

# **Java 3D**

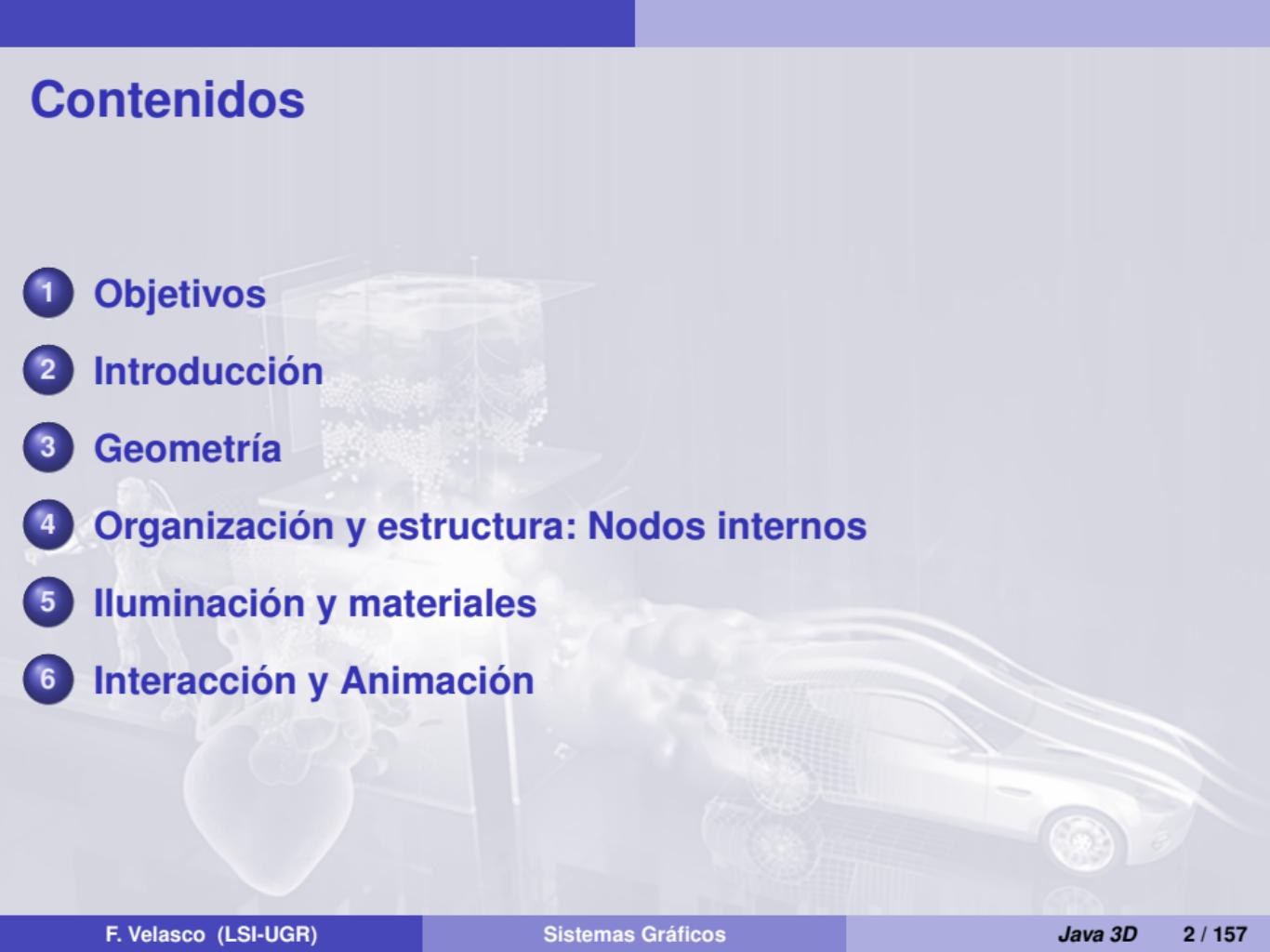
Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Sistemas Gráficos

Grado en Ingeniería Informática  
Curso 2016-2017

# Contenidos

- 
- 1 Objetivos
  - 2 Introducción
  - 3 Geometría
  - 4 Organización y estructura: Nodos internos
  - 5 Iluminación y materiales
  - 6 Interacción y Animación

# Objetivos

- Saber implementar un grafo de escena con Java3D
  - ▶ Crear e importar geometría
  - ▶ Organizar la geometría mediante transformaciones
  - ▶ Añadir luces a la escena
  - ▶ Añadir materiales a la geometría
  - ▶ Capturar y procesar la interacción del usuario, mediante teclado y ratón
  - ▶ Añadir animación

# Introducción

- Descripción muy breve de la biblioteca
- **Completar la formación con:**
  - ▶ La documentación:
    - ★ <https://java3d.java.net>
    - ★ <https://docs.oracle.com/javase/7/docs/api/>
  - ▶ La colección Getting Started with the Java 3D API  
(disponible en Prado)
  - ▶ La bibliografía propuesta
  - ▶ Programando y probando los conceptos

# Contenidos

1 Objetivos

2 Introducción

3 Geometría

- Primitivas básicas
- Creación de geometría
- Importación de geometría

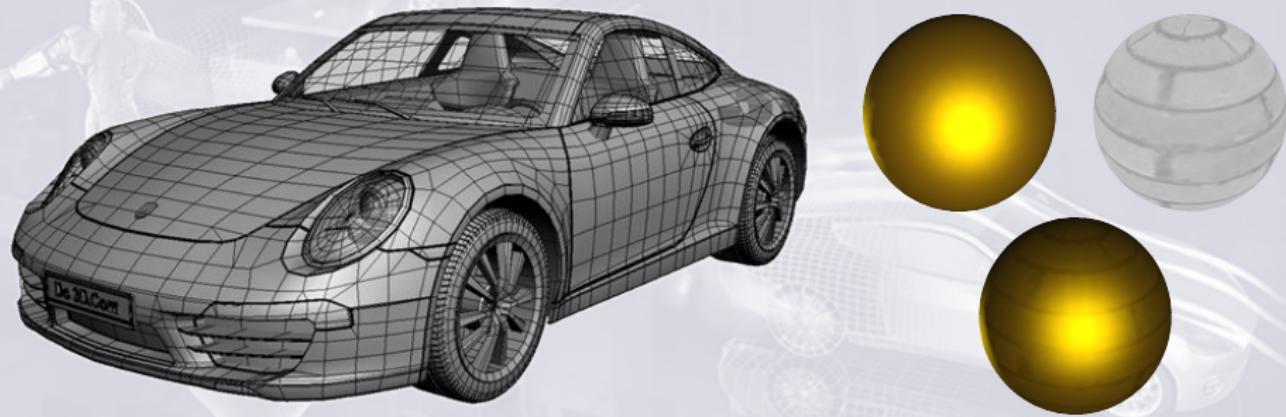
4 Organización y estructura: Nodos internos

5 Iluminación y materiales

6 Interacción y Animación

# Figuras geométricas 3D

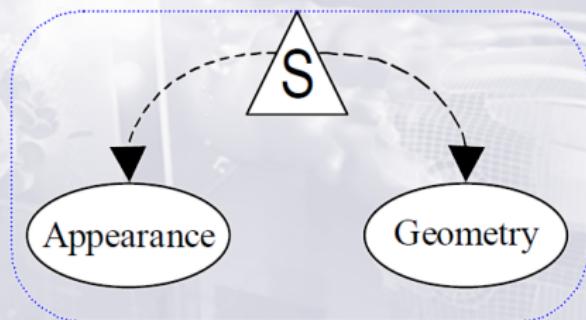
- En Java 3D se representan mediante B-Rep
- Para cada vértice se especifica
  - ▶ Coordenada cartesiana (obligatorio)
  - ▶ Vector normal (obligatorio para realizar cálculo de iluminación)
  - ▶ Coordenada de textura (obligatorio para visualizar con texturas)



# Clase para representar geometría

## • Shape3D

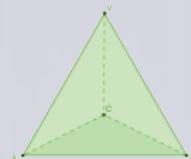
- ▶ Subclase de Leaf
- ▶ Representa una figura geométrica 3D mediante 2 NodeComponent
  - ▶ **Geometry**
    - ★ La información sobre la geometría de la figura
      - Vértices, Caras
  - ▶ **Appearance**
    - ★ Un aspecto, único para todo el objeto
      - Material, Textura



# Modos de creación de geometría

- Mediante primitivas básicas

- ▶ Caja, Cono,  
Cilindro, Esfera



- Definiendo toda la información *manualmente*

- ▶ Geometría: Vértices, sus coordenadas
  - ▶ Topología: Conexiones entre vértices para formar polígonos.  
Triángulos o cuadrángulos planos convexos
  - ▶ Atributos de los vértices: vector normal, coordenadas de textura

- Usando un cargador de geometría

- ▶ La figura se ha creado con otro software (Blender, 3D Studio, etc.)
  - ▶ Se importa en Java 3D

# Primitivas básicas

- Clase **Primitive**
- No forma parte del paquete Java 3D sino del de utilidades
- Subclase de Group, **no es** un Shape3D
  - ▶ No es una geometría, es una colección de geometrías
    - ★ Por ejemplo, Box es una colección de 6 caras
  - ▶ No se le pueden aplicar los métodos de Shape3D, sí los de Group además de los suyos propios.
- Las figuras se crean con el origen en su centroide
- Su diseño hace que no sea fácilmente derivable
- No se recomienda para aplicaciones de cierta complejidad

# Box

Subclase de Primitive > Group

- Colección de 6 rectángulos (Shape3D) forman una caja

- Construcción:

```
Box (float xDim, float yDim, float zDim,  
     int primFlags, Appearance aspecto)
```

- Se pueden obtener con

```
Shape3D getShape (int cara)
```

- ▶ Estas constantes indexan sus caras:

```
Box.FRONT, Box.BACK, Box.RIGHT,  
Box.LEFT, Box.TOP, Box.BOTTOM
```

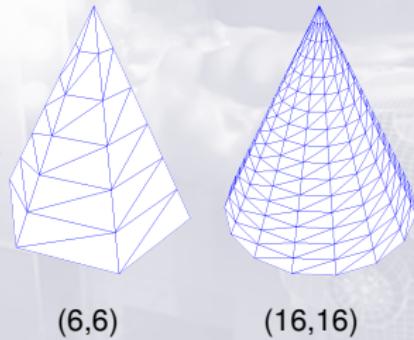
# Cone

Subclase de Primitive > Group

- Colección de una base (Cone.CAP) y el resto (Cone.BODY)
- Construcción:

```
Cone (float radio, float altura,  
       int primFlags, int resX, int resY,  
       Appearance aspecto)
```

- ▶ resX: Lados del polígono de la base (15 por defecto)
- ▶ resY: Divisiones a lo largo de la altura (1 por defecto)



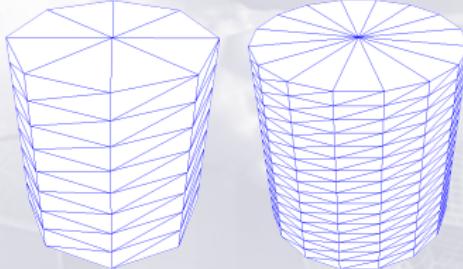
# Cylinder

Subclase de Primitive > Group

- Colección de una base (Cylinder.BOTTOM), una tapa (Cylinder.TOP) y el resto (Cylinder.BODY)
- Construcción:

```
Cylinder (float radio, float altura,  
          int primFlags, int resX, int resY,  
          Appearance aspecto)
```

- ▶ resX: Lados del polígono de la base (15 por defecto)
- ▶ resY: Divisiones a lo largo de la altura (1 por defecto)



(8,8)

(16,16)

# Sphere

Subclase de Primitive > Group

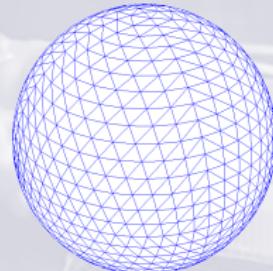
- Colección de solo un elemento (`Sphere.BODY`)
- Construcción:

```
Sphere (float radio,  
        int primFlags, int nivelDivision  
        Appearance aspecto)
```

- ▶ `nivelDivision`: Medida de la resolución (15 por defecto)



(16)



(64)

# Flags de las primitivas

- **ENABLE\_APPEARANCE\_MODIFY**

Permite modificar su aspecto en tiempo de ejecución

- **GENERATE\_NORMALS**

Genera normales para los vértices

- ▶ Por defecto si se usa un constructor sin este parámetro

- **GENERATE\_NORMALS\_INWARD**

Genera las normales hacia el interior

- ▶ Por ejemplo, para usar una esfera como entorno

- **GENERATE\_TEXTURE\_COORDS**

Genera coordenadas de textura

# Métodos habituales

- `Shape3D getShape (int parte)`
  - ▶ Se obtiene como Shape3D la parte indicada
- `void setAppearance (int parte, Appearance ap)`
  - ▶ Se asigna un nuevo aspecto a una parte concreta
- `void setAppearance (Appearance ap)`
  - ▶ Se asigna un nuevo aspecto a todas las partes

# Creación de geometría

## Objetos Shape3D

- A partir de un objeto **Geometry**
  - ▶ Contiene como mínimo:
    - ★ Vértices y caras
  - ▶ Además, para cada vértice, se puede añadir:
    - ★ Un vector normal, una coordenada de textura
- Y un objeto **Appearance**
  - ▶ Define atributos para calcular el color al visualizar la escena
- Se construye un objeto **Shape3D**
  - ▶ Constructor
    - ★ `figura = new Shape3D ( objetoGeometry, objetoAppearance );`
  - ▶ Métodos set
    - ★ `figura = new Shape3D ();`
    - `figura.setGeometry ( objetoGeometry );`
    - `figura.setAppearance ( objetoAppearance );`

# Shape3D

- Capacidades para modificar el aspecto de un Shape3D vivo
  - ▶ Shape3D.ALLOW\_APPEARANCE\_READ
  - ▶ Shape3D.ALLOW\_APPEARANCE\_WRITE
- ¿Cómo crear una figura?

**Ejemplo:** Clase que deriva de Shape3D

```
public class MyFigure extends Shape3D {  
  
    public MyFigure (Appearance anAppearance) {  
        super();  
        this.setGeometry (createGeometry);  
        this.setAppearance (anAppearance);  
    }  
  
    private Geometry createGeometry () {  
        // Código que crea y devuelve la geometria  
    }  
}
```

# Clase Geometry

- Clase abstracta
- Describe tanto la geometría como la topología de las figuras 3D
- Componente que forma parte de un Shade3D
- Sus subclases principales se pueden agrupar en
  - ▶ Geometría basada en vértices **no** indexados
    - ★ Cada vértice se usa una sola vez  
En un cubo, habría que definir 24 vértices = 6 caras x 4 vértices/cara
    - ★ Subclases derivadas de  
`GeometryArray > Geometry`
  - ▶ Geometría basada en vértices indexados
    - ★ Los vértices pueden reusarse  
En un cubo, se definirían solo 8 vértices. Cada uno se usaría 3 veces
    - ★ Subclases derivadas de  
`IndexedGeometryArray > GeometryArray > Geometry`

# Geometría no indexada vs. indexada

- Geometría no indexada

: Información en una geometría no indexada

```
// Cada vértice aparece varias veces, una vez por cara
// Se dan los vértices en orden antihorario para formar las caras
```

```
Vértices_y_Caras = {C, B, A,     C, A, V,     C, V, B,     A, B, V}
```

- Geometría indexada

: Información en una geometría indexada

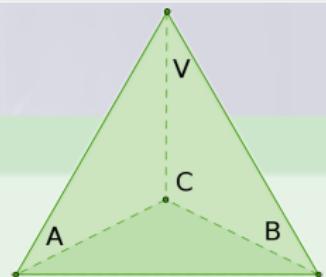
```
// Cada vértice se define una sola vez
```

```
Vértices = {C, A, B, V}
```

```
// Las caras se definen con índices a los vértices
```

```
// Los índices se dan en sentido antihorario
```

```
Caras = {0, 2, 1,     0, 1, 3,     0, 3, 2,     1, 2, 3}
```



# Geometry

## Procedimiento

- 1 Definir los datos a usar
- 2 Construir un objeto vacío
- 3 Llenarlo con los datos
- 4 Asociarlo a objetos Shape3D

## Ejemplo: Proceso de creación de geometría

```
// Definición de datos
float[] vertices = { 0.0f,0.0f,0.0f,
                      1.0f,0.0f,0.0f,  0.0f,1.0f,0.0f };

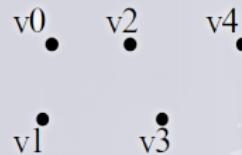
// Construcción del objeto vacío
GeometryArray geometriaTriangulo =
    new TriangleArray (vertices.length/3,
                      GeometryArray.COORDINATES);

// Llenado de datos
geometriaTriangulo.setCoordinates (0, vertices);

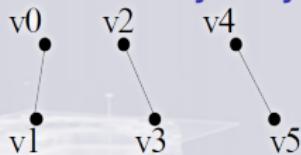
// Creación del Shape3D
Shape3D triangulo = new Shape3D (geometriaTriangulo);
```

# Geometría no encadenada

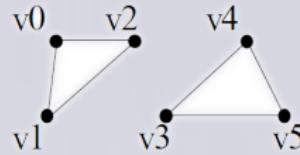
Todas las clases derivan de **GeometryArray**



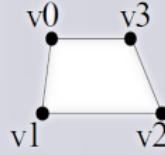
PointArray



LineArray



TriangleArray

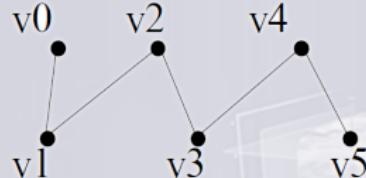


QuadArray

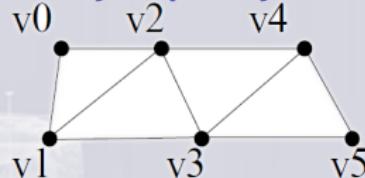
- Con el aspecto se define
  - ▶ Color, trazo de las líneas, relleno de los polígonos
- Los polígonos de 4 lados deben ser
  - ▶ Coplanares
  - ▶ Que sus aristas no se crucen
  - ▶ Formar un polígono convexo
- Constructores
  - ▶ TriangleArray (int numeroDeVertices, int formato)

# Geometría encadenada

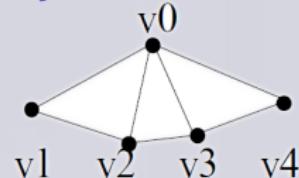
Todas las clases derivan de **GeometryStripArray > GeometryArray**



LineStripArray



TriangleStripArray



TriangleFanArray

- Constructores

- ▶ **TriangleStripArray (int numVertices, int formato, int[] longCadenas)**

## Ejemplo: LineStripArray



```

// Vectores de coordenadas y de cadenas
float[] coordenadas = { 0,0,0, 2,0,0, 2,1,0, 1,2,0, 0,1,0 };
int[] cadenas = { 2, 3 };

// Primitiva: 2 cadenas de 2 y 3 vértices respectivamente
GeometryArray lineas = new LineStripArray (
    coordenadas.length / 3, GeometryArray.COORDINATES, cadenas );
lineas.setCoordinates (0, coordenadas);
  
```

# Formato de los vértices

- Define el tipo de información que se almacena de cada vértice
  - ▶ `GeometryArray.COORDINATES`: Coordenadas, obligatorio
  - ▶ `GeometryArray.NORMALS`: Vector normal
  - ▶ `GeometryArray.TEXTURE_COORDINATE_2`: Coord. de textura 2D
- Se usan varios componiéndolos con el operador |
- Java 3D crea los arrays para cada componente
- Se rellenan los datos con los métodos set\*

# Métodos para *rellenar* datos

(los tipos pueden ser double, Point3d, y Vector3d)

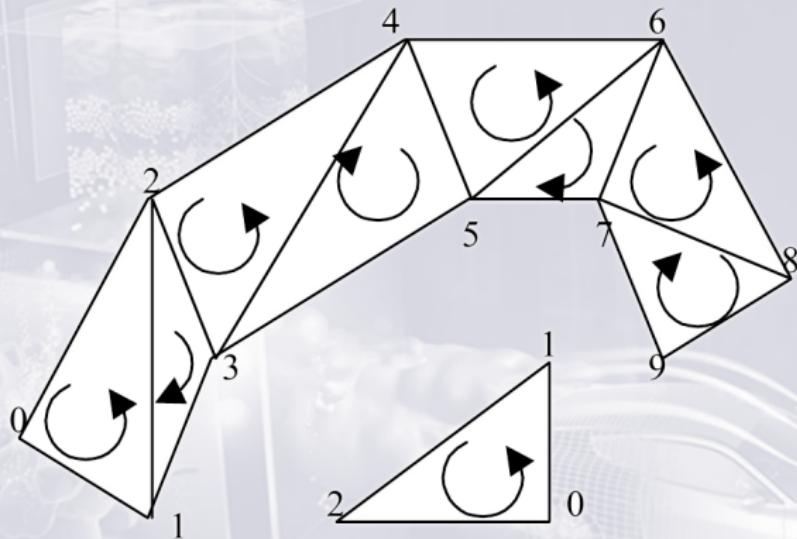
- Coordenadas de los vértices
  - ▶ void setCoordinates (int i, float[] coordenadas)
  - ▶ void setCoordinates (int i, Point3f[] coordenadas)

- Vectores normales de los vértices
  - ▶ void setNormals (int i, float[] normales)
  - ▶ void setNormals (int i, Vector3f[] normales)

- Coordenadas de textura de los vértices
  - ▶ void setTextureCoordinates (int conjunto, int i, float[] coordenadasTextura)
  - ▶ void setTextureCoordinates (int conjunto, int i, TextCoord2f[] coordenadasTextura)

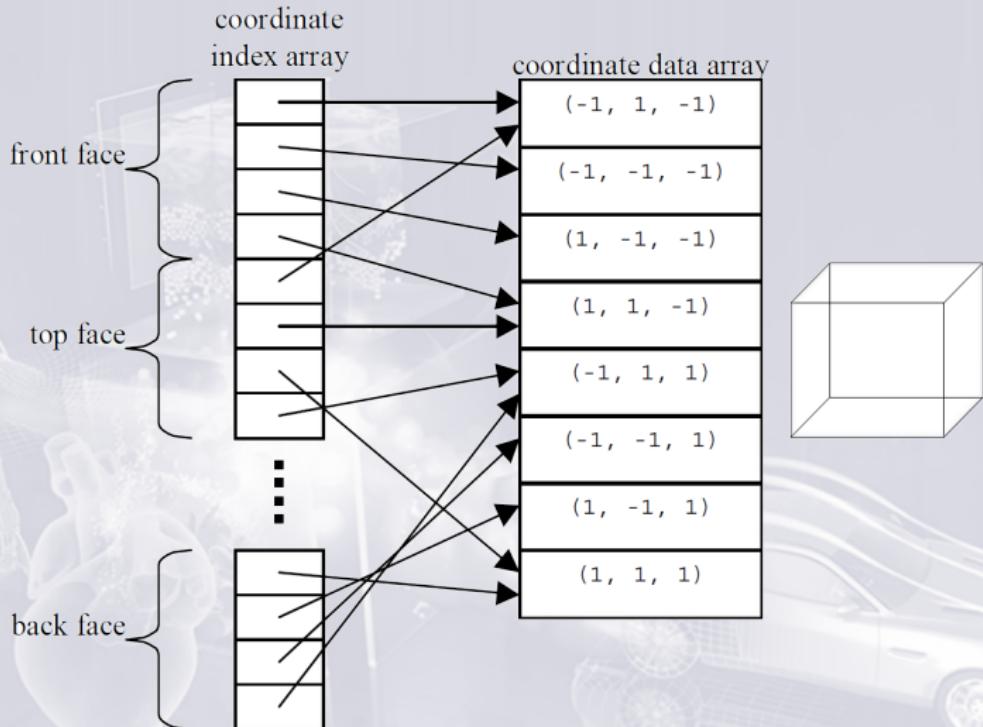
# Orden de los vértices

- Debe ser en sentido antihorario



# Geometría basada en vértices indexados

Clase **IndexedGeometryArray**, subclase de **GeometryArray > Geometry**



# Clases derivadas de IndexedGeometryArray

- IndexedPointArray
- IndexedLineArray
- IndexedTriangleArray
- IndexedQuadArray
- IndexedGeometryStripArray
  - ▶ IndexedLineStripArray
  - ▶ IndexedTriangleStripArray
  - ▶ IndexedTriangleFanArray

# Geometría indexada

## Constructores y métodos habituales

- Constructores para geometría indexada no encadenada
  - ▶ `IndexedTriangleArray (int numVertices, int formato, int numIndices)`
- Constructores para geometría indexada encadenada
  - ▶ `IndexedTriangleStripArray (int numVertices, int formato, int numIndices, int[] longCadenas)`
- Métodos que añaden las subclases
  - ▶ `void setCoordinateIndices (int i, int[] indicesCoord)`
  - ▶ `void setNormalIndices (int i, int[] indicesNormales)`
  - ▶ `void setTextureCoordinateIndices (int conjunto, int i, int[] indicesCoordTextura)`

# Ejemplos

## Geometría indexada no encadenada



### Ejemplo: IndexedLineArray

```
float[] coordenadas = { 0.0f, 0.0f, 0.0f,
    5.0f, 0.0f, 0.0f,  0.0f, 5.0f, 0.0f,  0.0f, 0.0f, 5.0f };

int[] indices = { 0, 1, 1, 3, 3, 0, 2, 0, 2, 1, 2, 3 };

IndexedGeometryArray lineas = new IndexedLineArray (
    coordenadas.length/3, GeometryArray.COORDINATES,
    indices.length );

lineas.setCoordinates (0, coordenadas);
lineas.setCoordinateIndices (0, indices);
```

# Ejemplos

## Geometría indexada encadenada

### Ejemplo: IndexedLineStripArray

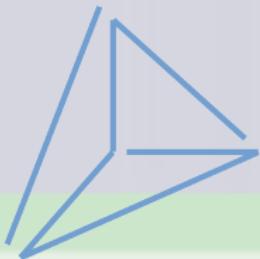
```
float[] coordenadas = { 0.0f, 0.0f, 0.0f,
    5.0f, 0.0f, 0.0f,  0.0f, 5.0f, 0.0f,  0.0f, 0.0f, 5.0f };

int[] indices = { 0, 1, 3, 0, 2, 1, 2, 3 };

int[] cadenas = { 6, 2 };

IndexedGeometryArray lineas = new IndexedLineStripArray (
    coordenadas.length/3, GeometryArray.COORDINATES,
    indices.length, cadenas );

lineas.setCoordinates (0, coordenadas);
lineas.setCoordinateIndices (0, indices);
```



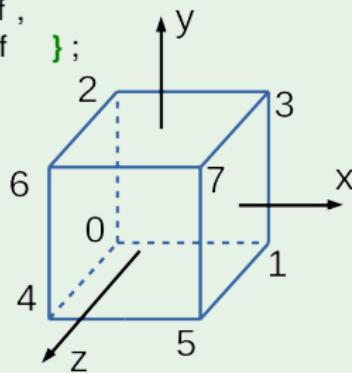
# Ejemplos

## Ejemplo completo de un cubo mediante índices

### Ejemplo: Un cubo (vértices y normales) mediante índices (1)

```
// Coordenadas de los vértices
float[] vertices = {
    -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f,
    -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f,
    -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f,
    -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f};

// Indices para formar las caras
int[] indicesVertices = {
    0, 2, 3, 1, // Cara trasera
    4, 5, 7, 6, // Frontal
    1, 3, 7, 5, // Derecha
    0, 4, 6, 2, // Izquierda
    0, 1, 5, 4, // Inferior
    2, 6, 7, 3 }; // Superior
```



# Ejemplo

## Ejemplo: Un cubo (vértices y normales) mediante índices (2)

```
// Coordenadas de las normales (vectores unitarios)
float[] normales = {
    1.0f,  0.0f,  0.0f,      // Cara derecha
   -1.0f,  0.0f,  0.0f,     // Izquierda
    0.0f,  1.0f,  0.0f,     // Superior
    0.0f, -1.0f,  0.0f,     // Inferior
    0.0f,  0.0f,  1.0f,     // Frontal
    0.0f,  0.0f, -1.0f };   // Posterior

// Indices para indicar las normales
int[] indicesNormales = {
    5, 5, 5, 5,      // Cara trasera
    4, 4, 4, 4,      // Frontal
    0, 0, 0, 0,      // Derecha
    1, 1, 1, 1,      // Izquierda
    3, 3, 3, 3,      // Inferior
    2, 2, 2, 2 };   // Superior
```

# Ejemplo

## Ejemplo: Un cubo (vértices y normales) mediante índices (y 3)

```
IndexedGeometryArray cuboGeometry = new IndexedQuadArray ( 
    vertices.length/3 ,
    GeometryArray.COORDINATES | GeometryArray.NORMALS,
    indicesVertices.length );

cuboGeometry.setCoordinates (0, vertices);
cuboGeometry.setCoordinateIndices (0, indicesVertices);

cuboGeometry.setNormals(0, normales);
cuboGeometry.setNormalIndices(0, indicesNormales);

// Se asume que ya se tiene el aspecto
Shape3D cuboShape = new Shape3D (cuboGeometry, aspecto);
```

# Importación de geometría

- Preferible cuando se desea modelar objetos complejos
- Java 3D incorpora clases para importar el formato .obj
  - ▶ Geometría (archivos .obj) y Aspecto (archivos .mtl)
- Los modelos se buscan usando los términos `wavefront model`



Modelo realizado por Turn 10 Studios descargado de [tf3dm.com](http://tf3dm.com)

# Importación de geometría

## Clase ObjectFile

- Permite cargar el formato de Wavefront Technologies
- El formato se basa en archivos de texto
  - ▶ El archivo `.obj` puede contener:
    - ★ Vértices
    - ★ Coordenadas de textura
    - ★ Vectores normales
    - ★ Definición de polígonos, normalmente triángulos
  - ▶ El archivo `.mtl` contiene definiciones de materiales

# Clase ObjectFile

## Constructor

- `ObjectFile ()`
- `ObjectFile (int flags)`
  - ▶ Flags
    - ★ `ObjectFile.RESIZE`,  
Escala el objeto a la caja englobante (-1,-1,-1) – (1,1,1)
    - ★ `ObjectFile.STRIPIFY`  
Intenta crear cadenas de triángulos
    - ★ `ObjectFile.TRIANGULATE`  
En caso de que el modelo tenga polígonos de más de 3 lados

# Clase Scene

## Uso del importador

### Ejemplo: Uso del importador

```
Scene escena = null;
ObjectFile archivo = new ObjectFile (ObjectFile.RESIZE |
    ObjectFile.STRIPIFY | ObjectFile.TRIANGULATE);
try {
    escena = archivo.load ("modelos/ferrariCalifornia.obj");
} catch (FileNotFoundException |
    ParsingErrorException |
    IncorrectFormatException e) {
    System.out.println (e);
    System.exit(1);
}
BranchGroup rama = new BranchGroup ();
rama.addChild (escena.getSceneGroup());
```

# Ejemplo de archivo .obj

## Ejemplo: Parte de un archivo .obj

```
# Wavefront OBJ file
# Archivo de materiales
mtllib california.mtl
# Vértices El primero es el 1 y así sucesivamente
v 0.80341 -0.22432 0.07015
v 0.83425 -0.22218 0.11258
.
.
#
# Coordenadas de textura
vt 0.02089 0.93665
vt 0.02412 0.92041
.
.
#
# Vectores normales
vn -0.51776 -0.85330 -0.06165
vn -0.59393 -0.72737 -0.34377
.
.
#
# Triángulos vértice / coord.textura / normal
usemtl redColor
f 4529/1/1 4530/2/2 4531/3/3
f 4532/4/4 4529/1/1 4531/3/3
.
.
```

# Ejemplo de archivo .mtl

## Ejemplo: Parte de un archivo .mtl

```
# Blender MTL file

newmtl redColor
Ns 1000.000000
Ka 0.019482 0.019482 0.019482
Kd 0.800000 0.062764 0.079847
Ks 0.386484 0.780355 0.700265
Ni 1.000000
d 1.000000
illum 2

newmtl tire
Ka 0.8 0.8 0.8
Kd 1 1 1
Ks 0.376471 0.376471 0.376471
illum 2
Ns 27.8576
map_Kd tireA0.png
```

# Contenidos

1 Objetivos

2 Introducción

3 Geometría

4 Organización y estructura: Nodos internos

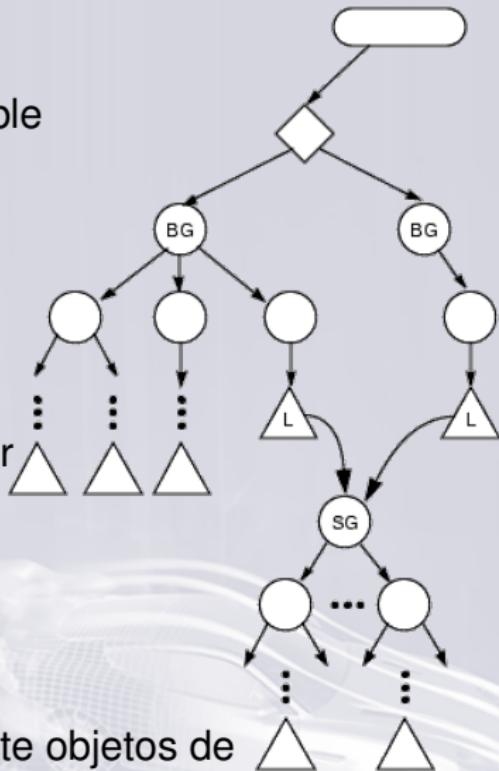
- Clase BranchGroup
- Clase TransformGroup
- Clase SharedGroup
- Clase Switch

5 Iluminación y materiales

6 Interacción y Animación

# Nodos internos

- Nodos que tienen un número variable de *hijos* en el grafo de escena
  - ▶ Indexados
  - ▶ Recorridos:
    - ★ En orden aleatorio
    - ★ En paralelo
- Usados para organizar y estructurar la información del grafo
- Al igual que los nodos terminales, solo pueden tener un *padre*
  - ▶ Según la relación padre-hijo
- Representados en Java 3D mediante objetos de clases derivadas de la clase *Group*



# Clase BranchGroup

Clase derivada de **Group**

- Es la raíz de un subgrafo que puede ser compilado
- Es el único nodo que puede ser añadido a un **Locale** (o a un **VirtualUniverse**)
  - ▶ Todo el subgrafo se considera *vivo*
- El único que puede ser insertado y borrado estando vivo
- Operaciones habituales:
  - ▶ `void addChild (Node hijo)`
  - ▶ `void removeChild (Node hijo)`
  - ▶ `void detach ()`
    - ★ Se separa de su parent si este es otro **BranchGroup**
- Capacidades habituales:
  - ▶ `Group.ALLOW_CHILDREN_EXTEND`
  - ▶ `Group.ALLOW_CHILDREN_WRITE`
  - ▶ `BranchGroup.ALLOW_DETACH`

# Clase BranchGroup

## Movimiento de ramas vivas

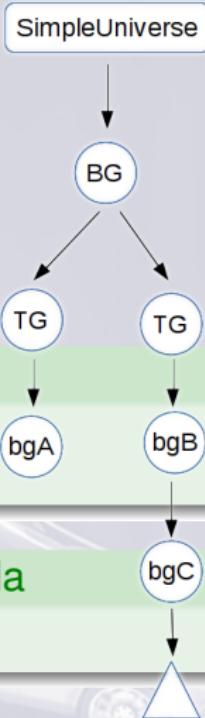
- Cuando la rama se hace viva, Java3D realiza optimizaciones
  - ▶ Se pierden cuando la rama deja de ser viva
  - ▶ ¿Cómo cambiar la rama de un sitio a otro?

**Forma incorrecta:** La rama se optimiza de nuevo

```
bgC.detach ();
bgA.addChild (bgC);
```

**Forma recomendada:** La rama se mantiene optimizada

```
bgA.moveTo (bgC);
```



# Clase BranchGroup

## Ejemplo

- Se suele usar como raíz de las partes de la escena que se desean manejar como un todo
- Es la clase de la que derivan las clases diseñadas para representar elementos de alto nivel de la escena

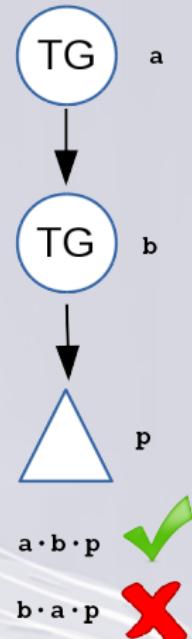
### Ejemplo: Uso habitual de la clase BranchGroup

```
public class RuedaBicicleta extends BranchGroup {  
    . . .  
}  
  
public class Bicicleta extends BranchGroup {  
    . . .  
}
```

# Clase TransformGroup

Clase derivada de **Group**

- Almacena una transformación geométrica
  - ▶ Escalados, rotaciones, traslaciones
  - ▶ Una composición de varias de ellas
- La transformación se aplica a los hijos de este grupo
- Hace uso de un objeto **Transform3D**
- Constructores y métodos específicos
  - ▶ `TransformGroup ()`
  - ▶ `TransformGroup (Transform3D transformacion)`
  - ▶ `void setTransform (Transform3D transformacion)`
- Capabilities para modificar un **TransformGroup vivo**
  - ▶ `TransformGroup.ALLOW_TRANSFORM_READ`
  - ▶ `TransformGroup.ALLOW_TRANSFORM_WRITE`



# Clase TransformGroup

## Ejemplo

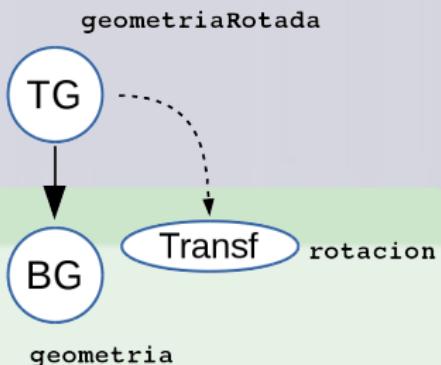
### Ejemplo: Uso de TransformGroup

```
// Se crea la transformación
Transform3D rotacion = new Transform3D ();
rotacion.rotZ (Math.PI/2);

// Se construye el TransformGroup con dicha transformación
TransformGroup geometriaRotada = new TransformGroup (rotacion);

// Se crea la geometría y se "cuelga" del TransformGroup
BranchGroup geometria = crearGeometria ();
geometriaRotada.addChild (geometria);

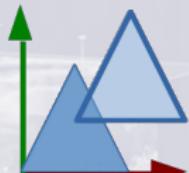
// El contenido del subgrafo geometria se ve afectado
// por una rotación de 90º respecto al eje Z
```



# Transformaciones geométricas

## Transformaciones habituales

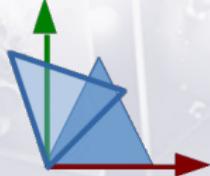
- Traslación



- Escalado uniforme



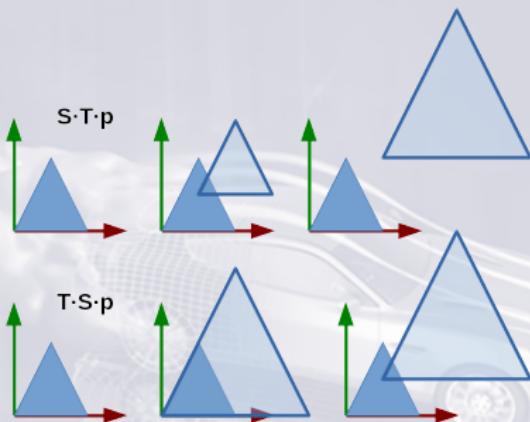
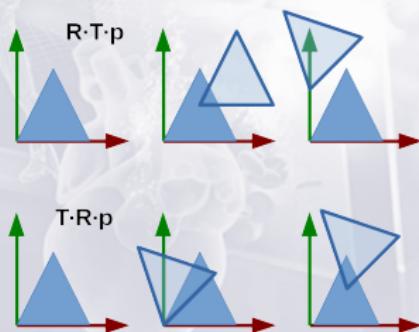
- Rotación



# Transformaciones geométricas

## Composición no conmutativa de transformaciones

- Composiciones conmutativas
  - ▶ Cualquier combinación de escalados y rotaciones
  - ▶ Cualquier combinación de traslaciones
- Composiciones no conmutativas
  - ▶ Las traslaciones con las rotaciones o los escalados



# Clase Transform3D

## Constructor, escalados, traslaciones y rotaciones

- Constructor: **Transform3D()**
- **Escalado:** Cambio de tamaño según un factor (1 lo deja igual)
  - ▶ Escalado uniforme
    - ★ `void set (double escala)`
  - ▶ Escalado no uniforme
    - ★ `void setScale (Vector3d escala)`
- **Traslación:** Desplazamiento según los valores indicados
  - ▶ `void set (Vector3f trans)` (puede ser `Vector3d`)
- **Rotación:** Giro al rededor de un eje que pasa por el origen
  - ▶ Ángulo en radianes
    - ★ Para usar ángulos sexagesimales:  
`double Math.toRadians (double angulo)`
    - ★ Para usar el número  $\pi$ : `Math.PI`

# Transform3D

## Rotación

- Ejes del sistema de coordenadas
  - ▶ `rotX (double ang)`   `rotY (double ang)`   `rotZ (double ang)`
- Un eje arbitrario, que pasa por el origen
  - ▶ `void set (AxisAngle4f rot)`
  - ▶ `void set (Quat4f rot)`
    - ★ Construcción del quaternion: `Quat4f()`
    - ★ Configuración del quaternion: `void set (AxisAngle4f rotacion)`
  - ★ Los objetos `AxisAngle4f` combinan eje de rotación y ángulo
    - `AxisAngle4f (float x, float y, float z, float angulo)`
    - `AxisAngle4f (Vector3f eje, float angulo)`  
(Con el sufijo `d` para doble precisión)

# Composición de transformaciones

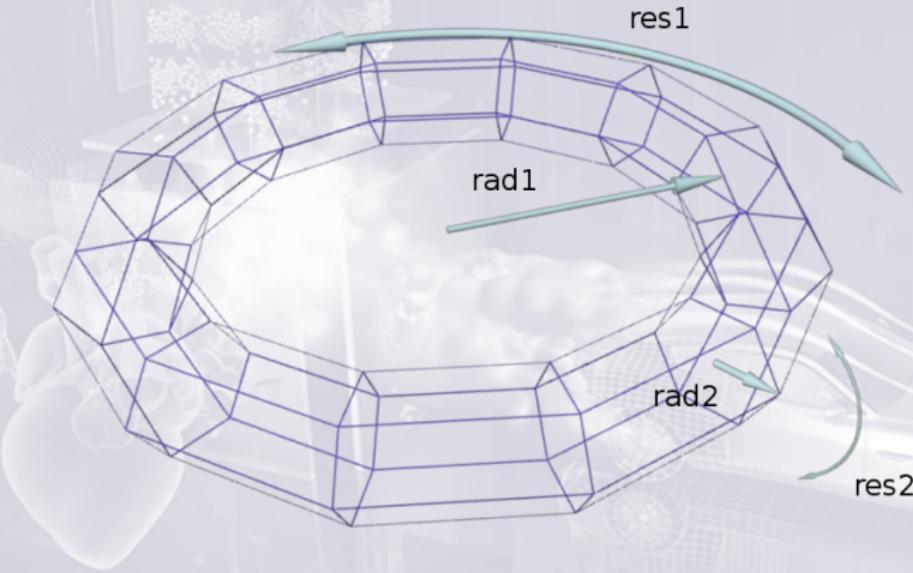
- void mul (Transform3D transformacion)  
 $this = this \cdot transformacion$
- void mul (Transform3D transf1, Transform3D transf2)  
 $this = transf1 \cdot transf2$

## • Operador de igualdad

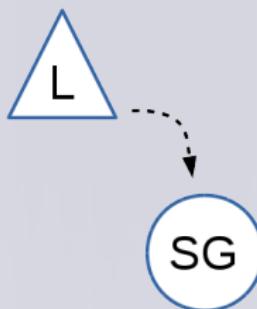
- ▶ boolean equals (Transform3D transformacion)
  - ★ Compara si las 2 transformaciones son iguales

# Ejercicio

- Crear una clase denominada `Torus` que derive de `Shape3D` y que represente a la figura geométrica `Toro`
  - ▶ Tendrá un único método público: su constructor
    - ★ `Torus (float rad1, float rad2, int res1, int res2, Appearance app)`



# Clase SharedGroup



- Permite reutilizar subgrafos de escena
- No puede tener *padre*
- Se utiliza mediante un nodo hoja Link
- Puede compilarse antes de ser referenciado por un Link
- El subgrafo que representa:
  - ▶ Puede tener como nodos terminales
    - ★ Shape3D, Light, Link
  - ▶ No puede tener como nodos terminales
    - ★ Background, Behavior

# Clase SharedGroup

## Ejemplo

### Ejemplo: Uso de SharedGroup

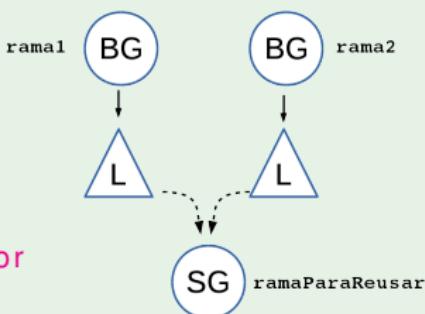
```
// Se crea la rama que quiere ser reutilizada
SharedGroup ramaParaReusar = crearEscenaReusable();

ramaParaReusar.compile();
```

```
BranchGroup rama1 = new BranchGroup();
BranchGroup rama2 = new BranchGroup();
```

// Modo erróneo de usar un SharedGroup  
 rama1.addChild (ramaParaReusar); // Error

// Modo correcto de usar y reusar un SharedGroup  
 rama1.addChild (new Link (ramaParaReusar));
 rama2.addChild (new Link (ramaParaReusar));



# Clase Switch

- Permite visualizar u ocultar sus hijos según una condición
  - ▶ Visualización de un hijo concreto
    - ★ `void setWhichChild (int indice)`
  - ▶ Visualización de varios hijos mediante una máscara
    - ★ `void setChildMask (BitSet mascara)`
    - ★ Entonces, `setWhichChild (Switch.CHILD_MASK)`
  - ▶ Para visualizar todos los hijos
    - ★ `void setWhichChild (Switch.CHILD_ALL)`
  - ▶ Para no visualizar ninguno
    - ★ `void setWhichChild (Switch.CHILD_NONE)`
  - ▶ Si la rama está viva necesita la capability
    - ★ `Switch.ALLOW_SWITCH_WRITE`

# Clase Switch

## Ejemplo

### Ejemplo: Uso de la clase Switch - Visualización de un solo hijo

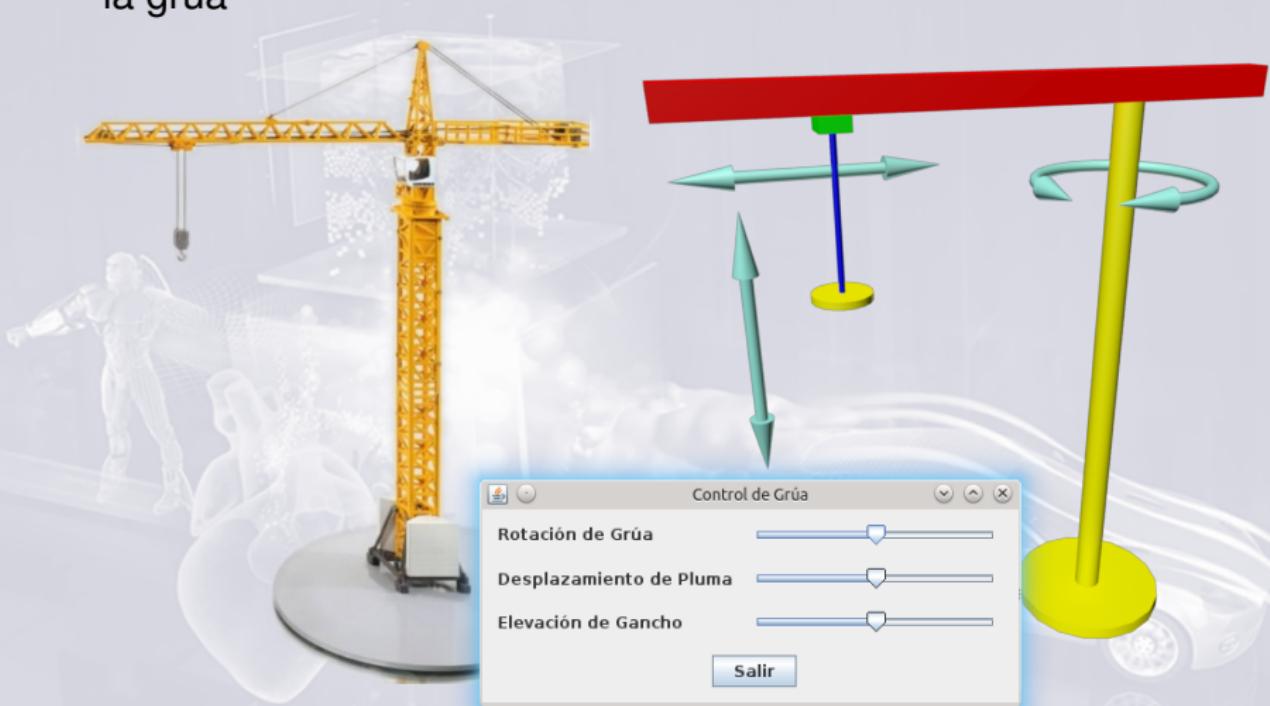
```
Switch padre = new Switch ();
padre.setCapability (Switch.ALLOW_SWITCH_WRITE);
padre.addChild (hijo0);
. . . // Se añaden todos los hijos
// Se indica el índice del hijo a visualizar
padre.setWhichChild (1);
```

### Ejemplo: Uso de la clase Switch - Visualización de varios hijos

```
// Primero se define la máscara
BitSet mascara = new BitSet (padre.numChildren ());
mascara.set (0, true); // Se indican los hijos a visualizar
mascara.set (2); // No poner boolean equivale a poner true
// Luego se asigna la máscara al objeto Switch
padre.setChildMask (mascara);
// Por último, se llama a setWhichChild
padre.setWhichChild (Switch.CHILD_MASK);
```

# Ejercicio

- Realizar una aplicación Java donde se modele y se pueda mover la grúa



# Contenidos

1 Objetivos

2 Introducción

3 Geometría

4 Organización y estructura: Nodos internos

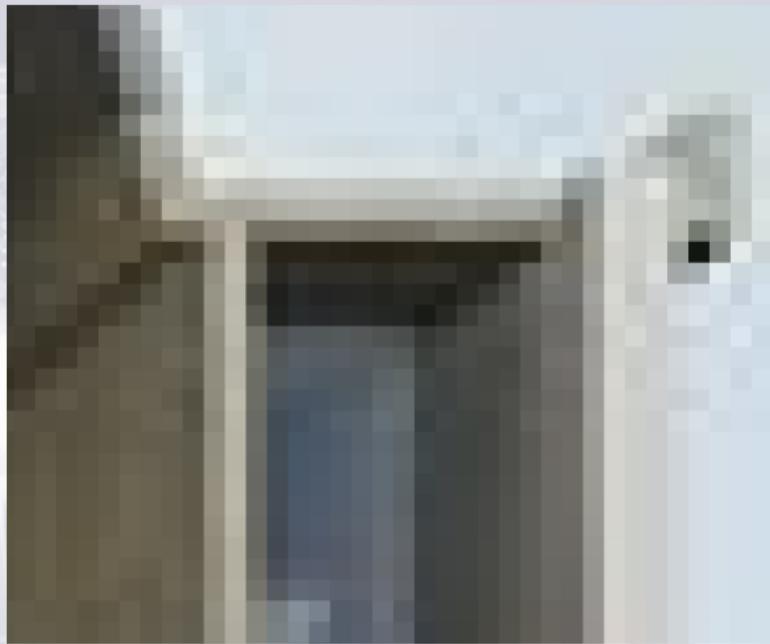
5 Iluminación y materiales

- Fuentes de luz
- Materiales de Lambert / Blinn
- Texturas

6 Interacción y Animación

# Introducción

Captando la realidad - Recordatorio de Modelo de iluminación



# Modelo de iluminación

Recordatorio

## Rendering

- Proceso de cálculo desarrollado por un ordenador destinado a generar una imagen o secuencia de imágenes.
- Influyen los elementos siguientes:

**Geometría:** (*Paso de 3-D a 2-D*)

Proyección, ocultación, distorsiones de la perspectiva, etc.

**Fotometría:** (*Luz reflejada por los objetos de la escena*)

Tipo, intensidad y dirección de la iluminación, reflectancia de las superficies, etc.

## Modelo de Iluminación

Conjunto de propiedades físicas de los objetos y de la luz que intervienen en el proceso de rendering.

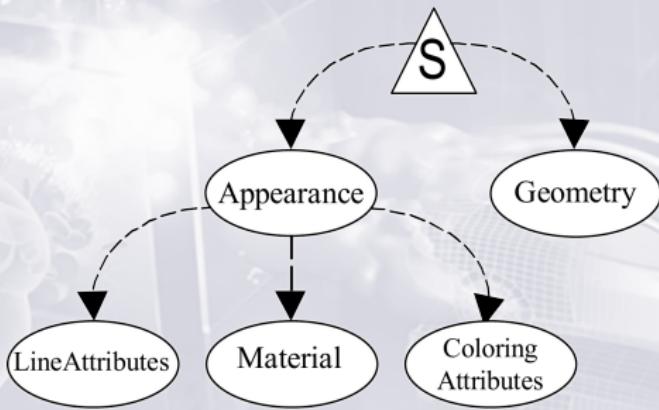
# Modelos de iluminación en Java 3D

- Un color por vértice
  - ▶ Asignado al crear la geometría
  - ▶ No tiene en cuenta otros parámetros
- Clase **Appearance**
  - ▶ Modelo no fotorrealista
    - ★ Basado en un color plano
  - ▶ **Modelo fotorrealista**
    - ★ Basado en propiedades físicas de los materiales
    - ★ Tiene en cuenta:
      - Las fuentes de luz
      - La posición del observador
    - ★ Permite el uso de texturas



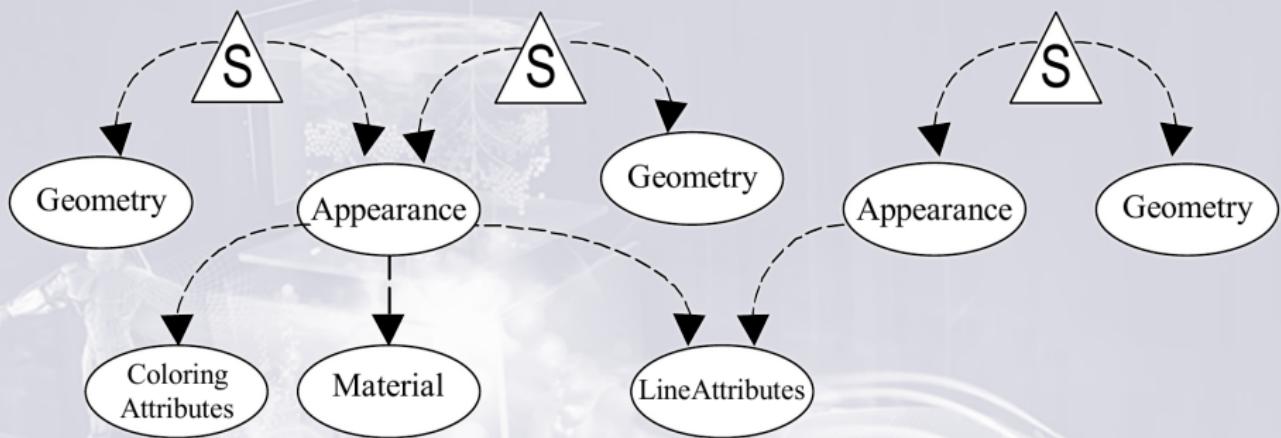
# Clase Appearance

- Es una componente de Shape3D
- Determina cómo se visualiza la geometría de dicho Shape3D
- No contiene directamente información sobre el aspecto
- Contiene referencias a diversos componentes de aspecto



# Clase Appearance

- Los aspectos y sus componentes pueden ser compartidos



- Se establecen con métodos **set\***
  - Ejemplo: `void setMaterial (Material mat)`

# Modelo fotorrealista de Java 3D

## Modelo de Lambert / Blinn

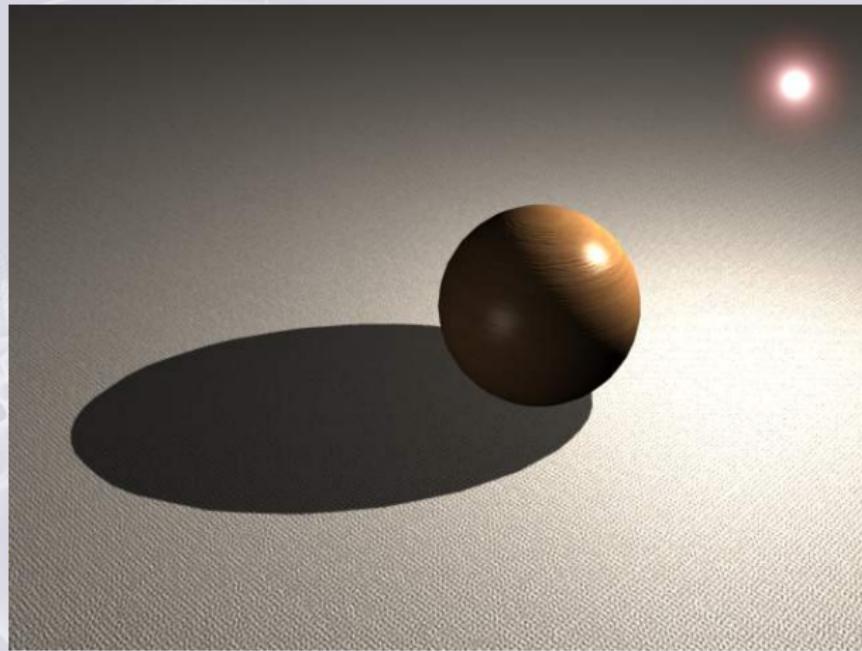


- Zonas
  - ▶ Iluminadas solo por luz indirecta
  - ▶ Iluminadas por luz directa
  - ▶ Iluminadas por luz directa y que producen brillos
- En el material, según este modelo de iluminación, se van a distinguir 3 componentes
  - ▶ Ambiental: Cómo reacciona el material ante la luz indirecta
  - ▶ Difusa: Cómo reacciona el material ante la luz directa
  - ▶ Especular: La poseen aquellos materiales que producen brillos y están aplicados a figuras con superficie pulida

# Modelo fotorrealista de Java 3D

## Consideraciones

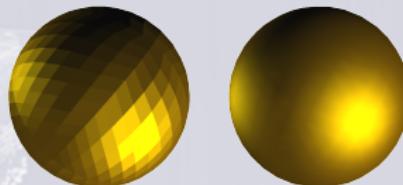
- Sombras propias vs. sombras arrojadas



# Modelo fotorrealista de Java 3D

## Consideraciones

- Sombreado Plano y Gouraud

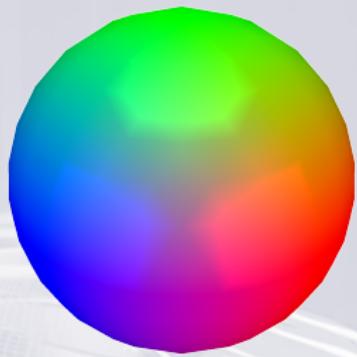


- Vector de la luz y del observador
  - ▶ Si son constantes se reducen considerablemente los cálculos
    - ★ La luz direccional implica un vector constante
    - ★ El vector del observador es constante por defecto
- Influencia de la luz
  - ▶ Se establece un entorno
  - ▶ Las figuras que caen en dicho entorno son iluminadas
  - ▶ A cada objeto no le pueden influir más de 8 luces simultáneamente

# Modelo fotorrealista de Java 3D

## Consideraciones

- Interacción entre objetos
  - ▶ No se considera (sombras arrojadas, reflejos, etc.)
- El modelo de iluminación se aplica solo en los vértices



- Ejercicio: ¿Por qué se ve la esfera de la derecha peor?

# Modelo fotorrealista de Java 3D

## Requisitos

- Relativos a las fuentes de luz
  - ▶ Deben existir
  - ▶ Tener un ámbito de influencia
  - ▶ Estar añadidas al grafo de escena
- Relativos a las figuras
  - ▶ Deben tener vectores normales
  - ▶ El **Appearance** de la figura debe tener un **Material**

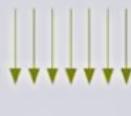
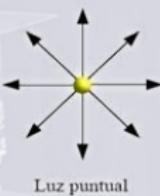
# Fuentes de luz

## Clase Light

- Clase abstracta que deriva de Leaf > Node

- Sus subclases son:

- ▶ AmbientLight
- ▶ DirectionalLight
- ▶ PointLight
  - ★ SpotLight

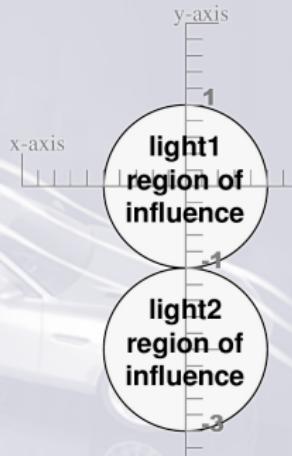
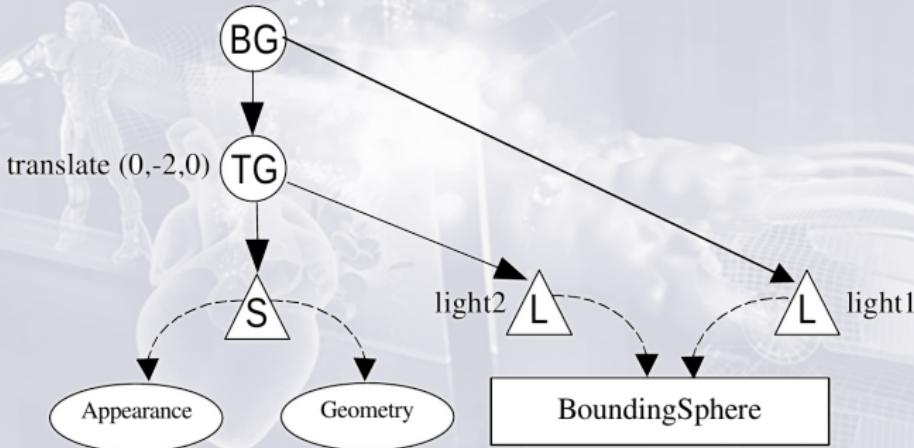


- Métodos

- ▶ `void setColor (Color3f color)`
- ▶ `void setInfluencingBounds (Bounds ámbito)`
- ▶ `void setEnable (boolean onOff)`
  - ★ Capacidad para encender y apagar la fuente estando viva `Light.ALLOW_STATE_WRITE`

# Clase Bounds

- Clase abstracta que define un volumen convexo
- Clases derivadas habituales
  - ▶ BoundingSphere (`Point3d centro, double radio`)  
`BoundingSphere ()` Equivalente a  $((0,0,0), 1.0)$
  - ▶ BoundingBox (`Point3d desde, Point3d hasta`)  
`BoundingBox ()` Equivalente a  $((-1,-1,-1), (1,1,1))$
- Le afectan las transformaciones, según su posición en el grafo



# Alcance de una fuente de luz

- Reducción de la influencia de una luz a grupos ([Group](#)) concretos
- No sustituye al [Bounds](#)
- Métodos:
  - ▶ `void addScope (Group alcance)`
  - ▶ `void removeScope (int posición)`
- Capability: [Light.ALLOW\\_SCOPE\\_WRITE](#)
- Si no hay un alcance asignado,  
se considera que *todo* el grafo está alcanzado

# Clase AmbientLight

- Representa la luz que hay en la escena resultado de múltiples reflexiones
- Java 3D no la calcula, la define el desarrollador
- Es constante en todo su volumen de influencia
- Solo interviene en el cálculo de la componente ambiental del modelo de iluminación
- Constructor
  - ▶ `AmbientLight (Color3f color)`
  - ▶ `AmbientLight ()` color blanco

# Clase DirectionalLight

- Representa una fuente de luz muy lejana, por ejemplo, el sol
- Sus rayos llegan a la escena paralelos
- Intensidad constante en todo el volumen de influencia
- Se define indicando el vector de dirección de los rayos de luz
- Interviene en el cálculo de las componentes difusa y especular del modelo de iluminación
- Constructor
  - ▶ `DirectionalLight (Color3f color, Vector3f direccion)`
  - ▶ `DirectionalLight ()` color blanco y dirección  $(0, 0, -1)$
- Cambio de dirección estando viva
  - ▶ Capacidad `DirectionalLight.ALLOW_DIRECTION_WRITE`
  - ▶ Método `void setDirection (Vector3f nuevaDireccion)`

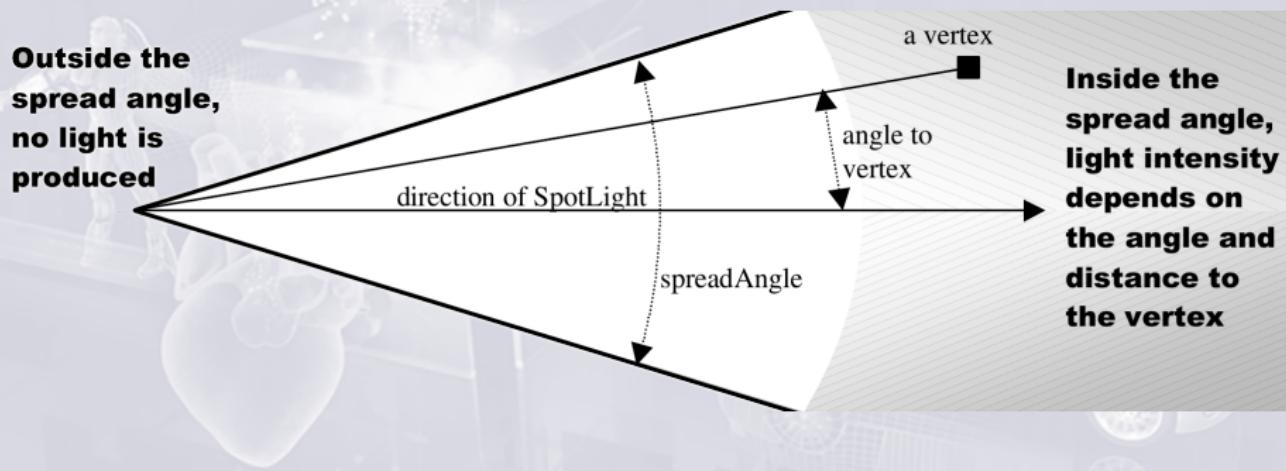
# Clase PointLight

- Fuente de luz situada en una posición concreta
- Emite sus rayos en todas las direcciones y su intensidad puede atenuarse con la distancia
- Interviene en el cálculo de las componentes difusa y especular del modelo de iluminación
- Constructor
  - ▶ `PointLight (Color3f color, Point3f posición,  
Point3f atenuación)`
  - ▶ `PointLight ()` color blanco, posic. (0, 0, 0) y atenuac. (1, 0, 0)  
$$\text{atenuacion} = \frac{1}{\text{constante} + \text{lineal} \cdot \text{distancia} + \text{cuadratica} \cdot \text{distancia}^2}$$
- Cambios estando viva
  - ▶ Capabilities `ALLOW_POSITION_WRITE`, `ALLOW_ATTENUATION_WRITE`
  - ▶ Métodos `void setPosition (Point3f nuevaPosicion)`  
`void setAttenuation (Point3f nuevaAtenuacion)`

# Clase SpotLight

Subclase de PointLight

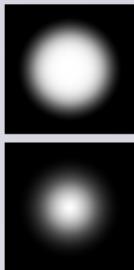
- Añade una dirección, un ángulo y una concentración (atenuación angular)
- Interviene en el cálculo de las componentes difusa y especular del modelo de iluminación



# Clase SpotLight

- Constructor

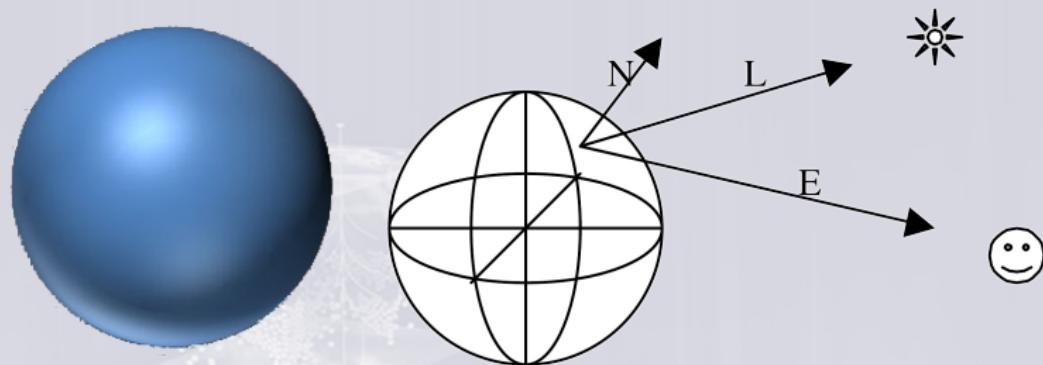
- ▶ `SpotLight (Color3f color, Point3f posición,  
Point3f atenuación, Vector3f dirección,  
float ángulo, float concentración)`
- ▶ `SpotLight ()` dirección  $(0, 0, -1)$ , ángulo  $\pi$  radianes,  
concentración 0
- ▶ Consideraciones sobre algunos parámetros:
  - ★ Ángulo: Se mide desde el vector de dirección
  - ★ Concentración (c):
    - Toma valores en  $[0, 128]$
    - Atenúa según la expresión  $atenuacion = \cos^c \alpha$



- Cambios estando viva

- ▶ Capabilities añadidas `ALLOW_DIRECTION_WRITE`,  
`ALLOW_SPREAD_ANGLE_WRITE`, `ALLOW_CONCENTRATION_WRITE`
- ▶ Métodos
  - `void setDirection (Vector3f nuevaDirección)`
  - `void setSpreadAngle (float nuevoAngulo)`
  - `void setConcentration (float nuevaConcent)`

# Definición de un material en Java 3D



- Material según el modelo de Lambert / Blinn
  - ▶ Zonas iluminadas solo por luz indirecta
    - ★ Se define el **color ambiental**
  - ▶ Zonas iluminadas por luz directa
    - ★ Se define el color **difuso**
  - ▶ Zonas iluminadas por luz directa y que producen brillos
    - ★ Se define el color **especular** y el **brillo**

# Material

## Componente Emisiva

- Simula que un objeto emite luz
- Su color es constante e independiente de las fuentes de luz
- Se puede combinar con las otras componentes



# Clase Material

## Constructores

- `Material (`  
    `Color3f ambientColor,`  
    `Color3f emissiveColor,`  
    `Color3f diffuseColor,`  
    `Color3f specularColor,`  
    `float shininess ) // Debe estar entre 1.0f y 128.0f`
- `Material () ≈ Material (`  
    `new Color3f (0.2f, 0.2f, 0.2f),`  
    `new Color3f (0.0f, 0.0f, 0.0f),`  
    `new Color3f (1.0f, 1.0f, 1.0f),`  
    `new Color3f (1.0f, 1.0f, 1.0f),`  
    `64 )`

# Clase Material

## Setters

- Capability: `Material.ALLOW_COMPONENT_WRITE`
- Pueden recibir tanto un `Color3f`
  - ▶ `void setAmbientColor (Color3f color)`
  - ▶ `void setEmissiveColor (Color3f color)`
  - ▶ `void setDiffuseColor (Color3f color)`
  - ▶ `void setSpecularColor (Color3f color)`
- Como las componentes RGB
  - ▶ `void setAmbientColor (float r, float g, float b)`
  - ▶ `void setEmissiveColor (float r, float g, float b)`
  - ▶ `void setDiffuseColor (float r, float g, float b)`
  - ▶ `void setSpecularColor (float r, float g, float b)`
  - ▶ `void setShininess (float shininess)`

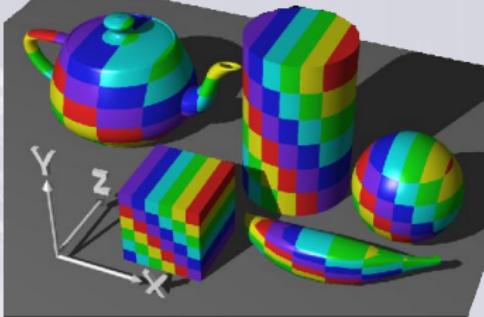
# Clase Material

## Ejemplos

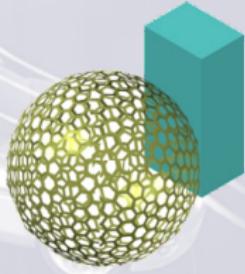
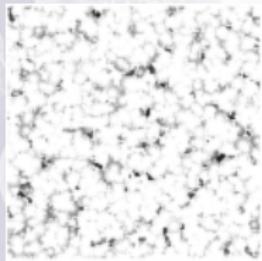
Material	Color difuso	Color Especular	Brillo
Azul mate	.00 .00 .99	.22 .22 .67	1
Azul brillante	.00 .00 .99	.99 .99 .99	75
Azul plástico	.20 .20 .70	.85 .85 .85	22
Azul metálico	.00 .00 .20	.00 .00 .99	2
Aluminio	.37 .37 .37	.89 .89 .89	17
Cobre	.30 .10 .00	.75 .30 .00	10
Oro	.49 .34 .00	.89 .79 .00	17
Ónix	.00 .00 .00	.72 .72 .72	23

# Texturas

- Permiten usar una imagen para colorear un objeto



- O modificar otro tipo de características (no incluido en Java3D)



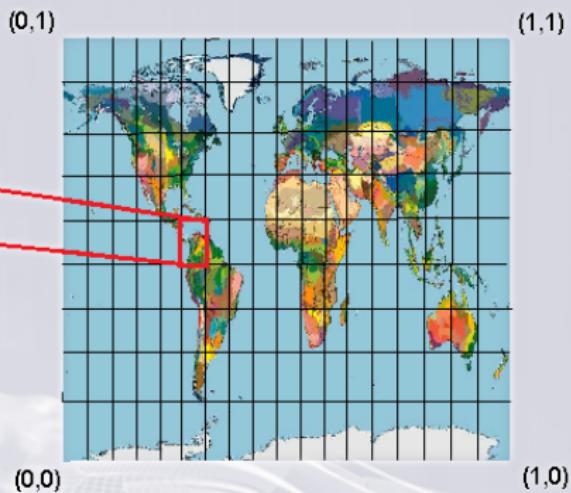
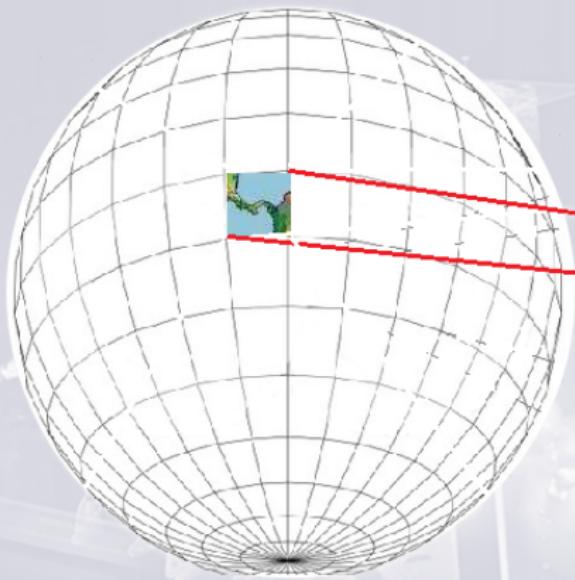
# Texturas

## Requisitos

- El `Shape3D` debe poseer coordenadas de textura
- El aspecto debe tener la componente `Texture`
- La textura debe estar habilitada (`setEnabled (true)`)
  - ▶ Para habilitar/deshabilitar la textura estando viva
    - ★ Se requiere que la textura tenga la capacidad `Texture.ALLOW_ENABLE_WRITE`
  - ▶ Para modificar cualquier otro parámetro
    - ★ Se requiere que el aspecto tenga la capacidad `Appearance.ALLOW_TEXTURE_WRITE`
    - ★ Entonces, se crea otra textura y se sustituye la antigua por completo

# Texturas

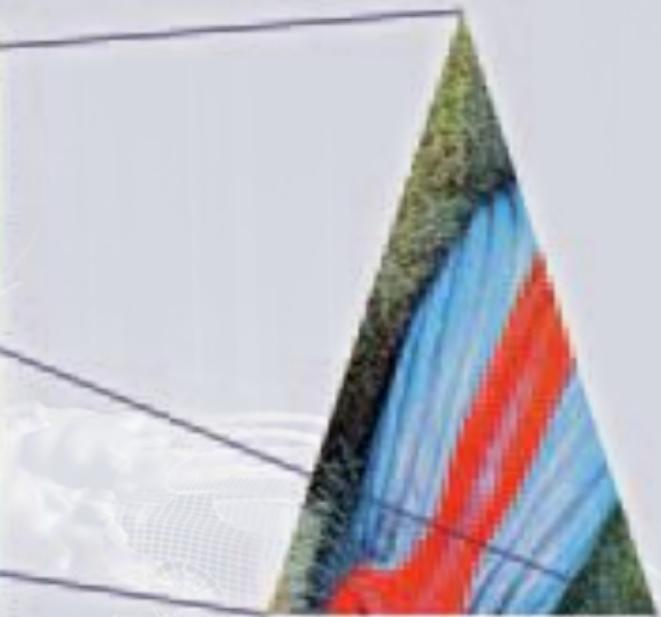
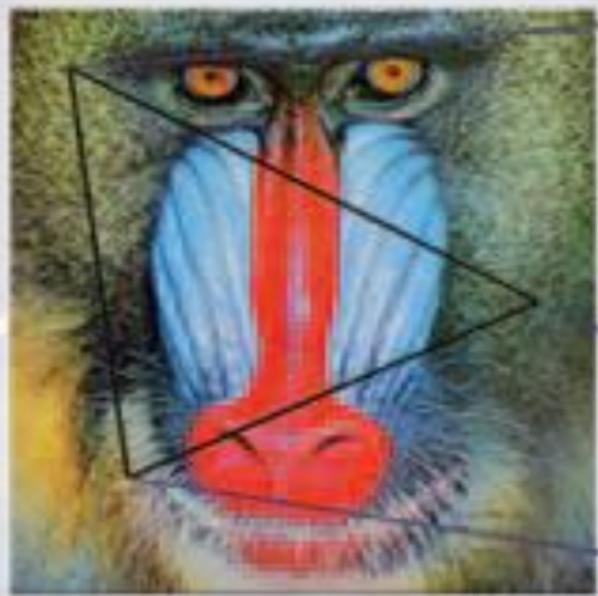
## Coordenadas de textura (1)



El polígono marcado tiene estas coordenadas de textura:  
 $(0.278, 0.444) - (0.333, 0.556)$

# Texturas

## Coordenadas de textura (2)



# Texturas

## Coordenadas de textura (3)



- Ejemplo: Texturizado de un cubo según la imagen

# Texturas

## Coordenadas de textura (4)

### Ejemplo: Coordenadas de textura (1)

```
float[] coordenadas = {  
    -1.0f, -1.0f, -1.0f,      // Vértice 0  
    1.0f, -1.0f, -1.0f,      // 1  
    -1.0f, 1.0f, -1.0f,      // 2  
    1.0f, 1.0f, -1.0f,      // 3  
    -1.0f, -1.0f, 1.0f,      // 4  
    1.0f, -1.0f, 1.0f,      // 5  
    -1.0f, 1.0f, 1.0f,      // 6  
    1.0f, 1.0f, 1.0f       // 7  
};
```

- Ejemplo: Definición de los vértices del cubo

# Texturas

## Coordenadas de textura (5)

### Ejemplo: Coordenadas de textura (2)

```
int[] indices = {  
    0, 2, 3, 1, // Cara trasera  
    4, 5, 7, 6, // Frontal  
    1, 3, 7, 5, // Derecha  
    0, 4, 6, 2, // Izquierda  
    0, 1, 5, 4, // Inferior  
    2, 6, 7, 3 // Superior  
};
```

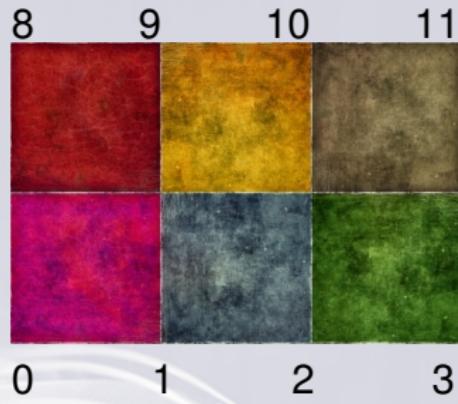
- Ejemplo: Definición de los índices que conforman las caras

# Texturas

## Coordenadas de textura (6)

### Ejemplo: Coordenadas de textura (3)

```
float[] coordenadasTextura = {  
    0.00f, 0.00f, // 0  
    0.33f, 0.00f, // 1  
    0.67f, 0.00f, // 2  
    1.00f, 0.00f, // 3  
    0.00f, 0.50f, // 4  
    0.33f, 0.50f, // 5  
    0.67f, 0.50f, // 6  
    1.00f, 0.50f, // 7  
    0.00f, 1.00f, // 8  
    0.33f, 1.00f, // 9  
    0.67f, 1.00f, // 10  
    1.00f, 1.00f // 11  
};
```



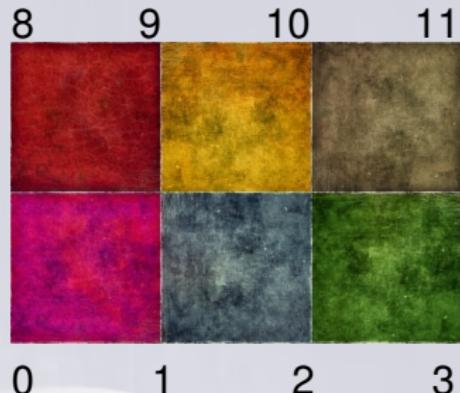
- Ejemplo: Definición de las coordenadas de textura

# Texturas

## Coordenadas de textura (7)

### Ejemplo: Coordenadas de textura (4)

```
int [] indicesTextura = {  
    0, 1, 5, 4,    // Cara trasera  
    1, 2, 6, 5,    // Frontal  
    2, 3, 7, 6,    // Derecha  
    4, 5, 9, 8,    // Izquierda  
    5, 6, 10, 9,   // Inferior  
    6, 7, 11, 10   // Superior  
};
```



- Ejemplo: Definición de los índices que indican, para cada vértice de cada cara, que coordenada de textura se tiene que usar

# Texturas

## Coordenadas de textura (8)

### Ejemplo: Coordenadas de textura (y 5)

```
// Creación del cubo

IndexedQuadArray cubo = new IndexedQuadArray (
    coordenadas.length/3,
    IndexedQuadArray.COORDINATES |
    IndexedQuadArray.TEXTURE_COORDINATE_2,
    indices.length);

cubo.setCoordinates (0, coordenadas);
cubo.setCoordinateIndices (0, indices);

cubo.setTextureCoordinates (0, 0, coordenadasTextura);
cubo.setTextureCoordinateIndices(0, 0, indicesTextura);
```

# Texturas

## Coordenadas de textura (y 9)

- **Cuidado:** Debe haber, al menos, tantos vértices como parejas de coordenadas de textura

### Ejemplo: Aumento del vector de vértices (si es necesario)

```
float[] extendCoordinateVector (float[] coordinates,
                               int textureCoordinateVectorSize) {
    int nPoints = coordinates.length/3;
    int nTextureCoordinates = textureCoordinateVectorSize/2;
    if (nPoints >= nTextureCoordinates) { return coordinates; }
    else {
        int i, j;
        int resultSize = nTextureCoordinates * 3;
        float[] result = new float[resultSize];
        for (i = 0; i < coordinates.length; i++) {
            result[i] = coordinates[i];
        }
        for (j = i; j < resultSize; j++) { result[j] = 0.0f; }
        return result;
    }
}
```

# Clase Texture

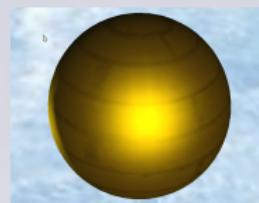
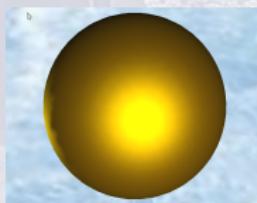
## Carga de la textura

- Se usa una clase para cargar la imagen
  - ▶ Puede estar en formato PNG o JPG
  - ▶ `TextureLoader tmp = new TextureLoader ("img.jpg", null);`
  - ▶ `Texture textura = tmp.getTexture();`
- Debería tener un tamaño potencia de 2 en cada dimensión
  - ▶ No necesariamente iguales ambas dimensiones
- La clase TextureLoader realiza el escalado si es necesario
- Es preferible preparar las imágenes previamente

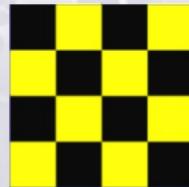
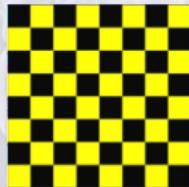
# Clase TextureAttributes

## Atributos de textura

- Una componente del aspecto que define cómo se aplica la textura
- Incluye:
  - ▶ Modo de textura
    - ★ Cómo se combinan los colores de la textura con los de la figura



- ▶ Transformación del mapa de textura
  - ★ Permite modificar las coordenadas de textura



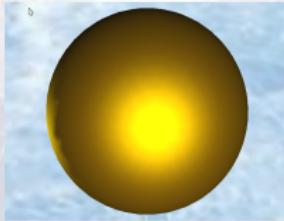
# Clase TextureAttributes

## Modo de textura

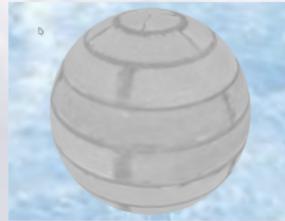
- `void setTextureMode (int modo)`
- El color final del objeto es una *mezcla* entre el color dado mediante el material y el color dado mediante la textura
  - ▶ **TextureAttributes.REPLACE** (modo por defecto)
    - ★ Solo se aplica el color de la textura
    - ★ No se obtienen luces y sombras
  - ▶ **TextureAttributes.MODULATE**
    - ★ Se mezcla el color de la textura con el color del material
    - ★ Se obtienen luces y sombras



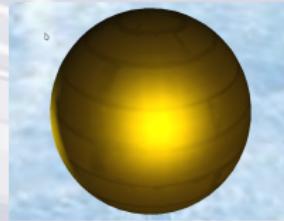
Textura



Solo Material



REPLACE

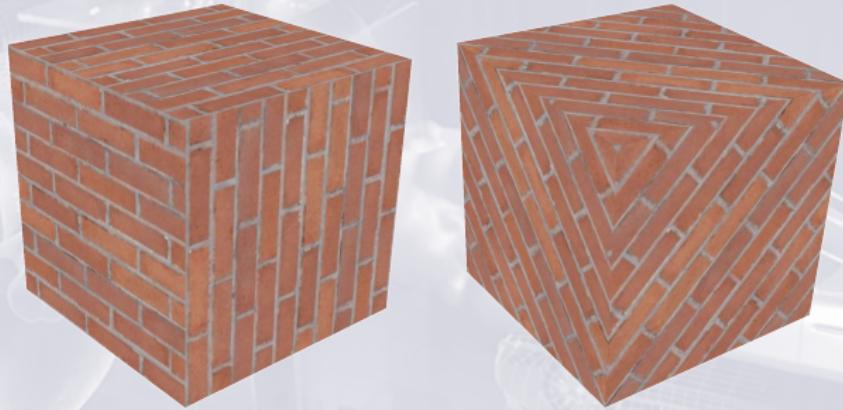


MODULATE

# Clase TextureAttributes

## Transformación del mapa de textura

- `void setTextureTransform (Transform3D transformacion)`
- Permite trasladar, rotar o escalar la textura en la figura
- Capability: `TextureAttributes.ALLOW_TRANSFORM_WRITE`
- Ejemplo: Textura girada 45°



# Creación de un Appearance fotorrealista

- Crear y configurar las componentes:
  - ▶ Material
  - ▶ Texture
  - ▶ TextureAttributes
- Crear y configurar el Appearance
  - ▶ Mediante sus métodos set\*
    - ★ setMaterial
    - ★ setTexture
    - ★ setTextureAttributes

# Creación de un Appearance fotorrealista

## Ejemplo

- Ejemplo de Appaerance que combina:
  - ▶ Un material Lambert / Blinn
  - ▶ Una textura en el canal difuso



## Ejemplo: Creación de un Appearance fotorrealista (1)

```
// Se crean y configuran los componentes  
  
// El material Lambert / Blinn  
Material material = new Material (  
    new Color3f (0.20f, 0.20f, 0.20f),      // Color ambiental  
    new Color3f (0.00f, 0.00f, 0.00f),      // Color emisivo  
    new Color3f (0.50f, 0.50f, 0.50f),      // Color difuso  
    new Color3f (0.70f, 0.70f, 0.70f),      // Color especular  
    17.0f );                                // Brillo
```

# Creación de un Appearance fotorrealista

## Ejemplo

### Ejemplo: Creación de un Appearance fotorrealista (y 2)

```
// La textura
Texture textura = new TextureLoader ("tierra.jpg",null).getTexture();

// Los atributos de textura, para combinar material y textura
TextureAttributes texAttr = new TextureAttributes ();
texAttr = texAttr.setTextureMode (TextureAttributes.MODULATE);

// Se crea y se configura el aspecto
Appearance aspecto = new Appearance ();
aspecto.setMaterial (material);
aspecto.setTexture (textura);
aspecto.setTextureAtributes (texAttr);

// La figura debe tener normales y coordenadas de textura
Sphere esfera = new Sphere (radio,
    Primitive.GENERATE_NORMALS | Primitive.GENERATE_TEXTURE_COORDS,
    aspecto);
```

# Contenidos

- 1 Objetivos
- 2 Introducción
- 3 Geometría
- 4 Organización y estructura: Nodos internos
- 5 Iluminación y materiales
- 6 Interacción y Animación
  - Clase Behavior
  - Interacción con teclado y ratón
  - Picking
  - Detección de colisiones
  - Animación

# Introducción

- **Interacción**

- ▶ Acción que se ejerce recíprocamente entre dos o más objetos, agentes, fuerzas, funciones, etc.

- **Animación**

- ▶ Dotar de movimiento a cosas inanimadas

- En ambos casos implica:

- ▶ Ejecutar código ...
    - ★ Cambiar una transformación, un aspecto, eliminar un nodo del grafo de escena, añadirlo, etc.
  - ▶ ... como respuesta a algo
    - ★ Una acción del usuario, en el caso de la interacción
    - ★ El paso del tiempo, en el caso de la animación

- La clase **Behavior** proporciona esa funcionalidad

# Clase Behavior

- Clase abstracta
- Permite ejecutar código en respuesta a un estímulo
  - ▶ La pulsación de una tecla
  - ▶ Una acción con el ratón
  - ▶ El paso del tiempo
- Se actúa sobre determinados objetos del grafo de escena
  - ▶ Deben tener las *capabilities* adecuadas

# Clase Behavior

## Desarrollo de una clase derivada

- La clase concreta que derive de Behavior tiene que:
  - ▶ Tener una referencia al objeto sobre el que se va a actuar
    - ★ En el constructor y/o en métodos set\*
  - ▶ Sobreescribir el método

```
public void initialization ()
```

    - ★ Es ejecutado cuando el *behavior* se hace vivo
    - ★ Debe establecer la condición de respuesta
  - ▶ Sobreescribir el método

```
public void processStimulus (Enumeration criterios)
```

    - ★ Es ejecutado cuando la condición de respuesta se produce
    - ★ Decodifica los criterios
    - ★ Actúa sobre los objetos referenciados
    - ★ Debe establecer la siguiente condición de respuesta

# Clase Behavior

Ejemplo: Rotación de una figura al pulsar una tecla (1)

Ejemplo: Clase concreta que implementa un comportamiento (1)

```
public class RotarFigura extends Behavior {  
    // Objeto referenciado  
    private TransformGroup referencia;  
    // Condición de respuesta  
    WakeupOnAWTEvent condicion = new WakeupOnAWTEvent (KeyEvent.  
        KEY_PRESSED);  
    // Otras variables de instancia necesarias  
    private Transform3D rotacion = new Transform3D ();  
    private double angulo = 0.0;  
  
    public RotarFigura (TransformGroup laReferencia) {  
        referencia = laReferencia;  
    }  
  
    @Override  
    public void initialize () {  
        wakeupOn (condicion);  
    }  
    . . .
```

# Clase Behavior

Ejemplo: Rotación de una figura al pulsar una tecla (y 2)

Ejemplo: Clase concreta que implementa un comportamiento (y 2)

```
    . . .
@Override
public void processStimulus (Enumeration criterios) {

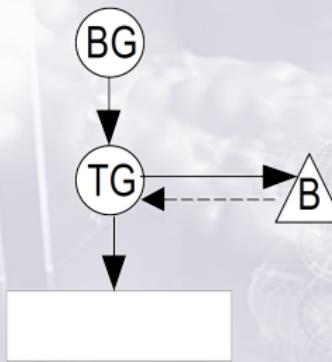
    // Se actúa sobre el objeto referenciado
    angulo += 0.1;
    rotacion.rotY (angulo);
    referencia.setTransform (rotacion);

    // Se establece la siguiente condición de respuesta
    wakeupOn (condicion);
}
```

# Clase Behavior

## Uso de la clase

- Preparar el grafo de escena
  - ▶ Establecer las *capabilities* adecuadas en los objetos que vayan a modificarse
- Construir el *behavior* con los objetos que deba modificar
- Ponerle un ámbito de activación
- Insertar el *behavior* en el grafo de escena



# Clase Behavior

## Ejemplo de uso

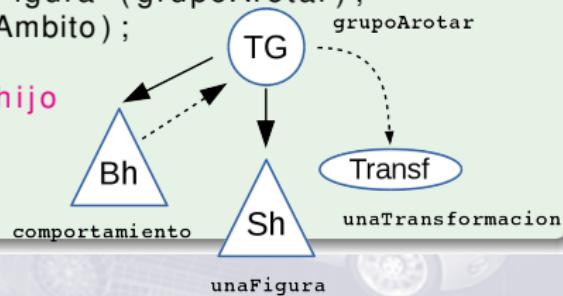
### Ejemplo: Uso de una clase derivada de Behavior

```
// La transformación que después se lee y se actualiza
Transform3D unaTransformacion = new Transform3D () ;

// El nodo que es modificado por el comportamiento
TransformGroup grupoArotar = new TransformGroup (unaTransformacion);
grupoArotar.setCapability (TransformGroup.ALLOW_TRANSFORM_READ) ;
grupoArotar.setCapability (TransformGroup.ALLOW_TRANSFORM_WRITE) ;

// El comportamiento
RotarFigura comportamiento = new RotarFigura (grupoArotar);
comportamiento.setSchedulingBounds (unAmbito);

// Se establecen las relaciones padre-hijo
grupoArotar.addChild (comportamiento);
grupoArotar.addChild (unaFigura);
unBranchGroup.addChild (grupoArotar);
```



# Clase WakeupCondition

- Clase abstracta que representa las condiciones de respuesta, también conocidas como disparadores
- Clases derivadas:
  - ▶ `WakeupCriterion`
    - ★ Clase abstracta que aglutina las condiciones de respuesta
  - ▶ Varias clases para *componer* condiciones
    - ★ `WakeupOr`
    - ★ `WakeupAnd`
    - ★ `WakeupAndOfOrs`
    - ★ `WakeupOrOfAnds`
- Métodos
  - ▶ `Enumeration triggeredElements ()`
    - ★ Devuelve la enumeración de las condiciones que han activado el Behavior

# Clase WakeupCondition

## Subclase WakeupCriterion (1)

- Clase abstracta que representa las condiciones de respuesta, también conocidas como disparadores
- Clases derivadas para **gestión de Teclado y Ratón**:
  - ▶ `WakeupOnAWTEvent`
    - ★ Pulsaciones de teclas y eventos de ratón
    - ★ `KeyEvent.KEY_TYPED`, `KEY_PRESSED`, `KEY_RELEASED`
    - ★ `MouseEvent.MOUSE_CLICKED`, `MOUSE_PRESSED`, `MOUSE_RELEASED`,  
`MOUSE_MOVED`, `MOUSE_DRAGGED`
    - ★ Constructor:
      - `WakeupOnAWTEvent (int unEvento)`
    - ★ Método:
      - `AWTEvent[] getAWTEvent ()`

# Clase WakeupCondition

## Subclase WakeupCriterion (2)

- Clase abstracta que representa las condiciones de respuesta, también conocidas como disparadores
- Clases derivadas para **detección de Colisiones**:
  - ▶ `WakeupOnCollisionEntry`
    - ★ Ocurre cuando dos objetos inician la colisión
  - ▶ `WakeupOnCollisionMovement`
    - ★ Ocurre mientras dos objetos están colisionando
  - ▶ `WakeupOnCollisionExit`
    - ★ Ocurre cuando dos objetos dejan de colisionar



# Clase WakeupCondition

Subclase **WakeupCriterion** (y 3)

- Clase abstracta que representa las condiciones de respuesta, también conocidas como disparadores
- Clases derivadas para **realización de Animación**:
  - ▶ `WakeupOnElapsedFrames`
    - ★ Ocurre cada cierto número de frames
    - ★ Constructor: `WakeupOnElapsedFrames (int n)`
  - ▶ `WakeupOnElapsedTime`
    - ★ Ocurre cada cierto número de milisegundos, o lo antes posible transcurrido ese tiempo
    - ★ Constructor: `WakeupOnElapsedTime (long milisegundos)`

# Clase WakeupCondition

## Composición de condiciones

- Pueden combinarse las condiciones mediante AND y OR
- Se usan las siguientes clases derivadas de WakeupCondition
- Constructores
  - ▶ `WakeupAnd (WakeupCriterion[] condiciones)`
  - ▶ `WakeupOr (WakeupCriterion[] condiciones)`
  - ▶ `WakeupAndOfOrs (WakeupOr[] condiciones)`
  - ▶ `WakeupOrOfAnds (WakeupAnd[] condiciones)`

# Interacción con el teclado

- Mediante una clase se capturan y se procesan las pulsaciones
- Puede disponer de:
  - ▶ Una referencia a la fachada para enviarle las respectivas órdenes
  - ▶ Varias referencias a los distintos objetos con los que se interactúa
- Las referencias se inicializarán
  - ▶ Mediante el constructor
  - ▶ Mediante métodos `set*`,  
con los cuales también se podrán actualizar
- La condición de respuesta será normalmente  
`KeyEvent.KEY_PRESSED`

# Interacción con el teclado

## Ejemplo (1)

### Ejemplo: Clase para procesar pulsaciones de teclado (1)

```
public class ProcesaTeclado extends Behavior {  
    // Objeto(s) referenciado(s)  
    // Condición de respuesta  
    WakeupOnAWTEvent condicion =  
        new WakeupOnAWTEvent (KeyEvent.KEY_PRESSED);  
  
    public ProcesaTeclado () {  
        // Constructor  
        // Puede tener parámetros para inicializar las referencias  
        // La clase puede tener métodos set para inicializar o  
        //     actualizar las referencias  
    }  
  
    @Override  
    public void initialize () {  
        wakeupOn (condicion);  
    }  
    . . .
```

# Interacción con el teclado

## Ejemplo (y 2)

### Ejemplo: Clase para procesar pulsaciones de teclado (y 2)

```
    . . .
    @Override
public void processStimulus (Enumeration criterios) {
    // Se extrae de los criterios la tecla pulsada
    WakeupOnAWTEvent unCriterio =
        (WakeupOnAWTEvent) criterios.nextElement();
    AWTEvent[] eventos = unCriterio.getAWTEvent();
    KeyEvent tecla = (KeyEvent) eventos[0];
    // Procesamiento de las teclas con carácter
    switch (tecla.getKeyChar ()) {
        // Diversos cases
        default : // Procesamiento de las teclas sin carácter
            switch (tecla.getKeyCode ()) {
                // Diversos cases
            } // del switch interno
    } // del switch externo
    // Se establece la siguiente condición de respuesta
    wakeupOn (condicion);
} // del método    } // de la clase
```

# Interacción con el ratón

- Se procede de manera similar a la interacción con el teclado
- Las condiciones de respuesta serán
  - ▶ MouseEvent.MOUSE\_PRESSED
  - ▶ MouseEvent.MOUSE\_RELEASED
  - ▶ MouseEvent.MOUSE\_CLICKED
  - ▶ MouseEvent.MOUSE\_DRAGGED
- Se usará la clase MouseEvent para conocer
  - ▶ Qué botones se han pulsado
  - ▶ La posición del puntero en el canvas

# Clase MouseEvent

- Para interpretar los eventos del ratón en la clase anterior pueden necesitarse los siguientes métodos:
  - ▶ `int getID ()`
    - ★ Indica el evento concreto que se ha producido
      - `MouseEvent.MOUSE_PRESSED`, etc.
  - ▶ `int getButton ()`
    - ★ Indica el botón que ha cambiado de estado:
      - `MouseEvent.BUTTON1`, el botón izquierdo
      - `MouseEvent.BUTTON2`, el botón central
      - `MouseEvent.BUTTON3`, el botón derecho
      - `MouseEvent.NOBUTTON`, ningún botón
  - ▶ `int getX (), int getY ()`
    - ★ Devuelve las posiciones X e Y del píxel clicado dentro del canvas

# Case InputEvent

- De esta clase derivan las clases `KeyEvent` y `MouseEvent`
- Los siguientes métodos se pueden consultar en ambas clases:
  - ▶ `boolean isShiftDown ()`
    - ★ Indica si la tecla *Mayúsculas* está pulsada cuando se ha producido el evento (una pulsación de otra tecla o un evento del ratón)
  - ▶ `boolean isControlDown ()`
    - ★ Similar con la tecla *Control*
  - ▶ `boolean isAltDown ()`
    - ★ Similar con la tecla *Alt*
  - ▶ `boolean isAltGraphDown ()`
    - ★ Similar con la tecla *Alt Graph*

# Interacción con el ratón

## Ejemplo (1)

### Ejemplo: Clase para procesar eventos del ratón (1)

```
public class ProcesaRaton extends Behavior {  
    // Objeto(s) referenciado(s)  
    // Condición de respuesta  
    private WakeupOnAWTEvent[] eventosAescuchar = {  
        new WakeupOnAWTEvent (MouseEvent.MOUSE_PRESSED),  
        new WakeupOnAWTEvent (MouseEvent.MOUSE_RELEASED),  
        new WakeupOnAWTEvent (MouseEvent.MOUSE_DRAGGED)  
    };  
    private WakeupCondition condicion;  
  
    public ProcesaRaton () {  
        // Constructor  
        // Puede tener parámetros para inicializar las referencias  
        // La clase puede tener métodos set para inicializar o  
        //     actualizar las referencias  
  
        condicion = new WakeupOr (eventosAescuchar);  
    }  
    . . .
```

# Interacción con el ratón

## Ejemplo (y 2)

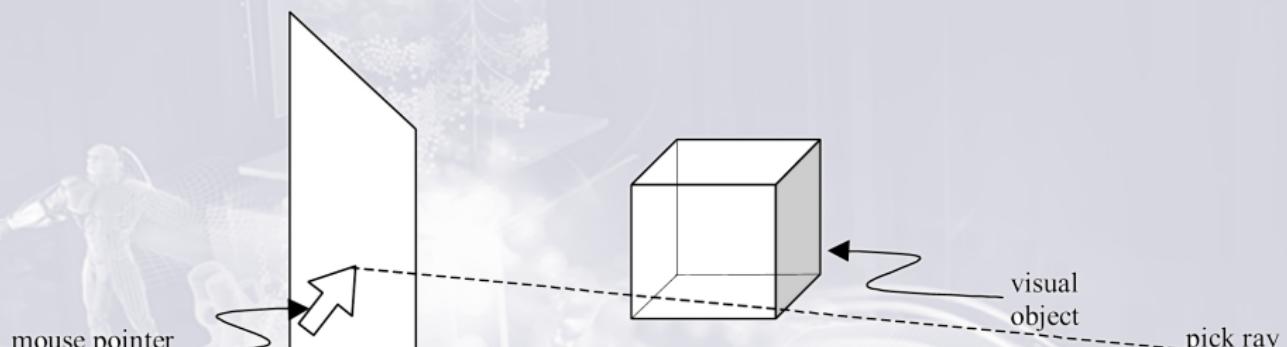
### Ejemplo: Clase para procesar eventos del ratón (y 2)

```
    . . .
@Override
public void initialize () {
    wakeupOn (condicion);
}

@Override
public void processStimulus (Enumeration cond) {
    WakeupOnAWTEvent c = (WakeupOnAWTEvent) cond.nextElement();
    AWTEvent[] e = c.getAWTEvent ();
    MouseEvent raton = (MouseEvent) e[0];
    switch (raton.getID ()) {
        case MouseEvent.MOUSE_CLICKED :
            // Procesamiento
            break;
    }
    // Establecer de nuevo la condición de respuesta
    wakeupOn (condicion);
}
```

# Picking

- Implementado como un comportamiento que responde a los botones del ratón



mouse pointer

image plate

visual  
object

pick ray

# Picking

## Optimización

- Es una operación costosa
- `void setPickable (false)`
  - ▶ El nodo que reciba este mensaje y TODOS sus descendientes no forman parte del proceso de picking
- Parámetro `PICK_BOUNDS` en vez de `PICK_GEOMETRY`
  - ▶ Usado en algunos constructores de clases de Picking
- Limitar el alcance
  - ▶ A las operaciones de picking se les pasa un nodo del grafo de escena a partir del cual se realiza la búsqueda
  - ▶ Evitar usar el nodo raíz del grafo

# Clase PickCanvas

- Permite realizar un Pick a partir de la posición indicada en el canvas con el ratón
- Constructor:
  - ▶ `PickCanvas (Canvas3D canvas, BranchGroup raizBusqueda)`
- Configuración (1):
  - ▶ `void setTolerance (float tolerancia)`
    - ★ Radio de búsqueda, en píxeles, en torno al píxel *clicado*
    - ★ Por defecto tiene un valor de 2.0
    - ★ Un valor de 0.0 acelera el proceso ligeramente pero hace más difícil seleccionar líneas o puntos
  - ▶ `void setMode (int modo)`
    - ★ `PickInfo.PICK_BOUNDS`, usa la caja enblobante de los objetos
    - ★ `PickInfo.PICK_GEOMETRY`, usa la geometría de los objetos

# Clase PickCanvas

- Configuración (2):

- `void setFlags (int máscara)`

- ★ Se le indica qué información debe calcular
    - ★ `PickInfo.NODE`
      - Calcula Shape3D intersecado
    - ★ `PickInfo.SCENEGRAPHPATH`
      - Calcula un objeto de la clase `SceneGraphPath`
      - Contiene la transformación acumulada desde la raíz del grafo hasta el Shape3D intersecado
    - ★ `PickInfo.CLOSEST_INTERSECTION_POINT`,
      - Calcula el punto de intersección más cercano en coordenadas locales del Shape3D intersecado

# Clase PickCanvas

- Configuración (y 3):

- ▶ `void setFlags (int máscara)`
  - ★ `PickInfo.CLOSEST_GEOM_INFO`
    - Calcula solo la información relativa a la geometría más cercana
  - ★ `PickInfo.ALL_GEOM_INFO`
    - Calcula la información de todas las geometrías que intersecan el rayo del pick

- Uso:

- ▶ `void setShapeLocation (MouseEvent e)`
  - ★ Se le indica la posición del ratón para hacer el pick
- ▶ `PickInfo pickClosest ()`
  - ★ Devuelve información sobre la geometría más cercana
- ▶ `PickInfo[] pickAllSorted ()`
  - ★ Devuelve información sobre todas las geometrías intersecadas, ordenadas desde la más cercana a la más lejana

# Clase PickInfo

- Representa la información asociada a una geometría intersecada en un Pick
- Métodos:
  - ▶ `Node getNode ()`
    - ★ Devuelve el Shape3D sobre el que se ha clicado
  - ▶ `SceneGraphPath getSceneGraphPath ()`
    - ★ Devuelve el camino de nodos del grafo, desde el Locale hasta el nodo terminal clicado
    - ★ Permite obtener la transformación acumulada
  - ▶ `Point3d getClosestIntersectionPoint ()`
    - ★ Devuelve el punto concreto donde se ha hecho clic en la geometría intersecada
    - ★ El punto está en el sistema de coordenadas local de la geometría

# Ejemplo

Seleccionar un Shape3D para asignarle un Appearance (1)

Clase: PickForSettingAppearance (1)

```
public class PickForSettingAppearance extends Behavior {  
    private Appearance appearance;  
    private WakeupOnAWTEvent condition;  
    private PickCanvas pickCanvas;  
    private Canvas3D canvas;  
  
    public PickForSettingAppearance (Canvas3D aCanvas) {  
        canvas = aCanvas;  
        condition = new WakeupOnAWTEvent (MouseEvent.MOUSE_CLICKED);  
    }  
  
    public void setAppearance (Appearance ap, BranchGroup bg) {  
        appearance = ap;  
        pickCanvas = new PickCanvas (canvas, bg);  
        pickCanvas.setTolerance (0.0f);  
        pickCanvas.setMode (PickInfo.PICK_BOUNDS);  
        pickCanvas.setFlags (PickInfo.NODE | PickInfo.CLOSEST_GEOM_INFO);  
        setEnable (true);  
    }  
}
```

# Ejemplo

Seleccionar un Shape3D para asignarle un Appearance (y 2)

Clase: PickForSettingAppearance (y 2)

```
@Override  
public void initialize () {  
    setEnable (false);  
    wakeupOn (condition);  
}  
  
@Override  
public void processStimulus (Enumeration cond) {  
    WakeupOnAWTEvent c = (WakeupOnAWTEvent) cond.nextElement ();  
    AWTEvent[] e = c.getAWTEvent ();  
    MouseEvent mouse = (MouseEvent) e[0];  
    pickCanvas.setShapeLocation (mouse);  
    PickInfo pi = pickCanvas.pickClosest ();  
    Shape3D shape = (Shape3D) pi.getNode ();  
    shape.setAppearance (appearance);  
    setEnable (false);  
    wakeupOn (condition);  
}
```

# Ejemplo

Conocer la coordenada exacta donde se ha hecho clic (1)

## Clase: PickForGettingXYZ (1)

```
public class PickForGettingXYZ extends Behavior {  
    private Point3d point;  
    private WakeupOnAWTEvent condition;  
    private PickCanvas pickCanvas;  
    private Canvas3D canvas;  
  
    public PickForGettingXYZ (Canvas3D aCanvas) {  
        canvas = aCanvas;  
        condition = new WakeupOnAWTEvent (MouseEvent.MOUSE_CLICKED);  
    }  
  
    public void initSearch (BranchGroup bg) {  
        pickCanvas = new PickCanvas (canvas, bg);  
        pickCanvas.setTolerance (0.0f);  
        pickCanvas.setMode (PickInfo.PICK_GEOMETRY);  
        pickCanvas.setFlags (PickInfo.SCENEGRAPHPATH |  
                            PickInfo.CLOSEST_INTERSECTION_POINT);  
        setEnable (true);  
    }  
}
```

# Ejemplo

Conocer la coordenada exacta donde se ha hecho clic (y 2)

## Clase: PickForGettingXYZ (y 2)

```
@Override  
public void initialize () {  
    setEnable (false);  
    wakeupOn (condition);  
}  
  
@Override  
public void processStimulus (Enumeration cond) {  
    WakeupOnAWTEvent c = (WakeupOnAWTEvent) cond.nextElement ();  
    AWTEvent[] e = c.getAWTEvent ();  
    MouseEvent mouse = (MouseEvent) e[0];  
    pickCanvas.setShapeLocation (mouse);  
    PickInfo pi = pickCanvas.pickClosest ();  
    point = pi.getClosestIntersectionPoint ();  
    SceneGraphPath sgp = pi.getSceneGraphPath ();  
    Transform3D t3d = sgp.getTransform ();  
    t3d.transform (point);  
    // Realizar lo que proceda con esa posición  
    setEnable (false);    wakeupOn (condition);    }    }
```

# Ejemplo

## Clase Pick única (1)

### Clase: GenericPick (1)

```
public class GenericPick extends Behavior {  
    private AppStatus status;  
    private WakeupOnAWTEvent[] conditionsToListen = {  
        new WakeupOnAWTEvent (MouseEvent.MOUSE_PRESSED),  
        new WakeupOnAWTEvent (MouseEvent.MOUSE_RELEASED),  
        new WakeupOnAWTEvent (MouseEvent.MOUSE_DRAGGED)  
    };  
    private WakeupCondition condition;  
    private PickCanvas pickCanvas;  
    private Canvas3D canvas;  
  
    public GenericPick (Canvas3D aCanvas) {  
        canvas = aCanvas;  
        condition = new WakeupOr (conditionsToListen);  
        status = AppStatus.Nothing;  
    }  
}
```

# Ejemplo

## Clase Pick única (2)

### Clase: GenericPick (2)

```
// Diversos métodos set* cambiarán el estado
//      y establecerán otros objetos

public void setStatus (AppStatus aStatus , BranchGroup bg) {
    status = aStatus;
    pickCanvas = new PickCanvas (canvas , bg);
    pickCanvas.setTolerance (0.0f);
    pickCanvas.setMode (PickInfo.PICK_GEOMETRY);
    pickCanvas.setFlags (PickInfo.SCENEGRAPHPATH |
                        PickInfo.CLOSEST_GEOM_INFO |
                        PickInfo.CLOSEST_INTERSECTION_POINT);
}

@Override
public void initialize () {
    wakeupOn (condition);
}
```

# Ejemplo

## Clase Pick única (y 3)

### Clase: GenericPick (y 3)

```
@Override  
public void processStimulus (Enumeration cond) {  
    WakeupOnAWTEvent c = (WakeupOnAWTEvent) cond.nextElement();  
    AWTEvent[] e = c.getAWTEvent ();  
    MouseEvent mouse = (MouseEvent) e[0];  
    switch (status) {  
        case AppStatus.Nothing :  
            break;  
        case // Diversos estados de la aplicación  
            // Según cada estado  
            //      se realiza el pick y el procesamiento que proceda  
        }  
        // Establecer de nuevo la condición de respuesta  
        wakeupOn (condition);  
    }  
}
```

# Métodos `setUserData` y `getUserData`

- Permiten *enlazar* los Shape3D pickados con las clases del diseño

## Ejemplo: Uso de `setUserData` y `getUserData` (1)

```
// Al definir la clase y construir el grafo

class Lampara extends BranchGroup {
    private Light luz;
    Lampara () {
        // Se construye el grafo de la lámpara
        .
        .
        .
        // A cada Primitive que se use ...
        Cylinder base = new Cylinder ( ... );
        // ... se le asigna this como datos externos
        base.setUserData (this);
        .
        .
        .
    }
    void enciendeApaga () {
        luz.setEnable ( !luz.getEnable() );
    }
}
```

# Métodos setUserData y getUserData

Ejemplo: Uso de setUserData y getUserData (y 2)

```
// Al implementar el pick

class PickLampara extends Behavior {
    .
    .
    void processStimulus (Enumeration cond) {
        .
        .
        // Se obtiene el Shape3D concreto como siempre
        Shape3D shape = (Shape3D) pi.getNode ();

        // Se accede a la primitiva que contiene dicho Shape3D
        Primitive primitiva = (Primitive) shape.getParent();

        // Se obtiene el objeto externo asociado a dicha primitiva
        Lampara lampara = (Lampara) primitiva.getUserData();

        // Se le envía el mensaje que corresponda
        lampara.enciendeApaga();

        .
    }
}
```

# Detección de colisiones

- Se implementa con clases Behavior
- Usando como disparadores objetos de las clases:
  - ▶ `WakeupOnCollisionEntry`
  - ▶ `WakeupOnCollisionMovement`
  - ▶ `WakeupOnCollisionExit`
- Constructor:
  - ▶ `WakeupOnCollisionEntry (Node nodo, int modo)`
  - ▶ Donde modo puede ser:
    - ★ `WakeupOnCollisionEntry.USE_BOUNDS`
    - ★ `WakeupOnCollisionEntry.USE_GEOMETRY`
- ¿Cómo saber con qué se colisiona?
  - ▶ `SceneGraphPath getTriggeringPath ()`

# Detección de colisiones

## Ejemplo (1)

### Ejemplo: Una clase para detectar colisiones (1)

```
public class Colision extends Behavior {  
    // Se detectan colisiones del objetoA contra otros  
    private Node objetoA;  
    private WakeupOnCollisionEntry condicion;  
  
    Colision (Node unNodo) {  
        objetoA = unNodo;  
        // Se usa BOUNDS por rapidez  
        condicion = new WakeupOnCollisionEntry (objetoA,  
            WakeupOnCollisionEntry.USE_BOUNDS);  
        // El ámbito del Behavior se ajusta al del objetoA  
        this.setSchedulingBounds (objetoA.getBounds());  
    }  
  
    public void initialize () {  
        wakeupOn (condicion);  
    }  
    . . .
```

# Detección de colisiones

## Ejemplo (y 2)

**Ejemplo:** Una clase para detectar colisiones (y 2)

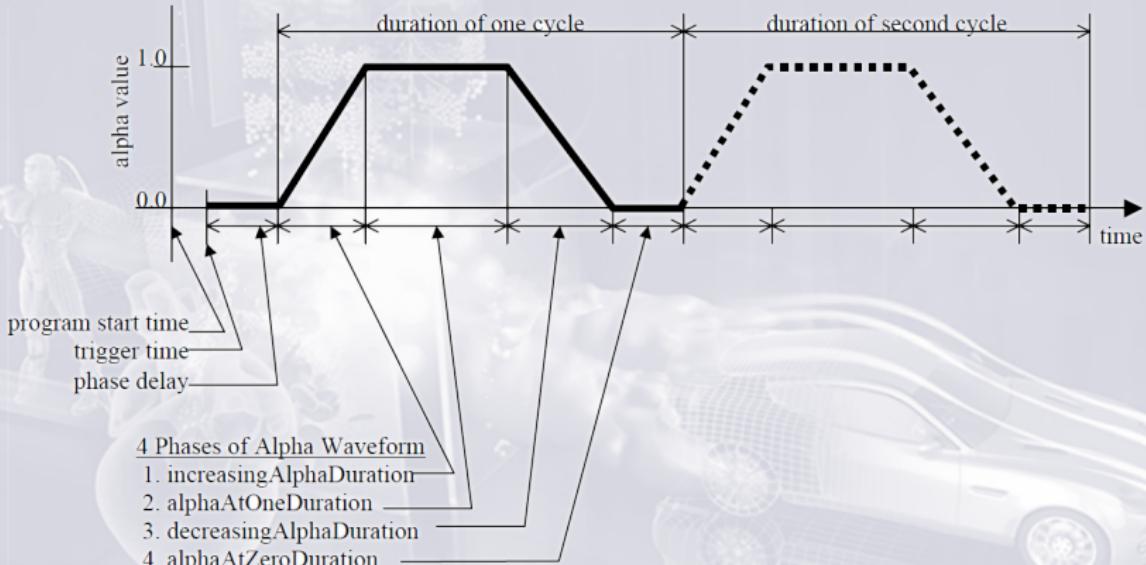
```
public class Colision extends Behavior {  
    . . .  
  
    public void processStimulus (Enumeration criteria) {  
        // Se obtiene el objeto contra el que ha colisionado objetoA  
        WakeupOnCollisionEntry disparo = (WakeupOnCollisionEntry)  
            criteria.nextElement();  
        Node objetoB = disparo.getTriggeringPath () . getObject ();  
        // Se procesa la colisión  
        . . .  
        // Se establece de nuevo el disparador  
        wakeupOn (condicion);  
    }  
}
```

# Animación

- Se basa en clases Behavior
- Cambia algún parámetro del grafo con el paso del tiempo
- Clase [Alpha](#)
  - ▶ Un objeto Alpha produce un valor entre 0.0 y 1.0 (inclusive)
  - ▶ El valor cambia con el tiempo según la configuración del objeto
- Clase [Interpolator](#)
  - ▶ Usa un objeto Alpha para realizar animación
  - ▶ Puede cambiar la posición, orientación, tamaño, color, o transparencia de una figura

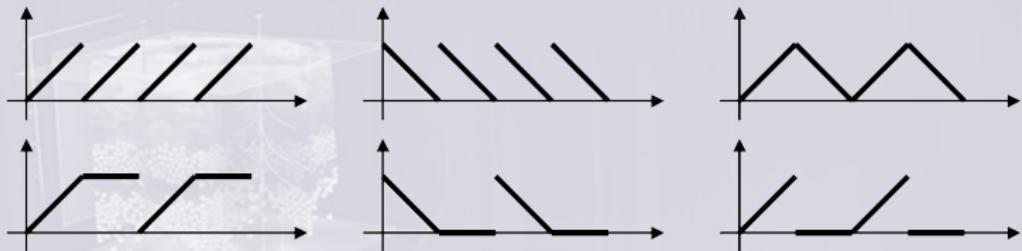
# Clase Alpha

- El **ciclo** del valor alpha tiene 4 fases:
  - Incremento, alpha en 1.0, decreto, alpha en 0.0
  - La duración de cada fase se especifica en milisegundos



# Clase Alpha

- No es obligatorio usar las 4 fases

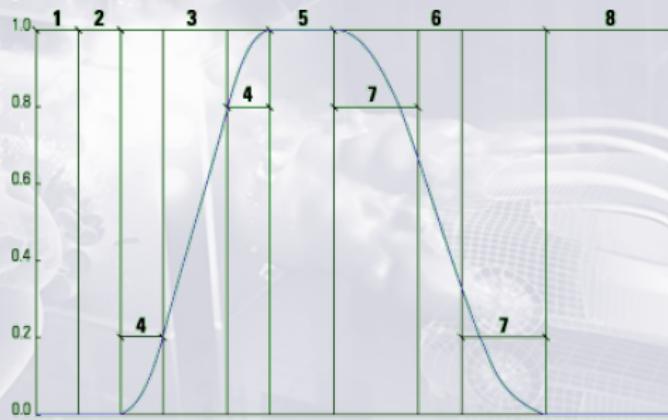


- Definiendo una duración de 0 en una fase deja de usarse
- Es más eficiente especificar un modo:
  - ▶ `Alpha.INCREASING_ENABLE`, habilita las fases 1 y 2
  - ▶ `Alpha.DECREASING_ENABLE`, habilita las fases 3 y 4
  - ▶ El OR de los dos anteriores

# Clase Alpha

- Constructor: `Alpha (`

- ▶ `int loopCount`, Repeticiones, -1 se repite indefinidamente
  - ▶ `int mode`, INCREASING\_ENABLE, DECREASING\_ENABLE o ambos
  - ▶ `long triggerTime`, `long delay`,
  - ▶ `long increasing`, `long increasingRamp`, `long alphaAtOne`,
  - ▶ `long decreasing`, `long decreasingRamp`, `long alphaAtZero`)



# Clase Interpolator

- Clase abstracta, se basa en un objeto Alpha
- Sus subclases permiten cambiar atributos en las figuras
- Subclases:
  - ▶ **SwitchValueInterpolator**  
Cambia una figura por otra, actúa sobre un grupo Switch
  - ▶ **ColorInterpolator**  
Cambia el color difuso, actúa sobre un Material
  - ▶ **ScaleInterpolator, PathInterpolator**  
Cambia la posición, orientación y tamaño de una figura, actúa sobre un TransformGroup

# Clase SwitchValueInterpolator

- Cambia entre las distintos *hijos* de un grupo `Switch`
  - ▶ Todos los hijos deben estar *colgados* antes de crear el interpolador
  - ▶ Si cambia el número de hijos después,  
el interpolador debe actualizarse
- Constructor:
  - ▶ `SwitchValueInterpolator (Alpha alpha,  
Switch grupoSwitch)`
- Métodos:
  - ▶ `void setFirstChildIndex (int indice)`
  - ▶ `void setLastChildIndex (int indice)`
  - ▶ `void setTarget (Switch grupoSwitch)`

# Clase ColorInterpolator

- Cambia una componente de un Material:  
Ambiental, Emisiva, Difusa, o Especular
- Requisitos:
  - ▶ La componente se indica con el método de Material  
`void setColorTarget (int colorTarget)`
    - ★ `Material.AMBIENT`
    - ★ `Material.AMBIENT_AND_DIFFUSE`
    - ★ `Material.DIFFUSE`
    - ★ `Material.EMISSIVE`
    - ★ `Material.SPECULAR`
  - ▶ El material debe tener las siguientes capabilities
    - ★ `Material.ALLOW_COMPONENT_READ`
    - ★ `Material.ALLOW_COMPONENT_WRITE`

# Clase ColorInterpolator

- Constructores:

- ▶ `ColorInterpolator (Alpha alpha, Material material)`
- ▶ `ColorInterpolator (Alpha alpha, Material material,  
Color3f startColor, Color3f endColor)`

- Métodos:

- ▶ `void setColorTarget (int colorTarget)`
- ▶ `void setStartColor (Color3f color)`
- ▶ `void setEndColor (Color3f color)`

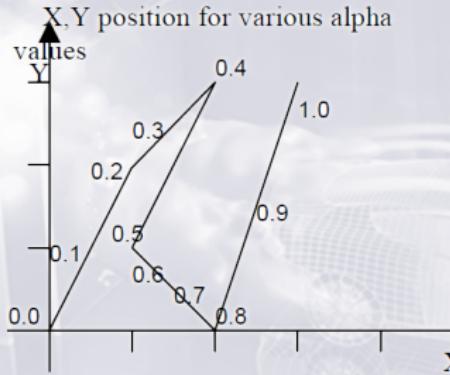
# Clase ScaleInterpolator

- Cambia el escalado de una figura uniformemente
  - ▶ El resto de transformaciones como traslaciones o rotaciones son sobreescritas a valores por defecto
- Constructor:
  - ▶ `ScaleInterpolator (Alpha alpha,  
TransformGroup transform)`
- Métodos:
  - ▶ `void setMinimumScale (float scale)`
  - ▶ `void setMaximumScale (float scale)`

# Clase abstracta PathInterpolator

- Actuando sobre un TransformGroup puede cambiar la posición, la rotación y el escalado
- Almacena un camino mediante vectores de valores (posiciones, orientaciones, tamaños)
  - La primera posición de los vectores se asocia al valor 0.0 de alpha
  - La última al valor 1.0 y las intermedias deben ir en orden creciente

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)



# Clase abstracta PathInterpolator

## Subclase PositionPathInterpolator

- Hay 1 vector que almacena posiciones
- Constructor

- ▶ `PositionPathInterpolator (Alpha alpha,  
TransformGroup transformGroup,  
Transform3D transform,  
float[] alphas,  
Point3f[] positions)`

### Ejemplo: Recorrido triangular

```
float[] alphas = {0.0f, 0.33f, 0.67f, 1.0f};  
Point3f[] positions = {  
    new Point3f (0.0f, 0.0f, 0.0f), new Point3f (1.0f, 1.0f, 0.0f),  
    new Point3f (2.0f, 0.0f, 0.0f), new Point3f (0.0f, 0.0f, 0.0f)  
};
```

# Clase abstracta PathInterpolator

## Subclase RotationPathInterpolator

- Hay 1 vector que almacena rotaciones, representadas como Quaterniones
- Constructor
  - ▶ `RotationPathInterpolator (Alpha alpha,  
TransformGroup transformGroup,  
Transform3D transform,  
float[] alphas,  
Quat4f[] rotations)`

### Ejemplo: RotationPathInterpolator con Quaterniones

```
float[] alphas = {0.0f, 0.33f, 0.67f, 1.0f};  
Quat4f[] rotations = new Quat4f[4];  
for (int i = 0; i<4; i++) rotations[i] = new Quat4f ();  
rotations[0].set (new AxisAngle4f (0.0f, 0.0f, 1.0f,  
(float) Math.toRadians (60));  
....
```

# Clase abstracta PathInterpolator

Subclase **RotPosPathInterpolator**

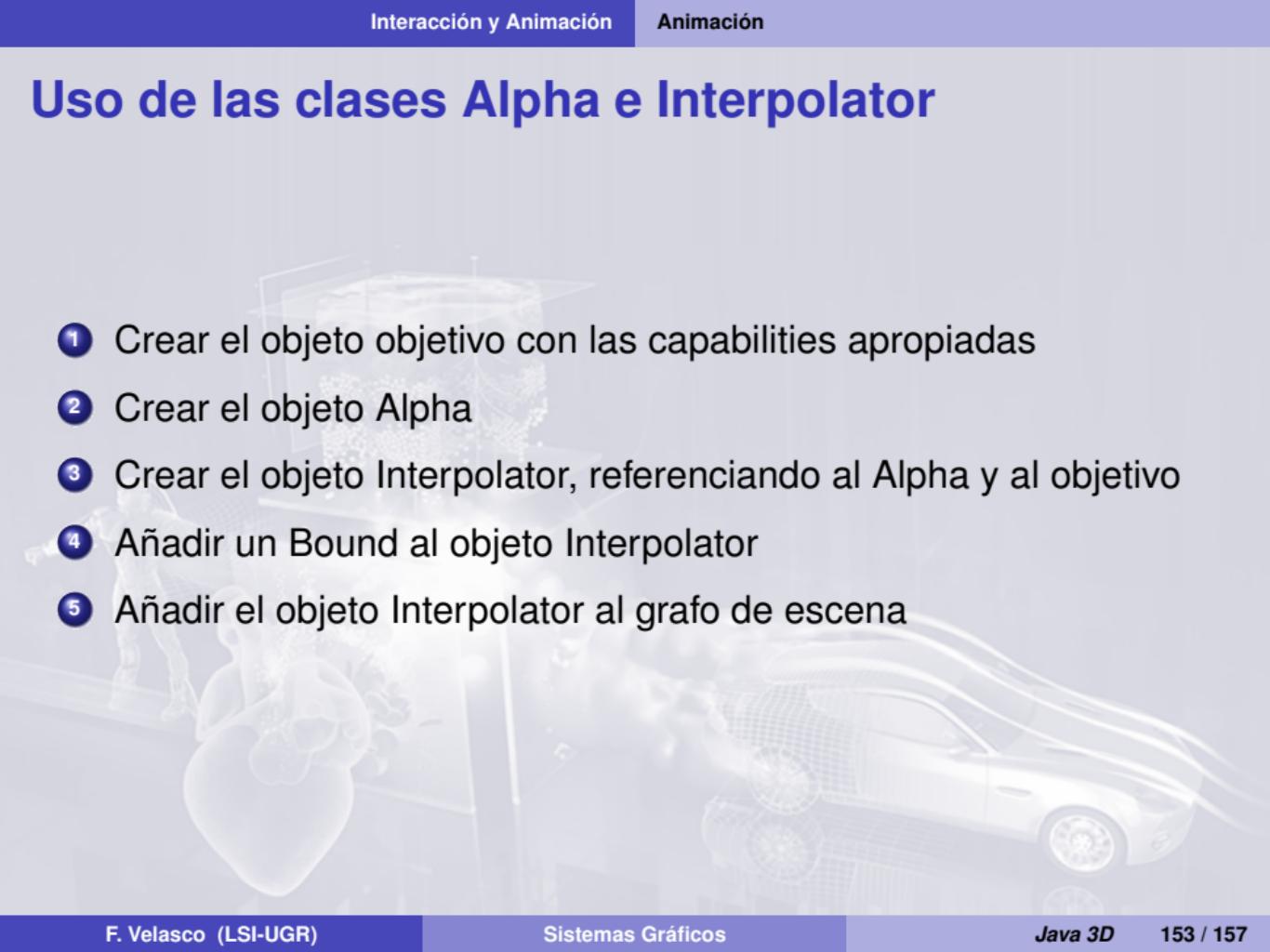
- Se combina un cambio de posición y de orientación
- Hay 2 vectores,  
se almacenan rotaciones como Quaterniones y posiciones
- Constructor
  - ▶ `RotPosPathInterpolator (Alpha alpha,  
TransformGroup transformGroup,  
Transform3D transform,  
float[] alphas,  
Quat4f[] rotations  
Point3f[] positions)`

# Clase abstracta PathInterpolator

Subclase **RotPosScalePathInterpolator**

- Se combina un cambio de posición, de orientación y de tamaño
- Hay 3 vectores, se almacenan rotaciones como Quaterniones, posiciones y tamaños
- Constructor
  - ▶ `RotPosScalePathInterpolator (Alpha alpha,  
TransformGroup transformGroup,  
Transform3D transform,  
float[] alphas,  
Quat4f[] rotations  
Point3f[] positions  
float[] scales)`

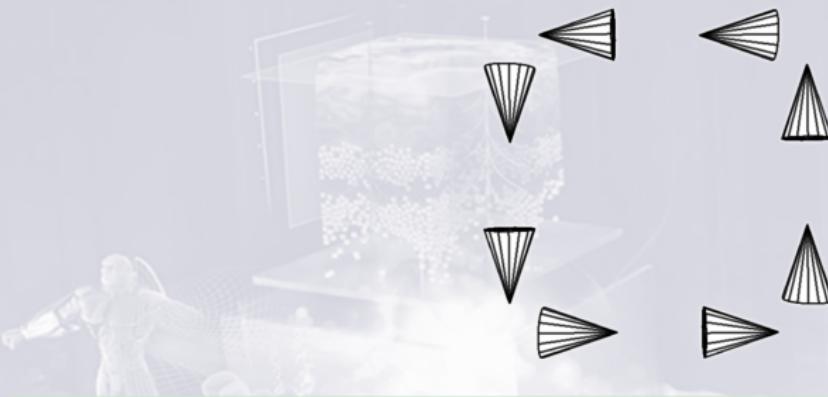
# Uso de las clases Alpha e Interpolator

- 
- A faint background image featuring a 3D wireframe model of a human heart on the left and a sleek sports car on the right, both set against a dark, atmospheric background.
- ① Crear el objeto objetivo con las capabilities apropiadas
  - ② Crear el objeto Alpha
  - ③ Crear el objeto Interpolator, referenciando al Alpha y al objetivo
  - ④ Añadir un Bound al objeto Interpolator
  - ⑤ Añadir el objeto Interpolator al grafo de escena

# Uso de las clases Alpha e Interpolator

## Ejemplo

- Moviendo un *vehículo*



## Ejemplo: Animación con RotPosPathInterpolator (1)

```
// El vehículo cuelga de un TransformGroup  
// que controla su posición y orientación  
Transform3D transf = new Transform3D ();  
TransformGroup tg = new TransformGroup (transf);  
tg.setCapability (TransformGroup.ALLOW_TRANSFORM_READ);  
tg.setCapability (TransformGroup.ALLOW_TRANSFORM_WRITE);
```

# Uso de las clases Alpha e Interpolator

## Ejemplo

### Ejemplo: Animación con RotPosPathInterpolator (2)

```
// Objeto Alpha. Se emplean 8 segundos en el recorrido completo
Alpha alfa = new Alpha (-1, 8000);

// Datos para el interpolador
// Vector de alfas. Se emplea un 20% del tiempo en las rectas
//      y un 5% en las curvas
float[] alfas = {0.0f,0.2f,0.25f,0.45f,0.5f,0.7f,0.75f,0.95f,1.0f};

// Vector de posiciones
Point3f[] posiciones = {
    new Point3f( 0.6f, -0.4f, 0.0f), new Point3f( 0.6f, 0.4f, 0.0f),
    new Point3f( 0.4f, 0.6f, 0.0f), new Point3f(-0.4f, 0.6f, 0.0f),
    new Point3f(-0.6f, 0.4f, 0.0f), new Point3f(-0.6f, -0.4f, 0.0f),
    new Point3f(-0.4f, -0.6f, 0.0f), new Point3f( 0.4f, -0.6f, 0.0f),
    new Point3f( 0.6f, -0.4f, 0.0f)
};
```

# Uso de las clases Alpha e Interpolator

## Ejemplo

### Ejemplo: Animación con RotPosPathInterpolator (y 3)

```
// Vector de orientaciones
Quat4f[] orientaciones = new Quat4f[9];
for (int i = 0; i < 9; i++)
    orientaciones[i] = new Quat4f();
orientaciones[0].set (new AxisAngle4f (0f,0f,1f,0f));
orientaciones[1].set (new AxisAngle4f (0f,0f,1f,0f));
orientaciones[2].set (new AxisAngle4f (0f,0f,1f,(float) Math.PI/2));
orientaciones[3].set (new AxisAngle4f (0f,0f,1f,(float) Math.PI/2));
.
.
.
orientaciones[8].set (new AxisAngle4f (0f,0f,1f,0f));

// Interpolador
RotPosPathInterpolator interpolador = new RotPosPathInterpolator (
    alfa, tg, transf, alfas, orientaciones, posiciones);
interpolador.setSchedulingBounds (
    new BoundingSphere (new Point3d(), 2.0f));

// Grafo
tg.addChild(interpolador);
tg.addChild(cone);
```

# **Java 3D**

Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Sistemas Gráficos

Grado en Ingeniería Informática  
Curso 2016-2017