JAVA 3D – INTERAZIONI e ANIMAZIONI

Corso di Immagini e Multimedialità
Lezione 4.2

cristian.virgili@uniud.it

Interazione e Animazione

Fino ad ora gli esempi mostrati visualizzavano mondi piuttosto "statici"

Per rendere un mondo Java3D più interessante ed utile bisogna aggiungere *interazione* ed *animazione*

Si ha **interazione** quando il mondo cambia in risposta ad un'azione dell'utente

Si ha **animazione** quando il mondo cambia senza un intervento diretto dell'utente (e solitamente corrisponde con il passare del tempo)

Interazione e Animazione

Fino ad ora gli esempi mostrati visualizzavano mondi piuttosto “statici”

Per rendere un mondo Java3D più interessante ed utile bisogna aggiungere *interazione* ed *animazione*

Si ha **interazione** quando il mondo cambia in risposta ad un'azione dell'utente

Si ha **animazione** quando il mondo cambia senza un intervento diretto dell'utente (e solitamente corrisponde con il passare del tempo)

La classe Behavior

Scopo di un *Behavior* è modificare uno scene graph in risposta a qualche stimolo

- uno stimolo può essere la pressione di un tasto, un movimento del mouse, la collisione tra oggetti, il passare del tempo, ecc.

I cambiamenti prodotti possono essere l'aggiunta e/o la rimozione di oggetti dallo scene graph, il cambiamento di attributi degli oggetti
- le possibilità sono limitate soltanto dalle capability degli oggetti stessi

Esempi di Behavior

Stimolo	Oggetto da modificare			
	TransformGroup	Geometry	Scene Graph	View
Utente	Interazione	Specifico della applicazione	Specifico della applicazione	Navigazione
Collisione	Gli oggetti cambiano orientamento o posizione	Gli oggetti cambiano aspetto durante la collisione	Gli oggetti spariscono (esplodono)	La vista cambia con la collisione
Tempo	Animazione	Animazione	Animazione	Animazione

Implementazione di un Behavior

Un Behavior personalizzato deve implementare i metodi *initialize()* e *processStimulus()* della classe astratta *Behavior*

- il metodo *initialize()* è invocato quando lo scene graph contenente il *Behavior* diventa vivo

Esso è responsabile delle operazioni di inizializzazione del "trigger" degli eventi e dei valori iniziali del Behavior

Il trigger è specificato come un *WakeupCondition* o una combinazione di *WakeupCondition*

Implementazione di un Behavior (2)

Il metodo *processStimulus*() è invocato dal trigger quando si verifica la condizione specificata
-*processStimulus()* è responsabile della risposta agli eventi

Poiché molteplici eventi posso essere inglobati in un singolo WakeupCondition, questo metodo si occupa anche della decodifica dell'evento
- ad esempio tutte le azioni da tastiera posso essere inglobate in un *WakeupOnAWTEvent*

Solitamente questo metodo resetta anche il trigger

Implementazione di un Behavior (3)

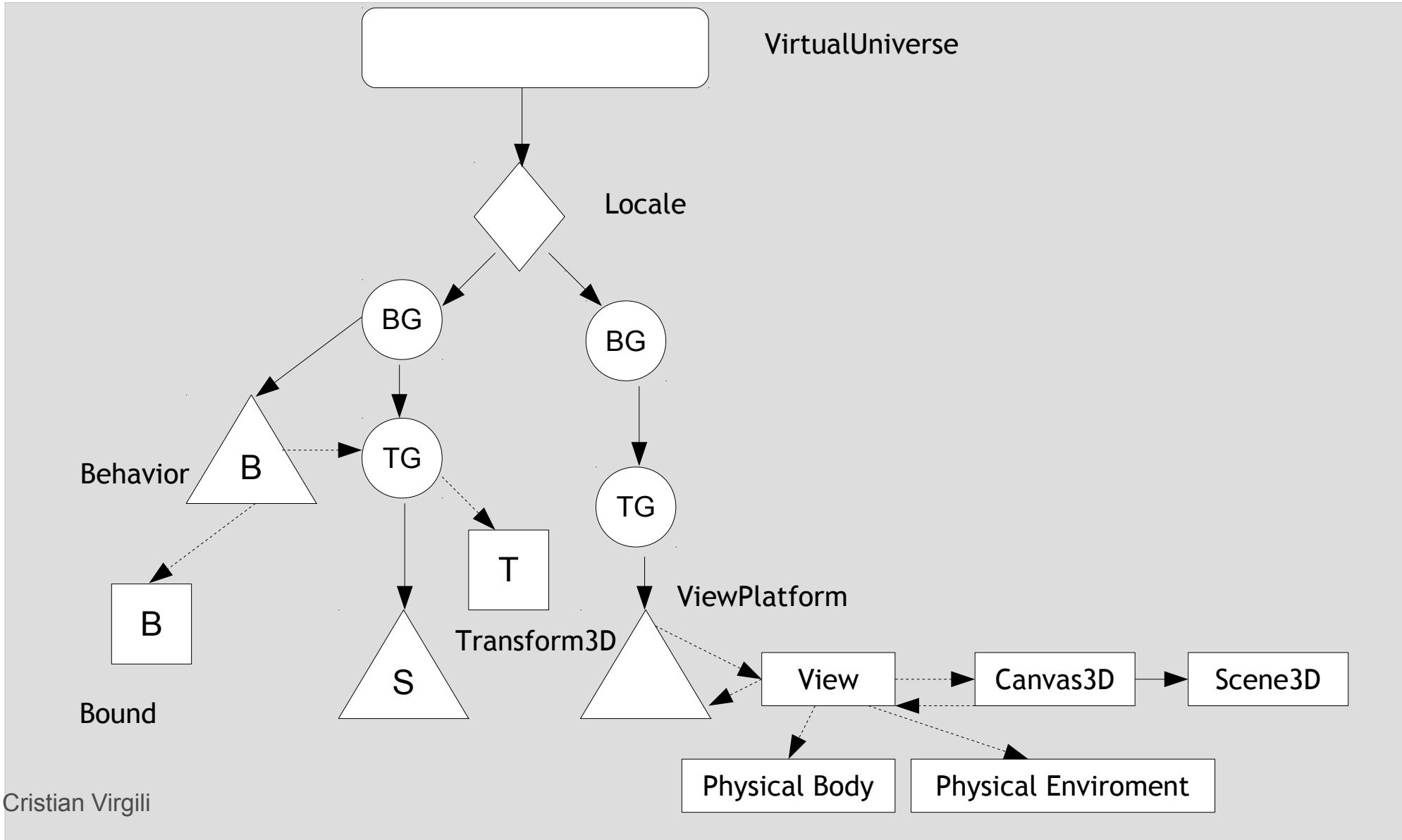
Solitamente per implementare un *Behavior* bisogna seguire i seguenti passi:

- scrivere (almeno) un costruttore: memorizzare un riferimento all'oggetto che si vuole modificare

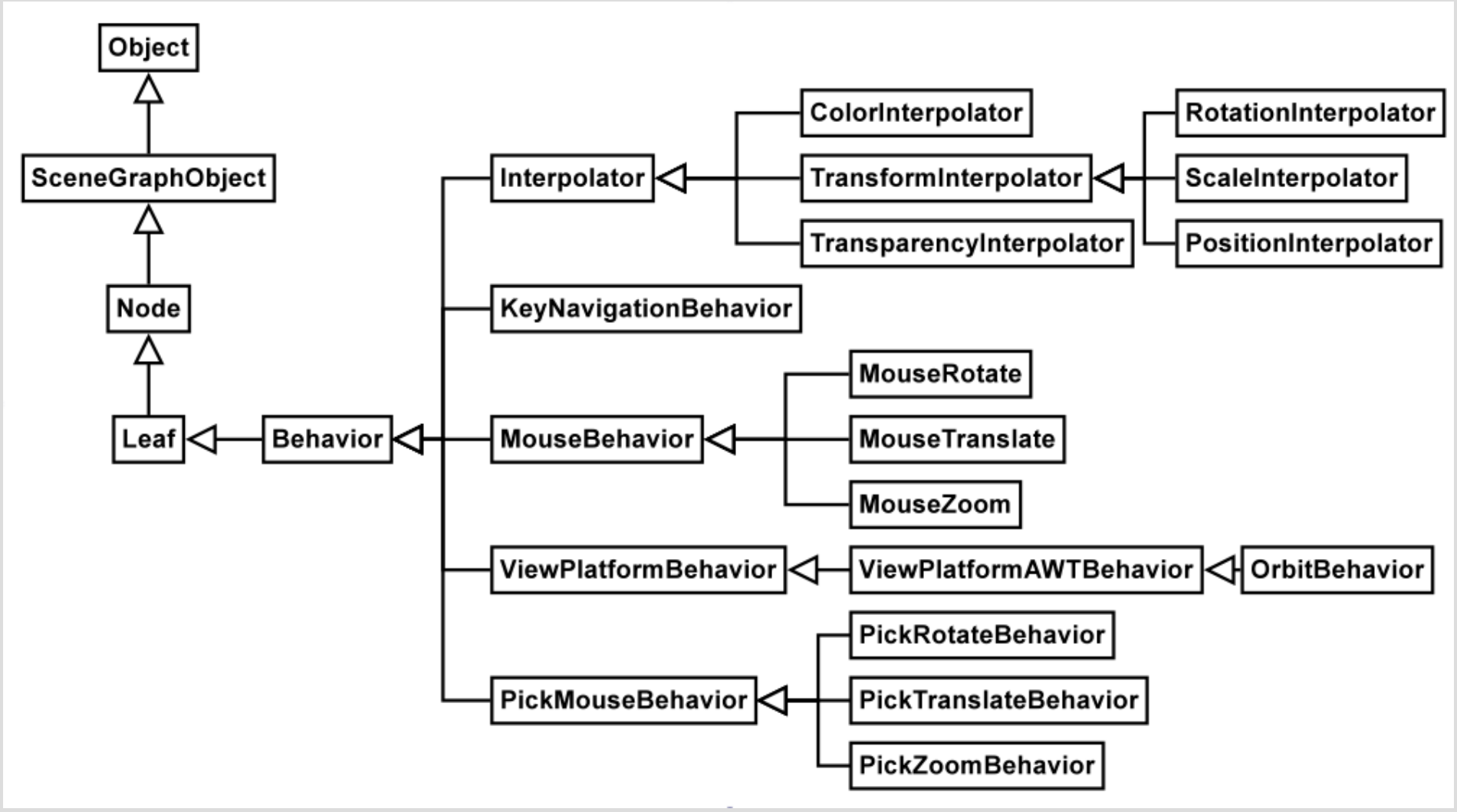
- sovrascrivere il metodo *initialize()*: specificare i criteri iniziali

- sovrascrivere il metodo *processStimulus()*: decodificare la condizione, elaborarla e resettare il trigger

SceneGraph di un Behavior



Behavior


Object
SceneGraphObject
Node
Leaf
Behavior
Interpolator
ColorInterpolator
TransformInterpolator
TransparencyInterpolator
RotationInterpolator
ScaleInterpolator
PositionInterpolator
KeyNavigationBehavior
MouseBehavior
MouseRotate
MouseTranslate
MouseZoom
ViewPlatformBehavior
ViewPlatformAWTBehavior
OrbitBehavior
PickMouseBehavior
PickRotateBehavior
PickTranslateBehavior
PickZoomBehavior

Esempio Behavior

```
//L'implementazione del Behavior personalizzato
class SimpleBehavior_1 extends Behavior
{
  private TransformGroup targetTG;
  private Transform3D rotation=new Transform3D();
  private double angle=0.0;

  public SimpleBehavior_1(TransformGroup targetTG)
  {
    this.targetTG=targetTG;
  }

  public void initialize()
  {
    //Questo Behavior rispondera' ad eventi di tastiera sul key pressed
    this.wakeupOn(new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
  }

  public void processStimulus(Enumeration criteria)
  {
    //Incrementa l'angolo
    angle+=0.1;
    //Evita problemi di overflow
    if (angle>2*Math.PI) angle=0;
    //imposta la rotazione dell'angolo
    rotation.rotY(angle);
    targetTG.setTransform(rotation);
    //Resetta il Behavior per continuare a rispondere ad eventi di tastiera
    this.wakeupOn(new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
  }
```

```
//Crea la scena
  private BranchGroup createSceneGraph()
  {
    //Crea la radice del branch graph
    BranchGroup objRoot=new BranchGroup();

    //Crea un gruppo per le trasformazioni affini
    TransformGroup objSpin=new TransformGroup();

    //Imposta la capacita' di scrivere la trasformazione
    objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

    //Aggiunge al gruppo un cubo colorato
    objSpin.addChild(new ColorCube(0.4));

    //Crea un behavior
    SimpleBehavior_1 rotator=new SimpleBehavior_1(objSpin);

    //Imposta un raggio d'azione del behavior
    BoundingSphere bounds=new BoundingSphere();
    rotator.setSchedulingBounds(bounds);

    //aggiunge l'interpolatore alla gruppo di trasformazione
    objSpin.addChild(rotator);

    //Aggiunge alla radice il gruppo
    objRoot.addChild(objSpin);

    return objRoot;
  }
```

```
 public void processStimulus(Enumeration criteria)
  {
    //Recupera gli stimoli che hanno attivato il behavior
    AWTEvent[] ev=null;
    while (criteria.hasMoreElements())
    {
      Object obj=criteria.nextElement();
      if (obj instanceof WakeupOnAWTEvent)
ev=((WakeupOnAWTEvent)obj).getAWTEvent();
    }



    if (ev!=null)
      for (int i=0;i<ev.length;i++) //scorre tutti gli eventi AWT in
cerca di un evento di tastiera
        if (ev[i] instanceof KeyEvent)
        {
          //Recupera l'evento
          KeyEvent key=(KeyEvent)ev[i];
          //Recupera il codice dell'evento
          int code=key.getKeyCode();
          if (code==KeyEvent.VK_LEFT) //Codice freccia a sinistra
          {
            //decrementa l'angolo
            angle-=0.1;
            //Evita problemi di overflow
            if (angle<0) angle=2*Math.PI;
          }
```

Uso del Behavior dx-sn (2)

```
        else if (code==KeyEvent.VK_RIGHT) //Codice freccia a destra
        {
          //Incrementa l'angolo
          angle+=0.1;
          //Evita problemi di overflow
          if (angle>2*Math.PI) angle=0;
        }
        //imposta la rotazione dell'angolo
        rotation.rotY(angle);
        targetTG.setTransform(rotation);
      }

  //Resetta il Behavior per continuare a rispondere ad eventi di tastiera
  this.wakeupOn(awt);
}
```

Esercizi

Utilizzando la classe *WakeupOnElapsedTime* fare ruotare il cubo automaticamente ad ogni secondo

Aggiungere la rotazione alla “Terra” costruita nella lezione precedente.

(Per casa) Provare ad utilizzare altre condizioni di wakeup e verificare come si comportano

http://download.java.net/media/java3d/javadoc/1.5.0/javax/media/j3d/WakeupCriterion.html

Usare la classe Behavior (1)

Il primo passo per aggiungere un Behavior in un'applicazione è verificare che lo scene graph contenga gli oggetti necessari al Behavior
- per esempio per usare SimpleBehavior_1 era necessario un *TransformGroup*
- Molti Behavior richiedono semplicemente un *TransformGroup*

Tuttavia le richieste di un *Behavior* dipendono dal *Behavior* stesso e dall'applicazione e possono essere anche molto complesse

Usare la classe Behavior (2)

Stabilito cosa serve al *Behavior*, bisogna decidere in quale punto dello scene graph inserirlo

L'ultimo passo è fornire al *Behavior* un *Bound*: il behavior è attivo solo quando il suo *Bound* interseca il volume attivo della *ViewPlatform*

Solo i *Behavior* attivi possono ricevere stimoli: in tal modo è possibile incrementare le prestazioni scegliendo *Bound* piccoli

Utility per la navigazione (1)

Negli esempi finora esposti l'osservatore non aveva la possibilità di “navigare” nel mondo virtuale

La possibilità di muoversi è una caratteristica fondamentale di quasi tutte le applicazioni 3D

Java3D permette la navigazione tramite delle classi di utility che implementano questa funzionalità

Utility per la navigazione (2)

Il view branch graph di un universo virtuale contiene una "view platform transform"

Se tale trasformazione cambia, l'effetto è quello di muovere e/o orientare l'osservatore

Da questo si intuisce che aggiungere la possibilità di navigare è semplice: basta aggiungere un *Behavior* che cambia la view platform transform in risposta agli eventi

```
TransformGroup vtg =
simpleU.getViewingPlatform().getViewPlatformTransform();
   Transform3D viewTransform = new Transform3D();
    //Sposto leggermente piu' in alto la posizione della
view
    Transform3D t3d=new Transform3D();
    t3d.setTranslation(new Vector3f(0.0f,0.3f,0.0f));
    vtg.setTransform(t3d);

 //Creo un behavior per la navigazione da tastiera
    KeyNavigatorBehavior keyNavBeh=new
KeyNavigatorBehavior(vtg);
    //Imposto il bound del behavior
    keyNavBeh.setSchedulingBounds(new BoundingSphere(new
Point3d(),10000.0));
    //Aggiungo il behavior alla scena
    scene.addChild(keyNavBeh);
    scene.compile();
simpleU.addBranchGraph(scene);
```

Movimenti di KeyNavigatorBehavior

Tasto	Movimento	Tasto+alt
←	Ruota a sinistra	Trasla a sinistra
→	Ruota a destra	Trasla a destra
↑	Muovi in avanti	
↓	Muovi indietro	
PgUp	Ruota verso l'alto	Trasla in alto
PgDown	Ruota verso il basso	Trasla in basso
+	Ripristina la back clip distance (e riposizionati nell'origine)	
-	Riduci la back clip distance	
=	Ritorna al centro dell'universo	

Utility per il Mouse (1)

Il package *com.sun.j3d.utils.behaviors.mouse* contiene i behavior per l'interazione con il mouse

L'interazione tramite mouse prevede:
- traslazione (movimento su un piano parallelo all'image plate)
- zooming (movimento in avanti ed indietro)
- rotazione

Utility per il Mouse (2)

Classe	Risposta all'azione del mouse	Azione del mouse
MouseRotate	Ruota l'oggetto	Movimento del mouse tenendo premuto il tasto sinistro
MouseTranslate	Trasla l'oggetto lungo un piano parallelo all'image plate	Movimento del mouse tenendo premuto il tasto destro
MouseZoom	Trasla l'oggetto lungo un piano ortogonale all'image plate	Movimento del mouse tenendo premuto il tasto centrale

```
//Crea la scena
private BranchGroup createSceneGraph()
{
  //Crea la radice del branch graph
  BranchGroup objRoot=new BranchGroup();

  //Crea un gruppo per le trasformazioni affini
  TransformGroup objRotate=new TransformGroup();
  objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

  //Aggiunge al gruppo un cubo colorato
  objRotate.addChild(new ColorCube(0.4));

  //Crea il behavior per ruotare il cubo
  MouseRotate myMouseRotate=new MouseRotate(objRotate);
  //Imposta un raggio d'azione del behavior
  myMouseRotate.setSchedulingBounds(new BoundingSphere());

  //assembla la scena
  objRoot.addChild(myMouseRotate);
  objRoot.addChild(objRotate);

  return objRoot;
}
```

Esercizio

Utilizzare la classe utility per il mouse sulla “terra”

Animazione (1)

Certi oggetti nel mondo reale cambiamo indipendentemente dalle azioni esterne

Lo stesso deve avvenire in un mondo Java3D

Per esempio, un orologio deve continuare a camminare anche senza alcuna interazione con l'utente: l'orologio è un tipico esempio di animazione

Animazione (2)

Come l'interazione anche l'animazione in Java3D
è implementata tramite i *Behavior*

Teoricamente è possibile creare qualsiasi animazione personalizzata usando i *Behavior*, in ogni caso Java3D fornisce una serie di classi di utility per le animazioni più comuni

Ovviamente queste classi sono basate sui *Behavior*

Animazione (3)

Un particolare insieme di classi per le animazioni sono gli interpolatori: un oggetto *Interpolator* manipola alcuni parametri di uno scene graph per creare animazioni basate sul tempo

Un altro insieme di animazioni servono ad “animare” gli oggetti visuali in risposta ai cambiamenti del punto di osservazione

Questo insieme include le classi *Billboard*, *OrientedShape3D* e *LOD* (Level of Detail) che non vedremo nel dettaglio.

Gli Interpolatori e la classe Alpha

Gli *Interpolator* sono *Behavior* personalizzati che usano un oggetto *Alpha*

Le azioni degli *Interpolator* comprendono il cambiamento di posizione, orientamento, dimensione, colore e trasparenza di un oggetto

Esistono *Interpolator* anche per altre azioni, incluse le combinazioni delle precedenti azioni

La classe Alpha (1)

Un oggetto *Alpha* produce un valore, detto valore Alpha nell'intervallo [0,1]
- il valore alpha cambia nel tempo in base ai parametri dell'oggetto stesso

Fissati i parametri di *Alpha* ed un particolare istante di tempo, c'è un solo valore alpha che l'oggetto *Alpha* può produrre
- disegnando il valore di alpha nel tempo si può vedere il "fronte d'onda" che l'oggetto *Alpha* produce

La classe Alpha (2)

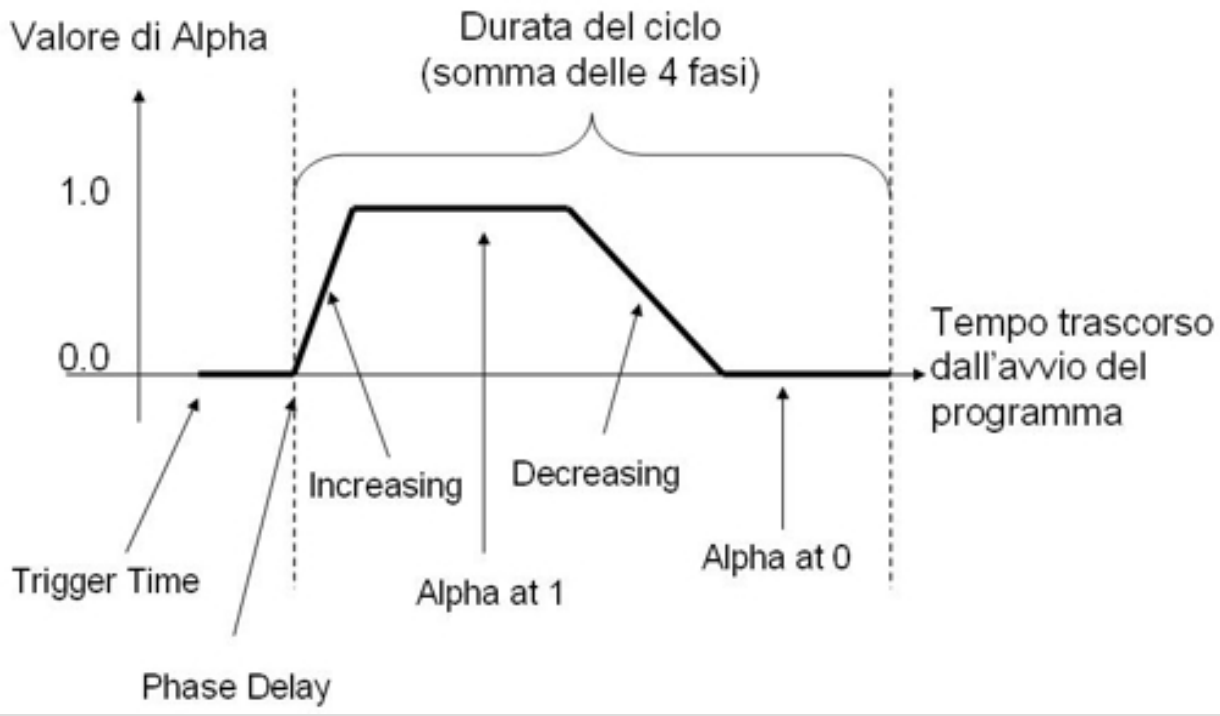
Il fronte d'onda ha quattro fasi: incremento di alpha, alpha ad uno, decremento di alpha, alpha a zero

La somma delle quattro fasi forma un **ciclo del fronte d'onda**

Queste quattro fasi corrispondono con i quattro parametri dell'oggetto Alpha

La durata delle quattro fasi è specificata da un valore intero indicante la durata in **millisecondi**

Ciclo del fronte d'onda



La classe Alpha (3)

Tutte le temporizzazioni sono relative allo “start time” di Alpha

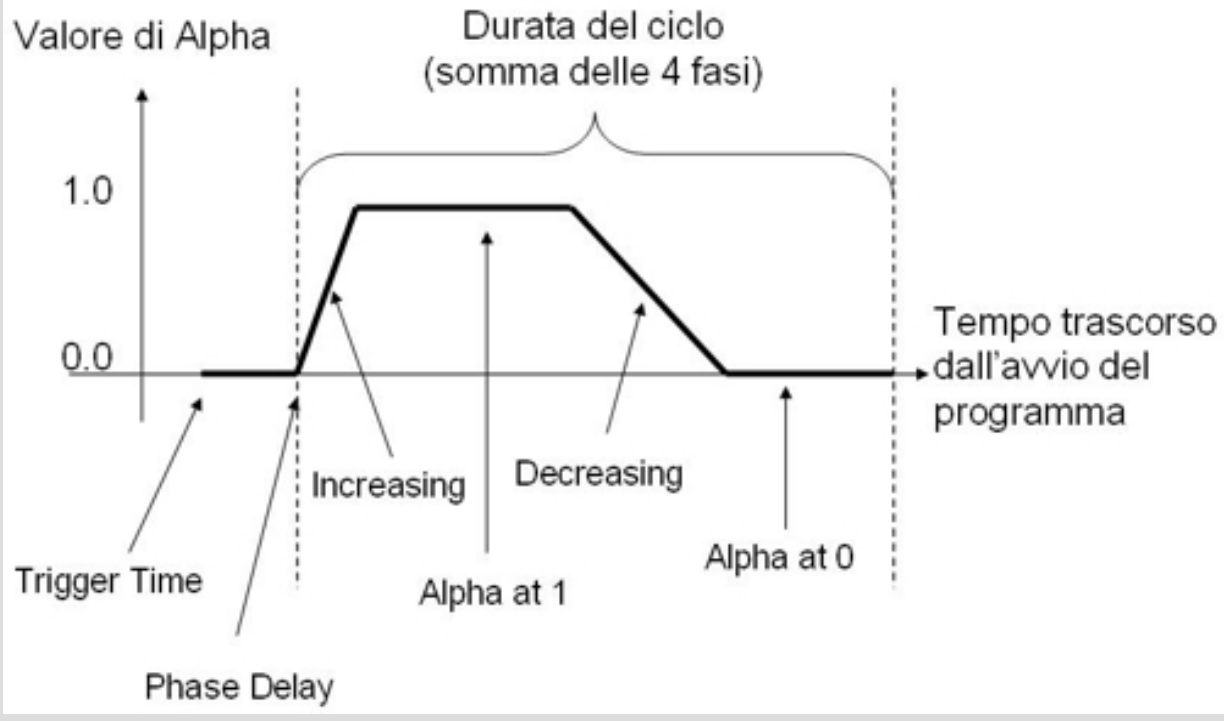
Lo start time di tutti gli oggetti Alpha è preso dallo start time di sistema: **tutti gli oggetti Alpha creati in differenti momenti avranno lo stesso start time**

Risultato: tutti gli interpolatori sono sincronizzati (anche quelli basati su Alpha diversi)

La classe Alpha (4)

Gli oggetti *Alpha* possono comunque fare partire i fronti d'onda in momenti differenti

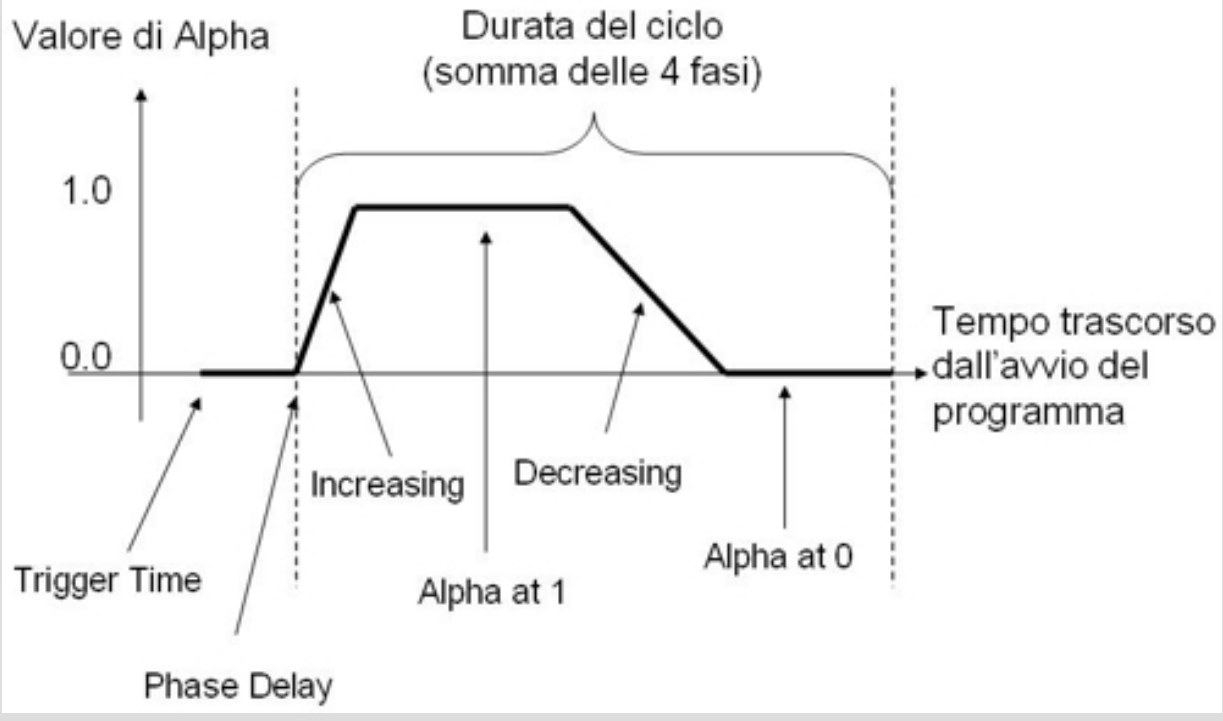
I parametri *TriggerTime* e *PhaseDelayDuration* permettono di ritardare l'inizio del primo ciclo



La classe Alpha (5)

Il *TriggerTime* specifica il tempo (assoluto e successivo allo *StartTime*) in cui iniziare le operazioni di Alpha

Il *PhaseDelayDuration* specifica il ritardo che deve intercorrere tra il *TriggerTime* ed il primo ciclo del fronte d'onda



La classe Alpha (6)

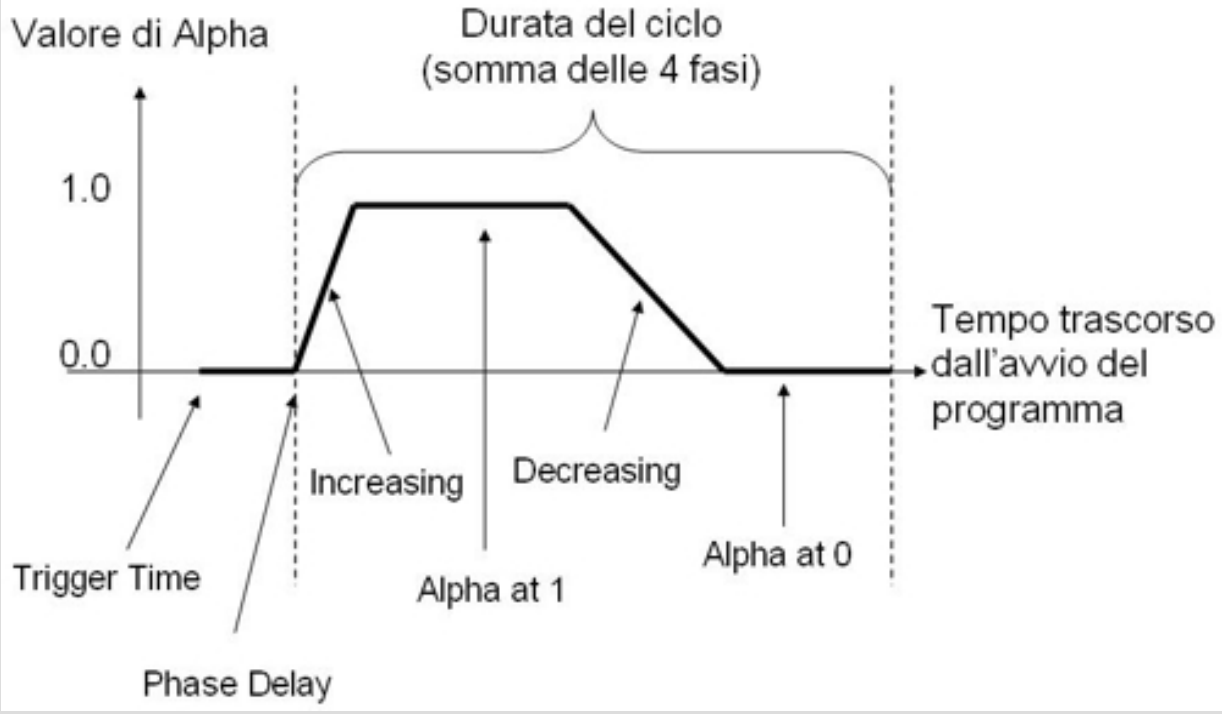
Un fronte d'onda può “ciclare” una, più o infinite volte
- il numero di cicli è specificato dal parametro *loopCount*

Quando *loopCount* è positivo questo indica il numero di cicli, un valore di **-1 indica un ciclo infinito**

Se il loopCount è maggiore di uno o è infinito allora il *PhaseDelayDuration* sarà usato solo nel primo ciclo

La classe Alpha (7)

- Alpha(**int** loopCount, **long** increasingAlphaDuration)
- Alpha(**int** loopCount, **int** mode, **long** triggerTime, **long** phaseDelayDuration, **long** increasingAlphaDuration, **long** increasingAlphaRampDuration, **long** alphaAtOneDuration, **long** decreasingAlphaDuration, **long** decreasingAlphaRampDuration, **long** alphaAtZeroDuration)



La classe Alpha (8)

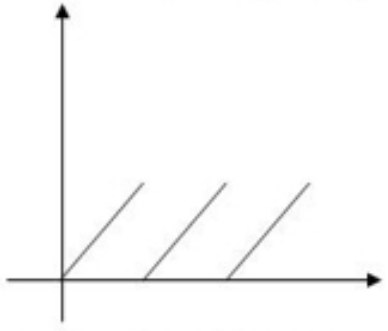
Un fronte d'onda non deve per forza usare tutte le fasi: può essere formato da uno, due, tre o quattro fasi

L'oggetto *Alpha* ha due modalità che specificano il sottoinsieme delle fasi da usare:
- INCREASING_ENABLE indica le prime due fasi
- DECREASING_ENABLE indica le seconde due fasi

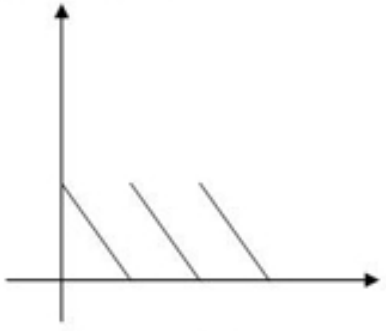
Una terza modalità (la combinazione delle due) indica l'uso di tutte le fasi

La classe Alpha (9)

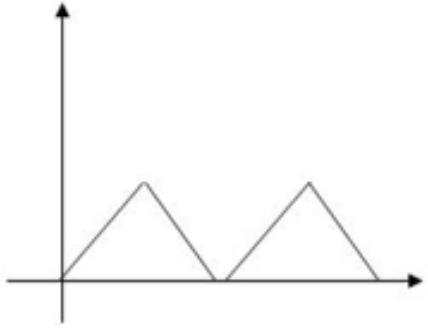
Esempi di possibili forme d'onda



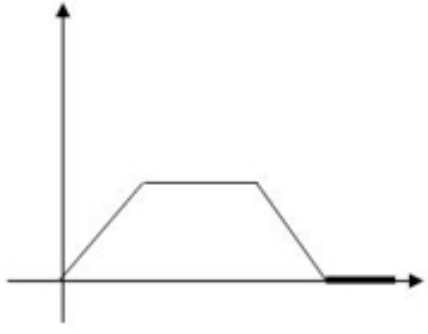
Increasing Enable con durata >0, tutti gli altri a 0



Decreasing Enable con durata >0, tutti gli altri a 0



Increasing e Decreasing con durata > 0, Alpha at 1 e Alpha at 0 con durata 0



Increasing e Decreasing a ENABLE, tutte e quattro le fasi con durate > 0

Interpolator (1)

L'uso degli interpolatori e di *Alpha* è molto simile all'uso di qualsiasi *Behavior*:

- creare l'oggetto target con le appropriate capability impostate
- creare un oggetto *Alpha*
- creare l'oggetto *Interpolator* referenziandogli *l'Alpha* ed il target
- aggiungere il *Bound* dell'*Interpolator*
- aggiungere l'*Interpolator* allo scene graph

Esempio di RotationInterpolator

```
//Crea un timer
   Alpha rotationAlpha=new Alpha(-1,8000);

   //Crea un interpolatore per le rotazioni collegato con il gruppo di trasformazione
   RotationInterpolator rotator=new RotationInterpolator(rotationAlpha,objSpin);

  BoundingSphere bounds=new BoundingSphere();
  //Imposta un raggio d'azione all'interpolatore
   rotator.setSchedulingBounds(bounds);

   //aggiunge l'interpolatore alla gruppo di trasformazione
   objSpin.addChild(rotator);
```

Interpolator (2)

Ogni interpolatore è un *Behavior* personalizzato che risponde agli stimoli di frame

Ad ogni chiamata del metodo *processStimulus()*, un interpolatore:
- controlla l'oggetto Alpha ad esso associato per recuperare il valore alpha
- modifica il target in base al valore alpha
- reimposta il suo trigger per attendere il prossimo frame (a meno che *Alpha* non sia terminato)

Interpolator (3)

Molti interpolatori memorizzano inoltre due valori usati come estremi per le azioni di interpolazione

Per esempio, il *RotationInterpolator* memorizza i due angoli entro cui deve avvenire la rotazione

Per ogni frame, l'interpolatore controlla il valore alpha ed esegue la rotazione appropriata sul *TransformGroup* associato

Interpolator (4)

La precedente descrizione non vale per gli *SwitchValueInterpolator* ed i *PathInterpolator*

Uno *SwitchValueInterpolator* sceglie uno tra i figli di un gruppo *Switch* in base al valore alpha: nessun tipo di interpolazione è realmente eseguita

Il comportamento dei *PathInterpolator* è un po' più complesso e verrà preso in esame successivamente

Interpolator (4)

Classe Interpolator	Usato per	Oggetto target
ColorInterpolator	Cambiare il colore di un oggetto	Material
PathInterpolator (astratta)	Creare percorsi	TransformGroup
PositionInterpolator	Cambiare la posizione di un oggetto	TransformGroup
RotationInterpolator	Cambiare l'orientamento di un oggetto	TransformGroup
ScaleInterpolator	Cambiare le dimensioni di un oggetto	TransformGroup
SwitchValueInterpolator	Scegliere un oggetto tra una collezione	Switch
TransparencyInterpolator	Cambiare la trasparenza di un oggetto	TransparencyAttributes

ColorInterpolator

Un *ColorInterpolator* ha un *Material* come target

Questo interpolatore cambia il colore gestito dal *Material*

Questo rende il *ColorInterpolator* molto potente: poiché più oggetti possono condividere lo stesso *Material*, un solo *ColorInterpolator* può modificare il colore di tutti gli oggetti contemporaneamente

ESERCIZIO

Sacricare dal sito e-learning il file InterpolatorApp.java e testare le classi ColorInterpolator

```
//Crea un interpolatore per le colorazioni collegato con il gruppo di trasformazione
    ColorInterpolator coloring=new ColorInterpolator(alpha,objColor);

    //Imposta i colori iniziale e finale dell'interpolatore
    coloring.setStartColor(new Color3f(1,0,0));
    coloring.setEndColor(new Color3f(0,0,1));

    //Imposta un raggio d'azione all'interpolatore
    coloring.setSchedulingBounds(bounds);
```

PositionInterpolator
RotationInterpolator

Un *PositionInterpolator* varia la posizione di un oggetto lungo un asse
- è possibile specificare le posizioni iniziale e finale della traslazione e l'asse di traslazione (default: 0, 1, asse x)

Un *RotationInterpolator* varia l'orientamento di un oggetto lungo un asse
- è possibile specificare gli angoli iniziale e finale della rotazione e l'asse di rotazione (default: 0, 2π, asse y)

ESERCIZIO

Testare le classi Position

```
//Crea un interpolatore per le posizioni collegato con il gruppo di trasformazione
    PositionInterpolator position=new PositionInterpolator(alpha,objMove);

    //Imposta un raggio d'azione all'interpolatore
    position.setSchedulingBounds(bounds);

    //Imposta la posizione iniziale
    position.setStartPosition(-1.0f);
```

e Rotation

```
//Crea un interpolatore per le rotazioni collegato con il gruppo di trasformazione
    RotationInterpolator rotation=new RotationInterpolator(alpha,objRotate);

    //Imposta un raggio d'azione all'interpolatore
    rotation.setSchedulingBounds(bounds);
```

ScaleInterpolator
SwitchInterpolator

Uno *ScaleInterpolator* varia le dimensioni di un oggetto
- è possibile specificare i fattori di scala iniziale e finale (default: 0, 1)

Uno *SwitchValueInterpolator* non genera un'interpolazione di punti

Esso seleziona uno dei figli di un gruppo *Switch*
- i valori di soglia per lo switching sono ottenuti dividendo l'intervallo [0.0, 1.0] per il numero di figli del gruppo *Switch*

ESERCIZIO

Testare ScaleInterpolator

```
//Crea un interpolatore per i ridimensionamenti collegato con il gruppo di trasformazione
ScaleInterpolator scaling=new ScaleInterpolator(alpha,objScale);

//Imposta un raggio d'azione all'interpolatore
scaling.setSchedulingBounds(bounds);
```

e Switch (già pronto...)

TransparecncyInterpolator

Un *TransparencyInterpolator* ha come target un *TransparencyAttributes*

Questo interpolatore cambia il valore di trasparenza del suo target

Poiché più oggetti possono condividere lo stesso *TransparencyAttributes*, un singolo *TransparencyInterpolator* può modificare la trasparenza di più oggetti contemporaneamente

PathInterpolator (1)

I *PathInterpolator* memorizzano due o più valori per l'interpolazione

Il core Java3D fornisce *PathInterpolator* per:
- posizione
- rotazione
- posizione e rotazione
- posizione, rotazione e scaling

Target del *PathInterpolator* è un *TransformGroup*

PathInterpolator (2)

I *PathInterpolator* memorizzano un insieme di valori, o *knot* (nodo), usati due alla volta per l'interpolazione

- il valore alpha determina quale coppia di nodi bisogna usare

- i nodi assumono valori nel range [0.0, 1.0], esattamente come i valori di alpha

Il primo e l'ultimo nodo devono avere valori 0.0 e 1.0, i nodi rimanenti devono essere memorizzati in ordine crescente nel *PathInterpolator*

PathInterpolator (3)

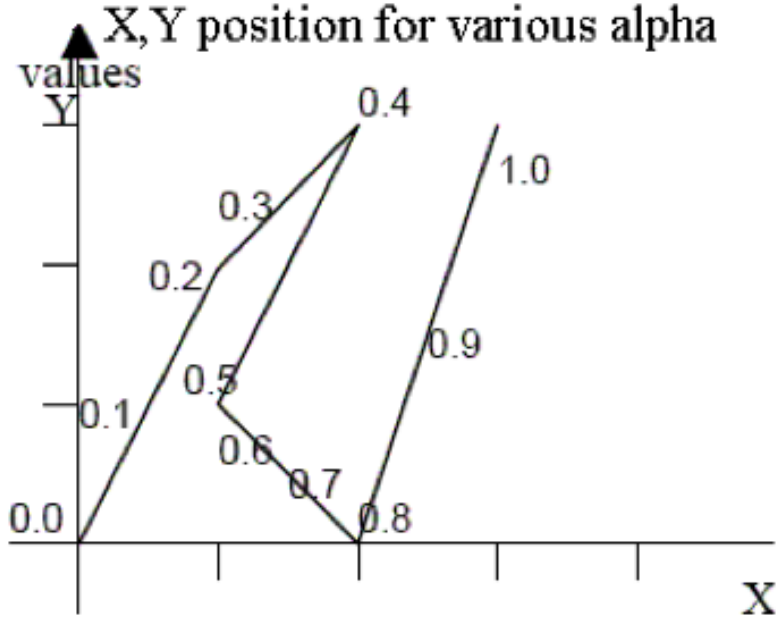
Ad ogni nodo corrisponde un valore (posizione, rotazione e/o scala) usato per l'interpolazione

Considerato un valore alpha per generare l'interpolazione vengono considerati il nodo con il valore più grande minore o uguale ad alpha ed il suo successivo

I nodi sono specificati in ordine in modo da usare coppie di nodi adiacenti al variare di alpha

PathInterpolator (4)

knot	knot value	position(x, y, z)
0	0.0	(0.0, 0.0, 0.0)
1	0.2	(1.0, 2.0, 0.0)
2	0.4	(2.0, 3.0, 0.0)
3	0.5	(1.0, 1.0, 0.0)
4	0.8	(2.0, 0.0, 0.0)
5	1.0	(3.0, 3.0, 0.0)



alpha value (a)	knots used
0.0	0
0.0 < a < 0.2	0, 1
0.2	1
0.2 < a < 0.4	1, 2
0.4	2
0.4 < a < 0.5	2, 3

alpha value (a)	knots used
0.5	3
0.5 < a < 0.8	3, 4
0.8	4
0.8 < a < 1.0	4, 5
1.0	5

Esercizio

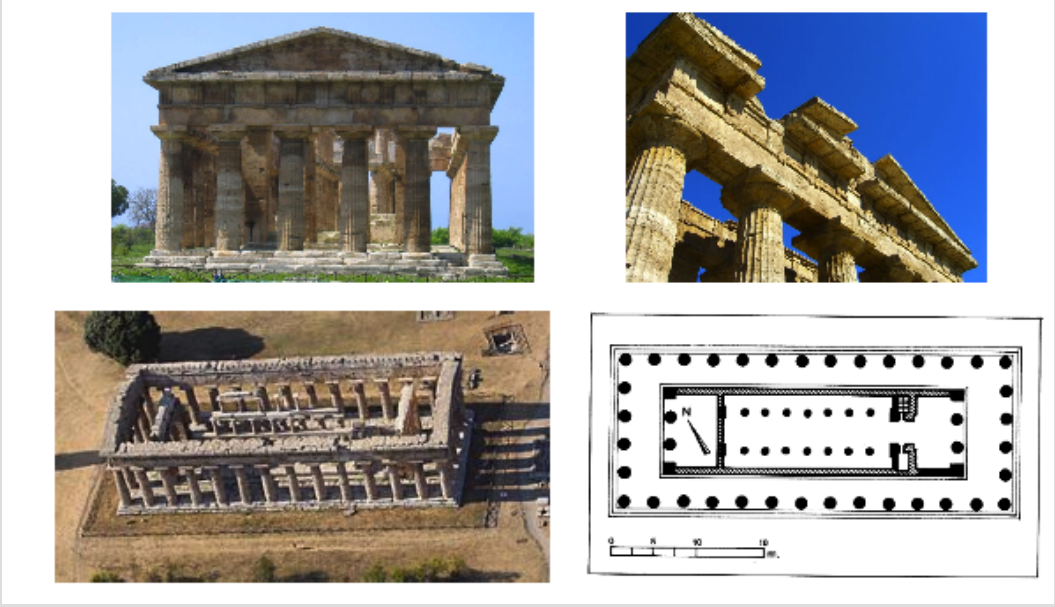
Riprendendo l'esercizio riproduzione della Terra:

- Aggiungere la Luna (texture fornita sul sito di e-learning).

- Trascurando la rotazione della Terra attorno al Sole, far orbitare la Luna intorno alla Terra.

Si può assumere che l'eclittica sia sullo stesso piano dell'equatore e non è necessario rispettare la scala nella distanza orbitale lunare.

Progetto Finale



Riprendendo l'esercizio della lezione scorsa (riproduzione della facciata del tempio di Poseidone di Pæstum), procedere nella ricostruzione dell'intero edificio con la possibilità di navigare nel mondo virtuale