Tutorial para principiantes de Java 3D

Este tutorial está dirigido a programadores de Java y que cuenten con experiencia previa en graficación, por lo que se recomienda estarfamiliarizado con los siguientes términos: pixel, imagen plana, RGB y dibujo (render) ya que son las bases para continuar con este tutorial. No es necesario conocer los conceptos de z-buffer, transformaciones en 3D u otros de graficas 3D pero estos pueden ser útiles.

**Requerimientos:**

Consultar la página de Java3D

**Capítulo1**

**Objetivos**

Después de leer este capítulo podra hacer lo siguiente:

* Será capáz de explicar los conceptos generales de Java 3D.
* Podrá describir la estructura básica de un programa en Java 3D.
* Reconocerá las distintas clases del Java 3D API.
* Escribirá pequeños programas con movimiento en Java 3D.

El Java 3D API es una interface para escribir programas que desplieguen gráficas tridimensionales además de interactuar con ellas. Java 3D es una extensión del Java 2 JDK. El API contiene un conjunto de contructores de alto nivel para crear y manipular geometrías en 3D, así como de las técnicas necesarias para dibujar la geometría en una imagen. Java 3D contiene las funciones necesarias para crear imágenes, visualizaciones, animaciones y aplicaciones con objetos gráficos interactivos.

**¿Que es Java 3D?**

El Java 3D API es un conjunto de clases que sirven como interface para dibujar gráficas sofisticadas en tercera dimensión utilizando sonido, si se deseea.

El programador puede utilizar constructores de alto nivel para crear y manipular las geometrías de los objetos tridimensionales. Estos objetos geométricos se localizan dentro de un universo virtual que sirve para dibujar los objetos en una imagen. El diseñó de este API se puede adecuar para que se puedan crear espacios virtuales con una gran variedad de tamaños por lo que se pueden definir dimensiones galácticas o subatómicas.

Este API realiza los detalles de dibujo (render) de una forma automática, además de que toma ventaja del procesamiento en paralelo, por eso el dibujo en Java 3D se puede optimizar. Otra posibilidad es la de optimizar automáticamente el despliege de los objetos que forman a la escena.

Un programa en Java 3D crea instancias de los objetos que componen a la escena, utilizando una estructura conocida como Grafo de Escena. El grafo de escena es el método que se utiliza para ordenan los objetos que forman parte de un escenario, y se representa por medio una estructura de árbol en donde se muestra cual es el contenido del universo virtual y cómo será dibujado.

Un programa escrito en Java 3D se puede ejecutar como una aplicación o un *applet,* siempre y cuando se tenga un navegador capaz de desplegar programas en Java 3D(1).

**El Java 3D API**

Todo programa en Java 3D es un conjunto de objetos ordenados en base a una serie de jerarquías basada en la herencia. Esta colección de objetos describen a un espacio virtual que es la guía para dibujarlos en una imagen posteriormente. Este API contiene aproximadamente 100 clases contenidas dentro del paquete *javax.media.j3d*. A estas clases por lo general se les conoce como el nucleo (core) de Java 3D.

Aunque existen cientos de variables y métodos dentro de las clases de Java 3D, en el universo virtual se pueden crear animaciones utilizando unas pocas clases, que es lo que se enseñará en esta sección.

El programa que se hará, es un ejemplo muy simple en Java 3D pero muy completo llamado HolaJava3D que despliega un cubo en rotación. Este programa se construirá de tal manera que poco a poco se va a ir incrementando su complejidad, por lo que se tienen varias versiones, con el fín de mostrar cada una de sus partes.

Además de las clases de Java 3D se utilizarán las clases del paquete *com.sun.j3d.utils* conocidas como las utilerías de Java 3D, ya que en estas se encuentran las clases de más bajo nivel, por lo que son necesarias para cualquier programa en Java 3D.

Estas utilerías tienen cuatro categorias: contenedores, contructores de escena, clases de geometrías y utilerías. En un futuro se incorporarán NURBS. Estas utilerías reducen significativamente el número de líneas de código, ya que además se hace uso de las clases contenidas en el j*ava.awt* y j*avax.vecmath.* El paquete j*ava.awt* define el Abstarct Windowing Toolkit (AWT) que sirve para crear la ventana en donde se despliega la imagen. El paquete j*avax.vecmath* define una clase matemática para realizar operaciones con puntos, vectores, matrices y otras funciones matemáticas.

El término de objeto visual se utiliza para referir a un objeto en el grafo de la escena (un cubo o una esfera por ejemplo). Por el contrario, el término objeto se utiliza únicamente para cuando se hacen instacias de una clase. El término contenedor se utiliza para aquellos objetos visuales que pertenecen a un grafo de escena.

(1) Para obtener un navegador que sea capaz de desplegar Java 3D, se encuentra en **java.sun.com**

**Construir una grafica de escena**

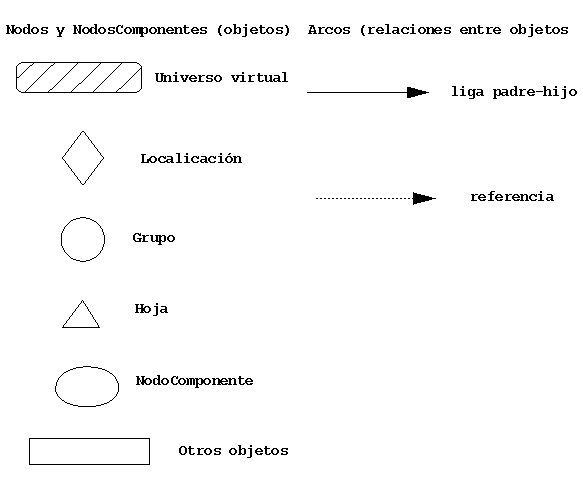
El universo virtual se crea utilizando una grafo de escena. Esta grafo de escena se construlle gracias a la correcta utilización de las clases de Java 3D, ya que se van incorporando poco a poco las geometrías, los sonidos, las luces, la ubicación , la orientación y la apariencia visual asi como el sonido de los objetos. Una definición básica de un grafo de escena, es que es una estructúra de datos, representada por medio de nodos y arcos. Un nodo es un dato y un arco es la relación que existe entre los datos. Los nodos en un grafo de escena define a las instancias que se utilizan en las clases de Java3D. Los arcos representan a las dos posibles relaciones que existen entre las instancias de Java 3D.

La relación más común es la de padre-hijo. Un nodo grupo puede tener una gran cantidad de hijos pero únicamente un padre. Un nodo hoja puede tener un padre pero ningun hijo. La otra relación es la referencia, esta asocia a un objeto NodoComponte con un Nodo de la gráfica de escena. Un objeto NodoComponente define los atributos de la geometría así como los atributos utilizados para dibujar a los objetos visuales.

Un grafo de escena se va construyendo como un grafo acíclico-directo (directed-acyclic grap, DAG) compuesto de nodos y arcos. En un grafo directo los arcos tienen una sola dirección, por lo que un grafo aciclico-directo tiene un solo sentido por lo que los ciclos no estan permitidos. Se comienza con un nodo y la dirección del grafo no puede regresar al mismo nodo.

Existe una sola ruta desde la raiz del DAG hacia cada una de las hojas, a esta ruta se le conoce con el nombre de ruta del grafo de escena. El significado de cada ruta radica en que indica el estado en que se encuentra cada elemento de la escena, representado por medio de una hoja, este estado se refiere a ubicación, orientación y tamaño de cada objeto. Cuando se plasman los objetos en la imagen Java 3D puede tomar ventaja de esta jerarquia, lo que permite que el desplegue de los objetos sea más ordenado. Por eso hay que señalar, que el programador no tiene control sobre el orden en que se dibujarán los objetos . (2)

La representación gráfica de el grafo de escena sirve para diseñar tanto el programa cómo la documentación, un ejemplo de esto es la figura 1.1 en donde se muestran los símbolos que se utilizan para contruir cualquier grafo de escena.

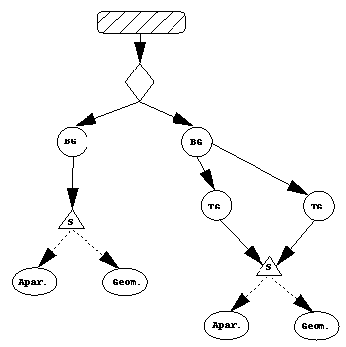


**Figura1.1 Símbolos que reperesentan los objetos de un grafo de escena**

Cada uno de los símbolos que estan a la izquierda de la figura 1.1 representan a un objeto que será usado en el grafo de escena. Los dos primeros símbolos representan a dos clases muy específicas: VirtualUniverse y Locale. Los tres siguientes son los que representan a las clases Group, Leaf y NodeComponent utilizados para indicar las subclases de un objeto determinado. El último se para representar a cualquier otra clase de objeto.

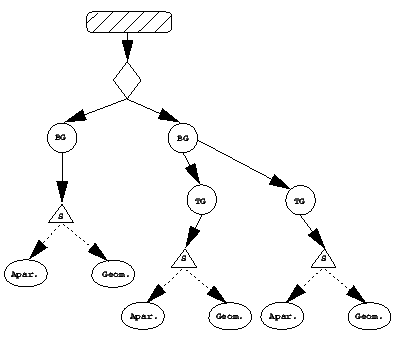
La flecha sólida representa la relación padre-hijo entre dos objetos, la flecha punteada indica la referencia con otro objeto. Pero con estas referencias en muy fácil construir una grafica de escena ilegal, el único peligro que se corre, es que el programa puede compilar pero no dibujar nada. Cuando un programa en Java 3D define una escena ilegal, el sistema mismo detecta el problema reportando una exepción, pero el programa puede sigue corriendo pero sin desplegar nada, por lo que es necesario detenerlo.

En la figura1.2 se muestra el grafo de escena ilegal, ya que en cada grafo de escena existe un solo Universo Virtual que a su vez tiene una lista de objetos del tipo Locale que a su vez hacen referencias a los objetos que estan en ese Universo Virtual, volviendo al ejemplo de la figura 1.2 el error consiste en que un objeto 3D únicamente puede tener un solo padre y no dos como es este caso.



**Figura 1.2 Ejemplo de un Grafo de Escena Ilegal**

En la figura 1.3 muesta una posible solución.



**Figura 1.3 Correción del Grafo de Escena de la figura 1.2**

***Clasificación de la herencia en las clases alto nivel***

Podemos hacer un resumen de los tres niveles del árbol de clases que aparece en la figura 1.4. Las clases VirtualUniverse, Locale, Group, y Leaf aparecen en un mismo nivel, el resto del grafo de escena está compuesto de objetos SceneGraphObjects. Esta clase SceneGraphObjects es la superclase que contiene a todas las clases del tipo Core y Utility.

De SceneGraphObject derivan dos clases más Node y NodeComponent, la primera contiene la mayor parte de los objetos que componen al grafo de escena, un objeto Node puede ser tanto un nodo Group o un objeto Leaf.

***Clase Node***

Es una superclase abstracta que contiene a las clases Group y Leaf. La clase Node define a algunos de los métodos más importantes así como a sus clases y son precisamente estas las que componen al grafo de escena.

***Clase Group***

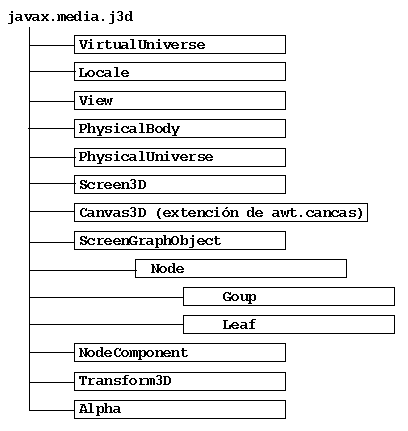
Es una superclase que se utiliza para indicar tanto la ubicación como la orientación de los objetos visuales en el espacio virtual. Dos de las clases que le pertenecen son la BranchGroup y la TransformGroup. En la representación gráfica del grafo de escena el circulo representa a la clase Group y tiene las letras BG como siglas.

***Clase Leaf***

Es una superclase utilizada para indicar tanto la forma, el sonido y el comportamiento de los objetos visuales en el universo virtual. Algunas de las subclases de Leaf son Shape3D, Light, Behavior y Sound. Estos objetos no pueden tener hijos pero pueden referenciar a otros NodeComponents.

***Clase NodeComponent***

Es una superclase que especifica las geometrías, la apariencia, la textura y las propiedades de los materiales que se utilizarán en el nodo Shape3D. NodeComponents no forma parte del grafo de escena pero es referenciado por el. Un objeto NodeComponent puede referenciarse por más de un objeto Shape3D.



**Figura 1.4 Resumen de la clasificación de la herencia en las clases de alto nivel de Java 3D**

***Pasos para escribir programas en Java 3D***

Las subclases de SceneGraphObject son bloques que se van conjuntando dentro del grafo de escena. Los siete pasos básicos para desarrollar un programa en Java 3D son:

1. Crear un objeto Canvas3D

2. Crear un objeto VirtualUniverse

3. Crear un objeto Locale y agregarlo al objeto VirtualUniverse

4 . Construir un grafo de escena

* Crear un objeto View
* Crear un objeto ViewPlatform
* Crear un objeto PhysicalBody
* Crear un objeto PhysicalEnvironment object
* Añadir los objetos ViewPlatform, PhysicalBody, PhysicalEnvironment y Canavas3D al objeto View.

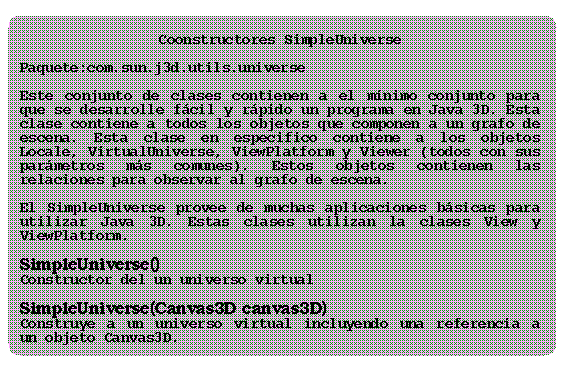
5. Construir el contenido de una rama(s) grupo.

6. Compilar la(s) rama(s) grupo.

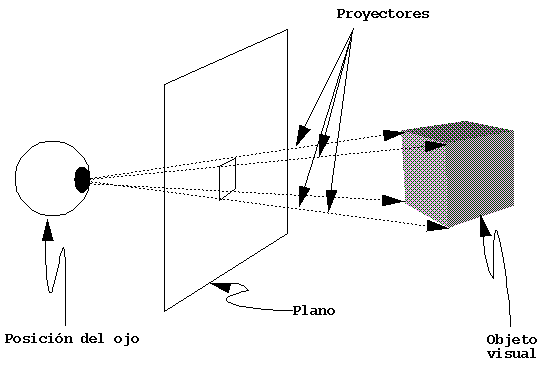
7. Insertar las subgraficas denro de Locale

Estos pasos ignoran algunos detalles pero se ilustra el concepto general que todo programa en Java 3D debe incluir. Ya que si se siguen estos pasos, la ventaja que se tien, es que en un futuro la estructura de los programas será muy similar además de que al utilizar la clase VirtualUniverse reduce significativamente el tiempo y el esfuerzo que se emplea para programar sin embargo no permite observar varias vistas del universo virtual.

Las cajas grises que apartir de este momento se incluirán contienen segmentos de definiciones en donde se hará una breve descripción de los constructores, métodos o parámetros de una clase, esto con el fín de que el lector pueda conocer algunas de las bases de la programación del Java 3D API, hay que aclarar, que no se cubren la totalidad de los metodos y constructores de cada clase, por lo que se recomienda leer tambien el The Java 3D API Specification.

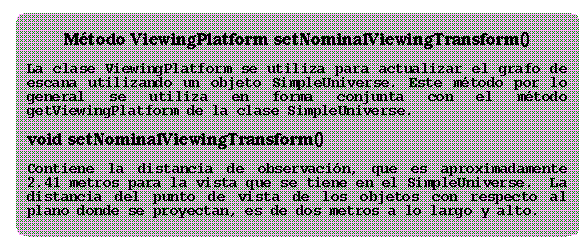


El objeto SimpleUniverse crea una vista completa del grafo de escena. Esta vista del grafo de escena incluye a una plano, que consiste en un rectangulo en donde se proyecta el contenido en forma de imagen. El objeto Canvas3D es el vínculo para que dicha imagen se dibuje en una ventana de la pantalla de la computadora. En la figura 1.6 se muestra la relación que tiene el plano con la posición del ojo (virtual) y el universo virtual. la posición del ojo se encuentra detraz del plano y los objetos en frente de este se van dibujando en el. El dibujo se puede hacer proyectando los objetos en el plano, para ilustrar esta idea se en la figura 1.6 se muestran cuatro proyectores. Por convenienca el plano se coloca en el centro del universo virtual, la orientación que se tiene, es la siguiente; el eje de las positivo de la Z esta orientado en dirección a el usuario (es la profundidad), el eje de las X es horizontal y sus valores positivos estan a la derecha; el eje de las Y es vertical al centro del plano y sus valores positivos estan orientados hacia arriba, por consecuencia el punto (0,0,0) es el centro del plano.

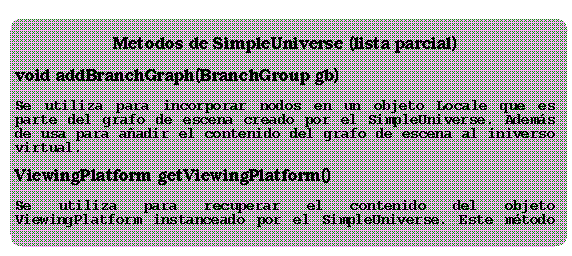


***Figura 1.6 Ejemplo conceptual del dibujo del contenido del universo virtual en el Plano con respecto a la posición del ojo.***

Un programa en java 3D mueve la parte posterior del punto de vista con respecto a la ubicación del origen. El objeto SimpleUniverse pertenece a la clase ViewingPlatform, la que a su vez contiene al método setNominalViewingTransform que es el que define la posición del ojo, por conveniencia esta corresponde a (0,0,2.41) y se encuentra observando en dirección al eje negativo de las Z, partiendo del origen.



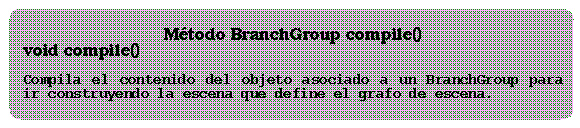
Despues de crear los objetos Canvas3D y SimpleUniverse, el siguiente paso es la creación del contenido del grafo de escena. Tanto su estructura cómo su contenido por lo general varia de un programa a otro, lo que significa que no se puede crear una clase que defina el contenido de cualquier universo virtual.

Despues de crear el contenido de un grafo de escena, se incorpora en el universo utilizando el método addBranchGraph contenido en el SimpleUniverse, el método addBranchGraph hace una instancia a BranchGroup, que a su vez se añade como hijo utilizando un objeto Locale creado por el SimpleUniverse.

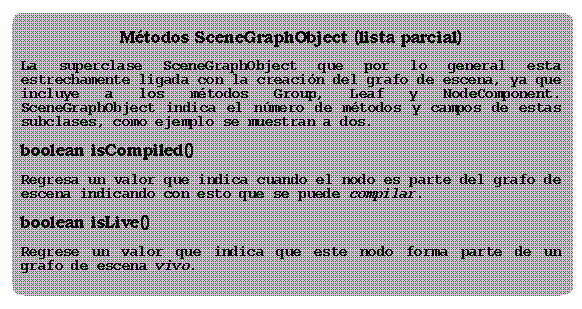
**Algunos conceptos de java 3D**

Antes de entrar de lleno a la creación de un grafo de escena, es necesario establecer dos conceptos, estos son *vivo* y *compilado*. En el momento que se inserta un grafo de escena en Locale, se dice que esta *vivo* y en consecuencia, cada uno de los objetos que esta en el grafo de escena *vuelve a la vida*, esto nos permite realizar las siguientes acciones. Los *objetos vivos* pueden dibujarse inmediatamente, sin embargo sus parámetros no se pueden alterar, ya que pierden la capacidad de modificar su "comportamiento".

Por otro lado, los objetos del BranchGroup se pueden *compilar*, esto significa que Java 3D ordena el contenido del objeto BrancGroup de tal manera que se establece todos y cada uno de sus ancestros haciendo más eficiente el dibujo de la escena. Se recomienda mucho utilizar esta técnica cómo un paso previo al de dibujarlos. Es mejor compilar únicamente los objetos del tipo BranchGroup e insertarlos posteriormente en los de tipoLocale.



Los conceptos *compilado* y *vivo* vienen incluidos en la clase SceneGraphObject, por lo que a continuación se describen brevemente en el siguiente cuadro gris.



Hay que hacer la observación, de que esto no "comienza el dibujo" de la escena. El dibujo en Java 3D se puede definir como un cíclo infinito que va dibujando el contenido del grafo de escena, esto se realiza en el momento que se tiene una instancia del tipo View. Una vez que se ha comenzado el dibujo en Java 3D se siguen las siguientes operaciones, todo esto se debe tomar como concepto.

               mientras(sea\_verdad)

               {

                             Proceso de entrada

                             SI(petición de salida) romper

                                              Ejecutar comportamiento (animación)

                                              Recorrer el grafo de escena

                                              y dibujar los objetos

              }

              Limpiar y salir

***Proceso conceptual de dibujar una escena en Java 3D***

**Ejemplo de un programa en Java 3D**

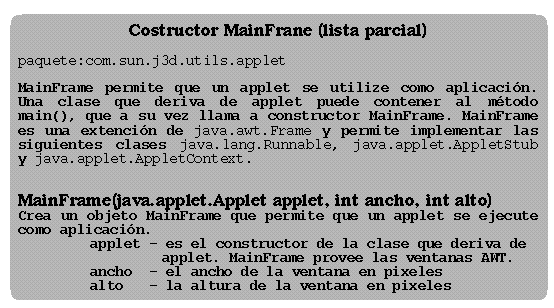
Un programa en Java 3D por lo general comienza definiendo una clase que se extende de Applet, para ilustrar esto se mostrará el código de un programa que hace uso de las clases de Java 3D, el nombre de la clase es HolaJava3Da, los programas en Java 3D se pueden escribir para que se ejecuten como aplicaciones, pero al utilizar la clase Applet, ayuda a que el despliegue en ventanas sea más fácil y rápido.

La clase principal en un programa de Java 3D define un método en donde ir construyendo el contenido del grafo de escena, en el ejemplo que aqui se proporciona este método se define y se llama crearGrafoDeEscena().

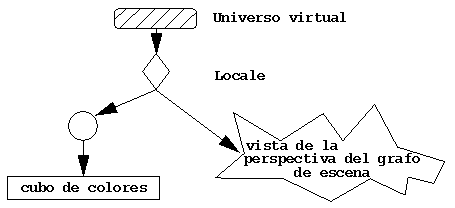
Los pasos que se siguen son los siguientes. Paso 1, crear un objeto Canvas3D, ver linea 16 del código. Paso 2, crear un objeto SimpleUniverse, ver linea 23 del código. Paso 2a, configurar el objeto SimpleUniverse, ver linea 27. Paso 3, construir el contenido de la rama del grafo de escena, esto se hace al llamar al método crearGrafoDeEscena(). Paso 4, compilar el contenido del grafo de escena, ver linea 20. Para terminar, el paso 5, es agregar el contenido del grafo de escena en el Locale perteneciente al SimpleUniverse, ver la linea 29.

1. **import java.applet.Applet;**
2. **import java.awt.BorderLayout;**
3. **import java.awt.Frame;**
4. **import java.awt.event.\*;**
5. **import com.sun.j3d.utils.applet.MainFrame;**
6. **import com.sun.j3d.utils.universe.\*;**
7. **import com.sun.j3d.utils.geometry.ColorCube;**
8. **import javax.media.j3d.\*;**
9. **import javax.vecmath.\*;**
10. **public class HolaJava3Da extends Applet {**
11. **public HolaJava3Da()**
12. **setLayout(new BorderLayout());**
13. **// Creacion del Canvas**
14. **Canvas3D canvas3D = new Canvas3D(null);**
15. **add("Center", canvas3D);**
16. **BranchGroup escena = crearGrafoDeEscena();**
17. **escena.compile();**
18. **// Creacion del universo**
19. **SimpleUniverse simpleU = new SimpleUniverse(canvas3D);**
20. **// Ubica la vista al frente de la escena para ver a todos los**
21. **// objetos**
22. **simpleU.getViewingPlatform().setNominalViewingTransform();**
23. **simpleU.addBranchGraph(escena);**
24. **} // fin del HolaJava3Da (constructor)**
25. **public BranchGroup crearGrafoDeEscena() {**
26. **// Crear la raiz**
27. **BranchGroup objRaiz = new BranchGroup();**
28. **objRaiz.addChild(new ColorCube(0.4));**
29. **return objRaiz;**
30. **} // fin del metodo**
31. **// Para correr el programa como un applet**
32. **public static void main(String[] args) {**
33. **Frame frame = new MainFrame(new HolaJava3Da(), 256, 256);**
34. **} // fin del main**
35. **} // fin de la clase HolaJava3Da**

Es en el paso 3 en donde propiamente se crea el contenido del grafo de escena, pero para examinar a más detalle cuales son sus partes, hay que examinar las lineas 32 a 39 que muestran como definir a un objeto gráfico estático con nombre ColorCube. Este cubo de color se ubica en el origen del sistema de coordenadas del universo virtual. Con esta ubicación y orientación, en el momento de dibujarlo lo que se obtiene es simplemente un rectángulo.



Como se mencionó, la clase HolaJava3Da deriva de Applet, esto no impide que se pueda ejecutar como aplicación ya se se hace uso de la clase MainFrame, con esta clase se puede hacer uso de las ventanas del AWT que permiten a un applet ejecutarse como aplicación, tambien se puede indicar el tamaño de la ventana en el momento de construir la clase, ver lineas 42-48.



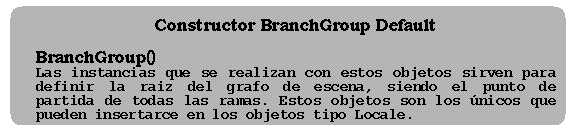
**Grafo de escena que describe a ejemplo HolaJava3Da**

***Descripción de las clases utilizadas.***

Para que se entienda cual es la función que tienen las clases que se utilizarón en el ejemplo HolaJava3Da, a continuación se describen algunas de estas.

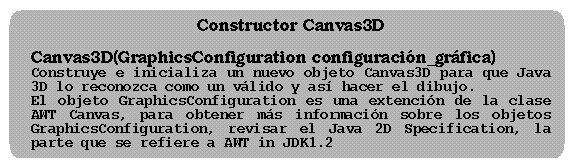
**Clase BranchGroup**

Los objetos definidos con esta clase se utilizan para definir a los grafos de escena. Las instancias de BranchGroup son las raices, apartir de donde se generan las demás subgraficas, es por esta razón que este tipo de objetos son los únicos que pueden ser hijos de los objetos Locale, aunque pueden tener cuantos hijos se desee. Estos hijos así mismo pueden ser objetos del tipo Group o Leaf.



**Clase Canvas3D**

Esta clase se deriva de la clase Canvas perteneciente al Abstract Windowing Toolkit(AWT). Un objeto Canvas3D se puede tomar como el punto de observación para ver la ramificación del grafo de escena.

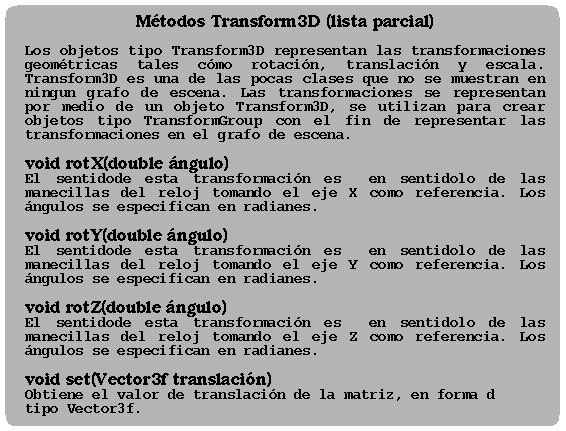


**Clase Transform3D**

Este típo de objetos representan las transformaciones que se aplican a los objetos geométricos en el espacio 3D, tales como, translación y rotación. Por lo general estos objetos solamente se utilizan para crear objetos de tipo TransformGroup. Un objeto Transform3D se puede construir al combinar varios objetos Transform3D. Entonces un objeto TransformGroup se puede contruir al utilizar más de un objeto Transform3D.

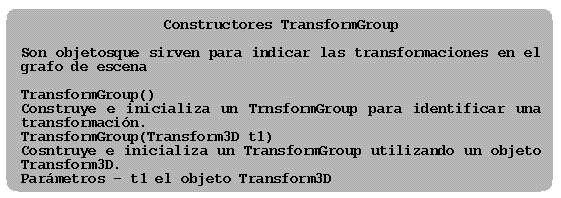


Los objetos Transform3D pueden combinar distintas transformaciones en un objeto, translación, rotación y escala. Cuando se trabaja la rotación, los ángulos se deben expresar en radianes; una rotación completa es 2¶, una manera fácil para definir los ángulos es utilizar la constante ***Math.PI,*** o bien escribir el valor directamente en radianes. Algunas aproximaciones basicas son: 45° es 0.785, 90° son 1.57 y 180° es 3.14.

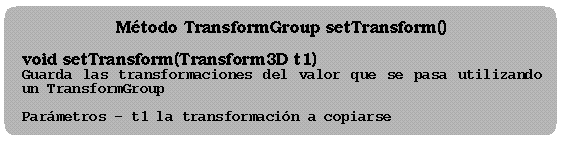


**Clase TransformGroup**

Es una subclase de Group,las instancias tipo TransformGroup se utilizan para la crecaión de los grafos de escena y tienen un conjunto de objetos del tipo nodo hijo. Los objetos TransformGroup tienen el control de las transformaciones geométricas. Una transformación se reliza por medio de un objeto Transfom3D.

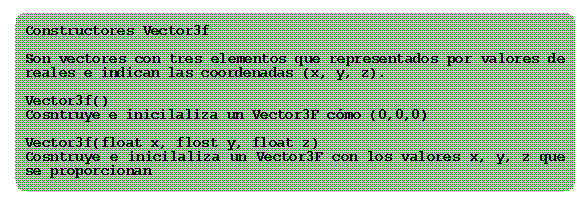


La transformación que se realizó se almacena en un objetos Transform3D se copia despues en un objeto TransformGroup en el instante que este se crea, o utilizando el método setTransform().



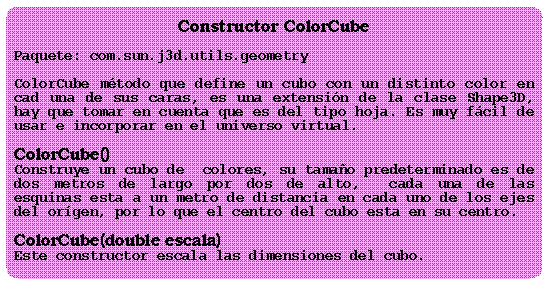
**Clase Vector3f**

Es una clase matemática del paquete *javax.vecmath* que sirve para crear en un vector de dimensión tres los valores que se utilizarán para indicar las translaciones de las geometrías. Los objetos Vector3f no se utilizan de forma directa para la construcción del grafo de escena. Unicamente se utilizan para indicar traslaciones, normales, etc.



**Clase Cubo de color**

Es una clase que se localiza en el paquete *com.sun.j3d.utils.geometry* y define tanto la geometría y los colores de cada una de las caras de un cubo ubicado en el orígen. El valor predeterminado que se utiliza, es de 2 metros tanto en altura como en profundidad. Si no se a rotado el cubo su ubicación es el origen, la cara roja se puede considerar el frente, otros colores que se pueden utilizar son azul, magenta, amarillo, verde y cyan.



***Rotar el cubo.***

Para realizar una simple rotación del cubo, en primer lugar se debe utilizar un objeto Transform3D para crear la transformación deseada.

El código que se encuentra en la página siguiente muestra como realizar dicha transformación. En primer lugar se incorpora un objeto tipo TransformGroup en el grafo de escena que rota el cubo en el eje X, para esto, se crea un objeto rotate , ver linea 17 del código, la rotación se indica utilizando el método rotX(), linea 20. El objeto TransformGroup que se encarga de guardar las transformaciones se crea en la linea 24.

Se especifican dos parámetros para la rotación: uno para el eje X y su respectivo ángulo de rotación. Para escoger el eje en que se va a aplicar la transformación , debe escogerse el método adecuado. El ángulo de rotación es el argumento que se proporciona cómo argumento de cada método, este ángulo se debe dar en radianes, por ejemplo ¶/4 es 1/8 o que es lo mismo, una rotación completa o 45°.

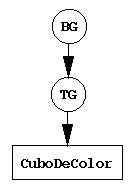
1. **import java.applet.Applet;**
2. **import java.awt.BorderLayout;**
3. **import java.awt.Frame;**
4. **import java.awt.event.\*;**
5. **import com.sun.j3d.utils.applet.MainFrame;**
6. **import com.sun.j3d.utils.geometry.ColorCube;**
7. **import com.sun.j3d.utils.universe.\*;**
8. **import javax.media.j3d.\*;**
9. **import javax.vecmath.\*;**
10. **public class HolaJava3Db extends Applet {**
11. **public BranchGroup createSceneGraph() {**
12. **//Crea la raiz**
13. **BranchGroup objRoot = new BranchGroup();**
14. **// rata el objeto con respecto a la matriz de transformacion**
15. **Transform3D rotate = new Transform3D();**
16. **Transform3D tempRotate = new Transform3D();**
17. **rotate.rotX(Math.PI/4.0d);**
18. **tempRotate.rotY(Math.PI/5.0d);**
19. **rotate.mul(tempRotate);**
20. **TransformGroup objRotate = new TransformGroup(rotate);**
21. **objRoot.addChild(objRotate);**
22. **objRotate.addChild(new ColorCube(0.4));**
23. **// Permite que jav3d realize optimizaciones de la escena**
24. **objRoot.compile();**
25. **return objRoot;**
26. **} // fin del metodo CreateSceneGraph**
27. **// Crea una escena y la pega el universo virtual**
28. **public HolaJava3Db() {**
29. **setLayout(new BorderLayout());**
30. **Canvas3D canvas3D = new Canvas3D(null);**
31. **add("Center", canvas3D);**
32. **BranchGroup scene = createSceneGraph();**
33. **// Crear el universo virtual**
34. **SimpleUniverse simpleU = new SimpleUniverse(canvas3D);**
35. **simpleU.getViewingPlatform().setNominalViewingTransform();**
36. **simpleU.addBranchGraph(scene);**
37. **} // fin de HolaJava3Db (constructor)**
38. **public static void main(String[] args) {**
39. **Frame frame = new MainFrame(new HolaJava3Db(), 256, 256);**
40. **} // fin del main**
41. **} // fin de la clase HolaJava3Db**

**Código fuente del programa HolaJava3d en donde se ilustra como rotar al cubo**

Despues de crear la transformación con el objeto Transform3D, **rotate**, ver linea 17, este a su vez sirve para la creación de un objeto TransformGroup, **objRotate**, ver linea 24. El objeto Transform3D se utiliza en el grafo de escena. en cambio el objeto objRotate hace del objeto ColorCube su hijo, ver linea 26, pero a su vez el objeto objRoot hace de objRotate su hijo, ver linea 27.

Los métodos Transform3D que se utilizan son rotX() y rotY(), para conocer más de estos, consultar el segmento que se tiene en la página15 de este tutorial. Hasta este momento es cuando un grafo de escena contiene un objeto TransformGroup, ubicado en la ruta del Cubo de colores. Cada uno de estos objetos son indispensables en el momento de recorrer el árbol que representa al grafo de escena. El objeto BranchGroup es el único que puede ser hijo de un Locale. El objeto TransformGroup tambien es el único que puede cambiar la ubicación, la orientación o el tamaño de un objeto visual, en este caso lo que se modifica el la orientación del Cubo de colores.

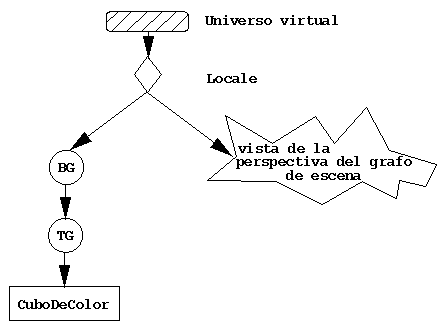
El grafo de escena que representa a el código de las lineas 11 a 32 es el siguiente:



**Combinación de Transformaciones**

Para transladar y rotar a un objeto se deben aplicar dos transformaciones en forma combinada, por lo que se necesita una matriz de transformación, por lo que que se utilizarán dos objetos del tipo Transform3D, en el ejemplo del programa Holajava3Db la rotación del cubo se realiza en los ejes x , y en forma individual, lineas 20-21, es por eso que se utilizan dos objetos Transform3D, lineas 17-18, entonces las transformaciones se combinan al multliplicar a ambos objetos, linea 22, por último el resultado se manda llamar por medio del objeto TransformGroup, linea 24.

El grafo de escena que representa al programa Holajava3Db, es el siguiente.



Como se puede observar este grafo de escena ya contiene a un TransformGroup en la ruta del Cubo de color.

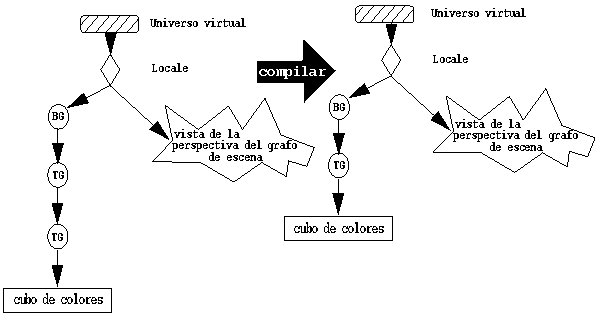
***Desempeño y propiedades***

El grafo de escena que utiliza un programa en Java 3D se puede utilizar para realizar el dibujo de la escena en forma inmediata, sin embargo esto no es muy eficiente, por lo que se necesita utilizar una organización eficiente para representar la organización del universo virtual, en Java 3D existen dos formas para lograrlo, la primera es compilar cada rama del grafo de escena. La otra es insertar directaente cada rama en el universo virtual, la que a continuación se comentará es la primera.

**Compilar el contenido**

El objeto BranchGroup tiene un metodo que realiza esta compilación, que en otras palabras es recorrer el contenido de cada rama del grafo de escena para ir incorporando los elementos a la escena, esto permite obtener un despliege más eficiente.

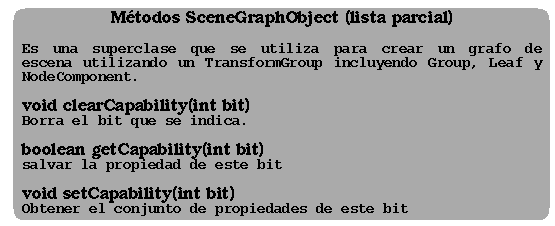
Una de las posibles optimizaciones puede ser; el combinar a lo largo del grafo de escena, los TransformGroups en uno solo, por ejemplo si se tienen dos objetos TransformGroup ubicados en una misma rama, estos dos se pueden representar como uno solo, en la siguiente figura se ilustra conceptualmete este proceso, en donde se ilustra como se modifica la estructura de un grafo de escena en el momento que se compila.



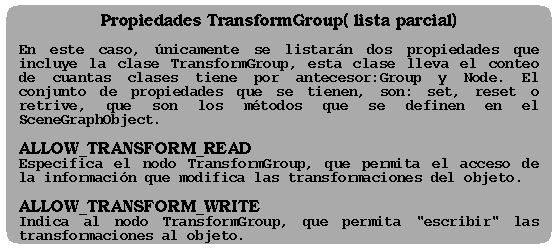
**Propiedades**

Una vez que una rama se vuelve "viva" o "compilada", el sistema de dibujo de java 3D convierte esta rama en una estructura interna más eficiente con lo que se acelera el dibujo, otra ventaja que se tiene, es que muchas de las transformaciones se realizan internamente , como por ejemplo, el combinar más de una transformación u objetos en un grafo de escena, lo que nos permite realizar las animaciones.

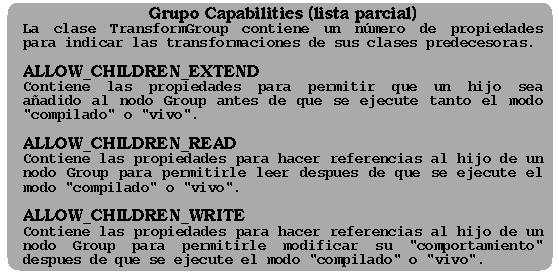
Existen ciertos casos, en donde se neceitan modificar algunos parámetros en el grafo de escena una vez que se han dibujado los objetos, y asi se pueden realizar animaciones muy facilmente, para que esto suceda, se cuenta conuna serie de parámetros y se a los que se les da el nombre de *capabilities.* Por lo tanto, un objeto SceneGraphObject tiene una serie de bits que determinan que capacidades estan permitidas en unobjeto, despues que se ha dibujado los objetos. A continuación se da una breve lista.



Para explicar mejor cules son los parámetros que nos permiten modificar el comportamiento de un objeto, el primero es leer los nuevos valores, contenidos en un objeto TransforGroup, para el caso de que ya se necesite modificar los valores de la transformación, se dice que se debe contar con la capacidad de escribir. Por lo que acontinuación se da una lista parcial de estas capacidades, y que posteriormente se utilizarán para implementar una animación.



Existen otras capabilities pero que pertenecen al la clase Group, por lo que a continuación se listan algunas de ellas.



***Animación***

Para hacer una animación en Java 3Dse debe utilizar la clase Behavior, que permite interactuar con los objetos visuales. Este "comportamiento" puede variar para cada objeto y el programador puede definir con antelación un número predeterminado de "comportamientos" (animaciones) y una vez que se especifica a que objeto se le va a aplicar, en forma automática, el propio Java 3D realiza las adecuadas transformaciones de ubicación, orientación, color y demás atributos.

Hay que señalar que la animación es distinta a la interacción, ya que la primera se ejecuta como respuesta al paso del tiempo, y la segunda esta condicionada a lo que el usuario haga.

Cada objeto visual que esta en el universo virtual puede tener su propio comportamiento, o por el contrario depender de más de uno. Para indicar estos "comportamientos", el programador debe de crear distintos objetos, para cada uno de estos, añadirlos al grafo de escena correspondiente e inmediatamente despues añadir el objeto visual para que establezcan las referencias apropiadas a cada objeto.

Hay que señalar que en caso de que un universo virtual contenga muchas animaciones, se necesita más poder de cómputo para realizar todos los cálculos, de otra manera se degradará el dibujo.

Para tratar de resolver este problema, Java 3D permite crear un límite espacial para indicar en donde dicho comportamiento se llevará a cabo. A este límite se le conoce como región de acción. Este comportamiento no se lleva a cabo o se activa hasta que un *ViewPlarform activation volume* intersecta l región que define un objeto tipo Behavior, estas regiones permiten a Java 3D ser más eficiente en el momento de que el universo virtual contiene muchas animaciones.

Pero para realizar una animación, es necesario contar con un interpolador, ya que se necesita una función que dependa del tiempo, en base a esto, se cuenta con un objeto tipo Interpolator que maneja este parámetro en el universo virtual, por ejemplo, el objeto RotationInterpolator manipula la rotación de un objeto visual, este efecto es el producto de la herencia de un TransformGroup.

A continuación se enumeran los pasos que se deben seguir para realizar una animación, utilizando un objeto tipo interpolador.

1. Crear un TransformGroup

* Ver ALLOW\_TRANSFORM\_WRITE

2. Crear un objeto Alpha

* Indica los parámetros del tiempo, que sirven para variar el tiempo.

3. Crear un objeto interpolador

* Este tiene cómo refencia a los objetos Alpha y TransforGroup
* Se encarga de ajustar los parámetros de la animación

4. Especificar la región de acción

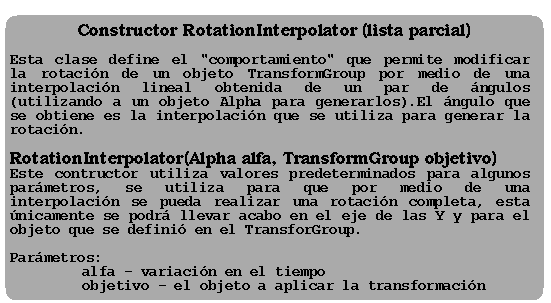
5. Se enparenta objeto tipo behavior como hijo del TransformGroup

***Comenzar la animación***

Para indicar la acciónes que se llevarán a acabo para cambiar la ubicación (PositionInterpolator), la orientació (RotationInterpolator), el tamaño (ScaleInterpolator), el color (ColorInterpolator) o la transparencia (TransparencyInterpolator) de los objetos y cada uno de estos se necesita un interpolador ya que es mucho más fácil de controlar e incluso combinar, pero en este tutorial solo se hará uso de uno, el RotationInterpolator.

**Clase RotationInterpolator**

Esta clase se utiliza para especificar la rotación de un objeto visual o un grupo de estos. Este tipo de objetos cambia el comportamniento de un objeto TransformGroup ya que la rotación depende de un objeto Alpha, el valor de este objeto Alpha cambia con respecto al tiempo. Al objeto RotationInterpolator se le pueden especificar distintos parámetros, como el eje de rotación, el ángulo de inicio y fin.



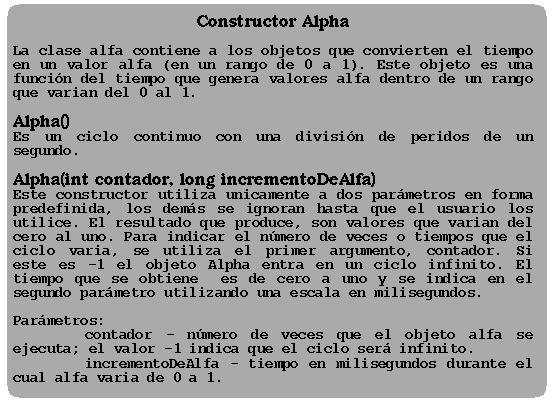
**Funciones para variación del tiempo.**

Estas acciones se deben llevar a cabo por medio de un objeto tipo Alpha.

**Clase Alpha**

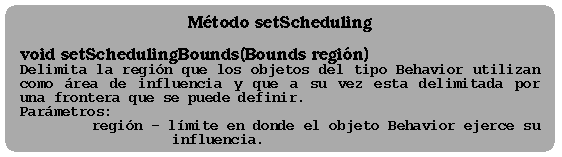
Estos objetos se utilizan para crear una función que varie respecto al tiempo. La clase Alpha establece un valor entre creo y uno, este valor depende del tiempo y de los parámetros del objeto Alpha y por lo general se utilizan para crear animaciones de los objetos.

Exuisten diez parámetros distintos que esta clase puede utilizar, lo que da al programador una flexibilidad muy grande, pero aqui no se profundizará mucho en estos, sólo se enseñará como hacer instancias para hacer una rotación.



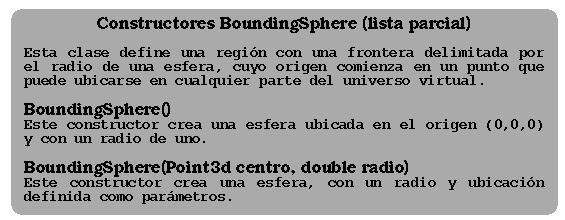
**Región de acción**

Como se menciono en la página anterior, cada objeto behavior tiene su radio de acción, por lo que se utiliza el método setSchedulingBounds perteneciente a la clase Behavior. Existen varias alternativas para indicar este campo de acción, el más sencillo, es crear una objeto BoundingSphere, pero hay que señalar la existencia de objetos BoundingBox y BoundingPolytope.



**Clase BoundingSphere**

Crea una esfera en que define el campo de acción, los parámetros que se utilizan son para indicar el origen y el radio, la posición predeterminada es el origen (0,0,0). El radio es muy importante ya que debe ser lo bastante grande para que contenga al objeto visual.



El siguiente código muestra cómo utilizar una clase que hace uso de la interpolación para crear una animación, esta consiste en realizar una rotación continua por una duración de cuatro segundos.

**Paso 1** Se crea el objeto TransformGroup para permitir la animación cuando se ejecute el programa, ver linea 18, la abilitación sucede en la linea 19.

**Paso 2** Se crea un objeto tipo alpha para la interpolación y así controlar la rotación, para que sea continua, ver linea 28. Los dos parámetros que se que se utilizan son el número de iteraciones y el tiempo que debe durar uno ciclo, el tiempo se indica en milisegundos, para ese caso, el valor 4000 indica que cada rotación durará 4 segundos.

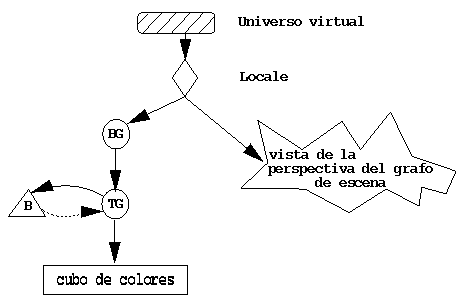
**Paso 3** Se crea un objeto de l tipo RotationInterpolator en donde se guarde las interpolaciones, ver lineas 30-31, este interpolador debe estar referenciado al objeto tipo alfa, en este caso se incluye en en cosntructor mismo, el ejeY es en donde se realiza la rotación, esto ya viene predeterminado.

**Paso 4** Se especifica la región en donde se llevará a cabo el "comportamiento", se crea un objeto tipo BoundingSphere con sus argumentos predefinidos, ver linea 35, finalmente se activa su "comportamineto", linea 36.

**Paso 5** El paso final es hacer de este comportamiento hijo del TransformGroup, linea 38.

1. **import java.applet.Applet;**
2. **import java.awt.BorderLayout;**
3. **import java.awt.Frame;**
4. **import java.awt.event.\*;**
5. **import com.sun.j3d.utils.applet.MainFrame;**
6. **import com.sun.j3d.utils.geometry.ColorCube;**
7. **import com.sun.j3d.utils.universe.\*;**
8. **import javax.media.j3d.\*;**
9. **import javax.vecmath.\*;**
10. **public class HolaJava3Dc extends Applet {**
11. **public BranchGroup createSceneGraph() {**
12. **// Crear la raiz de la grafica**
13. **BranchGroup objRoot = new BranchGroup();**
14. **// Crear el grupo de transformacion y lo an~ade a la raiz**
15. **TransformGroup objSpin = new TransformGroup();**
16. **objSpin.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);**
17. **objRoot.addChild(objSpin);**
18. **// Crea una hoja, y la an~ade a la escena**
19. **// ColorCube es una clase de Utility**
20. **objSpin.addChild(new ColorCube(0.4));**
21. **// Crea un objeto Behavior con el cual poder aplicar transformaciones**
22. **// al objeto que se quiera y an~adirlo a la escena**
23. **Alpha rotationAlpha = new Alpha(-1, 4000);**
24. **RotationInterpolator rotator =**
25. **new RotationInterpolator(rotationAlpha, objSpin);**
26. **// El borde de la esfera especifica la region en donde se**
27. **// puede manejar el comportamiento del objeto**
28. **// La esfera esta en el origen y tiene un radio de 100**
29. **BoundingSphere bounds = new BoundingSphere();**
30. **rotator.setSchedulingBounds(bounds);**
31. **objSpin.addChild(rotator);**
32. **return objRoot;**
33. **} // fin del metodo CreateSceneGraph**
34. **public HolaJava3Dc() {**
35. **setLayout(new BorderLayout());**
36. **// Creacion del Canvas**
37. **Canvas3D canvas3D = new Canvas3D(null);**
38. **add("Center", canvas3D);**
39. **BranchGroup escena = createSceneGraph();**
40. **// Creacion del universo**
41. **SimpleUniverse simpleU = new SimpleUniverse(canvas3D);**
42. **// Ubica la vista al frente de la escena para ver a todos los**
43. **// objetos**
44. **simpleU.getViewingPlatform().setNominalViewingTransform();**
45. **simpleU.addBranchGraph(escena);**
46. **} // fin del HolaJava3Da (constructor)**
47. **// Para correr el programa como un applet**
48. **public static void main(String[] args) {**
49. **Frame frame = new MainFrame(new HolaJava3Da(), 256, 256);**
50. **} // fin del main**
51. **} // fin de la clase HolaJava3Dc**

Este programa crea un grafo de escena, en donde el objeto que se utiliza para hacer la rotación es hijo de objSpin que si recuerda, ya era padre,por lo que se viola la restricción de no utilizar ciclos, pero esto no es así ya que esta es una referencia y esta no se define como un arco dentro de un grafo de escena y se representa como una flecha puntuada, ver la siguiente figura.

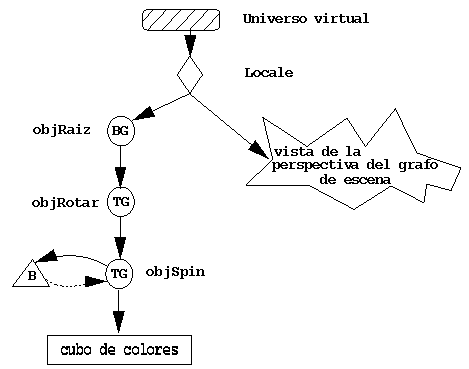


**Combinación de los objetos tipo Behavior y Transformation**

Para conjuntar a estos dos objetos, se tiene el siguiente programa en donde existen dos objetos: objRotate y ObjSpin, la diferencia de uno con respecto al otro, es que el primero define una rotación estática y el segundo es continua.

1. **import java.applet.Applet;**
2. **import java.awt.BorderLayout;**
3. **import java.awt.Frame;**
4. **import java.awt.event.\*;**
5. **import com.sun.j3d.utils.applet.MainFrame;**
6. **import com.sun.j3d.utils.geometry.ColorCube;**
7. **import com.sun.j3d.utils.universe.\*;**
8. **import javax.media.j3d.\*;**
9. **import javax.vecmath.\*;**
10. **public class HolaJava3Dd extends Applet {**
11. **public BranchGroup createSceneGraph() {**
12. **// Crear la raiz de la grafica**
13. **BranchGroup objRoot = new BranchGroup();**
14. **// rota el objeto con la matriz de transformaciones**
15. **Transform3D rotate = new Transform3D();**
16. **Transform3D tempRotate = new Transform3D();**
17. **rotate.rotX(Math.PI/4.0d);**
18. **tempRotate.rotY(Math.PI/5.0d);**
19. **rotate.mul(tempRotate);**
20. **TransformGroup objRotate = new TransformGroup(rotate);**
21. **// Crea el nodo de transformacion y permite que el comportamiento**
22. **// pueda modificarse cuando se esta corriendo el programa, con**
23. **// TRANSFORM\_WRITE**
24. **TransformGroup objSpin = new TransformGroup();**
25. **objSpin.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);**
26. **objRoot.addChild(objRotate);**
27. **objRotate.addChild(objSpin);**
28. **// Crea una hoja, y la an~ade a la escena**
29. **// ColorCube es una clase de Utility**
30. **objSpin.addChild(new ColorCube(0.4));**
31. **// Crea un objeto Behavior con el cual poder aplicar transformaciones**
32. **// al objeto que se quiera y an~adirlo a la escena**
33. **Transform3D yAxis = new Transform3D();**
34. **Alpha rotationAlpha = new Alpha(-1, 4000);**
35. **RotationInterpolator rotator =**
36. **new RotationInterpolator(rotationAlpha, objSpin, yAxis,**
37. **0.0f, (float) Math.PI\*2.0f);**
38. **// El borde de la esfera especifica la region en donde se**
39. **// puede manejar el comportamiento del objeto**
40. **// La esfera esta en el origen y tiene un radio de 100**
41. **BoundingSphere bounds = new BoundingSphere();**
42. **rotator.setSchedulingBounds(bounds);**
43. **objSpin.addChild(rotator);**
44. **return objRoot;**
45. **} // fin del metodo CreateSceneGraph**
46. **public HolaJava3Dd() {**
47. **setLayout(new BorderLayout());**
48. **// Creacion del Canvas**
49. **Canvas3D canvas3D = new Canvas3D(null);**
50. **add("Center", canvas3D);**
51. **BranchGroup escena = createSceneGraph();**
52. **// Creacion del universo**
53. **SimpleUniverse simpleU = new SimpleUniverse(canvas3D);**
54. **// Ubica la vista al frente de la escena para ver a todos los**
55. **// objetos**
56. **simpleU.getViewingPlatform().setNominalViewingTransform();**
57. **simpleU.addBranchGraph(escena);**
58. **} // fin del HolaJava3Da (constructor)**
59. **// Para correr el programa como un applet**
60. **public static void main(String[] args) {**
61. **Frame frame = new MainFrame(new HolaJava3Da(), 256, 256);**
62. **} // fin del main**
63. **} // fin de la clase HolaJava3Dd**

Grafo de escena que describe al programa HolaJava3Dd.



1. **import java.applet.Applet;**
2. **import java.awt.BorderLayout;**
3. **import java.awt.Frame;**
4. **import java.awt.event.\*;**
5. **import com.sun.j3d.utils.applet.MainFrame;**
6. **import com.sun.j3d.utils.geometry.ColorCube;**
7. **import com.sun.j3d.utils.universe.\*;**
8. **import javax.media.j3d.\*;**
9. **import javax.vecmath.\*;**
10. **public class HolaJava3Dd extends Applet {**
11. **public BranchGroup createSceneGraph() {**
12. **// Crear la raiz de la grafica**
13. **BranchGroup objRoot = new BranchGroup();**
14. **// rota el objeto con la matriz de transformaciones**
15. **Transform3D rotate = new Transform3D();**
16. **Transform3D tempRotate = new Transform3D();**
17. **rotate.rotX(Math.PI/4.0d);**
18. **tempRotate.rotY(Math.PI/5.0d);**
19. **rotate.mul(tempRotate);**
20. **TransformGroup objRotate = new TransformGroup(rotate);**
21. **// Crea el nodo de transformacion y permite que el comportamiento**
22. **// pueda modificarse cuando se esta corriendo el programa, con**
23. **// TRANSFORM\_WRITE**
24. **TransformGroup objSpin = new TransformGroup();**
25. **objSpin.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);**
26. **objRoot.addChild(objRotate);**
27. **objRotate.addChild(objSpin);**
28. **// Crea una hoja, y la an~ade a la escena**
29. **// ColorCube es una clase de Utility**
30. **objSpin.addChild(new ColorCube(0.4));**
31. **// Crea un objeto Behavior con el cual poder aplicar transformaciones**
32. **// al objeto que se quiera y an~adirlo a la escena**
33. **Transform3D yAxis = new Transform3D();**
34. **Alpha rotationAlpha = new Alpha(-1, 4000);**
35. **RotationInterpolator rotator =**
36. **new RotationInterpolator(rotationAlpha, objSpin, yAxis,**
37. **0.0f, (float) Math.PI\*2.0f);**
38. **// El borde de la esfera especifica la region en donde se**
39. **// puede manejar el comportamiento del objeto**
40. **// La esfera esta en el origen y tiene un radio de 100**
41. **BoundingSphere bounds = new BoundingSphere();**
42. **rotator.setSchedulingBounds(bounds);**
43. **objSpin.addChild(rotator);**
44. **return objRoot;**
45. **} // fin del metodo CreateSceneGraph**
46. **public HolaJava3Dd() {**
47. **setLayout(new BorderLayout());**
48. **// Creacion del Canvas**
49. **Canvas3D canvas3D = new Canvas3D(null);**
50. **add("Center", canvas3D);**
51. **BranchGroup escena = createSceneGraph();**
52. **// Creacion del universo**
53. **SimpleUniverse simpleU = new SimpleUniverse(canvas3D);**
54. **// Ubica la vista al frente de la escena para ver a todos los**
55. **// objetos**
56. **simpleU.getViewingPlatform().setNominalViewingTransform();**
57. **simpleU.addBranchGraph(escena);**
58. **} // fin del HolaJava3Da (constructor)**
59. **// Para correr el programa como un applet**
60. **public static void main(String[] args) {**
61. **Frame frame = new MainFrame(new HolaJava3Da(), 256, 256);**
62. **} // fin del main**
63. **} // fin de la clase HolaJava3Dd**

**Capítulo 2 Creación de Geometrías.**

***Objetivos***

Después de leer este capítulo, será capaz de:

 Utilizar las clases que definen las primitivas (geometrías).

 Escribir clases para construir objetos.

 Definir las geometrías utilizando el núcleo de las clases .

 Definir apariencia (atributos visuales) en las geometrías.

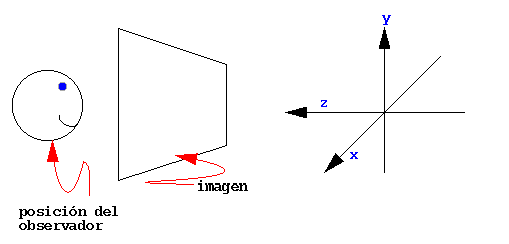
En este capítulo se explicarán los conceptos básicos para construir dentro del universo virtual un conjunto de objetos geométricos. Existesn tres formas para contruir una geometría en Java 3D, la primera y más fácil es haciendo uso de las clases que definen cuerpos geometricos ya preconstriudos, como son cajas, conos, cilindros y esferas. Otra es indicar las coordenadas de los vértices que construyen a un objeto, por lo que se pueden utilizar, puntos, segmentos de lineas y/o superficies de polígonos. Y la última es leer la geometría de un archivo. En este capítulo solamente se tratarán las dos primeras.

Tambien se tocarán los principios para dar efectos visuales a los objetos y operaciones matemáticas.

**El Sistema de Coordenadas en el Universo Virtual.**

Como se mencionó en el capítulo pasado, una instacia de la clase VirtualUniverse marca la raiz de grafo de escena en cualquier programa de Java 3D, a este espacio virtual tambien lo podemos definir como el espacio tridimensional en donde residen todos los objetos. Un objeto Locale en un universo virtual, es el que establece el sistema de coordenadas cartesiano para ese universo, se pueden tener varios objetos Locale en un mismo universo.

La función principal de los objetos Locale, es servir de punto de refencia para los objetos que estan contenidos en el universo virtual, este sistema de coordenadas, tiene una orientación de la mano derecha (right-handed). La dirección del eje de las X es positivo hacia la derecha, el de las Y es hacia arriba y el de la Z esta orientado hacia el usuario (profundidad), las unidades que se utilizan son metros. La siguiente figura ilustra cual es la orientación del universo virtual con respecto al observador.



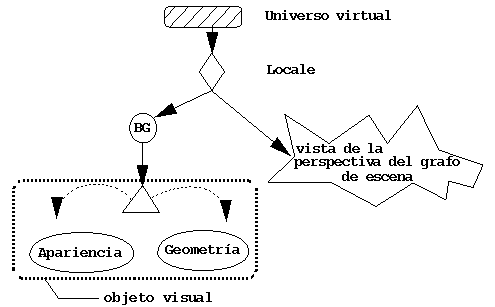
***Definiciones básicas de los objetos visuales***

Se presentará una descripción de las clases tipo Shape3D y NodeComponent, para finalmente definir como utilizar geometrías más complejas con apariencia.

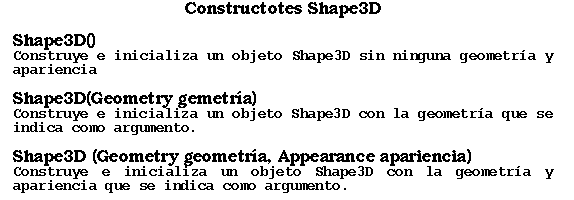
**Cómo realizar una instancia utilizando la clase Shape3D para construir un objeto.**

Un nodo del tipo Shape3D en un grafo de escena, sirve para indicar que hay un objeto visual. Shape3D es uno de las subclases de la clase Leaf, por lo que este tipo de objetos únicamente pueden ser hojas en el árbol del grafo de escena. Otra cosa que hay que tomar en cuenta, es que este tipo de objetos no tiene información acerca la forma o el color del objeto. Esa información se guarda en los objetos NodeComponent que despues son referenciados por los objetos Shape3D. Un objeto Shape3D tambien puede referenciar a otro del tipo Geometry y Appearance.

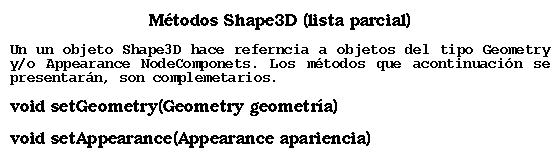
En los grafos de escena que se utilizarón en el capítulo pasado, el símbolo que se utilizo para identificar un objeto es el triángulo, a partir de ahora, un objeto visual se representará con el mismo triángulo que será un objeto del tipo Shape3D y dos ovalos que son los NodeComponets, todos ellos contenidos dentro de un rectángulo punteado. Como ejemplo se muestra en la siguiente figura.

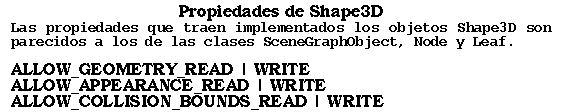


Un objeto visual se puede definir utilizando un objeto Shape3D y un nodo Geometry, opcionalmente el objeto Shape3Dtambien se puede referenciar a un nodo Appearance sin problema alguno. Los constructores para Shape3D muestran que se pueden crear con más de un nodo de referencia.



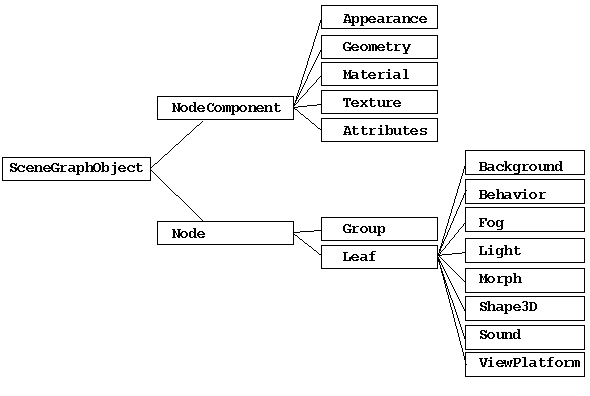
Un objeto Shape3D no esta "vivo" ni "compilado", es por eso que sus componentes pueden alterarse con los métodos que a continuación se muestran. Dichos métodos se pueden utilizar objetos Shape3D "vivos" o "compilados" si se han establecido que propiedades tiene el objeto.





**Componentes tipo Nodo.**

Este tipo de objetos contienen las especificaciones exactas de los atributos que tiene cada objeto. Muchas de las subclases NodeComponent definen algunos atributos visuales, la siguiente figura muestra parte de la jerarquia que prevalece en la clase NodeComponent, asi como sus descendientes.



**Definiendo las clases de un Objeto Visual.**

El mismo objeto visual, puede aparacer sin ningun problema en más de una ocación en un universo virtual, esto hace que se tenga más cuidado en el momento de crearlo. Hay que decir, que existen varias formas para crear una clase que defina a un objeto visual.

El siguiente segmento de código, ilusta en forma general, el esqueleto de una clase llamada ObjetoVisual, en donde se encuentra en forma esquemática, la organización de un objeto visual. Los métodos se encuentran vacios en el código. Este ejemplo no se utiliza en ningun otro programa, ya que es demasiado general.

1. public class ObjetoVisual extends Shape3D{
2. private Geometry voGeometria;
3. private Appearance voApariencia;
4. // Se crea un objeto del tipo Shape3D con una geometría y una apariencia.
5. //La geometría se construye en el método creaGeometria
6. // La aparinecia se crea en el método creaApariencia
7. public objetoVisual() {
8. voGeometria = creaGeometria();
9. voApariencia = creaApariencia();
10. this.setGeometry(voGeometria):
11. this.setAppearance(voApariencia):
12. }
13. private Geometry creaGeometria(){
14. // codigo para crear la geometria
15. }
16. private Appearance creaApariencia(){
17. // codigo para crear la apariencia
18. }
19. }// fin de la clase ObjetoVisual

La organización de la clase ObjetoVisual, es similar a la clase que define a el cubo de color. La clase VisualObject por lo general es en donde se sugiere que sea el punto de partida para comenzar a construir a las clases que se utilizan en el grafo de escena. Cada programador puede organizar su espacio como a el mejor le acomode. Para completar la información del código anterior, se sugiere leer el código fuente del la clase ColorCube que se encuentra en el paquete com.sun.j3d.utils.geometry.

En el momento que se utiliza a un Shape3D, facilita mucho la creación de un objeto visual y es tan fácil de usar, que se parece al ejemplo del cubo de color. El constructor puede utilizarce para crear un objeto nuevo para insertarlo como hijo de algun Group, y todo en una sola linea de códico, por ejemplo, en la siguiente linea de código, ObRaiz es una instacia de Group, creando de esta forma un objeto visual e incorporandolo al mismo tiempo como hijo de objRaiz en el grafo de escena.

* objRaiz.addChild(new ObjetoVisual());

El constructor ObjetoVisual crea un objeto que a su vez crea un objeto Shape3D, que sirvepara referencial a los NodeComponents creados a su vez por los métodos creaGeometria() y creaApariencia(). El método creaGeometria() crea una componente-nodo-geometría para utilizarlo en el objeto visual. El método creaApariencia() es el responsable de crear un nodo-componente que define la apariencia del objeto.

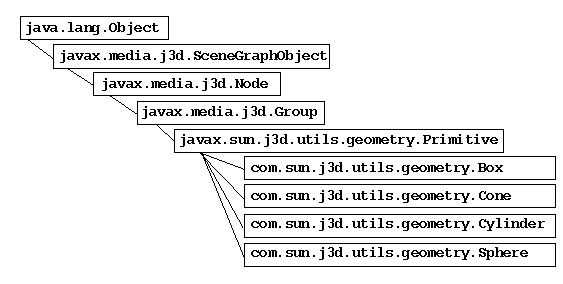
Otra posible organización, es definir una clase contenedora que no derive del API de Java3D. En este diseño, la clase que contiene al objeto visual puede tener un nodo del tipo Group o uno del tipo Shape3D como raiz de la subgrafica. Esta clase tendrá que definir a algunos métodos que regresen una referencia a la raiz. Esta técnica es más tediosa pero más fácil de entender, por lo que en algunos de los ejemplos que se aqui se presentan, se hará uso de esta técnica.

Una tercera organización, es utilizar las clases Box, Cone, Cylinder y Sphere definidas en el paquete *com.sun.j3d.utils.geometry*. Cada una de estas clases, son extenciones de Primitive, la que a su vez pertenece a Group. El diseño detallado de cada una de estas, no se discute en este tutorial, pero el código fuente de todas ellas, se encuentra disponible en el Java3D API.

**Clases Geométricas.**

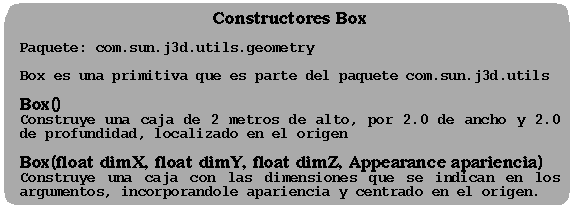
Aqui trataremos el tema, de como crear cajas, conos, cilindros y esferas, utilizando las primitivas que vienen en las utilerias de Java3D, con estas primitivas es la forma más fácil y rápida para crear el contenido de un universo virtual. Estos objetos son muy versátiles por que se les puede cambiar facilmente sus dimensiones, su color y textura aunque no su forma. Tambien da al programador la facilidad de contar con varias instancias del mismo objeto, pero modificando su color y apariencia en cada uno.

Las clases Box, Cone, Cylinder y Sphere se encuentran en el paquete com.sun.j3d.utils.geometry, la superclase de estas primitivas es la clase Primitive, a continuación se muestra la gerarquia de este paquete.

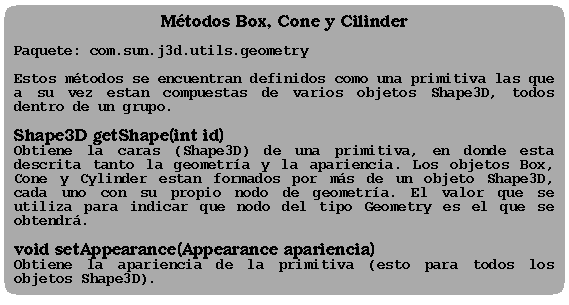


**BOX**

Esta primitiva, crea una caja tridimensional, Las dimensiones predeterminadas que tiene son de 2 metros tanto en largo, ancho y alto, se encuentra centrado en el origen por lo que sus esquinas estan en (-1,-1,-1) y (1,1,1). El largo, ancho y altura se pueden indicar en el momento que se crea el cubo, y para modificar su ubicación y orientación debe utilizarce un objeto TransformGroup.

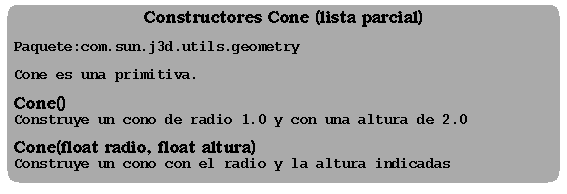


Algunos de los constructores solamente cambian por su clase, pero Box, Cone y Cilinder comparten casi los mismos métodos. La referencia siguiente lista los métodos de estas clases.



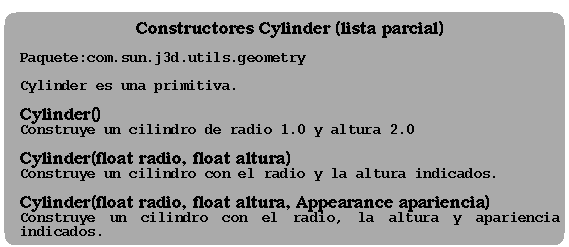
**Cone**

Define a un cono, centrado en el origen con su eje central alineado en el eje X. El radio predeterminado es de 1.0 y con una altura de 2.0 metros. El centro del cono se encuentra en el origen del su centroide.



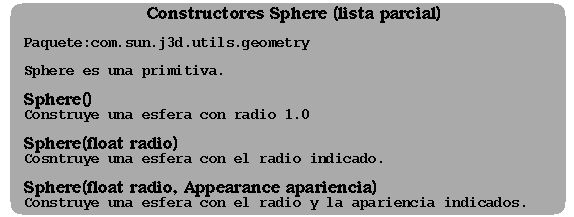
**Cilindro**

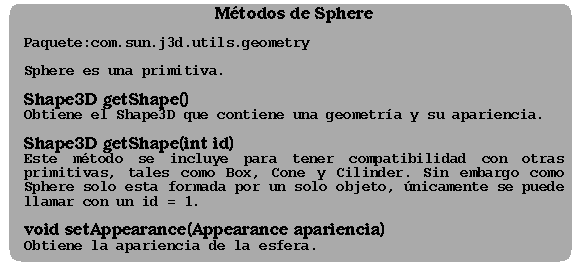
Crea un objeto cilíndrico, ubicado en el origen alineado a lo largo del eje Y. Los valores predeterminados para su radio es de 1.0 con una altura de 2.0 metros.



**Sphere**

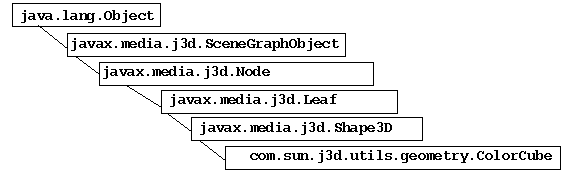
Crea un objeto esferico, ubicado en el centro del sistema de coordenadas y con un radio predeterminado de 2.0 metros.





**Cubo de color**

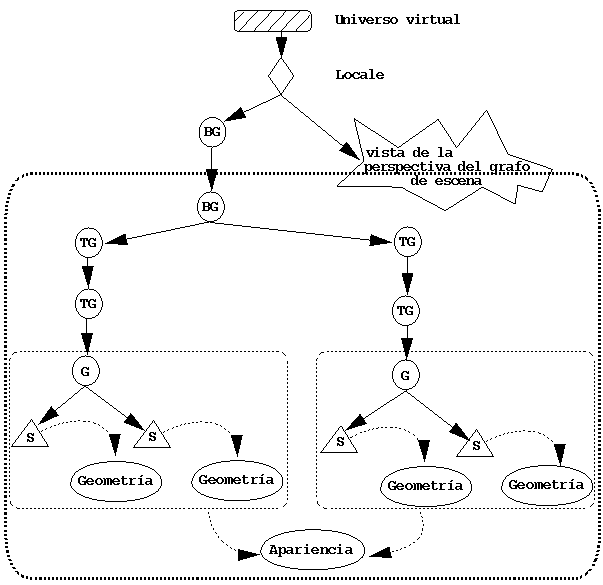
Esta clase es un contraste para las anteriores ya que deriva de una distina promitiva gráfica, pero es una subclase de Shape3D. La jerarquia de este cubo de color se muestra a continuación.



Esta clase es la única que presenta luces y colores propios, por lo que es apropiada para hacer pruebas o prototipos de las escenas.

***Ejemplo: Crear un Yóyo utilizando dos conos para formarlo.***

En esta parte crearemos un programa llamado YoyoDeConos, cuyo objetivo principal será construir con dos conos, un yóyo tridimensional con movimiento, para lo que utilizaremos los behaviors del Java3D API, se encontrará ubicado en el centro y orientado en dirección del eje de las Y y la translación será en el de las X. A continuación se muestra el grafo de escena que describe al programa.



En este grafo de escena, se puede observar que el objeto que construye al Yóyo comienza con un objeto del tipo BranchGroup como la raiz, a continuación este se divide en dos objetos TransformGroup que sirven para indicar la translación y posterior rotación de cada uno de los conos.

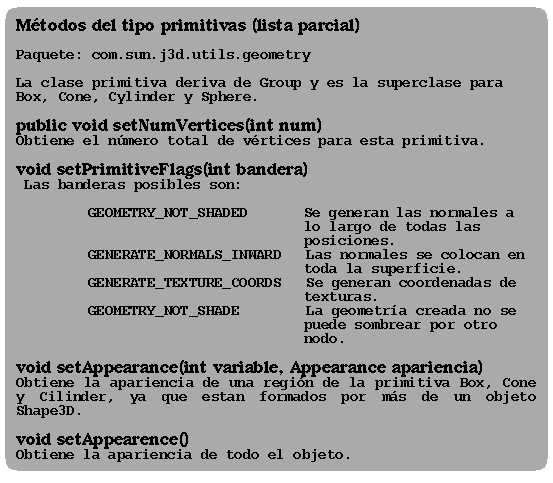
Este grafo de escena tambien nos sirve para ilustrar, que varias subgraficas pueden formar un mismo universo virtual. Los nodos tipo Shape3D (que en el diagrama se representan con la letra S), sirven para representar en forma interna a los conos, los que a su vez son hijos del objeto Group, es por eso que los conos (o cualquier otra primitiva) no pueden reutilizarce otra vez, por reglas del grafo de escena.

A continuación se muestra el código fuente, que construye el yóyo.

1. **/\***
2. **\* Este programa demuestra:**
3. **\* 1. Como escribir un objeto visual**
4. **\* En este programa, la clase YoyoCono define un objeto visual**
5. **\* independientemente de las demas clases**
6. **\* 2. Se utiliza a Cone para crear la superficie.**
7. **\*/**
8. **import java.applet.Applet;**
9. **import java.awt.BorderLayout;**
10. **import java.awt.Frame;**
11. **import java.awt.event.\*;**
12. **import com.sun.j3d.utils.applet.MainFrame;**
13. **import com.sun.j3d.utils.geometry.Cone;**
14. **import com.sun.j3d.utils.universe.\*;**
15. **import javax.media.j3d.\*;**
16. **import javax.vecmath.\*;**
17. **public class YoyoDeConos extends Applet {**
18. **/////////////////////////////////////////////////**
19. **//**
20. **// crear la grafica de la escena**
21. **//**
22. **public class YoyoCono{**
23. **private BranchGroup yoyoBG;**
24. **////////////////////////////////////////////**
25. **//**
26. **// crear el objeto3D con geometria y apariencia**
27. **// la geometra es creada en el metodo yoyoGeometry**
28. **// la apariencia es creada en el metodo yoyoAppearance**
29. **//**
30. **public YoyoCono() {**
31. **yoyoBG = new BranchGroup();**
32. **Appearance app = new Appearance();**
33. **// rotar el objeto utilizando la matriz de transformacion**
34. **Transform3D rotar = new Transform3D();**
35. **Transform3D transladar = new Transform3D();**
36. **transladar.set(new Vector3f(0.1f, 0.0f, 0.0f));**
37. **TransformGroup yoyoTGT1 = new TransformGroup(transladar);**
38. **yoyoBG.addChild(yoyoTGT1);**
39. **rotar.rotZ(Math.PI/2.0d);**
40. **TransformGroup yoyoTGR1 = new TransformGroup(rotar);**
41. **Cone cono1 = new Cone(0.6f, 0.2f);**
42. **cono1.setAppearance(app);**
43. **yoyoTGR1.addChild(cono1);**
44. **yoyoTGT1.addChild(yoyoTGR1);**
45. **transladar.set(new Vector3f(-0.1f, 0.0f, 0.0f));**
46. **TransformGroup yoyoTGT2 = new TransformGroup(transladar);**
47. **yoyoBG.addChild(yoyoTGT2);**
48. **rotar.rotZ(-Math.PI/2.0d);**
49. **TransformGroup yoyoTGR2 = new TransformGroup(rotar);**
50. **Cone cono2 = new Cone(0.6f, 0.2f);**
51. **cono2.setAppearance(app);**
52. **yoyoTGR2.addChild(cono2);**
53. **yoyoTGT2.addChild(yoyoTGR2);**
54. **yoyoBG.compile();**
55. **} //fin del constructor YoyoCono**
56. **public BranchGroup getBG(){**
57. **return yoyoBG;**
58. **}**
59. **} // fin de la clase YoyoCono**
60. **/////////////////////////////////////////////////**
61. **//**
62. **// crear la grafica de la escena**
63. **//**
64. **public BranchGroup createSceneGraph() {**
65. **BranchGroup objRaiz = new BranchGroup();**
66. **// Crea el grupo de transformacion, inicializa la matriz como indentidad**
67. **// Abilita la capacidad de modificar el comportamiento de las geometrias**
68. **// con TRANSFORM\_WRITE, y la incorpora a la raiz**
69. **TransformGroup objSpin = new TransformGroup();**
70. **objSpin.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);**
71. **objRaiz.addChild(objSpin);**
72. **objSpin.addChild(new YoyoCono().getBG());**
73. **// Crea un nuevo objeto de Compotamiento para implementar la**
74. **// transformacion que se desee en cualquir obkjeto**
75. **// an~adiendolo despues a la escena**
76. **Transform3D yAxis = new Transform3D();**
77. **Alpha rotacionAlfa = new Alpha(-1, 4000);**
78. **RotationInterpolator rotacion =**
79. **new RotationInterpolator(rotacionAlfa, objSpin);**
80. **BoundingSphere frontera =**
81. **new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);**
82. **rotacion.setSchedulingBounds(frontera);**
83. **objSpin.addChild(rotacion);**
84. **// optimiza el desplige de la escena**
85. **objRaiz.compile();**
86. **return objRaiz;**
87. **} // fin del metodo CreateSceneGraph de yoyo1**
88. **// Crear la escena y an~adirla al universo virtual**
89. **public YoyoDeConos() {**
90. **setLayout(new BorderLayout());**
91. **Canvas3D canvas3D = new Canvas3D(null);**
92. **add("Center", canvas3D);**
93. **BranchGroup escena = createSceneGraph();**
94. **SimpleUniverse simpleU = new SimpleUniverse(canvas3D);**
95. **simpleU.getViewingPlatform().setNominalViewingTransform();**
96. **simpleU.addBranchGraph(escena);**
97. **} // fin del constructor YoyoDeConos**
98. **// Para que corra como aplicacion**
99. **public static void main(String[] args) {**
100. **Frame cuadro = new MainFrame(new YoyoDeConos(), 256, 256);**
101. **} // fin del metodo principal de conoyoyo**
102. **} // fin de la clase YoyoDeConos**

En las lineas 49 a 54 se crean los dos conos para el yóyo, en las lineas 52 a 54 se crea el objeto que será la unión entre ambos objetos, este mismo proceso se repite para el segundo cono del yóyo.

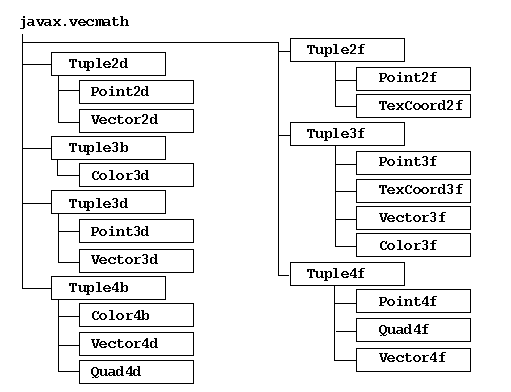
En la linea 39 se crea un objeto del tipo Appearance con los parámetros predeterminados, posteriormente se utilizan en los dos conos para que tengan apariencia, esto se realizá en las lineas 52 y 63.



**Clases Matemáticas**

Para crear objetos visuales, utilizamos las clases y subclases contenidas en Geometry. Muchas de estas subclases sirven para definir puntos, lineas o polígonos rellenos, pero en este momento vamos a discutir varias clases matemáticas (Point\*, Color\*, Vector\*, TextCoor\*, Tuple\*), que necesitamos para indicar y manipularla relación que necesitamos establecer en cada uno de los vértices de las geométrias y que forzozamente se necesitan conocer.

Utilizamos un asterisco al final de cada una de las clases, con el fin de indicar que tienen varias representaciones, por ejempo Tuple\* indica que existen las siguientes clases: Tuple2f, Tuple2d, tuple3d, Tuple3f, Tuple3d, Tuple4b, Tunple4f Tuple4d. Para cada caso el primer número indica uantos elementos tiene en cada tupla, y la letra inda el tipo, 'f' signifca real, 'd' indica que es de doble presición y real y 'b' para byte. Por lo que Tuple3f utiliza tres valores reales de simple presición.



1. **import java.applet.Applet;**
2. **import java.awt.BorderLayout;**
3. **import java.awt.Frame;**
4. **import java.awt.event.\*;**
5. **import com.sun.j3d.utils.applet.MainFrame;**
6. **import com.sun.j3d.utils.universe.\*;**
7. **import javax.media.j3d.\*;**
8. **import javax.vecmath.\*;**
9. **public class EjesApp extends Applet {**
10. **/////////////////////////////////////////////////**
11. **//**
12. **// crear el grupo**
13. **//**
14. **public class Ejes{**
15. **private BranchGroup ejesBG;**
16. **////////////////////////////////////////////**
17. **//**
18. **// crear la subgrafica ejes**
19. **//**
20. **public Ejes() {**
21. **ejesBG = new BranchGroup();**
22. **// crear la linea del eje X**
23. **LineArray LineasEjeX = new LineArray(2, LineArray.COORDINATES );**
24. **ejesBG.addChild(new Shape3D(LineasEjeX));**
25. **LineasEjeX.setCoordinate(0, new Point3f(-1.0f, 0.0f, 0.0f));**
26. **LineasEjeX.setCoordinate(1, new Point3f( 1.0f, 0.0f, 0.0f));**
27. **Color3f red = new Color3f(1.0f, 0.0f, 0.0f);**
28. **Color3f green = new Color3f(0.0f, 1.0f, 0.0f);**
29. **Color3f blue = new Color3f(0.0f, 0.0f, 1.0f);**
30. **// crear la linea pare el eje Y**
31. **LineArray LineasEjesY = new LineArray(2,**
32. **LineArray.COORDINATES | LineArray.COLOR\_3 );**
33. **ejesBG.addChild(new Shape3D(LineasEjesY));**
34. **LineasEjesY.setCoordinate(0, new Point3f( 0.0f,-1.0f, 0.0f));**
35. **LineasEjesY.setCoordinate(1, new Point3f( 0.0f, 1.0f, 0.0f));**
36. **LineasEjesY.setColor(0, green);**
37. **LineasEjesY.setColor(1, blue);**
38. **// crear la linea del eje Z**
39. **Point3f z1 = new Point3f( 0.0f, 0.0f,-1.0f);**
40. **LineArray LineasEjesZ = new LineArray(10,**
41. **LineArray.COORDINATES | LineArray.COLOR\_3**
42. **);**
43. **ejesBG.addChild(new Shape3D(LineasEjesZ));**
44. **LineasEjesZ.setCoordinate(0, z1);**
45. **LineasEjesZ.setCoordinate(1, z2);**
46. **LineasEjesZ.setCoordinate(2, z2);**
47. **LineasEjesZ.setCoordinate(3, new Point3f( 0.1f, 0.1f, 0.9f));**
48. **LineasEjesZ.setCoordinate(4, z2);**
49. **LineasEjesZ.setCoordinate(5, new Point3f(-0.1f, 0.1f, 0.9f));**
50. **LineasEjesZ.setCoordinate(6, z2);**
51. **LineasEjesZ.setCoordinate(7, new Point3f( 0.1f,-0.1f, 0.9f));**
52. **LineasEjesZ.setCoordinate(8, z2);**
53. **LineasEjesZ.setCoordinate(9, new Point3f(-0.1f,-0.1f, 0.9f));**
54. **Color3f colores[] = new Color3f[9];**
55. **colores[0] = new Color3f(0.0f, 1.0f, 1.0f);**
56. **for(int v = 0; v < 9; v++){**
57. **colores[v] = red;**
58. **}**
59. **LineasEjesZ.setColors(1, colores);**
60. **} // fin del constructor ejes**
61. **public BranchGroup getBG(){**
62. **return ejesBG;**
63. **}**
64. **} // fin de la clase Ejes**
65. **/////////////////////////////////////////////////**
66. **//**
67. **// crear el gropo de la escena**
68. **//**
69. **public BranchGroup createSceneGraph() {**
70. **BranchGroup objRaiz = new Ejes().getBG();**
71. **// optimiza el desplegue de la escena**
72. **objRaiz.compile();**
73. **return objRaiz;**
74. **} // fin del metodo CreateSceneGraph**
75. **// Crear una escena y la pega al universo virtual**
76. **public EjesApp() {**
77. **setLayout(new BorderLayout());**
78. **Canvas3D canvas3D = new Canvas3D(null);**
79. **add("Center", canvas3D);**
80. **BranchGroup escena = createSceneGraph();**
81. **// SimpleUniverse**
82. **SimpleUniverse simpleU = new SimpleUniverse(canvas3D);**
83. **// para ver a todos los objetos de la escena**
84. **simpleU.getViewingPlatform().setNominalViewingTransform();**
85. **simpleU.addBranchGraph(escena);**
86. **} // fin del constructor coneyoyo**
87. **// tpara que corra como aplicacion**
88. **public static void main(String[] args) {**
89. **Frame frame = new MainFrame(new EjesApp(), 256, 256);**
90. **} // fin del metod principal axisapp**
91. **} // fin de la clase ejesapp**
92. **import javax.media.j3d.\*;**
93. **import javax.vecmath.\*;**
94. **public class Ejes extends Shape3D{**
95. **////////////////////////////////////////////**
96. **//**
97. **// crear los ejes**
98. **//**
99. **public Ejes() {**
100. **this.setGeometry(createGeometry());**
101. **}**
102. **private Geometry createGeometry(){**
103. **// crear la linea del eje X**
104. **IndexedLineArray LineasDeEjes = new IndexedLineArray(18, GeometryArr**
105. **ay.COORDINATES, 30);**
106. **LineasDeEjes.setCoordinate( 0, new Point3f(-1.0f, 0.0f, 0.0f));**
107. **LineasDeEjes.setCoordinate( 1, new Point3f( 1.0f, 0.0f, 0.0f));**
108. **LineasDeEjes.setCoordinate( 2, new Point3f( 0.9f, 0.1f, 0.1f));**
109. **LineasDeEjes.setCoordinate( 3, new Point3f( 0.9f,-0.1f, 0.1f));**
110. **LineasDeEjes.setCoordinate( 4, new Point3f( 0.9f, 0.1f,-0.1f));**
111. **LineasDeEjes.setCoordinate( 5, new Point3f( 0.9f,-0.1f,-0.1f));**
112. **LineasDeEjes.setCoordinate( 6, new Point3f( 0.0f,-1.0f, 0.0f));**
113. **LineasDeEjes.setCoordinate( 7, new Point3f( 0.0f, 1.0f, 0.0f));**
114. **LineasDeEjes.setCoordinate( 8, new Point3f( 0.1f, 0.9f, 0.1f));**
115. **LineasDeEjes.setCoordinate( 9, new Point3f(-0.1f, 0.9f, 0.1f));**
116. **LineasDeEjes.setCoordinate(10, new Point3f( 0.1f, 0.9f,-0.1f));**
117. **LineasDeEjes.setCoordinate(11, new Point3f(-0.1f, 0.9f,-0.1f));**
118. **LineasDeEjes.setCoordinate(12, new Point3f( 0.0f, 0.0f,-1.0f));**
119. **LineasDeEjes.setCoordinate(13, new Point3f( 0.0f, 0.0f, 1.0f));**
120. **LineasDeEjes.setCoordinate(14, new Point3f( 0.1f, 0.1f, 0.9f));**
121. **LineasDeEjes.setCoordinate(15, new Point3f(-0.1f, 0.1f, 0.9f));**
122. **LineasDeEjes.setCoordinate(16, new Point3f( 0.1f,-0.1f, 0.9f));**
123. **LineasDeEjes.setCoordinate(17, new Point3f(-0.1f,-0.1f, 0.9f));**
124. **LineasDeEjes.setCoordinateIndex( 0, 0);**
125. **LineasDeEjes.setCoordinateIndex( 1, 1);**
126. **LineasDeEjes.setCoordinateIndex( 2, 2);**
127. **LineasDeEjes.setCoordinateIndex( 3, 1);**
128. **LineasDeEjes.setCoordinateIndex( 4, 3);**
129. **LineasDeEjes.setCoordinateIndex( 5, 1);**
130. **LineasDeEjes.setCoordinateIndex( 6, 4);**
131. **LineasDeEjes.setCoordinateIndex( 7, 1);**
132. **LineasDeEjes.setCoordinateIndex( 8, 5);**
133. **LineasDeEjes.setCoordinateIndex( 9, 1);**
134. **LineasDeEjes.setCoordinateIndex(10, 6);**
135. **LineasDeEjes.setCoordinateIndex(11, 7);**
136. **LineasDeEjes.setCoordinateIndex(12, 8);**
137. **LineasDeEjes.setCoordinateIndex(13, 7);**
138. **LineasDeEjes.setCoordinateIndex(14, 9);**
139. **LineasDeEjes.setCoordinateIndex(15, 7);**
140. **LineasDeEjes.setCoordinateIndex(16,10);**
141. **LineasDeEjes.setCoordinateIndex(17, 7);**
142. **LineasDeEjes.setCoordinateIndex(18,11);**
143. **LineasDeEjes.setCoordinateIndex(19, 7);**
144. **LineasDeEjes.setCoordinateIndex(20,12);**
145. **LineasDeEjes.setCoordinateIndex(21,13);**
146. **LineasDeEjes.setCoordinateIndex(22,14);**
147. **LineasDeEjes.setCoordinateIndex(23,13);**
148. **LineasDeEjes.setCoordinateIndex(24,15);**
149. **LineasDeEjes.setCoordinateIndex(25,13);**
150. **LineasDeEjes.setCoordinateIndex(26,16);**
151. **LineasDeEjes.setCoordinateIndex(27,13);**
152. **LineasDeEjes.setCoordinateIndex(28,17);**
153. **LineasDeEjes.setCoordinateIndex(29,13);**
154. **return LineasDeEjes;**
155. **} // fin de la creacion de los ejes createGeometry()**
156. **} // fin de la clase Ejes**
157. **import javax.vecmath.\*;**
158. **class ConstantesDeColores{**
159. **public static final Color3f red = new**
160. **Color3f(1.0f,0.0f,0.0f);**
161. **public static final Color3f green = new**
162. **Color3f(0.0f,1.0f,0.0f);**
163. **public static final Color3f blue = new**
164. **Color3f(0.0f,0.0f,1.0f);**
165. **public static final Color3f yellow = new**
166. **Color3f(1.0f,1.0f,0.0f);**
167. **public static final Color3f cyan = new**
168. **Color3f(0.0f,1.0f,1.0f);**
169. **public static final Color3f magenta = new**
170. **Color3f(1.0f,0.0f,1.0f);**
171. **public static final Color3f white = new**
172. **Color3f(1.0f,1.0f,1.0f);**
173. **public static final Color3f black = new**
174. **Color3f(0.0f,0.0f,0.0f);**
175. **}**
176. **import java.applet.Applet;**
177. **import java.awt.BorderLayout;**
178. **import java.awt.Frame;**
179. **import java.awt.event.\*;**
180. **import com.sun.j3d.utils.applet.MainFrame;**
181. **import com.sun.j3d.utils.universe.\*;**
182. **import com.sun.j3d.utils.geometry.ColorCube;**
183. **import javax.media.j3d.\*;**
184. **import javax.vecmath.\*;**
185. **import Axis;**
186. **public class EjesEjemApp extends Applet {**
187. **/////////////////////////////////////////////////**
188. **//**
189. **// crear la grafica de la escena**
190. **//**
191. **public BranchGroup createSceneGraph() {**
192. **BranchGroup objRaiz = new BranchGroup();**
193. **objRaiz.addChild(new Axis());**
194. **// Crea el grupo de transformacion, inicializa la matriz como indentidad**
195. **// Abilita la capacidad de modificar el comportamiento de las geometrias**
196. **// con TRANSFORM\_WRITE, y la incorpora a ;a raiz**
197. **TransformGroup objSpin = new TransformGroup();**
198. **objSpin.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);**
199. **// Crear un nuevo objeto Behavior con el ual que puede manipular**
200. **// las transformaciones de un objeto e incorporarlo a la escena**
201. **Transform3D yAxis = new Transform3D();**
202. **Alpha rotationAlpha = new Alpha(-1, 4000);**
203. **RotationInterpolator rotator =**
204. **new RotationInterpolator(rotationAlpha, objSpin);**
205. **BoundingSphere bounds =**
206. **new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);**
207. **rotator.setSchedulingBounds(bounds);**
208. **Transform3D trans = new Transform3D();**
209. **trans.set(new Vector3f(0.5f, 0.0f, 0.0f));**
210. **TransformGroup objTrans = new TransformGroup(trans);**
211. **objRaiz.addChild(objSpin);**
212. **objSpin.addChild(objTrans);**
213. **objSpin.addChild(rotator);**
214. **objTrans.addChild(new ColorCube(0.1));**
215. **// optimiza el desplegue de la escena.**
216. **objRaiz.compile();**
217. **return objRaiz;**
218. **} // fin del metodo CreateSceneGraph perteneciente a EjesEjemApp**
219. **// Crear la escena y an~adirla al universo virtual**
220. **public EjesEjemApp() {**
221. **setLayout(new BorderLayout());**
222. **Canvas3D canvas3D = new Canvas3D(null);**
223. **add("Center", canvas3D);**
224. **BranchGroup escena = createSceneGraph();**
225. **// SimpleUniverse**
226. **SimpleUniverse simpleU = new SimpleUniverse(canvas3D);**
227. **// Muestra a los objetos de la escena**
228. **simpleU.getViewingPlatform().setNominalViewingTransform();**
229. **simpleU.addBranchGraph(escena);**
230. **} // fin de la constructor EjesEjemApp**
231. **// Para que corracomo aplicacion**
232. **public static void main(String[] args) {**
233. **Frame cuadro = new MainFrame(new EjesEjemApp(), 256, 256);**
234. **} // fin del metodo principal EjesEjemApp**
235. **} // fin de la clase EjesEjemApp**
236. **\***
237. **\* Demuestra que al utilizar atributos de Apariencia**
238. **\* unicamente se despliegan las lienea de una geometria**
239. **\*/**
240. **import java.applet.Applet;**
241. **import java.awt.BorderLayout;**
242. **import java.awt.Frame;**
243. **import java.awt.event.\*;**
244. **import com.sun.j3d.utils.applet.MainFrame;**
245. **import com.sun.j3d.utils.universe.\*;**
246. **import javax.media.j3d.\*;**
247. **import javax.vecmath.\*;**
248. **public class YoyoDeLin extends Applet {**
249. **/////////////////////////////////////////////////**
250. **//**
251. **// crea la rama escena**
252. **//**
253. **public class Yoyo extends Shape3D{**
254. **////////////////////////////////////////////**
255. **//**
256. **// Crea un objeto Shape3D con geometria y apariencia**
257. **// La geometria es creada en el metodo GeometriaYoyo**
258. **// La apariencia es creada en el objeto AparienciaYoyo**
259. **//**
260. **public Yoyo() {**
261. **this.setGeometry(GeometriaYoyo());**
262. **this.setAppearance(AparienciaYoyo());**
263. **} // fin del constructor Yoyo**
264. **////////////////////////////////////////////**
265. **//**
266. **// crea la geometria del yoyo**
267. **// cuatro triangulos representan al yoyo**
268. **// strip indices\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**
269. **// 0 0N+0 a 1N+0 ( 0 a N )**
270. **// 1 1N+1 a 2N+1**
271. **// 2 2N+2 a 3N+2**
272. **// 3 3N+4 a 4N+3**
273. **//**
274. **private Geometry GeometriaYoyo() {**
275. **TriangleFanArray tfa;**
276. **int N = 17;**
277. **int totalN = 4\*(N+1);**
278. **Point3f coords[] = new Point3f[totalN];**
279. **int stripCounts[] = {N+1, N+1, N+1, N+1};**
280. **float r = 0.6f;**
281. **float w = 0.4f;**
282. **int n;**
283. **double a;**
284. **float x, y;**
285. **// conjunto de puntos para los triangulos**
286. **coords[0\*(N+1)] = new Point3f(0.0f, 0.0f, w);**
287. **coords[1\*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);**
288. **coords[2\*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);**
289. **coords[3\*(N+1)] = new Point3f(0.0f, 0.0f, -w);**
290. **for(a = 0,n = 0; n < N; a = 2.0\*Math.PI/(N-1) \* ++n){**
291. **x = (float) (r \* Math.cos(a));**
292. **y = (float) (r \* Math.sin(a));**
293. **coords[0\*(N+1)+n+1] = new Point3f(x, y, w);**
294. **coords[1\*(N+1)+N-n] = new Point3f(x, y, w);**
295. **coords[2\*(N+1)+n+1] = new Point3f(x, y, -w);**
296. **coords[3\*(N+1)+N-n] = new Point3f(x, y, -w);**
297. **}**
298. **tfa = new TriangleFanArray (totalN,**
299. **TriangleFanArray.COORDINATES,**
300. **stripCounts);**
301. **tfa.setCoordinates(0, coords);**
302. **return tfa;**
303. **} // fin del metodo GeometriaYoyo**
304. **////////////////////////////////////////////**
305. **//**
306. **// crea la apariencia del yoyo**
307. **//**
308. **private Appearance AparienciaYoyo () {**
309. **Appearance appearance = new Appearance();**
310. **PolygonAttributes AtribPolig = new PolygonAttributes();**
311. **AtribPolig.setPolygonMode(PolygonAttributes.POLYGON\_LINE);**
312. **appearance.setPolygonAttributes(AtribPolig);**
313. **return appearance;**
314. **} // fin del metodo AparienciaYoyo**
315. **} // fin de la clase Yoyo**
316. **/////////////////////////////////////////////////**
317. **//**
318. **// crea la rama escena**
319. **//**
320. **public BranchGroup createSceneGraph() {**
321. **BranchGroup objRaiz = new BranchGroup();**
322. **// Crea el grupo de transformacion, inicializa la matriz como indentidad**
323. **// Abilita la capacidad de modificar el comportamiento de las geometrias**
324. **// con TRANSFORM\_WRITE, y la incorpora a ;a raiz**
325. **TransformGroup objSpin = new TransformGroup();**
326. **objSpin.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);**
327. **objRaiz.addChild(objSpin);**
328. **objSpin.addChild(new Yoyo());**
329. **// Crea un nuevo objeto de Compotamiento para implementar la**
330. **// transformacion que se desee en cualquir obkjeto**
331. **// an~adiendolo despues a la escena**
332. **Transform3D yAxis = new Transform3D();**
333. **Alpha rotacionAlfa = new Alpha(-1, 4000);**
334. **RotationInterpolator rotacion =**
335. **new RotationInterpolator(rotacionAlfa, objSpin);**
336. **BoundingSphere bounds =**
337. **new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);**
338. **rotacion.setSchedulingBounds(bounds);**
339. **objSpin.addChild(rotacion);**
340. **// optimiza el despliege de la escena**
341. **objRaiz.compile();**
342. **return objRaiz;**
343. **} //fin del CreateSceneGraph**
344. **//crea la escena para incorporarla el universo**
345. **public YoyoDeLin() {**
346. **setLayout(new BorderLayout());**
347. **Canvas3D canvas3D = new Canvas3D(null);**
348. **add("Center", canvas3D);**
349. **BranchGroup escena = createSceneGraph();^M**
350. **// SimpleUniverse**
351. **SimpleUniverse simpleU = new SimpleUniverse(canvas3D);**
352. **// objetos de la escena**
353. **simpleU.getViewingPlatform().setNominalViewingTransform();**
354. **simpleU.addBranchGraph(escena);**
355. **} // fin del constructor YoyoDeLin**
356. **// para que corra como aplicacion**
357. **public static void main(String[] args) {**
358. **Frame cuadro = new MainFrame(new YoyoDeLin(), 256, 256);**
359. **} // fin del metodo YoyoDeLin**
360. **} // end of class YoyoDeLin**
361. **\* YoyoDePuntos.java demuestra como usar Appearance para dibujar**
362. **\* unicamente los puntos de una geometria**
363. **\*/**
364. **import java.applet.Applet;**
365. **import java.awt.BorderLayout;**
366. **import java.awt.Frame;**
367. **import java.awt.event.\*;**
368. **import com.sun.j3d.utils.applet.MainFrame;**
369. **import com.sun.j3d.utils.universe.\*;**
370. **import javax.media.j3d.\*;**
371. **import javax.vecmath.\*;**
372. **public class YoyoDePuntos extends Applet {**
373. **/////////////////////////////////////////////////**
374. **//**
375. **// crear la rama de la escena**
376. **//**
377. **public class Yoyo extends Shape3D{**
378. **////////////////////////////////////////////**
379. **//**
380. **// crear el objeto3D con geometria y apariencia**
381. **// la geometra es creada en el metodo Geometriayoyo**
382. **// la apariencia es creada en el metodo AparienciaYoyo**
383. **//**
384. **public Yoyo() {**
385. **this.setGeometry(Geometriayoyo());**
386. **this.setAppearance(AparienciaYoyo());**
387. **} // fin del constuctor Yoyo**
388. **////////////////////////////////////////////**
389. **//**
390. **// crear la geometria del yoyo**
391. **// cuatro triangulos vacios representan al yoyo**
392. **// strip indices\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**
393. **// 0 0N+0 a 1N+0 ( 0 a N )**
394. **// 1 1N+1 a 2N+1**
395. **// 2 2N+2 a 3N+2**
396. **// 3 3N+4 a 4N+3**
397. **//**
398. **private Geometry Geometriayoyo() {**
399. **TriangleFanArray tfa;**
400. **int N = 17;**
401. **int totalN = 4\*(N+1);**
402. **Point3f coords[] = new Point3f[totalN];**
403. **int stripCounts[] = {N+1, N+1, N+1, N+1};**
404. **float r = 0.6f;**
405. **float w = 0.4f;**
406. **int n;**
407. **double a;**
408. **float x, y;**
409. **// conjunto de puntos que defien los cuatro triangulos**
410. **coords[0\*(N+1)] = new Point3f(0.0f, 0.0f, w);**
411. **coords[1\*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);**
412. **coords[2\*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);**
413. **coords[3\*(N+1)] = new Point3f(0.0f, 0.0f, -w);**
414. **for(a = 0,n = 0; n < N; a = 2.0\*Math.PI/(N-1) \* ++n){**
415. **x = (float) (r \* Math.cos(a));**
416. **y = (float) (r \* Math.sin(a));**
417. **coords[0\*(N+1)+n+1] = new Point3f(x, y, w);**
418. **coords[1\*(N+1)+N-n] = new Point3f(x, y, w);**
419. **coords[2\*(N+1)+n+1] = new Point3f(x, y, -w);**
420. **coords[3\*(N+1)+N-n] = new Point3f(x, y, -w);**
421. **}**
422. **tfa = new TriangleFanArray (totalN,**
423. **TriangleFanArray.COORDINATES,**
424. **stripCounts);**
425. **tfa.setCoordinates(0, coords);**
426. **return tfa;**
427. **} // fin del metodo Geometriayoyo**
428. **////////////////////////////////////////////**
429. **//**
430. **// crear apariencia yoyo**
431. **//**
432. **private Appearance AparienciaYoyo () {**
433. **Appearance apariencia = new Appearance();**
434. **PolygonAttributes poligAtribut = new PolygonAttributes();**
435. **poligAtribut.setPolygonMode(PolygonAttributes.POLYGON\_POINT);**
436. **apariencia.setPolygonAttributes(poligAtribut);**
437. **return apariencia;**
438. **} // fin metodo AparienciaYoyo**
439. **} // fin de la clase Yoyo**
440. **/////////////////////////////////////////////////**
441. **//**
442. **// crear escena**
443. **//**
444. **public BranchGroup createSceneGraph() {**
445. **BranchGroup objRaiz = new BranchGroup();**
446. **// Crea el grupo de transformacion, inicializa la matriz como indentidad**
447. **// Abilita la capacidad de modificar el comportamiento de las geometrias**
448. **// con TRANSFORM\_WRITE, y la incorpora a ;a raiz**
449. **TransformGroup objSpin = new TransformGroup();**
450. **objSpin.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);**
451. **objRaiz.addChild(objSpin);**
452. **objSpin.addChild(new Yoyo());**
453. **// Crea un nuevo objeto de Compotamiento para implementar la**
454. **// transformacion que se desee en cualquir obkjeto**
455. **// an~adiendolo despues a la escena**
456. **Transform3D yAxis = new Transform3D();**
457. **Alpha rotacionAlfa = new Alpha(-1, 4000);**
458. **RotationInterpolator rotacion =**
459. **new RotationInterpolator(rotacionAlfa, objSpin);**
460. **BoundingSphere frontera =**
461. **new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);**
462. **rotacion.setSchedulingBounds(frontera);**
463. **objSpin.addChild(rotacion);**
464. **// optimiza el despliege de la escena**
465. **objRaiz.compile();**
466. **return objRaiz;**
467. **} // fin del metodo CreateSceneGraph**
468. **// Crea la escena y la liga al universo virtual**
469. **public YoyoDePuntos() {**
470. **setLayout(new BorderLayout());**
471. **Canvas3D canvas3D = new Canvas3D(null);**
472. **add("Center", canvas3D);**
473. **BranchGroup escena = createSceneGraph();**
474. **// SimpleUniverse**
475. **SimpleUniverse simpleU = new SimpleUniverse(canvas3D);**
476. **// objetos de la escena**
477. **simpleU.getViewingPlatform().setNominalViewingTransform();**
478. **simpleU.addBranchGraph(escena);**
479. **} // fin del constructor YoyoDePuntos**
480. **// para que corra como aplicacion**
481. **public static void main(String[] args) {**
482. **Frame cuadro = new MainFrame(new YoyoDePuntos(), 256, 256);**
483. **} // end of main method of YoyoDePuntos**
484. **} // fin de la clase YoyoDePuntos**
485. **\***
486. **\* Este programa demuestra el back face culling, como una superficie continua**
487. **\* La clase Strip crea una superficie utilizando el metodo TriangleStripArray.**
488. **\* Esta superficie es se ubica dentro de la escena con el RotationInterpolator**
489. **\* utilizando los strip spins. Cuando estos strip spins estan atraz de las**
490. **\* caras de cada triangulo, entonces desaparecen.**
491. **\***
492. **\* Para crear un strip torcido, se utiliza la clase (Torcido).**
493. **\* Este objeto es dibujado como un triangulo relleno.**
494. **\* Un segundo objeto torcer visual se hace utilizando la misma geometria**
495. **\* pero con distinta Appearance pcra asi dibujar unicamente las lineas.**
496. **\* Al utilizar unicamente la linea, ayuda a ver donde esta cada uno de los**
497. **\* poligonos.**
498. **\*/**
499. **import java.applet.Applet;**
500. **import java.awt.BorderLayout;**
501. **import java.awt.Frame;**
502. **import java.awt.event.\*;**
503. **import com.sun.j3d.utils.applet.MainFrame;**
504. **import com.sun.j3d.utils.universe.\*;**
505. **import javax.media.j3d.\*;**
506. **import javax.vecmath.\*;**
507. **public class TriangTorcidos extends Applet {**
508. **/////////////////////////////////////////////////**
509. **//**
510. **// creae un objeto visual llamado Torcido**
511. **//**
512. **public class Torcido extends Shape3D{**
513. **////////////////////////////////////////////**
514. **//**
515. **// crear la subgrafica torcer**
516. **//**
517. **public Torcido() {**
518. **this.setGeometry(createGeometry());**
519. **this.setAppearance(createAppearance());**
520. **} // fin del constructor torcer**
521. **Geometry createGeometry(){**
522. **TriangleStripArray torcerLamina;**
523. **Color3f azul = new Color3f(0.0f, 0.0f, 1.0f);**
524. **// crear el triangulo strip para el torcer**
525. **int N = 80;**
526. **int ContadorPlieges[] = {N};**
527. **torcerLamina = new TriangleStripArray(N,**
528. **TriangleStripArray.COORDINATES | TriangleStripArray.COLOR\_3,**
529. **ContadorPlieges**
530. **);**
531. **double a;**
532. **int v;**
533. **for(v = 0, a=0.0; v < N; v+=2, a=v\*2.0\*Math.PI/(N-2)){**
534. **torcerLamina.setCoordinate(v, new Point3d(**
535. **0.7\*Math.sin(a)+0.2\*Math.cos(a),**
536. **0.3\*Math.sin(a),**
537. **0.7\*Math.cos(a)+0.2\*Math.cos(a)));**
538. **torcerLamina.setCoordinate(v+1, new Point3d(**
539. **0.7\*Math.sin(a)-0.2\*Math.cos(a),**
540. **-0.3\*Math.sin(a),**
541. **0.7\*Math.cos(a)-0.2\*Math.cos(a)));**
542. **torcerLamina.setColor(v, azul);**
543. **torcerLamina.setColor(v+1, azul);**
544. **}**
545. **return torcerLamina;**
546. **}**
547. **// crear la Apariencia para el Torcido Strip**
548. **//**
549. **// Este metodo crea la Appearance para el torcer strip.**
550. **// Las lineas comentadas contienen el setCullFace con el que**
551. **// se puede resolver el problema de que desaparecen las Torcidoed Strip**
552. **Appearance createAppearance(){**
553. **Appearance torcerAparien = new Appearance();**
554. **PolygonAttributes AtribPolig = new PolygonAttributes();**
555. **// AtribPolig.setCullFace(PolygonAttributes.CULL\_NONE);**
556. **torcerAparien.setPolygonAttributes(AtribPolig);**
557. **return torcerAparien;**
558. **}**
559. **} // fin de la clase Torcido**
560. **/////////////////////////////////////////////////**
561. **//**
562. **// crear el grupo escena y su rama**
563. **//**
564. **public BranchGroup createSceneGraph() {**
565. **BranchGroup contenidoRaiz = new BranchGroup();**
566. **// Crear el nodo de transformacion, inicializando en la mat. indentidad**
567. **// An~ade esta e la raiz**
568. **TransformGroup objSpin = new TransformGroup();**
569. **objSpin.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);**
570. **contenidoRaiz.addChild(objSpin);**
571. **Shape3D torcer = new Torcido();**
572. **objSpin.addChild(torcer);**
573. **// Dublica la geometria de torcer strip y el conjunto de apariencia**
574. **// para el nuevo Objeto3D que forma una linea.**
575. **// An~ade POLYGON\_FILLED y POLYGON\_LINE a la grafica de la escena**
576. **// en un mismo punto.**
577. **// Esto permite mostrar los triangulos de la figura de Mobius cuando**
578. **// son clipped.**
579. **// El PolygonOffset es un conjunto que prevee el stitching.**
580. **//**
581. **PolygonAttributes AtribPolig = new PolygonAttributes();**
582. **AtribPolig.setCullFace(PolygonAttributes.CULL\_NONE);**
583. **AtribPolig.setPolygonMode(PolygonAttributes.POLYGON\_LINE);**
584. **AtribPolig.setPolygonOffset(0.001f);**
585. **Appearance polyAppear = new Appearance();**
586. **polyAppear.setPolygonAttributes(AtribPolig);**
587. **objSpin.addChild(new Shape3D(torcer.getGeometry(), polyAppear));**
588. **Alpha rotacionAlfa = new Alpha(-1, 16000);**
589. **RotationInterpolator rotacion =**
590. **new RotationInterpolator(rotacionAlfa, objSpin);**
591. **// Crea una esfera que marca la region en donde se podra controlar**
592. **// el comportamiento, la esfera se localiza en el origen**
593. **// con un radio de 1**
594. **BoundingSphere fronteras = new BoundingSphere();**
595. **rotacion.setSchedulingBounds(fronteras);**
596. **objSpin.addChild(rotacion);**
597. **// hace el fonde color blanco**
598. **Background fondo = new Background(1.0f, 1.0f, 1.0f);**
599. **fondo.setApplicationBounds(fronteras);**
600. **contenidoRaiz.addChild(fondo);**
601. **// Optimiza el desplegue de la escena**
602. **contenidoRaiz.compile();**
603. **return contenidoRaiz;**
604. **} // fin del metodo CreateSceneGraph de TriangTorcidos**
605. **// Crea una escena y la pega a la raiz**
606. **public TriangTorcidos() {**
607. **setLayout(new BorderLayout());**
608. **Canvas3D canvas3D = new Canvas3D(null);**
609. **add("Center", canvas3D);**
610. **BranchGroup escena = createSceneGraph();**
611. **// SimpleUniverse es una clase de Utility**
612. **SimpleUniverse simpleU = new SimpleUniverse(canvas3D);**
613. **// Obtiene el tipo de plataforma en que se trabaja**
614. **simpleU.getViewingPlatform().setNominalViewingTransform();**
615. **simpleU.addBranchGraph(escena);**
616. **} // fin del constructor TriangTorcidos**
617. **// Para permitir correrlo como aplicacion**
618. **public static void main(String[] args) {**
619. **System.out.println("TriangTorcidos ");**
620. **System.out.println("Este programa demuestra caras ocultas.");**
621. **System.out.println("Se tienen dos objetos visuales rotados.");**
622. **System.out.println("uno tipo malla y otro solido.");**
623. **System.out.println("El mallado solo muestra las partes que");**
624. **System.out.println("componen al objeto solido");**
625. **Frame ventana = new MainFrame(new TriangTorcidos(), 256, 256);**
626. **} // fin del metodo TriangTorcidos**
627. **} // fin de la clase TriangTorcidos**