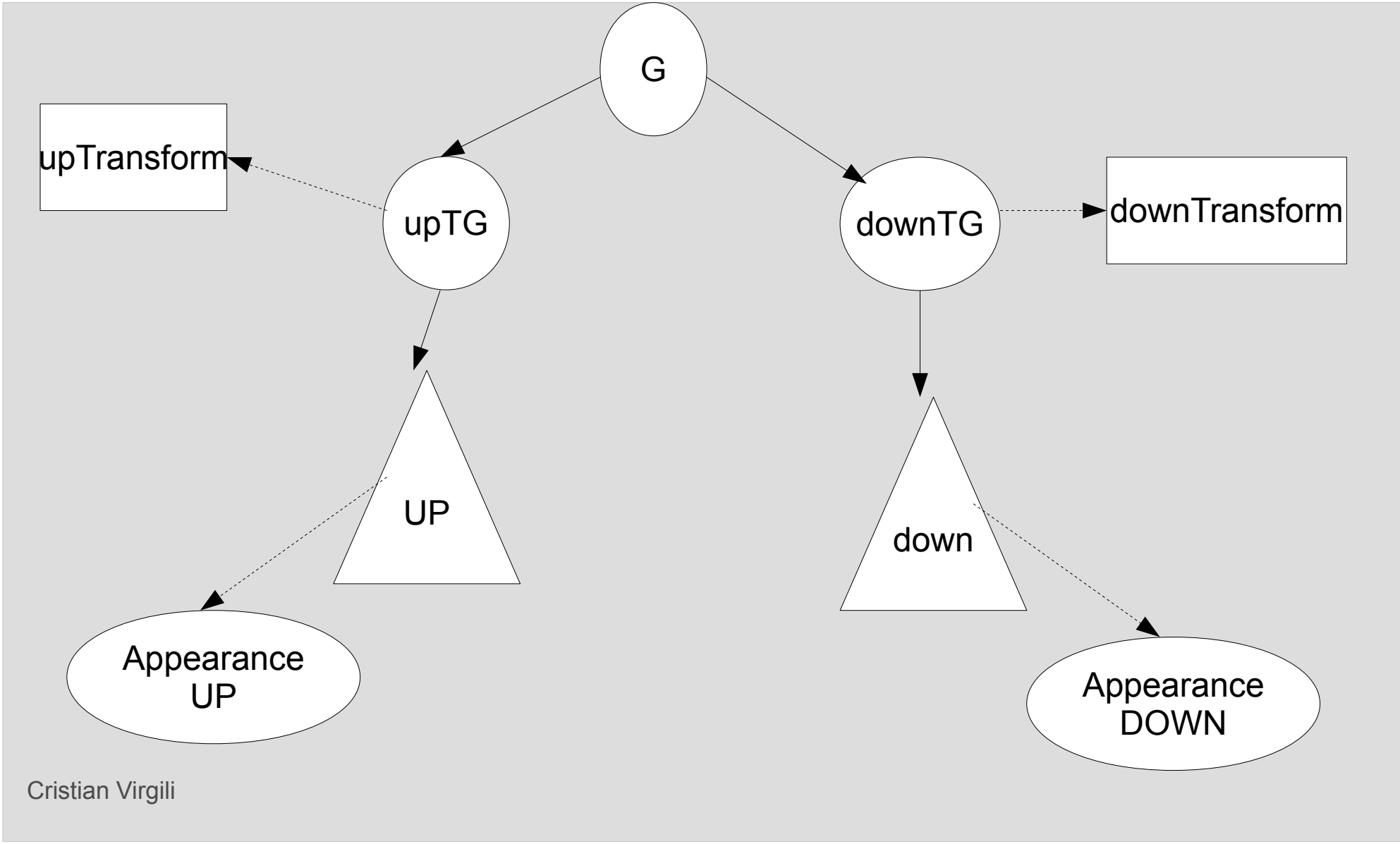
OGGETTI COMPOSTI

	Vantaggi	Svantaggi
Shape3D	Creazione di forme arbitrarie Possibilità di condividere le strutture dati.	Complessità di gestione Impossibilità di manipolare i sottocomponenti.
GROUP	Creazione di scene complesse composte da molti elementi (altri gruppi o foglie). Possibilità di manipolare i sottocomponenti.	Creazione di nuovi oggetti per ogni istanziazione. Flessibilità limitata.

ESEMPIO DI GROUP: TROTTOLA

```
public class Trottola extends Group {
static final protected Appearance appearance = new Appearance() ;

        static final protected Transform3D upTransform = new Transform3D (
                        new Matrix3d(1.0, 0.0, 0.0, 0.0, 0.5, 0.0, 0.0, 0.0, 1.0),
                        new Vector3d(0.0, 0.5, 0.0),
                        1.0);

        static final protected Transform3D downTransform = new Transform3D (
                        new Matrix3d(1.0, 0.0, 0.0, 0.0, -0.5, 0.0 , 0.0, 0.0, 1.0) ,
                        new Vector3d(0.0 , -0.5, 0.0) ,
                        1.0);

        protected TransformGroup upTG = new TransformGroup(upTransform);
        protected TransformGroup downTG = new TransformGroup(downTransform);
        protected Cone upCone = new Cone();
        protected Cone downCone = new Cone() ;

        public Trottola ()      {
                upCone.setAppearance ( appearance ) ;
                downCone . setAppearance ( appearance ) ;
                upTG.addChild (upCone) ;
                downTG.addChild (downCone) ;
                addChild(upTG) ;
                addChild(downTG ) ;
        }
}
```

Scene Graph
Trottola



ESERCIZIO
RELAZIONE FINALE 3.6

- Implementare una classe derivata da Shape3D per la creazione di tronchi di piramide quadrata.

- Implementare una classe derivata da Group per la creazione di una piramide Maya (senza scalinata e porta).