Tutorial de JAVA 3D

Orlando Soto
Facultad de Ciencia
L.C.C

Indice

        - Clases Vector
        - Clases TexCoord
    - Clases Geometry
        - Clase GeometryArray
        - Paso 1: Construcción de un objeto GeometryArray vacío
        - Paso 2: Rellenar con Datos el Objeto GeometryArray
        - Paso 3: Hacer que los Objetos Shape3D Referencien a los Objetos GeometryArray
        - Subclases de GeometryArray
        - Subclases de GeometryStripArray
        - Subclases de IndexedGeometryArray
        - Axis.java es un ejemplo de IndexedGeometryArray
    - Atributos y Apariencia
        - NodeComponent Appearance
        - Compartir Objetos NodeComponent
        - Clases Attribute
        - Ejemplo: Recortar la cara trasera
- Crear Contenidos Sencillos en Java 3D
    - Cargadores
        - Ejemplo de Usos de un Loader
        - Cargadores Disponibles Públicamente
        - Interfaces y Clases Base del Paquete Loader
        - Escribir un Loader
    - GeometryInfo
        - Sencillo Ejemplo de GeometryInfo
        - Clases para GeometryInfo
    - Texto 2D
        - Ejemplo de Text2D
        - Clases Usadas para Crear Objetos Text2D
    - Texto 3D
        - Ejemplo de Text3D
        - Clases Usadas en la Creación de Objetos Text3D
    - Fondo
        - Ejemplos de fondos
        - La Clase BoundingLeaf
    - Datos de Usuario
- Interacción en Java 3D
    - Comportamiento: la Base para Interacción y Animación
        - Aplicaciones de Behavior
        - Introducción a la clases Behavior
    - Behavior Básico
        - Escribir una Clase Behavior
        - Usar una Clase Behavior
        - API de la Clase Behavior
    - Condiciones de Disparo: Cómo se Disparan los Comportamientos
        - WakeupCondition
        - WakeupCriterion

        - Clases WakeupCriterion Específicas
        - WakeupCondition Composition
    - Clases de Comportamientos Útiles para la Navegación por Teclado
        - Programa de Ejemplo de KeyNavigatorBehavior
        - Clases KeyNavigatorBehavior y KeyNavigator
    - Clases de Utilidad para Interactuar con el Ratón
        - Usar las Clases de Comportamiento del Ratón
        - Fundamentos del Comportamiento del Ratón
        - Clases Específicas de Comportamientos de Ratón
        - MouseNavigation
    - Picking
        - Usar las Clases de Utilidad de Picking
        - El API Corazón de Clases Picking de Java 3D
        - Clases Generales del Paquete Picking
        - Clases de Comportamientos Picking Específicas
- Animación en Java 3D
    - Los Interpoladores y los Objetos Alpha Proporcionan Animaciones Basadas en el Tiempo
        - Alpha
        - Usar Objetos Interpolator y Alpha
        - Ejemplo de uso de Alpha y RotationInterpolator
        - El API Alpha
        - Clases de Comportamiento Interpolator
        - API Corazón de Interpolator
        - Clases PathInterpolator
    - La Clase Billboard
        - Usar un Objeto Billboard
        - Programa de Ejemplo de Billboard
        - El API Billboard
    - Animaciones de Nivel de Detalle (LOD)
        - Usar un Objeto DistanceLOD
        - Ejemplo de uso de DistanceLOD
        - El API DistanceLOD
        - API de LOD (Level of Detail)
    - Morph
        - Usar un Objeto Morph
        - Ejemplo de Aplicación Morph: Walking
        - El API Morph
- Iluminación en Java 3D
    - Sombreado en Java 3D
    - Receta para Iluminar Objetos Visuales
        - Ejemplos de Luces Sencillas
        - Dónde Añadir un Objeto Light en un Escenario Gráfico
    - Clase Light
        - Luz Ambiente
        - Luz Direccional
        - Punto de Luz

        - SpotLight
        - Aplicaciones de Fuentes de Luz
        - Ejemplos de Iluminación
    - Objetos Material
        - Ejemplos sencillos de Material
        - Propiedades Geometry color, ColoringAttributes, y Material
    - Superficies Normales
    - Especificar la Influencia de las Luces
        - Alternativa a los Límites de Influencia: BoundingLeaf
        - Ámbito de Límites de Influencia de las Luces
    - Crear Objetos Brillantes-en-la-Oscuridad, Sombras y Otros Problemas de Iluminación
        - Objetos Brillantes-en-la-Oscuridad
        - Calcular Sombras
        - Crear Sombras
        - Programa de Ejemplo de Sombras
        - Tópico Avanzado: El Papel del Objeto View en el Sombreado
- Texturas en Java 3D
    - ¿Qué es el Texturado?
    - Texturado Básico
        - Sencilla Receta de Texturado
        - Sencillos Ejemplos de Programas de Textura
        - Más sobre las Coordenadas de Textura
        - Preview de Algunas Opciones de Texturado
        - Opciones de Textura
        - Texture3d
    - Algunas Aplicaciones de Texturado
        - Texturado de Geométricos Primitivos
        - Texturado de Líneas
        - Usar Texturas Text2D
    - Atributos de Textura
        - Modo de Textura
        - Textura con Color de Mezcla
        - Modo de Corrección de Perspectiva
        - Transformación del Mapeo de Textura
        - API TextureAttributes
    - Generación Automática de Coordenadas de Textura
        - Formato de Generación de Textura
        - Modo de Generación de Textura
        - Cómo usar un Objeto TexCoordGeneration
        - API TexCoordGeneration
    - Múltiples Niveles de Textura (Mipmaps)
        - ¿Qué es el Texturado Multi-Nivel (MIPmap)?
        - Ejemplos de Texturas Multi-Nivel
        - Filtros de Reducción para Múltiples Niveles de Textura
        - Modo Mipmap
    - API de Texture, Texture2D, y Texture3d

        - Filtros de Reducción y Ampliación
        - API Texture
        - API de Texture2D
        - API de Texture3d
    - API de TextureLoader y NewTextureLoader
        - API de TextureLoader
        - API de NewTextureLoader

Java 3D

Introducción

El API Java 3D es un interface para escribir programas que muestran e interactuan con gráficos tridimensionales. Java 3D es una extensión estándard del JDK 2 de Java. El API Java 3D proporciona una colección de constructores de alto-nivel para crear y manipular geometrías 3D y estructuras para dibujar esta geometría. Java 3D proporciona las funciones para creación de imágenes, visualizaciones, animaciones y programas de aplicaciones gráficas 3D interactivas.

¿Qué es el API 3D de Java?

El API 3D de Java es un árbol de clases Java que sirven como interface para sistemas de renderizado de gráficos tridimensionales y un sistema de sonido. El programador trabaja con constructores de alto nivel para crear y manipular objetos geométricos en 3D. Estos objetos geométricos residen en un universo virtual, que luego es renderizado. El API está diseñado con flexibilidad para crear universos virtuales precisos de una ámplia variedad de tamaños, desde astronómicos a subatómicos.
A pesar de toda esta funcionalidad, el API es sencillo de usar. Los detalles de renderizado se manejan automáticamente. Aprovechándose de los Threads Java, el renderizador Java 3D es capaz de renderizar en paralelo. El renderizador también puede optimizarse automáticamente para mejorar el rendimiento del renderizado.
Un programa Java 3D crea ejemplares de objetos Java 3D y los sitúa en un estructura de datos de escenario gráfico. Este escenario gráfico es una composición de objetos 3D en una estructura de árbol que especifica completamente el contenido de un universo virtual, y cómo va a ser renderizado.

Los programas Java 3D pueden escribirse para ser ejecutados como aplicaciones solitarias o como applets en navegadores que hayan sido extendidos para soportar Java 3D, o ámbos.

¿Qué Software se Necesita?

Te aconsejo que te des una vuelta por la Home Page de Java 3D en Sun: http://java.sun.com/products/java-media/3d

El API Java 3D

Todo programa Java 3D está, al menos, parcialmente ensamblado por objetos del árbol de clases Java 3D. Esta colección de objetos describe un universo virtual, que va a ser renderizado. El API define unas 100 clases presentadas en el paquete javax.media.j3d
Hay cientos de campos y métodos en las clases del API Java 3D. Sin embargo, un sencillo universo virtual que incluya animación puede construirse con unas pocas clases. Este capítulo describe un conjunto mínimo de objetos y sus interacciones para renderizar un universo virtual.
Esta página incluye el desarrollo de un sencillo pero completo programa Java 3D, llamado HelloJava3Dd.java, que muestra un cubo giratorio. El programa de ejemplo se desarrolla de forma incremental, y se presenta en varias versiones, para demostrar cada parte del proceso de programación Java 3D.
Además del paquete corazón de Java 3D, se usan otros paquetes para escribir programas Java 3D. Uno de estos paques es com.sun.j3d.utils el que normalmente se referiere como clases de utilidades de Java 3D. El paquete de las clases corazón incluye sólo las clases de menor nivel necesarias en la programación Java 3D.
Las clases de utilidades son adiciones convenientes y poderosas al corazón. Estas clases se dividen en cuatro categorías: cargadores de contenidos, ayudas a

la construcción del escenario gráfico, clases de geometría y utilidades de conveniencia.
Al utilizar las clases de utilidades se reduce significativamente el número de líneas de código en un programa Java 3D. Además de las clases de los paquetes corazón y de utilidades de Java 3D, todo programa 3D usa clases de los paquetes java.awt y javax.vecmath. El paquete java.awt define el "Abstract Windowing Toolkit" (AWT). Las clases AWT crean una ventana para mostrar el renderizado. El paquete javax.vecmath define clases de vectores matemáticos para puntos, vectores, matrices y otros objetos matemáticos.
En el resto del texto, el término **objeto visual** se utilizará para referirnos a un "objeto del escenario gráfico" (por ejemplo, un cubo o una esfera). El término **objeto** sólo se usará para referirse a un ejemplar de una clase. El término **contenido** se usará para referirnos a objetos visuales en un escenario gráfico como un todo.

Construir un Escenario Gráfico

Un universo virtual Java 3D se crea desde un escenario gráfico. Un escenario gráfico se crea usando ejemplares de clases Java 3D. El escenario gráfico está ensamblado desde objetos que definen la geometría, los sonidos, las luces, la localización, la orientación y la apariencia de los objetos visuales y sonoros.
Una definición común de un escenario gráfico es una estructura de datos compuesta de nodos y arcos. Un nodo es un elemento dato y un arco es una relación entre elementos datos. Los nodos en un escenario gráfico son los ejemplares de las clases Java 3D. Los arcos representan dos tipos de relaciones entre ejemplares Java 3D.
La relación más común es padre-hijo. Un nodo **Group** puede tener cualquier número de hijos, pero sólo un padre. Un nodo hoja sólo puede tener un padre y no puede tener hijos. La otra relación es una **referencia**. Una referencia asocia un objeto **NodeComponent** con un nodo del escenario gráfico. Los objetos

NodeComponent definen la geometría y los atributos de apariencia usados para renderizar los objetos visuales.

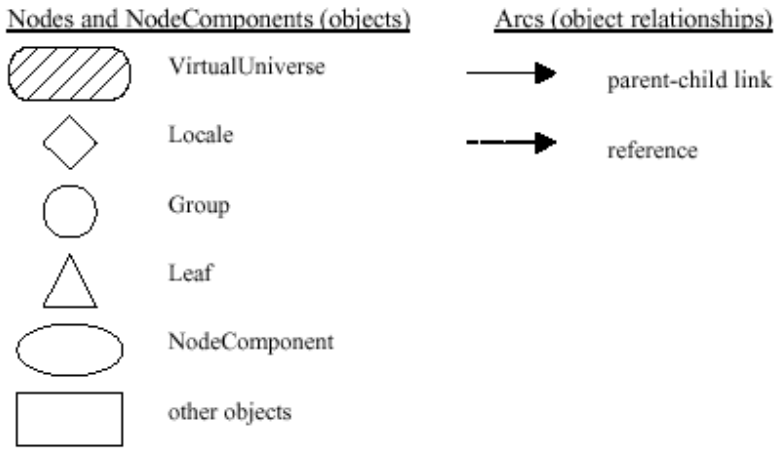
Un escenario gráfico Java 3D está construido de objetos nodos con relaciones padre-hijo formando una estructura de árbol. En una estructura de árbol, un nodo es el raíz. Se peude acceder a otros nodos siguiendo los arcos desde el raíz. Los nodos de un árbol no forman bucles. Un escenario gráfico está formado desde los árboles con raíces en los objetos **Locale**. Los **NodeComponents** y las referencias a arcos no forman parte del escenario gráfico.

Sólo existe un camino desde la raíz de un árbol a cada una de las hojas; por lo tanto, sólo hay un camino desde la raíz hasta el escenario gráfico de cada nodo hoja. El camino desde la raíz de un escenario gráfico hasta una hoja especificada es el camino al escenario gráfico del nodo hoja. Como un camino de un escenario gráfico trata exactamente con un sola hoja, hay un camino de escenario gráfico para cada hoja en el escenario.

Todo camino de escenario gráfico en un escenario gráfico Java 3D especifica completamente la información de estado de su hoja. Esta información incluye, la localización, la orientación y el tamaño del objeto visual. Consecuentemente, los atributos visuales de cada objeto visual dependen sólo de su camino de escenario gráfico. El renderizador Java 3D se aprovecha de este echo y renderiza las hojas en el orden que él determina más eficiente. El programador Java 3D normalmente no tiene control sobre el orden de renderizado de los objetos.

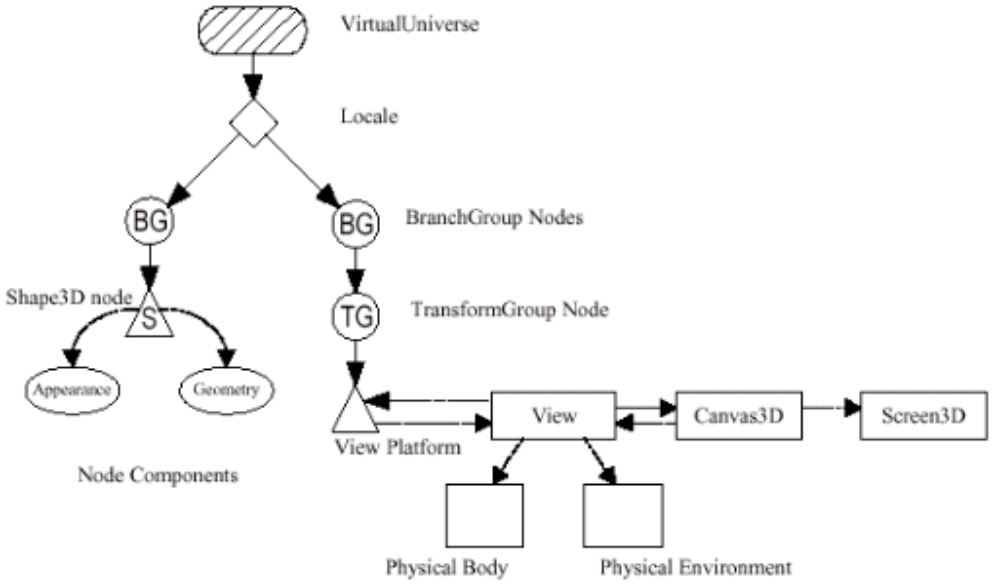
Las representaciones gráficas de un escenario gráfico pueden servir como herramienta de diseño y/o documentación para los programas Java 3D. Los escenarios gráficos se dibujan usando símbolos gráficos estándards como se ve en la Figura 1-1. Los programas Java 3D podrían tener más objetos que los que hay en su escenerio gráfico.

Para diseñar un universo virtual Java 3D se dibuja un escenario gráfico usando un conjunto de símbolos estándard. Después de completar el diseño, este escenerio gráfico es la especificación para el programa. Después de completar el programa, el mismo escenario gráfico es una representación concisa del programa (asumiendo que se siguió la especificación).



Cada uno de los símbolos mostrados al lado izquierdo de la Figura 1.1 representa un sólo objeto cuando se usa en un escenario gráfico. Los dos primeros símbolos representan objetos de clases específicas: **VirtualUniverse** y **Locale**. Lo siguientes tres símbolos de la izquierda representan objetos de las clases **Group**, **Leaf**, y **NodeComponent**. Estos tres símbolos normalmente tienen anotaciones para indicar la subclase del objeto específico. El último símbolo se usa para representar otras clases de objetos.

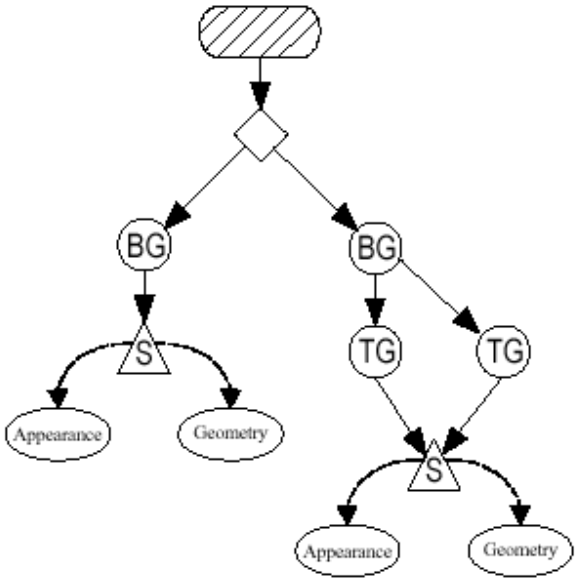
El símbolo de la flecha sólida representa una relación padre-hijo entre dos objetos. La flecha punteada es una referencia a otro objeto. Los objetos referenciados pueden ser compartidos entre diferentes ramas de una escenario gráfico. En la Figura 1-2, podemos ver un sencillo escenario gráfico:



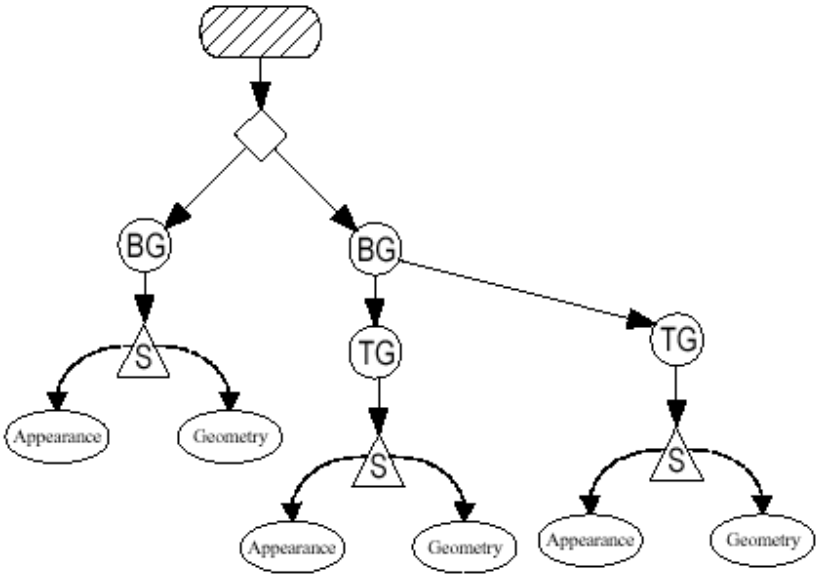
Es posible crear un escenario gráfico ilegal. Podemos ver uno en la Figura 1-3. Este escenario es ilegal porque viola las propiedades de un DAG. El problema son los dos objetos **TransformGroup** que tienen al mismo objeto **Shape3D** como hijo. Recuerda que una hoja sólo puede tener un padre. En otras palabras, sólo puede haber un camino desde el objeto **Locale** hasta la hoja (o un camino desde la hoja hasta el objeto **Loale**).

Podríamos pensar que la estructura mostrada en la figura 1-3 define tres objetos visuales en un universo virtual. Pero el escenario gráfico define dos objetos visuales que re-usan el objeto visual (Shape3D) del lado derecho de la figura. Conceptualmente, cada objeto **TransformGroup** que apadrina al ejemplar compartido de **Shape3D** podría situar una imagen en el objeto visual en diferentes localizaciones. Sin embargo, es un escenario gráfico ilegal porque el arco padre-hijo no forma un árbol. En este ejemplo, el resultado es que el objeto **Shape3D** tiene más de un padre.

Las explicaciones del árbol y de las estructuras DAG son correctas. Sin embargo, el sistema de ejecución Java 3D reporta el error en términos de la relación hijo-padre. Un resultado de la limitación de la estructura de árbol es que cada objeto **Shape3D** está limitado a un sólo padre. Para el ejemplo de la Figura 1-3, se lanzará una excepción **'multiple parent'** en el momento de la ejecución. La Figura 1-4, con un padre para cada objeto **Shape3D**, muestra una posible solución para este escenario gráfico.



Un programa Java 3D que define un escenario gráfico ilegal podría compilarse, pero no se renderiza. Cuando se ejecuta un programa Java 3D que define un escenario gráfico ilegal, el sistema Java 3D detecta el problema y lanza una excepción. El programa podría estár ejecutandose y consecuentemente deberíamos pararlo. Sin embargo, no se renderizará ninguna imagen.



Cada escenario gráfico tiene un sólo **VirtualUniverse**. Este objeto tiene una lista de objetos **Locale**. Un objeto **Locale** proporciona una referencia a un punto en el universo virtual. Podemos pensar en los objetos **Locale** como marcas de tierra que determinan la localización de los objetos visuales en el universo virtual.

Es técnicamente posible para un programa Java 3D tener más de un objeto **VirtualUniverse**, y así definir más de un universo virtual. Sin embargo, no hay ninguna forma de comunicación entre los universos virtuales. Además, un objeto de un escenario gráfico no puede existir en más de un universo virtual. Es altamente recomendable usar uno y sólo un ejemplar de **VirtualUniverse** en cada programa Java 3D.

Mientras que un objeto **VirtualUniverse** podría referenciar muchos objetos **Locale**, la mayoría de los programas Java 3D tiene un sólo objeto **Locale**. Cada objeto**Locale** puede servir de raíz para varios sub-gráficos del escenario gráfico. Por ejemplo, si nos referimos a la Figura 1-2 podremos observar las dos ramas sub-gráficas que salen desde el objeto **Locale**.

Un objeto **BranchGroup** es la raíz de un sub-gráfico, o rama gráfica. Hay dos categorias de escenarios sub-gráficos: la rama de vista gráfica y la rama de contenido gráfico. La rama de contenido gráfico especifica el contenido del universo virtual - geometría, apariencia, comportamiento, localización, sonidos y luces. La rama de vista gráfica especifica los parámetros de visualización, como la posición de visualización y la dirección. Juntas, las dos ramas especifican la mayoría del trabajo que el renderizador tiene que hacer.

Árbol de Clases de Alto Nivel del API Java 3D

En la Figura 1-5 podemos ver los tres primeros niveles del árbol de clases del API Java 3D. En esta parte del árbol aparecen las clases **VirtualUniverse**, **Locale**, **Group**, y **Leaf**.

SceneGraphObject es la superclase de casi todas las clases corazón y de utilidad de Java 3D. Tiene dos subclases: **Node** y **NodeComponent**. Las subclases de **Node** proporcionan la mayoría de los objetos de un escenario gráfico. Un objeto **Node** es un objeto nodo **Group** o un objeto nodo **Leaf**.

Clase Node

La clase **Node** es una superclase abstracta de las clases **Group** y **Leaf**. Esta clase

define algunos de los métodos importantes de sus subclases. Las subclases de **Node** componen escenarios gráficos.

Clase Group

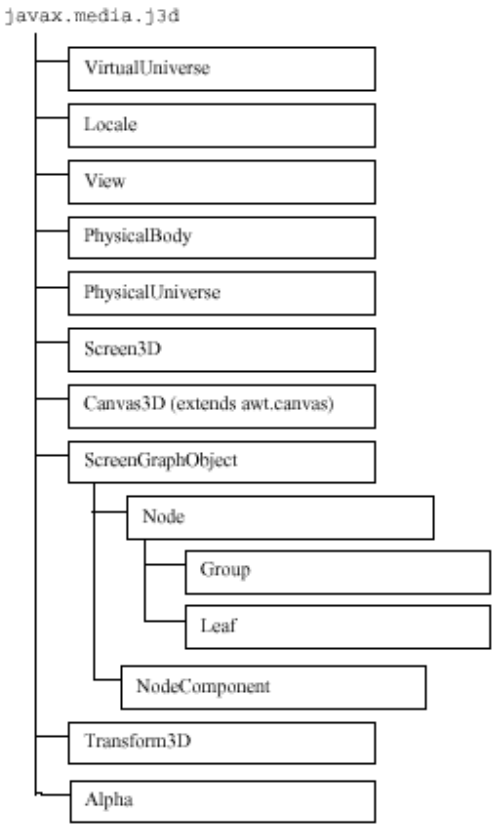
La clase **Group** es la superclase usada en especificación de localización y orientación de objetos visuales en el universo virtual. Dos de las subclases de **Group** son: **BranchGroup** y **TransformGroup**. En la representación gráfica de un escenario gráfico, los simbolos de **Group** (círculos) normalmente se anotan con **BG** para **BranchGroups**, **TG** para **TransformGroups**, etc. La Figura 1-2 muestra algunos ejemplos de esto.

Clase Leaf

La clase **Leaf** es la superclase usada para especificar la forma, el sonido y comportamiento de los objetos visuales en el universo virtual. Algunas de las subclases de **Leaf** son: **Shape3D**, **Light**, **Behavior**, y **Sound**. Estos objetos podrían no tener hijos pero podrían referenciar a **NodeComponents**.

Clase NodeComponent

La clase **NodeComponent** es la superclase usada para especificar la geometría, la apariencia, la textura y las propiedades de material de un nodo **Shape3D** (Leaf). Los **NodeComponent**s no forman parte del escenario gráfico, pero son referenciados por él. un **NodeComponent** podría ser referenciado por más de un objeto **Shape3D**.



Receta para Escribir Programas Java 3D

Las subclases de **SceneGraphObject** son los ladrillos que se ensamblan en los escenarios gráficos. La línea básica de desarrollo de un programa Java 3D consiste en siete pasos (a los que la especificación del API Java 3D se referiere como un **Receta**) presentados a continuación. Esta receta puede usarse para ensamblar muchos útiles programas Java 3D.

1. Crear un Objeto **Canvas3D**
2. Crear un objeto **VirtualUniverse**
3. Crear un objeto **Locale**, adjuntarlo al objeto **VirtualUniverse**
4. Construir la rama de vista gráfica
    - Crear un objeto **View**
    - Crear un objeto **ViewPlatform**
    - Crear un objeto **PhysicalBody**
    - Crear un objeto **PhysicalEnvironment**
    - Adjuntar los objetos **ViewPlatform**, **PhysicalBody**, **PhysicalEnvironment**, y **Canvas3D** al objeto **View**
5. Construir la(s) rama(s) gráfica(s) de contenido

6. Compilar la(s) rama(s) gráfica(s)
7. Insertar los subgráficos dentro del objeto **Locale**

Esta receta ignora algunos detalles pero ilustra el concepto fundamental para toda la programación Java 3D: crear la rama gráfica del escenario gráfico es la programación principal. En vez de ampliar esta receta, la siguiente sección explica una forma sencilla de construir un escenario gráfico muy similar con menos programación.

Una Sencilla Receta para Escribir Programas Java 3D

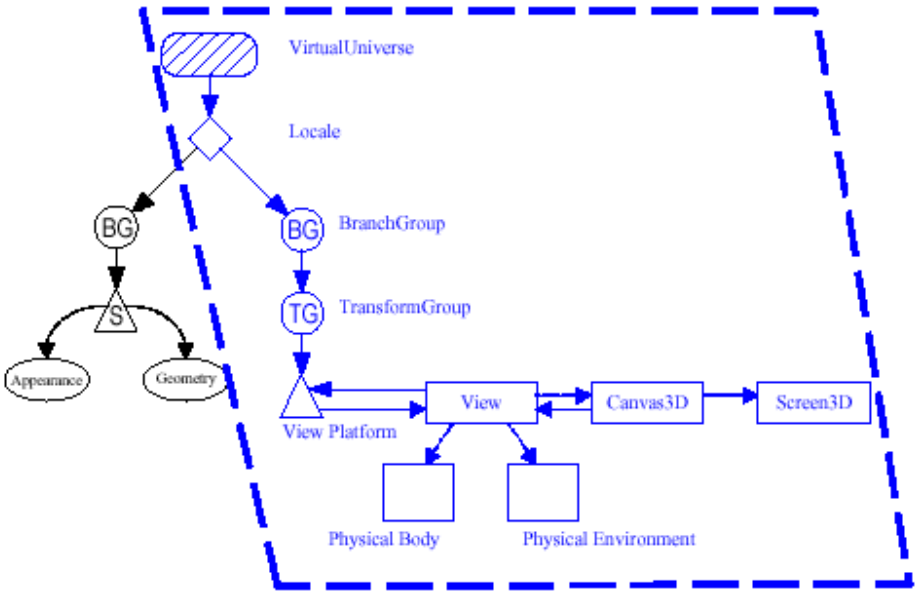
Los programas Java 3D escritos usando la receta básica tienen ramas de vista gráfica con idéntica estructura. La regularidad de la estructura de las ramas de vista gráfica tambien se encuentra en la clase de utilidad **SimpleUniverse**. Los ejemplares de esta clase realizan los pasos 2, 3 y 4 de la receta básica. Usando la clase **SimpleUniverse** en programación Java 3D se reduce significativamente el tiempo y el esfuerzo necesario para crear las ramas de vista gráfica. Consecuentemente, el programador tiene más tiempo para concentrarse en el contenido. Esto de lo que se trada el escribir programas Java 3D.
La clase **SimpleUniverse** es un buen punto de inicio en la programación Java 3D, porque permite al programador ignorar las ramas de vista gráfica. Sin embargo, usar **SimpleUniverse** no permite tener varias vistas de un universo virtual.
La clase **SimpleUniverse** se usa en todos los ejemplos de programación de este tutorial.

La clase SimpleUniverse

El constructor de **SimpleUniverse** crea un escenario gráfico que incluye un objeto **VirtualUniverse** y **Locale**, y una rama de vista gráfica completa. Esta rama gráfica creada usa un ejemplar de las clases de conveniencia **ViewingPlatform** y **Viewer** en lugar de las clases corazón usadas para crear una rama de vista gráfica. Observa que **SimpleUniverse** sólo usa indirectamente los objetos **View** y **ViewPlatform** del corazón Java 3D. Los objetos **SimpleUniverse** suministran la

funcionalidad de todos los objetos que hay dentro del recuadro azul de la Figura 1-7.

El paquete com.sun.j3d.utils.universe contiene **SimpleUniverse**, **ViewingPlatform**, y clases **Viewer** de conveniencia.



Al usar los objetos **SimpleUniverse** la receta básica se simplifica:

1. Crear un objeto **Canvas3D**
2. Crear un objeto **SimpleUniverse** que referencia al objeto **Canvas3D** anterior
   - Personalizar el objeto **SimpleUniverse**
3. Construir la rama de contenido
4. Compilar la rama de contenido gráfico
5. Insertar la rama de contenido gráfico dentro del objeto **Locale** de **SimpleUniverse**

Constructores de SimpleUniverse

Paquete: **com.sun.j3d.utils.universe**

Esta clase configura un entorno de usuario mínimo para obtener rápida y fácilmente un programa Java 3D y ejecutarlo.

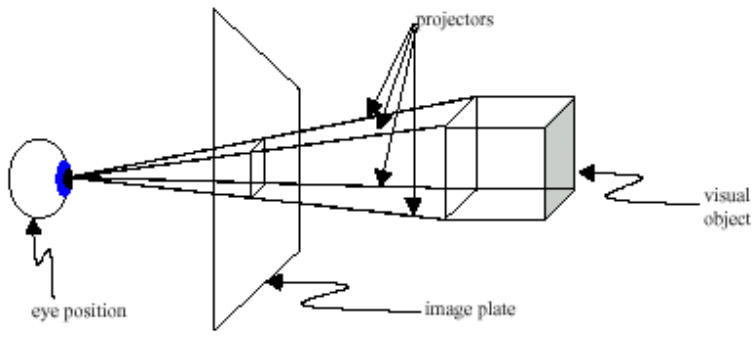
Esta clase de utilidad crea todos los objetos necesarios para la rama de vista gráfica. Especificamente crea los objetos **Locale**, **VirtualUniverse**, **ViewingPlatform**, y **Viewer** (todos con sus valores por defecto). Los objetos tiene las relaciones

apropiadas para formar la rama de vista gráfica.

SimpleUniverse proporciona toda la funcionalidad necesaria para muchas aplicaciones Java 3D básicas. **Viewer** y **ViewingPlatform** son clases de conveniencia. estas clases usan las clases **View** y **ViewPlatform** del corazón Java.

SimpleUniverse()
Construye un sencillo universo virtual.
SimpleUniverse(Canvas3D canvas3D)
Construye un sencillo universo virtual con una referencia al objeto **Canvas3D** nombrado.

El objeto **SimpleUniverse** crea una rama de vista gráfica completa para un universo virtual. Esta rama incluye un plato de imagen. Un plato de imagen es el rectángulo conceptual donde se proyecta el contenido para formar la imagen renderizada. El objeto **Canvas3D**, que proporciona una imagen en una ventana de nuestra pantalla, puede ser el plato de imagen.
La Figura 1-9 muestra la relación entre el plato de imagen, la posición del ojo, y el universo virtual. La posición del ojo está detrás del plato de imagen. Los objetos visuales delante del plato de imagen son renderizados en el plato de imagen. El renderizado puede ser como una proyección de los objetos visuales sobre el plato de imagen. Esta idea se ilustra con los cuatro proyectores de la imagen (líneas punteadas).



Por defecto, el plato de imagen está centrado en el origen de **SimpleUniverse**. La orientación por defecto es mirando hacia abajo el eje Z. Desde esta posición, el

eje X es una línea horizontal que atraviesa el plato de imagen con los valores positivos hacia la derecha. El eje Y es una línea vertical que atraviesa el centro del plato de imagen, con los valores positivos arriba. Consecuentemente, el punto (0,0,0) es el centro del plato de imagen.
Los típicos programas Java 3D mueven la vista hacía atrás (z positivo) para hacer que los objetos se acerquen, al origen dentro de la vista. La clase **SimpleUniverse** tiene un miembro que es un objeto de la clase **ViewingPlatform**. Esta clase tiene un método **setNominalViewingTransform** que selecciona la posición del ojo para que esté centrado en (0, 0, 2.41) buscando en dirección z negativa hacia el origen.

El Método **ViewingPlatform setNominalViewingTransform()**

Paquete: com.sun.j3d.utils.universe

La clase **ViewingPlatform** se usa para configurar la rama de vista gráfica de un escenario gráfico Java 3D en un objeto **SimpleUniverse**. Este método normalmente se usa en conjunción con el método **getViewingPlatform** de la clase **SimpleUniverse**.

void setNominalViewingTransform()

Selecciona la distancia nominal de la vista a una distancia de aproximadamente 2,42 metros en la vista de transformación de un SimpleUniverse. Desde esta distancia y con el campo de vista por defecto, los objetos con 2 metros de altura o de anchura generalmente entran en el plato de imagen.

Después de crear los objetos **Canvas3D** y **SimpleUniverse**, el siguiente paso es la creacción de la rama de contenido gráfico. La regularidad de estructura encontrada en la rama de vista gráfica no existe para la rama de contenido gráfico. La rama de contenido varia de un programa a otro haciendo imposible obtener los detalles de su construcción en una receta. Esto también significa que no hay una clase de "contenido sencillo" para ningún universo que podamos querer ensamblar.

Después de crear la rama de contenido gráfico, se inserta dentro del universeo usando el método **addBranchGraph** de **SimpleUniverse**. Este método toma un ejemplar de **BranchGroup** como único argumento. Este **BranchGroup** se añade como hijo del objeto **Locale** creado por **SimpleUniverse**.

Lista Parcial de Métodos de **SimpleUniverse**

Paquete: com.sun.j3d.utils.universe

void addBranchGraph(BranchGroup bg)

Se usa para añadir Nodos al objeto **Locale** del escenario gráfico creado por el **SimpleUniverse**. Se usa para añadir una rama de contenido gráfico al universo virtual.

ViewingPlatform getViewingPlatform()

Se usa para recuperar el objeto **ViewingPlatform** del **SimpleUniverse** ejemplarizado. Este método se usa con el método **setNominalViewingTransform()** de **ViewingPlatform** para ajustar la localización de la posición de vista.

Alguna Terminología Java 3D

Insertar una rama gráfica dentro de un **Locale** la hace **viva**, y consecuentemente, cada uno de los objetos de esa rama gráfica también están vivos. Hay algunas consecuencias cuando un objeto se convierte en **vivo**. Los objetos vivos estan sujetos s renderización. Los parámetros de los objetos vivos no pueden ser modificados a menos que la capacidad correspondiente haya sido seleccionada especificamente antes de que el objeto esté vivo.

Los objetos **BranchGroup** pueden ser compilados. Compilar un **BranchGroup** lo convierte a él y a todos sus ancestros en una forma más eficiente para el renderizado. Compilar los objetos **BranchGroup** está recomendado como el último paso antes de hacerlo vivir. Es mejor compilar solo los objetos **BranchGroup** insertados dentro de objetos **Locale**

Método **BranchGroup compile()**

void compile()

compila la fuente **BranchGroup** asociada con este objeto creado y cacheando un escenario gráfico compilado.

Los conceptos de **compilado** y **vivo** se implementan en la clase **SceneGraphObject**. Abajo podemos ver los dos métodos de la clase **SceneGraphObject** que se relacionan con estos conceptos.

Lista Parcial de Métodos de **SceneGraphObject**

SceneGraphObject es la superclase usada para crear un escenario gráfico incluyendo **Group**, **Leaf**, y **NodeComponent**. **SceneGraphObject** proporciona varios métodos y campos comunes para sus subclases:

boolean isCompiled()

Devuelve una bandera indicando si el nodo forma parte de un escenario gráfico que ha sido compilado.

boolean isLive()

Devuelve una bandera que indica si el nodo forma parte de un escenario gráfico vivo.

Observa que no hay un paso "Empezar a renderizar" en ninguna de las recetas anteriores. El renderizador Java 3D empieza a funcionar en un bucle infinito cuando una rama gráfica que contiene un ejemplar de **View** se vuelve vivo en un universo virtual. Una vez arrancado, el renderizador Java 3D realiza las operaciones mostradas en el siguiente listado:

```
while(true) {
        Procesos de entrada
        If (petición de salida) break
                Realiza comportamientos
                Atraviesa el escenario gráfico
                y renderiza los objetos visuales
}
Limpieza y salida
```

Las secciones anteriores explicaban la construcción de un sencillo universo virtual sin una rama de contenido gráfico. La creacción de esta rama es el objetivo de las siguientes secciones.

Ejemplo de la Receta Sencilla: HelloJava3Da

El programa Java 3D típico empieza definiendo una nueva clase que extiende la clase **Applet**. El ejemplo HelloJava3Da.java es una clase definida para extender la clase **Applet**. Los programas Java 3D podrían escribirse como aplicaciones, pero usar Applets nos ofrece una forma más sencilla de producir una aplicación con ventanas.

La clase principal de un programa Java 3D normalmente define un método para construir la rama de contenido gráfico. En el ejemplo **HelloJava3Da** dicho método está definido como **createSceneGraph()**. Los pasos de la receta sencilla se implementan en el constructor de la clase **HelloJava3Da**. El paso 1, crear un objeto **Canvas3D**, se completa en la línea 4. El paso 2, crear un objeto **SimpleUniverse**, se hace en la línea 11. El paso 2a, personalizar el objeto **SimpleUniverse**, se realiza en la línea 15. El paso 3, construir la rama de contenido, se realiza en la llamada al método **createSceneGraph()**. El paso 4, compilar la rama de contenido gráfico, se hace en la línea 8. Finalmente el paso 5, insertar la rama de contenido gráfico en el objeto **Locale** del **SimpleUniverse**, se completa en la línea 16:

Fragmento de código 1-1. La clase HelloJava3D

```
1. public class HelloJava3Da extends Applet {
2.     public HelloJava3Da() {
3.       setLayout(new BorderLayout());
4.       Canvas3D canvas3D = new Canvas3D(null);
5.       add("Center", canvas3D);
6.
7.       BranchGroup scene = createSceneGraph();
8.       scene.compile();
9.
10.      // SimpleUniverse is a Convenience Utility class
```

```
11.     SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
12.
13.     // This moves the ViewPlatform back a bit so the
14.     // objects in the scene can be viewed.
15.     simpleU.getViewingPlatform().setNominalViewingTransform();
16.
17.     simpleU.addBranchGraph(scene);
18.   } // end of HelloJava3Da (constructor)
```

El paso 3 de esta sencilla receta es crear la rama de contenido gráfico. Esta rama se crea en el Fragmento de código 1-2. Probablemente sea la rama de contenido gráfico más sencilla posible. Contiene un objeto gráfico estático, un **ColorCube**. Éste está localizado en el origen del sistema de coordenadas del universo virtual. Con la localización y orientación dadas de la dirección de la vista del cubo, el cubo aparece como un rectángulo cuando es renderizado. La imagen que mostrará este programa la podemos ver en la Figura 1-12:

Fragmento de código 1-2. Método createSceneGraph de la clase HelloJava3D

```
1.    public BranchGroup createSceneGraph() {
2.      // Create the root of the branch graph
3.      BranchGroup objRoot = new BranchGroup();
4.
5.      // Create a simple shape leaf node, add it to the scene graph.
6.      // ColorCube is a Convenience Utility class
7.      objRoot.addChild(new ColorCube(0.4));
8.
9.      return objRoot;
10.   } // end of createSceneGraph method of HelloJava3Da
11. } // end of class HelloJava3Da
```

La clase **HelloJava3Da** está derivada de Applet pero el programa puede ejecutarse como una aplicación con el uso de la clase **MainFrame**. La clase Applet se usa como clase base para hacer más fácil la escritura de un programa Java 3D que se ejecuta en una ventana. **MainFrame** proporciona un marco AWT (ventana) para un applet permitiendo que el applet se ejecute como una aplicación. El tamaño de la ventana de la aplicación resultante se especifica en la construcción

de la clase **MainFrame**. El Fragmento de Código 1-3 muestra el uso de la clase **MainFrame** en **HelloJava3Da.java**.

Lista Parcial de Constructores de **MainFrame**

paquete: com.sun.j3d.utils.applet

MainFrame crea un applet en una aplicación. Una clase derivada de Applet podría tener un método **main()** que llame al constructor **MainFrame**. **MainFrame** extiende java.awt.Frame e implementa java.lang.Runnable, java.applet.AppletStub, y java.applet.AppletContext. La clase **MainFrame** es Copyright © 1996-1998 de Jef Poskanzer email: jef@acme.com en http://www.acme.com/java/.

MainFrame(java.applet.Applet applet, int width, int height)

Crea un objeto **MainFrame** que ejecuta un applet como una aplicación.

Parámetros:

- applet - el constructor de una clase derivada de Applet. MainFrame proporciona un marco AWT para este applet.
- width - la anchura de la ventana en pixels
- height - la altura de la ventana en pixels

Fragmento de código 1-3. Método Main() de HelloJava3Da Invoca a MainFrame

```
1.  // The following allows this to be run as an application
2.  // as well as an applet
3.
4.  public static void main(String[] args) {
5.  Frame frame = new MainFrame(new HelloJava3Da(), 256, 256);
6.  } // end of main (method of HelloJava3Da)
```

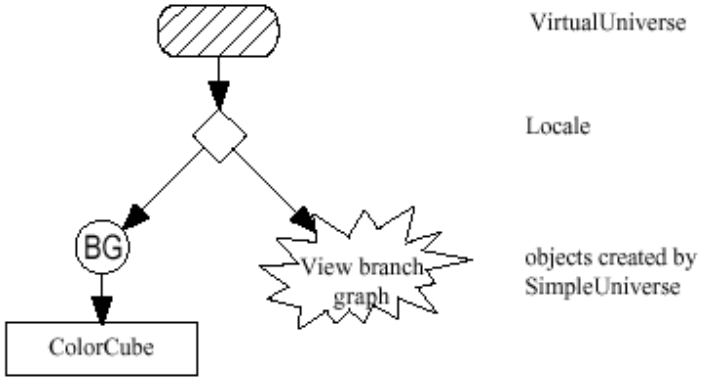
Los tres fragmentos de código anteriores (1-1, 1-2, y 1-3) forman un programa Java 3D completo cuando se usan las sentencias import adecuadas. Aquí podemos ver las sentencias import necesarias para compilar la clase **HelloJava3Da**. Las clases más comunmente usadas en Java 3D se encuentran en los paquetes javax.media.j3d, o javax.vecmath. En este ejemplo, sólo la clase de utilidad **ColorCube** se encuentra en el paquete com.sun.j3d.utils.geometry.

Consecuentemente, la mayoría de los programas Java 3D tienen las sentencias **import** mostradas en el Fragmento de Código 1-4 con la excepción de **ColorCube**.

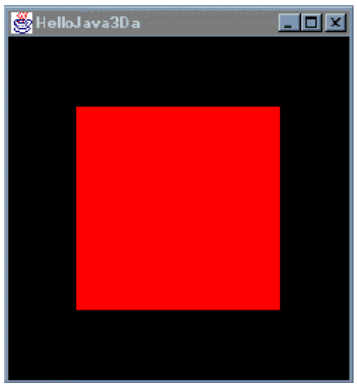
Fragmento de código 1-4. Sentencias Import para HelloJava3Da

```
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.*;
```

En el programa de ejemplo **HelloJava3Da.java**, sólo se sitúo un objeto gráfico en una única localización. En la figura 1-11 podemos ver el escenario gráfico resultante:



Compilamos el programa con el comando javac HelloJava3Da.java. Y lo ejecutamos con el comando: java HelloJava3Da. La imagen producida por el programa **HelloJava3Da** se puede ver en la Figura 1-12.



Como no se explica cada línea de código del ejemplo **HelloJava3Da**, las ideas básicas de ensamblar un programa Java 3D deberían estar claras habiendo leído el ejemplo. La siguiente sección presenta cada una de las clases usadas en el programa.

Clases Java 3D Usadas en HelloJava3Da

Para añadir un poco de entendimiento del API Java 3D y el ejemplo **HelloJava3Da** aquí presentamos una sinopsis de las clases del API Java 3D usadas en **HelloJava3Da**.

Clase BranchGroup

Los objetos de este tipo se usan para formar escenarios gráficos. Los ejemplares de **BranchGroup** son la raíz de los sub-gráficos. Los objetos **BranchGroup** son los únicos que pueden ser hijos de los objetos **Locale**. Los objetos **BranchGroup** pueden tener varios hijos. Los hijos de un objeto **BranchGroup** pueden ser otros objetos **Group** o **Leaf**.

Constructor por defecto de **BranchGroup**

BranchGroup()

Los ejemplares de **BranchGroup** sirven como raíz para las ramas del escenario gráfico; los objetos **BranchGroup** son los únicos objetos que pueden insertarse en un conjunto de objetos **Locale**.

Clase Canvas3D

La clase **Canvas3D** deriva de la clase **Canvas** del AWT. Al menos un objeto **Canvas3D** debe ser referenciado en la rama de vista gráfica del escenario gráfico.

Constructor de **Canvas3D**

Canvas3D(GraphicsConfiguration graphicsconfiguration)

Construye e inicializa un nuevo objeto **Canvas3D** que el Java 3D puede renderizar dando un objeto **GraphicsConfiguration** válido. Es una extensión de la clase **Canvas** del AWT.

Clase Transform3D

Los objetos **Transform3D** representan transformaciones de geometrías 3D como una traslación o una rotación. Estos objetos normalmente sólo se usan en la creacción de un objeto **TransformGroup**. Primero, se construye el objeto **Transform3D**, posiblemente desde una combinación de objetos **Transform3D**. Luego se construye el objeto **TransformGroup** usando el objeto **Transform3D**.

Constructor por Defecto de **Transform3D**

Un objeto de transformación generalizado se representa internamente como una matriz de 4x4 doubles de punto flotante. La representación matemática es la mejor forma. Un objeto **Transform3D** no se usa en un escenario gráfico. Se usa para especificar la transformación de un objeto **TransformGroup**.

Transform3D()

Construye un objeto **Transform3D** que representa la matriz de identidad (no la transformación).

Un objeto **Transform3D** puede representar una traslación, una rotación, un escalado, o una combinación de éstas. Cuando se especifica una rotación, el ángulo se expresa en radianes. Una rotación completa es **2 PI radianes**. Una forma de especificar ángulos es usar la constante Math.PI. Otra forma es especificar los valores directamente. Algunas aproximaciones son: 45º es 0.785, 90º es 1.57, y 180º es 3.14.

Lista Parcial de Métodos de **Transform3D**

Los objetos **Transform3D** representan transformaciones geométricas como una rotación, traslación y escalado. **Transform3D** es una de las pocas clases que no se usan directamente en un escenario gráfico. Las transformaciones representadas por objetos **Transform3D** se usan para crear objetos **TransformGroup** que si se usan en escenarios gráficos.

void rotX(double angle)

Selecciona el valor de esta transformación a una rotación en contra del sentido del

reloj sobre el eje-x. El ángulo se especifica en radianes.

void rotY(double angle)

Selecciona el valor de esta transformación a una rotación en contra del sentido del reloj sobre el eje-y. El ángulo se especifica en radianes.

void rotZ(double angle)

Selecciona el valor de esta transformación a una rotación en contra del sentido del reloj sobre el eje-z. El ángulo se especifica en radianes.

void set(Vector3f translate)

Selecciona el valor transacional de esta matriz al valor del parámetro **Vector3f**, y selecciona los otros componentes de la matriz como si ésta transformación fuera una matriz idéntica.

Clase TransformGroup

Como una subclase de la clase **Group**, los ejemplares de **TransformGroup** se usan en la creacción de escenarios gráficos y tienen una colección de objetos nodos como hijos. Los objetos **TransformGroup** contienen transformaciones geométricas como traslaciones y rotaciones. La transformación normalmente se crea en un objeto **Transform3D**, que no es un objeto del escenario gráfico.

Constructores de **TransformGroup**

Los objetos **TransformGroup** son contenedores de transformaciones en el escenario gráfico.

TransformGroup()

Construye e inicializa un **TransformGroup** usando una identidad de transformación.

TransformGroup(Transform3D t1)

Construye e inicializa un **TransformGroup** desde un objeto **Transform3D** pasado:
Parámetros:

- t1 - el objeto transform3D

La transformación contenida en un objeto **Transform3D** se copia a un objeto **TransformGroup** o cuando se crea el **TransformGroup**, o usando el método **setTransform()**.

Método **setTransform()** de TransformGroup

void setTransform(Transform3D t1)

Selecciona el componente de transformación de este **TransformGroup** al valor de la transformación pasada.

Parámetros:

- t1 - la transformación a copiar.

Clase Vector3f

Vector3f es una clase matemática que se encuentra en el paquete javax.vecmath para especificar un vector usando tres valores de punto flotante. Los objetos **Vector** se usan frecuentemente para especificar traslaciones de geometrías. Los objetos **Vector3f** no se usan directamente en la construcción de un escenario gráfico. Se usan para especificar la traslaciones, superficies normales, u otras cosas.

Constructores de **Vector3f**

Un vector de 3 elementos que es representado por puntos flotantes de precisión sencilla para las coordenadas x, y, y z.

Vector3f()

Construye e inicializa un **Vector3f** a (0,0,0).

Vector3f(float x, float y, float z)

Construye e inicializa un **Vector3f** desde las coordenadas x, y, z especificadas.

Clase ColorCube

ColorCube es una clase de utilidad que se encuentra en el paquete com.sun.j3d.utils.geometry que define la geometría y colores de un cubo centrado en el origen y con diferentes colores en cada cara. El objeto **ColorCube** es un cubo que tiene 2 métros de arista. Si un cubo sin rotar se sitúa en el origen (como en **HelloJava3Da**), se verá la cara roja desde la localización de visión nominal. Los otros colores son azul, magenta, amarillo, verde y cian.

Constructores de **ColorCube**

Paquete: com.sun.j3d.utils.geometry

Un **ColorCube** es un objeto visual, un cubo con un color diferente en cada cara. **ColorCube** extiende la clase **Shape3D**; por lo tanto, es un nodo hoja. **ColorCube** es fácil de usar cuando se pone junto a un universo virtual.

ColorCube()
Construye un cubo de color del tamaño por defecto. Por defecto, una esquina está situada a 1 metro de cada uno de los ejes desde el origen, resultando un cubo que está centrado en el origen y tiene 2 metros de alto, de ancho y de profundo.
ColorCube(double scale)
Construye un cubo de color escalado por el valor especificado. El tamaño por defecto es 2 metros de lado. El **ColorCube** resultante tiene esquinas en (scale, scale, scale) y (-scale, -scale, -scale).

Rotar el Cubo

Una simple rotación del cubo puede hacer que se vea más de una de sus caras. El primer paso es crear la transformación deseada usando un objeto **Transform3D**. El Fragmento de Código 1-5 incorpora un objeto **TransformGroup** en el escenario gráfico para rotar el cubo sobre el eje x. Primero se crea la transformación de rotación usando el objeto **rotate** de **Transform3D**. Este objeto se crea en la línea 6. La rotación se especifica usando el método **rotX()** de la línea 8. Entonces se crea

el objeto **TransformGroup** en la línea 10 para contener la transformación de rotación.

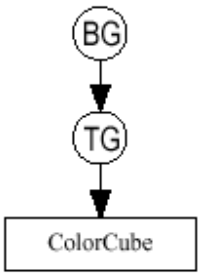
Dos parámetros especifican la rotación: el eje de revolución, y el ángulo de rotación. El eje se elige seleccionando el método apropiado. El ángulo de rotación es el valor que se le pasa como argumento. Como el ángulo de rotación se especifica en radianes, el valor **PI/4** es **1/8** de una rotación completa, o 45 grados. Después de crear el objeto Transform3D, **rotate**, se usa en la creacción del objeto TransformGroup **objRotate** (línea 10). El objeto Transform3D se usa en el escenario gráfico. Entonces el objeto **objRotate** hace que **ColorCube** sea su hijo (línea 11). A su vez, el objeto **objRoot** hace a **objRotate** como su hijo (línea 12).

Fragmento de código 1-5. Una Rotación en la Rama de Contenido Gráfico

```
1. public BranchGroup createSceneGraph() {
2.     // Create the root of the branch graph
3.     BranchGroup objRoot = new BranchGroup();
4.
5.     // rotate object has composite transformation matrix
6.     Transform3D rotate = new Transform3D();
7.
8.     rotate.rotX(Math.PI/4.0d);
9.
10.    TransformGroup objRotate = new TransformGroup(rotate);
11.    objRotate.addChild(new ColorCube(0.4));
12.    objRoot.addChild(objRotate);
13.    return objRoot;
14. } // end of createSceneGraph method
```

La rama de contenido gráfico ahora incluye un objeto **TransformGroup** en el camino del escenario gráfico hacia el objeto **ColorCube**.Cada uno de los caminos del escenario gráfico es necesario. El objeto **BranchGroup** es el único que puede ser hijo de un **Locale**. El objeto **TransformGroup** es el único que puede cambiar la localización, la orientación, o el tamaño de un objeto visual. En este caso el objeto **TransformGroup** cambia la orientación. Por supuesto, el objeto **ColorCube** es necesario para suministrar el objeto visual.

Aquí podemos ver la imagen producida por el Fragmento de Código 1-5.



Ejemplo de Combinación de Transformaciones: HelloJava3Db

Frecuentemente un objeto visual se traslada y se rota, o se rota sobre dos ejes. En cualquier caso, se especifican dos transformaciones diferentes para un sólo objeto visual. Las dos transformaciones pueden combinarse en una matriz de transformaciones y contenerse en un sólo objeto **TransformGroup**. Podemos ver un ejemplo en el Fragmento de Código 1-6.

En el programa HelloJava3Db.java se combinan dos rotaciones. Crear estas dos rotaciones simultáneas requiere combinar dos objetos **Transform3D** de rotación. El ejemplo rota el cubo sobre los ejes x e y. Se crean dos objetos **Transform3D**, uno por cada rotación (líneas 6 y 7). Las rotaciones individuales se especifican para los dos objetos **TransformGroup** (líneas 9 y 10). Luego las rotaciones se combinan mediante la multiplicación de los objetos **Transform3D** (línea 11). La combinación de las dos transformaciones se carga en el objeto **TransformGroup** (línea 12).

Fragmento de código 1-6. Dos Transformaciones de Rotación en HelloJava3Db

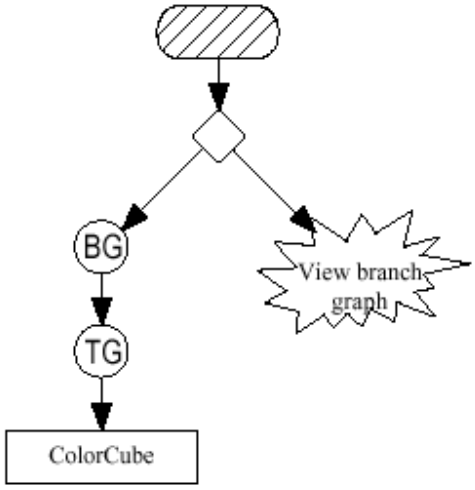
```
1. public BranchGroup createSceneGraph() {
2.     // Create the root of the branch graph
3.     BranchGroup objRoot = new BranchGroup();
4.
5.     // rotate object has composite transformation matrix
6.     Transform3D rotate = new Transform3D();
7.     Transform3D tempRotate = new Transform3D();
8.
9.     rotate.rotX(Math.PI/4.0d);
10.    tempRotate.rotY(Math.PI/5.0d);
11.    rotate.mul(tempRotate);
```

```
12.  TransformGroup objRotate = new TransformGroup(rotate);
13.
14.  objRotate.addChild(new ColorCube(0.4));
15.  objRoot.addChild(objRotate);
16.  return objRoot;
```

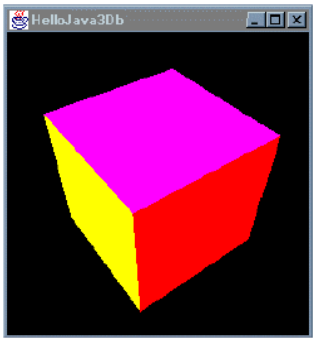
Tanto el Fragmento de Código 1-5 como el Fragmento de código 1-6 podrían reemplazar al Fragmento de Código 1-2. El Fragmento de código 1-6 se usa en **HelloJava3Db.java**. Aquí puedes encontrar el ejemplo completo:

HelloJava3Db.java

En la Figura 1.-14 podemos ver el escenario gráfico creado en **HelloJava3Db.java**. La rama de vista gráfica es la misma producida en **HelloJava3Da**, que está construida por un **SimpleUniverse** y representada por una gran estrella. La rama de contenido gráfico ahora incluye un **TransformGroup** en el camino del escenario gráfico hacia el objeto **ColorCube**.



La imagen de la figura 1-15 muestra el **ColorCube** girado del **HelloJava3Db**.



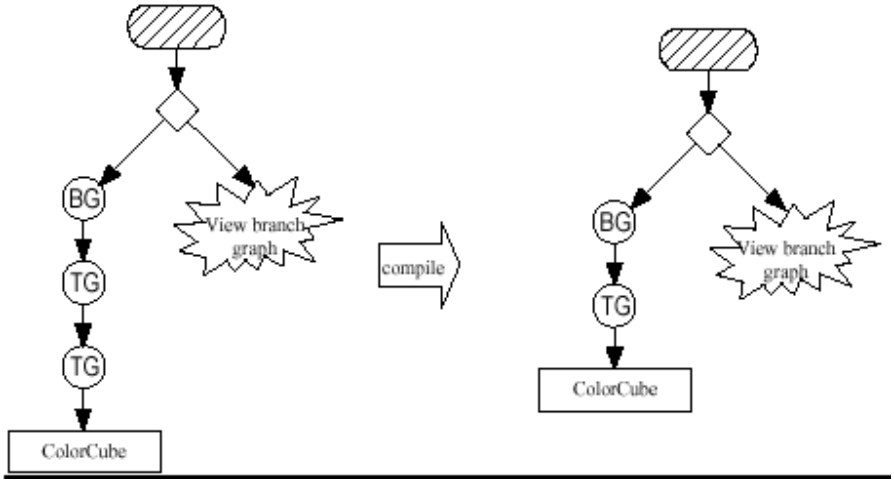
Capacidades y Rendimiento

El escenario gráfico construido por un programa Java 3D podría usarse directamente para renderizar. Sin embargo, la representación no es muy eficiente. La flexibilidad construida dentro de cada objeto escenario gráfico (que no se van a discutir en este tutorial) crean un representación sub-optima del universo virtual. Para mejorar el rendimiento de la renderización se usa una representación más eficiente del universo virtual.
Java 3D tiene una representación interna para una universo virtual y los métodos para hacer la conversión. Hay dos formas para hacer que el sistema Java 3D haga la conversión de la representación interna. Una forma es compilar todas las ramas gráficas. La otra forma es insertar una rama gráfica en un universo virtual para darle vida.

Compilar Contenidos

El objeto **BranchGroup** tiene un método compilador. Llamando a este método se convierte la rama gráfica completa que hay debajo del **BranchGroup** a la representación interna de Java 3D de la rama gráfica. Además de la conversión, la representación interna podría optimizarse de una o varias maneras.
Las posibles optimizaciones no se especifican en el API Java 3D. Sin embargo, se puede ganar en eficiencia de varias formas. Una de las posibles optimizaciones es combinar **TransformGroups** con caminos de escenario gráfico. Por ejemplo, si un escenario gráfico tiene dos objetos **TransformGroup** en una relacion padre-hijo pueden ser representados por un objeto **TransformGroup**. Otra posibilidad es combinar objetos **Shape3D** que tienen una relación estática física. Estos tipos de optimizaciones se hacen posibles cuando las capacidades no se configuran.
La Figura 1-16 presenta una representación conceptual de la conversión a una representación más eficiente. El escenario gráfico del lado izquierdo es compilado y transformado en la representación interna mostrada en el lado derecho. La figura

sólo representa el concepto de representación interna, no como Java 3D realmente lo hace.



- **Capacidades**

Una vez que una rama gráfica empieza a vivir o es compilada el sistema de renderizado Java 3D la convierte a una representación interna más eficiente. El efecto más importante de esta conversión es la mejora del rendimiento de renderizado.

Pero también tiene otros efectos, uno de ellos es fijar el valor de transformaciones y otros objetos en el escenario gráfico. A menos que especificamente lo proporcionemos en el programa, este no tendrá la capacidad de cambiar los valores de los objetos del escenario gráfico una vez que estén vivos.

Hay casos en que un programa necesita la capacidad de cambiar estos valores después de que estén vivos. Por ejemplo, cambiar el valor de un objeto **TransformGroup** crea animaciones. Para que esto suceda, la transforamción debe poder cambiar después de estar viva. La lista de parámetros a los que se puede acceder, y de que forma, se llama capacidades del objeto.

Cada **SceneGraphObject** tiene un conjunto de bits de capacidad. Los valores de estos bits determinan que capacidades existen para el objeto después de compilarlo o de darle vida. El conjunto de capacidades varía con la clase.

Lista Parcial de Métodos de **SceneGraphObject**

SceneGraphObject es la superclase de casi cualquier clase usada para crear un escenario gráfico, incluyendo **Group**, **Leaf**, y **NodeComponent**.

void clearCapability(int bit)
Borra el bit de capacidad especificado.
boolean getCapability(int bit)
Recupera el bit de capcidad especificado.
void setCapability(int bit)
Configura el bit de capacidad especificado.

Como ejemplo, para poder leer el valor de la transformación representada por un objeto **TransformGroup**, esta capacidad debe activarse antes de compilarlo o darle vida. De forma similar, para poder cambiar el valor de la transformación en un objeto **TransformGroup**, su capacidad de escribir transformación debe configurarse antes de compilarlo o darle vida. Intentar hacer un cambio en un objeto vivo o compilado para el que la propiedad adecuada no se ha configurado resultará en una excepción.
En la siguiente sección, las animaciones se crean usando una transformación de rotación que varía con el tiempo. Para que esto sea posible, el objeto **TransformGroup** debe tener su capacidad **ALLOW_TRANSFORM_WRITE** activada antes de que sea compilado o se le de vida.

Lista Parial de Capcidades de **TransformGroup**

Las dos capacidades listadas aquí son las únicas definidas por **TransformGroup**. Éste hereda varias capacidades de sus clases ancestros: **Group** y **Node**. La configuración de capacidades se puede seleccionar, eliminar o recuperar usando los métodos definidos en **SceneGraphObject**.

ALLOW_TRANSFORM_READ
Especifa que el nodo **TransformGroup** permite acceder a la información de transformación de su objeto.

ALLOW_TRANSFORM_WRITE

Especifica que el nodo **TransformGroup** permite escribir la información de transformación de su objeto.

Las capacidades también controlan el acceso a otros aspectos de un objeto **TransformGroup**. Los objetos **TransformGroup** heredan configuración de capacidades de sus clases ancestros: **Group** y **Node**. En el siguiente bloque de referencia podemos ver algunas de esas capacidades.

Lista Parcial de Capacidades de **Group**

TransformGroup hereda varios bits de capacidades de sus clases ancestros.

ALLOW_CHILDREN_EXTEND

Permite que se puedan añadir hijos al nodo **Group** después de que esté compilado o vivo.

ALLOW_CHILDREN_READ

Permite que se puedan leer las referencias a los hijos del nodo **Group** después de que esté compilado o vivo.

ALLOW_CHILDREN_WRITE

Permite que se puedan escribir las referencias a los hijos del nodo **Group** después de que esté compilado o vivo.

Añadir Comportamiento de Animación

En Java 3D, **Behavior** es una clase para especificar animaciones o interacciones con objetos visuales. El comportamiento puede cambiar virtualmente cualquier atributo de un objeto visual. Un programador puede usar varios comportamientos predefinidos o especificar un comportamiento personalizado. Una vez que se ha especificado un comportamiento para un objeto visual, el sistema Java 3D actualiza automáticamente la posición, la orientación, el color, u otros atributos del objeto visual.

La distinción entre animación e interacción es si el comportamiento es activado en respuesta al paso del tiempo o en respuesta a actividades del usuario, respectivamente.

Cada objeto visual del universo virtual puede tener sus propio comportamiento predefinido. De echo, un objeto visual puede tener varios comportamientos. Para especificar un comportamiento para un objeto visual, el programador crea objetos que especifiquen el comportamiento, añade el objeto visual al escenario gráfico y hace las referencias apropiadas entre los objetos del escenario gráfico y los objetos **Behavior**.

En un universo virtual con muchos comportamientos, se necesita una significante potencia de cálculo para calcular los comportamientos. Como tanto el renderizador como el comportamiento usan el mismo procesador, es posible que la potencia de cálculo que necesita el comportamiento degrade el rendimiento del renderizado. Java 3D permite al programador manejar este problema especificando un límite espacial para que el comportamiento tenga lugar. Este límite se llama región programada. Un comportamiento no está activo a menos que el volumen de activación de **ViewPlatform** intereseccione con una región progamada del **Behavior**. En otras palabras, si nadie en el bosque ve el árbol caer, éste no cae. La característica de región programada hace más eficiente a Java 3D en el manejo de universos virtuales con muchos comportamientos.

Un **Interpolator** es uno de las muchas clases de comportamientos predefinidos en el paquete corazón de Java 3D. Basado en una función de tiempo, el objeto **Interpolator** manipula los parámetros de un objeto del escenario gráfico. Por ejemplo, para el **RotationInterpolator**, manipula la rotación especificada por un **TransformGroup** para afectar la rotación de los objetos visuales que son ancestros de **TransformGroup**.

La siguiente lista enumera los pasos envueltos para especificar una animación con un objeto **interpolator**. Los cinco pasos forman una receta para crear un comportamiento de animación con interpolación:

1. Crear un **TransformGroup** fuente.
   Selecciona la capacidad **ALLOW_TRANSFORM_WRITE**.
2. Crear un objeto **Alpha** (función de tiempo en Java 3D)
   Especifica los parámetros de tiempo para el **alpha**
3. Crear el objeto **interpolator**.
   Tiene referencias con los objetos **Alpha** y **TransformGroup**.
   Personalizar los parámetros del comportamiento.
4. Especificar la región programada.
   Configura la región programada para el comportamiento.
5. Hacer el comportamiento como hijo del **TransformGroup**

Especificar un Comportamiento de Animación

Una acción de comportamiento puede ser cambiar la localización (PositionInterpolator), la orientación (RotationInterpolator), el tamaño (ScaleInterpolator), el color (ColorInterpolator), o la transpariencia (TransparencyInterpolator) de un objeto visual. Como se mencionó antes, los **Interpolators** son clases de comportamiento predefinidas. Todos los comportamientos mencionados son posibles sin usar un **interpolator**; sin embargo, los **interpolators** hacen mucho más sencilla la cracción de comportamientos. Las clases **Interpolators** existen para proporcionar otras acciones, incluyendo combinaciones de estas acciones.

Clase RotationInterpolator

Esta clase se usa para especificar un comportamiento de rotación de un objeto visual o de un grupo de objetos visuales. Un objeto **RotationInterpolator** cambia un objeto **TransformGroup** a una rotación especififca en repuesta a un valor de un objeto **Alpha**. Como el valor de este objeto cambia cada vez, la rotación también cambia. Un objeto **RotationInterpolator** es flexible en la especificación del eje de rotación, el ángulo de inicio y el ángulo final.

Para rotaciones constantes sencillas, el objeto **RotationInterpolator** tiene el siguiente constructor que puede usarse para eso:

Lista Parcial de Constructores de **RotationInterpolator**

Esta clase define un comportamiento que modifica el componente rotacional de su **TransformGroup** fuente linearizando la interpoalización entre un par de ángulos

especificados (usando el valor generado por el objeto **Alpha** especificado). El ángulo interpolado se usa para generar una transformación de rotación.

RotationInterpolator(Alpha alpha, TransformGroup target)
Este constructor usa valores por defecto de algunos parámetros del interpolador para construir una rotación completa sobre el eje y, usando el **TransformGroup** especificado.
Parámetros:

- alpha - la función de variación de tiempo para referencia.
- target - el objeto **TransformGroup** a modificar.

El objeto **TransformGroup** de un interpolador debe tener la capacidad de escritura activada.

Funciones de Variación de Tiempo: Mapear un Comportamiento en el Tiempo

Mapear una acción en el tiempo se hace usando un objeto **Alpha**. La especificación de este objeto puede ser compleja.

Clase Alpha

Los objetos de la clase **Alpha** se usan para crear una función que varía en el tiempo. La clase **Alpha** produce un valor entre cero y uno, inclusives. El valor que produce depende de la hora y de los parámetros del objeto **Alpha**. Los objetos **Alpha** se usan comunmente con un comportamiento I**nterpolator** para proporcionar animaciones de objetos visuales.
Alpha tiene diez parámetos, haciendo la programación tremendamente flexible. Sin entrar en detalles de cada parámetros, saber que un ejemplar de **Alpha** puede combinarse fácilmente con un comportamiento para proporcionar rotaciones sencillas, movimientos de péndulo, y eventos de una vez, como la apertura de puertas o el lanzamiento de cohetes.

Constructor de **Alpha**

La clase **Alpha** proporciona objetos para convertir la hora en un valor alpha (un

valor entre 0 y 1). El objeto **Alpha** es efectivamente una función de tiempo que genera valores alpha entre cero y uno. La función "f(t)" y las características del objeto **Alpha** están determinadas por parámetros definidos por el usuario:

Alpha()

Bucle continuo con un periodo de un segundo.

Alpha(int loopCount, long increasingAlphaDuration)

Este constructor toma sólo **loopCount** e **increasingAlphaDuration** como parámetros y asigna los valores por derecto a todos los demás parámetros, resultando un objeto **Alpha** que produce valores desde cero a uno crecientes. Esto se repite el número de veces especificado por **loopCount**. Si **loopCount** es -1, el objeto alpha se repite indefinidamente. El tiempo que tarde en ir desde cero hasta uno está especificando en el segundo parámetro usando una escala de milisegundos.

Parámetros:

- loopCount - número de veces que se ejecuta este objeto alpha; un valor de -1 especifica un bucle indefinido..
- increasingAlphaDuration - tiempo en milisegundos que tarda el objeto alpha en ir de cero a uno.

Región Progamada

Como se mencionó anteriormente, cada comportamiento tiene unos límites programados. Estos límites se configuran usando el método **setSchedulingBounds** de la clase **Behavior**.

Hay varias formas de especificar una región programada, la más sencilla es crear un objeto **BoundingSphere**. Otras opciones incluyen **BoundingBox** y **BoundingPolytope**.

Método **setSchedulingBounds** de Behavior

void setSchedulingBounds(Bounds region)

Selecciona la región programada del **Behavior** a unos límites especificados.

Parámetros:

- region - Los límites que contienen la región programada del **Behavior**.

Clase BoundingSphere

Especificar un límite esférico se consigue especificando un punto central y un rádio para la esfera. El uso normal de este tipo de límites es usar el centro a (0, 0, 0). Entonces el radio se selecciona lo suficientemente grande como para contener el objeto visual, incluyendo todas las posibles localizaciones del objeto.

Lista Parcial de Constructores de **BoundingSphere**

Esta clase define una región de límites esférica que está definida por un punto central y un rádio.

BoundingSphere()

Este constructor crea una límite esférico centrado en el origen (0, 0, 0) con un radio de 1.

BoundingSphere(Point3d center, double radius)

Construye e inicializa un **BoundingSphere** usando el punto central y el rádio especificados.

Ejemplo de Comportamiento: HelloJava3Dc

El Fragmento de Código 1-7 muestra un ejemplo completo del uso de las clases interpoladoras para crear una animación. La animación creada con este código es una rotación continúa con un tiempo de rotación total de 4 segundos.

El paso 1 de la receta es crear el objeto **TransformGroup** para modificarlo durante la ejecución. El objeto **TransformGroup** fuente de un interpolador debe tener activada la capacidad de escritura. El objeto **TransformGroup** llamado **objSpin** se crea en la línea 7. La capacidad de escritiura de **objSpin** se selecciona en la línea 8.

El paso 2 es crear un objeto alpha para dirigir la interpolación. Los dos parámetros especificados en la línea 16 del fragmento de código son el número de

interacciones del bucle y el tiempo de un ciclo. El valor de "-1" especifica un bucle contínuo. El tiempo se especifica en milisegundos por lo que el valor de 4000 significa 4 segundos. Por lo tanto, el comportamiento es rotar cada cuatro segundos.

El paso 3 de la receta es crear el objeto **interpolator**. El objeto **RotationInterpolator** se crea en las líneas 21 y 22. El interpolador debe tener referencias a la transformación fuente y al objeto alpha. Esto se consigue en el constructor. En este ejemplo se usa el comportamiento por defecto del **RotationInterpolator** para hacer una rotación completa sobre el eje y.

El paso 4 es especificar una región programada. Se usa un objeto **BoundingSphere** con sus valores por defecto. El objeto **BoundingSphere** se crea en la líena 25. La esfera se configura como los límites del comportamiento en la línea 26.

El paso final, el 5, hace del comportamiento un hijo del **TransformGroup**. Esto se consigue en la línea 27.

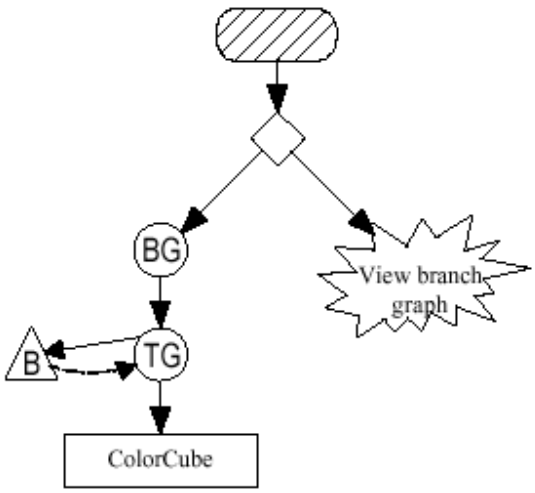
Fragmento de código 1-7. Método createSceneGraph con Comportamiento RotationInterpolator

```
1. public BranchGroup createSceneGraph() {
2.     // Create the root of the branch graph
3.     BranchGroup objRoot = new BranchGroup();
4.
5.     // Create the transform group node and initialize it to the
6.     // identity. Add it to the root of the subgraph.
7.     TransformGroup objSpin = new TransformGroup();
8.     objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
9.     objRoot.addChild(objSpin);
10.
11.    // Create a simple shape leaf node, add it to the scene graph.
12.    // ColorCube is a Convenience Utility class
13.    objSpin.addChild(new ColorCube(0.4));
14.
15.    // create time varying function to drive the animation
16.    Alpha rotationAlpha = new Alpha(-1, 4000);
```

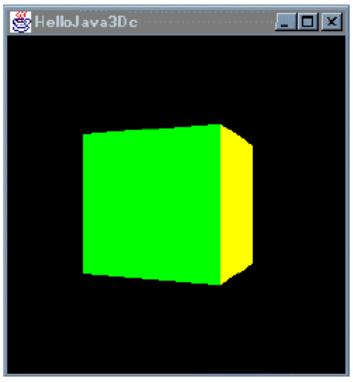
```
17.
18.   // Create a new Behavior object that performs the desired
19.   // operation on the specified transform object and add it into
20.   // the scene graph.
21.   RotationInterpolator rotator =
22.       new RotationInterpolator(rotationAlpha, objSpin);
23.
24.   // a bounding sphere specifies a region a behavior is active
25.   BoundingSphere bounds = new BoundingSphere();
26.   rotator.setSchedulingBounds(bounds);
27.   objSpin.addChild(rotator);
28.
29.   return objRoot;
30. } // end of createSceneGraph method
```

Este fragmento de código se usa con otros fragmentos anteriores para crear el programa de ejemplo HelloJava3Dc.java. Al ejecutar la aplicación veremos como se renderiza el **ColorCube** con un comportamiento de rotación cada cuatro segundos.

El programa **HelloJava3Dc** crea el escenario gráfico de la Figura 1-18. El objeto **rotation** es tanto hijo del **TransformGroup** como una referencia a él. Aunque esta relación parece violar la restricciones de bucles dentro del escenaio gráfico, no lo hace. Recuerda que los arcos de referencia (flecha punteada) no son parte del escenario gráfico. La línea punteada desde el **Behavior** hacia el **TransformGroup** es esta referencia.



La imagen de la Figura 1-19 muestra un marco de la ejecución del programa HelloJava3Dc.



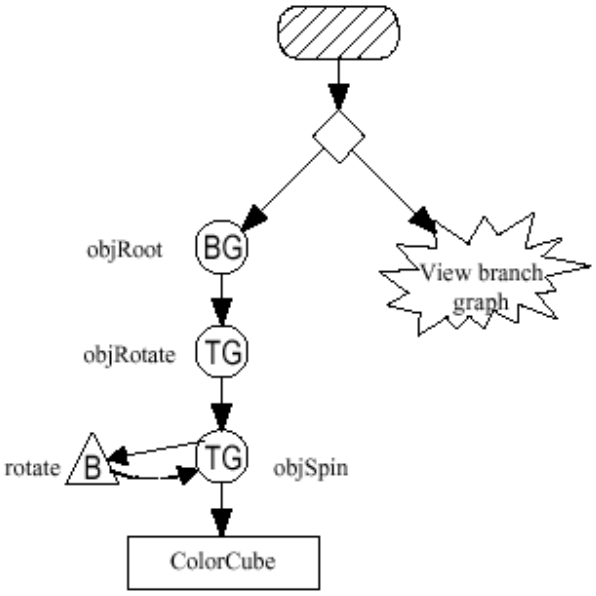
- **Ejemplo de Combinación de Transformation y Behavior: HelloJava3Dd**

Por supuesto, podemos combinar comportamientos con las transformaciones de rotación de los ejemplos anteriores. HelloJava3Dd.java hace esto. En la rama de contenido gráfico, hay objetos llamados **objRotate** y **objSpin**, que distinguen entre la rotación estática y el comportamiento de rotación (bucle continuo) del objeto cube respectivamente. El escenario resultante de este fragmento de código podemos verlo en la figura 1.20.

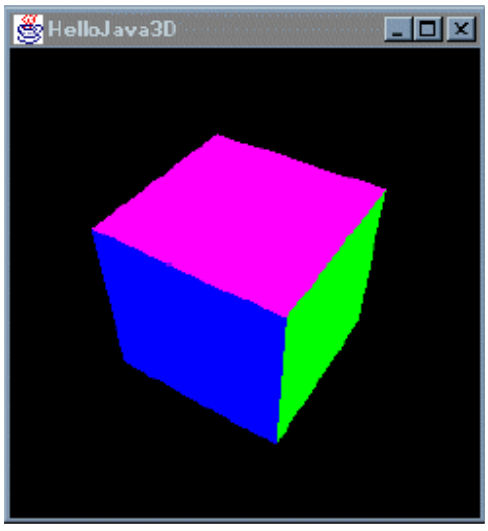
Fragmento de código 1-8. Rama de Contenido Gráfico para un ColorCube giratorio

```
1. public BranchGroup createSceneGraph() {
2.     // Create the root of the branch graph
3.     BranchGroup objRoot = new BranchGroup();
4.
5.     // rotate object has composite transformation matrix
6.     Transform3D rotate = new Transform3D();
7.     Transform3D tempRotate = new Transform3D();
8.
9.     rotate.rotX(Math.PI/4.0d);
10.    tempRotate.rotY(Math.PI/5.0d);
11.    rotate.mul(tempRotate);
12.
13.    TransformGroup objRotate = new TransformGroup(rotate);
14.
```

```
15.  // Create the transform group node and initialize it to the
16.  // identity. Enable the TRANSFORM_WRITE capability so that
17.  // our behavior code can modify it at runtime. Add it to the
18.  // root of the subgraph.
19.  TransformGroup objSpin = new TransformGroup();
20.  objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
21.
22.  objRoot.addChild(objRotate);
23.  objRotate.addChild(objSpin);
24.
25.  // Create a simple shape leaf node, add it to the scene graph.
26.  // ColorCube is a Convenience Utility class
27.  objSpin.addChild(new ColorCube(0.4));
28.
29.  // Create a new Behavior object that performs the desired
30.  // operation on the specified transform object and add it into
31.  // the scene graph.
32.  Transform3D yAxis = new Transform3D();
33.  Alpha rotationAlpha = new Alpha(-1, 4000);
34.
35.  RotationInterpolator rotator =
36.      new RotationInterpolator(rotationAlpha, objSpin, yAxis,
37.      0.0f, (float) Math.PI*2.0f);
38.
39.  // a bounding sphere specifies a region a behavior is active
40.  // create a sphere centered at the origin with radius of 1
41.  BoundingSphere bounds = new BoundingSphere();
42.  rotator.setSchedulingBounds(bounds);
43.  objSpin.addChild(rotator);
44.
45.  return objRoot;
46. } // end of createSceneGraph method of HelloJava3Dd
```



La imagen de la Figura 1-21 muestra un marco del ColorCube en movimiento del programa **HelloJava3Dd**.



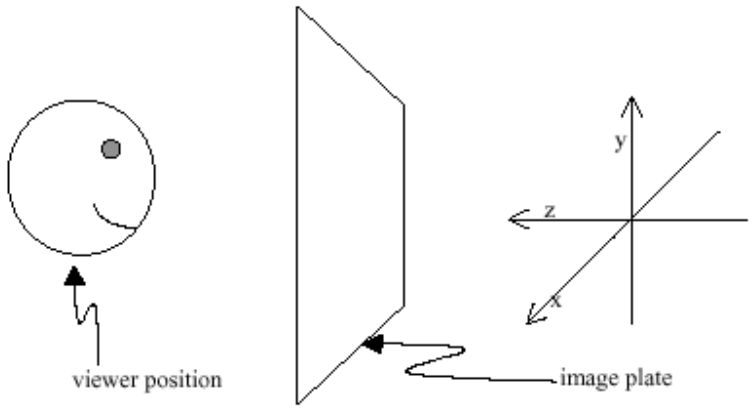
Crear Geometrías en Java 3D

Hay tres formas principales de crear contenidos geométricos. Una forma es usar las clases de utilidades geométricas para **box**, **cone**, **cylinder**, y **sphere**. Otra forma es especificar coordenadas de vértices para puntos, segmentos de líneas y/o

superficies poligonales. Una tercera forma es usar un cargador geométrico. Esta página demuestra la creacción de contenidos geométricos de las dos primeras formas.

Sistema de Coordenadas del Mundo Virtual

Como se explicó en la página anterior, un ejemplar de la clase **VirtualUniverse** sirve como raíz para el escenario gráfico de todos los programas Java 3D. El término 'Universo Virtual' comunmente se refiere al espacio virtual de tres dimensiones que rellenan los objetos Java 3D. Cada objeto **Locale** del universo virtual establece un sistema de coordenadas Cartesianas.
Un objeto **Locale** sirve como punto de referencia para los objetos visuales en un universo virtual. Con un **Locale** en un **SimpleUniverse**, hay un sistema de coordenadas en el universo visrtual.
El sistema de coordenadas del universo virtual Java 3D es de mano derecha. El eje X es positivo hacia la derecha, el eje Y es positivo hacia arriba y el eje Z es positivo hacia el espectador, con todas las unidades en metros. La Figura 2-1 muestra la orientación con respecto al espectador en un **SimpleUniverse**.

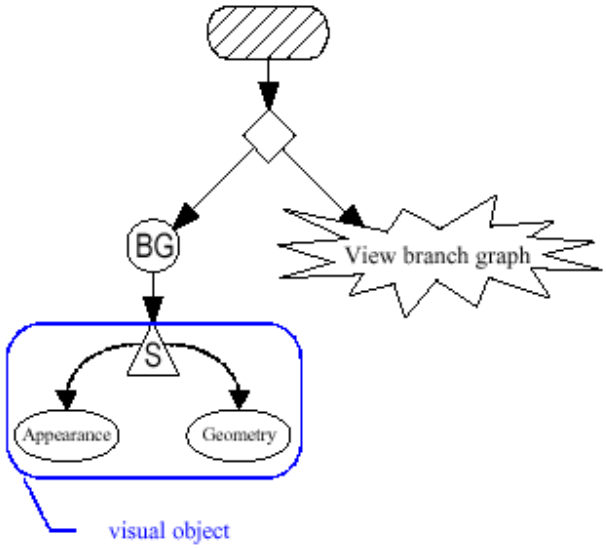


Definición Básica de Objeto Visual

Un Ejemplar de Shape3D Define un Objeto Visual

Un nodo **Shape3D** de escenario gráfico define un objeto visual. **Shape3D** es una de las subclases de la clase **Leaf**; por lo tanto, los objetos **Shape3D** sólo pueden ser hojas en un escenario gráfico. El objeto **Shape3D** no contiene información sobre la forma o el color de un objeto visual. Esta información está almacenada en los objetos **NodeComponent** referidos por el objeto **Shape3D**. Un objeto **Shape3D** puede referirse a un componente nodo **Geometry** y a un componente nodo **Appearance**.

En los escenarios gráficos de la página anterior, el símbolo de objeto genérico (rectángulo) fue utilizado para representar el objeto **ColorCube**. El sencillo escenario gráfico de la Figura 2-2 muestra un objeto visual representado como un hoja **Shape3D** (triángulo) y dos **NodeComponents** (óvalos) en lugar del rectángulo genérico.



Un objeto visual se puede definir usando sólo un objeto **Shape3D** y un nodo componente **Geometry**. Opcionalmente, el objeto **Shape3D** también se refiere a un nodo componente **Appearance**. Los constructores de **Shape3D** muestran que se pueden crear sin referencias a componentes nodos, con sólo una referencia a un nodo **Geometry**, o con referencias a ámbos tipos de componentes.

Constructores de **Shape3D**

Shape3D()

Construye e inicializa un objeto **Shape3D** sin ningún tipo de componentes.

Shape3D(Geometry geometry)

Construye e inicializa un objeto **Shape3D** con la geometría especificada y un componente de apariencia nulo.

Shape3D(Geometry geometry, Appearance appearance)

Construye e inicializa un objeto **Shape3D** con los componentes de geometría y apariencia especificados.

Mientras que el objeto **Shape3D** no esté vivo o compilado, las referencias a los componentes pueden modicarse con los métodos del siguiente recuadro. Estos métodos pueden usarse sobre objetos **Shape3D** vivos o compilados si se configuran las capacidades del objeto primero. El otro recuadro lista las capacidades de **Shape3D**.

Lista Parcial de Métodos de **Shape3D**

Un objeto **Shape3D** referencia a objetos **NodeComponente**: **Geometry** y/o **Appearance**. Junto con los métodos de configuración mostrados aquí, también existen los complementarios métodos **get**

void setGeometry(Geometry geometry)
void setAppearance(Appearance appearance)

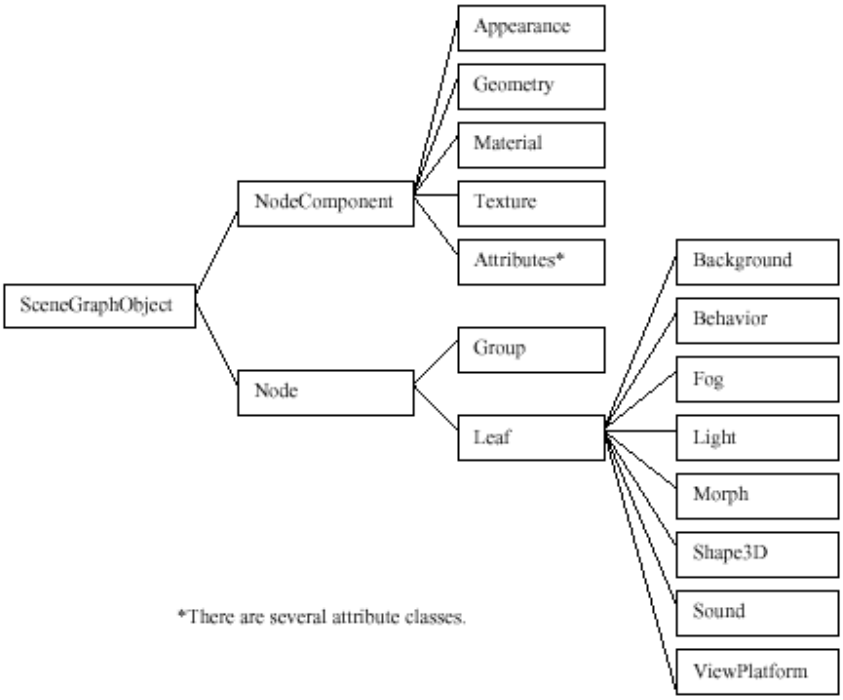
Capacidades de **Shape3D**

Los objetos **Shape3D** también heredan capacidades de las clases **SceneGrpahObject**, **Node**, y **Leaf**.

ALLOW_GEOMETRY_READ | WRITE
ALLOW_APPEARANCE_READ | WRITE
ALLOW_COLLISION_BOUNDS_READ | WRITE

NodeComponent

Los objetos **NodeComponent** contienen las especificaciones exactas de los atributos de un objeto visual. Cada una de las muchas subclases de **NodeComponent** define ciertos atributos visuales. La Figura 2-3 muestra una parte del árbol del API Java 3D que contiene las clases **NodeComponent** y sus descendientes.



Definir Clases de Objetos Visuales

El mismo objeto visual puede aparecer muchas veces en un sólo universo virtual. Tiene sentido definir una clase para crear el objeto visual en lugar de construir cada objeto visual desde el principio. Hay varias formas de diseñar una clase que define un objeto visual.

El fragmento de código 2-1 muestra el código esqueleto de la clase **VisualObject** como ejemplo de una organización posible para una clase de un objeto visual. Los métodos no tienen código. El código de **VisualObject** no aparece en los ejemplos porque no es particularmente útil.

```
1. public class VisualObject extends Shape3D{
2.
```

```
3.    private Geometry voGeometry;
4.    private Appearance voAppearance;
5.
6.    // create Shape3D with geometry and appearance
7.    // the geometry is created in method createGeometry
8.    // the appearance is created in method createAppearance
9.    public VisualObject() {
10.
11.   voGeometry = createGeometry();
12.   voAppearance = createAppearance();
13.   this.setGeometry(voGeometry);
14.   this.setAppearance(voAppearance);
15.   }
16.
17.   private Geometry createGeometry() {
18.   // code to create default geometry of visual object
19.   }
20.
21.   private Appearance createAppearance () {
22.   // code to create default appearance of visual object
23.   }
24.
25. } // end of class VisualObject
```

La organización de la clase **VisualObject** en el Fragmento de Código 2-1 es similar a la clase **ColorCube** que extiende un objeto **Shape3D**. Sugerimos la clase **VisualObject** como punto de arranque para definir clases con contenidos personalizados para usarlos en construcción de escenarios gráficos. Para ver un ejemplo completo de la organización de esta clase puedes leer el código fuente de la clase **ColorCube** que está en el paquete com.sun.j3d.utils.geometry, que está disponible en la distribución del API Java 3D.

Usar **Shape3D** como base para la creación de clases de objetos visuales facilita su uso en programas Java 3D. Las clases de objetos visuales pueden usarse tan facilmente como la clase **ColorCube** en los ejemplos **HelloJava3D** de la página anterior. Se puede llamar al constructor e insertar el objeto creado como hijo de

algún **Group** en una línea del codigo. En la siguiente línea de ejemplo, **objRoot** es un ejemplar de **Group**. Este código crea un **VisualObject** y lo añade como hijo de **objRoot** en el escenario gráfico:

```
objRoot.addChild(new VisualObject());
```

El constructor **VisualObject** crea el **VisualObject** creando un objeto **Shape3D** que referencia al **NodeComponents** creado por los métodos createGeometry() y createAppearance(). El método createGeometry() crea un NodeComponent **Geometry** para suarlo en el objeto visual. El método createAppearance() es responsable de crear el **NodeComponent** que define la **Appearance** del objeto visual.

Otra posible organización de un objeto visual es definir una clase contenedor no derivada del API de clases Java 3D. En este diseño, la clase del objeto visual podría contener un **Group Node** o un **Shape3D** como la raíz del sub-gráfico que define. La clase debe definir método(s) para devolver una referencia a su raíz. Esta técnica tiene un poco más de trabajo, pero podría ser más fácil de entender.

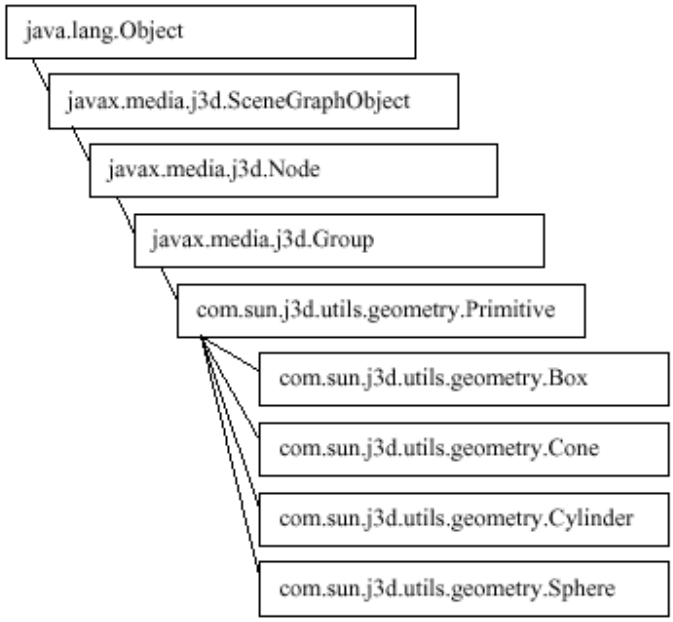
Una tercera organización posible de una clase de objeto visual es una similar a las clases **Box**, **Cone**, **Cylinder**, y **Sphere** definidas en el paquete com.sun.j3d.utils.geometry. Cada clase desciende de **Primitive**, que a su vez desciende de **Group**. Los detalles de diseño de **Primitive** y sus descendientes no se explican en este turorial, pero el código fuente de todas estas clases está disponible con la distribución del API Java 3D. Del código fuente de la clase **Primitive** y de otras clases de utilidad, el lector puede aprender más sobre esta aproximación al diseño de clases.

Clases de Utilidades Geométricas

Esta sección cubre las clases de utilidad para crear gráficos primitivos geométricos como cajas, conos, cilindros y esferas. Los primitivos geométricos son la segunda forma más fácil para crear contenidos en un universo virtual. La más fácil es usar la clase ColorCube.

Las clases primitivas proporcionan al programador más flexibilidad que la clase **ColorCube**. Un objeto **ColorCube** define la geometría y el cólor en un componente

Geometry. Consecuentemente, todo en el **ColorCube** es fijo, excepto su tamaño. El tamaño de un **ColorCube** sólo se especifica cuando se crea.
Un objeto primitivo proporciona más flexibilidad especificando la forma sin especificar el color. En una clase de utilidad geométrica primitiva, el programador no puede cambiar la geometría, pero puede cambiar la apariencia. Las clases primitivas le dan al programador la flexibilidad de tener varios ejemplares de la misma forma geométrica primitiva donde cada una tiene una apariencia diferente haciendo una referencia a un **NodeComponent** de apariencia diferente.
Las clases de utilidad **Box**, **Cone**, **Cylinder** y **Sphere** están definidas en el paquete com.sun.j3d.utils.geometry. En la Figura 2-3 podemos ver la porción del paquete com.sun.j3d.utils.geometry que contiene las clases primitivas.



- **Box**

La clase geométrica **Box** crea cubos de 3 dimensiones. Los valores por defecto para la longitud, anchura y altura son 2 metros, con el centro en el origen, resultando en un cubo con esquinas en (-1, -1, -1) y (1, 1, 1). La longitud, la anchura y la altura pueden especificarse en el momento de la creacción del objeto. Por supuesto, se pueden usar **TransformGroup** junto con el camino del escenario

gráfico a un **Bos** para cambiar la localización u orientación de los ejemplares creados con **Box**.

Box

Paquete:

com.sun.j3d.utils.geometry

Box desciende de **Primitive**, otra clase del paquete com.sun.j3d.utils.geometry.

Box()

Construye un cubo por defecto con 2,0 metros de altura, anchura y profundidad, centrado en el origen.

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

Construye un cubo con las dimensiones y apariencia dadas, centrado en el origen.

Mientras que los constructores son diferentes para cada clase, las clases **Box**, **Cone**, y **Cylinder** comparten los mismos métodos:

Métodos de **Box, Cone, y Cylinder**

Paquete:

com.sun.j3d.utils.geometry

Estos métodos están definidos en todas las clases primitivas: **Box**, **Cone**, y **Cylinder**. Estos Primitivos se componen de varios objetos **Shape3D** en un grupo.

Shape3D getShape(int id)

Obtiene una de las caras (Shape3D) del primitivo que contiene la geometría y apariencia. Los objetos **Box**, **Cone**, y **Cylinder** están compuestos por más de un objeto **Shape3D**, cada uno con su propio componente **Geometry**.

void setAppearance(Appearance appearance)

Selecciona la apariencia del primitivo (para todos los objetos **Shape3D**).

- **Cone**

La clase **Cone** define conos centrados en el origen y con el eje central alineado con el eje Y. Los valores por defecto para el radio y la altura son 1,0 y 2,0 metros respectivamente.

Lista Parcial de Constructores de **Cone**

Paquete:

com.sun.j3d.utils.geometry

Cone desciende de **Primitive**, otra clase del paquete com.sun.j3d.utils.geometry.

Cone()
Construye un cono con los valores de radio y altura por defecto.
Cone(float radius, float height)
Construye un cono con el radio y altura especificados.

- **Cylinder**

La clase **Cylinder** crea objetos cilíndricos con sus eje central alineado con el eje Y. Los valores por defecto para el radio y la altura son 1,0 y 2,0 metros respectivamente.

Lista Parcial de Constructores de **Cylinder**

Paquete:

com.sun.j3d.utils.geometry

Cylinder desciende de **Primitive**, otra clase del paquete com.sun.j3d.utils.geometry.

Cylinder()
Construye un cilindro con los valores de radio y altura por defecto.
Cylinder(float radius, float height)
Construye un cilindro con los valores de radio y altura especificados.
Cylinder(float radius, float height, Appearance appearance)

Construye un cilindro con los valores de radio, altura y apariencia especificados.

- **Sphere**

La clase **Sphere** crea objetos visuales esféricos con el centro en el origen. El radio por defecto es de 1,0 metros.

Lista Parcial de Constructores de **Sphere**

Paquete:

com.sun.j3d.utils.geometry

Sphere desciende de **Primitive**, otra clase del paquete com.sun.j3d.utils.geometry.

Sphere()

Construye una esfera con el radio por defecto (1,0 metros)

Sphere(float radius)

Construye una esfera con el radio especificado.

Sphere(float radius, Appearance appearance)

Construye una esfera con el radio y la apariencia especificados.

Métodos de **Sphere**

Paquete:

com.sun.j3d.utils.geometry

Como una extensión de **Primitive**, un **Sphere** es un objeto **Group** que tiene un sólo objeto hijo **Shape3D**.

Shape3D getShape()

Obtiene el **Shape3D** que contiene la geometría y la apariencia.

Shape3D getShape(int id)

Este método se incluye por compatibilidad con otras clases primitivas: **Box**, **Cone**, y **Cylinder**. Sin embargo, como una **Sphere** sólo tiene un objeto **Shape3D**, sólo

puede ser llamado con id = 1.

void setAppearance(Appearance appearance)

Selecciona la apariencia de la esfera.

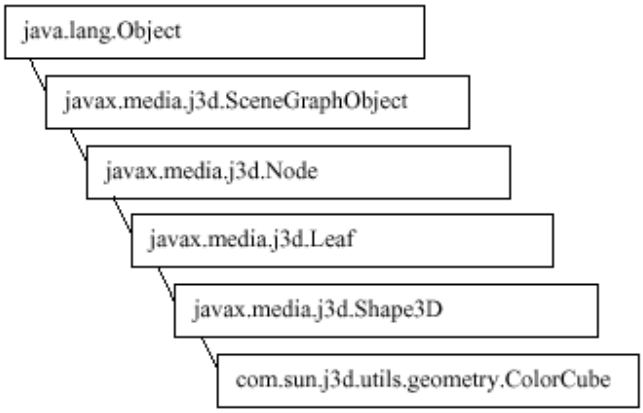
Más Sobre los Geométricos Primitivos

La geometría de una clase de utilidad primitiva no define el color. La **Geometry** que no define el color deriva su color desde su componente **Appearance**. Sin una referencia a un nodo **Appearance**, el objeto visual sería blanco, el color por defecto.

La clase **Primitive** define valores por defecto comunes para **Box**, **Cone**, **Cylinder**, y **Sphere**. Por ejemplo, define el valor por defecto para el número de polígonos usado para representar superficies.

ColorCube

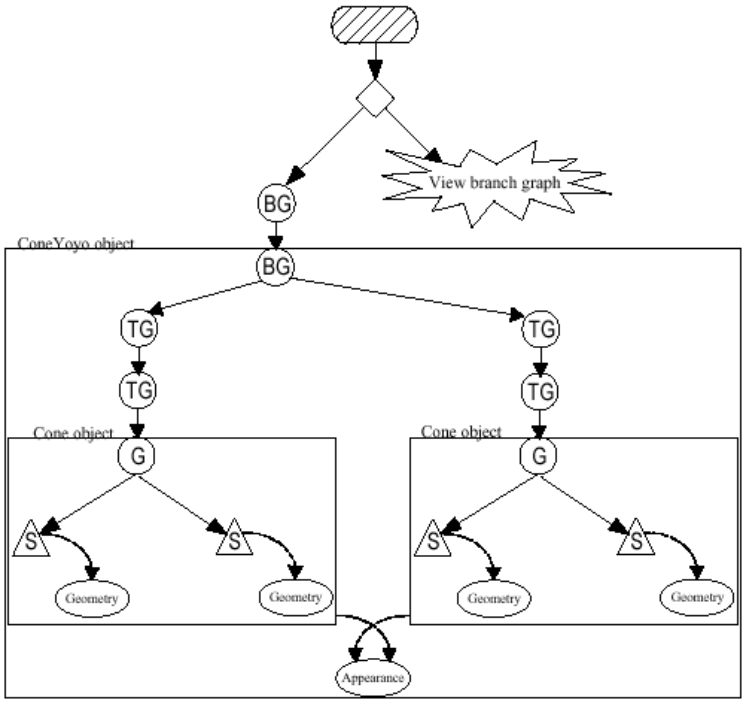
La clase **ColorCube** se presenta aquí para constrastarla con las clases primtivas. Esta clase desciende de otra parte del árbol de clases Java 3D. Este árbol de clases podemos verlo en la Figura 2-5.



ColorCube es la única clase distribuida con el API Java 3D que permite a los programadores ignorar los problemas con los colores y las luces. Por esta razón, la clase **ColorCube** es útil para ensamblar rápidamente escenario gráficos para probar prototipos.

Ejemplo: Crear un Simple Yo-Yo desde dos Conos

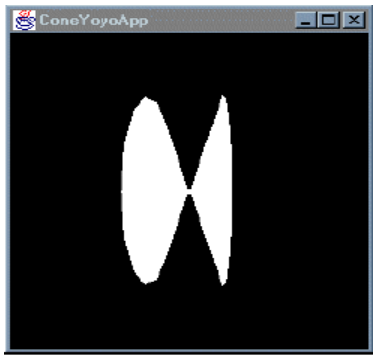
Esta sección presenta un sencillo ejemplo que usa la clase **Cone**: ConeYoyoApp.java. El objetivo del programa es renderizar un yo-yo. Se usan dos conos para formar el yo-yo. Se puede usar el API de comportamientos de Java 3D para hacer que el yo-yo se mueva de arriba a arriba, pero esto va más allá del ámbito de esta página. El programa gira el yo-yo para que se puedan apreciar las geometrías. El diagrama del escenario gráfico de la Figura 2-6 muestra el diseño de las clases **ConeYoyo** y **ConeYoyoApp** del programa de ejemplo.
La posición por defecto del objeto **Cone** es con su caja de limites centrada en el origen. La orientación por defecto es con la punta del objeto **Cone** en dirección a los positivos del eje Y. El yo-yo está formado por dos conos que se rotan sobre el eje Z y se trasladan a lo largo del eje X para poner juntas las puntas de los dos conos en el origen. Se pordrían utilizar otras combinaciones de rotaciones o traslaciones para conseguir que se juntaran las puntas de los conos.



En la rama gráfica que empieza con el objeto **BranchGroup** creada por el objeto **ConeYoyo**, el camino de escenario gráfico a cada objeto **Cone** empieza con el

objeto **TransformGroup** que especifica la traslación, seguido por el **TransformGroup** que especifica la rotación, y termina en el objeto **Cone**.
Varios escenarios gráficos podrían presentar el mismo mundo virtual. Tomando el escenario gráfico de la Figura 2-6 como ejemplo, se pueden hacer algunos cambios obvios. Un cambio elimina el objeto **BranchGroup** cuyo hijo es el objeto **ConeYoyo** e inserta el objeto **ConeYoyo** directamente en el objeto **Locale**. Otro cambio combina los dos objetos **TransformGroup** dentro del objeto **ConeYoyo**. Las transformaciones se han mostrado sólo como ejemplos.
Los nodos **Shape3D** de los objetos **Cone** referencian a los componentes **Geometry**. Estos son internos al objeto **Cone**. Los objetos **Shape3D** del **Cone** son hijos de un **Group** en el **Cone**. Como los objetos **Cone** descienden de **Group**, el mismo **Cone** (u otro objeto Primitivo) no puede usarse más de una vez en un escenario gráfico. Abajo podemos ver un ejemplo de mensaje de error producido cuando intentamos usar el mismo objeto **Cone** dos veces en un único escenario gráfico. Este error no existe en el programa de ejemplo distribuido en este tutorial.

```
Exception in thread "main" javax.media.j3d.MultipleParentException:
Group.addChild: child already has a parent
    at javax.media.j3d.GroupRetained.addChild(GroupRetained.java:246)
    at javax.media.j3d.Group.addChild(Group.java:241)
    at ConeYoyoApp$ConeYoyo.<init>(ConeYoyoApp.java:89)
    at ConeYoyoApp.createSceneGraph(ConeYoyoApp.java:119)
   at ConeYoyoApp.<init>(ConeYoyoApp.java:159)
   at ConeYoyoApp.main(ConeYoyoApp.java:172)
```



La Figura 2-8 muestra una de las posibles imagenes renderizadas por **ConeYoyoApp.java** como el objeto **ConeYoyo**. En el Fragmento de Código 2-2 podemos ver el código del programa.

Las líneas de 14 a 21 crean los objetos de una mitad del escenario gráfico del yo-yo. La líneas 23 a 25 crean la relaciones entre estos objetos. El proceso se repite para la otra mitad del yo-yo en las líneas 27 a 38.

La línea 12 crea **yoyoAppear**, un componente nodo **Appearance** con valores por defecto, para usarlo en los objeto **Cone**. La líneas 21 a 34 seleccionan los parámetros de los dos conos.

Fragmento de Código 2-2. La clase ConeYoyo del programa ConeYoyoApp.java

```
1. public class ConeYoyo{
2.
3.    private BranchGroup yoyoBG;
4.
5.    // create Shape3D with geometry and appearance
6.    //
7.    public ConeYoyo() {
8.
9.    yoyoBG = new BranchGroup();
10.   Transform3D rotate = new Transform3D();
11.   Transform3D translate = new Transform3D();
12.   Appearance yoyoAppear = new Appearance();
13.
14.   rotate.rotZ(Math.PI/2.0d);
15.   TransformGroup yoyoTGR1 = new TransformGroup(rotate);
16.
17.  translate.set(new Vector3f(0.1f, 0.0f, 0.0f));
18.   TransformGroup yoyoTGT1 = new TransformGroup(translate);
19.
20.   Cone cone1 = new Cone(0.6f, 0.2f);
21.   cone1.setAppearance(yoyoAppear);
22.
23.   yoyoBG.addChild(yoyoTGT1);
24.   yoyoTGT1.addChild(yoyoTGR1);
```

```
25.   yoyoTGR1.addChild(cone1);
26.
27.   translate.set(new Vector3f(-0.1f, 0.0f, 0.0f));
28.   TransformGroup yoyoTGT2 = new TransformGroup(translate);
29.
30.   rotate.rotZ(-Math.PI/2.0d);
31.   TransformGroup yoyoTGR2 = new TransformGroup(rotate);
32.
33.   Cone cone2 = new Cone(0.6f, 0.2f);
34.   cone2.setAppearance(yoyoAppear);
35.
36.   yoyoBG.addChild(yoyoTGT2);
37.   yoyoTGT2.addChild(yoyoTGR2);
38.   yoyoTGR2.addChild(cone2);
39.
40.   yoyoBG.compile();
41.
42.   } // end of ConeYoyo constructor
43.
44.   public BranchGroup getBG(){
45.   return yoyoBG;
46.  }
47.
48. } // end of class ConeYoyo
```

Geometrías Primitivas

El árbol de clases de la Figura 2-4 muestra **Primitive** como la superclase de **Box**, **Cone**, **Cylinder**, y **Sphere**. Define un número de campos y métodos comunes a estas clases, así como los valores por defecto para los campos.
La clase **Primitive** proporciona una forma de compartir nodos componentes **Geometry** entre ejemplares de un primitivo del mismo tamaño. Por defecto, todos los primitivos del mismo tamaño comparte un nodo componente de geometría. Un ejemplo de un campo definido en la clase **Primitive** es el entero GEOMETRY_NOT_SHARED. Este campo especifica la geometría que se está creando

y que no será compartido con otros. Seleccionamos esta bandera para evitar que la geometría sea compartida entre primtivos de los mismos parámetros (es decir, esferas con rádio 1).

```
myCone.setPrimitiveFlags(Primitive.GEOMETRY_NOT_SHARED);
```

Lista Parcial de Métodos **Primitive**

Paquete:

com.sun.j3d.utils.geometry

Primitive desciende de **Group** y es la superclase de **Box**, **Cone**, **Cylinder**, y **Sphere**.

public void setNumVertices(int num)

Selecciona el número total de vértices en este primitivo.

void setPrimitiveFlags(int fl)

Las banderas de **Primitive** son:

- GEOMETRY_NOT_SHARED
  Se generan normalmente junto las posiciones.
- GENERATE_NORMALS_INWARD
  Normalmente se lanzan junto con las superficies.
- GENERATE_TEXTURE_COORDS
  Se generan las coordenadas de textura.
- GEOMETRY_NOT_SHARED
  La geometría creada no se compartirá con ningún otro nodo.

void setAppearance(int partid, Appearance appearance)

Selecciona la apariencia de una subparte dando un **partid**. Los objetos **Box**, **Cone**, y **Cylinder** están compuestos por más de un objeto **Shape3D**, cada uno potencialmente tiene su propio nodo **Appearance**. El valor usado para **partid** especifica el componente **Appearance** seleccionado.

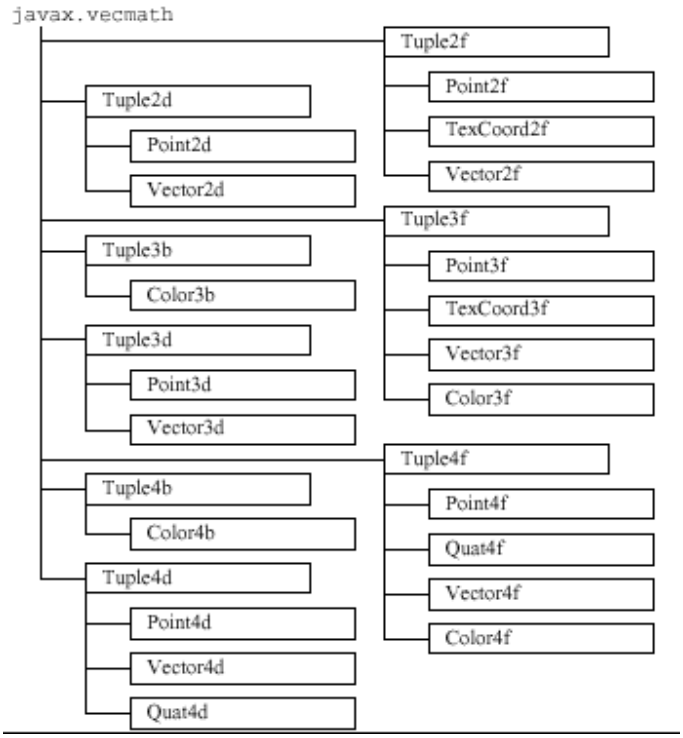
void setAppearance()

Selecciona la apariencia principal del primitivo (todas las subpartes) a una apariencia blanca por defecto.

Otros constructores adicionales de **Box**, **Cone**, **Cylinder**, y **Sphere** permiten la especificación de banderas **Primitive** en el momento de la creacción. Puedes consultar la especificación del API Java 3D para más información.

Clases Matemáticas

Para crear objetos visuales, se necesitan la clase **Geometry** y sus subclases. Muchas de éstas subclases describen primitivos basados en vértices, como puntos, líneas y polígonos rellenos. Las subclases se verán en una próxima sección, pero antes veremos varias clases matemáticas (Point*, Color*, Vector*, TexCoord*) usadas para especificar datos relacionados con los vértices
Nota: el asterisco usado arriba es un comodín para representar variaciones en el nombre de la clases. Por ejemplo, **Tuple*** se refiere a todas las clases: **Tuple2f**, **Tuple2d**, **Tuple3b**, **Tuple3f**, **Tuple3d**, **Tuple4b**, **Tuple4f**, y **Tuple4d**. En cada caso el número indica el número de elementos del **Tuple**, y la letra indica el tipo de los datos. **_f_** indica un tipo de coma flotante de simple precisión, **_d_** indica un tipo de coma flotante de doble precisión, y **_b_** es para bytes. Por eso **Tuple3f** es una clase que manipula valores en coma flotante de simple precisión.
Todas estas clases matemáticas están en el paquete javax.vecmath.*. Este paquete define varias clases **Tuple*** como superclases genéricas abstractas. Otras clases más útiles descienden de las clases **Tuple**. En la Figura 2-9, podemos ver algunas de las clases del árbol.



Cada vértice de un objeto visual podría especificar hasta cuatro objetos **javax.vecmath**, representadno coordenadas, colores. superficies normales, y coordenadas de textura. Normalmente se usan las siguientes clases:

- Point* (para coordenadas)
- Color* (para colores)
- Vector* (para superficies normales)
- TexCoord* (para coordenadas de textura)

Observa que las coordenadas (objetos Point*) son necesarios para posicionar cada vértice. Los otros datos son opcionales, dependiendo de cómo se renderice el primitivo. Por ejemplo, se podría definir un color (un objeto Color*) para cada vértice y los colores del primitive se interpolan entre los colores de los vértices. Si se permite la iluminación, serán necesarias las superficies normales (y por lo tanto los objetos Vector*). Si se permite el mapeo de texturas, podrían necesitarse las coordenadas de texturas.
(Los objetos Quat* representan 'quaternions', que sólo se usan para transformaciones de matrices 3D avanzadas.)

Como todas las clases útiles descienden de las clases abstractas **Tuple***, es importante familiarizarse con sus constructores y métodos:

Constructores de **Tuple2f**

Paquete:

javax.vecmath

Las clases **Tuple*** casi nunca se utilizan directamente en programas Java pero proporcionan la base para las clases **Point***, **Color***, **Vector***, y **TexCoord***. En particular **Tuple2f** proporciona la base para **Point2f**, **Color2f**, y **TexCoord2f**. Los constructores listados aquí están disponibles para estas subclases. **Tuple3f** y **Tuple4f** tienen un conjunto similar de constructores:

Tuple2f()
Construye e inicializa un objeto **Tuple** con las coordenadas (0,0).
Tuple2f(float x, float y)
Construye e inicializa un objeto **Tuple** con las coordenadas x e y especificadas.
Tuple2f(float[] t)
Construye e inicializa un objeto **Tuple** desde el array especificado.
Tuple2f(Tuple2f t)
Construye e inicializa un objeto **Tuple** desde los datos de otro objeto **Tuple**.
Tuple2f(Tuple2d t)
Construye e inicializa un objeto **Tuple** desde los datos de otro objeto **Tuple**.

Lista Parcial de Métodos de **Tuple2f**

Paquete:

javax.vecmath

Las clases **Tuple*** casi nunca se utilizan directamente en programas Java pero proporcionan la base para las clases **Point***, **Color***, **Vector***, y **TexCoord***. En particular **Tuple2f** proporciona la base para **Point2f**, **Color2f**, y **TexCoord2f**. Los

métodos listados aquí están disponibles para estas subclases. **Tuple3f** y **Tuple4f** tienen un conjunto similar de métodos:

void set(float x, float y)
void set(float[] t)
Seleccionan el valor de este tuple desde los valores especificados.
boolean equals(Tuple2f t1)
Devuelven **true** si los datos del **Tuple t1** son iguales a los datos correspondientes de este tuple.
final void add(Tuple2f t1)
Selecciona el valor de este tuple al vector suma de si mismo y **Tuple t1**.
void add(Tuple2f t1, Tuple2f t2)
Selecciona el valor de este tuple al vector suma de los tuples t1 y t2.
void sub(Tuple2f t1, Tuple2f t2)
Selecciona el valor de este tuple al vector resta de los tuples t1 y t2 (this = t1 - t2).
void sub(Tuple2f t1)
Selecciona el valor de este tuple al vector resta de su mismo menos t1.
void negate()
Niega el valor de este vector.
void negate(Tuple2f t1)
Selecciona el valor de este tuple a la negación del tuple t1.
void absolute()
Selecciona todos los componentes de este tuple a sus valores absolutos.
void absolute(Tuple2f t)
Selecciona todos los componentes del tuple parámetros a sus valores absolutos, y sitúa los valores modificados en este tuple.

Hay sutíles, pero predecibles diferencias entre los constructores y métodos de **Tuple***, debido al número y tipo de los datos. Por ejemplo, **Tuple3d** difiere de **Tuple2f**, porque tiene un método constructor:
Tuple3d(double x, double y, double z);
que espera tres y no dos, parametros de pundo flotante de doble precision, no simple precisión.

Todas las clases **Tuple*** tienen miembros públicos. Para **Tuple2***, hay x e y. Para **Tuple3*** los miembros son x, y, y z. Para **Tuple4*** los miembros son x, y, z, y w.

Clases Point

Los objetos **Point*** normalmente representan coordenadas de un vértice, aunque también pueden representar la posición de una imagen, fuente de un punto de luz, localización espacial de un sonido, u otro dato posicional. Los constructores de las clases **Point*** son muy similares a los de **Tuple***, excepto en que devuelven objetos **Point***.

Lista Parcial de Métodos de **Point3f**

Paquete:

javax.vecmath

Las clases **Point*** descienden de las clases **Tuple***. Cada ejemplar de las clases **Point*** representa un solo punto en un espacio de dos, tres o cuatro dimensiones. Además de los métodos de **Tuple***, las clases **Point*** tienen métodos adicionales, algunos de los cuales podemos ver aquí:

float distance(Point3f p1)
Devuelve la distancia Euclideana entre este punto y el punto p1.
float distanceSquared(Point3f p1)
Devuelve el cuadrado de la distanciá Euclideana entre este punto y el punto p1.
float distanceL1(Point3f p1)
Devuelve la distancia L (Manhattan) entre este punto y el punto p1. La distancia L es igual a=
abs(x1 - x2) + abs(y1 - y2) + abs(z1 - z2)

Clases Color

Los objetos **Color*** representan un color, que puede ser para un vértice, propiedad de un material, niebla, u otro objeto visual. Los colores se especifican con **Color3*** o **Color4***, y sólo para datos de byte o coma flotante de simple precisión. Los objetos **Color3*** especifican un color como una combinación de valores rojo, verde y azul (RGB). Los objetos **Color4*** especifican un valor de transpariencia, además del RGB. (por defecto, los objetos **Color3*** son opacos). Para los tipos de datos de tamaño byte, los valores de colores van desde 0 hasta 255 inclusives. Para tipos de datos de coma flotante de simple precisión, los valores van entre 0,0 y 1,0 inclusives.

De nuevo, los constructores para las clases **Color*** son similares a los de **Tuple***, excepto en que devuelven objetos **Color***. Las clases **Color*** no tienen métodos adicionales, por eso sólo tratan con los métodos que heredan de sus superclases **Tuple***.

Algunas veces es conveniente crear constantes para los colores que se usan repetidamente en la creacción de objetos visuales. Por ejemplo,

```
Color3f rojo = new Color3f(1.0f, 0.0f, 0.0f);
```

ejemplariza el objeto **Color3f** rojo que podría usarse varias veces. Podría ser útil crear una clase que contenga varias cosntantes de colores. Abajo podemos ver un ejemplo de una de estas clases que aparece en el Fragmento de Código 2-1.

```
1. import javax.vecmath.*;
2.
3. class ColorConstants{
4.     public static final Color3f rojo = new Color3f(1.0f,0.0f,0.0f);
5.     public static final Color3f verde = new Color3f(0.0f,1.0f,0.0f);
6.     public static final Color3f azul = new Color3f(0.0f,0.0f,1.0f);
7.     public static final Color3f amarillo = new Color3f(1.0f,1.0f,0.0f);
8.     public static final Color3f cyan = new Color3f(0.0f,1.0f,1.0f);
9.     public static final Color3f magenta = new Color3f(1.0f,0.0f,1.0f);
10.    public static final Color3f blanco = new Color3f(1.0f,1.0f,1.0f);
11.    public static final Color3f negro = new Color3f(0.0f,0.0f,0.0f);
12. }
```

Clases Vector

Los objetos **Vector*** frecuentemente representan superficies normales en vértices aunque también pueden representar la dirección de una fuente de luz o de sonido. De nuevo, los constructores de las clases **Vector*** son similares a los de **Tuple***. Sin embargo, los objetos **Vector*** añaden muchos métodos que no se encuentran en las clases **Tuple***.

Lista Parcial de Métodos de **Vector3f**

Paquete:

javax.vecmath

Las clases **Vector*** descienden de las clases **Tuple***. Cada ejemplar de las clases **Vector*** representa un solo vector en un espacio de dos, tres o cuatro dimensiones. Además de los métodos de **Tuple***, las clases **Vector*** tienen métodos adicionales, algunos de los cuales podemos ver aquí:

float length()
Devuelve la longitud de este vector.
float lengthSquared()
Devuelve el cuadrado de la longitud de este vector.
void cross(Vector3f v1, Vector3f v2)
Selecccciona este vector para que sea el producto cruzado de los vectores v1 y v2.
float dot(Vector3f v1)
Calcula y devuelve el punto del producto de este vector y el vector v1.
void normalize()
Normaliza este vector.
void normalize(Vector3f v1)
Selecciona el valor de este vector a la normalización del vector v1.
float angle(Vector3f v1)
Devuelve el ángulo en radianes entre este vector y el vectorv1, el valor devuelto está restringido al ango [0,PI].

Clases TexCoord

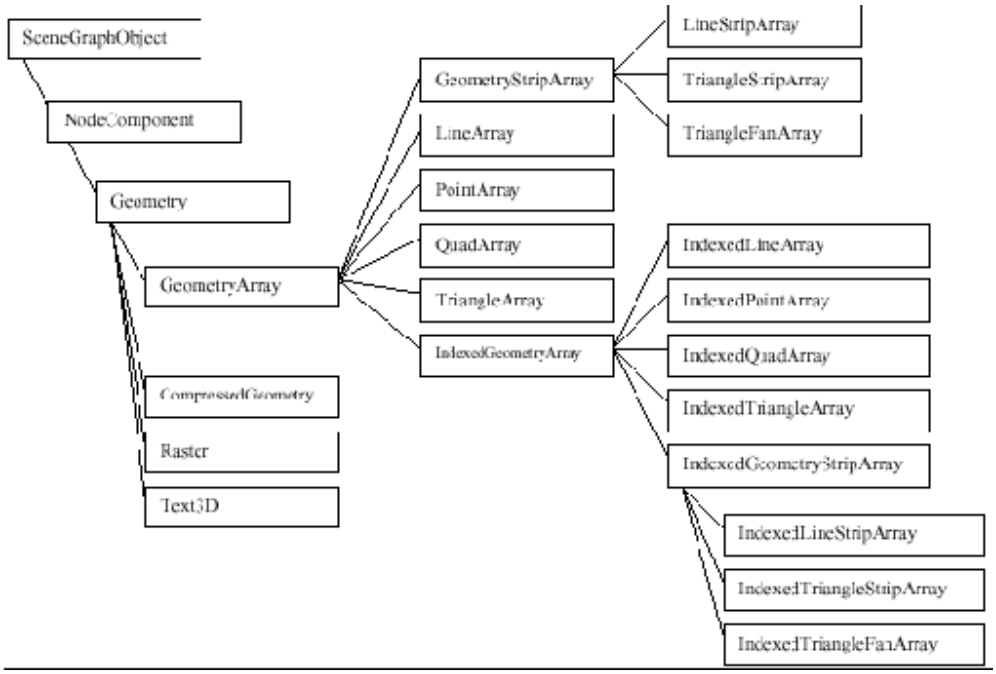
Hay sólo dos clases **TexCoord*** que pueden usarse para representar las coordenadas de textura de un vértice: **TexCoord2f** y **TexCoord3f**. **TexCoord2f** mantiene las coordenadas de textura como una pareja de coordenadas (s, t); **TexCoord3f** como un trio (s, t, r).
Los constructores para las clases **TexCoord*** también son similares a los de **Tuple***. como las clases **Color***, las clases **TexCoord*** tampoco tienen métodos adicionales, por eso sólo tratan con los métodos que heredan de sus superclases.

Clases Geometry

En los gráficos 3D por ordenador, todo, desde el más sencillo triángulo al más complicado de los modelos Jumbo, está modelado y renderizado con datos basados en vértices. Con Java 3D, cada objeto **Shape3D** debería llamar a su método **setGeometry()** para referenciar uno y sólo uno objeto **Geometry**. Para ser más preciso, **Geometry** es una superclase abstracta, por eso los objetos referenciados son ejemplares de una de sus subclases.
Estas subclases se dividen en tres categorías principales:

- Geometría basada en vértices no indexados (cada vez que se renderiza un objeto visual, sus vértices sólo podrían usarse una vez)
- Geometría basada en vértices indexados (cada vez que se renderiza un objeto visual, sus vértices se podrían reutilizar)
- Otros objetos visuales (las clases **Raster**, **Text3D**, y **CompressedGeometry**)

Esta sección cubre las dos primeras categorías. El árbol de clases de **Geometry** y sus subclases se muestran en la Figura 2-10:



Clase GeometryArray

Como se podría deducir de los nombres de las clases, las subclases de **Geometry** podrían usarse para especificar puntos, líneas y polígonos rellenos (triángulos y cuadriláteros). Estos primitivos basados en vértices son subclases de la clases abstracta **GeometryArray**, que indica que cada una tiene un array que mantine los datos de los vértices.

Por ejemplo, si se usa un objeto **GeometryArray** para especificar un triángulo, se define un array de tres elementos: un elemento para cada vértice. Cada elemento de este array mantiene la localización de las coordenbadas para su vértice (que puede estar definido con un objeto **Point*** o similar). Además de la localización de las coordenadas, se pueden definir otros tres arrays opcionales para almacenar el color, la superficie normal, y las coordenadas de textura. Estos arrays que contienen las coordenadas, los colores, las superficies normales y las coordenadas de texturas, son los "arrays de datos".

Hay tres pasos en la vida de un objeto **GeometryArray**:

1. Construcción de un objeto vacío.
2. Rellenar el objeto con datos.

3. Asociar (referenciar) el objeto desde (uno o más) objetos **Shape3D**.

Paso 1: Construcción de un objeto GeometryArray vacío

Cuando se construye por primera vez un objeto **GeometryArray**, se deben definir dos cosas:

- el número de vértices (arrays de elementos) necesarios.
- el tipo de datos (coordenadas de localización, color, superficie normal, y/o coordenadas de textura) a almacenar en cada vértice. Esto se llama formato del vértice.

Hay un sólo constructor para **GeometryArray**:

Constructor de **GeometryArray**

GeometryArray(int vertexCount, int vertexFormat)

Construye un objeto **GeometryArray** vacío con el número de vértices y su formato especificado. Se pueden añadir banderas "OR" para describir los datos de cada vértice. Las constantes usasas para especificar el formato son:

- **COORDINATES:**
  Especifica que este array de vértice contiene coordenadas. Es obligatorio.
- **NORMALS:**
  Especifica que este array de vértice contiene superficies normales.
- **COLOR_3:**
  Especifica que este array de vértice contiene colores sin transparencia.
- **COLOR_4:**
  Especifica que este array de vértice contiene colores con transparencia.
- **TEXTURE_COORDINATE_2:**
  Especifica que este array de vértice contiene coordenadas de textura 2D.
- **TEXTURE_COORDINATE_3:**
  Especifica que este array de vértice contiene coordenadas de textura 3D.

Por cada bandera seleccionada, se crea internamente en el objeto **GeometryArray** su correspondiente array. Cada uno de estos arrays tiene el tamaño **vertexCount**.

Veamos cómo trabaja este constructor, pero primero recordemos que **GeometryArray** es una clase abstracta. Por lo tanto, realmente llamamos al constructor de una de sus subclases, por ejemplo **LineArray**. (Un objeto **LineArray** describe un conjunto de vértices, y cada dos vértices definen los puntos finales de

una línea. El constuctor y otros métodos de **LineArray** son muy similares a los de su superclase **GeometryArray**. **LineArray** se explica con más detalle más adelante.)

El Fragmento de Código 2-4 muestra la clase **Axis** del programa AxisApp.java que usa varios objetos **LineArray** para dibujar líneas que representan los ejes X, Y y Z. El objeto **axisXlines**, crea un objeto con dos vértices (para dibujar una línea entre ellos), con sólo los datos de las coordenadas de localización. El objeto **axisYLines** también tiene dos vértices, pero permite color RGB, así como las coordenadas de posición de cada vértice. Por lo tanto, la línea del eje Y podría dibujarse con colores interpolados desde un vértice hasta el otro. Finalmente, el objeto **axisZLines** tiene diez vértices con coordenadas y datos de color para cada vértice. Se podrían dibujar cinco líneas con colores interpolados, una línea por cada pareja de vértices. Observa el uso de la operación "OR" para los formatos de los vértices de los ejes Y y Z

Fragmento de código 2-4, Consctructores de GeometryArray.

```
1.  // construye un objeto que representa el eje X
2.  LineArray axisXLines= new LineArray (2, LineArray.COORDINATES);
3.
4.  // construye un objeto que representa el eje Y
5.  LineArray axisYLines = new LineArray(2, LineArray.COORDINATES
6.          | LineArray.COLOR_3);
7.
8.  // construye un objeto que representa el eje Z
9.  LineArray axisZLines = new LineArray(10, LineArray.COORDINATES
10.         | LineArray.COLOR_3);
```

¡Cuidado! la clase **Axis** de AxisApp.java es diferente de la clase **Axis** definida en Axis.java, que sólo usa un objeto **LineArray**. Debemos asegurarnos de coger la clase correcta.

- **Paso 2: Rellenar con Datos el Objeto GeometryArray**

Después de construir el objeto **GeometryArray**, asignamos valores a los arrays, correspondiendo a los formatos de los vértices asignados. Esto se podría hacer vértice por vértice, o usando un array para aisgnar datos a muchos vértices con una sóla llamada de método. Los métodos disponibles son:

Lista Parcial de Métodos de **GeometryArray**

GeometryArray es la superclase para **PointArray**, **LineArray**, **TriangleArray**, **QuadArray**, **GeometryStripArray**, y **IndexedGeometryArray**.

void setCoordinate(int index, float[] coordinate)
void setCoordinate(int index, double[] coordinate)
void setCoordinate(int index, Point* coordinate)

Selecciona las coordenadas asociadas con el vértice en el índice especificado para este objeto.

void setCoordinates(int index, float[] coordinates)
void setCoordinates(int index, double[] coordinates)
void setCoordinates(int index, Point*[] coordinates)

Selecciona las coordenadas asociadas con los vértices empezando por el índice especificado para este objeto.

void setColor(int index, float[] color)
void setColor(int index, byte[] color)
void setColor(int index, Color* color)

Selecciona el color asociado con el vértice en el índice especificado para este objeto.

void setColors(int index, float[] colors)
void setColors(int index, byte[] colors)
void setColors(int index, Color*[] colors)

Selecciona los colores asociados con los vértices empezando por el índice especificado para este objeto.

void setNormal(int index, float[] normal)
void setNormal(int index, Vector* normal)

Selecciona la superficie normal asociada con el vértice en el índice especificado para este objeto

void setNormals(int index, float[] normals)

void setNormals(int index, Vector*[] normals)

Selecciona las superficies normales asociadas con los vértices empezando por el índice especificado para este objeto.

void setTextureCoordinate(int index, float[] texCoord)

void setTextureCoordinate(int index, Point* coordinate)

Selecciona la coordenada de textura asociada con el vértice en el índice especificado para este objeto.

void setTextureCoordinates(int index, float[] texCoords)

void setTextureCoordinates(int index, Point*[] texCoords)

Selecciona las coordenadas de textura asociadas con los vértices empezando por el índice especificado para este objeto.

El Fragmento de Código 2-5 usa los métodos de **GeometryArray** para almacenar coordenadas y colores en los objetos **LineArray**, El objeto **axisXLines** sólo llama al método setCoordinate() para almacenar los datos de las coordenadas de posición. El objeto **axisYLines** llama tanto a setColor() y setCoordinate() para cargar los valores del color RGB y las coordenadas de posición. Y el objeto **axisZLines** llama diez veces a setCoordinate() una para cada vértice y llama una vez a setColors() para cargar todos los vértices con una sola llamada:

Fragmento de código 2-5, Almacenar Datos en un Objeto GeometryArray.

```
1.  axisXLines.setCoordinate(0, new Point3f(-1.0f, 0.0f, 0.0f));
2.  axisXLines.setCoordinate(1, new Point3f( 1.0f, 0.0f, 0.0f));
3.
4.  Color3f red = new Color3f(1.0f, 0.0f, 0.0f);
5.  Color3f green = new Color3f(0.0f, 1.0f, 0.0f);
6.  Color3f blue = new Color3f(0.0f, 0.0f, 1.0f);
7.  axisYLines.setCoordinate(0, new Point3f( 0.0f,-1.0f, 0.0f));
8.  axisYLines.setCoordinate(1, new Point3f( 0.0f, 1.0f, 0.0f));
9.  axisYLines.setColor(0, green);
10. axisYLines.setColor(1, blue);
11.
12. axisZLines.setCoordinate(0, z1);
13. axisZLines.setCoordinate(1, z2);
```

```
14.  axisZLines.setCoordinate(2, z2);
15.  axisZLines.setCoordinate(3, new Point3f( 0.1f, 0.1f, 0.9f));
16.  axisZLines.setCoordinate(4, z2);
17.  axisZLines.setCoordinate(5, new Point3f(-0.1f, 0.1f, 0.9f));
18.  axisZLines.setCoordinate(6, z2);
19.  axisZLines.setCoordinate(7, new Point3f( 0.1f,-0.1f, 0.9f));
20.  axisZLines.setCoordinate(8, z2);
21.  axisZLines.setCoordinate(9, new Point3f(-0.1f,-0.1f, 0.9f));
22.
23.  Color3f colors[] = new Color3f[9];
24.  colors[0] = new Color3f(0.0f, 1.0f, 1.0f);
25.  for(int v = 0; v < 9; v++)
26.  colors[v] = red;
27.  axisZLines.setColors(1, colors);
```

El color por defecto para los vértices de un objeto **GeometryArray** es blanco, a menos que especifiquemos **COLOR_3** o **COLOR_4** en el formato del vértice. Cuando se especifica cualquiera de estos valores, el color por defecto del vértice es negro. Cuando se renderizan líneas o polígonos rellenos con diferentes colores en cada vértice, los colores se sombrean (interpolan) entre los vértices usando un sombreado **Gouraud**.

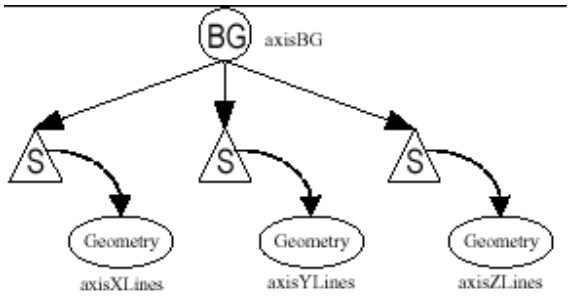
Paso 3: Hacer que los Objetos Shape3D Referencien a los Objetos GeometryArray

Finalmente, el Fragmento de Código 2-6 muestra cómo se referencian los objetos **GeometryArray** por objetos **Shape3D** recien creados. A su vez, los objetos **Shape3D** se añaden a un **BranchGroup**, que en algún lugar se añade al escenario gráfico general. (Al contrario que los objetos **GeometryArray**, que son **NodeComponents**, **Shape3D** es una subclase de **Node**, por eso pueden ser añadidos como hijos al escenario gráfico.)

Fragmento de código 2-6, Objetos GeometryArray referenciados por objetos Shape3D.

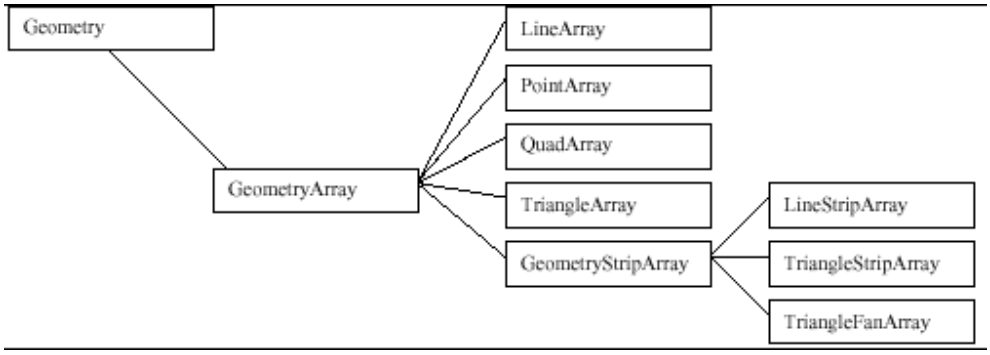
```
1.  axisBG = new BranchGroup();
2.
3.  axisBG.addChild(new Shape3D(axisYLines));
4.  axisBG.addChild(new Shape3D(axisZLines));
```

La Figura 2-11 muestra el escenario gráfico parcial creado por la clase **Axis** en AxisApp.java.



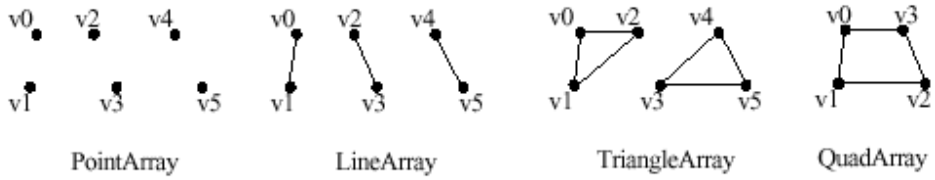
Subclases de GeometryArray

Como se explicó en la sección anterior, la clase **GeometryArray** es una superclase abstracta para subclases más útiles, como **LineArray**. La Figura 2-12 muestra el árbol de subclases de **GeometryArray**. La principal distinción entre estas subclases es cómo el renderizador Java 3D decide renderizar sus vértices.



La Figura 2-13 muestra ejemplos de las cuatro subclases de **GeometryArray**: **PointArray**, **LineArray**, **TriangleArray**, y **QuadArray** (las únicas que no son también subclases de **GeometryStripArray**). En esta figura, los tres conjuntos más a la izquierda muestran los mismos seis vértices renderizando seis puntos, tres líneas, o dos triángulos. La cuarta imagen muestra cuatro vértices difiniendo un

cuadrilátero. Observa que ninguno de los vértices están compartido: cada línea o polígono relleno se renderiza independiente,ente de cualquier otra.



Por defecto, se rellena el interior de los triángulos y cuadriláteros. En la última sección, aprenderemos los atributos que pueden influenciar en el modo de renderizado de los primivitivos rellenos.

Estas cuatro subclases heredan sus constructores y métodos de **GeometryArray**. Abajo podemos ver sus constructores, para sus métodos, podemos volver atrás a la lista de los métodos de **GeometryArray**.

Constructores de Subclases de **GeometryArray**

Construyen objetos vacíos con el número de vértices especificados y el formato de los vértices. Se pueden añadir banderas "OR" para describir los datos de cada vértice. Las banderas de formato son las mismas que las definidas en la superclase **GeometryArray**.

PointArray(int vertexCount, int vertexFormat)
LineArray(int vertexCount, int vertexFormat)
TriangleArray(int vertexCount, int vertexFormat)
QuadArray(int vertexCount, int vertexFormat)

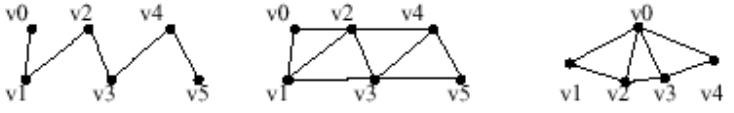
Para ver el uso de estos constructores y métodos, podemos volver a cualquiera de los fragmentos de código 2-4, 2-5 o 2-6, que usan objetos **LineArray** .

Si estámos dibujando cuadriláteros, debemos tener cuidado de no crear los vértices en una geometría cóncava, auto-intereseccionada, o geometría no plana. Si lo hacemos, podrían no renderizarse apropiadamente.

Subclases de GeometryStripArray

Las cuatro subclases de **GeometriArray** descritas anteriormente no permite reutilizar vértices. Algunas configuraciones geométricas invitan a reutilizar los vértices, por eso podrían resultar clases especializadas para un mejor rendimiento del renderizado.

GeometryStripArray es una clase abstracta de la que se derivan tipos primitivos (para crear líneas y superficies compuestas). **GeometryStripArray** es la superclase de **LineStripArray**, **TriangleStripArray**, y **TriangleFanArray**. La Figura 2-14 muestra un ejemplar de cada tipo y cómo se reutilizan los vértices. **LineStripArray** renderiza las líneas conectadas. **TriangleStripArray** resulta en triángulos que comparten un lado, reusando el vértice renderizado más recientemente. **TriangleFanArray** reutiliza el primer vértice en su lámina.



GeometryStripArray tiene un constructor distinto al de **GeometryArray**. El constructor **GeometryStripArray** tiene un tercer parámetro, que es un array contador de vértices para cada lámina, permitiendo que un sólo objeto mantenga varias láminas. (**GeometryStripArray** también presenta un par de métodos de consulta, getNumStrips() y getStripVertexCounts(), que raramente se usan.)

Constructores de Subclases de **GeometryStripArray**

Construyen objetos vacíos con el número de vértices especificados y el formato de los vértices. Se pueden añadir banderas "OR" para describir los datos de cada vértice. Las banderas de formato son las mismas que las definidas en la superclase **GeometryArray**. Se soportan múltiples láminas. La suma de los contadores de vértices para todas las láminas (del array **stripVertexCounts**) debe ser igual al contador de todos los vértices (vtxCount).

LineStripArray(int vtxCount, int vertexFormat, int stripVertexCounts[])

TriangleStripArray(int vtxCount, int vertexFormat, int stripVertexCounts[]))
TriangleFanArray(int vtxCount, int vertexFormat, int stripVertexCounts[]))

Observa que Java 3D no soporta primitivos rellenos con más de cuatro lados. El programador es responsable de usar mosaicos para descomponer polígonos más complejos en objetos Java 3D. La clases de utilidad **Triangulator** convierte polígonos complejos en triángulos

La clase **Triangulator**

Paquete:

com.sun.j3d.utils.geometry

Se usa para convertir polígonos no triángulares en triángulos para renderizarlos con Java 3D. Los polígonos pueden ser cóncavos, no planos y pueden contener agujeros (puedes ver GeometryInfo.setContourCounts()). Los polígonos no planos se proyectan al plano más cercano.

Sumario de Constructores

Triangulator()
Crea un objeto **Triangulator**.

Sumario de Métodos

void triangulate(GeometryInfo ginfo)
Esta rutina convierte el objeto **GeometryInfo** desde un tipo primitivo **POLYGON_ARRAY** a un tipo primitivo **TRIANGLE_ARRAY** usándo técnicas de descomposición de polígonos.
Parámetros:

- ginfo - el objeto com.sun.j3d.utils.geometry.GeometryInfo a triangular.

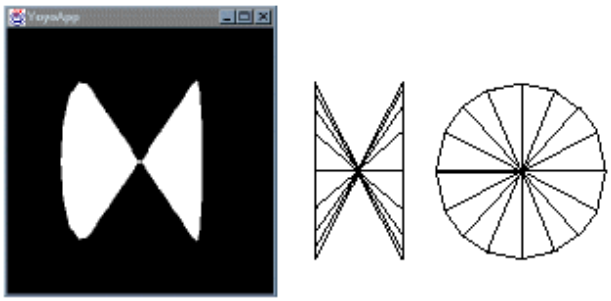
Ejemplo de uso:

```
Triangulator tr = new Triangulator();
tr.triangulate(ginfo); // ginfo contains the geometry
shape.setGeometry(ginfo.getGeometryArray()); // shape is a Shape3D
```

El código del Yo-yo Demuestra TriangleFanArray

El objeto **Yoyo** del programa YoyoApp.java muestra cómo usar un objeto **TriangleFanArray** para modelar la geometría de un yo-yo. El **TriangleFanArray** contiene cuatro abanicos: dos caras exteriores (discos circulares) y dos caras internas (conos). Sólo se necesita un objeto **TriangleFanArray** para representar los cuatro abanicos.

La Figura 2-15 muestra tres renderizaciones del **TriangleFanArray**. La primera vista muestra su renderizado por defecto, como polígonos rellenos blancos. Sin embargo, es díficil ver los detalles, especialmente la localización de los vértices. Para mostrar mejor los triángulos, las otras dos vistas muestran el **TriangleFanArray** con sus vértices conectados con líneas. Para renderizar lo que serían los polígonos rellenos con líneas, puedes ver la clases **PolygonAttributes** más adelante.



En el Fragmento de Código 2-7 el método yoyoGeometry() crea y devuelve el **TriangleFanArray** deseado. Las líneas 15-18 calculan los puntos centrales para los cuatro abanicos. Cada abanico tiene 18 vértices, que se calculan en las líneas 20-28. Las líneas 30-32 construyen el objeto **TriangleFanArray** vacío, y la línea 34 es donde las coordenadas calculadas préviamente (en las líneas 15-28) se almacenan en el objeto.

Fragmento de Código 2-7, el metodo yoyoGeometry() crea un objeto TriangleFanArray

```
1. private Geometry yoyoGeometry() {
2.
3.    TriangleFanArray tfa;
4.    int N = 17;
5.    int totalN = 4*(N+1);
6.    Point3f coords[] = new Point3f[totalN];
```

```
7.    int stripCounts[] = {N+1, N+1, N+1, N+1};
8.    float r = 0.6f;
9.    float w = 0.4f;
10.   int n;
11.   double a;
12.   float x, y;
13.
14.   // set the central points for four triangle fan strips
15.   coords[0*(N+1)] = new Point3f(0.0f, 0.0f, w);
16.   coords[1*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);
17.   coords[2*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);
18.   coords[3*(N+1)] = new Point3f(0.0f, 0.0f, -w);
19.
20.   for (a = 0,n = 0; n < N; a = 2.0*Math.PI/(N-1) * ++n){
21.      x = (float) (r * Math.cos(a));
22.      y = (float) (r * Math.sin(a));
23.
24.      coords[0*(N+1)+N-n] = new Point3f(x, y, w);
25.      coords[1*(N+1)+n+1] = new Point3f(x, y, w);
26.      coords[2*(N+1)+N-n] = new Point3f(x, y, -w);
27.      coords[3*(N+1)+n+1] = new Point3f(x, y, -w);
28.   }
29.
30.   tfa = new TriangleFanArray (totalN,
31.   TriangleFanArray.COORDINATES,
32.       stripCounts);
33.
34.   tfa.setCoordinates(0, coords);
35.
36.   return tfa;
37.} // end of method yoyoGeometry in class Yoyo
```

El yo-yo totalmente blanco es sólo un punto de arranque. La Figura 2-16 muestra un objeto similar, modificado para incluir colores en cada vértice. El método yoyoGeometry() modificado, que incluye colores en el objeto **TriangleFanArray**, se muestra en el Fragmento de Código 2-8. Las líneas 23-26, 36-39 y 46 especifican los valores de color para cada vértice.

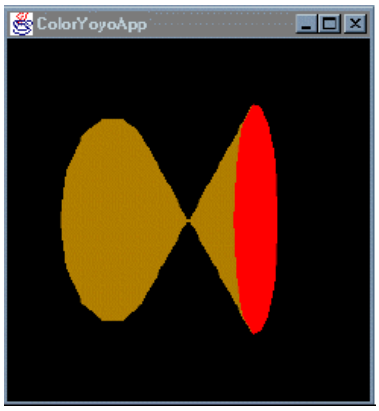
Existen más posibilidades para especificar la apariencia de un objeto visual a través del uso de luces, texturas, y propiedades de materiales de un objeto visual.

Fragmento de Código 2-8, Método yoyoGeometry() Modificado para añadir colores

```
1. private Geometry yoyoGeometry() {
2.
3.    TriangleFanArray tfa;
4.    int N = 17;
5.    int totalN = 4*(N+1);
6.    Point3f coords[] = new Point3f[totalN];
7.    Color3f colors[] = new Color3f[totalN];
8.    Color3f red = new Color3f(1.0f, 0.0f, 0.0f);
9.    Color3f yellow = new Color3f(0.7f, 0.5f, 0.0f);
10.   int stripCounts[] = {N+1, N+1, N+1, N+1};
11.   float r = 0.6f;
12.   float w = 0.4f;
13.   int n;
14.   double a;
15.   float x, y;
16.
17.   // set the central points for four triangle fan strips
18.   coords[0*(N+1)] = new Point3f(0.0f, 0.0f, w);
19.   coords[1*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);
20.   coords[2*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);
21.   coords[3*(N+1)] = new Point3f(0.0f, 0.0f, -w);
22.
23.   colors[0*(N+1)] = red;
24.   colors[1*(N+1)] = yellow;
25.   colors[2*(N+1)] = yellow;
26.   colors[3*(N+1)] = red;
27.
28.   for(a = 0,n = 0; n < N; a = 2.0*Math.PI/(N-1) * ++n){
29.      x = (float) (r * Math.cos(a));
30.      y = (float) (r * Math.sin(a));
31.      coords[0*(N+1)+n+1] = new Point3f(x, y, w);
32.      coords[1*(N+1)+N-n] = new Point3f(x, y, w);
```

```
33.      coords[2*(N+1)+n+1] = new Point3f(x, y, -w);
34.      coords[3*(N+1)+N-n] = new Point3f(x, y, -w);
35.
36.      colors[0*(N+1)+N-n] = red;
37.      colors[1*(N+1)+n+1] = yellow;
38.      colors[2*(N+1)+N-n] = yellow;
39.      colors[3*(N+1)+n+1] = red;
40.  }
41.   tfa = new TriangleFanArray (totalN,
42.         TriangleFanArray.COORDINATES|TriangleFanArray.COLOR_3,
43.         stripCounts);
44.
45.   tfa.setCoordinates(0, coords);
46.   tfa.setColors(0,colors);
47.
48.   return tfa;
49. } // end of method yoyoGeometry in class Yoyo
```

Habrás observado las diferencias entre las líneas 36 a 39. El código se ha escrito para hacer la cara frontal de cada triángulo en la geometría la parte exterior del yo-yo.



Subclases de IndexedGeometryArray

Las subclases de **GeometryArray** descritas anteriormente declaran los vértices de forma borrosa. Solo las subclases de **GeometryStripArray** tienen incluso limitada la reutilización e vértices. Muchos objetos geométricos invitan a la reutilización de vértices. Por ejemplo, para definir un cubo, cada uno de sus ocho vértices se usa

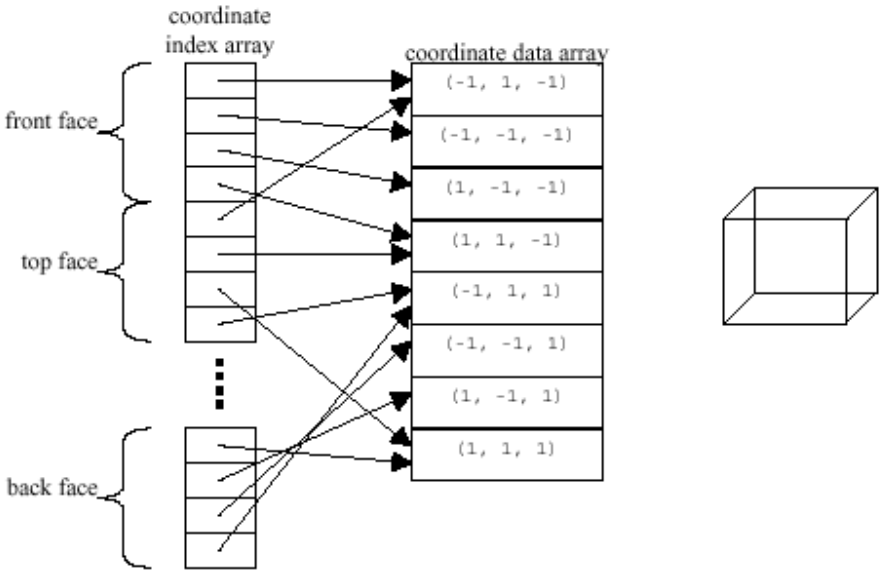
por tres diferentes cuadrados. En el peor de los casos, un cubo requiere especificar 24 vértices, aunque sólo ocho son realmente necesarios (16 de los 24 son redundantes).

Los objetos **IndexedGeometryArray** proporcionan un nivel de extra de indirección, por eso se puede evitar los vértices redundantes. Todavía se deben proporcionar los arrays de información basada en vértices, pero los vértices se pueden almacenar en cualquier orden, y cualquier vértice se puede reutilizar durante el renderizado. A estos arrays que contienen información sobre las coordenadas, el color, etc. se les llama "Arrays de datos".

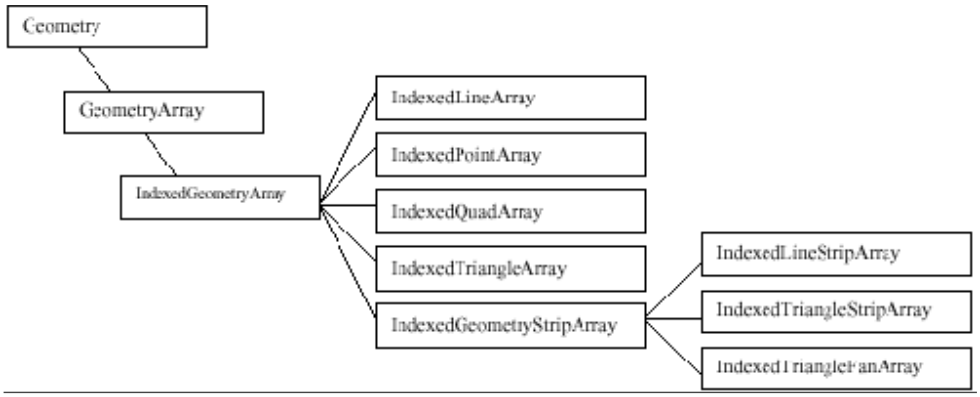
Sin embargo, los objetos **IndexedGeometryArray** también necesitan arrays adicionales ("arrays de índices") que contienen índices a los "arrays de datos". Hay hasta cuatro "arrays de índice": índices de coordenadas, índices de colores, índices de superficies normales, e índices de coordenadas de textura, que cooresponden con los "arrays de datos". El número de arrays de índices es siempre igual al número de arrays de datos. El número de elementos en cada array de índice es el mismo y normalmente mayor que el número de elementos en cada array de datos.

El "array de índices" podría tener múltiples referencias al mismo vértice en el "array de datos". Los valores en estos "arrays de índices" determinan el orden en que se accede a los datos del vértice durante el renderizado. La Figura 2-17 muestra como ejemplo la relación entre los arrays de índice y de coordenadas para un cubo.

Merece la pena mencionar que hay que pagar un precio por la reutilización de los vértices proporcionada por la geometría indexada - lo pagamos en rendimiento. El indexado de geometrías en el momento de la renderización añade más trabajo al proceso de renderizado. Si el rendimiento es un problema, debemos usar láminas siempre que sea posible y evitar la geometría indexada. La geometría indexada es útil cuando la velocidad no es crítica y tenemos alguna memoria que ganar usándola, o cuando la indexación proporciona programación de conveniencia.



Las subclases de **IndexedGeometryArray** son paralelas a las subclases de **GeometryArray**. En la Figura 2-18 podemos ver el árbol de herencia de I**ndexedGeometryArray**.



Los constructores para **IndexedGeometryArray**, **IndexedGeometryStripArray**, y sus subclases son similares a los constructores de **GeometryArray** y **GeometryStripArray**. Las clases de datos indexados tienen un parámetro adicional para definir cúantos índices se usan para describir la geometría (el número de elemento en el array de índices).

Constructores de Sublcases de I**ndexedGeometryArray**

Construyen un objeto vacío con el número de vértices especificado, el formato de los vértices, y el número de índices en este array.

IndexedGeometryArray(int vertexCount, int vertexFormat, int indexCount)
IndexedPointArray(int vertexCount, int vertexFormat, int indexCount)
IndexedLineArray(int vertexCount, int vertexFormat, int indexCount)
IndexedTriangleArray(int vertexCount, int vertexFormat, int indexCount)
IndexedQuadArray(int vertexCount, int vertexFormat, int indexCount)

Constructores de Subclases de I**ndexedGeometryStripArray**

Construye un objeto vacío con el número de vertices especificado, el formado de los vértices, el número de índices de este array, y un array contador de vértices por cada lámina.

IndexedGeometryStripArray(int vc, int vf, int ic, int stripVertexCounts[]))
IndexedLineStripArray(int vc, int vf, int ic, int stripVertexCounts[]))
IndexedTriangleStripArray(int vc, int vf, int ic, int stripVertexCounts[]))
IndexedTriangleFanArray(int vc, int vf, int ic, int stripVertexCounts[]))

I**ndexedGeometryArray**, I**ndexedGeometryStripArray**, y sus subclases heredan métodos desde **GeometryArray** y **GeometryStripArray** para cargar los "arrays de datos". Las clases de datos indexados han añadido métodos para cargar índices dentro de los "arrays de índices".

Lista Parcial de Métodos de **IndexedGeometryArray**

void setCoordinateIndex(int index, int coordinateIndex)
Selecciona el índice de coordenada asociado con el vértice en el índice especificado para este objeto.
void setCoordinateIndices(int index, int[] coordinateIndices)
Selecciona los índices de coordenadas asociados con los vértices que empiezan en el índice especificado para este objeto.
void setColorIndex(int index, int colorIndex)
Selecciona el índice de color asociado con el vértice en el índice especificado para este objeto.
void setColorIndices(int index, int[] colorIndices)
Selecciona los índices de colores asociados con los vértices que empiezan en el índice especificado para este objeto.

void setNormalIndex (int index, int normalIndex)

Selecciona el índice de superficie normal asociado con el vértice en el índice especificado para este objeto.

void setNormalIndices (int index, int[] normalIndices)

Selecciona los índices de superficies normales asociados con los vértices que empiezan en el índice especificado para este objeto.

void setTextureCoordinateIndex (int index, int texCoordIndex)

Selecciona el índice de coordenada de textura asociado con el vértice en el índice especificado para este objeto.

void setTextureCoordinateIndices (int index, int[] texCoordIndices)

Selecciona los índices de coordenadas texturas asociados con los vértices que empiezan en el índice especificado para este objeto.

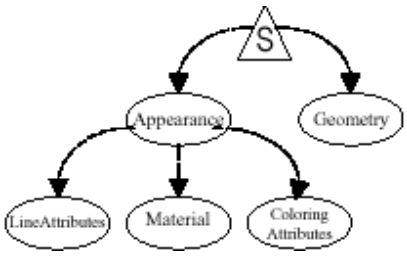
Axis.java es un ejemplo de IndexedGeometryArray

El fichero Axis.java define el objeto visual **Axis** muy útil para dibujar los ejes y el origen de un universo virtual. También sirve como ejemplo de geometría indexada. El objeto **Axis** define 18 vértices y 30 índices para especificar 15 líneas. Hay cinco líneas por eje para crear una sencilla flecha 3D.

Atributos y Apariencia

Los objetos **Shape3D** podrían referenciar tanto a objetos **Geometry** y **Appearance**. Como se explicó anteriormente, el objeto **Geometry** especifica la información para cada vértice de un objeto visual. La información por vértices de un objeto **Geometry** puede especificar el color de los objetos visuales. Los datos de un objeto **Geometry** normalmente son insuficientes para describir totalmente cómo es un objeto. En muchos casos, también se necesita un objeto **Appearance**.
Un objeto **Appearance** no contiene información sobre cómo debe aparecer un objeto **Shape3D**, pero si sabe donde encontrar esos datos. Un objeto **Appearance** (ya que es una subclase de **NodeComponent**) podría referenciar varios objetos de otras subclases de la clase abstracta **NodeComponent**. Por lo tanto la información

que describe la apariencia de un primitivo geométrico se dice que está almacenada dentro de un "paquete de apariencia", como se ve en la Figura 2-19.



Un objeto **Appearance** puede referenciar a varias subclases diferentes de **NodeComponent** llamados objetos de atributos de apariencia, incluyendo:

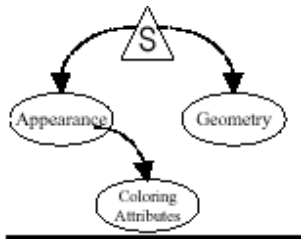
- PointAttributes
- LineAttributes
- PolygonAttributes
- ColoringAttributes
- TransparencyAttributes
- RenderingAttributes
- Material
- TextureAttributes
- Texture
- TexCoordGeneration

A un objeto **Appearance** con objetos atributos se le llama un paquete de apariencia. Para referenciar cualquiera de estos nodos componentes, un objeto **Appearance** tiene un método con un nombre óbvio. Por ejemplo, para que un objeto **Appearance** se refiera a un objeto **ColoringAttributes**, se usa el método Appearance.setColoringAttributes(). Un sencillo ejemplo de código de parecería está en Fragmento de código 2-9.

Fragmento de Código 2-9, Usando objetos Appearance y ColoringAttributes de NodeComponent .

```
1. ColoringAttributes ca = new ColoringAttributes();
2. ca.setColor (1.0, 1.0, 0.0);
3. Appearance app = new Appearance();
4. app.setColoringAttributes(ca);
5. Shape3D s3d = new Shape3D();
6. s3d.setAppearance (app);
7. s3d.setGeometry (someGeomObject);
```

En la Figura 2-20 podemos ver el escenario gráfico resultante de el código anterior.



- **NodeComponent Appearance**

Los dos siguientes bloques de referencia listan los constructores y otros métodos de la clase **Appearance**.

Constructor de **Appearance**

El constructor por defecto de **Appearance** crea un objeto con todas las referencias a objetos inicializadas a null. Los valores por defecto, para componentes con referencias nulas, normalmente son predecibles: puntos y líneas, se dibujan con un tamaño y anchura de 1 pixel y sin antialiasing, el color intrínseco es blanco, la transparencia desactivada, y el buffer de profundidad está activado y es accesible tanto para lectura como escritura.

Appearance()

Un componente **Appearance** normalmente referencia uno o más componentes atributo, llamando a los siguientes métodos:

Métodos de **Appearance** (Excluyendo iluminación y texturas)

Cada método selecciona su objeto **NodeComponent** correspondiente para que sea parte del paquete de apariencia actual.

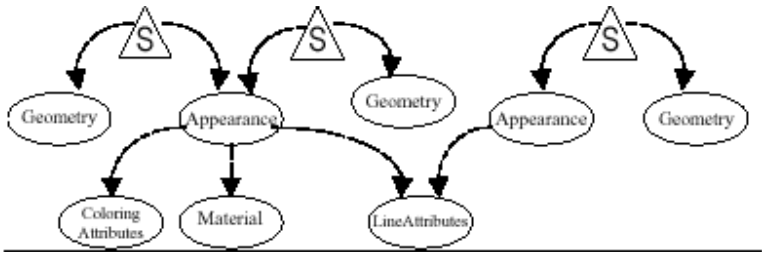
void setPointAttributes(PointAttributes pointAttributes)
void setLineAttributes(LineAttributes lineAttributes)
void setPolygonAttributes(PolygonAttributes polygonAttributes)
void setColoringAttributes(ColoringAttributes coloringAttributes)
void setTransparencyAttributes(TransparencyAttributes

transparencyAttributes)

void setRenderingAttributes(RenderingAttributes renderingAttributes)

Compartir Objetos NodeComponent

Es legal e incluso deseable que varios objetos referencien, y por lo tanto compartan, los mismos objetos **NodeComponent**. Por ejemplo, en la Figura 2-21. dos objetos **Shape3D** referencian al mismo componente **Appearance**. También, dos objetos **Appearance** diferentes comparten el mismo componente **LineAttributes**.



Compartir el mismo **NodeComponent** puede mejorar el rendimiento. Por ejemplo, si varios componentes **Appearance** comparten el mismo componente **LineAttributes**, lo que permite el antialias, el motor renderizador de Java 3D podría decidir agrupar el marco de trabajo antialias. Esto podría minimizar la activación y desactivación del antialias, lo que sería más rápido.

Observa que es ilegal que un Nodo tenga más demun padre. Sin embargo, como los **NodeComponents** son referenciados, no son objetos **Node**, por lo que realmente no tienen padres. Por lo tanto, los objetos **NodeComponent** pueden ser compartidos (referenciados) por cualquier número de otros objetos.

Clases Attribute

En esta sección se describen las seis primeras subclases de **NodeComponent** que pueden ser referenciadas por **Appearance** (excluyendo las de iluminación y texturas).

PointAttributes

Los objetos **PointAttributes** manejan el modo en que se redibujan los puntos primitivos. Por defecto, si un vértice se renderiza como un punto, rellena un único pixel. Podemos usar setPointSize() para hacer un punto más grande. Sin embargo, por defecto, un punto mayor se parece a un cuadrado, a menos que usemos setPointAntialiasingEnable(). Los puntos Antialiasing cambian los colores de los pixels para hacer que el punto parezca "redondeado" (o al menos, un cuadrado menos visible).

Constructores de **PointAttributes**

PointAttributes()

Crea un objeto componente que describe puntos de un pixel sin antialiasing.

PointAttributes(float pointSize, boolean state)

Crea un objeto componente que describe el tamaño de pixel para los puntos y si permite o no el antialiasing.

Métodos de **PointAttributes**

void setPointSize(float pointSize)

Describe el tamaño de pixels para los puntos.

void setPointAntialiasingEnable(boolean state)

Activa o desactiva el antialising de los puntos. Visualmente interesante sólo si el punto es mayor de un pixel.

LineAttributes

Los objetos **LineAttributes** cambian el modo en que se renderizan las líneas primitivas de tres formas. Por defecto, una línea se dibuja sólida rellena, de un pixel de ancho, y sin antialiasing. Podemos cambiar estos atributos llamando a los métodos setLinePattern(), setLineWidth(), y setLineAntialiasingEnable().

Constructores de **LineAttributes**

LineAttributes()

Crea un objeto componente que describe líneas rellenas de un pixel de ancho, sólidas rellenas, sin antialiasing.

LineAttributes(float pointSize, int linePattern, boolean state)

Crea un objeto componente que describe el tamaño de pixel para líneas, el patrón de uso para dibujo y si se activa o no el antialiasing.

Métodos de **LineAttributes**

void setLineWidth(float lineWidth)
Describe la anchura de pixels para líneas.
void setLinePattern(int linePattern)
donde **linePattern** es una de estas constantes: **PATTERN_SOLID** (por defecto), **PATTERN_DASH**, **PATTERN_DOT**, o **PATTERN_DASH_DOT**. Describe cómo se deberían rellenar los pixels de una línea.
void setLineAntialiasingEnable(boolean state)
Activa o desactiva el antialiasing.

PolygonAttributes

PolygonAttributes gobierna el modo en que se renderizan los polígonos primitivos de tres formas: cómo es rasterizado, si está recortado, y si se aplica un desplazamiento de profundidad especial. Por defecto, un polígono está relleno, pero setPolygonMode() puede cambiar el modo en el que se dibuja el polígono como un marco (líneas) o sólo con los puntos de los vértices. (En las últimas dos clases, **LineAttributes** o **PointAttributes** también afectaban a como se visualiza el primitivo). Se podría usar el método setCullFace() para reducir el número de polígonos que son renderizados. Si setCullFace() se selecciona a **CULL_FRONT** o **CULL_BACK**, como media, no se renderizadan la mitad de los polígonos.
Por defecto, los vértices se renderizan como marcos y los polígonos rellenos no siempre se rasterizan con los mismos valores de profundidad, lo que podría hacer el estrechamiento cuando el marco fuera totalmente visible. Con setPolygonOffset(), los valores de profundidad de los polígonos rellenos se pueden mover hacia el plato de imagen, para que el marco enmarque el objeto relleno de la forma apropiada. setBackFaceNormalFlip() es utíl para renderizar un polígono relleno, donde ambos lados del polígono van a ser sombreados.

Constructores de **PolygonAttributes**

PolygonAttributes()

Crea un objeto componente con polígonos rellenos por defecto, sin recortado y sin desplazamiento.

PolygonAttributes(int polygonMode, int cullFace, float polygonOffset)

Crea un objeto componente para renderizar polígonos como sus puntos, líneas o polígonos rellenos, con el recorte de caras y el desplazamiento especificados.

PolygonAttributes(int polygonMode, int cullFace,
float polygonOffset, boolean backFaceNormalFlip)

Crea un objeto componente similar al constructor anterior, pero tambien invierte cómo serán determinados los polígonos trasero y frontal.

Métodos de **PolygonAttributes**

void setCullFace(int cullFace)

donde **cullFace** es uno de los siguientes: **CULL_FRONT**, **CULL_BACK**, o **CULL_NONE**. Oculta (no renderiza) los polígonos de la cara frontal o trasera, o no recorta los polígonos en absoluto.

void setPolygonMode(int polygonMode)

donde **polygonMode** es uno de estos: **POLYGON_POINT**, **POLYGON_LINE**, o **POLYGON_FILL**. Renderizan los polígonos según sus puntos, sus líneas o polígonos rellenos (por defecto).

void setPolygonOffset(float polygonOffset)

donde **polygonOffset** es el desplazamiento del espacio de pantalla añadido para ajustar el valor de profundidad de los polígonos primitivos.

void setBackFaceNormalFlip(boolean backFaceNormalFlip)

donde **backFaceNormalFlip** determina si los vértices de los polígonos de las caras traseras deberían ser negados antes de iluminarlos. Cuando está bandera se selecciona a True y el recorte de la parte trasera está desactivado, un polígono se renderiza como si tuviera dos lados con oposicción normal.

ColoringAttributes

ColoringAttributes controla cómo se colorea cualquier primitivo. setColor() selecciona un color intrínseco, que en algunas situaciones específica el color del primitivo. También setShadeModel() determina si el color es interpolado entre primitivos (normalmente polígonos y líneas).

Constructores de **ColoringAttributes**

ColoringAttributes()

Crea un objeto componente usando blanco como el color intrínseco y **SHADE_GOURAUD** como el modelo de sombreado por defecto.

ColoringAttributes(Color3f color, int shadeModel)

ColoringAttributes(float red, float green, float blue, int shadeModel)

donde **shadeModel** es uno de **SHADE_GOURAUD**, **SHADE_FLAT**, **FASTEST**, o **NICEST**. Ambos constructores crean un objeto componente usando los parámetros especificados para el color intrínseco y el modelo de sombreado (en la mayoría de los casos **FASTEST** es también **SHADE_FLAT**, y **NICEST** es también **SHADE_GOURAUD**.)

Métodos de **ColoringAttributes**

void setColor(Color3f color)

void setColor(float red, float green, float blue)

Ambos métodos especifican el color intrínseco.

void setShadeModel(int shadeModel)

donde **shadeModel** es uno de estos: **SHADE_GOURAUD**, **SHADE_FLAT**, **FASTEST**, o **NICEST**. Especifica el modelo de sombreado para renderizar primitivos.

Como los colores también se pueden definir para cada vértice de un objeto **Geometry**, podría haber un confilcto con el cólor intrínseco definido por **ColoringAttributes**. En el caso de dicho conflicto, los colores definidos en el objeto **Geometry** sobreescriben al color intrínseco de **ColoringAttributes**. Si la iluminación está activada, también se ignora el color intrínseco de **ColoringAttributes**.

TransparencyAttributes

TransparencyAttributes maneja la transparencia de cualquier primitivo. setTransparency() define el valor de opacidad para el primitivo. setTransparencyMode() activa la transparencia y selecciona el tipo de rasterización usado para producir la transparencia.

Constructores de **TransparencyAttributes**

TransparencyAttributes()

Crea un objeto componente con el modo de transparencia de FASTEST.

TransparencyAttributes(int tMode, float tVal)

donde **tMode** es uno de **BLENDED**, **SCREEN_DOOR**, **FASTEST**, **NICEST**, o **NONE**, y **tVal** especifica la opacidad del objeto (0.0 denota total opacidad y 1.0, total transparencia). Crea un objeto componente con el método especificado para la renderización de transparencia y el valor de opacidad de la apariencia del objeto.

Métodos de **TransparencyAttributes**

void setTransparency(float tVal)

donde **tVal** especifca una opacidad de objeto donde (0.0 denota total opacidad y 1.0, total transparencia).

void setTransparencyMode(int tMode)

donde **tMode** (uno de **BLENDED**, **SCREEN_DOOR**, **FASTEST**, **NICEST**, o **NONE**) especifica cómo se realiza la transparencia.

RenderingAttributes

RenderingAttributes controla dos operaciones diferentes de renderizado pixel-a-pixel: el buffer de profundidad y el texteo alpha setDepthBufferEnable() y setDepthBufferWriteEnable() determinan si se usa y cómo se usa el buffer de profundidad para ocultar una superficie elininada. setAlphaTestValue() y setAlphaTestFunction() determinan si se usa y cómo la función alpha.

Constructores de **RenderingAttributes**

RenderingAttributes()

Crea un objeto componente que define estados de renderizado por-pixel con el buffer de profundidad activado y la función alpha desactivada.

RenderingAttributes(boolean depthBufferEnable,
boolean depthBufferWriteEnable,
float alphaTestValue, int alphaTestFunction)

donde **depthBufferEnable** activa y desactica las comparaciones del buffer de profundidad, **depthBufferWriteEnable** activa y desactiva la escritura en el buffer de profundidad, **alphaTestValue** se usa para comprobar contra una fuente de valores alpha entrantes, y **alphaTestFunction** es uno de **ALWAYS**, **NEVER**, **EQUAL**, **NOT_EQUAL**, **LESS**, **LESS_OR_EQUAL**, **GREATER**, o **GREATER_OR_EQUAL**, lo que denota el tipo de prueba alpha activa. Crea un objeto componente que define los estados de renderizado para comparaciones del buffer de produndidad y pruebas alpha.

Métodos de **RenderingAttributes**

void setDepthBufferEnable(boolean state)

activa y desactiva la prueba del buffer de profundidad.

void setDepthBufferWriteEnable(boolean state)

activa y desactiva la escritura en el buffer de seguridad.

void setAlphaTestValue(float value)

especifica el valor a usar en la prueba contra valores alpha entranres.

void setAlphaTestFunction(int function)

donde **function** es uno de: **ALWAYS**, **NEVER**, **EQUAL**, **NOT_EQUAL**, **LESS**, **LESS_OR_EQUAL**, **GREATER**, o **GREATER_OR_EQUAL**, que denota el tipo de prueba alpha a realizar. Si la función es **ALWAYS** (por defecto), entonces la prueba alpha está efectivamente desactivada.

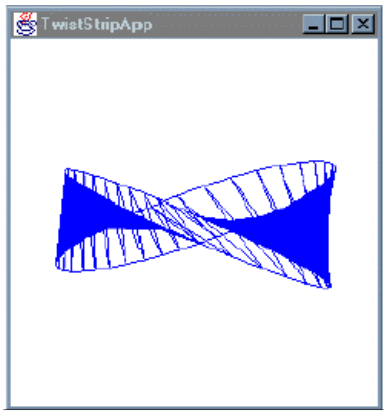
Atributos de Apariencia por Defecto

El constructor de **Appearance** por defecto inicializa un objeto **Appearance** con todos los atributos seleccionados a **null**. La siguiente tabla lista los valores por defecto para dichos atributos con referencia **null**.

color	white (1, 1, 1)
texture environment mode	TEXENV_REPLACE
texture environment color	white (1, 1, 1)
depth test enable	true
shade model	SHADE_GOURAUD
polygon mode	POLYGON_FILL
transparency enable	false
transparency mode	FASTEST
cull face	CULL_BACK
point size	1.0
line width	1.0
point antialiasing enable	false
line antialiasing enable	false

- **Ejemplo: Recortar la cara trasera**

Los Polígonos tienen dos caras. Para muchos objetos visuales, sólo se necesita renderizar una de las caras. Para reducir el poder de cálculo necesario para renderizar las superficies polígonales, el renderizador puede recortar las caras innecesarias. El comportamiento de recortado se define mediante el **PolygonAttribute** del componente **Appearance**. La cara frontal de un objeto es la cara cuyos vértices están definidos en orden contrario a las agujas del reloj. TwistStripApp.java crea un objeto visual (un tornado) que rota sobre su eje Y. Mientras el tornado rota, algunas partes parecen desaparecer. Las piezas desaparecidas se notan fácilmente en el Figura 2-22.
Realmente, **TwistStripApp** define dos objetos visuales, con la misma geometría - que un tornado. Uno de los objetos visuales se renderiza como un marco, y el otro como una superficie sólida. Como los dos objetos tienen la misma localización y orientación, el objeto visual marco sólo es visible cuando no se ve el objeto sólido.



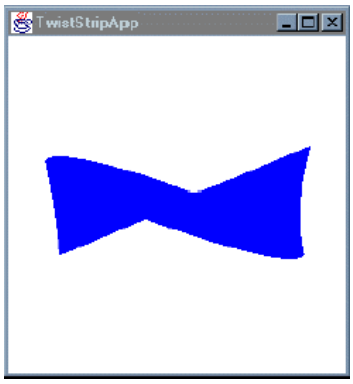
La razón por la que desaparecen los polígonos es que se ha especificado el modelo de recortado, con su valor por defecto **CULL_BACK**. Los tirángulos de la superficie desaparecen cuando su lado trasero (cara trasera) da hacia el plato de imagen. Esta característica permite al sistema de renderizado ignorar las superficies triangulares que no son necesarias, se quiera o no.

Sin embargo, algunas veces el recorte de la cara trasera es un problema, como en el **TwistStripApp**. El problema tiene una solución sencilla: desactivar el recortado. Para hacer esto, creamos un componente **Appearance** que referencie al componente **PolygonAttributes** que desactiva el recortado, como se ve en el fragmento de código 2-10.

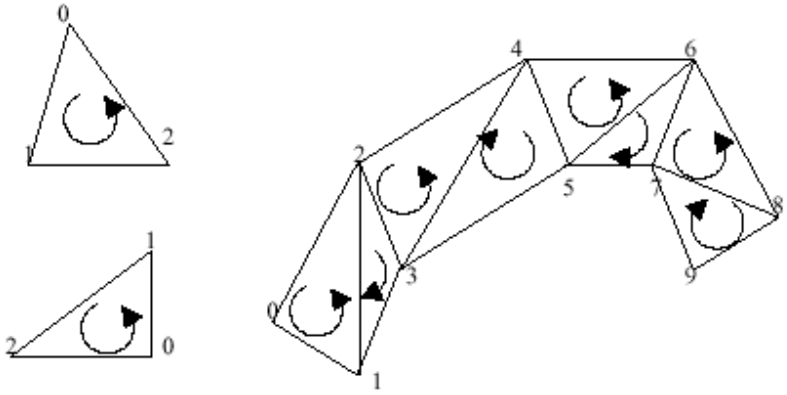
Framento de código 2-10, Desactivar el recortado de la cara trasera para el tornado

```
1.  PolygonAttributes polyAppear = new PolygonAttributes();
2.  polyAppear.setCullFace(PolygonAttributes.CULL_NONE);
3.  Appearance twistAppear = new Appearance();
4.  twistAppear.setPolygonAttributes(polyAppear);
5.  // several lines later, after the twistStrip TriangleStripArray has
6.  // been defined, create a Shape3D object with culling turned off
7.  // in the Appearance bundle, and add the Shape3D to the scene graph
8.  twistBG.addChild(new Shape3D(twistStrip, twistAppear));
```

En la Figura 2-23, la desactivación del recorte de las caras traseras realmente funciona. Ahora se renderizan todos los polígonos, no importa la dirección en la que se muestren.



La cara frontal de un polígono es el lado en el que los vértices aparecen en orden contrario a las agujas del reloj. Está normalmente es referida como la "**Regla de la Mano Derecha**". La regla usada para derterminar la cara frontal de un marco geométrico (es dedir, triángulo, cuadrado) alterna cada elemento del marco. La figura 2-24 muestra ejemplos del uso de la regla de la mano derecha para determinar las caras frontales.



Crear Contenidos Sencillos en Java 3D

Cargadores

Una clase **Loader** lee ficheros de escenas 3D (no ficheros Java 3D) y crea representaciones Java 3D de sus contenidos que pueden ser añadidos selectivamente a un mundo Java 3D y argumentados por otro código Java 3D. El paquete com.sun.j3d.loaders proporciona el contenido principal para convertir los ficheros creados en otras aplicaciones en aplicaciones Java 3D. Las clases cargadoras implementan el interface **Loader** definido en el paquete com.sun.j3d.loaders.

Como hay una gran variedad de formatos de ficheros para porpósitos de representación de escenas 3D (por ejemplo, .obj, .vrml, etc.) y siempre habrá más formatos de ficheros, el código real para cargar un fichero no forma parte de Java 3D o del paquete loaders; sólo se incluye el interface para el mecanismo de carga. Con la definición del interface, el usuario de Java 3D puede desarrollar clases cargadoras de ficheros con el mismo interface que las otras clases cargadoras.

Ejemplo de Usos de un Loader

Sin una clase que realmente lea el fichero, no es posible cargar su contenido. Con una clase **Loader** es sencillo. La siguiente lista presenta la receta para usar un cargador.

1. encontrar un cargador (si no hay ninguno disponible, los escribimos)
2. importar la clase cargador para nuestro formato de fichero
3. importar otras clases necesarias
4. declarar una variable Scene (no usar el constructor)
5. crear un objeto loader
6. cargar el fichero en un bloque **try**, asignar el resultado a la variable **Scene**
7. insertar el Scene dentro del escenariog gráfico

Con el JDK 1.2 se distribuye una clase basada en este ejemplo, se encuentra en jdk1.2/demo/java3d/ObjLoad. El Fragmento de Código 3-1 presenta un extracto del código de esta demo.
La clase **ObjectFile** se distribuye con el paquete com.sun.j3d.loaders como ejemplo. La Tabla 3-1 muestra algunos otros ejemplos de cargadores disponibles.

Clase ObjectFile

Paquete:

com.sun.j3d.loaders

Implementa:

Loader

La clase **ObjectFile** implementa el interface **Loader** para el formato de fichero de **Wavefront** ".obj", un formato de ficheros de objetos 3D estándard creado por el uso de **Wavefront's Advanced Visualizer _**. Los ficheros **Object** están basados en texto soportanto tanto geometría poligonal como de forma libre (curvas y superficies). El cargador de ficheros .obj de Java 3D soporta un subconjunto de formatos de ficheros, pero es completamente suficiente para cargar casi todos los ficheros **Object** disponibles. La geometría de forma libre no está soportada.

Fragmento de Código 3-1, un extracto de jdk1.2/demo/java3d/ObjLoad/ObjLoad.java

```
1.  import com.sun.j3d.loaders.objectfile.ObjectFile;
2.  import com.sun.j3d.loaders.ParsingErrorException;
3.  import com.sun.j3d.loaders.IncorrectFormatException;
4.  import com.sun.j3d.loaders.Scene;
5.  import java.applet.Applet;
6.  import javax.media.j3d.*;
7.  import javax.vecmath.*;
8.  import java.io.*;
9.
10. public class ObjLoad extends Applet {
11.
```

```
12.  private String filename = null;
13.
14.  public BranchGroup createSceneGraph() {
15.  // Create the root of the branch graph
16.  BranchGroup objRoot = new BranchGroup();
17.
18.  ObjectFile f = new ObjectFile();
19.  Scene s = null;
20.  try {
21.      s = f.load(filename);
22.  }
23.  catch (FileNotFoundException e) {
24.      System.err.println(e);
25.      System.exit(1);
26.  }
27.  catch (ParsingErrorException e) {
28.      System.err.println(e);
29.      System.exit(1);
30.  }
31.  catch (IncorrectFormatException e) {
32.      System.err.println(e);
33.      System.exit(1);
34.  }
35.
36.   objRoot.addChild(s.getSceneGroup());
37.  }
```

Este programa trata sobre añadir comportamientos (el efecto por defecto, o la interacción con el ratón - cubierto en el Capítulo 4) y luces (Capítulo 6) para proporcionar una renderización sombreada del modelo del objeto. Por supuesto, podemos hacer muchas otras cosas con el modelo en un programa Java 3D como añadir animaciones, añadir otras geométrias, cambiar el color del modelo, etc. Dentro de la distribución del JDK 1.2 tenemos un ejemplo de **loader** en jdk1.2/demos/java3d/lightwave/Viewer.java. Este cargador carga las luces y animaciones espacificadas en un fichero .lws de Lightwave.

Cargadores Disponibles Públicamente

En Java 3D existen varias clases cargadoras. La siguiente tabla lista los formatos de ficheros cuyos cargadores están disponibles públicamente. En el momento de escribir esto, al menos hay disponible una clase cargador por cada uno de estos formatos de fichero:

Formato de Fichero	Descripción
3DS	3D-Studio
COB	Caligari trueSpace
DEM	Digital Elevation Map
DXF	AutoCAD Drawing Interchange File
IOB	Imagine
LWS	Lightwave Scene Format
NFF	WorldToolKit NFF format
OBJ	Wavefront
PDB	Protein Data Bank
PLAY	PLAY
SLD	Solid Works (prt and asm files)
VRT	Superscape VRT
VTK	Visual Toolkit
WRL	Virtual Reality Modeling Language

Tabla 3-1, Cargadores Java 3D disponibles Públicamente

Puedes localizar estos cargadores desde la página principal de Java 3D: http://java.sun.con/products/java-media/3d.

Interfaces y Clases Base del Paquete Loader

Esta gran varidad de cargadores existe para hacer más sencilla la escritura de cargadores para los diseñadores Java 3D. Las clases **Loader** son implementaciones del interface **Loader** que baja el nivel de dificultad para escribir un cargador. Como en el ejemplo, un programa que carga un fichero 3D realmente usa un cargador y un objeto escena. El cargador lee, analiza y crea la representación Java 3D de los contenidos del fichero. El objeto escena almacena

el escenario grafico creado por el cargador. Es posible cargar escenas desde más de un fichero (del mismo formato) usando el mismo objeto cargador y crear múltiples objetos escena. Los ficheros de diferentes formatos pueden combinarse en un programa Java 3D usando las clases cargadoras apropiadas.
El siguiente bloque de referencia lista los interface del paquete com.sun.j3d.loaders. Un **loader** implementa el interface **loader** y usa una clase que implementa el interface **scene**.

Sumario de Interfaces de **com.sun.j3d.loaders**

- **Loader**. El interface **Loader** se usa para especificar la localización y los elementos de un formato de fichero a cargar.
- **Scene**. El interface **Scene** es un conjunto de métodos usado para extraer información de escenario gráfico Java 3D de una utilidad cargador de ficheros.

Además de estos interfaces, el paquete com.sun.j3d.loaders proporciona implementaciones básicas de los interfaces.

Sumario de Clases de **com.sun.j3d.loaders**

- **LoaderBase**. Esta clase implementa el interface **Loader** y añade constructores. Esta clase es extendida por los autores para especificar clases cargadoras.
- **SceneBase**. Esta clase implementa el interface **Scene** y añade métodos usados por los cargadores. Esta clase también es usada por los programas que usan clases cargadoras.

Sumario de Métodos del Interface **Loader**

Paquete:

com.sun.j3d.loaders

El interface **Loader** se usa para especificar la localización y los elementos de un formato de fichero a cargar. Este interface se utiliza para darle a los cargadores de varios formatos de ficheros un interface público común. Idealmente el interface **Scene** será implementado para darle al usuario un interface consistente para

extraer los datos.

Scene load(java.io.Reader reader)

Este método carga el **Reader** y devuelve el objeto **Scene** que contiene la escena.

Scene load(java.lang.String fileName)

Este método carga el fichero nombrado y devuelve el objeto **Scene** que contiene la escena.

Scene load(java.net.URL url)

Este método carga el fichero nombrado y devuelve el objeto **Scene** que contiene la escena.

void setBasePath(java.lang.String pathName)

Este método selecciona el nombre del path base para los ficheros de datos asociados con el fichero pasado en el método load(String).

void setBaseUrl(java.net.URL url)

Este método selecciona el nombre de la URL base para los ficheros de datos asociados con el fichero pasado en el método load(String).

void setFlags(int flags)

Este método selecciona las banderas de carga para el fichero.

- **LOAD_ALL**. Esta bandera activa la carga de todos los objetos en la escena.
- **LOAD_BACKGROUND_NODES**. Esta bandera activa la carga de los objetos del fondo en la escena.
- **LOAD_BEHAVIOR_NODES**. Esta bandera activa la carga de comportamientos en la escena.
- **LOAD_FOG_NODES**. Esta bandera activa la carga de objetos niebla en la escena.
- **LOAD_LIGHT_NODES**. Esta bandera activa la carga de objetos luces en la escena.
- **LOAD_SOUND_NODES**. Esta bandera activa la carga de objetos de sonido en la escena.
- **LOAD_VIEW_GROUPS**. Esta bandera activa la carga de objetos vista (cámara) en la escena.

La clase **LoaderBase** proporciona una implementación para cada uno de los tres métodos load() del interface **Loader**. **LoaderBase** también implementa dos constructores. Observa que los tres métodos cargadores devuelven un objeto **Scene**.

Sumario de Constructores de la Clase **LoaderBase**

Paquete:

com.sun.j3d.loaders

Implementa:

Loader

Esta clase implementa el interface **Loader**. El autor de un cargador de ficheros debería extender esta clase. El usuario de un cargador de ficheros debería usar estos métodos.

LoaderBase()
Construye un **Loader** con los valores por defecto para todas las variables.
LoaderBase(int flags)
Construye un **loader** con las banderas especificadas.

Lista Parcial (métodos usados por usuarios) de la Clase **SceneBase**

Background[] getBackgroundNodes()
Behavior[] getBehaviorNodes()
java.lang.String getDescription()
Fog[] getFogNodes()
float[] getHorizontalFOVs()
Light[] getLightNodes()
java.util.Hashtable getNamedObjects()
BranchGroup getSceneGroup()
Sound[] getSoundNodes()
TransformGroup[] getViewGroups()

- **Escribir un Loader**

Como se mencionó arriba, la característica más importante de los cargadores es que podemos escribir el nuestro propio, lo que significa que todos los usuarios de Java 3D también pueden hacerlo!

Para escribir un cargador, debemos extender la clase **LoaderBase** definida en el paquete com.sun.j3d.loaders. El nuevo cargador usará la clase **Scene** del mismo paquete.

Los futuros cargadores deberían tener poca necesitad de subclasificar **SceneBase**, o de implementar directamente **Scene**, ya que la funcionalidad de **SceneBase** es bastante correcta. Esta clase es responsable del almacenamiento y recuperación de datos creados por un cargador mientras lee un fichero. Los métodos de almacenamiento (usados sólo por los autores del **Loader**) son todas las rutinas **add***. Los métodos recuperadores (usados principalmente por los usuarios de **Loader**) son todas las rutinas **get***.

Escribir un cargador de ficheros puede ser bastante complejo dependiendo de la complejidad del formato del fichero. La parte más dura es analizar el fichero. Por supuesto, tenemos que empezar con la documentación del formato de fichero para el que queremos escribir la clase **Loader**. Una vez que se entiende el formato, empezamos leyendo las clase bases de **loader** y **scene**. La nueva clase **loader** extenderá la clase base **loader** y usará la clase base **scene**.

En la extensión del clase **loader** base, la mayoría del trabajo será escribir métodos que reconozcan los distintos tipos de contenidos que se pueden representar en el formato del fichero. Cada uno de esos métodos crea el correspondiente componente Java 3D del escenario gráfico y lo almacena en un objeto **scene**.

Sumario de Constructores de la Clase **SceneBase**

Paquete: com.sun.j3d.loaders

Implementa: Scene

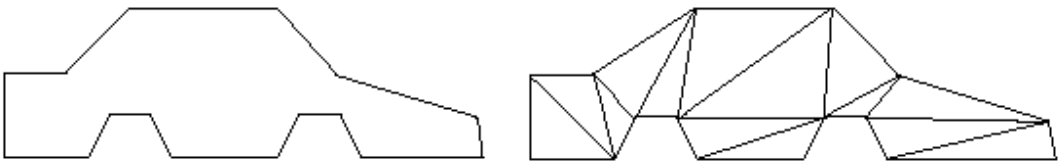
Esta clase implementa el interface **Scene** y lo amplía para incorporar utilidades que podrían usars los cargadores. Esta clase es responsable del almacenamiento

y recuperación de los datos de la escena.

SceneBase()
Crea un objeto **SceneBase** - no debería haber ninguna razón para usar este constructor excepto en la implementación de una nueva clase de **loader**.

GeometryInfo

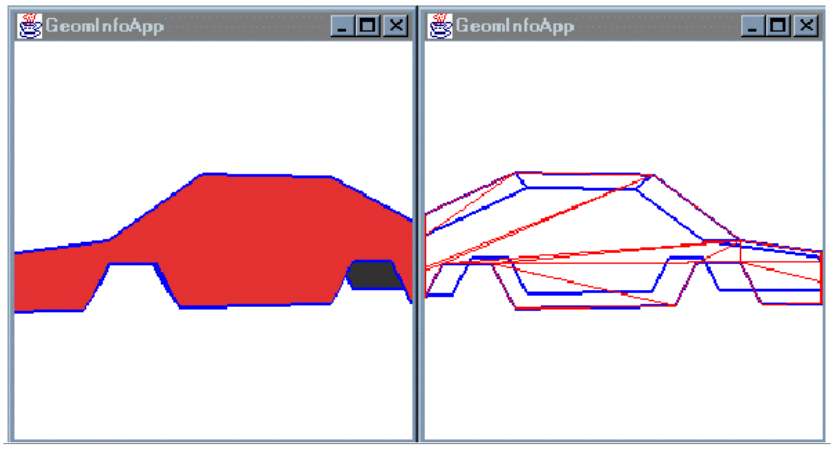
Si no tenemos acceso a los ficheros de modelos geométricos o a software de modelado geométrico, tenemos que crear nuestra geometría a mano. Como se mencionó en capítulos anteriores, esta codificación de geometría a mano requiere mucho tiempo y es una actividad muy propensa a errores. Como sabemos, cuando especifamos geometrías a través de las clases corazón, estamos limitados a triángulos y cuadrados. Usando las clase de utilidad **GeometryInfo** se puede mejorar el tiempo empleado y el trabajo tedioso de la creacción de geometrías. En lugar de especificar cada triángulo, podemos especificar polígonos arbitrarios, que pueden ser cóncavos, polígonos no planos, e incluso con agujeros. El objeto **GeometryInfo**, y otras clases de utilidad, convierten esta geometría en geometría tirangular que Java 3D puede renderizar.
Por ejemplo, si queremos crear un coche en Java 3D, en vez de especificar triángulos, podemos espcificar el perfil del coche como un polígono en un objeto **GeometryInfo**. Luego, usando un objeto **Triangulator**, el polígono puede subdividirse en triángulos. La imagen de la izquierda de la figura 3-2 muestra el perfil de un coche como un polígono. El imagen de la derecha es el polígonos subdividido en triángulos.



Si estamos interesados en el rendimiento, ¿y quién no?, usamos un objeto **Stripifier** para convertir los triángulos en franjas de triángulos. Si queremos

sombrear el objeto visual, usamos el **NormalGenerator** para calcular las superficies y la geometría.

El programa GeomInfoApp.java, usa las clases **GeometryInfo**, **Triangulator**, **Stripifier**, y **NormalGeneration** para crear un coche. La Figura 3-3 muestra dos renderizaciones podrucidas por **GeomInfoApp.java**. En ambas, las líneas azules muestran los contornos especificados en el objeto **GeometryInfo**. Los triángulos rojos (rellenos y sombreados a la ziquierda y enmarcados a la derecha) fueron calculados automáticamente por el objeto **GeometryInfo** con **Triangulation**, **NormalGeneration**, y **Stripification**.



Un sencillo polígono plano, similar al de la figura 3-2, especifica el perfil de un coche (cada lado) en el ejemplo **GeomInfoApp**.

- **Sencillo Ejemplo de GeometryInfo**

Usar un objeto **GeometryInfo** es tan sencillo como usar las clases corazón **GeomertryArray** si no más sencillo. En la creacción de un objeto **GeomertyInfo**, simplemente especificamos el tipo de geometría que vamos a necesitar. Las opciones son **POLYGON_ARRAY**, **QUAD_ARRAY**, **TRIANGLE_ARRAY**, **TRIANGLE_FAN_ARRAY**, y **TRIANGLE_STRIP_ARRAY**. Entonces seleccionamos las coordenadas y el contador de franjas. No tenemos que decirle al objeto **GeometryInfo** cuántas coordenadas hay en los datos; se calcularán automáticamente.

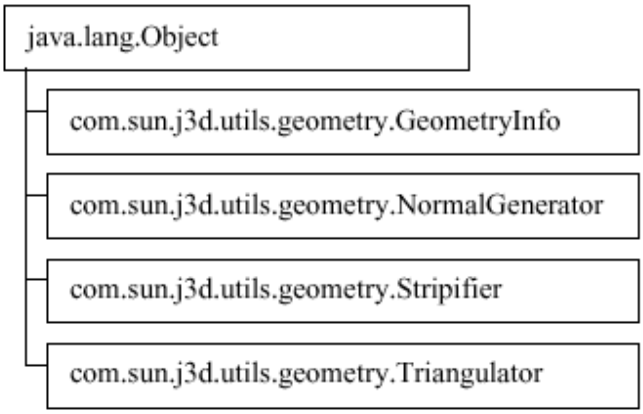
El Fragmento de Código 3-2 muestra un ejemplo de aplicación **GeometryInfo**. Las líneas 1-3 muestran la creacción de un objeto **GeometryInfo** y la especificación de la geometría inicial.
Después de haber creado el objeto **GeometryInfo**, podrían usarse las otras clases. Si queremos usar el **NormalGenerator**, por ejemplo, primero creamos un objeto **NormalGenerator**, luego le pasamos el objeto **GeometryInfo**. Las líneas 8 y 9 hacen exactamente esto.

Fragmento de Código 3-2, Usar un GeometryInfo, y las utilidades Triangulator, NormalGenerator, y Stripifier

```
1.  GeometryInfo gi = new GeometryInfo(GeometryInfo.POLYGON_ARRAY);
2.  gi.setCoordinates(coordinateData);
3.  gi.setStripCounts(stripCounts);
4.
5.  Triangulator tr = new Triangulator();
6.  tr.triangulate(gi);
7.
8.  NormalGenerator ng = new NormalGenerator();
9.  ng.generateNormals(gi);
10.
11. Stripifier st = new Stripifier();
12. st.stripify(gi);
13.
14. Shape3D part = new Shape3D();
15. part.setAppearance(appearance);
16. part.setGeometry(gi.getGeometryArray());
```

Clases para GeometryInfo

La clase **GeometryInfo** y sus clases relacionadas son miembros del paquete com.sun.j3d.util.geometry y son subclases de **Object**. La Figura 3-4 muestra el árbol de estas clases.



La clase **GeometryInfo** sólo tiene un constructor y en él especificamos el tipo de geometría a especificar por las coordenadas.

Sumario de Constrcutores de la Clase **GeometryInfo**

Paquete: com.sun.j3d.utils.geometry

Extiende: java.lang.Object

El objeto **GeometryInfo** es donde ponemos nuestra geometría si queremos usar las librerías de utilidades de Java 3D. Una vez que tenemos los datos en el objeto **GeometryInfo**, podemos enviarlo a cualquiera (o a todas) las clases de utilidades para realizar operaciones sobre ellas, como generar superficies o convertirlo en franjas largas para una renderización más eficiente. La geometría se carga tal como es en el objeto **GeometryArray** de Java 3D, pero hay unas pocas opciones para obtener datos del objeto. La propia **GeometryInfo** contiene algunas sencillas utilidades, como el cálculo de índices para datos no indexados y obtener datos no usados en nuestra información de geometría indexada ("compactación").

GeometryInfo(int primitive)

Construye y un objeto **GeometryInfo**, donde "primitive" es uno de

- **POLYGON_ARRAY** posiblemente multi-contorno, posiblemente polígonos no planos
- **QUAD_ARRAY** cada conjunto de cuatro vértices forma un cuadrado independiente
- **TRIANGLE_ARRAY** cada conjunto de tres vértices forma un triángulo

independiente

- **TRIANGLE_FAN_ARRAY** el array **stripCounts** indica cuántos vértices usar para cada pala de triángulo
- **TRIANGLE_STRIP_ARRAY** el array **stripCounts** cuantos vértices se usarán por cada franja de triángulo

La clase **GeometryInfo** tiene muchos métodos. Muchos de ellos son para seleccionar (u obtener) datos de coordenadas, colores, índices, superficies o de coordenadas de textura. La mayoría de las aplicaciones sólo usarán unos pocos de estos métodos. Sin embargo, es conveniente poder especificar la geometría a cualquier nivel de detalle y deja el resto calculado.

Lista Parcial de Métodos de la Clase **GeometryInfo**

void recomputeIndices()

Reconstruye los índices para garantizar la información de conexión.

void reverse()

Invierte el orden de todas las listas.

void setColorIndices(int[] colorIndices)

Selecciona el array de índices en un array Color.

void setColors(Color3f[] colors)

Selecciona el array de colores.

void setColors(Color4f[] colors)

Selecciona el array de colores, Hay otros métodos setColors.

void setContourCounts(int[] contourCounts)

Selecciona la lista del contador de contornos.

void setCoordinateIndices(int[] coordinateIndices)

Selecciona el array de índices en el array de coordenadas.

void setCoordinates(Point3f[] coordinates)

Selecciona el array de coordenadas.

void setCoordinates(Point3d[] coordinates)

Selecciona el array de coordenadas. Hay otros métodos setCoordinates.

void setNormalIndices(int[] normalIndices)

Selecciona el array de índices en el array de superficies normales.

void setNormals(Vector3f[] normals)

Selecciona el array de superficies normales.

void setNormals(float[] normals)

Selecciona el array de superficies normales.

void setStripCounts(int[] stripCounts)

Selecciona el array del contador de franjas.

void setTextureCoordinateIndices(int[] texCoordIndices)

Selecciona el array de índices en el array de coordenadas de textura.

void setTextureCoordinates(Point2f[] texCoords)

Selecciona el array de coordenadas de textura. Hay otros métodos setTextureCoordinates.

Todas las clases de 'ayuda' de **GeometryInfo** se usan de forma similar. Los siguientes bloques de referencia muestran los constructoes y métodos para **Triangulator**, **Stripifier**, y **NormalGenerator**, en este orden, que es el orden en que se usarían para un **POLYGON_ARRAY**.

La utilidad **Triangulator** sólo se usa con geometría **POLYGON_ARRAY**. Otros objetos **GeometryInfo** con otras geometrías primitivas sólo usarían **Stripifier** y **NormalGenerator**.

El constructor por defecto para la clase **Triangulator** simplemente crea un objeto **Triangulation**.

Sumario de Constructores de la Clase **Triangulator**

Paquete: com.sun.j3d.utils.geometry

Extiende: java.lang.Object

Triangulator es una utilidad para convertir polígonos arbitraríos en triángulos para que puedan ser renderizados por Java 3D. Los polígonos pueden ser cóncavos, no planos, y pueden contener agujeros.

Triangulator()

Creata un uevo ejemplar de **Triangulator**.

El único método de la clase **Triangulator** es para triangular un objeto **GeometryInfo**.

Sumario de Métodos de la Clase **Triangulator**

void triangulate(GeometryInfo gi)
Esta rutina convierte el objeto **GeometryInfo** desde el tipo primitivo **POLYGON_ARRAY** al tipo primitivo **TRIANGLE_ARRAY** usando técnicas de descomposición de polígonos.

El único constructor de la clase **Stripifier** crea un objeto **stripification**.

Sumario de Constructores de la Clase **Stripifier**

Paquete: com.sun.j3d.utils.geometry

Extiende: java.lang.Object

La utilidad **Stripifier** cambia el tipo primitivo del objeto **GeometryInfo** a una franja de triángulos. Las franjas se hacen analizando los triángulos en los datos originales y conectándolos juntos.

Para obtener un mejor resultado se debe realizar antes un **NormalGeneration** sobre el objeto **GeometryInfo**.

Stripifier()
Crea el objeto Stripifier.

El único método de la clase **Stripifier** es para convertir la geometría de un objeto **GeometryInfo**.

Sumario de Métodos de la Clase **Stripifier**

void stripify(GeometryInfo gi)
Cambia la geometría contenida en el objeto **GeometryInfo** en un array de franjas de triángulos.

La clase **NormalGenerator** tiene dos constructores. El primero construye un **NormalGenerator** con un valor por defecto para el ángulo de pliegue. El segundo constructor permite la especificación del ángulo de pliegue.

Sumario de Constructores de la Clase **NormalGenerator**

Paquete: com.sun.j3d.utils.geometry

Extiende: java.lang.Object

La utilidad **NormalGenerator** calcula y rellena en las superficies de un objeto **GeometryInfo**. Las superficies normales se estiman basándose en el análisis de la información de coordenadas indexadas. Si nuestros datos no están indexados, se creara una lista de índices.

Si dos (o más) triángulos del modelo comparten el mismo índice de coordenadas el **normalgenerator** intentará generar una superficie para el vértice, resultando en una superficie pulida. Si dos coordenadas no tienen el mismo índice entonces tendrán dos superficies separadas, incluso si tienen la misma posición. Esto resultará en un "pliegue" en nuestro objeto. Si sospechamos que nuestros datos no están indexados apropiadamente, debemos llamar a GeometryInfo.recomputeIndexes().

Por supuesto, algunas veces, nuestro modelo tiene un pliegue. Si dos superficies triangulares difieren por más de **creaseAngle**, entonces el vértice obtendrá dos superficies separadas, creando un pliegue discontinúo en el modelo. Esto es perfecto para el borde de un tabla o la esquina de un cubo, por ejemplo.

NormalGenerator()

Construye un **NormalGenerator** con el ángulo de pliegue por defecto (0.76794 radianes, o 44°).

NormalGenerator(double radians)

Construye un **NormalGenerator** con el ángulo de pliegue especificado en radianes.

Entre los métodos de la clase **NormalGenerator** se incluyen algunos para seleccionar u obtener el ángulo de pliegue, y cálculo de superficies para la geometría de un objeto **GeometryInfo**.

Sumario de Métodos de la Clase **NormalGenerator**

void generateNormals(GeometryInfo geom)
Genera superficies para el objeto GeometryInfo.
double getCreaseAngle()
Devuelve el valor actual para el ángulo de pliegue, en radianes.
void setCreaseAngle(double radians)
Selecciona el ángulo de pliegue en radianes.

Texto 2D

Hay dos formas de añadir texto a una escena Java 3D. Una forma es usar la clase **Text2D** y otra es usar la clase **Text3D**. Obviamente, la diferencia es que los objetos **Text2D** tienen dos dimensiones y los objetos **Text3D** tienen tres dimensiones. Otra diferencia significante es la forma en que se crean estos objetos.
Los objetos **Text2D** son polígonos rectangulares con el texto aplicado como una textura. Los objetos **Text3D** son objetos 3D geométricos creados como un extrusión del texto.
Como una subclase de **Shape3D**, los ejemplares de **Text2D** pueden ser hijos de objetos **group**. Para situar un objeto **Text2D** en una escena Java 3D, simplemente creamos el objeto **Text2D** y lo añadimos al escenario gráfico. Aquí tenemos una sencilla receta.

1. Crer un objeto **Text2D**
2. Añadirlo al escenario gráfico

Los objetos **Text2D** se implementan usando un polígono y una textura. El polígono es transparente para que sólo sea visible la textura. La textura es la cadena de texto seleccionada con los parámetros de fuente y tipo especifiados. Los tipos de

letras disponibles dependen de nuestro sistema. Normalmente, están disponibles Courier, Helvetica, TimesRoman, entre otros. Cualquier fuente disponible en el AWT también está disponible para aplicaciones **Text2D** (y **Text3D**).

Ejemplo de Text2D

El Fragmento de Código 3-3 muestra un ejemplo de adición de texto 2D a una escena. El objeto **Text2D** se crea en las líneas 21 a 23. En este constructor, se especifican la cadena de texto, el color, el tipo, el tamaño y el estilo de la fuente. El objeto **Text2D** se añade a la escena en la línea 24. Observa la sentencia import para Font (línea 5) usada para las constantes de estilos de fuente.

Fragmento de Código 3-3, un objeto Text2D

```
1.  import java.applet.Applet;
2.  import java.awt.BorderLayout;
3.  import java.awt.Frame;
4.  import java.awt.event.*;
5.  import java.awt.Font;
6.  import com.sun.j3d.utils.applet.MainFrame;
7.  import com.sun.j3d.utils.geometry.Text2D;
8.  import com.sun.j3d.utils.universe.*;
9.  import javax.media.j3d.*;
10. import javax.vecmath.*;
11.
12. // Text2DApp renders a single Text2D object.
13.
14. public class Text2DApp extends Applet {
15.
16. public BranchGroup createSceneGraph() {
17. // Create the root of the branch graph
18. BranchGroup objRoot = new BranchGroup();
19.
20. // Create a Text2D leaf node, add it to the scene graph.
21. Text2D text2D = new Text2D("2D text is a textured polygon",
22. new Color3f(0.9f, 1.0f, 1.0f),
23. "Helvetica", 18, Font.ITALIC));
24. objRoot.addChild(text2D);
```

Text2DApp.java es un programa completo que incluye el fragmento de código anterior. En este ejemplo, el objeto **Text2D** rota sobre el origen de la escena. Cuando se ejecuta la aplicación podemos ver, por defecto, que el polígono texturado no es visible cuando se ve desde atrás.

Algunos atributos de un objeto **Text2D** se pueden modificar variando le paquete de apariencia referenciado y/o el **NodeComponent**. El Fragmento de Código 3-4 muestra el código que modifica el objeto **text2d**, creado en el Fragmento de Código 3-3.

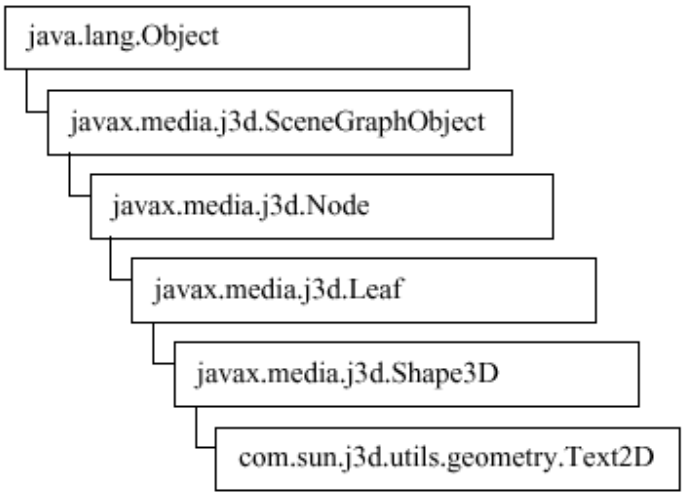
Fragmento de Código 3-4, Haciendo visibles los dos lados de un objeto Text2D

```
25.  Appearance textAppear = text2d.getAppearance();
26.
27.  // The following 4 lines of code make the Text2D object 2-sided.
28.  PolygonAttributes polyAttrib = new PolygonAttributes();
29.  polyAttrib.setCullFace(PolygonAttributes.CULL_NONE);
30.  polyAttrib.setBackFaceNormalFlip(true);
31.  textAppear.setPolygonAttributes(polyAttrib);
```

La textura creada por un objeto **Text2D** también puede aplicarse a otros objetos visuales. Ya que la aplicación de texturas a objetos visuales es el objetivo del Capítulo 7 lo dejaremos aquí.

- **Clases Usadas para Crear Objetos Text2D**

La única clase necesaria es la clase **Text2D**. Como podemos ver de la Figura 3-7, **Text2D** es una clase de utilidad que desciende de **Shape3D**.



Sumario de Constructores de la Clase **Text2D**

Paquete: com.sun.j3d.utils.geometry

Esta clase crea un rectángulo de textura mapeado que muestra la cadena de texto enviada por el usuario, dándole la apariencia suministrada en los parámetros de usuario. El tamaño del rectángulo (y su mapa de textura) está determinado por los parámetros de la fuente pasados al constructor. El objeto **Shape3D** resultante es un rectángulo transparente (excepto el texto) localizado en (0, 0, 0).

Text2D(java.lang.String text, Color3f color, java.lang.String fontName,
int fontSize, int fontStyle)

Constructor.

Con el constructor **Text2D**, hay un método. Este método selecciona el factor de escala para crear objetos **Text2D** mayores o menores que el tamaño de punto especificado. Este método no es útil en la versión 1.1.x del API, ya que sólo se utiliza cuando se especifica el texto. En la versión 1.2 se ha introducido un método setText() haciendo útil el setRectangleScaleFactor().

Sumario de Métodos de la Clase **Text2D**

void setRectangleScaleFactor(float newScaleFactor)

Selecciona el factor de escala usado para convertir la anchura/altura de la imagen.

Texto 3D

Otra forma de añadir texto a un mundo virtual Java 3D es crear un objeto **Text3D** para texto. Mientras que **Text2D** crea el texto con un textura, **Text3D** crea texto usando geometría. La geometría textual de un objeto **Text3D** es una extrusión de la fuente.
Crear un objeto **Text3D** es un poco más complicado que crear un objeto **Text2D**. El primer paso es crear un objeto **Font3D** con el tipo de fuente, el tamaño y el estilo seleccionado. Luego se crea un objeto **Text3D** para una cadena particular usando el objeto **Font3D**. Como la clase **Text3D** es una subclase de **Geometry**, el objeto **Text3D** es un **NodeComponent** que es referenciado por uno o más objetos **Shape3D**:

1. Crear un objeto **Font3D** desde una fuente AWT
2. Crear un objeto **Text3D** para un string usando el objeto **Font3D**, opcionalmente especificando un punto de referencia
3. Referenciar el objeto desde un objeto **Shape3D** añadido al escenario gráfico

Ejemplo de Text3D

El Fragmento de Código 3-5 muestra la construcción básica de un objeto **Text3D**. El objeto **Font3D** se crea en las líneas 19 y 20. El tipo usado es "Helvetica". Igual que en **Text2D**, cualquier tipo disponible en el AWT puede ser usado para **Font3D** y por lo tanto en el objeto **Text3D**. Este constructor de **Font3D** (líneas 19 y 20) también selecciona el tamaño de la fuente a 10 puntos y usa la extrusión por defecto.
La sentencia de las líneas 21 y 22 crea un objeto **Text3D** usando el objeto **Font3D** recientemente creado para la cadena "3DText" mientras especifica un punto de referencia para el objeto. Las últimas dos sentencias crean un objeto **Shape3D** para el objeto **Text3D** y lo añaden al escenario gráfico. Observa que la sentencia import de la línea 5 es necesaria porque se usa un objeto **Font** para la creacción de **Font3D**.

Fragmento de Código 3-5, Crear un objeto Visual Text3D

```
1.  import java.applet.Applet;
```

```
2.  import java.awt.BorderLayout;
3.  import java.awt.Frame;
4.  import java.awt.event.*;
5.  import java.awt.Font;
6.  import com.sun.j3d.utils.applet.MainFrame;
7.  import com.sun.j3d.utils.universe.*;
8.  import javax.media.j3d.*;
9.  import javax.vecmath.*;
10.
11. // Text3DApp renders a single Text3D object.
12.
13. public class Text3DApp extends Applet {
14.
15. public BranchGroup createSceneGraph() {
16. // Create the root of the branch graph
17. BranchGroup objRoot = new BranchGroup();
18.
19. Font3D font3d = new Font3D(new Font("Helvetica", Font.PLAIN, 10),
20. new FontExtrusion());
21. Text3D textGeom = new Text3D(font3d, new String("3DText"),
22. new Point3f(-2.0f, 0.0f, 0.0f));
23. Shape3D textShape = new Shape3D(textGeom);
24. objRoot.addChild(textShape);
```

La Figura 3-9 muestra un objeto **Text3D** que ilustra la extrusión del tipo. En la figura, la extrusión se muestra en gris mientras que el tipo se muestra en negro. Para recrear esta figura en Java 3D, son necesarios un objeto **Material** y otro **DirectionalLight**. No podemos seleccionar el color de los vértices individuales en el objeto **Text3D** porque no tenemos acceso a la geometría del objeto **Text3D**.

El texto de un objeto **Text3D** puede orientarse de una gran cantidad de formas. La orientación se especifica como el camino de dirección. Las direcciones son **right**, **left**, **up**, y **down**.

Cada objeto **Text3D** tiene un punto de referencia. El punto de referencia para un objeto **Text3D** es el origen del objeto. El punto de referencia de cada objeto se define por la combinación del camino y la alineación del texto. La Tabla 3-2 muestra los efectos de las especificaciones del camino y la alineación sobre la orientación del texto y la situación del punto de referencia.

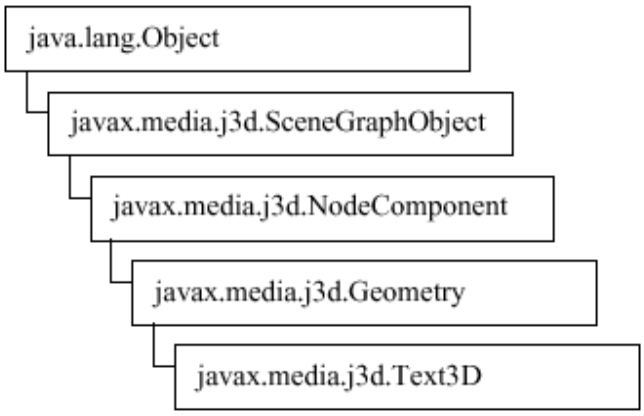
La situación del punto de referencia puede definirse explícitamente sobreescribiendo el camino y la alineación.

	ALIGN_FIRST (default)	ALIGN_CENTER	ALIGN_LAST
PATH_RIGHT (default)	Text3D	Text3D	Text3D
PATH_LEFT	D3txeT	D3txeT	D3txeT
PATH_DOWN	T e x t	T e x t	T e x t
PATH_UP	t x e T	t x e T	t x e T

Los objetos **Text3D** tienen superficies. La adicción de un paquete de apariencia que incluye un objeto **Material** a un objeto **Shape3D** referenciando la geometría **Text3D** permitirá la iluminación del objeto **Text3D**.

Clases Usadas en la Creación de Objetos Text3D

Esta sección presenta el material de referencia para tres clases usadas en la creacción de objetos **Text3D**: **Text3D**, **Font3D**, y **FontExtrusion**, en este orden. La Figura 3-10 muestra el árbol de clases de **Text3D**.



La clase **Text3D** define varios constructores. Cada uno permite especificar ninguno, uno o todos los atributos de un objeto **Text3D**.

Sumario de Constructores de la Clase **Text3D**

Un objeto **Text3D** es una cadena de texto que se ha convertido en una geometría 3D. El objeto **Font3D** determina la apariencia del objeto **NodeComponent**. Cada objeto **Text3D** tiene una posición - un punto de referencia que sitúa el objeto **Text3D**. El texto 3D puede situarse alrededor de su posición usando diferentes alineamientos y caminos.

Text3D()

Crea un objeto **Text3D** vacío. Los valores por defecto usados para este y otros constructores son:

- **font 3D null**
- **string null**
- **position (0,0,0)**
- **alignment ALIGN_FIRST**
- **path PATH_RIGHT**
- **character spacing 0.0**

Text3D(Font3D font3D)

Crea un objeto **Text3D** con el objeto **Font3D** dado.

Text3D(Font3D font3D, String string)

Crea un objeto **Text3D** dando un objeto **Font3D** y una cadena de texto.

Text3D(Font3D font3D, String string, Point3f position)

Crea un objeto **Text3D** dando un objeto **Font3D** y una cadena de texto. El punto de posición define un punto de referencia para el objeto **Text3D**. Su posición se define en relación a la esquina inferior iquierda frontal de la geometría.

Text3D(Font3D font3D, String string, Point3f position,
int alignment, int path)

Crea un objeto **Text3D** dando un objeto **Font3D** y una cadena de texto.

- **ALIGN_CENTER** alineamiento: el centro de la cadena se sitúa en el punto de posición.
- **ALIGN_FIRST** alineamiento: el primer caracter de la cadena se sitúa en el punto de posición.
- **ALIGN_LAST** alineamiento: el último caracter de la cadena se sitúa en el punto de posición.
- **PATH_DOWN** camino: las letras sucesivas se situarán debajo de la letra actual.
- **PATH_LEFT** camino: las letras sucesivas se situarán a la izquierda de la letra actual.
- **PATH_RIGHT** camino: las letras sucesivas se situarán a la derecha de la letra actual.
- **PATH_UP** camino: las letras sucesivas se situarán sobre la letra actual.

La clase **Text3D** también define varios métodos. Cada uno de ellos nos permite modificar (seleccionar) los atributos del objeto **Text3D**. Esta clase también define los correspondientes métodos **get***.

Sumario de Métodos de la Clase **Text3D**

void setAlignment(int alignment)

Selecciona la política de alineamiento para este objeto **Text3D NodeComponent**.

void setCharacterSpacing(float characterSpacing)

Selecciona el espaciado entre caracteres cuando se construye la cadena **Text3D**.

void setFont3D(Font3D font3d)

Selecciona el objeto **Font3D** usado para este objeto **Text3D NodeComponent**.

void setPath(int path)

Selecciona la dirección del camino del nodo.

void setPosition(Point3f position)

Selecciona el punto de referencia del nodo.

void setString(java.lang.String string)

Copia la cadena de caracteres desde el parámetro suministrado dentro del nodo **Text3D**.

Sumario de Capacidades de la Clase **Text3D**

- **ALLOW_ALIGNMENT_READ | WRITE** permite leer (escribir) el valor de alineamiento del texto.
- **ALLOW_BOUNDING_BOX_READ** permite leer el valor de la caja que rodea la cadena de texto.
- **ALLOW_CHARACTER_SPACING_READ | WRITE** permite leer (escribir) el valor del espaciado entre caracteres.
- **ALLOW_FONT3D_READ | WRITE** permite leer (escribir) la información del componente **Font3D**.
- **ALLOW_PATH_READ | WRITE** permite leer (escribir) el valor del camino del texto.
- **ALLOW_POSITION_READ | WRITE** permite leer (escribir) el valor de la posición del texto.
- **ALLOW_STRING_READ | WRITE** permite leer (escribir) el objeto **String**.

Cada objeto **Text3D** se crea desde un objeto **Font3D**. Un sólo objeto **Font3D** puede usarse para crear un número ilimitado de objetos **Text3D**. Un objeto **Font3D** contiene la extrusión geométrica de cada caracter en el tipo de letra. Un objeto **Text3D** copia las geometrías para formar la cadena especificada. Los objetos **Font3D** pueden ser recolectados por el recolector de basura sin afectar a los objetos **Text3D** creados a partir de él.

Sumario de Constructores de la Clase **Font3D**

Extiende: java.lang.Object

Una fuente 3D consiste en una fuente Java 2D y un camino de extrusión. Este camino de extrusión describe cómo varía el flanco de una letra en el eje z. El objeto **Font3D** se usa para almacenar letras 2D extrusionadas. Estas letras 3D pueden usarse para construir objetos **Text3D NodeComponent**. Las fuentes 3D personalizadas así como el almacenamiento de fuentes 3D en disco se cubriran en una futura versión de Java 3D.

También puedes ver : java.awt.Font, FontExtrusion, Text3D

Font3D(java.awt.Font font, FontExtrusion extrudePath)

Crea un objeto **Font3D** desde el objeto **Font** especificado.

Sumario de Métodos de **Font3D**

void getBoundingBox(int glyphCode, BoundingBox bounds)

Devuelve la caja 3D que rodea el código de letra específicado.

java.awt.Font getFont()

Devuelve la fuente Java 2D usada para crear este objeto **Font3D**.

void getFontExtrusion(FontExtrusion extrudePath)

Copia el objeto **FontExtrusion** usado para crear este objeto **Font3D** dentro del parámetro especificado.

La clase **Font** se usa en la creacción de un objeto **Font3D**.

Lista Parcial de Métodos de la Clase **Font**

Paquete: java.awt

Una clase **AWT** que crea una representación interna de las fuentas. **Font** desciende de **java.lang.Object**.

public Font(String name, int style, int size)

Crea un nuevo **Font** desde el nombre, estilo y tamaño de punto especificados.

Parámetros:

- **name** - el nombre del tipo de letra. Este puede ser un nombre lógico o un nombre de tipo de fuente. Un nombre lógico puede ser uno de: **Dialog**, **DialogInput**, **Monospaced**, **Serif**, **SansSerif**, o **Symbol**.
- **style** - el estilo para la fuente. El argumento estilo es una máscara de bits de enteros que puede ser **PLAIN**, o una unión de **BOLD** y/o **ITALIC** (por ejemplo, **Font.ITALIC** o **Font.BOLD|Font.ITALIC**). Cualquier otro bit del parámetro de estilo es ingorado. Si el argumento de estilo no conforma ninguna de las máscaras esperadas, el estilo se selecciona a **PLAIN**.
- **size** - el tamaño de punto de la fuente.

Sumario de Constructores de la Clase **FontExtrusion**

Extiende: java.lang.Object

El objeto **FontExtrusion** se usa para describir el camino de extrusión de un objeto **Font3D**. Este camino de extrusión se usa en conjunción con un objeto **Font2D**. El camino de extrusión define el fondo del contorno del texto 3D. Este contorno es perpendicular a la cara del texto. La extrusión tiene su origen en el lateral de la letra siendo 1.0 la altura de la letra más alta. El contorno debe ser monotónico en el eje X. El usuario es responsable de la sanidad de los datos y debe asegurarse de que esta **extrusionShape** no causa intersecciones en letras adyacentes o dentro de una sola letra. No está definida la salida para extrusiones que causan intersecciones.

FontExtrusion()
Construye un objeto **FontExtrusion** con los parámetros por defecto.
FontExtrusion(java.awt.Shape extrusionShape)
Construye un objeto **FontExtrusion** con la forma especificada.

Sumario de Métodos de la Clase **FontExtrusion**

java.awt.Shape getExtrusionShape()
Obtiene el parámetro **shape** de FontExtrusion.
void setExtrusionShape(java.awt.Shape extrusionShape)
Selecciona el parámetro **shape** de FontExtrusion.

Fondo

Por defecto, el fondo de un universo virtual Java 3D es negro sólido. Sin embargo, podemos especificar otros fondos para nuestros mundos virtuales. El API Java 3D proporciona una forma fácil de especificar un color sólido, una imagen, una geometría o una combinación de éstos como fondo.

Cuando especificamos una imagen para el fondo, se sobreescribie la especificación del color de fondo, si existe. Cuando se especifica una geometría, se dibuja sobre el color de fondo o la imagen.
La única parte espinosa es la especificación de un fondo geométrico. Toda la geometría de fondo se especifica como puntos en una esfera. Si nuestra geometría es un **PointArray**, que podría representar estrellas a años luz, o un **TriangleArray**, que podría representar montañas en la distancia. La geometria de fondo se proyecta sobre el infinito cuando se renderiza.
Los objetos **Background** tienen límites de aplicación, lo que nos permite que se puedan especificar diferentes fondos para diferentes regiones del mundo virtual. Un nodo **Background** está activo cuando su región de aplicación intersecciona con el volumen de activación del **ViewPlatform**.
Si están activos varios nodos **Background**,el nodo que está más "cercano" al ojo será el utilizado. Si no hay ningún nodo **Background** activo, la ventana se mostrará en negro. Sin embargo, la definición de "más cercano" no está especificada. Por cercano, se elige el fondo con los límites de aplicación más internos que encierra la **ViewPlatform**.
Es improbable que nuestra aplicación necesite iluminar la geometría del fondo -- en realidad el sistema visual humano no puede percibir los detalles visuales a grandes distancias. Sin embargo, una geometría de fondo si puede ser sombreada. La geometría del fondo podría no contener luces, pero las luces definidas en el escenario gráfico pueden influenciar en la geometría del fondo.
Para crear un fondo seguimos esta sencilla receta:

1. Crear un objeto **Background** especificando un color o una imagen.
2. Añadir geometría (opcional).
3. Proporcionar un límite de Aplicación o **BoundingLeaf**.
4. Añadir el objeto **Background** al escenario gráfico.

- **Ejemplos de fondos**

Como se explicó en la sección anterior, un fondo puede tener un color o una imagen. La Geometría puede aparecer en el fondo con el color o la imagen. Esta

sección proporciona un ejemplo de un fondo blanco sólido. Un segundo ejemplo muestra la adicción de geométria al fondo.

Ejemplo de Fondo Coloreado

Las líneas de código del Fragmento de Código 3-6 corresponden con los pasos de la receta anterior. Junto a la personalización del color, el único posible ajuste es para definir unos límites de aplicación más apropiados para el fondo (o usar un **BoundingLeaf**).

Fragmento de Código 3-6, Añadir un fondo coloreado

```
1. Background backg = new Background(1.0f, 1.0f, 1.0f);
2. //
3. backg.setApplicationBounds(BoundingSphere());
4. contentRoot.addChild(backg);
```

Ejemplo de Geometría de Fondo

De nuevo, las líneas de código en el Fragmento de Código 3-7 corresponden con lo pasos de la receta de creacción de un fondo. En este fragmento, se llama al método createBackGraph() para crear la geometría del fondo. Este método devuelve un objeto **BranchGroup**. Para un ejemplo más completo puedes ver el fichero BackgroundApp.java.

Fragmento de Código 3-7, añadir un fondo geométrico

```
1. Background backg = new Background(); //black background
2. backg.setGeometry(createBackGraph()); // add BranchGroup of background
3. backg.setApplicationBounds(new BoundingSphere(new Point3d(), 100.0));
4. objRoot.addChild(backg);
```

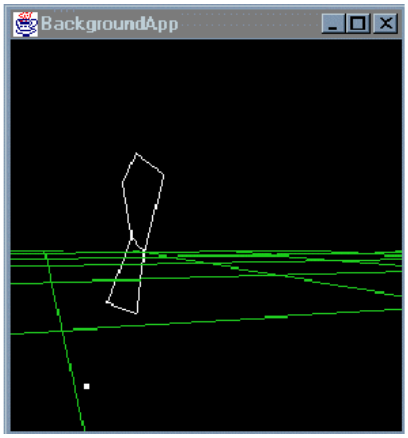
BackgroundApp.java

Para apreciar un fondo, necesitamos experimentarlo. BackgroundApp.java es una aplicación completa con un fondo geométrico. Esta aplicación nos permite movernos por un mundo virtual Java 3D. Mientras nos movemos, podemos ver el movimiento relativo entre la geometría local y la del fondo.

BackgroundApp.java usa la clase **KeyNavigatorBehavior** proprocionada por la librería de utilidades para visores de movimiento.

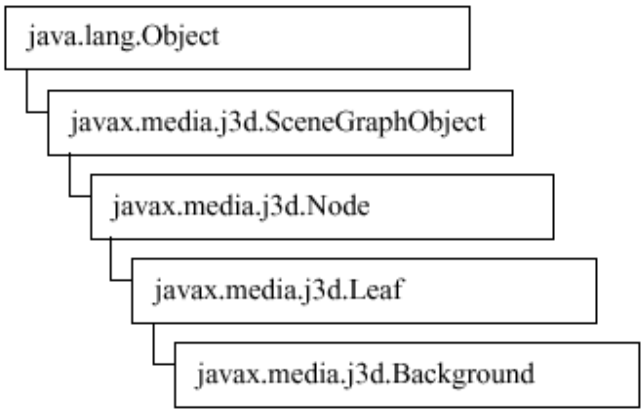
KeyNavigatorBehavior responde a las teclas de flechas, PgUp, y PgDn para el movimiento. La tecla Alt también juega un papel (para más detalles puedes ver el

Capítulo 4. Cuando ejecutes **BackgroundApp**, no te olvides de rotar para ver la _constellation_, así como viajar lejos en la distancia.



La clase Background

La Figura 3-13 muestra el árbol de clase de la clase **Background**. Como una extensión de la clase **Leaf**, un ejemplar de la clase **Background** puede ser un hijo de un objeto **Group**.



Background tiene varios constructores. Los constructores con parámetros nos permiten especificar un color o una imagen para el fondo. La geometría del fondo sólo se puede aplicar a través del método apropiado.

Sumario de Constructores de la Clase **Background**

El nodo hoja **Background** define un color sólido o una imagen para el fondo que se usa para rellenar la ventana al principio de cada nuevo marco. Opcionalmente permite referenciar geometrías de fondo. La geometría de fondo debe

representarse dentro de una esfera y es dibujada hacia el infinito. También especifica una región de aplicación en la que este fondo está activo.

Background()

Construye un nodo **Background** con un color por defecto (negro).

Background(Color3f color)

Construye un nodo **Background** con el color especificado.

Background(float r, float g, float b)

Construye un nodo **Background** con el color especificado.

Background(ImageComponent2D image)

Construye un nodo **Background** con la imagen especificada.

Cualquier atributo de un fondo puede seleccionarse a través de sus métodos.

Sumario de Métodos de la Clase **Background**

void setApplicationBoundingLeaf(BoundingLeaf region)

Selecciona la región de aplicación del **Background** a la hoja especificada.

void setApplicationBounds(Bounds region)

Selecciona la región de aplicación del **Background** a los límites especificados.

void setColor(Color3f color)

Selecciona el color del fondo.

void setColor(float r, float g, float b)

Selecciona el color del fondo.

void setGeometry(BranchGroup branch)

Selecciona la geometría del fondo al nodo **BranchGroup** especificado.

void setImage(ImageComponent2D image)

Selecciona la imagen del fondo.

Sumario de Capacidades de la Clase **Background**

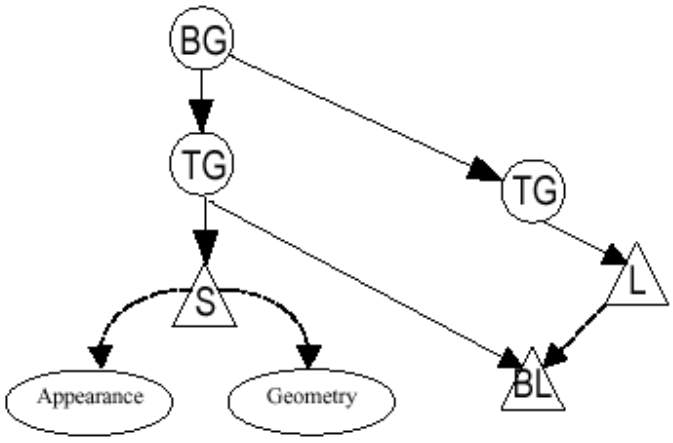
- **ALLOW_APPLICATION_BOUNDS_READ | WRITE** permite leer (escribir) al acceso a los límites de aplicación.
- **ALLOW_COLOR_READ | WRITE** permite leer (escribir) al acceso a su color
- **ALLOW_GEOMETRY_READ | WRITE** permite leer (escribir) al acceso a su geometría de fondo
- **ALLOW_IMAGE_READ | WRITE** permite leer (escribir) al acceso a su imagen

BoundingLeaf

Los **Bounds** (límites) se usan con luces, comportamientos, fondos y una gran variedad de otras aplicaciones en Java 3D. Los **Bounds** permiten al programador variar la acción, la apariencia, y/o el sonido sobre el campo virtual. La especificación de **Bounds** también permite al sistema de renderizado de Java 3D mejorar la ejecución del recortado y por lo tanto mejorar el rendimiento.

La especificación típica de límites utiliza un objeto **Bounds** para limitar una región. En el escenario gráfico resultante, los objetos **Bounds** se mueven con los objetos que lo referencian. Esto esta bién para muchas aplicaciones; sin embargo, podría haber situaciones en las que fuera deseable tener la región límite que se moviera independientemente de los objetos que usan los límites.

Por ejemplo, si un mundo incluye una fuente de luz estacionaria que ilumina unos objetos en movimiento, los límites de la luz deberían incluir el objeto en movimiento. Una forma de manejar esto podría ser crear los límites lo suficientemente grandes como para incluir todos los lugares donde se mueve el objeto. Esta no es la mejor respuesta en muchos casos. Una mejor solución es usar un **BoundingLeaf**. Situado en el escenario gráfico con el objeto visual, el **BoundingLeaf** se mueve con el objeto visual independientemente de la fuente de luz. La Figura 3-14 muestra un escenario gráfico con un una luz que usa un nodo **BoundingLeaf**.



Una aplicación interesante de un objeto **BoundingLeaf** sitúa un **BoundingLeaf** en la **viewPlatform**. Este **BoundingLeaf** puede usarse para un límite "siempre sobre"

para un comportamiento, o para unos límites de aplicación "aplica siempre" para fondos o nieblas. El Fragmento de Código 3-8 presenta un ejemplo de la aplicación **BoundingLeaf** usada con un **Background**.

El Fragmento de Código 3-8 presenta un ejemplo de cómo añadir un **BoundingLeaf** como un hijo de **PlatformGeometry** para proporcionar un límite de "aplica siempre" para un fondo. En este código se ha modificado el método estándard createSceneGraph() para que tome un sólo parámetro, que es el objeto **SimpleUniverse**. Esto es necesario para crear el objeto **PlatformGeometry**.

Las líneas 2, 3 y 4 crean el objeto **BoundingLeaf**, el objeto **PlatformGeometry** y hace del objeto **BoundingLeaf** un hijo de **PlatformGeometry**, en este orden. Si tuviera que haber más objeto **PlatformGeometry**, se añadirían en este punto. El objeto **PlatformGeometry** se añade la rama de vista gráfica en la línea 6.

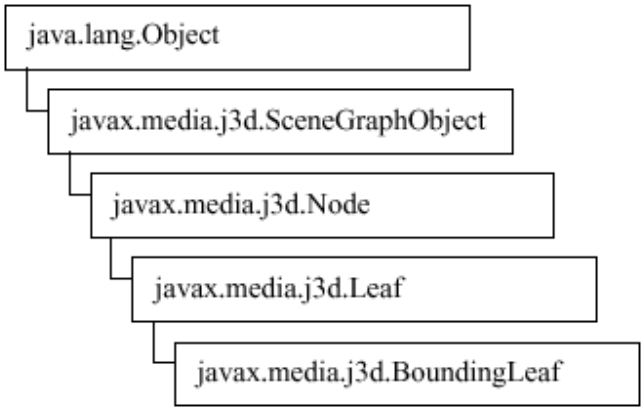
El objeto **BoundingLeaf** se selecciona como los limites de aplicación para el objeto **background** en la línea 11. El mismo objeto **BoundingLeaf** puede usarse para otros propósitos en este programa. Por ejemplo, puede usarse para los comportamientos. Observa que usando el **BoundingLeaf** en este programa como el **InfluencingBoundingLeaf** de una luz hace que esta luz no influya en todos los objetos del mundo virtual.

Fragmento de Código 3-8, Añadir un BoundingLeaf al ViewPlatform para un límite 'Aplica siempre'

```
1.  void createSceneGraph (SimpleUniverse su) {
2.  BoundingLeaf boundingLeaf = new BoundingLeaf();
3.  PlatformGeometry platformGeom = new PlatformGeometry();
4.  platformGeom.addChild(boundingLeaf);
5.  platformGeom.compile();
6.  simpleUniv.getViewingPlatform().setPlatformGeometry(platformGeom);
7.
8.  BranchGroup contentRoot = new BranchGroup();
9.
10. Background backg = new Background(1.0f, 1.0f, 1.0f);
11. backg.setApplicationBoundingLeaf(boundingLeaf);
12. contentRoot.addChild(backg);
```

La Clase BoundingLeaf

La clase **BoundingLeaf** extiende la clase **Leaf**. La Figura 3-15 representa el árbol de clases de BoundingLeaf.



El constructor sin parámetros de **BoundingLeaf** crea límites para una esfera. El otro constructor permite la especificación de límites para el objeto **BoundingLeaf**.

Sumario de Constructores de la Clase **BoundingLeaf**

El nodo **BoundingLeaf** define una región de límites que puede ser referenciada por otros nodos para definir un región de influencia, o una región programada.

BoundingLeaf()

Construye un nodo **BoundingLeaf** con un objeto esfera.

BoundingLeaf(Bounds region)

Construye un nodo **BoundingLeaf** con la región de límites especificada.

Sumario de Métodos de la Clase **BoundingLeaf**

Bounds getRegion()

Recupera la región de límites de este **BoundingLeaf**

void setRegion(Bounds region)

Selecciona la región de límites de este nodo **BoundungLeaf**

Datos de Usuario

Cualquier **SceneGraphObject** puede referenciar cualquier otro objeto como datos de usuario. Primero, deberíamos habernos dado cuenta de que casi cualquier clase del corazón del API Java 3D es un descendiente de **SceneGraphObject**. La lista de descendientes de **SceneGraphObject** incluye **Appearance**, **Background**, **Behavior**, **BranchGroup**, **Geometry**, **Lights**, **Shape3D**, y **TransformGroup**.
Las aplicación para el campo **UserData**, sólo está limitado por nuestra imaginación. Por ejemplo, una aplicación podría tener varios objetos recolectables. Cada uno de estos objetos podría tener algún texto informativo almacenado en el objeto de datos de usuario. Cuando el usuario recoge un objeto, se puede mostrar la información de los datos de usuario.
Otra aplicación podría almacenar algún valor calculado para un objeto de escenario gráfico como su posición en las coordenadas del mundo virtual. Y otra aplicación podría almacenar alguna información especifica de comportamiento que podría controlar el comportamiento aplicado a varios objetos.

Lista Parcial de Métodos de Datos de Usuario de **SceneGraphObject**

SceneGraphObject es una superclase común para todos los objetos componentes de un escenario gráfico. Estos incluyen **Node**, **Geometry**, **Appearance**, etc.

java.lang.Object getUserData()
Recupera el campo **userData** desde este objeto del escenario gráfico.
void setUserData(java.lang.Object userData)
Selecciona el campo **userData** asociado con este objeto del escenario gráfico.

Interacción en Java 3D

Comportamiento: la Base para Interacción y Animación

La interacción y la animación se especifican con objetos **Behavior**. La clase **Behavior** es una subclase abstracta que proporciona el mecanismo para incluir código que modifique el escenario gráfico. La clase **Behavior**, y sus descendientes, son enlaces a código del usuario que proporciona las modificaciones para los gráficos y los sonidos del universo virtual.
El propósito del objeto **Behavior** en un escenario gráfico es modificar el propio escenario gráfico, o los objetos que hay dentro de él, en respuesta a algunos estímulos. Un estímulo puede ser una pulsación de tecla, un movimiento del ratón, la colisión de objetos, el paso del tiempo, algún otro evento, o una combinación de estos. Los cambios producidos incluyen la adicción de objetos al escenario gráfico, la eliminación de objetos, cambio de atributos de los objetos del escenario gráfico, reordenación de los objetos del escenario gráfico, o una combinación de estos.
Las posibilidades sólo están limitadas por las capacidades de los objetos del escenarios gráfico.

Aplicaciones de Behavior

Como un comportamiento (Behavior) es un enlace entre un estímulo y una acción, si consideramos todas las combinaciones posibles entre estímulos y acciones podremos obtener todas las aplicaciones de los objetos **Behavior**. La siguiente tabla muestra algunas de las posibilidades de **Behavior**, listando los posibles estímulos hacia abajo, y los posibles cambios hacia la derecha.
La tabla no lista todas las combinaciones posibles, sólo las más simples (un estímulo resulta en un cambio). Algunas combinaciones de estímulos y cambios sólo tienen sentido en un entorno específico; estas se listan como "específicas de la aplicación".

Estímulo	**Objeto del cambio**			
(razón para el	**TransformGroup** (los objetos visuales	**Geometry** (los objetos	**Scene Graph** (añadir, eliminar	**View** (cambiar la

cambio)	cambian la orientación o la localización)	visuales cambian la forma o el color)	o intercambiar objetos)	localización o dirección de la vista)
usuario	interacción	específico de la aplicación	específico de la aplicación	navegación
colisiones	Los objetos visuales cambian su orientación o posición	Los objetos visuales cambian su apariencia con la colisión	Los objetos visuales desaparecen con la colisión	La vista cambia con la colisión
tiempo	animación	animación	animación	animación
Posición de la Vista	cartelera	nivel de detalles (LOD)	específico de la aplicación	específico de la aplicación

La cosas naturales, como los árboles, utilizan una tremenda cantidad de geometría para representar de forma segura todas la estructura de ramas, hojas y tronco. Una alternativa es usar un polígono texturado en lugar de la geometría. Esta técnica algunas veces es referida como la aproximación cartelera. Esto es cierto especialmente cuando se usa un comportamiento para orientar automáticamente el polígono texturado hacia el espectador para que sólo se vea el frente de la superficie texturada. Este comportamiento de orientación se llama comportamiento cartelera.

Esta aproximación es efectiva cuando el objeto a representar por la textura está lejano para que las partes individuales del objeto visual no sean fácilmente distinguibles. Para el ejemplo del árbol, si el espectador está tan alejado que las ramas son dificiles de distinguir, no merece la pena gastar recursos de memoria y de cálculo para representar todas las hojas del árbol. Esta técnica está recomendada para cualquier aplicación que requiera visualizar objetos complejos en la distancia. Sin embargo, si el espectador puede aproximarse a la cartelera, a cierta distancia el grado de profundidad del polígono textura podría ser detectado por el espectador.

El comportamiento de nivel de detalle (LOD) tiene una aplicación relacionada. Con LOD, los objetos visualmente complejos son representados por múltiples objetos visuales variando los niveles de detales (de ahí su nombre). La representación del objeto visual con menor nivel de detalle se usa cuando el espectador está lejos. La

representación con más nivel de detalle se usa cuando el espectador está muy cerca. El comportamiento LOD cambia automáticamente entre las representación del objeto basándose en la distancia al espectador.
Los comportamientos de cartelera y de nivel de detalle corresponden a clases extendidas desde **Behavior** que implementan estas aplicaciones comunes. Son posibles otros comportamientos especilizados y varios de ellos se pueden ver en la Figura 4-1. Por ejemplo, hay varias clases **MouseBehavior** que manipulan una transformación en respuesta a movimientos del ratón. Normalmente la transformación de la vista se cambia por el comportamiento del ratón para cambiar la vista en respuesta a una acción del ratón.
Observa también como los comportamientos pueden encadenarse. Por ejemplo, los movimientos del ratón o las pulsaciones de teclas pueden usarse para cambiar la vista. En respuesta al movimiento de la vista, podrían tener lugar otros comportamientos como la cartelera, o el nivel de detalles. Afortunadamente, cada comportamiento se especifica de forma separada.

Animación contra Interacción

Como la distinción entre animación e interacción usada en este tutorial está bastante bien, aquí hay un ejemplo para clarificar esta distinción. Si un usuario navega en un programa donde se proporciona un comportamiento, la vista se moverá en respuesta a eventos del teclado y/o ratón. El movimiento de la plataforma de la vista es una interacción porque es el resultado directo de una acción del usuario. Sin embargo, otras cosas podrían cambiar como resultado del movimiento de la plataforma de la vista, (por ejemplo, comportamientos de cartelera o LOD). Los cambios causados como resultado del movimiento de la plataforma de vista son indirectamente causados por el usuario y por lo tanto son animaciones.

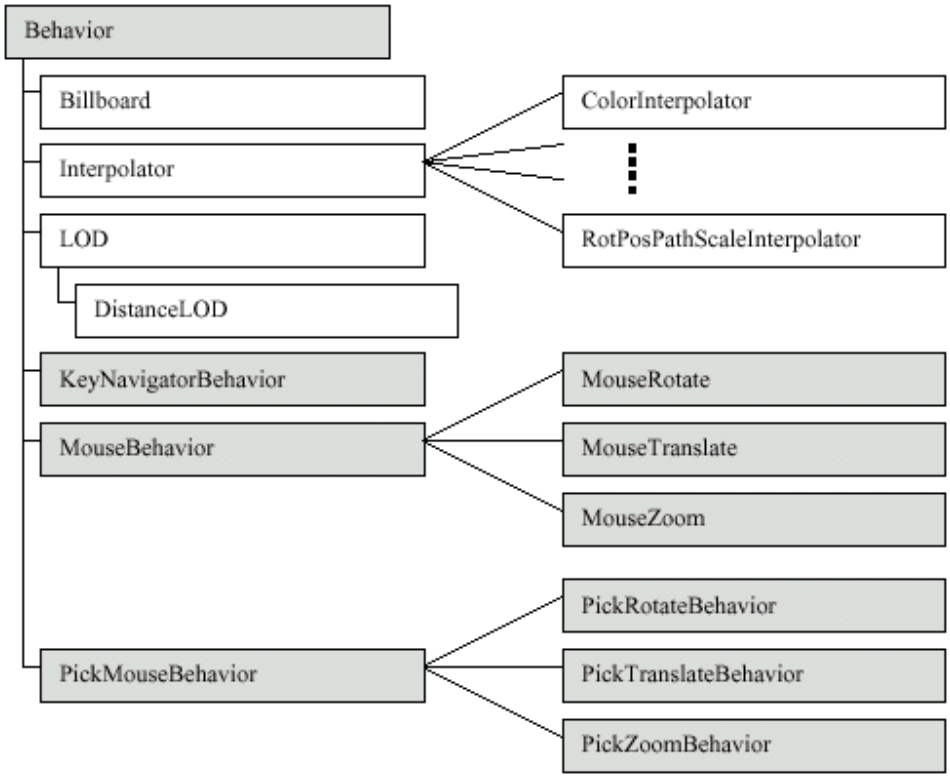
Introducción a la clases Behavior

La Figura 4-1 muesta especializaciones de la clase **Behavior** creadas en el API de Java 3D. También son posibles las clases especializadas de **Behavior** definidas

por el usuarios y están sólo limitadas por la imaginación del programador. Este módulo del tutorial cubre cada una de las clases de la Figura 4-1. Este capítulo cubre las clases sombreadas, el siguiente capítulo cubre el resto.

Behavior Básico

Como se explicó en la sección anterior, las clases **Behavior** se usan en muchas aplicaciones Java 3D y de muchas formas. Es importante entender las consideraciones de funcionamiento y programación de estas clases. Esta sección explica la clase **Behavior**, ofrece una receta para programar clases de comportamientos personalizadas, y muestra una aplicación de ejemplo que usa una clase **Behavior**.



Escribir una Clase Behavior

Esta sección explica cómo escribir una clase de comportamiento personalizado. Ya sabemos que hay clases de comportamiento que podemos usar. Sin embargo,

al ver cómo crear una clase **Behavior** aprenderemos como funciona. Por eso, incluso si no planeas usar una clase comportamiento, podrías querer leer esta sección. Las clases escritas en esta sección se usan en la siguiente.

Mecánismo de Behaviors

Una clase de comportamiento personalizado implementa los métodos de inicialización y processStimulus de la clase abstracta **Behavior**. Por supuesto, la clase de comportamiento personalizado, también tiene al menos un constructor y también podría tener otros métodos.

La mayoría de los comportamientos actuarán sobre un objeto del escenario gráfico para afectar al comportamiento. El objeto sobre el que actúa un comportamiento es referido como el objeto del cambio. Es a través de este objeto, u objetos, que el comportamiento afecta al mundo virtual. Aunque es posible tener un comportamiento que no tenga un objeto del cambio, la mayoría lo tienen.

El comportamiento necesita una referencia a su objeto(s) de cambio para poder realizar los cambios de comportamiento. Se puede usar el constructor para seleccionar la referencia del objeto de cambio. Si no se hace, otro método de la clase de comportamiento personalizado debe almacenar esta información. En cualquier caso, la referencia se hace en el momento en que se construye el escenario gráfico, que es el primer cálculo de comportamiento.

El método de inicialización se invoca cuando el escenario gráfico que contiene la clase de comportamiento se vuelve vivo. Este método de iniciación es responsable de seleccionar el evento de disparo inicial para el comportamiento y seleccionar la condición inicial de las variables de estado del comportamiento. El disparo se especifica como un objeto **WakeupCondition**, o una combinación de objetos **WakeupCondition**.

El método processStimulus se invoca cuando ocurre el evento de disparo especificado para el comportamiento. Este método es responsable de responder al evento. Como se pueden codificar muchos eventos en un sólo objeto **WakeupCondition** (por ejemplo, varias acciones de teclado podrían estár codificados en un **WakeupOnAWTEvent**), esto incluye la descodificación del evento. El método processStimulus responde al estímulo, normalmente modificando

el objeto de cambio, y, cuando es apropiado, reseteando el disparo. Abajo tenemos una receta para escribir una clase de comportamiento personalizada:

1. escribir (al menos uno) constructor
   almacenar una referencia al objeto del cambio.
2. sobreescribir public void initialization()
   especificar el criterio de diparo inicial
3. sobreescribir public void processStimulus()
   decodificar la condición de disparo
   actuar de acuerdo a la condición de disparo
   resetar el disparo si es apropiado

La receta anterior muestra los pasos básicos para crear una clase de comportamiento personalizada. Los comportamientos complejos podrían requerir más programación que la descrita en la receta. El uso de un objeto **Behavior** es otro problema y se discute en una sección posterior. Pero antes, usaremos esta receta para crear el siguiente ejemplo de clase **Behavior**.

Ejemplo de Clase Behavior Personalizada: SimpleBehavior

Para el ejemplo de comportamiento personalizado, la clase implementará un comportamiento sencillo para hacer que algo gire en respuesta a pulsaciones del teclado.

Para crear dicha clase, todo lo que necesitamos es un referencia a un **TransformGroup** (el objeto del cambio para esta clase), y una variable con el ángulo. En respuesta a una pulsación de tecla la variable del ángulo se modifica, y el ángulo de la fuente del **TransformGroup** se selecciona al valor del ángulo. Como el comportamiento actuará sobre un objeto **TransformGroup**, que está siendo rotado no es un problema.

Para crear esta clase no se necesita nada más que los tres ingredientes esenciales que se listarón en la receta: un constructor, el método initialization() y el método processStimulus. El constructor almacenará la referencia al objeto **TransformGroup**. El método initialization() selecciona el disparo inicial a **WakeOnAWTEvent**, y el ángulo de rotación a cero. Como se mencionó antes, el estímulo para un comportamiento se especifica como un objeto **WakeupCondition**.

Cómo sólo hay una posible condición de disparo, el método processStimulus no la descodifica. Es posible posteriormente descodificar el evento de pulsación de tecla para determinar qué tecla, o combinación de teclas, se pulsó.
El método processStimulus siempre incrementa la variable del ángulo, entonces lo usa para ajustar el objeto **TransformGroup**. El último trabajo de este método es resetear el disparo. En este ejemplo, el disparo siempre se resetea a una pulsación de tecla. Los comportamientos pueden cambiar el evento de disparo en el tiempo para cambiar comportamientos (otra razón para tener que descodificar el evento de disparo), o no seleccionar otro disparo para comportamientos de una sóla vez.
El Fragmento de Código 4-1 presenta la clase **SimpleBehavior** (SimpleBehaviorApp.java) que es una implementación de la clase de comportamiento personalizada. Las sentencias import son necesarias para esta clase. java.awt.event es necesaria para la interacción con el teclado. java.util.eumeration es necesaria para decodificar el **WakeupCondition**; y por lo tanto necesaria virtualmente para casi cualquier clase de comportamiento personalizado. También son necesarias las sentencias import normales del API Java 3D.

Fragmento de Código 4-1, Clase SimpleBehavior de SimpleBehaviorApp.java

```
1.  import java.awt.event.*;
2.  import java.util.Enumeration;
3.
4.  // SimpleBehaviorApp renders a single, rotated cube.
5.
6.  public class SimpleBehaviorApp extends Applet {
7.
8.      public class SimpleBehavior extends Behavior{
9.
10.     private TransformGroup targetTG;
11.     private Transform3D rotation = new Transform3D();
12.     private double angle = 0.0;
13.
```

```
14.     // create SimpleBehavior - set TG object of change
15.     SimpleBehavior(TransformGroup targetTG){
16.         this.targetTG = targetTG;
17.     }
18.
19.     // initialize the Behavior
20.     // set initial wakeup condition
21.     // called when behavior becomes live
22.     public void initialize(){
23.         // set initial wakeup condition
24.        this.wakeupOn(new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
25.     }
26.
27.     // called by Java 3D when appropriate stimulus occurs
28.     public void processStimulus(Enumeration criteria){
29.         // do what is necessary in response to stimulus
30.         angle += 0.1;
31.         rotation.rotY(angle);
32.         targetTG.setTransform(rotation);
33.         this.wakeupOn(new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
34.     }
35.
36.  } // end of class SimpleBehavior
```

Esta clase sólo demuestra la programación básica necesaria para este comportamiento sencillo. Se pude mejorar, por ejemplo, se podrían seleccionar el ángulo y/o el eje de rotación por métodos de la clase. La clase podría además personalizarse con un método para seleccionar una tecla específica, o conjunto de teclas, a las que responder.

Otra mejora definitiva de la clase podría prevenir la sobrecarga de la variable del ángulo, en la clase actual, el valor para el ángulo podría crecer sin límites incluso aunque los valores de 0.0 a 2P sean todo lo necesario. Aunque es improbable, es posible que esta variable genere una sobrecarga y cause una excepción en tiempo de ejecución.

Riesgos de Programación al Escribir Clases Behavior

En los tres pasos de la receta para crear una clase behavior personalizada, los dos errores más comunes son:

- olvidarse de seleccionar y resetear el disparo del comportamiento, y
- no volver de los métodos de la clase **Behavior**.

Obviamente, si no se selecciona el disparo inicial en el método initialization(), el comportamiento nunca será invocado. Un poco menos obvio es que el disparo debe seleccionarse de nuevo en el método processStimulus() si se desea un comportamiento repetido.
Como estos dos métodos (initialization() y processStimulus()) son llamados por el sistema Java 3D, deben volver para permitir que continúe el renderizado. Por ejemplo, si se desea un ejemplo peonza, el ángulo y el **TransformGroup** necesitan actualizarse periódicamente. Si nuestro comportamiento implementa este comportamiento sin deshilar un thread, no se renderizará nada más. También, hay una forma mucho mejor para conseguir este tipo de comportamiento.

Usar una Clase Behavior

Encontrar o escribir la clase **behavior** apropiada para nuestra aplicación es el principio para escribir un programa Java 3D interactivo. Esta sección cubre los problemas de programación en la adicción de objetos **behavior** a los programas.
El primer paso implica el asegurarnos de que el escenario gráfico hace provisiones para el **behavior**. Por ejemplo, para usar la clase **SimpleBehavior** de la sección anterior debe haber un **TransformGroup** en el escenario gráfico sobre el objeto a rotar. Muchos comportamientos sólo necesitan un único objeto **TransformGroup**; sin embargo, los requerimientos de un escenario gráfico para una comportamiento dependen de la aplicación y del propio comportamiento y podrían ser más complejos.
Habiendo establecido el soporte para un comportamiento, se debe añadir un ejemplar de la clase al escenario gráfico. Sin ser una parte de un escenario gráfico vivo, no hay forma de poder inicializar un comportamiento. De hecho, un objeto

behavior que no es parte de un escenario gráfico se convertirá en basura y será eliminado en la próxima recolección.

El último paso para añadir comportamiento es proporcionar unos límites para el comportamiento. Para mejorar la eficiencia, Java 3D usa los límites para realizar el recorte de ejecución. El comportamiento sólo está activo cuando sus límites interseccionan un volumen de activación de la **ViewPlatform**. Solo los comportamientos activos son elegibles para recibir estímulos. De esta forma, los estímulos pueden ser ignorados por algunos comportamientos. El programador tiene control sobre el recorte de ejecución a través de la selección de los límites del comportamiento.

La siguiente lista muestra una receta con los pasos para usar un objeto **behavior**.

1. preparar el escenario gráfico (añadiendo un **TransformGroup** u otros objetos necesarios)
2. insertar el objeto **behavior** en el escenario gráfico, referenciando el objeto del cambio
3. especificar los límites (o **SchedulingBoundingLeaf**)
4. seleccionar la capacidades de escritura (y lectura) del objeto fuente (según sea apropiado)

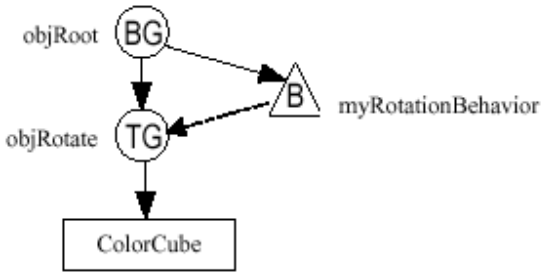
El Fragmento de Código 4-2 es un extracto del programa de ejemplo SimpleBehaviorApp.java y es la continuación del Fragmento de Código 4-1

Fragmento de Código 4-2, El método CreateSceneGraph en SimpleBehaviorApp.java

```
37.    public BranchGroup createSceneGraph() {
38.        // Create the root of the branch graph
39.        BranchGroup objRoot = new BranchGroup();
40.
41.        TransformGroup objRotate = new TransformGroup();
42.        objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
43.
44.        objRoot.addChild(objRotate);
45.        objRotate.addChild(new ColorCube(0.4));
46.
47.        SimpleBehavior myRotationBehavior = new SimpleBehavior(objRotate);
48.        myRotationBehavior.setSchedulingBounds(new BoundingSphere());
49.        objRoot.addChild(myRotationBehavior);
```

```
50.
51.       // Let Java 3D perform optimizations on this scene graph.
52.       objRoot.compile();
53.
54.   return objRoot;
55.   } // end of CreateSceneGraph method of SimpleBehaviorApp
```

Se necesita muy poco código para completar el programa de los fragmentos de código 4-1 y 4-2. El programa completo está en: SimpleBehaviorApp.java. La aplicación completa renderiza un objeto **ColorCube** en una escena estática hasta que se pulsa una tecla. En respuesta a la pulsación de la tecla, el **ColorCube** rota 0,1 radianes (unos 6°). La Figura 4-4 muestra el diagrama del escenario gráfico para la rama de contenido gráfico de esta aplicación.



El diagrama anterior muestra claramente la relación entre el objeto **behavior** y el objeto del cambio, el objeto **TransformGroup**. El ejemplo rota un **ColorCube**, pero la clase **Behavior** no está limitada a esto. Puede rotar cualquier objeto visual, o porción de una escena gráfica que sea hija de un objeto **TransformGroup**.

Este sencillo ejemplo no está pensado para demostrar todas las posibilidades de los comportamientos; es sólo un punto de arranque en la exploración de los comportamientos. En secciones posteriores veremos el API de la clase **Behavior**.

Riesgos de Programación al usar Objetos Behavior

En la receta de tres pasos para usar clases **Behavior**, los dos errores más comunes son:

- no especificar (correctamente) los límites, y
- no añadir un **behavior** al escenario gráfico.

La intersección de los límites de un **behavior** con el volumen de activación de una vista determina si el evento Java 3D considera el disparo del estímulo para el

behavior. Java 3D no avisará si no ponemos los límites -- el comportamiento nunca se disparará. También debemos mantener los límites de cada objeto **behavior** tan pequeños como sea posible para una mejora global del rendimiento. Como se mencionó arriba, un objeto **behavior** que no forma parte de un escenario gráfico será considerado basura y será eliminado en el siguiente ciclo del recolector de basura. Esto, también sucederá sin errores ni avisos.

¿Dónde Debería ir un Objeto Behavior en un Escenario Gráfico?

Los comportamientos pueden situarse en cualquier lugar del escenario gráfico. Los problemas para esta localización son: 1) el efecto de los límites, y 2) el mantenimiento del código.

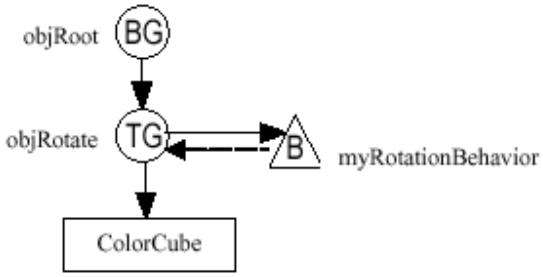
El objeto **bounds** referenciado por un objeto **behavior** está sujeto al sistema local de coordenadas creado en **SimpleBehaviorApp**, el objeto **SimpleBehavior** y **ColorCube** no están sujetos al mismo sistema local de coordenadas. En la aplicación de ejemplo esto no crea un problema. El objeto **TransformGroup** del ejemplo sólo rota el **ColorCube** para que los límites del objeto **myRotationBehavior** siempre encierren el objeto **ColorCube** permitiendo la interacción con el **ColorCube** cuando es visible.

Sin embargo, si el objeto **TransformGroup** se usara para trasladar el objeto **ColorCube**, sería posible moverlo fuera de la vista. Como el objeto **bounds** permanece con el objeto **behavior** en la escena, el usuario podría continuar moviendo el objeto. Mientras que el volumen de activación de una vista intereseccionа los límites del comportamiento, éste está activo.

Siendo posible interactuar con un objeto visual que no está en la vista no está mal (si esto es lo que queremos). El problema viene si la vista a cambiar dicho volumen de activación no intersecciona con límites del comportamiento, incluso para incluir el objeto visual, el comportamiento está inactivo. Por eso el objeto visual con el que queremos interactuar podría estar a nuestra vista pero inactivo. La mayoría de los usuarios consideran esto un problema (incluso si es intencional).

Hay dos soluciones a este problema. Una es cambiar el escenario gráfico para mantener los límites del comportamiento con el objeto visual. Esto se consigue

fácilmente como se demuestra en la Figura 4-5. La solución alternativa usa un objeto **BoundingLeaf** para los límites.



Recomendaciones de Diseño para la Clase Behavior

El mecanismo de escritura de un comportamiento personalizado es sencillo. Sin embargo, deberíamos tener en cuentra que un comportamiento pobremente escrito puede degradar el rendimiento del renderizado. Mientras que hay otras consideraciones en la escritura de un comportamiento, hay dos cosas que debemos evitar: quemar la memoria y condiciones de disparo innecesarios.

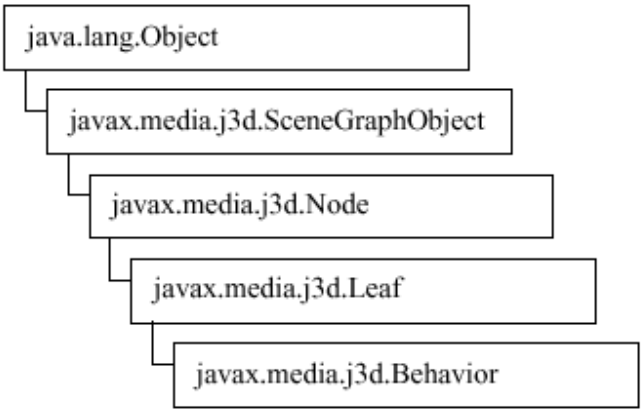
'Quemar la Memoria' es el término para la creacción de objetos innecesarios en Java. La quema de memoria excesiva causará la recolección de basura. Las pausas ocasionales en el renderizado son típicas de la quema de memoria ya que durante la recolección de basura, el renderizado se parará.

Los métodos de la clase **Behavior** frecuentemente son responsables de crear problemas de quema de memoria. Por ejemplo, en el Fragmento de Código 4-1 el **processStimulus** usa un 'new' en la invocación de **wakeupOn** (línea 24). Esto causa que se cree un nuevo objeto cada vez que se invoca a este método. El objeto se convierte en basura cada vez que se dispara el comportamiento.

Los problemas potenciales de la quema de memoria son fáciles de indentificar y evitar. Buscamos cualquier uso de 'new' en el código para encontrar la fuente de estos tipos de problemas. Siempre que sea posible, reemplazaremos el uso de 'new' con código que reutilice un objeto.

API de la Clase Behavior

Esta sección presenta los detalles del API de la clase **Behavior**. La Figura 4-6 muestra el árbol de clases del API Java 3D que incluye la clase **Behavior**. Como clase abstracta, la clase **Behavior** debe ser extendida antes de poder ejemplarizar un objeto **behavior**. Por supuesto, podemos escribir nuestras clases **behavior** personalizadas. Además, hay muchas clases **behavior** existentes en los paquetes de utilidad de Java 3D. Como una extensión de la clase **Leaf**, los ejemplares que extienden **Behavior** pueden ser hijos de un **group** en un escenario gráfico.



Anteriormente hemos visto los métodos processStimulus() e initialize(). Ahora vamos a ver el resto de los métodos de la clase **Behavior**.

El método wakeupOn() se usa en los métodos initialize() y processStimulus() para seleccionar el disparo para el comportamiento. El parámetro de este método es un objeto **WakeupCondition**. En secciones posteriores veremos **WakeupCondition**, y las clases relacionadas.

El método postId() permite a un comportamiento comunicarse con otro método. Una de las condiciones de disparo es **WakeupOnBehaviorPost**. Los objetos **Behavior** pueden estar coordinados para crear colaboraciones complejas usando el método postId() en conjunción con condiciones **WakeupOnBehaviorPost** apropiadas.

El método setEnable() proporciona la forma de desactivar un comportamiento incluso si los límites están activos. El valor por defecto es **true** (es decir, el objeto comportamiento está activado).

Un objeto **behavior** está activo sólo cuando sus límites intereseccionan con el volumen de activación de un **View**. Como es posible tener varias vistas en un universo virtual, un comportamiento puede hacerse activo por más de una vista.

El método getView() es útil con comportamientos que tratan con información por-vista (por ejemplo, Billboard, LOD) y con comportamientos en general para programar en el tiempo. Este método devuelve una referencia al objeto **View** primario asociado actualmente con el comportamiento. No existe el correspondiente método setView. La vista "primaria" se define como la primera vista adjunta a un **ViewPlatform** vivo, si hay más de una vista activa. Por eso, por ejemplo, los comportamientos **Billboard** podrían orientar hacia adelante esta vista primaria, en caso de varias vistas activas dentro del mismo escenario gráfico.

Sumario de Métodos de la Clase **Behavior**

Behavior es una clase abstracta que contiene el marco de trabajo para los componentes de comportamiento en Java 3D.

View getView()
Devuelve la vista primaria asociada con este comportamiento.
void initialize()
Inicializa este comportamiento.
void postId(int postId)
Postea la identidad especificada.
void processStimulus(java.util.Enumeration criteria)
Procesa un estímulo para este comportamiento.
void setEnable(boolean state)
Activa o desactiva este comportamiento.
void setSchedulingBoundingLeaf(BoundingLeaf region)
Selecciona la región de límites del comportamiento con los límites del leaf especificado.
void setSchedulingBounds(Bounds region)
Selecciona la región de límites del comportamiento con los límites especificados.
void wakeupOn(WakeupCondition criteria)
Define este criterio de disparo del comportamiento.

API ViewPlatform

Los Comportamientos están activos (dispuestos para ser disparados) sólo cuando sus límites (o BoundingLeaf) intersecciona con el volumen de activación de una **ViewPlatform**.

Lista Parcial de Métodos de la Clase **ViewPlatform**

Estos métodos de la clase **ViewPlatform** obtienen y seleccionan el radio del volumen de activación (esfera). El valor por defecto es 62.

float getActivationRadius()

Obtiene el radio de activación del ViewPlatform.

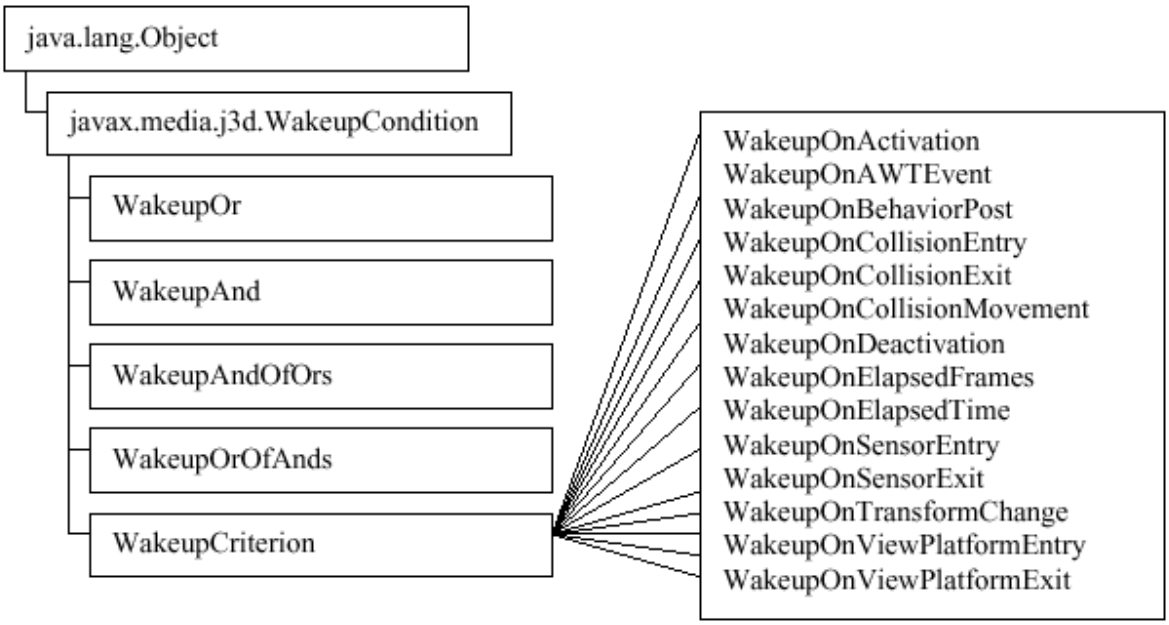
void setActivationRadius(float activationRadius)

Selecciona el radio de activación del ViewPlatform que define un volumen de activación alrededor de la plataforma.

Condiciones de Disparo: Cómo se Disparan los Comportamientos

Los comportamientos activados se disparan por la ocurrencia de uno o más estimulos especificados. El estimulo de disparo para un comportamiento se especifica usando descendientes de la clase **WakeupCondition**.

La clase abstracta, **WakeupCondition**, es la base para todas las clases de disparo del API Java 3D. Cinco clases extienden **WakeupCondition**, una es la clase abstracta **WakeupCriterion**, las otras cuatro permiten la composición de múltiples condiciones de disparo en una única condición de disparo. La Figura 4-7 muestra el árbol de clases.



Una condición de disparo para un objeto **behavior** se puede especificar como un criterio de disparo específico o como una combinación de criterios usando clases compuestas. Las siguientes secciones describen **WakeupCondition** y sus subclases.

- **WakeupCondition**

La clase **WakeupCondition** proporciona dos métodos. El primer método, allElements, devuelve una lista **enumeration** de todos los criterios de disparo para el objeto **WakeupCondition**. El otro método, triggeredElements, enumera qué criterio ha causado que el comportamiento sea disparado. Este método podría ser muy útil en el método processStimulus de un objeto **Behavior**.

Sumario de Métodos de **WakeupCondition**

La clase abstracta **WakeupCondition** es la base para todas las clases **wakeup**. Proporciona los siguientes métodos:

Enumeration allElements()
Devuelve una enumeración con todos los objetos WakeupCriterion en esta condición.

Enumeration triggeredElements()

Devuelve una enumeración de todos los objetos WakeupCriterion disparados en esta condición.

WakeupCriterion

WakeupCriterion es una clase abstracta para todas las clases **wakeup**. **WakeupCriterion** sólo proporciona un método: hasTriggered. Probablemente no necesitaremos usar este método ya que el método triggeredElements de **WakeupCondition** realiza esta operación por nosotros.

Sumario de Métodos de **WakeupCriterion**

boolean hasTriggered()

Devuelve **true** si el criterio disparó el comportamiento.

Clases WakeupCriterion Específicas

La Tabla 4-2 presenta las 14 clases **WakeupCriterion** específicas. Estas clases se usan para especificar las condiciones de disparo de los objetos **behavior**. Los ejemplares de estas clases se usan individualmente o en combinaciones.

Clase Criterio	Disparo
WakeupOnActivation	en la primera detección de una intersección del volumen de activación de un **ViewPlatform** con la región límite del objeto.
WakeupOnAWTEvent	cuando ocurre un evento AWT específico
WakeupOnBehaviorPost	cuando un objeto **behavior** envía un evento específico
WakeupOnCollisionEntry	en la primera detección de colisión del objeto especificaco con otro objeto del escenario gráfico.
WakeupOnCollisionExit	cuando el objeto especifico no colisiona con ningún otro objeto del escenario gráfico.
WakeupOnCollisionMovement	cuando el objeto especificado se mueve mientras colisiona con otro objeto del escenario gráfico
WakeupOnDeactivation	cuando el volumen de activación de un **ViewPlatform** deja de intereseccionar con los límites del objeto
WakeupOnElapsedFrames	cuando ha pasado un número determinado de frames

WakeupOnElapsedTime	cuando ha pasado un número de segundos determinado
WakeupOnSensorEntry	en la primera detección de cualquier sensor que intersecciona con los límites especificados
WakeupOnSensorExit	cuando un sensor que interseccionava con los límites del objeto deja de interseccionar con los límites especificados
WakeupOnTransformChange	cuando cambia la transformación dentro de un **TransformGroup** especificado
WakeupOnViewPlatformEntry	en la primera detección de intersección del volumen de activación de un **ViewPlatform** con los límites especificados
WakeupOnViewPlatformExit	cuando el volumen de activación de una vista deja de intereseccionar con los límites especificados

Comentarios Generales sobre WakeupCriterion

Varias clases **WakeupCriterion** se disparan con la "primera detección" de un evento. Lo que significa que el criterio sólo se disparará una vez por cada evento. Por ejemplo, un objeto **WakeupOnActivation** disparará la intersección del volumen de activación de un **ViewPlatform** y la región de límites del objeto **behavior** asociado. Mientras que la intersección persista, el **WakeupCondition** no se disparará de nuevo. Lo mismo es cierto para cualquier marco secuencial. Hasta que Java 3D detecte que los volumenes no intereseccionan más no se podrá disparar de nuevo el **WakeupCondition**.

Hay varias parejas de clases **WakeupCriterion** correspondientes (Entry/Exit o Activation/Deactivation). Este criterio sólo se disparará en elternancias estrictas empezando con los criterios de Entry o Activation.

WakeupOnActivation

Es posible que una región de límites intersecione con el volumen de activación de un **ViewPlatform** tan brevemente que no sea detectada. Consecuentemente, no se disparará ninguna condición de Activation o Deactivation. Bajo estas circunstancias, el comportamiento no se activa nunca.

Sumario de Constructores de **WakeupOnActivation**

Extiende: WakeupCriterion

Esta clase especifica la condición de disparo en la primera detección de una interesección del volumen de activación de un **ViewPlatform** con la región límite de su objeto. **WakeupOnActivation** está emparejado con **WakeupOnDeactivation** que veremos más adelante.

WakeupOnActivation()
Construye un nuevo criterio **WakeupOnActivation**.

WakeupOnAWTEvent
Varias de las clases **WakeupCriterion** tienen constructores y métodos dependientes del disparo. Por ejemplo, **WakeupOnAWTEvent** tiene dos constructores y un método. Los constructores permiten la especificación de eventos AWT usando constantes de clases AWT. El método devuelve el array de eventos AWT consecutivos que causaron el disparo.

Sumario de Constructores de **WakeupOnAWTEvent**

Extiende: WakeupCriterion

Esta clase especifica el disparo de un **Behavior** cuando ocurre un evento AWT específico.

WakeupOnAWTEvent(int AWTId)
Construye un nuevo objeto **WakeupOnAWTEvent**, donde **AWTId** es uno de KeyEvent.KEY_TYPED, KeyEvent.KEY_PRESSED, KeyEvent.KEY_RELEASED, MouseEvent.MOUSE_CLICKED, MouseEvent.MOUSE_PRESSED, MouseEvent.MOUSE_RELEASED, MouseEvent.MOUSE_MOVED, MouseEvent.MOUSE_DRAGGED, o uno de los otros muchos valores de eventos.
WakeupOnAWTEvent(long eventMask)
Construye un nuevo objeto **WakeupOnAWTEvent** usando valores ORed EVENT_MASK. Estos valores son: KEY_EVENT_MASK, MOUSE_EVENT_MASK,

MOUSE_MOTION_EVENT_MASK, u otros valores.

Sumario de métodos de **WakeupOnAWTEvent**

AWTEvent[] getAWTEvent()

Recupera el array de eventos AWT consecutivos que ocasionaron el disparo.

WakeupOnBehaviorPost

La condición **WakeupOnBehaviorPost** junto con el método postID de la clase **Behavior** proporcionan un mecanismo a través del cual se pueden coordinar los comportamientos. Un objeto **Behavior** puede postear un valor entero ID particular. Otro comportamiento puede especificar su condición de disparo, usando un **WakeupOnBehaviorPost**, cómo enviando un ID particular desde un objeto **Behavior** específico. Esto permite la creacción de objetos **Behavior** parentales como que uno abra una puerta y otro diferente la cierre. Para esta materia, incluso se pueden formular comportamientos más complejos usando comportamientos y coordinación posterior.

Sumario de Constructores de **WakeupOnBehaviorPost**

Extiende: WakeupCriterion

Esta clase especifica un disparo de un objeto **Behavior** cuando un comportamiento específico postea un evento específico.

WakeupOnBehaviorPost(Behavior behavior, int postId)

Construye un nuevo criterio **WakeupOnBehaviorPost**.

Como un **WakeupCondition** puede estar compuesto por varios objetos **WakeupCriterion**, incluyendo más de un **WakeupOnBehaviorPost**, los métodos para determinar la especificidad son necesarios para interpretar un evento de disparo.

Sumario de Métodos de **WakeupOnBehaviorPost**

Behavior getBehavior()

Devuelve el comportamiento especificado en este constructor.

int getPostId()

Recupera el **postId** especificado en el **WakeupCriterion**.

Behavior getTriggeringBehavior()

Devuelve el comportamiento que disparo este evento.

int getTriggeringPostId()

Devuelve el **postId** que causó el disparo del comportamiento.

El Fragmento de Código 4-3 y el Fragmento de Código 4-4 muestran un código parcial para un programa de ejemplo que usa posteo de comportamientos para coordinar comportamientos. El ejemplo abre y cierra una puerta. El código incluye una clase: **OpenBehavior**, y el código que crea los dos objetos **behavior**. El segundo objeto es un ejemplar de **CloseBehavior**, que es casi un duplicado exacto de **OpenBehavior**. En **CloseBehavior**, la condición es compartida en el método initialization (y el comportamiento opuesto completado).

Fragmento de Código 4-3, clase OpenBehavior, y un ejemplo de clases de comportamiento coordinadas

```
1.    public class OpenBehavior extends Behavior{
2.
3.       private TransformGroup targetTG;
4.       private WakeupCriterion pairPostCondition;
5.       private WakeupCriterion AWTEventCondition;
6.
7.       OpenBehavior(TransformGroup targetTG){
8.          this.targetTG = targetTG;
9.          AWTEventCondition = new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED);
10.      }
11.
12.      public void setBehaviorObjectPartner(Behavior behaviorObject){
13.         pairPostCondition = new WakeupOnBehaviorPost(behaviorObject, 1);
14.      }
15.
16.      public void initialize(){
```

```
17.         this.wakeupOn(AWTEventCondition);
18.     }
19.
20.     public void processStimulus(Enumeration criteria){
21.         if (AWTEventCondition.hasTriggered()){
22.             // make door open – code excluded
23.             this.wakeupOn(pairPostCondition);
24.             postId(1);
25.         } else {
26.             this.wakeupOn(AWTEventCondition);
27.         }
28.     }
29.
30. } // end of class OpenBehavior
```

Fragmento de Código 4-4, código para usar las clases OpenBehavior y CloseBehavior

```
1.    // inside a method to assemble the scene graph ...
2.
3.        // create the relevant objects
4.        TransformGroup doorTG = new TransformGroup();
5.        OpenBehavior openObject = new OpenBehavior(doorTG);
6.        CloseBehavior closeObject = new CloseBehavior(doorTG);
7.
8.        //prepare the behavior objects
9.        openObject.setBehaviorObjectPartner(closeObject);
10.       closeObject.setBehaviorObjectPartner(openObject);
11.
12.       // set scheduling bounds for behavior objects – code excluded
13.
14.       // assemble scene graph – code excluded
15.
```

Los objetos de estas dos clases responderán en estricta alternancia a los eventos de pulsación de teclas. El comportamiento **OpenBehavior** se disparará en respuesta a la primera pulsación. En su respuesta, señala el comportamiento **CloseBehavior** y selecciona su condición de disparo para que sea una señal para este objeto. El objeto **CloseBehavior** selecciona su condición de disparo para que

sea una pulsación de tecla en respuesta a la señal desde el objeto **OpenBehavior**. Puedes encontrar un programa de ejemplo en DoorApp.java.
La siguiente pulsación de tecla dispara el objeto **CloseBehavior**. Este objeto ahora realiza la misma función que acaba de realizar el objeto **OpenBehavior**: envía una señal y resetea su propia condición de disparo. El objeto **CloseBehavior** cierra la puerta en respuesta a la pulsación de tecla. De vuelta a las condiciones iniciales, la siguiente pulsación empezará de nuevo todo el proceso.

WakeupOnCollisionEntry
Java 3D puede detectar la colisión de objetos en el mundo virtual. Hay tres clases **WakeupCriterion** útiles para procesar la colisión de objetos: **WakeupOnCollisionEntry**, **WakeupOnCollisionMovement**, y **WakeupOnCollisionExit**.
Un Criterio **WakeupOnCollisionEntry** se disparará cuando un objeto colisione por primera vez. Luego, el criterio **WakeupOnCollisionMovement** disparará (potencialmente varios disparos) mientras dos objetos están en colisión hay un movimiento relativo entre los objetos. Finalmente, un sólo **WakeupOnCollisionExit** se diparará cuando finalice la colisión.
Java 3D sólo puede manejar una colisión por cada objeto a la vez. Una vez que se ha detectado una colisión de un objeto, las colisiones con otros objetos no se detectarán hasta que finalice la primera colisión. También puede ocurrir que una colisión sea tan breve que no sea detectada y por lo tanto no se disparará ninguna condición.
La detección de colisiones es más compleja que esta discusión sobre las condiciones de disparo. Sin embargo este tutorial no cubre la detección de colisiones en detalle, para esto puedes referirte a la Especificación del API Java 3D.

Sumario de Constructores de **WakeupOnCollisionEntry**

Extiende: WakeupCriterion

Esta clase especifica un disparo en la primera detección de colisión de un objeto especificado con otro objeto en el escenario gráfico. También puedes ver: **WakeupOnCollisionMovement**, y **WakeupOnCollisionExit**.

WakeupOnCollisionEntry(Bounds armingBounds)

Construye un nuevo criterio WakeupOnCollisionEntry.

WakeupOnCollisionEntry(Node armingNode)

Construye un nuevo cirterio WakeupOnCollisionEntry.

WakeupOnCollisionEntry(Node armingNode, int speedHint)

Construye un nuevo criterio WakeupOnCollisionEntry, donde speedHint es:

- USE_BOUNDS - Usa límites geométircos como una aproximación al cálculo de colisiones.
- USE_GEOMETRY - Usa geometría en el cálculo de colisiones.

WakeupOnCollisionEntry(SceneGraphPath armingPath)

Construye un nuevo criterio WakeupOnCollisionEntry con USE_BOUNDS como velocidad de choque.

WakeupOnCollisionEntry(SceneGraphPath armingPath, int speedHint)

Construye un nuevo criterio WakeupOnCollisionEntry, donde speedHint es USE_BOUNDS o USE_GEOMETRY.

WakeupOnCollisionExit

Sumario de Constructores de **WakeupOnCollisionExit**

Extiende: WakeupCriterion

Esta clase especifica un disparo cuando se termina la colisión de un objeto especificado con otro objeto en el escenario gráfico. También puedes ver: **WakeupOnCollisionMovement**, y **WakeupOnCollisionExit**.

WakeupOnCollisionExit(Bounds armingBounds)

Construye un nuevo criterio WakeupOnCollisionExit.

WakeupOnCollisionExit(Node armingNode)

Construye un nuevo criterio WakeupOnCollisionExit.

WakeupOnCollisionExit(Node armingNode, int speedHint)

Construye un nuevo criterio WakeupOnCollisionExit, donde speedHint es:

- USE_BOUNDS - Usa límites geométircos como una aproximación al cálculo de colisiones.
- USE_GEOMETRY - Usa geometría en el cálculo de colisiones.

WakeupOnCollisionExit(SceneGraphPath armingPath)

Construye un nuevo criterio WakeupOnCollisionExit.

WakeupOnCollisionExit(SceneGraphPath armingPath, int speedHint)

Construye un nuevo criterio WakeupOnCollisionExit, donde speedHint es USE_BOUNDS, o USE_GEOMETRY.

Sumario de Métodos de **WakeupOnCollisionExit**

Bounds getArmingBounds()

Devuelve los límites del objeto usado en la especificación de la condición de colisión.

SceneGraphPath getArmingPath()

Devuelve el path usado en la especificación de la condición de colisión.

Bounds getTriggeringBounds()

Devuelve el objeto Bounds que causó la colisión.

SceneGraphPath getTriggeringPath()

Devuelve el path que describe el objeto que causó la colisión.

WakeupOnCollisionMovement

Sumario de Constructores de **WakeupOnCollisionMovement**

Extiende: WakeupCriterion

Esta clase especifica un disparo cuando el objeto especificado se mueve durante la colisión con otro objeto en el escenario gráfico. También puedes ver: **WakeupOnCollisionEntry**, y **WakeupOnCollisionExit**.

WakeupOnCollisionMovement(Bounds armingBounds)

Construye un nuevo criterio WakeupOnCollisionMovement.

WakeupOnCollisionMovement(Node armingNode)

Construye un nuevo criterio WakeupOnCollisionMovement.

WakeupOnCollisionMovement(Node armingNode, int speedHint)

Construye un nuevo criterio WakeupOnCollisionMovement, donde speedHint es:

- USE_BOUNDS - Usa límites geométircos como una aproximación al cálculo de colisiones.
- USE_GEOMETRY - Usa geometría en el cálculo de colisiones.

WakeupOnCollisionMovement(SceneGraphPath armingPath)

Construye un nuevo criterio WakeupOnCollisionMovement.

WakeupOnCollisionMovement(SceneGraphPath armingPath, int speedHint)

Construye un nuevo criterio WakeupOnCollisionMovement, donde speedHint es USE_BOUNDS, o USE_GEOMETRY.

Sumario de Métodos de **WakeupOnCollisionMovement**

Bounds getArmingBounds()

Devuelve el objeto **Bounds** usado para especificar la condición de colisión.

SceneGraphPath getArmingPath()

Devuelve el path usado en la especificación de la condición de colisión.

Bounds getTriggeringBounds()

Devuelve el objeto **Bounds** que causó la colisión.

SceneGraphPath getTriggeringPath()

Devuelve el path que describe el objeto que causó la colisión.

WakeupOnDeactivation

Sumario de Constructores de **WakeupOnDeactivation**

Extiende: WakeupCriterion

Esta clase especifica un disparo para la primera detección de que el volumen de activación deja de interseccionar con la región de límites de este objeto. También puedes ver **WakeupOnActivation**.

WakeupOnDeactivation()
Construye un nuevo criterio WakeupOnDeactivation.

WakeupOnElapsedFrames

El objeto **WakeupOnElapsedFrames** se usa para disparar un objeto activo después de que haya pasado un número especificado de frames. Un **frameCount** de 0 especifica que se dispare en el siguiente frame.

Sumario de Constructores de **WakeupOnElapsedFrames**

Extiende: WakeupCriterion

Esta clase especifica un disparo cuando han pasado un número especificado de frames.

WakeupOnElapsedFrames(int frameCount)
Construye un nuevo criterio WakeupOnElapsedFrames.

Sumario de Métodos de **WakeupOnElapsedFrames**

int getElapsedFrameCount()
Devuelve el contador de marcos **WakeupCriterion** que fue utilizado cuando se construyó este objeto.

WakeupOnElapsedTime

Java 3D no puede garantizar el tiempo exacto entre disparos para un criterio **WakeupOnElapsedTime**. Un disparo ocurrirá en el momento especificado, o muy cercano.

Sumario de Constructores de **WakeupOnElapsedTime**

Extiende: WakeupCriterion

Esta clase especifica un disparo después de que hayan pasado un número de milisegundos especificado.

WakeupOnElapsedTime(long milliseconds)
Construye un nuevo criterio WakeupOnElapsedTime.

Sumario de Métodos de **WakeupOnElapsedTime**

long getElapsedFrameTime()
Devuelve el valor de tiempo que se utilizó en la construcción de este objeto.

WakeupOnSensorEntry

En Java 3D, cualquier dispositivo de entrada distinto del teclado o el ratón es un **sensor**. Un sensor es un concepto abstracto para un dispositivo de entrada. Cada sensor tiene un punto caliente definido en el sistema de coordenadas del sensor. La intersección del punto caliente de un sensor con una región puede detectarse con las clases **WakeupOnSensorEntry** y **WakeupOnSensorExit**.
Es posible que un sensor entre y salga de una región armada tan rápidamente que ninguna de las condiciones se dispare.

Sumario de Constructores de **WakeupOnSensorEntry**

Extiende: WakeupCriterion

Esta clase especifica un disparo en la primera detección de la intersección de cualquier sensor con los límites especificados.

WakeupOnSensorEntry(Bounds region)
Construye un nuevo criterio WakeupOnEntry.

Sumario de Métodos de **WakeupOnSensorEntry**

Bounds getBounds()
Devuelve la especificación de límites de este objeto.

WakeupOnSensorExit

Sumario de Constructores de **WakeupOnSensorExit**

Extiende: WakeupCriterion

Esta clase especifica un disparo en la primera detección de que un sensor que previamente intereseccionaba con los límites deja de intereseccionar con los límites especificados. También puedes ver **WakeupOnSensorEntry**.

WakeupOnSensorExit(Bounds region)
Construye un nuevo criterio WakeupOnExit.

Sumario de Métodos de **WakeupOnSensorExit**

Bounds getBounds()
Devuelve la especificación de límites de este objeto.

WakeupOnTransformChange

El criterio **WakeupOnTransformChange** es útil para detectar cambios en la posición o la orientación de objetos visuales en el escenario gráfico. Este criterio ofrece un alternativa a usar el método postId para crear comportamientos coordinados. Es especialmente útil cuando el comportamiento con el cual se desea coordinar ya está escrito, por ejemplo las utilidades de comportamientos presentadas anteriormente.

Sumario de Constructores de **WakeupOnTransformChange**

Extiende: WakeupCriterion

Esta clase especifica un disparo cuando cambia la transformación dentro de un **TransformGroup** especificado.

WakeupOnTransformChange(TransformGroup node)
Construye un nuevo criterio WakeupOnTransformChange.

Sumario de Métodos de **WakeupOnTransformChange**

TransformGroup getTransformGroup()
Devuelve el nodo **TransformGroup** usado en la creacción de este WakeupCriterion

WakeupOnViewPlatformEntry

La detección de la intersección del **ViewPlatform** con una región especificada se hace posible con las clases del criterio de **WakeupOnViewPlatfomEntry** y de

WakeupOnViewPlatformExit.

Es posible que el límite especificado intersecione con un volumen de la activación de ViewPlatform tan brevemente que no sea detectada. En este caso ni se accionan las condiciones de **WakeupOnViewPlatformEntry** ni de

WakeupOnViewPlatformExit.

Sumario de Constructores de **WakeupOnViewPlatformEntry**

Extiende: WakeupCriterion

Esta clase especifica un disparo en la primera intersección del **ViewPlatform** con los límites especificados.

WakeupOnViewPlatformEntry(Bounds region)
Construye un nuevo criterio WakeupOnEntry.

Sumario de Métodos de **WakeupOnViewPlatformEntry**

Bounds getBounds()
Devuelve la especificación de límites de este objeto.

WakeupOnViewPlatformExit

Sumario de Constructores de **WakeupOnViewPlatformExit**

Extiende: WakeupCriterion

Esta Class especifica un disparo en la primera detección de un Viewplatform que deja de interseccionar con el límite especificado. También puedes ver WakeupOnViewPlatformEntry.

WakeupOnViewPlatformExit(Bounds region)
Construye un nuevo criterio WakeupOnExit.

Sumario de Métodos de **WakeupOnViewPlatformExit**

Bounds getBounds()
Devuelve la especificación de límites de este objeto

- **WakeupCondition Composition**

Varios objetos WakeupCriteron pueden componer un solo WakeupCondition usando las cuatro clases presentadas en esta sección. Las primeras dos clases permiten la composición de un WakeupCondition desde una colección de objetos WakeupCriterion que son lógicamente ANDed u ORed juntos, respectivamente. El tercero y siguientes permiten la composición de ejemplares de las dos primeras clases en objeto WakeupCondition más complejos.

WakeupAnd

Sumario de Constructores de **WakeupAnd**

Extiende: WakeupCondition

Esta clase especifica cualquier número de criterios de disparo que son (AND) juntos de forma lógica.

WakeupAnd(WakeupCriterion[] conditions)
Construye una nueva condición WakeupAnd.

WakeupOr

Sumario de Consctructores de **WakeupOr**

Extiende: WakeupCondition

Esta clase especifica cualquiere número de criterios de disparo que son (OR) juntos de forma lógica.

WakeupAnd(WakeupCriterion[] conditions)
Construye una nueva condición WakeupOr.

WakeupAndOfOrs

Sumario de Consctructores de **WakeupAndOfOrs**

Extiende: WakeupCondition

Esta clase especifica cualquier número de criterios de disparo WakeupOr que son (AND) juntos de forma lógica.

WakeupAndOfOrs(WakeupOr[] conditions)
Construye una nueva condición WakeupAndOfOrs.

WakeupOrOfAnds

Sumario de Consctructores de **WakeupOrOfAnds**

Extiende: WakeupCondition

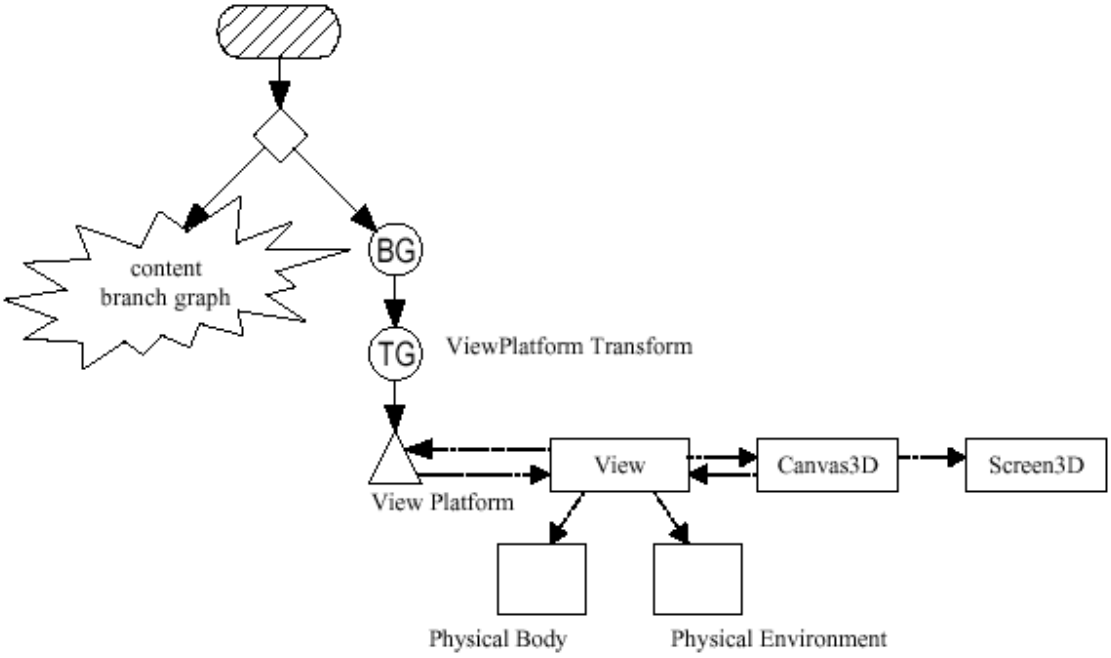
Esta clase especifica cualquier número de criterios de disparo WakeupAnd que son (OR) juntos de forma lógica.

WakeupOrsOfAnds(WakeupAnd[] conditions)
Construye una nueva condición WakeupOrOfAnds.

Clases de Comportamientos Útiles para la Navegación por Teclado

Hasta este momento, el espectador ha estado en una localización fija con una orientación fija. El poder mover el espectador es una capacidad importante en muchas aplicaciones de los gráficos 3D. Java 3D es capaz de mover el espectador. De echo hay clases utilitarias de Java 3D que implementan esta funcionalidad.
La Figura 4-8 muestra la rama gráfica básica para un universo virtual de Java 3D. En esta figura, se considera la plataforma de la visión **transform**. Si se cambia la transformación, el efecto es mover, o reorientar, o ambas, al espectador. De esto, podemos ver que el diseño básico de la navegación del teclado es simple:

hacemos que un objeto **behavior** cambie la transformación de la vista de la plataforma en respuesta a los movimientos dominantes.
Este diseño simple es exactamente el modo de trabajo de las clases utilitarias del teclado de Java 3D. Por supuesto podríamos construir nuestro propio comportamiento de navegación del teclado. El resto de esta sección explica cómo utilizar las clases de la navegación del teclado de Java 3D.



Cómo Navegar en un SimpleUniverse

Podría ser que pensaramos que necesitar el acceso a los grupos de objeto **Transform** de la plataforma significa abandonar la utilidad **SimpleUniverse**. Sin embargo, **SimpleUniverse**, y las clases relacionadas, proporcionan una combinación de métodos para extraer el objeto **ViewPlatformTransform**. Por lo tanto, podemos tener nuestro **SimpleUniverse** y navegar en él también!
Específicamente, la siguiente línea de código extrae el **ViewPlatformTransform** de un objeto de SimpleUniverse, **su**.

```
TransformGroup vpt = su.getViewingPlatform().getViewPlatformTransform();
```

Programa de Ejemplo de KeyNavigatorBehavior

Es fácil utilizar la clase de utilidad **KeyNavigatorBehavior** en un programa de Java 3D. Esta sección demuestra el uso de la clase en el programa del ejemplo de KeyNavigatorApp.java. En este programa podemos ver que los pasos necesarios para usar la clase **KeyNavigatorBehavior** son esencialmente idénticos a los de usar cualquier clase de comportamiento. Los pasos para usar **KeyNavigatorBehavior** se resumen en la siguiente lista.

1. crear un objeto **KeyNavigatorBehavior**, seleccionado el grupo de transformación
2. añadir el objeto **KeyNavigatorBehavior** al escenario gráfico
3. proporcionar unos límites (o BoundingLeaf) para el objeto **KeyNavigatorBehavior**

Como cualquier problema de programación, hay una variedad de maneras de implementar los pasos de esta receta. Un acercamiento es incorporar estos pasos en el método de createSceneGraph. El Fragmento de Código 4-5 muestra los pasos de la receta según la implementación para el programa del ejemplo de KeyNavigatorApp.java.

Fragmento de Código 4-5, usar la clase KeyNavigatorBehavior (parte 1)

```
1.    public BranchGroup createSceneGraph(SimpleUniverse su) {
2.        // Create the root of the branch graph
3.        TransformGroup vpTrans = null;
4.
5.        BranchGroup objRoot = new BranchGroup();
6.
7.        objRoot.addChild(createLand());
8.
9.       // create other scene graph content
10.
11.
12.      vpTrans = su.getViewingPlatform().getViewPlatformTransform();
13.      translate.set( 0.0f, 0.3f, 0.0f); // 3 meter elevation
14.      T3D.setTranslation(translate); // set as translation
15.      vpTrans.setTransform(T3D); // used for initial position
16.      KeyNavigatorBehavior keyNavBeh = new KeyNavigatorBehavior(vpTrans);
```

```
17.     keyNavBeh.setSchedulingBounds(new BoundingSphere(
18.                                   new Point3d(),1000.0));
19.     objRoot.addChild(keyNavBeh);
20.
21.     // Let Java 3D perform optimizations on this scene graph.
22.     objRoot.compile();
23.
24.     return objRoot;
25.   } // end of CreateSceneGraph method of KeyNavigatorApp
```

La ejecución del paso 1 de la receta en el método de createSceneGraph requiere el acceso al grupo de transformación de **ViewPlatform**. Esta implementación pasa el objeto **SimpleUniverse** (línea 34 del Fragmento de Código 4-6) al método createSceneGraph que lo hace disponible para tener acceso a la transformación de **ViewPlatform** (la línea 12 del Fragmento de Código 4-5).

Pasar el objeto **SimpleUniverse** al método de createSceneGraph permite acceder a otras características de la rama gráfica de la vista de **SimpleUniverse**, tales como **PlatformGeometry**, **ViewerAvatar**, o de agregar un **BoundingLeaf** a la rama gráfica de la vista.

La líneas 13 a 15 del Fragmento de Código 4-5 proporcionan una posición inicial para el espectador. En este caso, el espectador se mueve a una posición 0,3 metros sobre el origen del mundo virtual. Esto es solamente una posición inicial, y de ninguna manera limita la posición futura o la orientación del espectador.

Fragmento de Código 4-6, usar la clase KeyNavigatorBehavior (parte 2)

```
26.   public KeyNavigatorApp() {
27.     setLayout(new BorderLayout());
28.     Canvas3D canvas3D = new Canvas3D(null);
29.     add("Center", canvas3D);
30.
31.     // SimpleUniverse is a Convenience Utility class
32.     SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
33.
34.     BranchGroup scene = createSceneGraph(simpleU);
```

```
35.
36.      simpleU.addBranchGraph(scene);
37.   } // end of KeyNavigatorApp (constructor)
```

Cómo Crear una Aplicación Universal de un Comportamiento

Como con cualquier objeto comportamiento, el objeto **KeyNavigtorBehavior** está solo activo cuando sus límites interseccionan con el volumen de activación de un **ViewPlatform**. Esto puede estar particularmente limitado para un comportamiento de navegación, donde el comportamiento debe siempre estar activado. El Capítulo 3 discute una solución a este problema usando un **BoundingLeaf**.

- **Clases KeyNavigatorBehavior y KeyNavigator**

La utilidad de navegación del teclado se implementa como dos clases. En el tiempo de ejecución hay dos objetos. El primer objeto es el objeto **KeyNavigatorBehavior**, el segundo es un objeto **KeyNavigator**. La segunda clase no se documenta aquí ya que ni el programador ni el cliente deben saber que existe la segunda clase u objeto.

El objeto **KeyNavigatorBehavior** realiza todas las funciones típicas de una clase de comportamiento, excepto que llama al objeto **KeyNavigator** para realizar la función de processStimulus. La clase **KeyNavigator** toma el **AWTEvent** y lo procesa bajo al nivel de pulsaciones de teclas individuales. La Tabla siguiente muestra el efecto de las pulsaciones de teclas individuales. **KeyNavigator** implementa el movimiento con aceleración.

Movimientos de **KeyNavigatorBehavior**

Tecla	Movimiento	Movimiento Alt-tecla
<-	rotar a la izquierda	traslación lateral izquierda
->	rotar a la derecha	traslación lateral derecha
^	mover hacia adelante	
v	mover hacia atrás	
PgUp	rotar arriba	translación hacia arriba
PgDn	rotar abajo	translación hacia abajo

+ aumenta la distancia de salto (y vuelve al origen)

- reduce la distancia de salto

= vuelve al centro del universo

Sumario de Constructores de **KeyNavigatorBehavior**

Paquete: com.sun.j3d.utils.behaviors.keyboard

Extiende: Behavior

Esta clase es un sencillo comportamiento que invoca el **KeyNavigator** para modificar la transformación de la vista de la plataforma.

KeyNavigatorBehavior(TransformGroup targetTG)
Construye un nuevo comportamiento de navegación por teclado que opera sobre el grupo de transformación especificado.

Sumario de Métodos de **KeyNavigatorBehavior**

void initialize()
Sobreescribe el método initialize de **Behavior** para configurar los criterios de disparo.
void processStimulus(java.util.Enumeration criteria)
Sobreescribe el método stimulus de **Behavior** para manejar el evento.

Clases de Utilidad para Interactuar con el Ratón

El paquete de comportamientos de ratón (com.sun.j3d.utils.behaviors.mouse) contiene las clases del comportamiento en las cuales el ratón se utiliza como entrada de información para la interacción con los objetos visuales. Incluyendo las clases para traslaciones (moviéndose en un plano paralelo a la placa de la imagen), enfocando (que mueve hacia atrás y adelante), y los objetos visuales que rotan en respuesta a los movimientos del ratón.

La siguiente tabla resume las tres clases específicas del comportamiento del ratón incluidas en el paquete. Además de estas tres clases, está la clase abstracta **MouseBehavior**, y el interface **MouseCallback**. Esta clase abstracta y el interface se utilizan en la creación de las clases específicas del comportamiento del ratón y son útiles para crear comportamientos personalizados del ratón.

Sumario de las clases espécificas de **MouseBehavior**

Clase MouseBehavior	Acción en Respuesta a la Acción del Ratón	Acción del ratón
MouseRotate	rota el objeto visual sin moverlo	botón izquierdo pulsado con movimiento del ratón
MouseTranslate	translada el objeto visual en un plano paralelo al plato de imagen	boton derecho pulsado con movimiento del ratón
MouseZoom	translada el objeto visual en un plano orthogonal al plato de imagen	botón central pulsado con movimiento del ratón

Usar las Clases de Comportamiento del Ratón

La clases de comportamientos específicos del ratón son fáciles de usar; es esencialmente lo mismo que el de otras clases de comportamientos. La siguiente lista representa la receta para usarlas:

1. proporcionar capacidades de lectura y escritura para el **transformGroup** fuente
2. crear uno bjeto MouseBehavior
3. seleccionar el **transformGroup** fuente
4. proporcionar unos límites (o BoundingLeaf) para el objeto MouseBehavior
5. añadir el objeto **MouseBehavior** al escenario gráfico

Como con algunas otras recetas, los pasos no tienen que ser realizados en el orden dado. El paso dos se debe realizar antes del tres, del cuatro, y del cinco; los otros pasos se pueden realizar en cualquier orden. También, los pasos dos y tres se pueden combinar usando un constructor diferente.

El Fragmento de Código 4-7 presenta el método createSceneGraph del programa del ejemplo de MouseRotateApp.java. El escenario gráfico incluye el objeto **ColorCube**. El usuario puede rotar el **ColorCube** usando el ratón debido a la inclusión de un objeto **MouseRotate** en el escenario gráfico.

Fragmento de Código 4-7, usar la clase de utilidad MouseRotate

```
1.    public class MouseRotateApp extends Applet {
2.
3.        public BranchGroup createSceneGraph() {
4.            // Create the root of the branch graph
5.            BranchGroup objRoot = new BranchGroup();
6.
7.            TransformGroup objRotate = new TransformGroup();
8.            objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
9.            objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
10.
11.           objRoot.addChild(objRotate);
12.           objRotate.addChild(new ColorCube(0.4));
13.
14.           MouseRotate myMouseRotate = new MouseRotate();
15.           myMouseRotate.setTransformGroup(objRotate);
16.           myMouseRotate.setSchedulingBounds(new BoundingSphere());
17.           objRoot.addChild(myMouseRotate);
18.
19.           // Let Java 3D perform optimizations on this scene graph.
20.           objRoot.compile();
21.
22.           return objRoot;
23.   } // end of CreateSceneGraph method of MouseRotateApp
```

La misma receta funcionará para las otras clases de comportamiento del ratón. De hecho los tres comportamientos se pueden utilizar en la misma aplicación que funciona en el mismo objeto visual. Puesto que cada uno de los comportamientos del ratón lee el **transform** fuente antes de escribirlo, sólo se necesita un objeto **TransformGroup** incluso con tres comportamientos de ratón. El programa del ejemplo de MouseRotateApp.java hace apenas eso.

El siguiente ejemplo muestra cómo dos comportamientos del ratón trabajan en un solo mundo virtual. El programa MouseRotate2App.java del ejemplo crea un escenario gráfico con dos objetos **ColorCube** uno junto al otro en el mundo virtual. Cada uno de los **ColorCubes** tiene un objeto **MouseRotate** asociado a él. Puesto

que ambos objetos de comportamiento del ratón están activos, cuando el usuario hace clic y mueve el ratón, ambos **ColorCubes** rotan.
Si no quisieramos que ambos objetos rotaran, hay dos soluciones: 1) cambiar la posición del espectador, o cambiar los límites del comportamiento, de modo que solamente un comportamiento sesté activo, o 2) usar un mecanismo de selección para aislar el comportamiento.

Fundamentos del Comportamiento del Ratón

Las clases específicas de comportamiento del ratón (**MouseRotate**, **MouseTranslate**, y **MouseZoom**) son extensiones de la clase abstracta **MouseBehavior** e implementan el interface **MouseCallback**.

La Clase Abstracta MouseBehavior

Esta clase abstracta se presenta aquí en el evento que deseamos ampliarlo para escribir una clase personaliza de comportamiento del ratón. El método SetTransformGroup() es probablemente el único que utilizarán los usuarios de un ejemplar de **MouseBehavior**. Los otros métodos se crearón para los autores de las clases de comportamientos personalizados del ratón.

Sumario de Métodos de **MouseBehavior**

La clase base para todos los manipuladores de ratón (puedes ver **MouseRotate** y **MouseZoom** para ejemplos).

void initialize()

Inicializa el comportamiento.

void processMouseEvent(java.awt.event.MouseEvent evt)

Maneja eventos del ratón.

void processStimulus(java.util.Enumeration criteria)

Todos los manipuladores de ratón deben implementar este método de **Behavior** (para responder a los estimulos).

void setTransformGroup(TransformGroup transformGroup)

Selecciona el TransformGroup para el comportamiento.

void wakeup()

Dispara manualmente el comportamiento.

Interface MouseCallback

Una clase que implementa este interfaz proporciona al método transformChanged que será llamado cuando cambie el **transform** fuente de la manera especificada. Cada uno de los tres comportamientos específicos del ratón implementa esta clase. Un programador simplemente puede reemplazar el método transformChanged de una de esas clases para especificar un método que se llamará cuando se modifique el **transform**.

Sumario de Métodos del **Interface MouseBehaviorCallback**

Paquete: com.sun.j3d.utils.behaviors.mouse

void transformChanged(int type, Transform3D transform)

Las clases que implementan este interface que se registran con un **MouseBehaviors** serán llamadas cada vez que el comportamient actualice el **Transform**. El tipo es uno de **MouseCallback.ROTATE**, **MouseCallback.TRANSLATE**, o **MouseCallback.ZOOM**.

Clases Específicas de Comportamientos de Ratón

MouseRotate

Un escenario gráfico que incluye un objeto **MouseRotate** permite que el usuario rote objetos visuales en el mundo virtual. Los programas de ejemplo **MouseRotateApp**, **MouseRotate2App**, y **MouseBehaviorApp** demuestran el uso de esta clase.

Sumario de Constructores de **MouseRotate**

Paquete: com.sun.j3d.utils.behaviors.mouse

Extiende: MouseBehavior

MouseRotate es un objeto de comportamiento de Java3D que deja a los usuarios controlar la rotación de un objeto mediante una pulsación del botón izquierdo del ratón. Para utilizar esta utilidad, primero creamos un **TransformGroup** sobre el que operará este comportamiento. El usuario puede rotar cualquier objeto hijo del **TransformGroup** fuente.

MouseRotate()

Crea un comportamiento mouseRotate por defecto.

MouseRotate(TransformGroup transformGroup)

Crea un comportamiendo dando el transformgroup.

MouseRotate(int flags)

Crea un comportamiento con las banderas seleccionadas, donde las banderas son:

- MouseBehavior.INVERT_INPUT. Invierte la entradas.
- MouseBehavior.MANUAL_WAKEUP. Dispara manualmente el comportamiento.

Sumario de Métodos de **MouseRotate**

void setFactor(double factor)

Selecciona el factor multiplicador para los movimientos en los ejes x e y al valor **factor**

void setFactor(double xFactor, double yFactor)

Selecciona el factor multiplicador para los movimientos en los ejes x e y a los valores **xFactor** e **yFactor** respectivamente.

void setupCallback(MouseBehaviorCallback callback)

El método que se llama cada vez que se actualiza el transformgroup

void transformChanged(Transform3D transform)

Los usuarios pueden sobreescribir este método que es llamado cada vez que el comportamiento actualiza el transformgroup. La implementación por defecto no hace nada.

MouseTranslate

Un escenario gráfico que incluye un objeto **MouseTranslate** permite que el usuario mueva objetos visuales en un plano paralelo a la placa de la imagen en el mundo virtual.

Sumario de Constructores de **MouseTranslate**

Paquete: com.sun.j3d.utils.behaviors.mouse

Extiende: MouseBehavior

MouseTranslate es un objeto comportamiento de Java3D que permite a los usuarios controlar la traslación (X,Y) de un objeto mediante un movimiento de arrastre del ratón con el botón derecho.

MouseTranslate()

Crea un comportamiento de movimiento por defecto.

MouseTranslate(TransformGroup transformGroup)

Crea un comportamiento de movimiento dando un transformgroup.

MouseTranslate(int flags)

Crea un comportamiento de movimiento con banderas, donde las banderas son:

- MouseBehavior.INVERT_INPUT. Invierte la entradas.
- MouseBehavior.MANUAL_WAKEUP. Dispara manualmente el comportamiento.

Sumario de Métodos de **MouseTranslate**

void setFactor(double factor)

Selecciona el factor multiplicador para los movimientos en los ejes x e y al valor **factor**

void setFactor(double xFactor, double yFactor)

Selecciona el factor multiplicador para los movimientos en los ejes x e y a los valores **XFactor** e **yFactor** respectivamente.

void setupCallback(MouseBehaviorCallback callback)

El método que se llama cada vez que se actualiza el transformgroup

void transformChanged(Transform3D transform)

Los usuarios pueden sobreescribir este método que es llamado cada vez que el comportamiento actualiza el transformgroup. La implementación por defecto no hace nada.

MouseZoom

Un escenario gráfico que incluye un objeto **MouseZoom** permite a los usuarios mover objetos visuales en un plano orthogonal al plato de imagen en un mundo virtual.

Sumario de Constructores de **MouseZoom**

Paquete: com.sun.j3d.utils.behaviors.mouse

Extiende: MouseBehavior

MouseZoom es un objeto de comportamiento Java3D que permite a los usuarios controlar la traslación en el eje z de un objeto mediante un movimiento de arrastre del ratón con el botón central (alt-tecla en el PC con el ratón de dos botones).

MouseZoom()

Crea un comportamiento de zoom con ratón por defecto.

MouseZoom(TransformGroup transformGroup)

Crea un comportamiento de zoom dando el transformgroup.

MouseZoom(int flags)

Crea un comportamiento de zoom con banderas, donde las banderas son:

- MouseBehavior.INVERT_INPUT. Invierte la entradas.
- MouseBehavior.MANUAL_WAKEUP. Dispara manualmente el comportamiento.

Sumario de Métodos de **MouseZoom**

void setFactor(double factor)

Selecciona el factor multiplicador del movimiento sobre el eje z al valor **factor**.

void setupCallback(MouseBehaviorCallback callback)

El método que se llama cada vez que se actualiza el transformgroup

void transformChanged(Transform3D transform)

Los usuarios pueden sobreescribir este método que es llamado cada vez que el comportamiento actualiza el transformgroup. La implementación por defecto no hace nada.

- **MouseNavigation**

Las tres clases específicas de comportamiento del ratón se pueden utilizar para crear un universo virtual en el cual el ratón se utilice para la navegación. Cada una de las clases específicas del comportamiento del ratón tiene un constructor que toma un solo parámetro entero para las banderas. Cuando se utiliza **MouseBehavior.INVERT_INPUTS** como argumento a este constructor, el comportamiento del ratón responde en la dirección opuesta. Este comportamiento inverso es apropiado para cambiar el **transform** ViewPlatform. Es decir las clases del comportamiento del ratón se pueden utilizar para el control navegacional.

El programa de ejemplo MouseNavigatorApp.java utiliza casos de las tres clases específicas del comportamiento del ratón para la interacción navegacional. El Fragmento de Código 4-8 muestra el método createSceneGraph de este programa del ejemplo.

El **TransformGroup** fuente para cada uno de los objetos del comportamiento del ratón es el ViewPlatform transform. El objeto **SimpleUniverse** es un argumento al método createSceneGraph de modo que se puedan alcanzar los objetos **transform** de ViewPlatform.

Fragmento de Código 4-8 ,Usar clases de Comportamientos del Ratón para Navegación Interactiva en un Mundo Virtual.

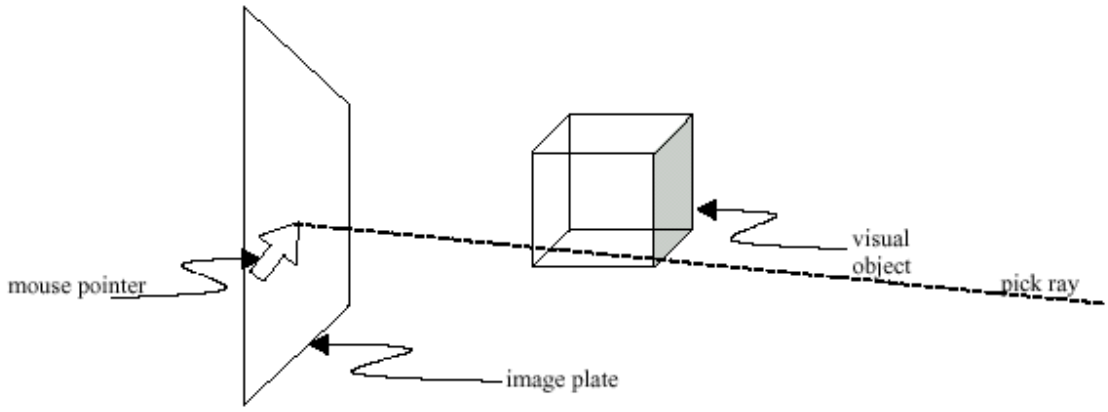
```
1.  public BranchGroup createSceneGraph(SimpleUniverse su) {
2.      // Create the root of the branch graph
3.      BranchGroup objRoot = new BranchGroup();
```

```
4.      TransformGroup vpTrans = null;
5.      BoundingSphere mouseBounds = null;
6.
7.      vpTrans = su.getViewingPlatform().getViewPlatformTransform();
8.
9.      objRoot.addChild(new ColorCube(0.4));
10.     objRoot.addChild(new Axis());
11.
12.     mouseBounds = new BoundingSphere(new Point3d(), 1000.0);
13.
14.     MouseRotate myMouseRotate = new
                        MouseRotate(MouseBehavior.INVERT_INPUT);
15.     myMouseRotate.setTransformGroup(vpTrans);
16.     myMouseRotate.setSchedulingBounds(mouseBounds);
17.     objRoot.addChild(myMouseRotate);
18.
19.     MouseTranslate myMouseTranslate = new
                        MouseTranslate(MouseBehavior.INVERT_INPUT);
20.     myMouseTranslate.setTransformGroup(vpTrans);
21.     myMouseTranslate.setSchedulingBounds(mouseBounds);
22.     objRoot.addChild(myMouseTranslate);
23.
24.     MouseZoom myMouseZoom = new
                        MouseZoom(MouseBehavior.INVERT_INPUT);
25.     myMouseZoom.setTransformGroup(vpTrans);
26.     myMouseZoom.setSchedulingBounds(mouseBounds);
27.     objRoot.addChild(myMouseZoom);
28.
29.     // Let Java 3D perform optimizations on this scene graph.
30.     objRoot.compile();
31.
32.     return objRoot;
33.  } // end of createSceneGraph method of MouseNavigatorApp
```

Los objetos **bounds** para los objetos de comportamientos de ratón se especifican como un **BoundingSphere** con un radio de 1000 metros. Si el usuario se sale de esta esfera, los objetos comportamiento se desactivarán.

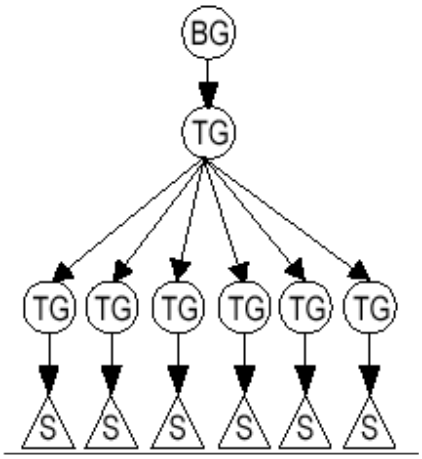
Picking

En el programa de ejemplo MouseNavigatorApp.java, ambos objetos **ColorCube** giran en respuesta a acciones del usuario. En esta aplicación, no hay forma de manipular los cubos de forma separada. "Picking" (Elección) le da al usuario una forma de interactuar con objetos visuales individuales en la escena.
Picking (elección) está implementado por un comportamiento tipicamente disparado por eventos de botones del ratón. En la selección de un objeto visual, el usuario sitúa el puntero del ratón sobre el objeto elegido y pulsa el botón del ratón. El objeto **behavior** se dispara por la pulsación de este botón y empieza la operación de selección. Se proyecta un rayo dentro del mundo virtual desde la posición del puntero del ratón paralela con la proyección. Se calcula la intersección de este rayo con los objetos del mundo virtual. El objeto visual que interseccione más cerca al plato de la imagen se selecciona para interacción. La Figura 4-11 muestra un rayo de selección proyectado en un mundo virtual.



En algunos casos la interacción no se hace directamente con el objeto seleccionado, sino con un objeto en el camino del escenario gráfico hasta el objeto. Por ejemplo, al seleccionar un objeto **ColorCube** para rotación, este objeto no se manipula, se manipula el objeto **TransformGroup** que hay sobre el **ColorCube** en el path del escenario gráfico. Por otro lado, su la operación de selección selecciona un objeto visual para el que se pensó un cambio de color, entonces el objeto visual seleccionado es requerido.

La determinación del objeto para un posterior procesamiento no siempre es sencilla. Si in objeto visual cúbico que va a ser rotado está compuesto por seis objetos **Shape3D** individuales junto con seis objetos **TransformGroup**, como en el escenario gráfico de la Figura 4-12, no es el objeto **TransformGroup** sobre el objeto **Shape3D** interseccionado el que necesita ser modificado. El 'cubo' se rota por la manipulación del objeto **TransformGroup** que es hijo del objeto **BranchGroup** en el escenario gráfico. Por esta razón, el resultado de algunas operaciones de selección es devolver el path del escenario gráfico para su posterior procesamiento.



La comprobación de intersecciones es necesita mucho cálculo. Por lo tanto, la selección es cara y se vuelve más cara con la complejidad del escena. El API Java 3D proporciona varias formas para que un programador pueda limitar la cantidad de cálculos realizados por la selección. Una forma importante es através de las capacidades y atributos de los nodos del escenario gráfico. Si un nodo es o no elegible se selecciona con el método setPickable() de la clase. Un nodo con setPickable() seleccionado a false no es elegible ni ninguno de sus hijos tampoco. Consecuentemente, estos nodos no se tienen en cuenta cuando se calculan las intersecciones.

Otra característica relacionada con la selección en la clase **Node** es la capacidad ENABLE_PICK_REPORTING. Esta capacidad sólo se aplica a nodos **Group**. Cuando se selecciona para un grupo, este objeto **group** siempre será incluido en el escenario gráfico devuelto por una operación de selección. Los nodos **Group** no se

necesitan para unidades en un escenario gráfico que serán excluidas cuando la capacidad no está seleccionada. No tener seleccionado correctamente los nodos del escenario gráfico es un fuente común de frustacciones en el desarrollo de aplicaciones que utilizan operaciones de selección.

Lista Parcial de Métodos de **Node**

Extiende: SceneGraphObject

Subclases: Group, Leaf

La clase **Node** proporciona una clase abstracta para todos los nodos **Group** y **Leaf**. Porporciona un marco de trabajo común para construir un escenario gráfico Java 3D, especificamente volúmenes, y las capacidades de selección y colisión.

void setBounds(Bounds bounds)
Selecciona los límites geométricos de un nodo.
void setBoundsAutoCompute(boolean autoCompute)
Activa/desactiva el cálculo automático de los límites geométricos de un nodo.
setPickable(boolean pickable)
Cuando se selecciona a true este nodo puede ser elegido. Cuando se selecciona a false indica que este nodo y sus hijos no son elegibles.

Lista Parcial de Capacidades de **Node**

ENABLE_PICK_REPORTING

Especifica que este nodo será reportado en el **SceneGraphPath** si ocurre una selección. Esta capacidad es sólo aplicable para nodos **Group**; es ignorado para nodos **leaf**. El valor por defecto para nodos **Group** es false. Los nodos interiores no necesitan ser únicos en un **SceneGraphPath** que no tiene seleccionado ENABLE_PICK_REPORTING serán excluidos del **SceneGraphPath**.

ALLOW_BOUNDS_READ | **WRITE**

Especifica que este nodo permite leer (escribir) la información de sus límites.

ALLOW_PICKABLE_READ | WRITE

Especifica que este nodo permite leer (escribir) su estado de selección.

Otra forma en la que un programador puede reducir el cáculo de selección es usar pruebas de interesección de límites en vez de pruebas de intersecciones geométricas. Varias clases relacionadas con la selección (pick) tiene constructores y/o métodos con un parámetro que se selecciona a uno de: **USE_BOUNDS** o **USE_GEOMETRY**. Cuando se selecciona **USE_BOUNDS**, la selección está determinada usando los límites de los objetos visuales, no la geometría real. La determinación de una selección usando los límites es significativamente más sencilla (computacionalmente) para todo excepto para las formas geométricas sencillas y por lo tanto, resulta en un mejor rendimiento. Por supuesto, la pérdida es que la selección no es tan precisa cuando se utilizan límites para su determinación.

Una tercera técnica de programación para reducir el coste de cálculo para la selección es limitar el ámbito de la prueba de selección a la porcion relevante del escenario gráfico. En cada clase de utilidad de selección se selecciona un nodo como el raíz para el gráfico a testear. Este nodo no es necesariamente el raíz de la rama de contenido gráfico. Por el contrario, el nodo pasado debería ser el raíz de la subrama de contenido que sólo contiene objetos elegibles, si es posible. Esta consideración pordría ser una mayor factor de determinación en la construcción de un escenario gráfico para algunas aplicaciones.

- **Usar las Clases de Utilidad de Picking**

Hay dos aproximaciones básicas para usar las características de selección de Java 3D, usar objetos de clases **picking**, o crear clases **picking** personalizadas y usar ejemplares de estas clases. El paquete **picking** incluye clases para pick/rotate, pick/translate, y pick/zoom. Es decir, un usuario puede elegir y rotar un objeto presionando el botón del ratón cuando el puntero está sobre el objeto

deseado y entonces arrastra el ratón (mientras mantiene pulsado el botón). Cada una de estas clases de **picking** usa un botón diferente del ratón haciendo posible el uso de objeto para las tres clases de **picking** en la misma aplicación simultáneamente.

Cómo un objeto comportamiento **picking** operará sobre cualquier objeto del escenario gráfico (con las capacidades apropiadas), sólo se necesita proporcionar un objeto **picking**. Las dos siguientes líneas de código son todo lo que necesitamos incluir en un programa Java 3D para usar las clases de selección:

```
PickRotateBehavior behavior = new PickRotateBehavior(root, canvas, bounds);
root.addChild(behavior);
```

El objeto behavior monitorizará cualquier evento de selección en el escenario gráfico (bajo el nodo raíz) y meneja los arrastres y pulsaciones del ratón. El root proporciona la porción del escenario gráfico a chequear para la selección, el canvas se situa donde está el ratón, y bounds son los límites del objeto de comportamiento **picking**.

Receta para usar las clases de utilidades de **picking**.

1. Crear nuestro escenario gráfico.
2. Crear un objeto behavior **picking** con la especificación de root, canvas, y bounds.
3. Añadir el objeto **behavior** al escenario gráfico.
4. Activar las capacidades apropiadas para los objetos del escenario gráfico.

Riesgos de Programación cuando se usan Objetos Picking

Los Riesgos más comunes incluyen; olvidarse de incluir el objeto **behavior** en el escenario gráfico, y no seleccionar los límites apropiados del objeto.

Otro problema común es no seleccionar las capacidades apropiadas para los objetos del escenario gráfico. Hay otros dos problemas menores, que deberíamos chequear si nuestra aplicación no funciona. Uno es no seleccionar apropiadamente el raíz del escenario gráfico. Otro problema potencial es no seleccionar apropiadamente el canvas. Ninguno de estos errores de programación generarán un aviso o mensaje de error.

Programa de Ejemplo MousePickApp

El Fragmento de Código 4-9 muestra el método createSceneGraph de la aplicación MousePickApp.java. Este programa usa un objeto **PickRotate** para proporcionar interacción.

Observa que como la construcción del objeto **picking** requiere un objeto **Canvas3D**, el método createSceneGraph difiere de versiones anteriores por la inclusión del parámetro canvas. Por supuesto, también cambia la correspondiente invocación a createSceneGraph.

Fragmento de Código 4-9, Método createSceneGraph de la aplicación MousePickApp.

```
1.  public BranchGroup createSceneGraph(Canvas3D canvas) {
2.      // Create the root of the branch graph
3.      BranchGroup objRoot = new BranchGroup();
4.
5.      TransformGroup objRotate = null;
6.      PickRotateBehavior pickRotate = null;
7.      Transform3D transform = new Transform3D();
8.      BoundingSphere behaveBounds = new BoundingSphere();
9.
10.     // create ColorCube and PickRotateBehavior objects
11.     transform.setTranslation(new Vector3f(-0.6f, 0.0f, -0.6f));
12.     objRotate = new TransformGroup(transform);
13.     objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
14.     objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
15.     objRotate.setCapability(TransformGroup.ENABLE_PICK_REPORTING);
16.
17.     objRoot.addChild(objRotate);
18.     objRotate.addChild(new ColorCube(0.4));
19.
20.     pickRotate = new PickRotateBehavior(objRoot,canvas, behaveBounds);
21.     objRoot.addChild(pickRotate);
22.
23.     // add a second ColorCube object to the scene graph
24.     transform.setTranslation(new Vector3f( 0.6f, 0.0f, -0.6f));
25.     objRotate = new TransformGroup(transform);
26.     objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
```

```
27.     objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
28.     objRotate.setCapability(TransformGroup.ENABLE_PICK_REPORTING);
29.
30.     objRoot.addChild(objRotate);
31.     objRotate.addChild(new ColorCube(0.4));
32.
33.     // Let Java 3D perform optimizations on this scene graph.
34.     objRoot.compile();
35.
36.     return objRoot;
37.   } // end of createSceneGraph method of MousePickApp
```

Este código es similar al de MouseRotate2App.java, pero es distinto en muchas cosas. Primero, en este programa sólo se usa un objeto **behavior**, mientras que MouseRotate2App usaba dos objetos **behavior** - uno por cada objeto visual. Aunque el código es similar, el comportamiento es diferente. Este programa permite al usuario seleccionar un objeto e interactuar con él. MouseRotate2App rotaba los dos objetos o ninguno.

El API Corazón de Clases Picking de Java 3D

Hay tres niveles de clases **picking** en Java 3D. El API corazón de Java 3D proporciona la menor funcionalidad. El paquete de utilidad **picking** proporciona clases de comportamientos generales, elegibles para personalización. El paquete **picking** también proporciona clases **picking** específicas que pueden usarse directamente en programas Java 3D.

Las clases corazón incluyen **PickShape** y **SceneGraphPath**, y métodos de **BranchGroup** y **Locale**. Estas clases proporcionan el mecanismo para especificar una forma usada en la comprobación de intersecciones con objetos visuales. Esta sección presenta el API de las clases **PickShape** y **SceneGraphPath**, y las clases y métodos relacionados.

Clases PickShape

Esta clase abstracta no proprociona ni constructores ni métodos. Proporciona abstracción para cuatro subclases: **PickBounds**, **PickRay**, **PickSegment**, y

PickPoint.

PickShape

Subclases Conocidas: PickBounds, PickRay, PickSegment, PickPoint

Una clase general para describir un forma de selección que puede usarse con métodos de selección de **BranchGroup** y **Locale**.

PickBounds

Los objetos **PickBounds** representan un límite para testear elecciones. Como una subclase de **PickShape**, los objetos **PickBounds** se usan con **BranchGroup** y **Locale** así como con clases del paquete **picking**.

Sumario de Constructores de **PickBounds**

Extiende: PickShape

Un límite para suministrar a los métodos de selección de **BranchGroup** y **Locale**.

PickBounds()

Crea un PickBounds.

PickBounds(Bounds boundsObject)

Crea un PickBounds con los límites especificados.

Sumario de Métodos de **PickBounds**

Bounds get()

Obtiene el **boundsObject** desde este PickBounds.

void set(Bounds boundsObject)

Selecciona el **boundsObject** dentro de este PickBounds.

PickPoint

Los objetos **PickPoint** representan un punto para selección. Como una subclase de **PickShape**, los objetos **PickBounds** se usan con **BranchGroup** y **Locale** así como con clases del paquete **picking**.

Sumario de Constructores de **PickPoint**

Extiende: PickShape

Suministra un punto a los métodos de selección de **BranchGroup** y **Locale**

PickPoint()
Crea un PickPoint en (0, 0, 0).
PickPoint(Point3d location)
Crea un PickPoint en location.

Sumario de Métodos de **PickPoint**

void set(Point3d location)
Selecciona la posición de este PickPoint. Existe un método get correspondiente.

PickRay

Los objetos **PickRay** representan un rayo (un punto y una dirección) para selección. Como una subclase de **PickShape**, los objetos **PickBounds** se usan con **BranchGroup** y **Locale** así como con clases del paquete **picking**.

Sumario de Constructores de **PickRay**

Extiende: PickShape

PickRay es una encapsulación de un rayo para pasarlo a los métodos de selección en **BranchGroup** y **Locale**

PickRay()
Crea un PickRay con origen y dirección de (0, 0, 0).
PickRay(Point3d origin, Vector3d direction)
Crea un rayo desde origin con dirección a direction.

Sumario de Métodos de **PickRay**

void set(Point3d origin, Vector3d direction)
Selecciona el rayo que apunte desde origin en dirección direction. Existe el correspondiente método get.

PickSegment

Los objetos **PickSegment** representan un segmento de línea (definida por dos puntos) para selección. Como una subclase de **PickShape**, los objetos **PickBounds** se usan con **BranchGroup** y **Locale** así como con clases del paquete **picking**.

Sumario de Constructores de **PickSegment**

Extiende: PickShape

PickRay es una encapsulación de un segmento pasado a los métodos de selección de **BranchGroup** y **Locale**

PickSegment()
Crea un PickSegment.
PickSegment(Point3d start, Point3d end)
Crea un PickSegment desde el punto start hata el punto end.

Sumario de métodos de **PickSegment**

void set(Point3d start, Point3d end)
Selecciona el segmento desde el punto start hasta el punto end. Existe el correspondiente método get.

SceneGraphPath

La clase **SceneGraphPath** se usa en la mayoría de las aplicaciones de selección. Esto es porque normalmente la selección implica encontrar un camino de

escenario gráfico en el que se encuentra un objeto para permite la manipulación del objeto o de un objeto **TransformGroup** en el path.
Un objeto **SceneGraphPath** representa el camino del escenario gráfico hacia el objeto elegido permitiendo la manipulación del objeto o de un objeto **TransformGroup** en el camino del objeto.

Introducción a **SceneGraphPath**

Un objeto **SceneGraphPath** representa el camino desde un **Locale** hasta un nodo temrinal en el escenario gráfico. Este camino consiste en un **Locale**, un nodo terminal, y un array de nodos internos que están en el path desde el **Locale** hasta el nodo terminal. El nodo terminal podría ser un nodo **Leaf** o un nodo **Group**. Un **SceneGraphPath** válido debe identificar únicamente un ejemplar de un nodo terminal. Para nodos que no están bajo un **SharedGroup**, el **SceneGraphPath** mínimo consiste en el **Locale** y el propio nodo terminal. Para nodos que están bajo un **SharedGroup**, el **SceneGraphPath** mínimo consiste en el **Locale**, el nodo terminal, y una lista de todos los nodos **Link** en el camino desde el **Locale** hacia el nodo terminal. Un **SceneGraphPath** opcionalmente podría contener otros nodos interiores que están en el camino. Un **SceneGraphPath** se verifica contra errores cuando se envía como argumento a otros métodos de Java 3D.

En el array de nodos internos, el nodo en el índice 0 es el nodo más cercano al **Locale**. El índice se incrementa a lo largo del camino hacia el nodo terminal, con el nodo de índice longitud-1 siendo el nodo más cercano la nodo terminal. El array de nodos no contiene ni el **Locale** (que no es un nodo) ni el nodo terminal.

Sumario de Constructores de **SceneGraphPath**

Cuando un **SceneGraphPath** es devuelto desde métodos de selección o colisión de Java 3D, también contiene el valor del objeto **transform LocalToVworld** del nodo terminal que era en efecto en el momento en que ocurrió la colisión o la selección. Obherva que **ENABLE_PICK_REPORTING** y **ENABLE_COLLISION_REPORTING** están desactivados por defecto. Esto significa

que los métodos de selección y colisión devolverán el **SceneGraphPath** mínimo por defecto.

SceneGraphPath()

Construye un objeto SceneGraphPath con parámetros por defecto.

SceneGraphPath(Locale root, Node object)

Construye un nuevo objeto SceneGraphPath.

SceneGraphPath(Locale root, Node[] nodes, Node object)

Construye un nuevo objeto SceneGraphPath.

Lista Parcial de Métodos de **SceneGraphPath**

boolean equals(java.lang.Object o1)

Devuelve true si el objeto o1 es del tipo **SceneGraphPath** y todos los datos miembros de o1 son iguales a los miembros de datos correspondientes en este **SceneGraphPath** y si los valores de transformación son iguales.

Transform3D getTransform()

Devuelve una copia del transform asociado con este SceneGraphPath; devuelve null si no hay transform.

int hashCode()

Devuelve un número 'hash' basado en los valores de los datos de este objeto.

boolean isSamePath(SceneGraphPath testPath)

Determina si dos objetos **SceneGraphPath** representan el mismo path del escenario grafico; algún objeto podría incluir un subconjunto diferente de nodos internos; sólo los nodos links internos, **Locale**, y el propio nodo son comparados.

int nodeCount()

Recupera el número de nodos de este path.

void set(SceneGraphPath newPath)

Selecciona los valores del path al path especificado.

void setLocale(Locale newLocale)

Selecciona el **Locale** de este path a los Locale especificado.

void setNode(int index, Node newNode)

Reemplaza el nodo en el índice especificado con newNode.

void setNodes(Node[] nodes)

Selecciona el objeto nodo de este path con los objetos nodos especificados.

void setObject(Node object)

Selecciona el nodo terminal de este path al objeto nodo especificado.

void setTransform(Transform3D trans)

Selecciona el componente transform de este **SceneGraphPath** al valor del transform pasado.

java.lang.String toString()

Devuelve una representación string de este objeto; el string contiene los nombres de las clases de todos los nodos en el **SceneGraphPath**, el método toString() de cualquier usuario asociado, también imprime el transform si no es nulo.

Métodos de Selección de BranchGroup y Local

En los siguientes bloques de referencia están los métodos de las clases **BranchGroup** y **Local** para chequeo de intersección con objetos **PickShape**. Este es el nivel de cálculo de selección más bajo proporcionado por el API Java 3D.

Métodos de selección de **BranchGroup** y **Locale** para su uso con **PickShape**

SceneGraphPath[] pickAll(PickShape pickShape)

Devuelve un array que referencia todos los ítems que son elegibles bajo este **BranchGroup** que intereseccionan con **PickShape**. El array resultante no está ordenado.

SceneGraphPath[] pickAllSorted(PickShape pickShape)

Devuelve un array ordenado de referencias a todos los ítems elegibles que interseccionan con el **pickShape**. **Element [0]** referencia el ítem más cercano al origen de **PickShape**, con los elementos siguientes alejándose del origen. Nota: si **pickShape** es del tipo **PickBounds**, el array resultante no está ordenado.

SceneGraphPath pickClosest(PickShape pickShape)

Devuelve un **SceneGraphPath** que referencia el ítem elegible que está más cercano al origen de **pickShape**. Nota: si **pickShape** es del tipo **PickBounds**, la respuesta es cualquier nodo elegible debajo de este **BranchGroup**.

SceneGraphPath pickAny(PickShape pickShape)

Devuelve una referencia a cualquir ítem elegible debajo de este **BranchGroup** que interseccionа con **pickShape**.

Clases Generales del Paquete Picking

Incluidas en el paquete com.sun.j3d.utils.behaviors.picking hay varias clases de comportamientos generales y específicos. Las clases generales son útiles para crear nuevos comportamientos de selección, entre las que se incluyen **PickMouseBehavior**, **PickObject**, y **PickCallback**. Las clases específicas de comportamiento del ratón, presentadas en la siguiente sección, son subclases de **PickMouseBehavior**.

Clase PickMouseBehavior

Esta es la clase base para los comportamientos de selección específicos proporcionados en el paquete. También es útil para extender clases de comportamientos de selección personalizados.

Sumario de Métodos de **PickMouseBehavior**

Paquete: com.sun.j3d.utils.behaviors.picking

Extiende: Behavior

Clase base que permite a los programadores añadir selección y manipulación del ratón en un escenario gráfico (puedes ver **PickDragBehavior** para un ejemplo de cómo extender esta clase base).

void initialize()

Este método debería ser sobreescrito para proporcionar estado inicial y la condición de disparo inicial.

void processStimulus(java.util.Enumeration criteria)

Este método debería sobreescribirse para proporcionar el comportamiento en respuesta a una condición de disparo.

void updateScene(int xpos, int ypos)
Las subclases deberían implementar esta función update.

Clase PickObject

La clase **PickObject** proporciona métodos para determinar qué objeto fué seleccionado por una operación de selección del usuario. Una amplia variedad de métodos resulta de las distitnas formas posibles de aplicaciones de selección. Es útil crear clases de selección personalizadas.

Sumario de Constructores de **PickObject**

Paquete: com.sun.j3d.utils.behaviors.picking

Extiende: java.lang.Object

Contiene métodos para ayudar en la selección. Un **PickObject** se crea dando un **Canvas3D** y un **BranchGroup**. **SceneGraphObjects** bajo el **BranchGroup** especificado pueden chequearse para determinar si han sido seleccionados.

PickObject(Canvas3D c, BranchGroup root)
Crea un PickObject.

Lista Parcial de Métodos de **PickObject**

PickObject tiene numerosos métodos para el cálculo de intersecciones de un **pickRay** con objeto del escenario gráfico. Algunos de los métodos sólo difieren en un parámetro. Por ejemplo el segundo método pickAll (no listado) existe con el firma de método: SceneGraphPath[] pickAll(int xpos, int ypos, int flag), donde **flag** es uno de: **PickObject.USE_BOUNDS**, o **PickObject.USE_GEOMETRY**.

Esta lista ha sido ordenada para excluir los métodos con parámetros de bandera. Estos métodos son idénticos a los incluidos en esta lista con la diferencia del parámetro bandera. Estos métodos son: **pickAll**, **pickSorted**, **pickAny**, y

pickClosest.

PickShape generatePickRay(int xpos, int ypos)

Crea un PickRay que empieza en la posición del espectador y apunta dentro de la escena en dirección a (xpos, ypos) especificados en el espacio de la ventana.

SceneGraphPath[] pickAll(int xpos, int ypos)

Devuelve un array que referencia todos los ítems que son elegibles debajo del **BranchGroup** (especificado en el constructor de PickObject) que interseccionan con un rayo que empieza en la posición del espectador y apunta dentro de la escena en dirección (xpos, ypos) especificados en el espacio de la ventana.

SceneGraphPath[] pickAllSorted(int xpos, int ypos)

Devuelve un array ordenado de referencias a todos los ítems Pickable bajo el **BranchGroup** (especificado en el constructor de PickObject) que interesecciona con el rayo que empieza en la posición del espectador y apunta a la dirección de (xpos, ypos) en el espacio de la ventana.

SceneGraphPath pickAny(int xpos, int ypos)

Devuelve una referencia a cualquier ítem que sea elegible debajo del **BranchGroup** (especificado en el constructor de PickObject) que interesecciona con el rayo que empieza en la posición del espectador y apunta a la dirección (xpos, ypos) en el espacio de la ventana.

SceneGraphPath pickClosest(int xpos, int ypos)

Devuelve una referencia al ítem que está más cercano al espectador y es elegible bajo el **BranchGroup** (especificado en el constructor de PickObject) que interesecciona con el rayo que empieza en la posición del espectador y apunta a la dirección (xpos, ypos) en el espacio de la ventana.

Node pickNode(SceneGraphPath sgPath, int node_types)

Devuelve una referencia a un nodo elegible que es del tipo especificado que está contenido en el **SceneGraphPath** especificado. Donde **node_types** es la OR lógica de uno o más: **PickObject.BRANCH_GROUP**, **PickObject.GROUP**, **PickObject.LINK**, **PickObject.MORPH**, **PickObject.PRIMITIVE**, **PickObject.SHAPE3D**, **PickObject.SWITCH**, **PickObject.TRANSFORM_GROUP**.

Node pickNode(SceneGraphPath sgPath, int node_types, int occurrence)

Devuelve una referencia a un nodo elegible que es del tipo especificado que está contenido en el **SceneGraphPath** especificado. Donde **node_types** está definido en el método anterior. El parámetro **occurrence** indica qué objeto devolver.

nterface PickingCallback

El interface **PickingCallback** proporciona un marco de trabajo para extender una clase de selección existente. En particular cada una de las clases específicas implementa este interface permitiendo al programador proporcionar un método que sea llamado cuando la operación de selección tenga lugar.

Sumario de Métodos del I**nterface PickingCallback**

Paquete:

com.sun.j3d.utils.behaviors.picking
void transformChanged(int type, TransformGroup tg)
Llamado por el **Behavior Pick** que es retro-llamado que es registrado cada vez que se intenta la selección. Los valores de tipos válidos son: **ROTATE**, **TRANSLATE**, **ZOOM** o **NO_PICK** (el usuario hace una selección pero no hay nada seleccionado realmente).

Clase Intersect

La clase **Intersect** proporciona varios métodos para comprobar la intersección de un objeto **PickShape** (clase corazón) y geometrías primitivas. Esta clases es útil para la creacción de clases picking personalizadas.

Sumario de Constructores de I**ntersect**

Paquete: com.sun.j3d.utils.behaviors.picking

Extiende: java.lang.Object

Contiene métodos estáticos para ayudar a las comprobaciones de intersección entre varias clases **PickShape** y geometrías primitivas (como **quad**, **triangle**, **line** y

point).

Intersect()

Crea un objeto intersect.

Lista Parcial de Métodos de **Intersect**

Esta clase tiene vaios métodos de intersección, algunos de los cuales sólo se diferencian por un tipo de parámetro. Por ejemplo el método: boolean pointAndPoint(PickPoint point, Point3f pnt) se diferencia del segundo método listado aquí en el tipo del parámetro pnt. La mayoría de los métodos listado aquí con un parámetro del tipo **Point3d** tienen un método correspondiente con un parámetro del tipo **Point3f**.

boolean pointAndLine(PickPoint point, Point3d[] coordinates, int index)

Devuelve true si el **PickPoint** y el objeto **Line** interseccionan. coordinates[index] y coordinates[index+1] definen la línea.

boolean pointAndPoint(PickPoint point, Point3d pnt)

Devuelve true si el **PickPoint** y el objeto **Point3d** interseccionan.

boolean rayAndLine(PickRay ray, Point3d[] coordinates, int index,
double[] dist)

Devuelve true si el **PickPoint** y el objeto **Line** interseccionan. coordinates[index] y coordinates[index+1] definen la línea.

boolean rayAndPoint(PickRay ray, Point3d pnt, double[] dist)

Devuelve true si el **PickPoint** y el objeto **Point3d** interseccionan.

boolean rayAndQuad(PickRay ray, Point3d[] coordinates, int index,
double[] dist)

Devuelve true si el **PickPoint** y el objeto **cuadrilátero** interseccionan.

boolean rayAndTriangle(PickRay ray, Point3d[] coordinates, int index,
double[] dist)

Devuelve true si el triángulo intersecciona con el rayo, la distancia desde el origen del rayo al punto de intersección, se almacena en dist[0]. coordinates[index], coordinates[index+1], ycoordinates[index+2] definen el triángulo.

boolean segmentAndLine(PickSegment segment, Point3d[] coordinates, int index,

double[] dist)

Devuelve true si la lína interseccionа con el segmento; la distanta desde el inicio del segmento a la intersección se almacena dist[0]. coordinates[index] y coordinates[index+1] definen la línea.

boolean segmentAndPoint(PickSegment segment, Point3d pnt, double[] dist)

Devuelve true si el **PickSegment** y el objeto **Point3d** interseccionan.

boolean segmentAndQuad(PickSegment segment, Point3d[] coordinates, int index,
double[] dist)

Devuelve true si el quad intersecciona con el segmento; la distancia desde el inicio del segmento al punto de intersección se almacena en dist[0].

boolean segmentAndTriangle(PickSegment segment, Point3d[] coordinates,
int index, double[] dist)

Devuelve true si el triangulo intersecciona con el segmento; la distancia desde el inicio del segmento al punto de intersección se almacena en dist[0].

Clases de Comportamientos Picking Específicas

Incluidas en el paquete com.sun.j3d.utils.behaviors.picking hay clases de comportamientos específicas: **PickRotateBehavior**, **PickTranslateBehavior**, y **PickZoomBehavior**. Estas clases permiten al usuario interactuar con un objeto seleccionado con el ratón. Los comportamientos individuales responden a los diferentes botones del ratón (izquierdo=rotar, derecho=trasladar, central=zoom). Estas clases son subclases de **PickMouseBehavior**.

Los objetos de estas clases pueden incorporarse en mundos virtuales de Java 3D para proporcionar interacción siguiendo la receta anterior. Como todas estas clases implementan el interface **PickingCallback**, la operación de elección pueden mejorarse con una llamada a un método definido por el usuario.

PickRotateBehavior

Esta clase permite al usuario seleccionar y rotar interactivamente un objeto visual. El usuario usa el botón izquierdo del ratón para seleccionar y rotar. Se puede usar

un ejemplar de **PickRotateBehavior** en conjunción con otras clases de selección específicas.

Sumario de Constructores de **PickRotateBehavior**

Paquete: com.sun.j3d.utils.behaviors.picking

Extiende: PickMouseBehavior

Implementa: PickingCallback

Un comportamiento de ratón que permite al usuario seleccionar y rotar objetos de un escenario gráfico; expándible a través de retro-llamada.

PickRotateBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds)
Crea un comportamiento que espera eventos del ratón en el escenario gráfico.
PickRotateBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds,
int pickMode)
Crea un comportamiento que espera eventos del ratón en el escenario gráfico. El parámetro pickMode se especifica como uno de **PickObject.USE_BOUNDS** o **PickObject.USE_GEOMETRY**. Nota: si pickMode se selecciona a **PickObject.USE_GEOMETRY**, todos los objetos geométricos del escenario gráfico están disponibles para su selección y deben tener activado su **ALLOW_INTERSECT**.

Sumario de Métodos de **PickRotateBehavior**

void setPickMode(int pickMode)
Selecciona el componente pickMode de este PickRotateBehavior a uno de **PickObject.USE_BOUNDS** o **PickObject.USE_GEOMETRY**. Nota: si pickMode se selecciona a **PickObject.USE_GEOMETRY**, todos los objetos geométricos del escenario gráfico están disponibles para su selección y deben tener activado su **ALLOW_INTERSECT**.
void setupCallback(PickingCallback callback)
Registra la clase retrollamada a llamar cada vez que el objeto seleccionado se

mueve.

void transformChanged(int type, Transform3D transform)

Método de retrollamda desde MouseRotate. Se usa cuando al selección con retrollamada está activa.

void updateScene(int xpos, int ypos)

Actualiza la escena para manipular cualquier nodo.

PickTranslateBehavior

Esta clase permite al usuario seleccionar y trasladar interactivamente un objeto visual. El usuario usa el botón derecho del ratón para seleccionar y trasladar. Se puede usar un ejemplar de **PickTranslateBehavior** en conjunción con otras clases de selección específicas.

Sumario de Constructores de **PickTranslateBehavior**

Paquete: com.sun.j3d.utils.behaviors.picking

Extiende: PickMouseBehavior

Implementa: PickingCallback

Un comportamiento de ratón que permite al usuario seleccionar y trasladar objetos de un escenario gráfico; expándible a través de retro-llamada.

PickTranslateBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds)

Crea un comportamiento que espera eventos del ratón para el escenario gráfico

PickTranslateBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds,
int pickMode)

Crea un comportamiento que espera eventos del ratón en el escenario gráfico. El parámetro pickMode se especifica como uno de **PickObject.USE_BOUNDS** o **PickObject.USE_GEOMETRY**. Nota: si pickMode se selecciona a **PickObject.USE_GEOMETRY**, todos los objetos geométricos del escenario gráfico están disponibles para su selección y deben tener activado su

ALLOW_INTERSECT.

Sumario de Métodos de **PickTranslateBehavior**

void setPickMode(int pickMode)

Selecciona el componente pickMode de este PickTranslateBehavior al valor pasado en pickMode.

void setupCallback(PickingCallback callback)

Registra la clase de retrollamada que será llamada cada vez que el objeto seleccionado se mueva.

void transformChanged(int type, Transform3D transform)

Método de retrollamada desde MouseTranslate. Se usa cuando la selección por retrollamada está activa.

void updateScene(int xpos, int ypos)

Actualiza la escena para manipular cualquier nodo.

PickZoomBehavior

Esta clase permite al usuario seleccionar y hacer zoom interactivamente un objeto visual. El usuario usa el botón central del ratón para seleccionar y hacer zoom. Se puede usar un ejemplar de **PickZoomBehavior** en conjunción con otras clases de selección específicas.

Sumario de Constructores de **PickZoomBehavior**

Paquete: com.sun.j3d.utils.behaviors.picking

Extiende: PickMouseBehavior

Implementa: PickingCallback

Un comportamiento de ratón que permite al usuario seleccionar y hacer zoom a objetos de un escenario gráfico; expándible a través de retro-llamada.

PickZoomBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds)

Crea un comportamiento que espera eventos del ratón para el escenario gráfico.

PickZoomBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds,
int pickMode)

Crea un comportamiento que espera eventos del ratón en el escenario gráfico. El parámetro pickMode se especifica como uno de **PickObject.USE_BOUNDS** o **PickObject.USE_GEOMETRY**. Nota: si pickMode se selecciona a **PickObject.USE_GEOMETRY**, todos los objetos geométricos del escenario gráfico están disponibles para su selección y deben tener activado su **ALLOW_INTERSECT**.

Sumario de Métodos de **PickZoomBehavior**

void setPickMode(int pickMode)

Selecciona el componente pickMode de este PickZoomBehavior al valor pasado en pickMode.

void setupCallback(PickingCallback callback)

Registra la clase de retrollamada a llamar cada vez que el objeto se seleccione.

void transformChanged(int type, Transform3D transform)

Método de retrollamada desde MouseZoom. Se usa cuando la selección con retrollamada está activa.

void updateScene(int xpos, int ypos)

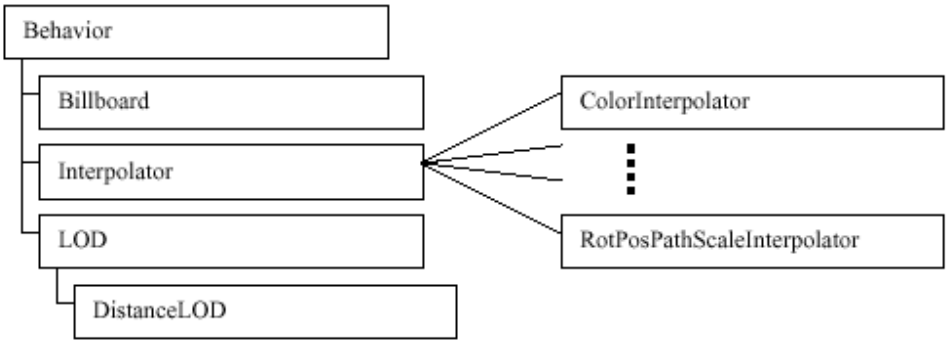
Actualiza la escena para manipular cualquier nodo.

Animación en Java 3D

La igual que las interacciones, las animaciones Java 3D se implementan usando ojetos **Behavior**. Como podrás imaginar, se puede crear cualquier animación

personalizada usando objetos **Behavior**. Sin embargo, el API Java 3D proporciona varias clases útiles para crear animaciones sin tener que crear una nueva clase. No debería sorprendernos que estas clases estén basadas en la clase **Behavior**. Un conjunto de clases de animación es conocido como interpoladores. Un objeto **Interpolator**, junto con un objeto **Alpha**, manipula algún parámetro de un objeto del escenario gráfico para crear animaciones basadas en el tiempo. El objeto **Alpha** proporciona el temporizado.

Otro conjunto de clases de animación animan objetos visuales en respuestas a cambios en la vista. Este conjunto de clases incluye los comportamientos **Billboard** y **Level of Detail (LOD)** que no están dirigidos por el paso del tiempo, sino por la posición u orientación de la vista. La Figura 5-1 muestra las clases de alto nivel del árbol de clases para animación.



Los Interpoladores y los Objetos Alpha Proporcionan Animaciones Basadas en el Tiempo

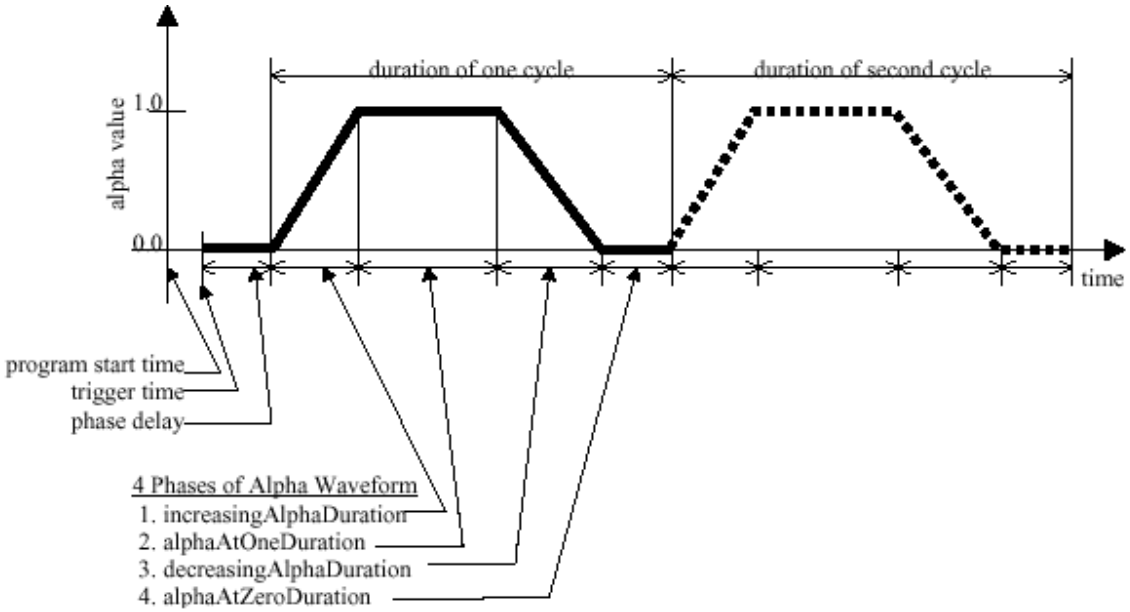
Un objeto **Alpha** produce un valor entre cero y uno, inclusives, dependiendo de la hora y los parámetros del objeto **Alpha**. Los **Interpolators** son objetos **behavior** personalizados que usan un objeto **Alpha** para proporcionar animaciones de objetos visuales. Las acciones de **Interpolator** incluyen el cambio de localización, orientación, tamaño, color o transpariencia de un objeto visual. Todos los comportamientos interpoladores podrían implementarse creando una clase **behavior** personalizada; sin embargo, usar un **interpolador** hacer más sencilla la

creacción de estas animaciones. Las clases **Interpolator** existen para otras acciones, incluyendo algunas combinaciones de estas acciones.

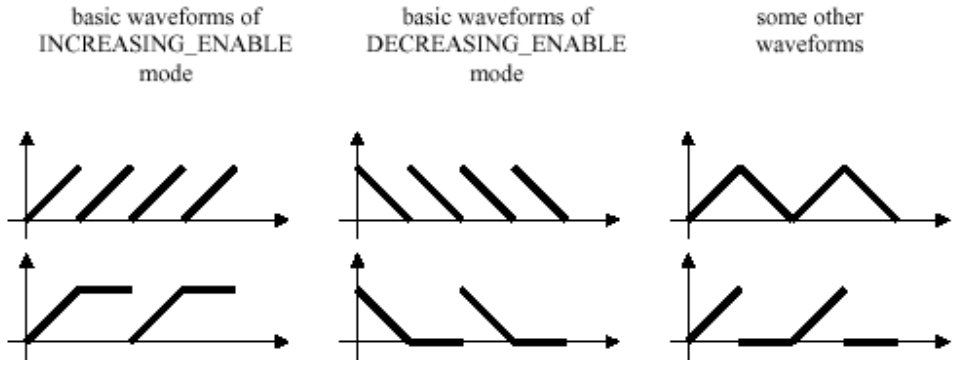
- **Alpha**

Un objeto **Alpha** produce un valor, llamado **valor alpha**, entre 0,0 y 1,0; ámbos inclusives. El **valor alpha** cambia con el tiempo según los parámetros especificados en el objeto **alpha**. Para unos parámetros especificos en un momento particular, sólo hay un **valor alpha** que el objeto **alpha** producirá. Dibujando el **valor alpha** sobre el tiempo veremos la forma de onda que produce el objeto **alpha**.
La forma de onda del objeto **alpha** tiene cuatro fases: incremento, alpha a uno, decremento, y alpha a cero. La colección de las cuatro fases es un ciclo de la forma de onda **alpha**. estas cuatro fases corresponden con cuatro parámetros del objeto **Alpha**. La duración de las cuatro fases se especifica por un valor entero expresando su duración en milisegundos. La Figura 5-2 muestra las cuatros fases de la forma de onda **Alpha**.
Todos los tiempos **alpha** son relativos al momento de inicio del objeto **Alpha**. El momento de arranque para todos los objetos **Alpha** se toma de la hora de arranque del sistema. Consecuentemente, los objetos **Alpha** creados en diferentes momentos tienen el mismo momento de inicio. Como resultado, todos los objetos interpoladores, incluso aquellos basados en diferentes objetos **Alpha**, están sincronizados.
Los objetos **Alpha** pueden iniciar sus formas de onda en diferentes momentos. El inicio de la primera forma de onda de un objeto **alpha** puede retrasarse usando alguno de los otros dos parámetros TriggerTime y PhaseDelayDuration. El parámetro TriggerTime especifica un tiempo después de **StartTime** para empezar la operación del objeto **Alpha**. Un tiempo especificado por el parámetros PhaseDelayDuration después de TriggerTime, empezará el primer ciclo de la forma de onda. La Figura 5-2 muestra el StartTime, TriggerTime y PhaseDelayDuration.

Una forma de onda **alpha** podría realizarse sólo una vez, repetirse un número determinado de veces, o hacer un ciclo contínuo. El número de ciclos se especifica en el parámetro loopCount. Cuando loopCount es positivo, especifica un número de ciclos. Si es -1 especifica un bucle contínuo. Cuando la forma de onda **alpha** se repite más de una vez, se repiten las cuatro fases del ciclo, pero no se repite el retardo de fase.



Una forma de onda **alpha** no siempre usa las cuatro fases. Podría estar formada por una, dos, tres o las cuatro fases de la forma de onda **Alpha**. La Figura 5-3 muestra seis de las quince formas de ondas posibles.



El objeto **alpha** tiene dos modos que especifican un subconjunto de fases a usar. El modo **INCREASING_ENABLE** indica que se usan las fases de incremento alpha y alpha a uno . El modo **DECREASING_ENABLE** indica que se usan las fases

decremento alpha y alpha a cero. Un tercer modo es la combiandción de los dos anteriores, lo que indica que se usarán las cuatro fases.
La especificación de modo sobreescribe las selecciones de los parámetros de duración. Por ejemplo, con el modo **INCREASING_ENABLE**, se ignoran los parámetros DecreasingAlphaDuration, DecreasingAlphaRampDuration, y AlphaAtZeroDuration. También, se puede especificar cualquier forma de onda especificando las duracciones de las fases no necesarias como cero, la especificación apropiada del modo incrementa la eficiciencia del objeto **Alpha**.

- **Usar Objetos Interpolator y Alpha**

La receta para usar objetos **Interpolator** y **Alpha** es muy similar a la usada para cualquier objeto **behavior**. La diferencia principal es la inclusión del objeto **Alpha**.

1. crear el objeto fuente con las capacidades apropiadas
2. crear el objeto **Alpha**
3. crear el objeto **Interpolator** que referencia al objeto **Alpha** y al objeto fuente
4. añadir límites al objeto **Interpolator**
5. añadir el objeto **Interpolator** al escenario gráfico

- **Ejemplo de uso de Alpha y RotationInterpolator**

ClockApp.java es un ejemplo de uso de la clase **RotationInterpolator**. La escena es la esfera de un reloj. El reloj es rotado por unos objetos **RotationInterpolator** y **Alpha** una vez por minuto.
En esta aplicación, el objeto fuente es un **TransformGroup**, que necesita la capacidad **ALLOW_TRANSFORM_WRITE**. Algunos otros interpoladores actúan sobre diferentes objetos fuente. Por ejemplo, la fuente para un objeto **ColorInterpolator** es un objeto **Material**. Un objeto **interpolator** selecciona los valores de su objeto fuente basándose en el valor **alpha** y en los valores que el propio objeto **interpolator** contiene.
El **interpolator** define los puntos finales de la animación. En el caso de **RotationInterpolator**, el objeto especifica los ángulos inicial y final de la rotación. El **alpha** controla la animación con respecto al tiempo y cómo se moverá el

interpolador desde un punto definido hasta el otro especificando las fases de la forma de onda **apha**

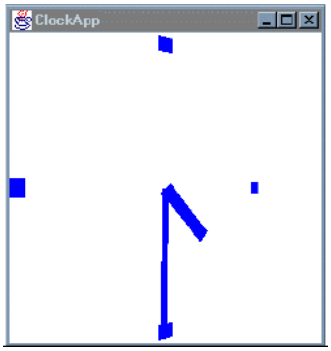
Esta aplicación usa las selecciones por defecto de **RotationInterpolator** con un ángulo inicial de cero y un ángulo final de 2P (una rotación completa). El eje por defecto de rotación es el eje y. El objeto **alpha** se selecciona para rotar contínuamente (loopCount = -1) con un periodo de un minuto (60.000 milisegundos). La combinación de estos dos objetos creará el efecto visual de una rotación completa cada minuto. El ciclo se repite continuamente. El resultado parece que el reloj está moviendose continuamente, no como si se parara y empezara de nuevo. El Fragmento de Código 5-1 muestra el método createSceneGraph de la aplicación ClockApp.java.

Fragmento de Código 5-1, usar un RotationInterpolator y un Alpha en un Reloj (de ClockApp)

```
1.  public BranchGroup createSceneGraph() {
2.  // Create the root of the branch graph
3.  BranchGroup objRoot = new BranchGroup();
4.
5.  // create target TransformGroup with Capabilities
6.  TransformGroup objSpin = new TransformGroup();
7.  objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
8.
9.  // create Alpha that continuously rotates with a period of 1 minute
10. Alpha alpha = new Alpha (-1, 60000);
11.
12. // create interpolator object; by default: full rotation about y-axis
13. RotationInterpolator rotInt = new RotationInterpolator(alpha, objSpin);
14. rotInt.setSchedulingBounds(new BoundingSphere());
15.
16. //assemble scene graph
17. objRoot.addChild(objSpin);
18. objSpin.addChild(new Clock());
19. objRoot.addChild(rotInt);
20.
21. // Let Java 3D perform optimizations on this scene graph.
```

```
22.  objRoot.compile();
23.
24.  return objRoot;
25.  } // end of CreateSceneGraph method of ClockApp
```

La Figura 5-5 es una escena renderizada por **ClockApp** a las 4:30. La cara del reloj es oblicua al espectador porque todo el reloj está rotando.



El programa ClockApp muestra una sencilla aplicación de **RotationInterpolator**. El objeto **Clock**, está definido en el fichero Clock.java, muestra una aplicación más avanzada del objeto **RotationInterpolator**. El objeto **clock** usa un objeto **RotationInterpolator** para animar cada manecilla del reloj. Sin embargo, sólo se usa un objeto **alpha**. No es necesario utilizar otro objeto **alpha** para coordinar las manecillas, como se mencionó anteriormente, todos los objetos **Alpha** se sicronizan con el momento de arranque del programa. Sin embargo, compartir un objeto **Alpha** ahorra memoria del sistema.

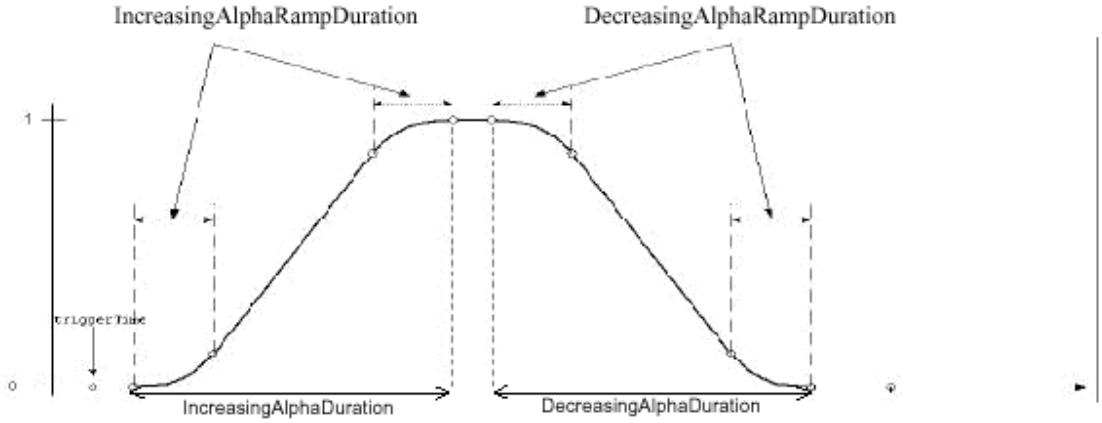
Algunas de las características especiales de la clase **Clock** son:

- La selección del ángulos inicial y final de las manecillas,
- la selección de los ejes de rotación, y
- la selección del recorte poligonal de varios componentes del reloj.

Suavizar la Forma de Onda Alpha

Además de la duración de las cuatro fases, el programador puede especificar una duración de la rampa para las fases de incremento y decremento **alpha**. Durante la duración de la rampa, el valor de **alpha** cambia gradualmente. En el caso de interpoladores del movimiento, parecerá como si el objeto visual acelerara y decelerara de un forma más natural, como en el mundo real.

El valor de la duración de la rampa se utiliza para las porciones inicial y final de la fase y por lo tanto la duración de la rampa está limitada a la mitad de la duración de la fase. La Figura 5-6 muestra una forma de onda **alpha** con IncreasingAlphaRampDuration y DecreasingAlphaRampDuration. Observa que el valor de **alpha** cambia linearmente entre los dos períodos de la rampa.



El programa del ejemplo, AlphaApp.java, demuestra el efecto de IncreasingAlphaRampDuration en una forma de onda **alpha**. En este programa hay tres objetos visuales car. Los tres coches comienzan al mismo tiempo en la misma coordenada X y viajan en paralelo. El coche superior no tiene ninguna rampa (duración de la rampa = 0), el coche inferior tiene una duración máxima de la rampa (mitad de la duración del incremento o decremento **alpha**), y el coche del medio tiene la mitad de la duración máxima de la rampa (un cuarto de la duración del incremento o decremento **alpha**). Cada coche tarda dos segundos en cruzar la vista. La Figura 5-7 muestra cuatro escenas renderizadas de esta aplicación.

Unos 0,4 segundos después de que los coches comiencen, la primera imagen (izquierda) de la Figura 5-7 fue capturada mostrando las posiciones de los coches. El coche superior, que procederá en un ratio constante en ausencia de una rampa,

ha viajado la mayor parte de la distancia en el primer marco. Los otros dos coches comienzan más lentamente y aceleran. Un segundo después (no mostrado), todos los coches han viajado la misma distancia. Las posiciones relativas se invierten durante la segunda mitad de la fase. Al final de los dos segundos, cada uno de los coches ha viajado la misma distancia.

El API Alpha

El API de la clase de la **Alpha** es correcto. Cuatro constructores cubren las aplicaciones más comunes. Una plétora de métodos, enumerada en el bloque de referencia siguiente, hace el trabajo fácil de modificar un objeto **alpha** para personalizar un objeto **alpha** para cualquier aplicación.

Sumario de Constructores de **Alpha**

Extiende: Object

La clase **Alpha** convierte un valor de tiempo en un valor **alpha** (un valor en el rango 0 a 1, inclusivo). El objeto **alpha** es efectivamente una función de tiempo que genera valores en el rango [0.1]. Un uso común de **alpha** proporciona valores **alpha** para los comportamientos del interpolador. Las características del objeto **alpha** están determinadas por parámetros definibles por el usuario.

Alpha()
Construye un objeto **Alpha** con el mode = INCREASING_ENABLE, loopCount = -1, increasingAlphaDuration = 1000, y todos los demás parámetros = 0, excepto StartTime. StartTime se selecciona en el momento de comienzo del programa.

Alpha(int loopCount, long increasingAlphaDuration)
Este constructor sólo toma loopCount e increasingAlphaDuration como parámetros, selecciona el modo a INCREASING_ENABLE y asigna 0 a todos los demás parámetros (excepto a StartTime).

Alpha(int loopCount, long triggerTime, long phaseDelayDuration,
long increasingAlphaDuration, long increasingAlphaRampDuration,
long alphaAtOneDuration)

Construye un nuevo objeto Alpha y selecciona el modo a INCREASING_ENABLE.

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

Este constructor toma todos los parámetros definidos por el usuario.

Lista Parcial de Métodos de **Alpha**

Cada uno de estos métodos tiene su correspondiente método get que devuelve un valor del tipo que corresponda al parámetro del método set.

boolean finished()

Comprueba si este objeto alpha ha finalizado totalmente su actividad.

void setAlphaAtOneDuration(long alphaAtOneDuration)

Selecciona el valor de alphaAtOneDuration al valor especificado.

void setAlphaAtZeroDuration(long alphaAtZeroDuration)

Selecciona el valor de alphaAtZeroDuration al valor especificado.

void setDecreasingAlphaDuration(long decreasingAlphaDuration)

Selecciona el valor de decreasingAlphaDuration al valor especificado.

void setDecreasingAlphaRampDuration(long decreasingAlphaRampDuration)

Selecciona el valor de decreasingAlphaRampDuration al valor especificado.

void setIncreasingAlphaDuration(long increasingAlphaDuration)

Selecciona el valor de increasingAlphaDuration al valor especificado.

void setIncreasingAlphaRampDuration(long increasingAlphaRampDuration)

Selecciona el valor de increasingAlphaRampDuration al valor especificado.

void setLoopCount(int loopCount)

Selecciona el valor de loopCount al valor especificado.

void setMode(int mode)

Selecciona el modo al especificado en este argumento. Este puede ser INCREASING_ENABLE, DECREASING_ENABLE, o un valor OR de los dos

- DECREASING_ENABLE - Especifica que se usan las fases 3 y 4
- INCREASING_ENABLE - Especifica que se usan las fases 1 y 2.

void setPhaseDelayDuration(long phaseDelayDuration)

Selecciona el valor de phaseDelayDuration al valor especificado.

void setStartTime(long startTime)

Selecciona el startTime al valor especificado; startTime selecciona la base (o cero) para todos los cálculos relativos al tiempo, el valor por defecto es la hora de arranque del sistema.

void setTriggerTime(long triggerTime)

Selecciona el valor triggerTime al valor especificado.

float value()

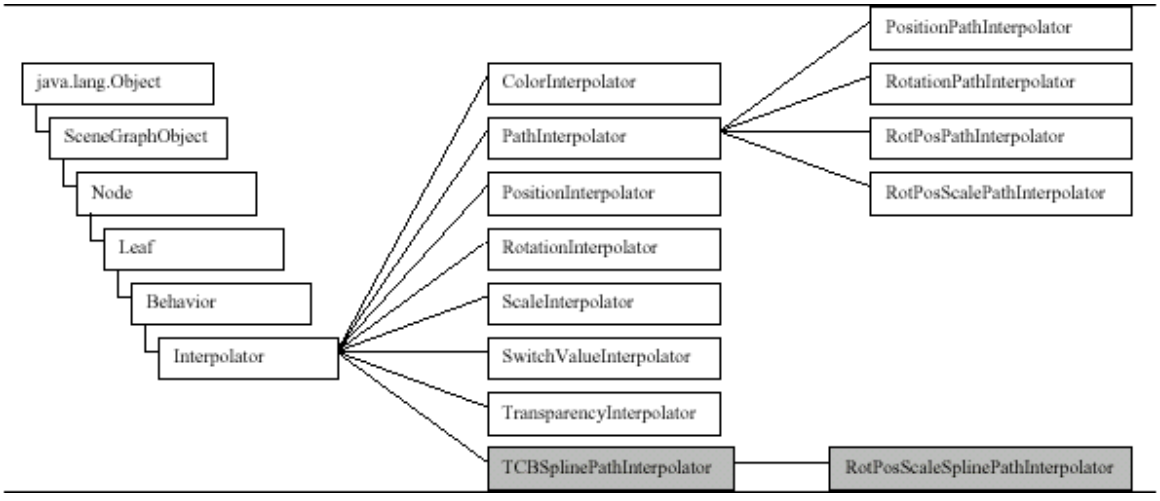
Esta función devuelve un valor entre 0,0 y 1,0 inclusives, basandose en la hora actual y todos los parámetros establecidos para este objeto **alpha**.

float value(long atTime)

Esta función devuelve un valor entre 0,0 y 1,0 inclusives, basandose en la hora actual y todos los parámetros establecidos para este objeto **alpha**.

Clases de Comportamiento Interpolator

La Figura 5-8 muestra las clases del **Interpolator** en los paquetes base y de utilidad. En esta figura, se puede ver que hay unas 10 clases **Interpolator**, y eso son todas las subclases de la clase **Interpolator**. También, esta clase es una extensión de **Behavior**. Los dos rectángulos sombreados representan clases **Interpolator** de utilidad, los otros rectángulos representan clases **Interpolator** básicas.



Cada **Interpolator** es un **Behavior** personalizado con un disparador para despertar cada marco. En el método processStimulus, de un objeto **Interpolator** controla su objeto **alpha** asociado para saber si el valor actual **alpha**, ajusta la fuente basándose en el valor **alpha**, entonces reajusta su disparador al marco siguiente (a menos que se haya terminado el **alpha**). Alguna de estas funcionalidades se proporcionan en la clase **Interpolator**. La mayoría de este comportamiento se implementa en cada clase individual **Interpolator**.

La mayoría de los objetos del **Interpolator** almacenan dos valores que se utilizan como los puntos finales para la acción interpolada. Por ejemplo, el **RotationInterpolator** guarda dos ángulos que son los extremos de la rotación proporcionada por este **Interpolator**. Por cada marco, el objeto **Interpolator** controla el valor **alpha** de su objeto **alpha** y hace el ajuste rotatorio apropiado a su objeto **TransformGroup** fuenete. Si el valor **alpha** es 0, entonces se usa uno de los valores; si el valor **alpha** es 1, se utiliza el otro valor. Para los valores **alpha** entre 0 y 1, el **Interpolator** interpola linearmente entre los dos valores basándose en el valor **alpha** y utiliza el valor que resulta para el ajuste del objeto fuente

Esta descripción general de **Interpolator** no describe bien las clases **SwitchValueInterpolator** ni **PathInterpolator**. El **SwitchValueInterpolator** elige uno entre los nodos hijos del objeto **Switch** fuente basándose en el valor **alpha**; por lo tanto, no se hace ninguna interpolación en esta clase.

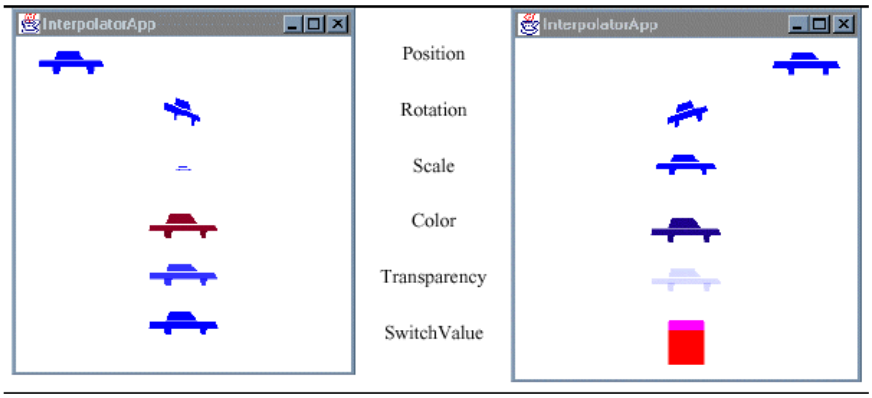
Mientras que varias de las clases **Interpolator** son similares, también se diferencian en algunos detalles. La siguiente tabla muestra algunas de las diferencias entre clases **Interpolator**.

Clase Interpolator	usada para	tipo de objeto fuente
ColorInterpolator	cambia el color difuso de un objeto(s)	Material
PathInterpolator	Clase Abstracta	TransformGroup
PositionInterpolator	cambia la posición de un objeto(s)	TransformGroup
RotationInterpolator	cambia la rotación (orientación) de un objeto(s)	TransformGroup
ScaleInterpolator	cambia el tamaño de un objeto(s)	TransformGroup
SwitchValueInterpolator	elige uno (cambia) entre una colección de	Switch

objetos

TransparencyInterpolator	cambia la transparencia de un objeto(s)	TransparencyAttributes

El programa del ejemplo, InterpolatorApp.java, demuestra seis clases no-abstractas **Interpolator** de la tabla anterior. En este programa, cada objeto **Interpolator** está dirigido por un solo objeto **alpha**. La Figura 5-9 muestra dos escenas renderizadas de **InterpolatorApp**. Los cambios en la posición, la rotación, la escala, el color, la transparencia, y el objeto visual (de arriba a abajo) son realizados por los objetos **PositionInterpolator**, **RotationInterpolator**, **ScaleInterpolator**, **ColorInterpolator**, **TransparencyInterpolator**, y **SwitchValueInterpolator**, respectivamente.



Riesgos de Programación de la Clase Interpolator

Los objetos **Interpolator** se derivan, y se relacionan de cerca, a objetos **Behavior**. Por lo tanto, usar objetos **Interpolator** da lugar a los mismos riesgos de programación que usar objeto **behavior**. Además de éstos, hay riesgos de programación **Interpolator** en general, y riesgos específicos para algunas clases **Interpolator**.

Un riesgo potencial de programación de **Interpolator** es no darse cuenta de que el objeto **Interpolator** contiene el valor de sus objetos fuente. Podríamos pensar que el **TransformGroup** fuente de un **RotationInterpolator** se puede utilizar para trasladar el objeto visual además de la rotación proporcionada por el **Interpolator**. Esto no es verdad. El objeto **transform** seleccionado en el objeto **TransformGroup**

fuente se reescribe en cada marco en que el objeto **alpha** está activo. Esto también significa que dos **Interpolator** no pueden tener el mismo objeto fuente. Otro riesgo general del **Interpolator** es no seleccionar apropiadamente las capacidades del objeto fuente. Fallar al hacer esto resultará en un error de tiempo de ejecución.

API Corazón de Interpolator

Como una clase abstracta, **Interpolator** sólo se usa para crear una subclase nueva. La clase **Interpolator** proporciona solamente un método para los usuarios de sus subclases. Los métodos útiles para la escritura de subclases no se enumeran aquí. La mayoría de la información necesaria para escribir una subclase del **Interpolator** puede obtenerse del Capítulo anterior

Lista Parcial de Métodos de la Clases **Interpolator**

Extiende: Behavior

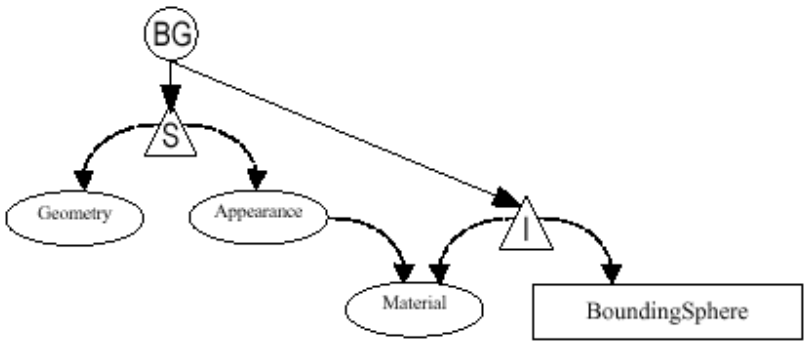
Subclases conocidas: ColorInterpolator, PathInterpolator, PositionInterpolator, RotationInterpolator, ScaleInterpolator, SwitchValueInterpolator, TCBSplinePathInterpolator, TransparencyInterpolator

El comportamiento **Interpolator** es un clase abstracta que proporciona bloques de construcción usados por varios interpoladores especializados.

void setAlpha(Alpha alpha)
Selecciona el objeto alpha de este interpolador al objeto alpha especificado.

ColorInterpolator
Un objeto **ColorInterpolator** tiene un objeto **Material** como su fuente. Este **Interpolator** cambia el color difuso del **Material** de la fuente. Esto hace del **ColorInterpolator** tan poderoso como limitado. La potencia viene de la capacidad de tener más de un objeto visual del mismo objeto **Material**. Así pues, un **ColorInterpolator** con una fuente **Material** puede afectar a más de un objeto visual.

La limitación es que los objetos visuales con un **Material** NodeComponent son solamente visibles cuando se iluminan.
La mayoría de los riesgos de programación potenciales son el resultado de la complejidad de escenas (iluminadas) sombreadas. La iluminación es tan compleja que es el tema de un capítulo entero, el Capítulo 6. Por ejemplo, el color de un objeto visual sombreado es la combinación de componentes specular, difuso, y ambiente. El **ColorInterpolator** cambia solamente uno de los tres componentes, el color difuso, así que en ciertas situaciones es enteramente posible que parezca que el **ColorInterpolator** no afecta al objeto visual.
Otro riesgo de programación potencial menos exótico es no agregar el objeto **Material** fuente del objeto **Shape3D**. La Figura 5-10 muestra un diagrama parcial del escenario gráfico de un **ColorInterpolator** y de su **Material** NodeComponent fuente.



El **ColorInterpolator** es diferente de otros **Interpolator** en el formato de sus métodos get. Los métodos get de **ColorInterpolator** tienen parámetros; por lo tanto se listan con los métodos set

Sumario de Constructores de **ColorInterpolator**

Extiende: Interpolator

Esta clase define un comportamiento que modifica el color difuso de su objeto **Material** fuente interpolandolo linearmente entre una pareja de colores especificados (usando el valor generado por el objeto **Alpha**).

ColorInterpolator(Alpha alpha, Material target)

Construye un colorInterpolator trivial con una fuente especificada, un color inicial negro y un color final blanco.

ColorInterpolator(Alpha alpha, Material target, Color3f startColor,
Color3f endColor)

Construye un colorInterpolator con la fuente especificada, y los colores inicial y final.

Lista Parcial de Métodos de **ColorInterpolator**

Los métodos get no siguen las convenciones de los otros interpoladores.

void setEndColor(Color3f color)

Selecciona el color final de este interpolator.

Correspondiente método get: void getEndColor(Color3f color)

void setStartColor(Color3f color)

Selecciona el color inicial de este interpolator.

Correspondiente método get: void getStartColor(Color3f color)

void setTarget(**Materia**l target)

Seleccion el componente fuente de este interpolator.

Correspondiente método get: Material getTarget()

PositionInterpolator

El **PositionInterpolator** varía la posición de un objeto visual a lo largo de un eje. La especificación de los puntos finales de la interpolación se hace con dos valores de coma flotante y un eje de traslación. El valor por defecto del eje de traslación es el eje X

Sumario de Constructores de **PositionInterpolator**

Extiende: Interpolator

Esta clase define un comportamiento que modifica el componente de translación de su **TransformGroup** fuente interpolandolo linearmente entre un par de posiciones especificadas (que usan el valor generado por el objeto **alpha**

especificado). La posición interpolada se utiliza para generar una traslación a lo largo del eje X local (o del eje especificado en la traslació) de este **Interpolator**.

PositionInterpolator(Alpha alpha, TransformGroup target)
Construye un positionInterpolator trivial con la fuente especificada, con el eje de traslación por defecto (X), una posición inicial de 0.0f, y una posición final de 1.0f.
PositionInterpolator(Alpha alpha, TransformGroup target,
Transform3D axisOfTranslation, float startPosition, float endPosition)
Construye un nuevo positionInterpolator que varía el componente translacional del TransformGroup (startPosition y endPosition) a lo largo del eje de traslación especificado.

Lista Parcial de Métodos de **PositionInterpolator**

Cada uno de estos métodos tiene un correspondiente método get sin parámetros que devuelve un valor del tipo correspondiente al parámetro del método set.

void setAxisOfTranslation(Transform3D axisOfTranslation)
Selecciona el eje de traslación de este interpolator.
void setEndPosition(float position)
Selecciona la posición final de este interpolator.
void setStartPosition(float position)
Selecciona la posición inicial de este interpolator.
void setTarget(TransformGroup target)
Selecciona la fuente de este interpolator.

RotationInterpolator

El RotationInterpolator varía la orientación rotacional de un objeto visual sobre un eje. La especificación de los puntos finales de la interpolación se hace con dos valores de ángulo en coma flotante y un eje de la rotación. El valor por defecto del eje de rotación es el eje Y positivo.

Sumario de Constructores de **RotationInterpolator**

Extiende: Interpolator

Esta clase define un comportamiento que modifica el componente rotacional de su **TransformGroup** fuente interpolandolo linearmente entre un par de ángulos especificados (que usan el valor generado por el objeto **alpha** especificado). El ángulo interpolado se utiliza para generar una rotación sobre el eje Y local de este **Interpolator**, o el eje especificado de rotación.

RotationInterpolator(Alpha alpha, TransformGroup target)
Construye un rotationInterpolator trivial con una fuente especificada el eje de rotación por defecto es (+Y), un ángulo mínimo de 0.0f, y un ángulo máximo de 2*pi radianes.
RotationInterpolator(Alpha alpha, TransformGroup target,
Transform3D axisOfRotation, float minimumAngle, float maximumAngle)
Construye un nuevo rotationInterpolator que varía el componente rotacional del componente.

Lista Parcial de Métodos de **RotationInterpolator**

Cada uno de estos métodos tiene un correspondiente método get sin parámetros que devuelve un valor del tipo correspondiente al parámetro del método set.

void setAxisOfRotation(Transform3D axisOfRotation)
Selecciona el eje de rotación de este interpolator.
void setMaximumAngle(float angle)
Selecciona el ángulo máximo de este interpolator, en radianes.
void setMinimumAngle(float angle)
Selecciona el ángulo mínimo de este interpolator, en radianes.
void setTarget(TransformGroup target)
Selecciona el nodo TransformGroup para este interpolator.

ScaleInterpolator

El **ScaleInterpolator** varia el tamaño de un objeto visual. La especificación de los puntos finales de la interpolación se hace con dos valores en coma flotante.

Sumario de Constructores de **ScaleInterpolator**

Extiende: Interpolator

Esta clase define un comportamiento que modifica el componente de la escala de su **TransformGroup** fuente interpolandolo linearmente entre un par de valores de escala especificados (que usan el valor generado por el objeto **alpha** especificado). El valor de escala interpolado se utiliza para generar una escala en el sistema de coordenadas local de este **Interpolator**.

ScaleInterpolator(Alpha alpha, TransformGroup target)

Construye un scaleInterpolator que varía el nodo TransformGroup de su fuente entre los dos valores **alpha**, una matriz de identidad, una escala mínima de 0.1f, y una escala máxima de 1.0f.

ScaleInterpolator(Alpha alpha, TransformGroup target, Transform3D axisOfScale,
float minimumScale, float maximumScale)

Construye un nuevo scaleInterpolator que varía el componente de escala de su nodo TransformGroup entre dos valores de escala (minimumScale y maximumScale).

Sumario de Métodos de **ScaleInterpolator**

Cada uno de estos métodos tiene un correspondiente método get sin parámetros que devuelve un valor del tipo correspondiente al parámetro del método set.

void setAxisOfScale(Transform3D axisOfScale)

Selecciona el eje de escala para este interpolator.

void setMaximumScale(float scale)

Selecciona la escala máxima para este interpolator.

void setMinimumScale(float scale)

Selecciona la escala mínima para este interpolator.

void setTarget(TransformGroup target)

Selecciona el TransformGroup fuente para este interpolator.

SwitchValueInterpolator

SwitchValueInterpolator no interpola entre dos valores como otros **Interpolator**. Selecciona uno de los hijos de un objeto **Switch** para renderizarlo. Los valores de umbral para cambiar a un hijo diferente se determinan uniformemente dividiendo el rango 0,0 a 1,0 por el número de hijos que teine el objeto **Switch**.

Un riesgo potencial específico de programación de **SwitchValueInterpolator** miente en el hecho de que el **Interpolator** no es actualizado cuando cambia el número hijos del objeto **Switch**. Más importante, se determinan los valores de umbral de la conmutación cuando se crea el objeto de **SwitchValueInterpolator**. Así pues, si el **Switch** no tiene ningún hijo antes de que se cree el **Interpolator**, o si el número de hijos cambia después de que se cree el objeto **Interpolator**, entonces el número de hijos en el objeto **Interpolator** debe ser actualizado. La ventaja es que podemos especificar un subconjunto de índices que el **Interpolator** utilizará. El subconjunto se limita a un conjunto secuencial de índices.

Sumario de Constructores de **SwitchValueInterpolator**

Extiende: Interpolator

Esta clase define un comportamiento que modifica el hijo seleccionado del nodo **switch** interpolandolo linearmente entre un par hijos especificados (usando el valor generado por el objeto **alpha**).

SwitchValueInterpolator(Alpha alpha, Switch target)

Construye un SwitchValueInterpolator que varía su índice de nodo Switch fuente entre 0 y n-1, donde n es es el número de hijos del nodo Switch fuente.

SwitchValueInterpolator(Alpha alpha, Switch target, int firstChildIndex,
int lastChildIndex)

Construye un SwitchValueInterpolator que varía el índice de nodo entre los dos valores proporcionados.

Lista Parcial de Métodos de **SwitchValueInterpolator**

Cada uno de estos métodos tiene un correspondiente método get sin parámetros que devuelve un valor del tipo correspondiente al parámetro del método set.

void setFirstChildIndex(int firstIndex)

Selecciona el primer índice de hijo para este interpolator.

void setLastChildIndex(int lastIndex)

Selecciona el último índice de hijo para este interpolator.

void setTarget(Switch target)

Selecciona la fuente de este interpolator.

Switch

La clase **Switch** se muestra aquí porque se utiliza en **SwitchValueInterpolator** (y más adelante en **DistanceLOD**). **Switch** se deriva de **Group** y es el padre de cero o más ramas del escenario gráfico. Un objeto **Switch** puede seleccionar cero, uno, o más, incluyendo todos, sus hijos que se renderizarán. Por supuesto un objeto **Switch** se puede utilizar sin un **Interpolator** u objeto **LOD**. El método más comúnmente usado es addChild() derivado de la clase **Group**.

Sumario de Constructores de **Switch**

Extiende: Group

El nodo **Switch** controla cuál de sus hijos será renderizado. Define un valor de selección de hijo (un valor switch) que puede seleccionar un sólo hijo, o puede seleccionar cero o más hijos usando una máscara para indicar que hijos son seleccionados para renderización.

Switch()

Construye un nodo Switch con los parámetros por defecto.

Switch(int whichChild)

Construye e inicializa un nodo Switch usando los índices de hijos especificados.

- CHILD_ALL todos los hijos son renderizados
- CHILD_MASK se usa la máscada childMask para seleccionar los hijos a renderizar

- CHILD_NONE no se renderiza ningún hijo.

Switch(int whichChild, java.util.BitSet childMask)
Construye e inicializa un nodo Switch usando la máscara y el índice de hijos especificados.

Lista Parcial de Métodos de **Switch**

Cada uno de estos métodos tiene un correspondiente método get sin parámetros que devuelve un valor del tipo correspondiente al parámetro del método set.

void setChildMask(java.util.BitSet childMask)
Selecciona la máscara de selección de hijos.
void setWhichChild(int child)
Selecciona el índice de selección de hijos que especifica qué hijo será renderizado.

Sumaro de Capacidades de **Switch**

ALLOW_SWITCH_READ | WRITE
Especifica que este nodo permite leer sus valores de selección de hijos de máscara y de hijo actual

TransparencyInterpolator
Un objeto **TransparencyInterpolator** tiene un NodeComponent **TransparencyAttributes** como su fuente. Este **Interpolator** cambia el valor de la transparencia del objeto fuente. Más de un objeto visual pueden compartir un objeto **TransparencyAttributes**. Así pues, un **TransparencyInterpolator** puede afectar a más de un objeto visual. También, debemos tener cuiddo con distintos modos de transparencia que pueden afectar el funcionamiento y el aspecto de representación del objeto visual. Puedes refierte a la especificación de Java 3d API para más información sobre la clase **TransparencyAttributes**.

Un riesgo potencial de programación de **TransparencyInterpolator** es no agregar al objeto fuente **TransparencyAttributes** el manojo de aspecto del objeto visual. Esto es similar a un problema potencial de **ColorInterpolator**.

Sumario de Constructores de **TransparencyInterpolator**

Extiende: Interpolator

Esta clase define un comportamiento que modifica la transpariencia de su objeto **TransparencyAttributes** fuente interpolandolo linearmente entre un par de valores de transpariencia especificados (usando el valor generado por el objeto **alpha**).

TransparencyInterpolator(Alpha alpha, TransparencyAttributes target)
Construye un transparencyInterpolator con una fuente especificada, una transpariencia mínima de 0.0f y una transpariencia máxima de 1.0f.
TransparencyInterpolator(Alpha alpha, TransparencyAttributes target,
float minimumTransparency, float maximumTransparency)
Construye un nuevo objeto transparencyInterpolator que varía al transparencia del **Material** fuente entre dos valores de transpariencia.

Sumario de Métodos de **TransparencyInterpolator**

Cada uno de estos métodos tiene un correspondiente método get sin parámetros que devuelve un valor del tipo correspondiente al parámetro del método set.

void setMaximumTransparency(float transparency)
Selecciona el valor máximo de transpariencia para este interpolator.
void setMinimumTransparency(float transparency)
Selecciona el valor mínimo de transpariencia para este interpolator.
void setTarget(TransparencyAttributes target)
Selecciona el objeto TransparencyAttributes fuente para este interpolator.

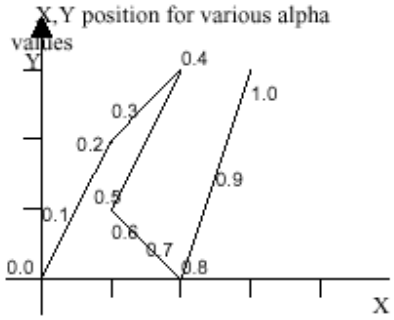
Clases PathInterpolator

Las clases **PathInterpolator** se diferencian de los otros **Interpolator** en que pueden guardar dos o más valores para la interpolación. El corazón de Java 3D proporciona las clases **PathInterpolator** para la interpolación de la posición, de la rotación, de la traslación y de la escala. La fuente de un objeto **PathInterpolator** es un objeto **TransformGroup** que cambia la posición, la orientación, y la escala, según sea apropiado, para sus objetos hijos.

Los objetos **PathInterpolator** almacenan un conjunto de valores, o de los nudos, que son utilizados por parejas para la interpolación. El valor **alpha** determina que dos valores son utilizados. Los valores de nudo están en el rango de 0,0 a 1,0 inclusivos, que corresponde al rango de los valores del objeto **alpha**. El primer nudo debe tener un valor de 0,0 y el último nudo debe tener un valor de 1,0. Los nudos restantes se deben grabar en orden creciente en el objeto **PathInterpolator**. Los valores del nudo corresponden con los valores para las variables de los parámetros (por ejemplo, posición o rotación) usado en la interpolación. Hay un valor de parámetro especificado para cada valor del nudo. El nudo con el mayor valor igual o menor que el valor **alpha**, y el nudo siguiente, serán utilizados. Los nudos se especifican en orden, para que en los cambios del valor **alpha**, se utilicen parejas adyacentes.

El panel izquierdo de la Figura 5-11 muestra los valores del nudo para un **PathInterpolator** de posición. Para propósitos ilustrativos, solamente se utilizan las posiciones 2D. El panel central de la figura asocia la posición del objeto visual sobre los valores **alpha**, 0,0 a 1,0. El panel derecho de la figura muestra los valores del nudo usados para los distintos valores **alpha** de este ejemplo. La combinación de los valores del nudo y de los parámetros del **alpha** determinan la animación.

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



alpha value (a)	knots used
0.0	0
0.0 < a < 0.2	0, 1
0.2	1
0.2 < a < 0.4	1, 2
0.4	2
0.4 < a < 0.5	2, 3
0.5	3
0.5 < a < 0.8	3, 4
0.8	4
0.8 < a < 1.0	4, 5
1.0	5

Aplicación de Ejemplo de PathInterpolator

Usar un objeto **PathInterpolator** sigue la misma receta que otros objetos **Interpolator**. La única diferencia es el número de valores usados para inicializar el objeto **PathInterpolator**

1. crear el objeto fuente con las capacidades apropiadas
2. crear el objeto Alpha
3. crear arrays de nudos y otros valores
4. crear el objeto pathInterpolator que referencia al objeto Alpha, al objeto fuente, y al array de selecciones
5. añadir límites al objeto Interpolator
6. añadir el objeto pathInterpolator al escenario gráfico

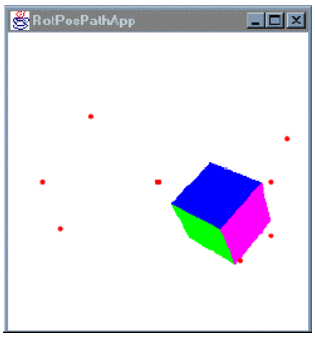
El programa del ejemplo RotPosPathApp.java utiliza un objeto **RotPosPathInterpolator** para animar un objeto **ColorCube** con un número de valores de posición y de rotación. El **RotPosPathInterpolator** graba conjuntos de rotaciones (como un array de Quat4f), de posiciones (como un array de Point3f), y de valores de nudo (como un array de float). El fragmento de código 5-2 muestra un extracto del ejemplo.

Fragmento de Código 5-2, un Fragmento del método CreateSceneGraph de RotPosPathApp.java

```
1.  public BranchGroup createSceneGraph() {
2.  BranchGroup objRoot = new BranchGroup();
3.
4.  TransformGroup target = new TransformGroup();
5.  Alpha alpha = new Alpha(-1, 10000);
6.  Transform3D axisOfRotPos = new Transform3D();
7.  float[] knots = {0.0f, 0.3f, 0.6f ,1.0f};
```

```
8.    Quat4f[] quats = new Quat4f[4];
9.    Point3f[] positions = new Point3f[4];
10.
11.   target.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
12.
13.   AxisAngle4f axis = new AxisAngle4f(1.0f,0.0f,0.0f,0.0f);
14.   axisOfRotPos.set(axis);
15.
16.   quats[0] = new Quat4f(0.0f, 1.0f, 1.0f, 0.0f);
17.   quats[1] = new Quat4f(1.0f, 0.0f, 0.0f, 0.0f);
18.   quats[2] = new Quat4f(0.0f, 1.0f, 0.0f, 0.0f);
19.
20.   positions[0]= new Point3f( 0.0f, 0.0f, -1.0f);
21.   positions[1]= new Point3f( 1.0f, -1.0f, -2.0f);
22.   positions[2]= new Point3f( -1.0f, 1.0f, -3.0f);
23.
24.   RotPosPathInterpolator rotPosPath = new RotPosPathInterpolator(
25.             alpha, target, axisOfRotPos, knots, quats, positions);
26.   rotPosPath.setSchedulingBounds(new BoundingSphere());
27.
28.   objRoot.addChild(target);
29.   objRoot.addChild(rotPosPath);
30.   target.addChild(new ColorCube(0.4));
31.
32.   return objRoot;
33. } // end of createSceneGraph method of RotPosPathApp
```

El fragmento de código 5-2 se basa en el método createSceneGraph de RotPosPathApp.java. La diferencia está en el número de nudos mostrados en el fragmento del código y los usados en el programa del ejemplo. RotPosPathApp.java define nueve nudos mientras que el fragmento de código 5-2 muestra solamente tres. La Figura 5-13 muestra una imagen de RotPosPathApp. En la imagen, un punto rojo se visualiza por cada uno de las nueve posiciones de nudo. Se reutiliza una de las posiciones, por eso en la figura se ven solo ocho puntos.



Cuando se ejecuta el programa del ejemplo **RotPosPathApp**, el **ColorCube** se mueve desde la posición de nudo a otra posición de nudo mientras que rota para alcanzar las distintas rotaciones de nudo. Como con todos los **Interpolator**, la animación que resulta depende de la combinación de los valores del **Interpolator** y de los parámetros **alpha** usados.

Según lo mencionado antes, hay una variedad de subclases de la clase de **PathInterpolator**. Además de estas subclases en el corazón Java 3D, hay pares de clases relacionadas en el paquete de utilidades. La clase **TCBPathSplineInterpolator** es una clase similar a **PathInterpolator**. Tiene una subclase en el paquete de utilidades.

En el ejemplo **RotPosPathApp**, la animación no parece natural debido a la combinación de las posiciones de nudo elegidas. El **ColorCube** se mueve a cada posición de nudo especificada y tan pronto como se alcanza esa posición, el movimiento cambia repentinamente para alcanzar la posición siguiente. Esto no parece natural puesto que este tipo de acción no sucede en el mundo real donde todos los objetos tienen cierta inercia.

TCBPathSplineInterpolator es una clase de utilidad que proporciona comportamiento y funciones similares a las de la clase **PathInterpolator**, pero suaviza el camino del objeto visual basándose en un tira en la posición del nudo. El camino de la tira imita el movimiento de objetos del mundo real. En el camino del movimiento de la tira, el objeto visual puede no pasar por todas las (o cualesquiera) posiciones especificadas de nudo. En el subdirectorio jdk1.2/demo/java3d/SplineAnim de la distribución de Java 3D puedes encontrar un ejemplo de utilización de la clase Splineanim.java.

PathInterpolator

PathInterpolator es una clase abstracta que proporciona el interfaz y las funciones básicas a sus subclases. Los objetos **PathInterpolator** graban los valores de nudo y calculan el índice de los valores de nudo que se utilizarán basándose en el valor **alpha** actual.

PathInterpolator

Extiende: Interpolator

Subclases Directas Conocidas: PositionPathInterpolator, RotationPathInterpolator,

RotPosPathInterpolator, RotPosScalePathInterpolator

Esta clase abstracta define la clase base para todos los **PathInterpolator**. Las subclases tienen acceso al método para calcular currentInterpolationValue dando el tiempo actual y el valor **alpha**. El método también calcula el currentKnotIndex, que se basa en el currentInterpolationValue. El currentInterpolationValue se calcula interpolandolo linearmente entre una serie de nudos predefinidos (que usan el valor generado por el objeto **alpha** especificado).

El primer nudo debe tener un valor de 0,0 y el último nudo debe tener un valor de 1,0. Un nudo intermedio con el índice k debe tener un valor terminantemente mayor que cualquier nudo con índice menor que k.

Lista Parcial de Métodos de **PathInterpolator**

int getArrayLengths()

Este método recupera la longitud del array de nudos.

void setKnot(int index, float knot)

Este métodos selecciona el nudo en el índice especificado para este interpolator.

RotPosPathInterpolator

Un objeto **RotPosPathInterpolator** varía la rotación y la posición de un objeto visual basándose en un conjunto de valores de nudo. El constructor es el más importante de las características del API de esta clase. En el constructor todos los valores y objetos relacionados se especifican. Debemos tener cuidado de que cada uno de los arrays debe ser de la misma longitud en este y en todos los objetos **PathInterpolator**.

Sumario de Constructores de **RotPosPathInterpolator**

Extiende: PathInterpolator

Esta clase define un comportamiento que modifica los componentes rotacionales y de translación de su **TransformGroup** fuente interpolandolo linearmente entre una serie de pares predefinidos de nudo/position y de nudo/orientation (que usan el valor generado por el objeto **alpha** especificado). La posición y la orientación interpoladas se utilizan para generar una transformación en el sistema de coordenadas local de este **Interpolator**.

RotPosPathInterpolator(Alpha alpha, TransformGroup target,
Transform3D axisOfRotPos, float[] knots, Quat4f[] quats,
Point3f[] positions)

Construye un nuevo interpolador que varía la rotación y la traslación del TransformGroup fuente.

Sumario de Métodos de **RotPosPathInterpolator**

void setAxisOfRotPos(Transform3D axisOfRotPos)

Selecciona el valor del eje de rotación para este interpolator.

void setPosition(int index, Point3f position)

Selecciona la posición del índice especificado para este interpolator.

void setQuat(int index, Quat4f quat)

Selecciona el "quaternion" en el índice especificaco para este interpolator.

void setTarget(TransformGroup target)

Selecciona el TransformGroup fuente para este interpolator.

La Clase Billboard

El término "cartelera" (Billboard) usado en el contexto de los gráficos de ordenador se refiere a la técnica de rotar automáticamente un objeto visual plano para que esté siempre de frente hacia el espectador. La motivación original para el comportamiento de cartelera era permitir el uso de un plano texturado como reemplazo de bajo costo para la geometría compleja. El comportamiento de cartelera todavía se utiliza comúnmente para esta aplicación, pero también se utiliza para otros propósitos, tales como mantener la información textual visible desde cualquier ángulo el ambiente virtual. En Java 3D, la técnica de cartelera se implementa en una subclase de la clase **Behavior**, de ahí la frase "comportamiento de cartelera" usado en la literatura de Java 3D.
La aplicación clásica de ejemplo del comportamiento de cartelera es representar árboles como texturas 2D. Por supuesto, si las texturas se orientan estáticamente, cuando el espectador se mueve, la naturaleza 2D de las texturas se revela. Sin embargo, si los árboles se reorientan de tal forma que siempre están paralelos a su superficie normal, aparecen como objetos 3D. Esto es especialmente cierto si los árboles están en el fondo de una escena o vistos en la distancia.

Usar un Objeto Billboard

El comportamiento **Billboard** funciona con los árboles porque los árboles parecen básicamente iguales cuando se ven de frente, de la parte posterior, o de cualquier ángulo. Puesto que el comportamiento **Billboard** hace que un objeto visual parezca exactamente igual cuando se ve desde cualquier ángulo, es apropiado utilizar **Billboard**s y las imágenes 2D para representar los objetos 3D geométricos que son simétricos sobre el eje Y tal como edificios cilíndricos, silos de grano, torres de agua, o cualquier objeto cilíndrico. El comportamiento **Billboard** también se puede

utilizar para objetos no simétricos cuando se ven desde suficiente distancia como para ocultar los detalles 2D.

Usar un objeto **Billboard** es similar a usar un **Interpolator** excepto en que no hay un objeto **alpha** para conducir la animación. La animación del objeto **Billboard** la dirige su posición relativa al espectador en el mundo virtual. Abajo podemos ver una receta para el uso **Billboard**.

1. crear un objeto TransformGroup con la capacidad ALLOW_TRANSFORM_WRITE
2. crear un objeto Billboard que referencie al objeto fuente TransformGroup
3. suministrar límites para el objeto Billboard
4. ensamblar el escenario gráfico

Riesgos de Programación de Billboard

Aunque el uso de un objeto **Billboard** es directo, hay un par de potenciales errores de programación. La primera cosa a observar es que el **TransformGroup** fuente se limpia cada vez que se actualiza. Por lo tanto, este **TransformGroup** no se puede utilizar para posicionar el objeto visual. Si intentamos utilizar la fuente para el posicionamiento, el **Billboard** funcionara, pero en la primera actualización de la rotación, se perderá la información de la posición en la fuente y el objeto visual aparecerá en el origen.

Sin la capacidad de **ALLOW_TRANSFORM_WRITE** seleccionada para la fuente, obtendremos un error de tiempo de ejecución. También, si no se fijan los límites, o no se hace correctamente, el objeto de **Billboard** no animará el objeto visual. Los límites se especifican típicamente por un **BoundingSphere** con un radio lo bastante grande para incluir el objeto visual. Justo igual otros objetos **behavior**, dejar el objeto **Billboard** fuera del escenario gáfico lo eliminará del mundo virtual sin ningún error o la alerta.

Hay un problema con la clase **Billboard** que no puede ser superado. En aplicaciones con más de una vista, cada objeto **Billboard** animará correctamente solamente una de las vistas. Esto es una limitación en el diseño de Java 3D y será tratada en una versión posterior.

Programa de Ejemplo de Billboard

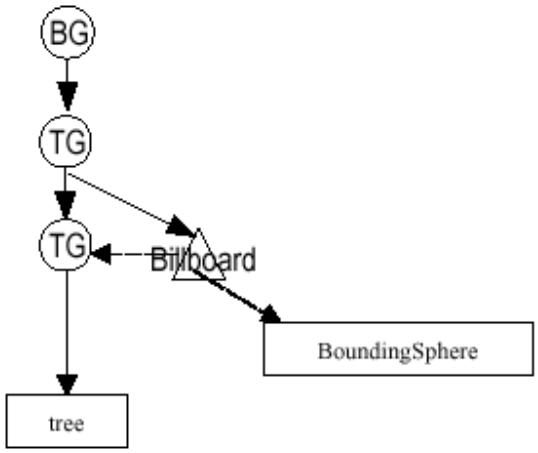
El programa del ejemplo BillboardApp.java crea un mundo virtual con árboles de un comportamiento **Billboard**. Aunque los árboles se crean a lo bruto (desde las aspas de un ventilador) no parecen como objetos 2D en el fondo.
Hay objetos **TransformGroup** para cada árbol en este ejemplo. Un **TransformGroup**, TGT, traslada simplemente el árbol a la posición para la aplicación. Los Transform TGT no se modifican en tiempo de ejecución. El segundo **TransformGroup**, TGR, proporciona la rotación para el árbol. El TGR es la fuente de **Billboard**.

Fragmento de Código 5-3, Extracto del método createSceneGraph de BillboardApp.java.

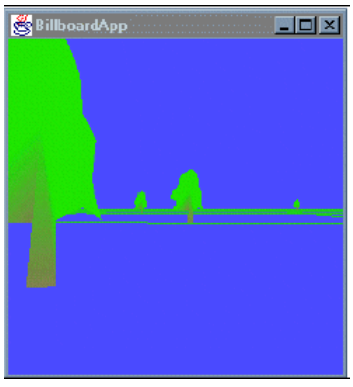
```
1.  public BranchGroup createSceneGraph(SimpleUniverse su) {
2.  BranchGroup objRoot = new BranchGroup();
3.
4.  Vector3f translate = new Vector3f();
5.  Transform3D T3D = new Transform3D();
6.  TransformGroup TGT = new TransformGroup();
7.  TransformGroup TGR = new TransformGroup();
8.  Billboard billboard = null;
9.  BoundingSphere bSphere = new BoundingSphere();
10.
11. translate.set(new Point3f(1.0f, 1.0f, 0.0f));
12. T3D.setTranslation(translate);
13. TGT.set(T3D);
14.
15. // set up for billboard behavior
16. TGR.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
17. billboard = new Billboard(TGR);
18. billboard.setSchedulingBounds(bSphere);
19.
20. // assemble scene graph
21. objRoot.addChild(TGT);
22. objRoot.addChild(billboard);
```

```
23.  TGT.addChild(TGR);
24.  TGR.addChild(createTree());
25.
26.  // add KeyNavigatorBehavior(vpTrans) code removed;
27.
28.  return objRoot;
29.  } // end of CreateSceneGraph method of BillboardApp
```

La Figura 5-15 muestra el diagrama del escenario gráfico de los objetos ensamblados en el Fragmento de Código 5-3.



La Figura 5-16 muestra una imagen renderizada del programa del ejemplo BillboardApp.java. El Fragmento de Código 5-3 muestra el código para colocar un árbol animado **Billboard** en un mundo virtual. El programa de BillboardApp coloca varios árboles en el paisaje virtual que es por lo que se ven cuatro árboles en la Figura 5-16.



El programa de ejemplo **BillboardApp** proporciona un **KeyNavigatorBehavior** de modo que el espectador pueda moverse alrededor y observar los árboles desde varias posiciones y orientaciones.

El API Billboard

El ejemplo muestra el uso del valor por defecto del objeto **Billboard**, que es rotar sobre un eje. En este modo de valor por defecto, el objeto visual se rotará solamente sobre el eje Y. Así pues, si los árboles en el programa **BillboardApp** se ven desde arriba o desde abajo, su geometría 2D sería revelada.
El modo alternativo es rotar alrededor de un punto. En este modo, la rotación sería alrededor de un punto tal que el objeto visual se vea siempre orthogonalmente desde cualquier posición de la vista. Es decir nunca será obvio que el objeto visual es de dos dimensiones. Una aplicación posible es representar la luna, u otros objetos esféricos distantes como círculos. Los objetos esféricos aparecen como círculos cuando se ven desde cualquier ángulo.

Sumario de Constructores de **Billboard**

Extiende: Behavior

El nodo de comportamiento **Billboard** funciona sobre el nodo **TransformGroup** para hacer que el eje local +z del **TransformGroup** apunte a la posición del ojo del espectador. Esto se hace sin importar las transformaciones sobre el nodo **TransformGroup** especificado en el escenario gráfico. Los nodos **Billboard** proporcionan a la mayoría de las ventajas para los objetos complejos, ásperos-simétricos. Un uso típico puede consistir en un cuadrilátero texturado con la imagen de un árbol.

Billboard()
Construye un nodo Billboard con los parámetros por defecto: mode = ROTATE_ABOUT_AXIS, axis =(0,1,0).
Billboard(TransformGroup tg)

Construye un nodo Billboard con los parámetros por defecto que opera sobre el nodo TransformGroup especificado.

Billboard(TransformGroup tg, int mode, Vector3f axis)

Construye un nodo Billboard con el eje y el modo especificados que opera sobre el nodo TransformGroup especificado.

Puedes ver el método setMode() para una explicación del parámetro mode.

Billboard(TransformGroup tg, int mode, Point3f point)

Construye un nodo Billboard con el punto de rotación especificado y el modo que opera sobre el nodo TransformGroup especificado.

Puedes ver el método setMode() para una explicación del parámetro mode.

Lista Parcial de Métodos de **Billboard**

void setAlignmentAxis(Vector3f axis)

Selecciona el eje de alineamiento.

void setAlignmentAxis(float x, float y, float z)

Selecciona el eje de alineamiento.

void setAlignmentMode(int mode)

Selecciona el modo de alineamiento, donde mode es uno de:

- ROTATE_ABOUT_AXIS - Especifica qué rotación debería sobre el eje especificado.
- ROTATE_ABOUT_POINT - Especifica qué rotación denería ser sobre el punto especificado y cual de los ejes Y del hijo debería corresponder con el eje Y de la vista del objeto.

void setRotationPoint(Point3f point)

Selecciona el punto de rotación.

void setRotationPoint(float x, float y, float z)

Selecciona el punto de rotación.

void setTarget(TransformGroup tg)

Selecciona el objeto TransformGroup fuente para este objeto Billboard.

Animaciones de Nivel de Detalle (LOD)

El nivel del detalle (**LOD**) es un término general para una técnica que varía la cantidad de detalle en un objeto visual basándose en un cierto valor del mundo virtual. La aplicación típica es variar el nivel del detalle basándose en la distancia al espectador. Cuanto mayor sea la distancia a un objeto visual, menos detalles aparecerán en la representación. Así pues, la reducción de la complejidad del objeto visual puede no afectar el resultado visual. Sin embargo, disminuir la cantidad de detalle en el objeto visual cuando está lejos del espectador reduce la cantidad de cálculo de renderizado. Si se hace bien, el ahorro de cálculos es significativo y se pueden hacer sin la pérdida de contenido visual.
La clase **DistanceLOD** proporciona comportamiento **LOD** basado en la distancia al espectador. Otras aplicaciones posibles de **LOD** incluyen variar el nivel del detalle basándose en la velocidad de representación (marcos por segundo) con la esperanza de mantener un ratio mínimo de marcos, la velocidad del objeto visual, o el nivel del detalle se podría controlar por las configuraciones del usuario.
Todo objeto **LOD** tiene uno o más objetos **Switch** como fuente. Según lo mencionado antes, un objeto **Switch** es un **group** especial que incluye cero, uno, o más, de sus hijos en el escenario gráfico para renderizar. Con un objeto **DistanceLOD**, la selección del hijo del objeto **Switch** fuente se controla por la distancia del objeto **DistanceLOD** a la vista basada en un conjunto de distancias de umbral.
Las distancias de umbral se especifican en un arrray que comienza con la distancia máxima que utilizará el primer hijo del **Switch** fuente. El primer hijo típicamente es el objeto visual más detallado. Cuando la distancia del objeto **DistanceLOD** a la vista es mayor que este primer umbral, se utiliza el segundo hijo de **Switch**. Cada umbral siguiente de la distancia debe ser mayor que el anterior y especifica la distancia en la cual se utiliza el hijo siguiente del **Switch** fuente.
Si se agrega más de un **Switch** como fuente del objeto **LOD**, cada **Switch** fuente se utiliza en paralelo. Es decir, seleccionan al hijo del mismo índice simultáneamente para cada uno de los **Switch** fuente. De esta manera, un objeto

visual complejo puede ser representado por objetos geométricos múltiples que son hijos de diversos nodos **Switch**.

- **Usar un Objeto DistanceLOD**

Usar un objeto **DistanceLOD** es similar a usar un **Interpolator** excepto en que no hay un objeto **alpha** para conducir la animación. La animación del objeto **LOD** la dirige su distancia relativa a la vista en el mundo virtual; de esta manera usar un objeto **DistanceLOD** es muy similar a usar un objeto **Billboard**. Usar un objeto de **DistanceLOD** también requiere fijar las distancias del umbral. Abajo tenemos uns receta que nos muestra los pasos para usar un **LOD**.

1. crear un objeto Switch fuente con la capacidad ALLOW_SWITCH_WRITE
2. crear un array de distancias de umbral para el objeto DistanceLOD
3. crear el objeto DistanceLOD usando el array de distancias de umbral
4. seleccionar el objeto Switch fuente para el objeto DistanceLOD
5. suministrar límites para el objeto DistanceLOD
6. ensamblar el escenario gráfico, incluyendo la adición de hijos al objeto Switch fuente

Riesgos de Programación de LOD

Incluso aunque que el uso de un objeto **LOD** es directo, hay un par de potenciales errores de programación. El error más común es no incluir el objeto **Switch** fuente en el escenario gráfico. Fijar el objeto **Switch** como la fuente del objeto **DistanceLOD** no lo incluye automáticamente en el escenario gráfico.
Sin la capacidad **ALLOW_SWITCH_WRITE** fijada para el objeto **Switch** fuente, se generará un error en tiempo de ejecución. También, si no se fijan los límites, o no se hace correctamente, el objeto **LOD** no animará el objeto visual. Los límites se especifican típicamente con un **BoundingSphere** con un radio lo bastante grande como para incluir el objeto visual. Igual que con otros objetos **behavior**, dejar el objeto **LOD** fuera del escenario gráfico lo eliminará del mundo virtual sin ningún error o alerta.

Hay un problema con las clases de **LOD** que no puede ser superada. Igual que con aplicaciones de **Billboard**, en las aplicaciones que tienen más de una vista, el objeto **LOD** animará correctamente sólo una de ellas.

Ejemplo de uso de DistanceLOD

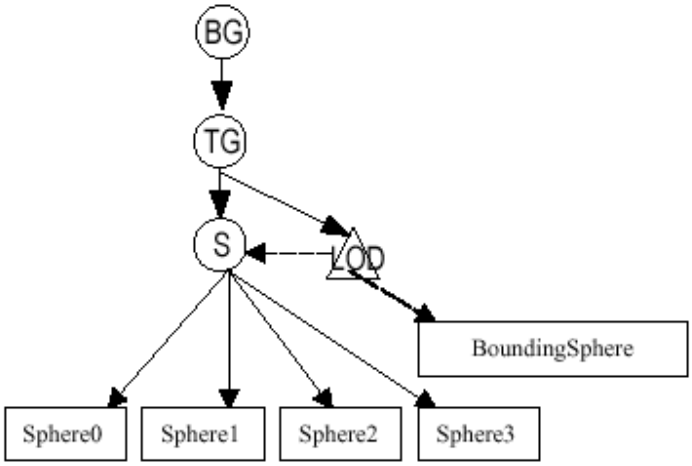
El Fragmento de Código 5-4 muestra un extracto del método createSceneGraph de DistanceLODApp.java.

Fragmento de Código 5-4, Extracto del método createSceneGraph en DistanceLODApp.

```
1.  public BranchGroup createSceneGraph() {
2.  BranchGroup objRoot = new BranchGroup();
3.  BoundingSphere bounds = new BoundingSphere();
4.
5.  // create target TransformGroup with Capabilities
6.  TransformGroup objMove = new TransformGroup();
7.
8.  // create DistanceLOD target object
9.  Switch targetSwitch = new Switch();
10. targetSwitch.setCapability(Switch.ALLOW_SWITCH_WRITE);
11.
12. // add visual objects to the target switch
13. targetSwitch.addChild(new Sphere(.40f, 0, 25));
14. targetSwitch.addChild(new Sphere(.40f, 0, 15));
15. targetSwitch.addChild(new Sphere(.40f, 0, 10));
16. targetSwitch.addChild(new Sphere(.40f, 0, 4));
17.
18. // create DistanceLOD object
19. float[] distances = { 5.0f, 10.0f, 20.0f};
20. DistanceLOD dLOD = new DistanceLOD(distances, new Point3f());
21. dLOD.addSwitch(targetSwitch);
22. dLOD.setSchedulingBounds(bounds);
23.
24. // assemble scene graph
25. objRoot.addChild(objMove);
```

```
26.   objMove.addChild(dLOD); // make the bounds move with object
27.   objMove.addChild(targetSwitch); // must add switch to scene graph
28.
29.   return objRoot;
30. } // end of CreateSceneGraph method of DistanceLODApp
```

La Figura 5-18 muestra el diagrama del escenario gráfico creado en el fragmento de código 5-4. Observa que el objeto **Switch** fuente es hijo de un objeto **TransformGroup** y es referido por el objeto **DistanceLOD**. Ambos lazos son necesarios.

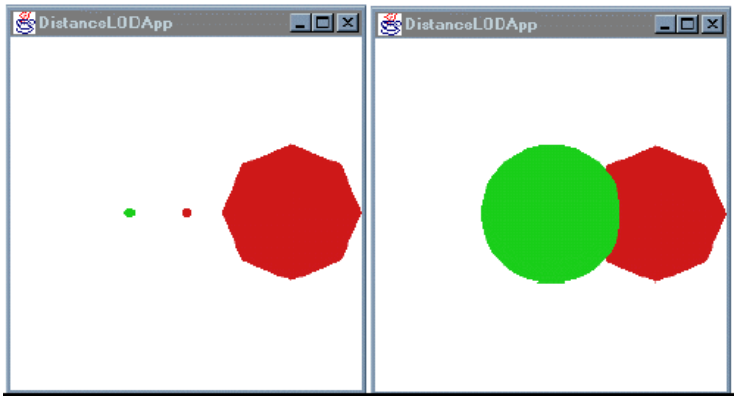


La Figura 5-19 muestra dos escenas renderizadas por **DistanceLODApp**. Cada escena tiene dos esferas estáticas y una esfera que se mueve. (en la escena derecha, se oculta la esfera de la izquierda.) La esfera móvil se representa por un objeto **DistanceLOD** con cuatro esferas de distinta complejidad geométrica. La esfera pequeña verde es la esfera más detallada usada por el objeto **DistanceLOD** en la distancia máxima. La esfera grande roja es la menos detallada del objeto **DistanceLOD** en la distancia mínima. Las dos esferas estáticas se incluyen para propósitos de comparación.

En esta aplicación el objeto **DistanceLOD** se representa por diversas esferas de color para ilustrar la conmutación. Cada objeto visual usado por un objeto **LOD** debería parecer tan normal como sea apropiado.

Se utiliza un **PositionInterpolator** para mover el objeto **DistanceLOD** hacia adelante y hacia atrás en la escena. Cuanto más lejos de la vista se mueva el objeto

DistanceLOD, más cambiaran los objetos visuales. Sin el cambio del color en esta aplicación, no sería fácil decir cuándo ocurre la conmutación.



En la práctica, normalmente necesitaremos experimentar con las distancias de umbral y las distintas representaciones para conseguir los efectos visuales deseados.

El API DistanceLOD

En Java 3D, la clase **LOD** proporciona las funciones básicas para todas las aplicaciones **LOD**. La clase **DistanceLOD** extiende la clase **LOD** para agregar cálculos '**Switch** según la distancia al espectador '. Varios métodos de la clase **LOD** son necesarios para el uso de un objeto **DistanceLOD**.

Sumario de Constructores de **DistanceLOD**

Esta clase define un nodo de comportamiento basado en la distancia que funciona sobre un nodo **Switch** para seleccionar uno de los hijos de ese nodo **Switch** basándose en la distancia de este nodo **LOD** hasta el espectador. Un array de n elementos que aumenta monotónicamente según la distancia especificada, tal que distances[0] está asociada al nivel más alto de detalle y distances[n-1] está asociado al nivel más bajo de detalle. De acuerdo con la distancia real desde el espectador a este nodo **DistanceLOD**, estos valores de la distancia de n [0, n-1] seleccionan entre de los niveles n+1 de detalle [0, n]. Si d es la distancia del espectador al nodo de **LOD**, entonces la ecuación para determinar el nivel del

detalle (hijo del nodo **Switch**) seleccionado es:

0, if d <= distances[0]

i, if distances[i-1] < d <= distances[i]

n, if d > distances[n-1]

Observa que tanto la posición como el array de distancias están especificados en el sistema de coordenadas local de este nodo.

DistanceLOD()

Construye e inicializa un nodo DistanceLOD con los valores por defecto.

DistanceLOD(float[] distances)

Construye e inicializa un nodo DistanceLOD con el array de distancias especificado y la posición por defecto de (0,0,0).

DistanceLOD(float[] distances, Point3f position)

Construye e inicializa un nodo DistanceLOD con el array de distancias y la posición especificados.

Sumario de Métodos de **DistanceLOD**

int numDistances()

Devuelve un contador del número de parámetros de distancia del LOD.

void setDistance(int whichDistance, double distance)

Selecciona una distancia de umbral particular.

void setPosition(Point3f position)

Selecciona la posición de este nodo LOD.

- **API de LOD (Level of Detail)**

Como clase abstracta, la clase **LOD** no se utiliza directamente en los programas de Java 3D. Los métodos de la clase **LOD** se utilizan para manejar el objeto **Switch** fuente de un objeto **DistanceLOD**. También, otras aplicaciones de **LOD** podían crearse extendiendo esta clase según sea apropiado.

Sumario de Constructores de **LOD**

Un nodo hoja **LOD** es una clase de comportamiento abstracto que opera sobre

una lista de nodos **Switch** para seleccionar uno de los hijos de los nodos **Switch**. La clase **LOD** se extiende para implentar distintos crierios de selección.

LOD()
Construye e inicializa un nodo LOD.

Sumario de Métodos de **LOD**

void addSwitch(Switch switchNode)
Añade el switch especificado a la lista de switches de este nodo LOD.
java.util.Enumeration getAllSwitches()
Devuelve el objeto enumeration de todos los switches.
void insertSwitch(Switch switchNode, int index)
Inserta el nodo switch especificado en el índice especificado.
int numSwitches()
Devuelve un contador de switches de este LOD.
void removeSwitch(int index)
Elimina el nodo switch del índice especificado.
void setSwitch(Switch switchNode, int index)
Reemplaza el nodo switch especificado con el nodo switch proporcionado.

Morph

Las clases **Interpolator** modifican varios atributos visuales del mundo virtual. Sin embargo, no hay un **Interpolator** para cambiar la geometría de un objeto visual. Esto es exactamente lo que hace la clase **Morph**. Un objeto **Morph** crea la geometría para un objeto visual con la interpolación de un conjunto de objetos **GeometryArray**. De esta forma la clase **Morph** es como las clases **Interpolator**. Pero, **Morph** no es un **Interpolator**; no es ni siquiera una extensión de la clase **Behavior**. La clase **Morph** extiende la clase **Node**.
El Capítulo 4 explica que todos los cambios en una escena viva o los objetos en un gráfico vivo de la escena se hacen normalmente con el método processStimulus de los objetos **Behavior**. Puesto que no hay una clase específica de

comportamiento para el uso con un objeto **Morph**, se debe escribir un clase personalizada para aplicarla con **Morph**. Si la clase **Morph** se considera una clase de animación o de interacción depende del estímulo para el comportamiento que diriga el objeto **Morph**.
Los objetos **Morph** se pueden utilizar para convertir las pirámides en cubos, gatos en perros, o cambiar cualquier geometría en cualquier otra geometría. La única limitación es que los objetos de la geometría usados para la interpolación sean de la misma clase, una subclase de **GeometryArray**, y con el mismo número de vértices. La restricción en el número de vértices no es un límite como parece a primera vista. En el Java 3D se distibuye un programa de ejemplo que convierte una pirámide en un cubo, Pyramid2Cube.java.
Los objetos **Morph** también se pueden utilizar para animar un objeto visual (por ejemplo., hacer que una persona corra). En el API Java 3D también puedes encontrar un programa que anima una mano, Morphing.java. Un tercer ejemplo de **Morph** que hace caminar a una figura de alambre es el tema de la sección siguiente.

- **Usar un Objeto Morph**

Entender el uso del objeto de **Morph** requiere saber cómo funcionan los objetos **Morph**. Afortunadamente, un objeto **Morph** no es muy complejo. Un objeto **Morph** graba un array de objetos **GeometryArray**. Podemos recordar del Capítulo 2 que **GeometryArray** es la superclase de **TriangleArray**, de **QuadStripArray**, de **IndexedLineStripArray**, y de **TriangleFanArray** (entre otros muchos).
El **GeometryArray** individual define completamente una especificación geométrica completa para el objeto visual incluyendo color, superficies normales, y coordenadas de la textura. Los objetos **GeometryArray** se pueden imaginar como marcos de una animación, o más correctamente, como las constantes en una ecuación para crear un nuevo objeto **GeometryArray**.
Además del array de objetos **GeometryArray**, un objeto **Morph** tiene un array de valores del peso- éstas son las variables en la ecuación. Usando el

GeometryArray y los pesos, un objeto **Morph** construye un nuevo objeto geometría usando el promedio de las coordenadas, el color, las superficies normales, y la información de coordenadas de la textura de los objetos de **GeometryArray**. Modificar los pesos cambia la geometría resultante.
Todo lo que se requiere para utilizar un objeto **Morph** es crear el array de objetos **GeometryArray** y fijar los valores de carga. Abajo podemos ver una receta para usar un objeto **Morph**.

1. crear un array de objetos GeometryArray
2. crear un objeto Morph con ALLOW_WEIGHTS_WRITE
3. ensamblar el escenario gráfico, incluyendo la adición de los hijos de los objetos Switch fuentes

Como se puede ver, usar un objeto **Morph** no es duro; sin embargo, estos pasos de la receta no proporcionan ni animación ni interacción. La animación o la interacción se proporciona a través de un objeto **Behavior**. Por lo tanto, usar un objeto **Morph** significa generalmente escribir una clase **Behavior**.
Un objeto **Morph** se puede referir a un paquete de aspecto. El manojo de aspecto se utiliza con el objeto **GeometryArray** creado por el objeto **Morph**. Debemos tener cuidado ya que el objeto **Morph** crea siempre un objeto **GeometryArray** con colores-por-vertice. Por consiguiente, se ignorarán las especificaciones de **ColoringAttributes** y de color difuso de **Material**. Puedes ver el Capítulo 6 para más información sobre colorear y sombrear objetos visuales.

Riesgos de Programación de Morph

Incluso tan simple como es el uso de **Morph**, hay un riesgo potencial de programación asociado (no mencionado todavía). Pesos que no suman 1,0 resultan en un error en tiempo de ejecución. Ya hemos mencionado la limitación del aspecto.

- **Ejemplo de Aplicación Morph: Walking**

Esta aplicación de **Morph** utiliza un objeto **Behavior** personalizado para proporcionar la animación. El primer paso de la receta es escribir el comportamiento personalizado.

En un comportamiento usado para animar un objeto **Morph**, el método processStimulus cambia los pesos del objeto **Morph**. Este proceso es solo tan complejo como necesario para alcanzar el efecto deseado de la animación o de la interacción. En este programa, el método processStimulus fija los valores de los pesos basándose en el valor de un objeto **alpha**. Esto sucede en cada marco de renderizado donde la condición del disparador se ha cumplido. El Fragmento de Código 5-5 demuestra el código para el comportamiento de personalizado del programa MorphApp.java. En este código, solamente el método processStimulus es interesante.

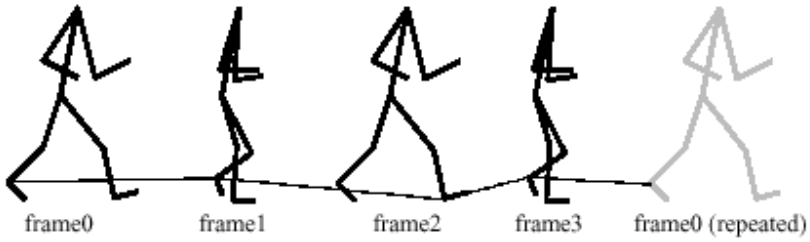
Fragmento de Código 5-5, Clase MorphBehavior de MorphApp.

```
1.  public class MorphBehavior extends Behavior{
2.
3.  private Morph targetMorph;
4.  private Alpha alpha;
5.  // the following two members are here for efficiency
6.  private double[] weights = {0, 0, 0, 0};
7.  private WakeupCondition trigger = new WakeupOnElapsedFrames(0);
8.
9.  // create MorphBehavior
10. MorphBehavior(Morph targetMorph, Alpha alpha){
11. this.targetMorph = targetMorph;
12. this.alpha = alpha;
13. }
14.
15. public void initialize(){
16. // set initial wakeup condition
17. this.wakeupOn(trigger);
18. }
19.
20. public void processStimulus(Enumeration criteria){
```

```
21.  // don't need to decode event since there is only one trigger
22.  weights[0] = 0; weights[1] = 0; weights[2] = 0; weights[3] = 0;
23.
24.  float alphaValue = 4f * alpha.value(); // get alpha
25.  int alphaIndex = (int) alphaValue; // which Geom obj
26.  weights[alphaIndex] = (double) alphaValue - (double)alphaIndex;
27.  if(alphaIndex < 3) // which other obj
28.  weights[alphaIndex + 1] = 1.0 - weights[alphaIndex];
29.  else
30.  weights[0] = 1.0 - weights[alphaIndex];
31.
32.  targetMorph.setWeights(weights);
33.
34.  this.wakeupOn(trigger); // set next wakeup condition
35.  }
36. } // end of class MorphBehavior
```

La clase **MorphBehavior** crea una animación de marcos usando dos objetos **GeometryArray** al mismo tiempo en un modelo cíclico. Esta clase es conveniente para cualquier animación de **morph** de cuatro marcos y se puede modificar fácilmente para acomodar otro número de marcos.

Con el comportamiento personalizado escrito, todo que lo resta es desarrollar los marcos para la animación. La Figura 5-21 muestra los dibujos a mano usados como los marcos para esta aplicación de ejemplo. Se podrían haber creado mejores marcos usando algún paquete 3D.



Las figuras negras pueden parecer dos marcos, cada uno repetido una vez, pero en realidad, son cuatro marcos únicos. La diferencia está en el orden que se especifican los vértcies.

El Fragmento de Código 5-6 presenta un extracto del método createSceneGraph. En este método se crean un objeto **MorphBehavior**, el objeto **alpha**, y un objeto **Morph**

y se ensamblan en el escenario gráfico. Se crean los objetos del marco **GeometryArray** usando algunos otros métodos (no mostrados aquí). El código completo lo tienes en MorphApp.java.

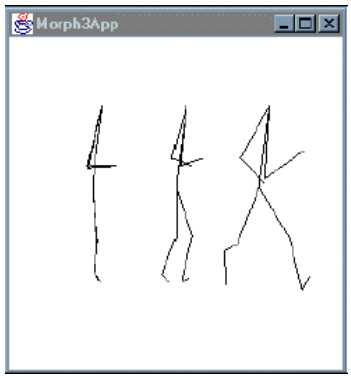
Fragmento de Código 5-6, un extracto del método createSceneGraph de MorphApp.

```
1.  public BranchGroup createSceneGraph() {
2.  // Create the root of the branch graph
3.  BranchGroup objRoot = new BranchGroup();
4.
5.  Transform3D t3d = new Transform3D();
6.  t3d.set(new Vector3f(0f, -0.5f, 0f));
7.  TransformGroup translate = new TransformGroup(t3d);
8.
9.  // create GeometryArray[] (array of GeometryArray objects)
10. GeometryArray[] geomArray = new GeometryArray[4];
11. geomArray[0] = createGeomArray0();
12. geomArray[1] = createGeomArray1();
13. geomArray[2] = createGeomArray2();
14. geomArray[3] = createGeomArray3();
15.
16. // create morph object
17. Morph morphObj = new Morph(geomArray);
18. morphObj.setCapability(Morph.ALLOW_WEIGHTS_WRITE);
19.
20. // create alpha object
21. Alpha alpha = new Alpha(-1, 2000); // continuous 2 sec. period
22. alpha.setIncreasingAlphaRampDuration(100);
23.
24. // create morph driving behavior
25. MorphBehavior morphBehav = new MorphBehavior(morphObj, alpha);
26. morphBehav.setSchedulingBounds(new BoundingSphere());
27.
28. //assemble scene graph
29. objRoot.addChild(translate);
30. translate.addChild(morphObj);
31. objRoot.addChild(morphBehav);
```

```
32.
33.   return objRoot;
34. } // end of CreateSceneGraph method of MorphApp
```

Es interesante observar que son posibles varias animaciones con los marcos creados para esta aplicación del ejemplo con diversas clases de comportamiento. La Figura 5-22 muestra una escena renderizada por Morph3App.

En este programa, otras tres clases de comportamiento crean animaciones basadas en alguno, o todos los objetos **GeometryArray** de **MorphApp**. Se llaman (de izquierda a derecha en la figura) "In Place", "Tango", y "Broken". No todas las animaciones son buenas. Por supuesto, para apreciar de verdad las animaciones, tenemos que ejecutar el programa.



El API Morph

Con la simplicidad de uso de la receta anterior, podríamos esperar un API sencillo -- y así es.

Sumario de Constructores de **Morph**

Extiende: Node

Los objetos **Morph** crean un nuevo objeto **GeometryArray** usando el promedio de peso de los objetos **GeometryArray**. Si se proporciona un objeto de apariencia, se utiliza con la geometría resultante. Los pesos se especifican con el método setWeights. Un objeto **Morph** se utiliza generalmente con un objeto **Behavior** personalizado para ajustar los pesos en el tiempo de ejecución para proporcionar

la animación (o la interacción).

Morph(GeometryArray[] geometryArrays)

Construye e inicializa un objeto Morph con el array de objetos GeometryArray especificado y un objeto Appearance null.

Morph(GeometryArray[] geometryArrays, Appearance appearance)

Construye e inicializa un objeto Morph con el array de objetos GeometryArray especificado y el objeto de apariencia especificado.

Lista Parcial de Métodos de **Morph**

void setAppearance(Appearance appearance)

Selecciona el componente de apariencia de este nodo Morph.

void setGeometryArrays(GeometryArray[] geometryArrays)

Selecciona el componente geometryArrays de este nodo Morph.

void setWeights(double[] weights)

Selecciona el vector de pesos de ese nodo Morph.

Sumario de Capacidades de **Morph**

ALLOW_APPEARANCE_READ | WRITE

Especfica que el nodo permite el acceso de lectura/escritura a su información de apariencia.

ALLOW_GEOMETRY_ARRAY_READ | WRITE

Especfica que el nodo permite el acceso de lectura/escritura a su información de geometría.

ALLOW_WEIGHTS_READ | WRITE

Especfica que el nodo permite el acceso de lectura/escritura a su vector de pesos.

Iluminación en Java 3D

Este módulo presenta las técnicas para proporcionar detalles de los objetos visuales a través de sombras y texturas. Esta página explica el modelo de iluminación y cómo utilizar luces en Java 3D para conseguir sombras.
Java 3D sombrea los objetos visuales basándose en la combinación de sus características materiales y en las luces del universo virtual. El sombreado resulta de aplicar un modelo de iluminación a un objeto visual en presencia de fuentes de luz. La siguiente sección da una descripción del modelo de iluminación usado en el renderizador de Java 3D y cómo la luz interactúa con las características materiales para proporcionar sombreado. Cada una de las siguientes secciones explica las características relevantes del API Java 3D para el modelo de iluminación.

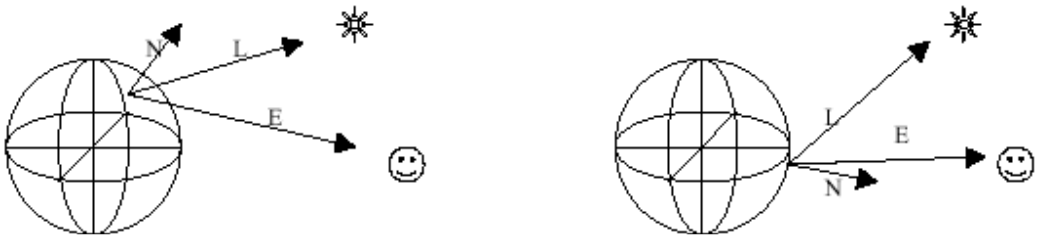
Sombreado en Java 3D

Sombrear objetos visuales en Java 3D depende de muchos factores. Esta sección proporciona una descripción abreviada del modelo de iluminación de Java 3D, del modelo de color, y del modelo de sombreado. La especificación del API Java 3D presenta una información más detallada sobre el modelo de iluminación de Java 3D. Como la mayor parte de la iluminación de Java 3D y del modelo de sombreado se basa en **OpenGL**, se puede encontrar más información en páginas sobre **OpenGL**.

Modelo de Iluminación

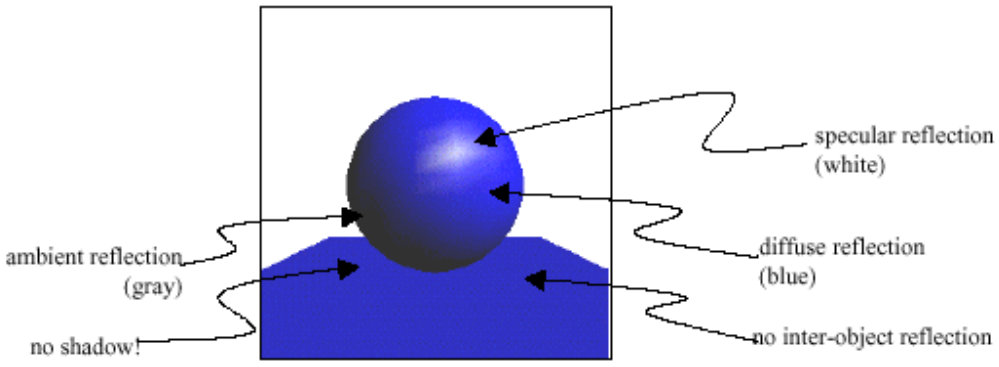
En el mundo real, los colores que percibimos son una combinación de las características físicas del objeto, de las características de las fuentes de luz, de las posiciones relativas de los objetos a las fuentes de luz, y del ángulo desde el cual se ve el objeto. Java 3D utiliza un modelo de iluminación para aproximar la física del mundo real. El resto de esta sección explica el modelo de iluminación de Java 3D en términos generales. La sección E.2 de la especificación del API Java 3D presenta las ecuaciones matemáticas para el modelo de iluminación Java 3D.

La ecuación del modelo de iluminación depende de tres vectores: la superficie normal (n), la dirección de la luz (l), y la dirección al ojo del espectador (e) además de las características materiales del objeto y de las características de la luz. La Figura 6-1 muestra los tres vectores para dos vértices de una superficie esférica. Los vectores para cada vértice pueden tener distintas direcciones dependiendo de especificidades de la escena. Cuando los vectores de la luz y del ojo varían, se cálculan en tiempo de ejecución. Por lo tanto, cada vértice de la esfera potencialmente se renderiza como una sombra diferente.



El modelo de iluminación incorpora tres tipos de reflexiones de luz del mundo real: ambiente, difuso, y especular. La reflexión ambiente resulta de la luz ambiente, luz constante de bajo nivel, en una escena. La reflexión difusa es la reflexión normal de una fuente de luz desde un objeto visual. Las reflexiones especulares son las reflexiones sobreiluminadas de una fuente de luz sobre un objeto, que ocurren en ciertas situaciones.

La Figura 6-2 muestra una esfera y un plano renderizados por Java 3D. Los tres tipos de reflexión se pueden ver en la esfera de la Figura 6-2. La parte más oscura de la esfera exhibe sólo la reflexión ambiente. El centro de la esfera está iluminado por la luz difusa y ambiente. Con una esfera azul y una luz blanca, la reflexión difusa es azul. La parte más brillante de la esfera es el resultado de la reflexión especular con reflexiones ambiente y difusa.



Ojo Local contra Vectores de Ojos Infinitos

Si cada vértice de cada objeto visual en una escena requiere un vector de luz, un vector del ojo, y el cálculo de la sombra, una porción significativa del cálculo de representación se utiliza en los vértices sombreados. La cantidad de cálculo puede reducirse si el vector de luz, o el vector del ojo, o ambos vectores son constantes. El vector de luz es constante cuando usa una luz direccional.. El vector del ojo es constante por defecto, aunque podemos especificar un vector variable del ojo usando un método del objeto **View**.

Efectos inter-objetos no Considerados

Mientras que el modelo de iluminación se basa en la física, los fenómenos físicos complejos no se modelan. Obviamente, la sombra echada por la esfera sobre el plano no está en la Figura 6-2. No tan obvio, también falta la luz reflejada de la esfera sobre el plano. También falta la luz reflejada desde el plano sobre la esfera que de nuevo se refleja en el plano... etcétera.

A menudo es difícil comprender la complejidad del cálculo de la acción de la luz. Consideremos la dificultad de calcular cómo cada gota del agua se comporta en una ducha. Las gotas vienen de la cabeza de la ducha en distintas direcciones. Cuando encuentran un objeto, la colisión que resulta produce muchas gotas más pequeñas que viajan en distintas direcciones. El proceso se repite muchas veces antes de que el agua se vaya por el desague. La complejidad de interacciones de la luz con los objetos visuales es muy semejante. Algunas de las diferencias entre los comportamientos del agua y la luz son que la luz no tiene ninguna adherencia (luz no se pega a los objetos visuales) y el efecto de la gravedad es insignificante para la luz.

Para reducir la complejidad del cálculo, el modelo de iluminación considera solamente un objeto visual al mismo tiempo. Consecuentemente, las sombras y las reflexiones inter-objetos no son renderizadas por el modelo de iluminación. Estos dos efectos requieren la consideración de todos los objetos junto con sus posiciones relativas en el momento de la representación. Se necesita

considerablemente más cálculo para renderizar una sola escena con efectos inter-objetos. Java 3D, y el resto de los sistemas gráficos en tiempo real, no hacen caso de efectos inter-objetos en la representación. Algunos de los efectos ignorados del mundo real se pueden agregar a las escenas cuando sea necesario.

Modelo de Color

El modelo del color no está basado en la física. Java 3D modela el color de las luces y los materiales como una combinación de rojo, verde, y azul. El color blanco, como el color de la luz o del material, es la combinación de los tres componentes con la intensidad máxima. Cada luz produce un solo color de luz especificado por un tuple RGB. El modelo de iluminación se aplica a cada uno de los componentes del color RGB. Por ejemplo, una bola roja en presencia de una luz azul no será visible puesto que la luz azul no se refleja desde un objeto rojo. En realidad, el color es una combinación de muchas longitudes de onda de la luz, no solo tres. El modelo de color RGB representa muchos colores, pero no todos.

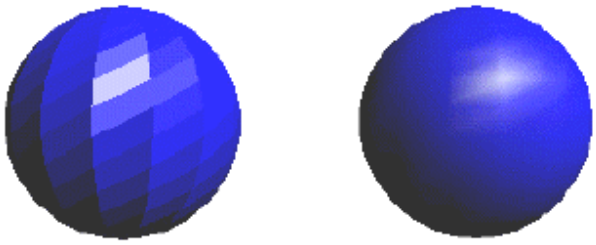
Influencia de las Luces

En Java 3D, la porción de una escena donde los objetos visuales son iluminados por una fuente de luz determinada se llama la región de influencia de ese objeto de luz. La región de influencia más simple para una fuente de luz usa un objeto **Bounds** y el método setInfluencingBounds() de **Light**. Cuando un objeto fuente de luz con límites de influencia intersecciona con los límites de un objeto visual, se utiliza la luz para sombrear todo el objeto. Los límites de influencian de una luz determinan qué objetos iluminar, no qué porciones de objetos se iluminan.

Modelo de Sombreado

El modelo de iluminación sombrea todos los vértice de un objeto visual por cada luz de influencia. La sombra de un vértice es la suma de las sombras proporcionadas por cada fuente de luz para la vértice. El resto de un objeto visual se sombrea basándose en la sombra de los vértices. El modelo de sombra de un objeto visual, especificado como atributo **Appearance NodeComponent**, determina cómo se hace el sombreado para el resto del objeto visual.

El **ColoringAttributes NodeComponent** especifica el modelo de sombra para los objetos visuales donde el modelo de sombra se especifica como uno de **SHADE_GOURAUD**, **SHADE_FLAT**, **FASTEST**, **NICEST**. Debemos tener cuidado ya que el color de un objeto **ColoringAttributes** nunca se utiliza en el sombreado. En el sombrado **Gouraud**, cada pixel se sombrea con un valor derivado de la interpolación trilinear del valor de la sombra de cada vértice del polígono que lo encierra. En el sombreado plano, todos los pixeles de un polígono se asignan el valor de la sombra a partir de un vértice del polígono. La Figura 6-3 muestra una esfera sombreada plana y una esfera sombrada **Gouraud**. La ventaja del sombreado plano es la velocidad en la representación del software. El sombreado de **Gouraud** tiene la ventaja de la apariencia visual.



Un último punto antes de ir a un ejemplo; los objetos fuentes de luz no son objetos visuales. Incluso si una fuente de luz se situa dentro de la vista, no será renderizada.

Receta para Iluminar Objetos Visuales

Se necesita una serie de pasos para permitir la iluminación de los objetos visuales en el universo virtual. Además de crear y de personalizar objetos para requisitos particulares de luz, cada objeto de luz debe ser agregado al escenario gráfico y haber especificado un objeto **Bound**. Cada objeto visual que se va a sombrear debe tener superficies normales y características de material. Abajo podemos ver la receta con los pasos necesarios:

1. Especificación de la fuente de Luz
    1. seleccionar los límites
    2. añadirla al escenario gráfico

2. Objeto Visual
    1. superficies
    2. propiedades de material

Si falta algunos de estos elementos no se podrá usar la iluminación. La presencia del objeto **Material** en un manojo **Appearance** de un objeto visual permite el modelo de iluminación para ese objeto. Sin el objeto **Material** el objeto visual será coloreado, pero no sombreado por el **ColoringAttribute** o los colores de vértice del objeto **Geometry**. Si ni uno ni otro están presentes, el objeto será blanco sólido. La Figura 6-5 muestra la excepción lanzada cuando un objeto visual tiene un objeto **Material** pero no tiene superficies normales.

javax.media.j3D.IllegalRenderingStateException: Cannot do lighting without
specifying normals in geometry object

Los errores con fuentes de luz pueden no ser fáciles de encontrar. No hay ningún aviso de que dejamos sin luz un escenario gráfico. Ni hay ninguna alerta por no fijar los límites de influencia de una fuente de luz. En cualquiera de estos casos, el objeto de luz no tendrá ninguna influencia sobre los objetos visuales en el escenario gráfico. Un objeto visual especificado correctamente para sombreado (es decir, uno con un objeto **Material**) en un escenario gráfico vivo pero fuera de los límites de influencia de todos los objetos fuente de luz se renderizará a negro. Es posible especificar correctamente una escena en la cual un objeto visual con las características materiales influenciadas por un objeto de luz y que se renderice a negro. La orientación relativa de la luz, el objeto visual, y la dirección de la visión entran en juego en el renderizado.

Ejemplos de Luces Sencillas

Según lo mencionado arriba, crear renderizados con sombras implica la especificación apropiada de la fuente de luz y de los objetos visuales. Hasta el momento, ni la clase **Light** ni los objetos **Material** se han discutido en detalle. Sin embargo, aprovechándo los valores por defecto del API y sus características,

podemos proceder a iluminar mundos virtuales. Los primitivos geométricos generan superficies normales cuando se solicitan. Los valores por defecto del objeto **Material** especifican un objeto visual razonable. Los valores por defecto de los constructores de la fuente de luz especifican fuentes de luz utilizables. Usando la clase **SimpleUniverse** con los dos métodos del Fragmento de código 6-1 se produce un universo virtual que incluye una sola esfera con las características materiales con sus valores por defecto iluminada por un solo objeto fuente de luz **AmbientLight**. El primer método del fragmento del código ensambla un objeto **Material** con un objeto **Apeareance** para la esfera. El segundo método crea un objeto **BranchGroup** para servir como la raíz de la rama de contenido gráfico, después agrega los objetos **Sphere** y **AmbientLight** al escenario gráfico. El objeto **Material** en el manojo del aspecto se agrega al objeto **Sphere** en la construcción de la esfera (líneas 12 y 13). Un valor por defecto **BoundingSphere** proporciona la región de influencia para el objeto **AmbientLight** (líneas 15 a 17). El diagrama del escenario gráfico de este mundo virtual aparece en La Figura 6-6.

Fragmento de Código 6-1, Crear un Escena con un Esfera Iluminada.

```
1.  Appearance createAppearance() {
2.  Appearance appear = new Appearance();
3.  Material material = new Material();
4.  appear.setMaterial(material);
5.
6.  return appear;
7.  }
8.
9.  BranchGroup createScene (){
10. BranchGroup scene = new BranchGroup();
11.
12. scene.addChild(new Sphere(0.5f, Sphere.GENERATE_NORMALS,
13. createAppearance()));
14.
15. AmbientLight lightA = new AmbientLight();
16. lightA.setInfluencingBounds(new BoundingSphere());
17. scene.addChild(lightA);
```

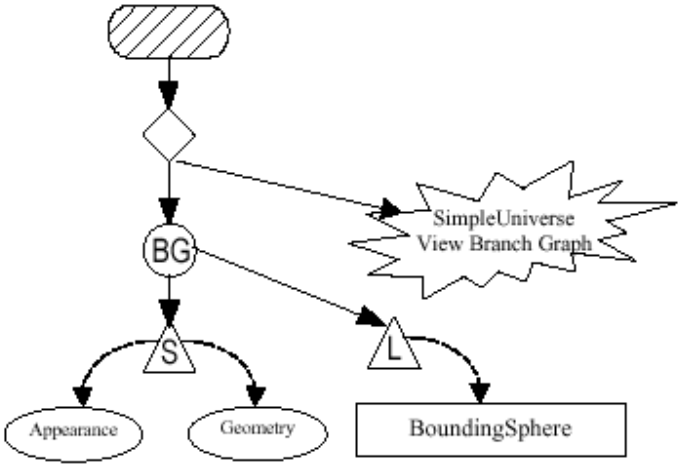
```
18.
19.   return scene;
20.   }
```

Las líneas 4 y 5 del Fragmento de código 6-1 podrían ser reemplazadas por la siguiente línea que crea y usa un objeto **Material** anónimo.

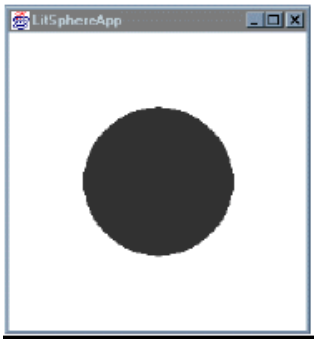
```
Appear.setMaterial(new Material());
```

Los objetos **Material** son completamente personalizables con los parámetros de su constructor, simplificando el uso de objetos **Material** anónimos. Por el contrario, crear un objeto **Light** anónimo hace mucho más díficil la adicción de límites de influencia. Debemos tener en cuenta que el nombramiento del objeto **Material** puede hacer el objeto sea más fácil de compartir entre varios manojos del aspecto, dando como resultado un funcionamiento mejor.

El **SimpleUniverse** proporciona los objetos **VirtualUniverse** y **Locale** junto con el la rama de vista gráfica para el diagrama de escenario gráfico mostrado en la Figura 6-6. Sin una transformación, el objeto **Sphere** y el objeto **BoundingSphere** estarán centrados en el origen, y se interseccionarán. El objeto **Sphere** se sombrea por la fuente **AmbientLight**. La Figura 6-7 muestra la imagen que resulta con un fondo blanco. La especificación del fondo no se muestra en el código.



La esfera de la Figura 6-7 es de color gris uniforme, que es el valor por defecto de la propiedad ambiente del material.

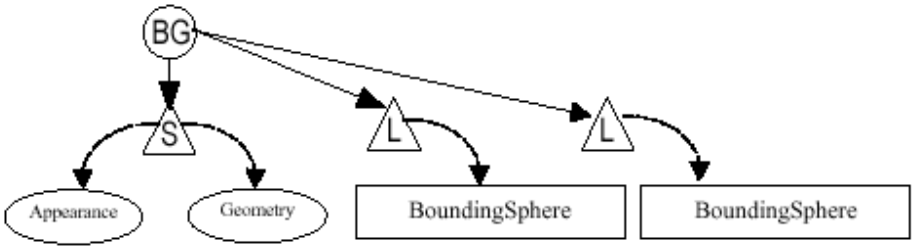


La Figura 6-7 muestra que las escenas iluminadas unicamente con luz ambiente son opacas. Como la iluminación ambiente es uniforme, produce la sombra uniforme. La luz ambiente está pensada para llenar de luz una escena donde otras fuentes no iluminan. La adición de una fuente **DirectionalLight** hará esta escena más interesante.

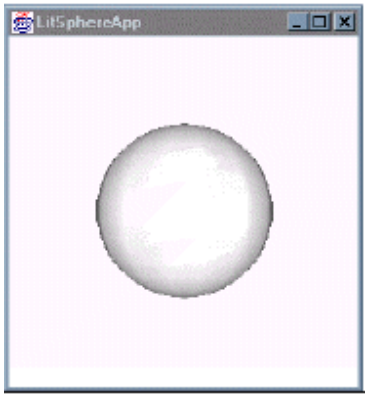
Insertando el Fragmento de código 6-2 en el Fragmento de código 6-1 se añade un **DirectionalLight** a la rama de contenido gráfico de la escena. Una vez más los valores por defecto se utilizan para la fuente de luz, y se utiliza un **BoundingSphere** por defecto para la región de influencia. La Figura 6-8 muestra el diagrama del escenario gráfico que resulta sin los objetos proporcionados por el **SimpleUniverse**.

Fragmento de Código 6-2, Añadir una Luz Direccional a la Escena.

```
1.  DirectionalLight lightD1 = new DirectionalLight();
2.  lightD1.setInfluencingBounds(new BoundingSphere());
3.  // customize DirectionalLight object
4.  scene.addChild(lightD1);
```

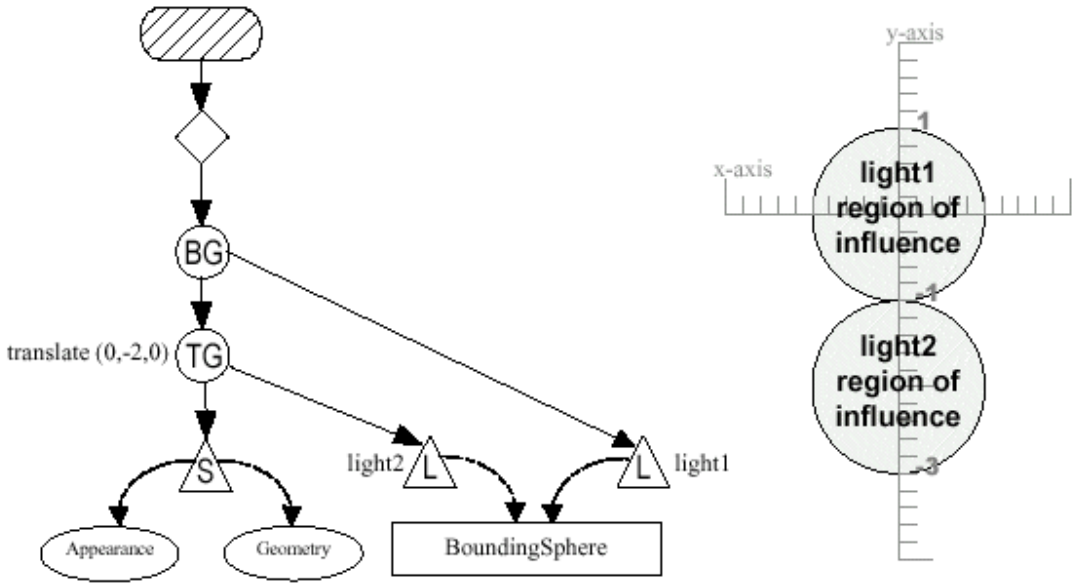


La Figura 6-9 muestra la imagen producida por la combinación de los dos fragmentos del código. La influencia del objeto **AmbientLight** apenas se puede ver con la fuente **DirectionalLight**. Obviamente, es necesaria la personalización de los requisitos particulares de los objetos de luz y/o las características materiales del objeto visual para crear escenas interesantes.



- **Dónde Añadir un Objeto Light en un Escenario Gráfico**

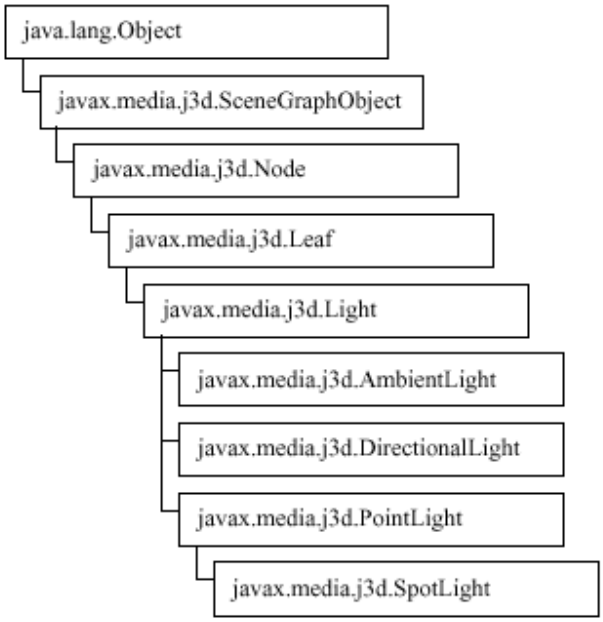
La influencia de un objeto de luz en el mundo no está afectada por la posición del objeto de luz en el escenario gráfico; sin embargo, el objeto **bounds** referenciado por la luz si lo está. El objeto **bounds** está sujeto a las coordenadas locales del escenario gráfico donde se inserta el objeto de luz. Consideremos la Figura 6-10 como ejemplo. El mismo objeto **BoundingSphere** referenciado por dos fuentes de luz proporciona dos regiones de influencia distintas debidas la traslación proporcionada por el objeto **TransformGroup**. El origen del sistema de coordenadas local del escenario gráfico debajo del **TransformGroup** está 2 metros por debajo del origen del mundo (Locale) y la otra región de la esfera de influencia.



Si dependen o no, los objetos de la fuente de luz del escenario gráfico en la Figura 6-10 influencian el sombreado (luz) del objeto visual iluminado si los límites del objeto visual interseccionan con la región de influencia de los objetos de luz. Especificar la región de influencia de una luz como un sólo **bounds** podría no funcionar para todas las aplicaciones.

Clase Light

El API Java 3D proporciona cuatro clases para luces. Todas se derivan de la clase **Light**. La Figura 6-11 muestra la jerarquía de clases de Java 3D relacionada con las luces. **Light**, una clase abstracta, proporciona los métodos y las constantes de capacidades asociadas para manipular el estado, color, y los límites de un objeto **Light**. El estado de la luz es un boleano que activa y desactiva la luz.



El siguiente bloque de referencia lista los métodos y las constantes de la clase **Light**. Debemos recordar que los límites seleccionados con setInfluencingBounds() activan una luz cuando el objeto **bounds** referenciado intersecciona con la vista.

Lista Parcial de Métodos de la Clase **Light**

Light es una clase abstracta que contiene variables de ejemplar comunes a todas las luces.

void setColor(Color3f color)

Selecciona el color actual de la luz.

void setEnable(boolean state)

Activa y desactiva la luz.

void setInfluencingBounds(Bounds bounds)

Selecciona los límites de influencia de la luz.

Sumario de Capacidades de la Clase **Light**

ALLOW_INFLUENCING_BOUNDS_READ | WRITE
ALLOW_STATE_READ | WRITE
ALLOW_COLOR_READ | WRITE.

- **Luz Ambiente**

Los objetos de luz ambiente proporcionan luz de la misma intensidad en todas las localizaciones y en todas las direcciones. Los objetos de luz ambiente modelan la luz reflejada desde otros objetos visuales. Si miramos la superficie inferior de nuestro escritorio, veremos la parte inferior del escritorio aunque ninguna fuente de luz esté dando directamente en esa superficie (a menos que tengamos una lámpara bajo el escritorio). La luz que brillaba hacia arriba en el fondo del escritorio se reflejó en el suelo y en otros objetos. En ambientes naturales con muchos objetos, la luz se refleja desde muchos objetos para proporcionar la luz ambiente. La clase **AmbientLight** de Java 3D simula este efecto.
El siguiente bloque de referencia enumera los constructores de la clase **AmbientLight**. La clase abstracta **Light** proporciona los métodos y las capacidades para esta clase (enumerada en el bloque de referencia anterior).

Sumario de Constructores de la clase **AmbientLight**

Un objeto fuente de luz ambiente proporciona la misma intensidad de luz en todas las localización y direcciones. Modela la compleja reflexión inter-objetos de la luz presente en escenas naturales.

AmbientLight()
Construye e inicializa un objeto fuente de luz ambiente usando los siguientes valores por defecto:

- lightOn true
- color (1, 1, 1)

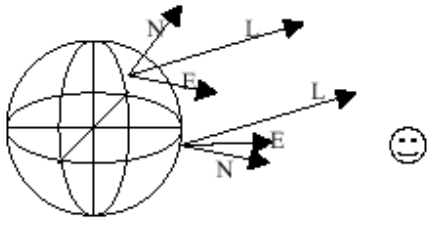
AmbientLight(Color3f color)
Construye e inicializa una luz ambiente usando los parámetros especificados.
AmbientLight(boolean lightOn, Color3f color)
Construye e inicializa una luz ambiente usando los parámetros especificados.

Mientras que podría ser natural pensar que una fuente de luz ambiente se puede aplicar globalmente, esto no es necesariamente cierto en un programa Java 3D. La influencia de la fuente **AmbientLight** está controlada por sus límites igual que otras fuentes de luz Java 3D. Se pueden utilizar varios objeto fuente **AmbientLight**

en un programa de Java 3D. No hay límite en el número de los objetos fuentes **AmbientLight** que se pueden utilizar.
Según lo mencionado en secciones anteriores, la sombra de un vértice es el resultado de las fuentes de luz, de las características materiales del objeto visual, y de su geometría relativa (distancia y orientación). Para las reflexiones ambiente, la geometría no es un factor. La propiedad ambiente del material sólo se utiliza para calcular la reflexión ambiente. El modelo de iluminación calcula la reflexión ambiente de la luz como el producto la intensidad del **AmbientLight** y la propiedad ambiente del material del objeto visual.

- **Luz Direccional**

Una fuente **DirectionalLight** aproxima fuentes de luz muy distantes tales como el sol. Al contratrio que las fuentes **AmbientLight**, las fuentes **DirectionalLight** proporcionan una sola dirección al brillo de luz. Para los objetos iluminados con una fuente **DirectionalLight**, el vector de luz es constante.
La Figura 6-12 muestra dos vértices de la misma esfera que están siendo iluminados por una fuente **DirectionalLight**. El vector de luz es igual para estos dos y para todos los vértices. Compara La Figura 6-12 con la Figura 6-1 para ver la diferencia. Puesto que todos los vectores de luz de una fuente **DirectionalLight** son paralelos, la luz no se atenúa. En otras palabras, la intensidad de una fuente **DirectionalLight** no varía con la distancia al objeto visual y la fuente **DirectionalLight**.



Los siguientes bloques de referencia listan los constructores y los métodos de

DirectionalLight, respectivamente.

Sumario de Constructores de la Clase **DirectionalLight**

Los objetos **DirectionalLight** modelan fuentes de luz muy distantes teniendo una dirección del vector de luz constante

DirectionalLight()

Construye e inicializa una fuente direccional usando los siguientes valores por defecto:

- lightOn true
- color (1, 1, 1)
- direction (0, 0, -1)

DirectionalLight(Color3f color, Vector3f direction)

Construye e inicializa una luz direccional con el color y la dirección especificados. Por defecto el estado es true (on).

DirectionalLight(boolean lightOn, Color3f color, Vector3f direction)

Construye e inicializa una luz direccional con el estado, el color y la dirección especificados.

Sumario de Métodos de la Clase **DirectionalLight**

void setDirection(Vector3f direction)

Selecciona la dirección de la luz.

void setDirection(float x, float y, float z)

Selecciona la dirección de la luz.

Sumario de Capacidades de la Clase **DirectionalLight**

Además de las Capacidades heredadas de la clase Light, los objetos DirectionalLight tienen la siguiente capacidad:

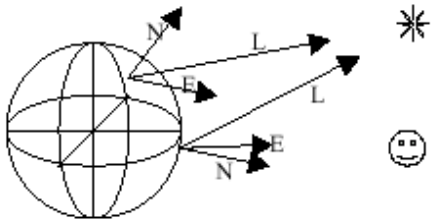
ALLOW_DIRECTION_READ | WRITE

Los **DirectionalLights** sólo participan en las porciones difusas y specular de la reflexión del modelo de la iluminación. Para las reflexiones difusas y specular, la geometría es un factor (al contrario que las reflexiones ambiente). Variar la

dirección de la fuente de luz cambiará el sombreado de los objetos visuales. Solo las características materiales difusas y specular se utilizan para calcular las reflexiones difusas y specular.

Punto de Luz

Un **PointLight** es el contrario de un **DirectionalLight**. Es una fuente de luz omnidireccional cuya intensidad se atenúa con la distancia y tiene una localización. (un **DirectionalLight** no tiene ninguna localización, solo una dirección). Los objetos **PointLight** se aproximan a bombillas, velas, u otras fuentes de luz sin reflectores o lentes.
Un modelo de ecuación cuadrática modela la atenuación de las fuentes **PointLight**. La ecuación se encuentra en la sección E.2 de la especificación del API Java 3D. La Figura 6-13 ilustra la relación de un objeto **PointLight** con una esfera. Observa que los vectores de luz no son paralelos.



Los siguientes bloques de referencia listan los constructores y los métosos de **PointLight**, respectivamente.

Sumario de Constructores de la Clase **PointLight**

El objeto **PointLight** especifica una fuente de luz atenuada en el espacio que irradia la luz igualmente en todas las direcciones desde la fuente de luz.

PointLight()
Construye e inicializa una fuente de punto de luz usando los siguientes valores por defecto:

- lightOn true
- color (1, 1, 1)

- position (0, 0, 0)
- attenuation (1, 0, 0)

PointLight(Color3f color, Point3f position, Point3f attenuation)

Construye e inicializa un punto de luz. Por defecto la luz está activa.

PointLight(boolean lightOn, Color3f color, Point3f position, Point3f attenuation)

Construye e inicializa un punto de luz.

Sumario de Métodos de la Clase **PointLight**

void setAttenuation(Point3f attenuation)

Selecciona los valores de atenuación actuales de la luz y los sitúa en el parámetro especificado. Los tres valores especificados en el objeto **Point3f** especifican los coeficientes constante, linear, y cuadrático, respectivamente.

$$\text{atenuación} = \frac{1}{\text{constate}+\text{linear}+\text{cuadrático}*\text{distancia}^2}$$

donde distancia es la medida desde la fuente de luz al vértice que está siendo sombreado.

void setAttenuation(float constant, float linear, float quadratic)

Selecciona los valores de atenuación actuales de la luz y los sitúa en el parámetro especificado. Ver la ecuación anterior.

void setPosition(Point3f position)

Selecciona la posición de la Luz.

void setPosition(float x, float y, float z)

Selecciona la posición de la Luz.

Sumario de Capacidades de la Clase **PointLight**

Además de las capacidades heredadas de la clase **Light**, los objetos **PointLight** tienen las siguientes capacidades.

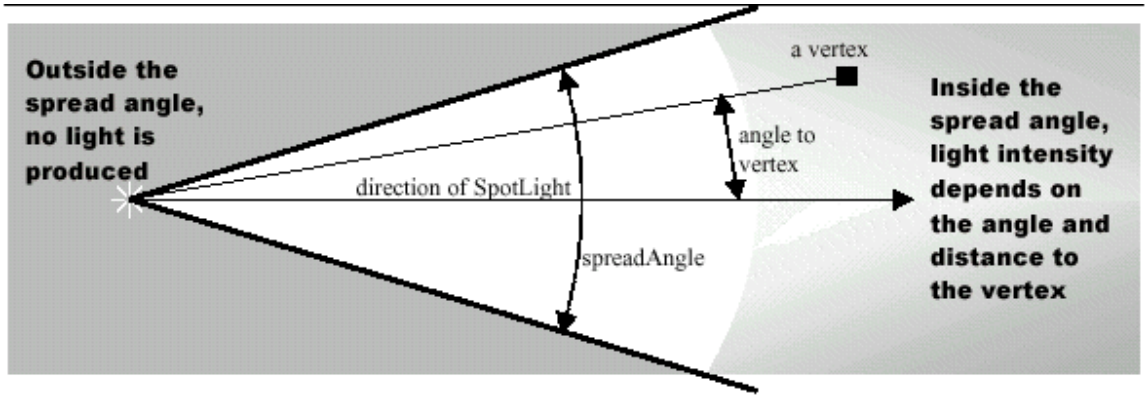
ALLOW_POSITION_READ | WRITE

ALLOW_ATTENUATION_READ | WRITE

Como **DirectionalLight**, **PointLight** participa solamente en las porciones difusa y de reflexión especular del modelo de iluminación. Para las reflexiones difusas y especular, la geometría es un factor. Variando la localización de un objeto **PointLight** se cambiará el sombreado de los objetos visuales en una escena.

- **SpotLight**

SpotLight es una subclase de **PointLight**. La clase **SpotLight** agrega dirección y concentración a los parámetros de posición y de atenuación de **PointLight**. Los objetos **SpotLight** crean manualmente modelos de fuentes de luz artificiales como flashes , lámparas, y otras fuentes con reflectores y/o lentes.
La intensidad de la luz producida por una fuente **SpotLight** produce la luz dentro de un ángulo especificado desde la dirección de la luz. Si el vértice exterior se sale del ángulo de la extensión de la luz, entonces no se produce ninguna luz. Por dentro del ángulo de extensión, la intensidad varía mediante el ángulo y la distancia al vértice. Una vez más una ecuación cuadrática modela la atenuación debido a la distancia. El parámetro concentración y una ecuación diferente gobiernan la variación de la intensidad debido al ángulo. Las ecuaciones que gobiernan estos lazos se encuentran en la sección E.2 de la especificación del API Java 3D. La Figura 6-14 ilustra en 2D cómo la intensidad de luz varía desde una fuente **PointLight** en 3D.



El ángulo de extensión de un objeto **SpotLight** podría hacer que la luz iluminara parte de un objeto visual. Esta es la única luz capaz de iluminar sólo una parte de un objeto visual.

Los siguientes bloques de referencia listan los constructores y métodos de **PointLight**, respectivamente.

Sumario de Constructores de la Clase **SpotLight**

SpotLight es una suclase de **PointLight** con los atributos de dirección, ángulo de extensión y concentración.

SpotLight()

Construye e inicializa una fuente de luz usando los siguientes valores por defecto:

- lightOn true
- color (1, 1, 1)
- position (0, 0, 0)
- attenuation (1, 0, 0)
- direction (0, 0, -1)
- spreadAngle PI (180 degrees)
- concentration 0.0

SpotLight(Color3f color, Point3f position, Point3f attenuation, Vector3f direction, float spreadAngle, float concentration)

Construye e inicializa un punto de luz. Puedes ver el sumario de métodos de **PointLight** para más infromación sobre la atenuación. Por defecto la luz está activa.

SpotLight(boolean lightOn, Color3f color, Point3f position, Point3f attenuation, Vector3f direction, float spreadAngle, float concentration)

Construye e inicializa un punto de luz. Puedes ver el sumario de métodos de **PointLight** para más infromación sobre la atenuación.

Sumario de Métodos de la Clase **SpotLight**

Además de los métodos listados anteriormente para **PointLight**, la clase **SpotLight** tiene los siguientes métodos:

void setConcentration(float concentration)

Selecciona la concentración del punto de luz.

void setDirection(float x, float y, float z)

Selecciona la dirección de la luz.

void setDirection(Vector3f direction)

Selecciona la dirección de la luz.

void setSpreadAngle(float spreadAngle)

Selecciona el ángulo de exposición de la luz.

Sumario de Capacidades de la Clase **SpotLight**

Además de las capacidades heredadas de la clase **Light**, los objetos **SpotLight** tienen las siguientes capacidades:

ALLOW_SPREAD_ANGLE_READ | WRITE
ALLOW_CONCENTRATION_READ | WRITE
ALLOW_DIRECTION_READ | WRITE

Como los objetos **DirectionalLight** y **PointLight**, los **SpotLights** participan solamente en las porciones difusas y reflexión especular del modelo de iluminación. Para las reflexiones difusas y especular, la geometría es un factor. Cambiar la localización o la orientación de una fuente de **SpotLight** cambiará el sombreado de los vértices dentro de la región de influencia de la luz.

Aplicaciones de Fuentes de Luz

Con todos los tipos de fuentes de luz, y la variedad de maneras de utilizarlas, veremos una pequeña guía de su uso típico en esta sección. En general, desearemos utilizar tan pocas fuentes de luz como se pueda para una aplicación dada. Cuántas son suficientes dependerá del efecto de iluminación deseado para la aplicación. El número de luces y la configuración de atributos es más una consideración artística que científica.

Desde un punto de vista artístico, a menudo es suficiente tener solo dos luces para una escena dada. Una luz proporciona la iluminación principal, la otra se

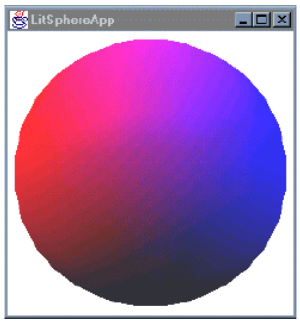
utiliza para completar la cara más oscura de los objetos. La luz principal normalmente se coloca a la derecha del espectador, el relleno a la izquierda del espectador. Una vez más éstas son pautas generales para lo que pueda ser un diseño artístico complejo.
Normalmente se prefiere incluir fuentes de luz direccionales para la mayoría de las aplicaciones puesto que el cálculo requerido en la representación es perceptiblemente menor que para los puntos de luz. Las fuentes de puntos de luz se usan muy raramente debido a la alta complejidad de cálculo.
Es normal incluir una sola fuente de luz ambiente con una gran región de influencia. Esto iluminará las partes posteriores de los objetos (como la "cara oscura de la luna"). El valor por defecto del color funcionará razonablemente bien. El tiempo requerido para incluir la luz ambiente es pequeño comparado con otras fuentes de luz. Dejar fuera una luz ambiente puede ser muy sensible en algunas escenas, y no ser notado en absoluto en otras.

Ejemplos de Iluminación

La interacción de la luz con los objetos es muy compleja en la naturaleza. Incluso en el mundo virtual donde es menos complejo el modelo de la iluminación, las fuentes de luz son simplistas, y las superficies son menos detalladas, el efecto de una fuente de luz en un objeto visual es algo complejo. Esta sección presenta algunos ejemplos de la iluminación para ayudar a clarificar las características, capacidades, y las limitaciones del modelo de iluminación en Java 3D.

Dos Luces Coloreadas

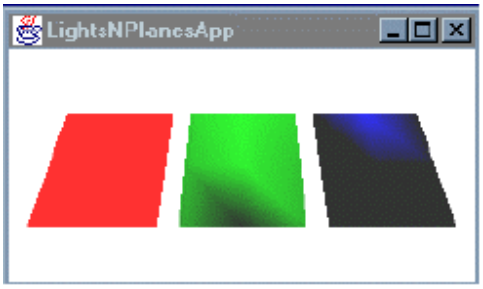
La Figura 6-15 muestra una sola esfera blanca iluminada por dos fuentes de luz direccionales, una roja y una azul. Aunque puede sorprendernos, la sombra que resulta es magenta. Mezclar rojo y azul da lugar a la púrpura, que es el resultado en el sistema de color sustractivo. Mezclar luces rojas y azules resulta en magenta, los resultados de un sistema de color aditivo.



En ausencia de luz, la esfera es negra. Si la única fuente de luz es roja, entonces la esfera aparecerá roja, o algo sombreada en rojo. Con la adicción de una fuente de luz azul, sólo son posibles el rojo, el azul y las mezclas de estos dos.

Diferentes Patrones de Iluminación

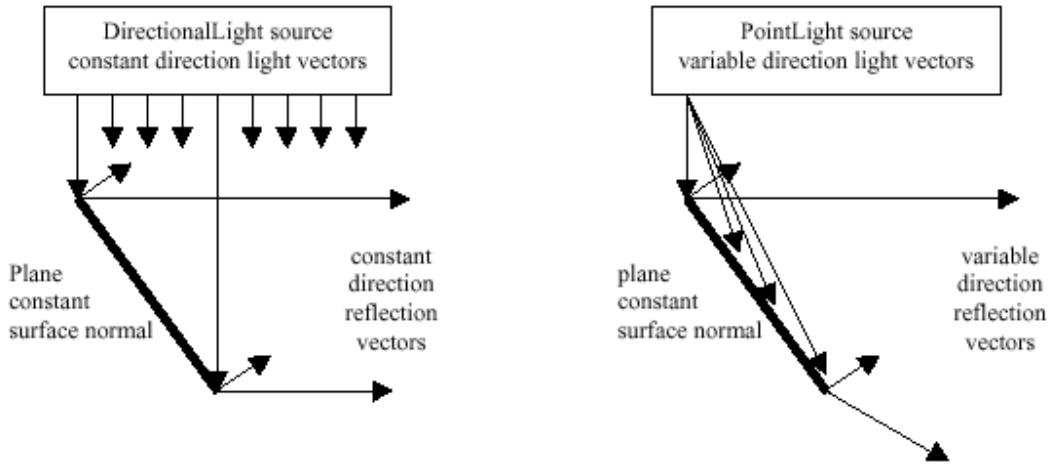
La siguiente aplicación ilustra las diferencia entre las fuentes de luz. En LightsNPlanesApp.java se iluminan tres planos con una fuente de luz distinta. De izquierda a derecha, los objetos **DirectionalLight**, **PointLight**, y **SpotLight** iluminan los planos. La Figura 6-16 muestra la imagen renderizada por la aplicación.



El **DirectionalLight** ilumina el plano uniformemente. El **PointLight**, situado directamente sobre el borde superior del plano del centro, ilumina el plano de forma irregular debido a la dirección variable de la luz con respecto a las superficies, y, en un grado inferior, a la atenuación de la luz. El **SpotLight**, también situado directamente sobre el centro de su plano, ilumina solamente una parte pequeña del tercer plano.

La Figura 6-17 ilustra la geometría implicada en la iluminación de los primeros dos planos. En la ilustración izquierda, los vectores de luz constantes de la fuente **DirectionalLight** en conjunción con los vectores normales constantes de un plano dan lugar a vectores constantes de la reflexión, e incluso de la iluminación del plano. En la ilustración derecha los vectores de luz variables de la fuente

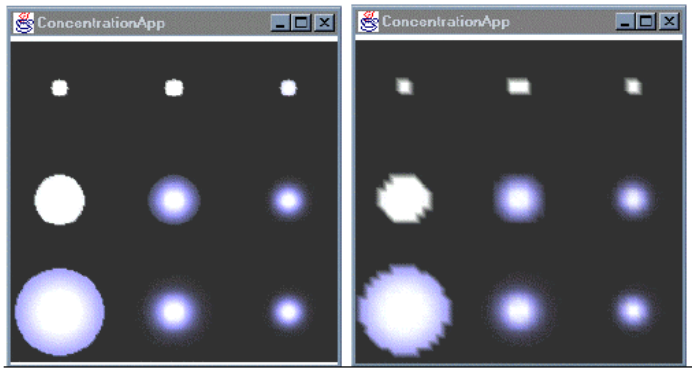
PointLight se combinan con los vectores normales constantes del plano dando por resultado las distintas direcciones para los vectores de reflexión, y una iluminación desigual del plano. El **SpotLight** es un caso especial de la fuente **PointLight** donde la influencia de la fuente de luz está limitada por el ángulo de la extensión.



Concentración y Ángulo de Extensión de SpotLights

La Figura 6-18 muestra las imágenes renderizadas a partir de versiones distintas del programa ConcentrationApp.java. Un plano es iluminado por nueve puntos de luz. Los valores del ángulo y de la concentración de la extensión para las luces de los puntos varían con la posición. El ángulo de la extensión varía por cada fila con valores de .1, .3 y .5 (radianes) desde la fila superior a la inferior, respectivamente. La concentración varía por cada columna con valores de 1,0, 50,0, y 100,0 desde la columna de la izquierda a la de la derecha, respectivamente.

Los valores de concentración no tienen ningún efecto para la fila superior, el ángulo de la extensión es el único factor. En la fila inferior, la concentración tiene un efecto para cada uno de los ángulos de la extensión. El azul en las imágenes es el color difuso del material.



ConcentrationApp.java demuestra dos limitaciones del modelo de iluminación. La primera es el renderizado de los artefactos representados en la Figura 6-18. Artefactos similares son visibles en la Figura 6-16. Los modelos desiguales de iluminación para los planos verdes y rojos son debidos al pequeño número de vértices usados para representar los planos. Recordamos que el modelo de iluminación se aplica solamente en los vértices. Cuantos más vértices, mayor es el efecto de sombreado y más tardará en renderizarse.

La diferencia entre las imágenes izquierda y derecha de la Figura 6-18 es debido a la diferencia en el número de las vértices usados para representar el plano. La versión del programa que generó la imagen izquierda utilizó 16 veces más vértices que la que está a la derecha (2.500 vértices contra 40.000). Los artefactos de la imagen derecha son un resultado de la reducción de la densidad de vértices en la superficie y la triangulación impuesta por el sistema de renderizado de Java 3D.

Limitar el Número de Luces

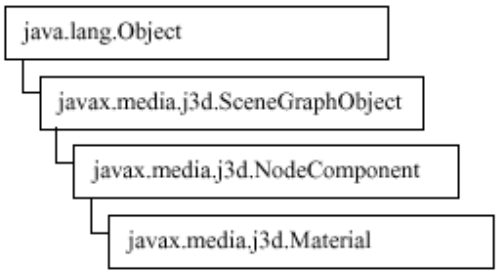
La segunda limitación demostrada en ConcentrationApp no se ve en la representación. El plano de **ConcentrationApp** son realmente cuatro objetos planos uno al lado de otro. Esto se hizo para superar una potencial limitación del sistema de representación subyacente. La especificación de **OpenGL** requiere soporte para ocho fuentes de luz simultáneas. Si el plano de **ConcentrationApp** fuera un objeto visual, entonces **OpenGL** limitaría el número de luces a ocho en algunas máquinas.

Usando los límites de influencia para seleccionar solamente las fuentes de luz relevantes para un objeto visual, Java 3D crea dinámicamente las especificaciones

de iluminación para las luces mientras que se renderizan los objetos visuales. Mientras que ningún objeto sea iluminado por más de ocho luces, los programas de Java 3D no están limitados en el número de luces en un mundo virtual.
Por eso proporcionar al cuadro planos más pequeños y los límites apropiados para asegurarse de que ningún plano se ve influenciado por más de ocho luces, en el ejemplo parece que hay nueve luces (realmente diez, con la luz ambiente) iluminando un plano. Necesita un poco más programación, pero el programa que resulta es más portable. Mientras que muchas implementaciones de **OpenGL** utilizan más de ocho luces simultáneas, si estamos planeando distribuir nuestros programas, debemos tener en cuante esta limitación potencial.
En esta sección, algunos ejemplos muestran alguna de las características y las limitaciones de la iluminación Java 3D. La intención de esta sección es dar a los lectores algunos ejemplos de programas básicos y algunas figuras de ejemplo para comparar con sus propios programas. No es posible proporcionar ejemplos de cada posible situación de iluminación, pues los factores en la representación son demasiado diferentes.
Una última cosa, **PointLight** y **SpotLight** utilizan la especificación de atenuación. La atenuación se especifica por los términos constantes en la ecuación cuadrática inversa basada en la distancia entre la luz y el vértice (véase el bloque de la referencia anterior). Encontrar la atenuación apropiada para una aplicación específica es un problema artístico. No se incluye ningun programa de ejemplo de atenuación en este tutorial.

Objetos Material

Las características materiales de un objeto visual se especifican en el objeto **Material** de un manojo de aspecto. **Material** es una subclase de **NodeComponent**. La Figura 6-19 muestra la jerarquía de clases del API Java 3D para **Material**.



El objeto **Material** especifica colores ambiente, difusos, especular, y emisivo y un valor de brillantez. Cada uno de los tres primeros colores se utiliza en el modelo de iluminación para calcular la reflexión correspondiente. El color emisivo permite que los objetos visuales "brillen intensamente en la oscuridad". El valor de brillantez se utiliza solamente para calcular reflexiones especulares.
Los siguientes bloques de referencia enumeran los constructores y los métodos de la clase **Material**.

Sumario de Constructores de la Clase **Material**

El objeto **Material** define la aparienia de un objeto bajo la iluminación.

Material()
Construye e inicializa un objeto **Material** usando los siguientes valores por defecto:

- ambientColor (0.2, 0.2, 0.2)
- emissiveColor (0, 0, 0)
- diffuseColor (1, 1, 1)
- specularColor (1, 1, 1)
- shininess 0.0

Material(Color3f ambientColor, Color3f emissiveColor, Color3f diffuseColor,
Color3f specularColor, float shininess)
Construye e inicializa un nuevo objeto **Material** usando los parámetros especificados.

Lista Parcial de Métodos de la Clase **Material**

void setAmbientColor(Color3f color)
Selecciona el color ambiente de este **Material**.
void setAmbientColor(float r, float g, float b)

Selecciona el color ambiente de este **Material**.

void setDiffuseColor(Color3f color)

Selecciona el color difuso de este **Material**.

void setDiffuseColor(float r, float g, float b)

Selecciona el color difuso de este **Material**.

void setDiffuseColor(float r, float g, float b, float a)

Selecciona el color difuso más alpha de este **Material**.

void setEmissiveColor(Color3f color)

Selecciona el color emisivo de este **Material**.

void setEmissiveColor(float r, float g, float b)

Selecciona el color emisivo de este **Material**.

void setLightingEnable(boolean state)

Activa o desactiva la iluminación de objetos visuales que referencian este objeto.

void setShininess(float shininess)

Selecciona la brillantez de este **Material**.

void setSpecularColor(Color3f color)

Selecciona el color especular de este **Material**.

void setSpecularColor(float r, float g, float b)

Selecciona el color especular de este **Material**.

java.lang.String toString()

Devuelve una representación String de los valores de este **Material**.

Sumario de Capacidades de la Clase **Material**

Además de las Capacidades heredadas de **NodeComponent**, los objetos **Material** tienen la siguiente capacidad:

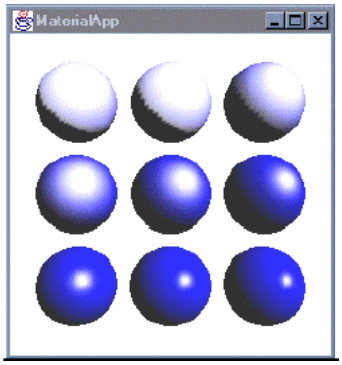
ALLOW_COMPONENT_READ | WRITE

Permite ller/escribir información de los campos individuales del componente.

Ejemplos sencillos de Material

Las reflexiones especulares ocurren naturalmente en los objetos lisos. En general, cuanto más lisa sea una superficie, más definida e intensa es la reflexión especular. Cuando una superficie es suficientemente lisa, actúa como un espejo que refleja la luz sin cambiar el color de la luz. Por lo tanto, el color especular de un objeto normalmente es blanco. Cambiamos el color specular de un **Material** para alterar la intensidad de una reflexión specular (por ejemplo, Color3f(0.8f, 0.8f, 0.8f)).

El valor de brillantez controla el rango de la extensión del ángulo de la visión para el cual se puede ver una reflexión especular. Una brillantez más alta resulta en reflexiones especulares más pequeñas. La Figura 6-20 muestra nueve esferas distintas iluminadas por una fuente de luz. Cada esfera tiene un valor de brillantez distinto.



Un objeto **Material** se asocia a un objeto visual a través de un objeto **Apearance** de la misma manera que lo hacen los atributos del aspecto. El método SetMaterial() de la clase **Appearance** referencia un objeto **Material** para ese objeto **Appearance**.

Propiedades Geometry color, ColoringAttributes, y Material

Hay tres maneras de especificar el color para un objeto visual: color por-vértice especificado en la geometría con los métodos getColor(), **ColoringAttributes** de un nodo **Appearance**, y el objeto **Material**. Java 3D permite que creemos objetos visuales sin usar ninguna, alguna, o las tres formas de especificar color.

Cuando se ha hecho más de una especificación del color, dos sencillas reglas determinan qué especificación del color toma la precedencia.

- Color **Material** se utiliza solamente cuando la representación ilumina objetos y color de **ColoringAttributes** sólo se utiliza cuando se renderizan objetos no iluminados.
- Geometría por-vértice siempre tiene precedencia sobre **ColoringAttributes** o **Material**.

Las reglas pueden ser más claras cuando el problema se divide en objetos iluminados o apagados. La iluminación está activa para un objeto cuando se referencia un objeto **Material**. Inversamente, cuando no se asocia ningún objeto **Material** al objeto visual, la iluminación está desactivada para ese objeto. Observa que una escena puede tener tanto objetos iluminados como apagados.
Cuando la iluminación está activa para un objeto (es decir, se referencia un objeto **Material**), se utilizan el color material o el color de la geometría por-vértice para sombrear. Si esta presente, el color por-vértice reemplaza los colores de **Material** difusos y ambiente. Observa que el color de **ColoringAttributes** nunca se utiliza para la iluminación de objetos. La siguiente Tabla resume las relaciones:

Color de Geometry por Vértice	Color ColoringAttributes	Resultado
NO	NO	Color Material
SI	NO	Color Geometry
NO	SI	Color Material
SI	SI	Color Geometry

Cuando la iluminación está desactivada para un objeto (es decir, no se referencia un objeto **Material**), se usan el color de **ColoringAttributes** o el color de por-vértice para colorear. Si está presente, el color de la geometría por-vértice reemplaza el color de **ColoringAttributes**. La siguiente Tabla resume las relaciones.

Color Geometry por Vértice	Color ColoringAttributes	Resultado
NO	NO	blanco plano
SI	NO	Color Geometry
NO	SI	ColoringAttributes
SI	SI	Color Geometry

Superficies Normales

Según lo mencionado en secciones anteriores, las superficies normales son necesarias para sombrear los objetos visuales. Al crear objetos visuales usando clases **Geometry**, utilizamos uno de los métodos setNormal() para especificar los vectores de los vértices.

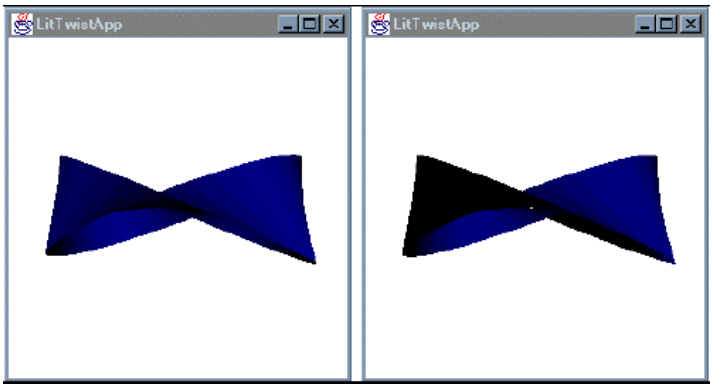
El **NormalGenerator** incluido con los utilidades de Java 3D genera superficies normales al especificar los objetos visuales que usan objetos **GeometryInfo**. Para generar superficies normales, ponemos nuestro objeto visual **Geometry** y llamamos a NormalGenerator.generateNormals().

Los primitivos geométricos generan sus propias superficies normalea cuando son especificados.

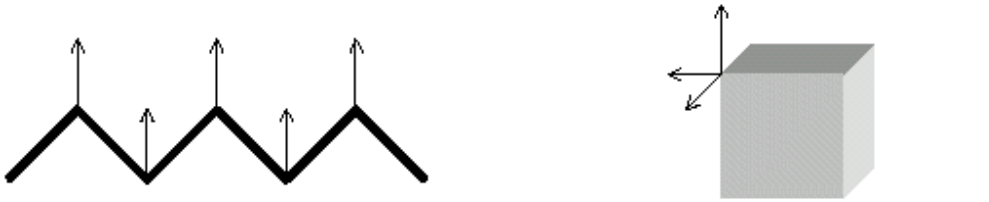
No importa cómo se especifican (o se generan) las superficies normales, sólo se especifica una superficie normal por vértice. Esto conduce a algunos problemas interesantes. Por ejemplo, cuando las dos superficies normales de polígonos son visibles, la normal es solamente correcta para una de las superficies normales. El resultado es que las caras posteriores sean renderizadas (si se renderizan) solamente con las características materiales de ambiente. Las reflexiones difusa y especular requieren la especificación normal apropiada.

Este problema común se soluciona especificando caras normales detrás al contrario que las superficies normales delanteras. Utilizamos el método setBackFaceNormalFlip() de un objeto **PolygonAttributes** para este propósito.

La Figura 6-21 muestra dos imágenes sombreadas de una tira doblada. La imagen de la izquierda fue renderizada desde la superficie frontal, y la derecha muestra las superficies normales traseras.



Cuando un vértice es compartido por las superficies normales o varian las orientaciones, tener solamente una superficie normal por vértice puede dar lugar a problemas. Consideremos los ejemplos ilustrados en la Figura 6-22. La geometría ilustrada en la cara del lado izquierdo de la Figura 6-22 muestra la sección transversal de una superficie donde cada polígono se orienta a un ángulo de 90° de sus vecinos. Si se selecciona la superficie normal como el normal actual para una superficie, es muy incorrecto para su vecino. Si las superficies normales se especifican según lo mostrado, entonces la superficie estará sombreada constantemente entre los vértices con superficies paralelas. Un problema similar ocurre con la geometría del cubo mostrada en la cara derecha en la Figura 6-22. La solución a ambos problemas es aumentar el número de vértices para aumentar el número de superficies normales. Esto, por supuesto, aumenta el uso de la memoria y el tiempo de la renderización.



normals along a polygonal surface (cross-section)

three possible normals for a vertex

Especificar la Influencia de las Luces

En ejemplos anteriores, la especificación de los límites que influencian un objeto de luz se consigue al referirse a un objeto **Bounds**. Esto conecta la localización de los límites que influencian a la localización de la luz. (En secciones anteriores se

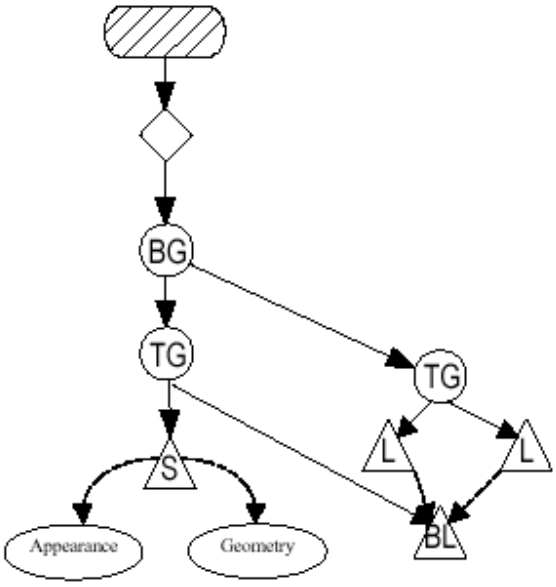
explicó cómo las transformaciones en el escenario gráfico afectan a los volúmenes de limites usados para especificar los límites de influencia de las luces.) Mientras que esto se hace trivial para mover luces junto con los objetos visuales que se iluminan, otras aplicaciones necesitan una especificación más flexible de la influencia de luces. Afortunadamente, el API Java 3D proporciona un método alternativo para especificar los límites de influencia y una manera de limitar el ámbito en adición de los límites.

Alternativa a los Límites de Influencia: BoundingLeaf

Un objeto **BoundingLeaf** es un alternativa a un objeto **Bounds** de influencia. Un objeto **BoundingLeaf** es referido por otros nodos de la hoja para definir una región de influencia. Como descendiente de **SceneGraphObject**, los ejemplares de **BoundingLeaf** se agregan al escenario gráfico. El objeto **BoundingLeaf** está sujeto al sistema de coordenadas local de su posición en el escenario gráfico, que podría ser independiente del sistema de coordenadas del objeto de luz. Es decir, usar un **BoundingLeaf** permite a una luz y a sus límites de influencia moverse independientemente.

Una llamada a setInfluencingBoundingLeaf() para un objeto de luz especifica el argumento **BoundingLeaf** como los límites de influencia de la luz. Esta especificación reemplaza cualquier especificación regional de los límites de influencia. Un objeto **BoundingLeaf** puede ser compartido por varios objetos de luz.

La Figura 6-23 muestra el diagrama del escenario gráfico para una aplicación de ejemplo de un objeto **BoundingLeaf** con objetos de luz. En esta escena, se mueven dos luces junto con un objeto **TransformGroup** (a la derecha). Estas luces podían ser ejemplares de **PointLight** o de **SpotLight**. Sin embargo, la influencia de estas luces no cambia cuando las luces se mueven. La influencia de las luces se mueve cuando el **TransformGroup** izquierdo cambia la localización del objeto **BoundingLeaf**. Podemos comparar este diagrama del escenario gráfico con el que está en la Figura 6-10.



En la Figura 6-10, si se mueve la luz, su región de influencia también se mueve. También, según lo demostrado en la Figura 6-10, la región de influencia de dos luces que comparten el mismo objeto **Bounds** pueden o no pueden tener la misma región de influencia. Cuando dos o más luces comparten el mismo objeto **BoundingLeaf**, tienen siempre la misma región de la influencia.

Ámbito de Límites de Influencia de las Luces

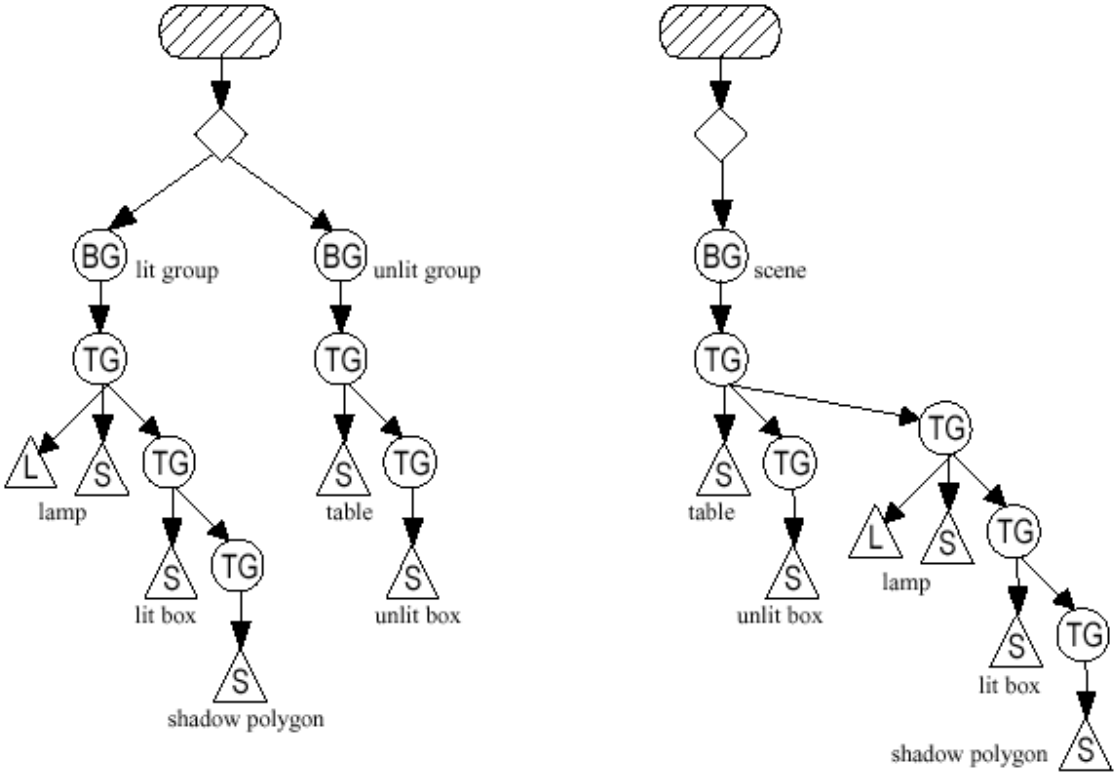
Una región de limites, con un objeto **Bounds** o un objeto **BoundingLeaf**, especifica la región de influencia de un objeto de luz. Un ámbito especificado puede además limitar la influencia de una luz a una porción del escenario gráfico. Como valor por defecto, todas las luces tienen el alcance del mundo virtual en el cual reside. La adición de una especificación del alcance reduce además la influencia de una luz a los objetos visuales en el escenario gráfico debajo del **group(s)** especificado. Por ejemplo, consideremos la aplicación siguiente.

Ejemplo de Ámbito de Iluminación

La escena consiste en una lámpara y algunos objetos visuales en una mesa. La lámpara tiene una sombra, por eso no todos los objetos, ni toda la mesa, debe ser iluminada por la lámpara. El interior (pero no el exterior) de la lámpara también se

debe iluminar (en este ejemplo, la sombra de la lámpara es completamente opaca). Sabemos que Java 3D no proporcionará la obstrucción por nosotros. Usando sólo un volumen de limitación, la influencia de la luz puede ser controlada, pero podría ser muy difícil, especialmente si se iluminan y apagan objetos que están uno cerca del otro, o se mueven.

Especificar un ámbito de limitaciones para la luz nos permite controlar limitaciones complejas de la influencia más fácilmente. La única consideración es mantener los objetos iluminados y apagados en partes separadas del escenario gráfico. Nuestro pensamiento inicial pudo ser comenzar a construir el escenario gráfico **BranchGroups** para los objetos iluminados y apagados, pero éso no es a menudo necesario ni recomendado para la mayoría de las aplicaciones.



El diagrama del escenario gráfico de la izquierda de la Figura 6-24 muestra un acercamiento nativo a la construcción del escenario gráfico. La organización no es natural y será difícil de manipular en una aplicación animada. Por ejemplo, si la mesa se mueve, la lámpara y otros objetos deben moverse con ella. En el escenario gráfico de la izquierda, mover la mesa (mediante la manipulación de

TransformGroup) no moverá la lámpara o el rectángulo ilumiando; solamente el rectángulo apagado se moverá con la mesa.

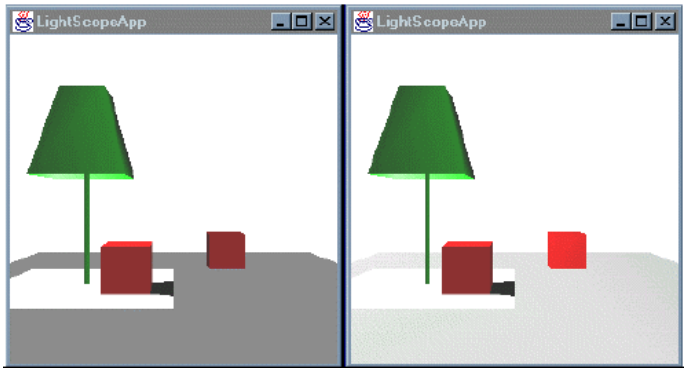
El diagrama del escenario gráfico de la derecha representa una organización más natural para la escena. Los objetos en la mesa son hijos del **TransformGroup** que coloca la mesa. Si la mesa se mueve (mediante la manipulación de **TransformGroup**) los objetos de la mesa se moveran con ella.

La escena de ejemplo se crea en LightScopeApp.java. La Figura 6-25 muestra dos imágenes renderizadas del programa de ejemplo. La imagen izquierda utiliza ámbito de luz para limitar la influencia de la luz de la lámpara a la lámpara y el rectángulo iluminado. La imagen derecha no utiliza scoping; por lo tanto, la luz de la lámpara ilumina el ' rectángulo no iluminado'.

El área brillante debajo de la lámpara (no representada en ningún diagrama del escenario gráfico) es un polígono situado justo sobre la tapa de la mesa. Este polígono brillante representa la parte de la mesa que es iluminada por la lámpara. El área brillante aparece más ligera que el resto de la mesa (incluso en la imagen derecha de la Figura 6-9) porque sus superficies se alinean más cercanas al punto de luz de la lámpara.

La sombra no aparece iluminada en ninguna imagen de la Figura 6-25 porque su característica **Material** difusa es negra. La sombra puede crearser con el uso del scoping solamente si un nodo adicional del grupo que se utiliza en el escenario gráfico.

La sombra en esta escena fue creada a mano. Las técnicas para crear sombras automáticamente (incluso dinámicamente) se discuten en la siguiente sección.



Tampoco se representan en ningún diagrama del escenario gráfico las tres fuentes de luz adicionales: dos fuentes de luz direccionales y una fuente de luz ambiente. Éstas son necesarias para simular la luz de una escena natural.
El siguiente bloque de referencia muestra los métodos de la clase **Light** usados para especificar limitaciones del scoping y el uso de los objetos **BoundingLeaf** para especificar los límites de influencia.

Lista Parcial de Métodos de la Clases **Light**

Otros métodos de la clase **Light** aparecieron en secciones anteriores.

void addScope(Group scope)
Añade el ámbito especificado a la lista de ámbitos de este nodo.
java.util.Enumeration getAllScopes()
Devuelve un objeto Enumeration con todos los ámbitos.
void insertScope(Group scope, int index)
Inserta el ámbito especificado por el nodo grupo en el índice especificado.
int numScopes()
Devuelve un contador con los ámbitos de luces.
void removeScope(int index)
Elimina el ámbito del nodo en la posición de índice especificada.
void setInfluencingBoundingLeaf(BoundingLeaf region)
Selecciona la región de influencia de la luz al **BoundingLeaf** especificado.
Seleccionar un **BoundingLeaf** sobreescribe un objeto **Bounds**.
void setScope(Group scope, int index)
Selecciona el ámbito de herencias en el índice especifiado. Por defecto las luces tienen ámbitos sólo para los límites de su región de influencia.

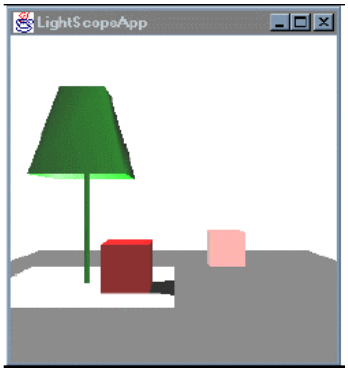
Otra ventaja de usar alcances para limitar la influencia de una luz: puede reducir el tiempo de renderizado. Calcular la intersección de los límites para un objeto visual con los límites que influencian de una luz es más complejo que determinar el alcance de una luz. Debemos tener cuidado con que ni el uso de los límites de influencia ni los alcances limitará la influencia de una luz a una parte de un objeto visual.

Crear Objetos Brillantes-en-la-Oscuridad, Sombras y Otros Problemas de Iluminación

Las secciones anteriores cubren las aplicaciones típicas de iluminación en Java 3D. Esta sección cubre algunas de las características y técnicas menos utilizadas.

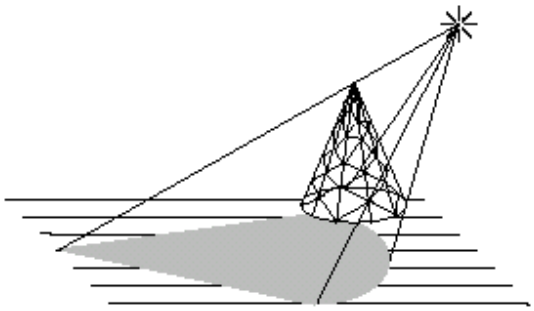
Objetos Brillantes-en-la-Oscuridad

El objeto **Material** permite la especificación de un color emisivo. Esto se puede utilizar para crear el efecto de un objeto brillante en la oscuridad. Tener un color emisivo no hace del objeto visual una fuente de luz; no iluminará otros objetos visuales. El **Material** Emisivo es también útil en aplicaciones especiales, tales como indicar un objeto especial o un objeto que se ha escogido.
La Figura 6-26 muestra la escena del programa del ámbito de luz donde se le ha dado color emisivo al rectángulo no iluminado. Comparemos esta imagen con la imagen izquierda de la Figura 6-25. Como podemos ver, el uso del color emisivo sólo modifica al objeto visual que lo tiene. Este ejemplo también demuestra que el uso eficaz del color emisivo, como con la mayoría de los parámetros de la iluminación, es más un problema artístico que de programación.



Calcular Sombras

La complejidad de calcular sombras es tan grande que no forma parte de ningún sistema de gráficos en tiempo real. La complejidad viene de cálcular si la fuente de luz alcanza o no un vértice. Todo polígono de otro objeto visual debe ser considerado al calcular la respuesta.



El sombreado es mucho más complejo en realidad. Las fuentes de luz no son fuentes puramente direccionales ni perfectas. Por lo tanto, las sombras no tienen bordes sostenidos. Ignorando la realidad, como hacemos a menudo en gráficos, echemos una ojeada a las formas de simular sombras.

Crear Sombras

Hay dos partes básicas al simular (o al falsificar) sombras: calcular donde están las sombras, y crear geometrías (o texturas) para servir como sombras. Hay varias maneras de calcular la localización de la sombra, pero los detalles de las distintas técnicas de sombreado están más allá del alcance de esta guía. Las dos secciones siguientes cubren dos técnicas generales para crear el contenido de la sombra.

Sombrear Polígonos

Un polígono especificado sin propiedades **Material** se puede utilizar como polígono de sombra, llamado un polígono sombra coloreado. El color del polígono sombra, especificado por geometría o con un objeto **ColoringAttributes**, se elige para aparecer como el objeto en sombra. Los polígonos sombra especificados de esta manera pueden parecer falsos en escenas complejas.

Los polígonos sombra especificados con las características **Material** pero fuera de la influencia de uno o más objetos de luz se llaman polígonos sombra sombreados. Los polígonos sombra son sombreados por los objetos de luz que los influencian tal que parecen más realistas. Obviamente, especificar un polígono sombra sombreado es más complejo que especificar un polígono sombra coloreado.
No importa cómo se especifique un polígono sombra, la posición del polígono sombra es justo arriba, o en frente de, el polígono al que da sombra. Mientras que la adición de polígonos sombra no da lugar normalmente a más polígonos para renderizar (debido a la obstrucción de otros polígonos) crea más objetos en el universo virtual lo que puede degradar el funcionamiento de la renderización.
En vez de crear los polígonos sombra, las sombras pueden crearse cambiando la influencia de luces para excluir polígonos 'en la sombra'. El ámbito de luces es útil para este propósito. Sin embargo, puesto que la influencia se determina en base al objeto, puede ser complejo calcular cómo subdividir los objetos visuales que se sombrean parcialmente.

Sombrear Texturas

Como las sombras anteriores, las sombras naturales son complejas. Una sombra natural raramente tiene un borde recto y una sombra constante. Se puede usar el texturado para hacer sombras más realistas. Hay dos maneras básicas de usar texturado para crear sombras: aplicando textura a los polígonos sombra, o la aplicación de texturas a los objetos visuales.
Como el texturado no se ha cubierto todavía (Capítulo 7), y el cálculo de las texturas de la sombra (incluso off-line) es difícil (y va más allá del alcance de esta guía) este es un tema pendiente para otro libro.

Mover Objetos, Mover Sombras

Debemos tener presente que la adición de sombras a una aplicación hace la aplicación mucho más compleja. Por ejemplo, cuando un cubo con una sombra gira, la sombra gira y se deforma. Para esa materia, las luces móviles hacen que

las sombras se muevan también. En cualquier caso, el movimiento agrega otro nivel de complejidad a la programación de sombras.

Programa de Ejemplo de Sombras

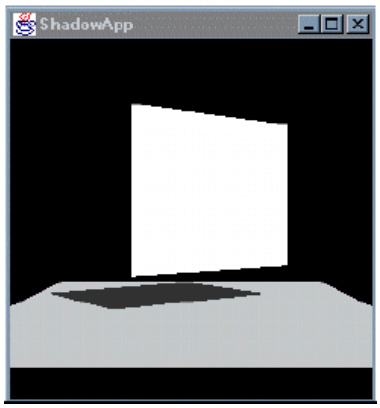
El programa ShadowApp.java da un ejemplo de cómo se pueden crear polígonos sombra sencillos. El programa define una clase para crear los polígonos sombra. La clase sombra crea un polígono sombra para cada geometría dada como entrada de información. El Fragmento de Código 6-3 muestra la clase **SimpleShadow** usada para crear polígonos sombra en ShadowApp.java. La Figura 6-28 muestra la escena renderizada con una sombra.

Fragmento de Código 6-3 Clase Shadow para Crear Polígonos Sombreados.

```
1.  public class SimpleShadow extends Shape3D {
2.  SimpleShadow(GeometryArray geom, Vector3f direction,
3.  Color3f col, float height) {
4.
5.  int vCount = geom.getVertexCount();
6.  QuadArray poly = new QuadArray(vCount, GeometryArray.COORDINATES
7.  | GeometryArray.COLOR_3
8.  );
9.
10. int v;
11. Point3f vertex = new Point3f();
12. Point3f shadow = new Point3f();
13. for (v = 0; v < vCount; v++) {
14. geom.getCoordinate(v, vertex);
15. shadow.set( vertex.x + (vertex.y-height) * direction.x,
16. height + 0.0001f,
17. vertex.z + (vertex.y-height) * direction.y);
18. poly.setCoordinate(v, shadow);
19. poly.setColor(v, col);
20. }
21.
22. this.setGeometry(poly);
```

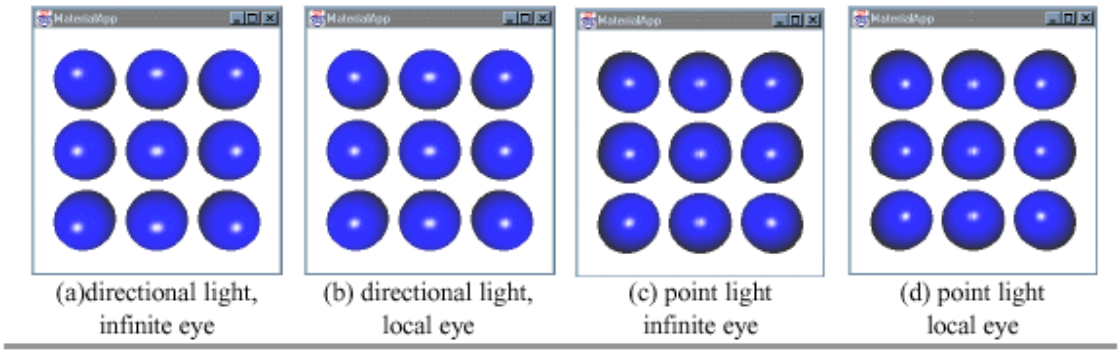
23. }

Varias asunciones hechas en la clase **SimpleShadow** (para hacerlo fácil) limitan las aplicaciones de esta clase. **SimpleShadow** está limitada en que: proyecta solamente a los planos, sólo considera una luz, sólo hace algunas orientaciones, no considera las dimensiones del plano sobre el que se está proyectando. Escribir una clase de fines generales para el cálculo de la sombra es una empresa importante.



- **Tópico Avanzado: El Papel del Objeto View en el Sombreado**

La vista (o las vistas) asociadas a un escenario gráfico juegan una gran variedad de papeles en cómo se renderiza una escena. Esta sección no explica todos los papeles del objeto **View**. La especificación del API Java 3D proporciona una referencia completa a la clase **View**. Esta sección menciona solamente dos métodos de la clase **View** útiles para entender el sombreado de objetos visuales. Según lo mencionado en la sección "Vectores de ojo local contra ojos infinito", el vector del ojo es constante como valor por defecto. Esto se conoce como un vector de ojo infinito. Es decir, la escena se renderiza como si fuera vista desde el infinito. Tener un ojo infinito reduce perceptiblemente el cálculo de renderización. Sin embargo, la imagen que resulta puede parecer incorrecta. La Figura 6-29 muestra las imágenes renderizadas a partir de una escena usando un ojo infinito y un ojo local usando diversas fuentes de luz.



(a)directional light, infinite eye | (b) directional light, local eye | (c) point light infinite eye | (d) point light local eye

Para apreciar completamente las imágenes de la Figura 6-29 necesitamos conocer la geometría de la escena. La escena son nueve esferas en una organización planar. Cada una de las imágenes se ve con el mismo campo visual desde la misma posición. Las únicas variables son si la luz es un **DirectionalLight** o un **PointLight**, y si el ojo es infinito o local. El **DirectionalLight** tiene dirección (0, 0, -1), el **PointLight** se coloca en (0, 0, 1).

Las imágenes (a) y (c) de la Figura 6-29 se renderizan con un ojo infinito. En estas imágenes, los vectores del ojo son constantes, así que las reflexiones especulares están básicamente en la misma posición para cada esfera. Las imágenes (b) y (d) de la Figura 6-29 se renderizan con un ojo local. Los vectores del ojo varían en estas imágenes, así que las reflexiones especulares están en distinta posición para cada esfera. Observemos también que la reflexión difusa (azul) en las esferas varía sólo con la fuente de luz. El vector del ojo sólo desempeña un papel en el cálculo de la reflexión especular.

Una vez más la característica de la visión del ojo infinito se utiliza para reducir el cálculo, y por lo tanto el tiempo de la renderización. La imagen (a) de la Figura 6-29 tarda un menor tiempo para renderizarse y la imagen (d) tarda el mayor tiempo. Las imágenes (b) y (c) tardan una cantidad casi igual de tiempo, que es menor que el tiempo de la imagen (d), pero mayor que el de la imagen (a). El tiempo real para renderizar varía con el sistema utilizado. La diferencia es más pronunciada en los sistemas que renderizan por software.

Lista Parcial de Métodos (Relacionados con el Sombreado) de la Clase **View**

El objeto **View** contiene todos los parámetros necesarios para rendereizar una

escena tridimensional desde un punto de vista.

void setLocalEyeLightingEnable(boolean flag)

Selecciona una bandera que india si se usa el ojo local para calacular las proyecciones de perspectivas.

void setWindowEyepointPolicy(int policy)

Selecciona la política del modelo de vista de ojo de la ventana a uno de :

- RELATIVE_TO_FIELD_OF_VIEW,
- RELATIVE_TO_SCREEN, RELATIVE_TO_WINDOW

El objeto **View** se puede conseguir desde un **SimpleUniverse** usando los métodos apropiados. Entonces el objeto **View** se puede manipular como en el siguiente ejemplo:

```
SimpleUniverse su = new SimpleUniverse(canvas);
su.getViewer().getView().setLocalEyeLightingEnable(true);
```

Texturas en Java 3D

El aspecto de muchos objetos del mundo real depende de su textura. La textura de un objeto es realmente la geometría relativamente fina de la superficie de un objeto. Para apreciar la superficie del papel las texturas juegan con el aspecto de los objetos del mundo real, consideremos una alfombra. Incluso cuando todas las fibras de una alfombra son del mismo color la alfombra no aparece con un color constante debido a la interacción de la luz con la geometría de las fibras. Aunque Java 3D es capaz de modelar la geometría de las fibras individuales de la alfombra, los requisitos de memoria y el funcionamiento de la renderización para una alfombra del tamaño de una habitación modelada a tal detalle harían dicho modelo inútil. Por otra parte, tener un polígono plano de un solo color no hace un reemplazo convincente para la alfombra en la escena renderizada.

Hasta ahora en el tutorial, los detalles de los objetos visuales los ha proporcionado la geometría. Consecuentemente, los objetos visualmente ricos, como los árboles,

pueden requerir mucha geometría que a cambio requiere mucha memoria y cálculo de renderización. A cierto nivel de detalle, el rendimiento puede llegar a ser inaceptable. Este capítulo muestra cómo añadir el aspecto del detalle superficial a un objeto visual sin la adición de más geometría con el uso de texturas

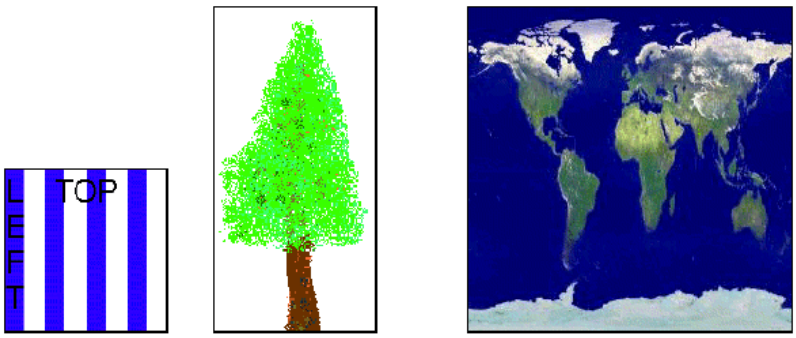
¿Qué es el Texturado?

Una alfombra puede ser un ejemplo extremo en términos de complejidad y de densidad de la geometría superficial, pero está lejos de ser el único objeto para el cual percibimos textura. Los ladrillos, el cemento, la madera, los céspedes, las paredes, y el papel son sólo algunos de los objetos que no se representan bien con polígonos planos (no-texturados). Pero, igual que con la alfombra, el coste de representar la textura superficial en los primitivos geométricos para estos objetos sería muy alto.

Una posible alternativa a modelar la fibra de la alfombra es modelar la alfombra como un polígono plano con muchos vértices, asignando colores a los vértcies para darle variaciones de color. Si los vértices están suficientemente cercanos, se puede reproducir la imagen de la alfombra. Esto requiere significativamente menos memoria que el modelo que incluye las fibras de la alfombra; sin embargo, el modelo todavía requiere demasiada memoria para un tamaño de habitación razonable. Esta idea, de representar la imagen del objeto en una superficie plana, es la idea básica de texturado. Sin embargo, con el texturado, la geometría puede ser muy sencilla.

El texturado, también llamado mapeo de textura, es una manera de añadir riqueza visual a una superficie sin la adición de los detalles geométricos finos. La riqueza visual la proporciona una imagen, también llamada textura, que da el aspecto del detalle superficial para el objeto visual. La imagen se mapeda dentro de la geometría del objeto visual en el momento de la renderización. De ahí el término mapeo de textura.

La Figura 7-1 muestra algunas de las texturas usadas en los programas de ejemplo de este capítulo. Como podemos ver, una textura puede proporcionar riqueza visual a objetos de distintos tamaños.



Texturado Básico

El texturado de polígonos en un programa de Java 3D se consigue a través de la creacción del manojo de apariencia apropiado y cargando la imagen de la textura dentro de él, especificando la localización de la imagen de la textura en la geometría, y fijando los atributos de texturado. Como veremos, especificar texturas puede ser muy complejo. Afortunadamente, hay clases de utilidad para ayudarnos en el proceso y las configuraciones de los valores por defecto para las opciones texturado son las apropiadas para las aplicaciones de texturado básicas.
Para explicar el texturado, la siguiente sección presenta una sencilla receta; y en secciones posteriores se desarrolla un programa de ejemplo basado en la receta, además de explicar el texturado.

Sencilla Receta de Texturado

Debido a la flexibilidad del texturado en el API Java 3D, el número de opciones relacionadas con el texturado puede ser un poco fastidioso. Incluso así, el texturado no es necesariamente difícil. Para hacer fácil el trabajo de especificación de textura, seguimos los pasos de la siguiente receta.

La receta solo subraya los pasos relacionados directamente con el texturado. Deberíamos haber observado que la geometría y la apariencia se fijan en un objeto **Shape3D** que se agrega al escenario gráfico.

1. Preparar las Imágenes de Texura
2. Cargar la Textura
3. Configurar la textura en el manojo Appearance
4. Especificar las TextureCoordinates del Geometry

Al igual que muchas de las recetas de este tutorial, algunos de estos pasos se pueden realizar en cualquier orden. De hecho, los pasos de esta receta se pueden realizar en cualquier orden (siempre que los los pasos 2 y 3 se hagan juntos).

Texturado Paso 1: Preparar las Imágenes de Texturas

Esta receta comienza con un paso sin programación: "preparar las imágenes de textura". Crear y corregir imágenes de textura es algo que normalmente se hace externamente a los programas de Java 3D. De hecho, la mayoría de las imágenes de textura están preparadas antes de que se comience el programa. Hay dos tareas esenciales en la preparación de la imagen de textura: 1. asegurarnos de que las imágenes sean de dimensiones aceptables, y 2. asegurarnos de que las imágenes se graban en un formato de fichero que pueda ser leído. Por supuesto se puede editar la imagen para alcanzar el color, la transparencia, y las características deseadas.

Por razones de eficiencia , Java 3D necesita que el tamaño de la imagen de la textura sea una potencia matemática de dos (1, 2, 4, 8, 16, ...) en todas las dimensiones. No cumplir esta restricción dará lugar a una excepción en tiempo de ejecución.

Si una imagen no es de las dimensiones aceptables, debe ser modificada (escalada o recortada) para cumplir los requisitos de dimensión antes de que sea utilizada. La edición de la imagen se puede hacer en una gran variedad de programas incluyendo el **API Java Advanced Imaging**. En la Figura 7-1, las dos imágenes más pequeñas son de 128 por 128, el árbol es de 256 por 128, y la tierra es de 256 por 256.

En lo que concierne a los formatos de fichero, se puede utilizar cualquier formato de fichero siempre que proporcionemos un métodos para cargarlo. Los programas de este capítulo cargan texturas usando la clase de utilidad **TextureLoader**. Un objeto **TextureLoader** carga JPEG, GIF, y otros formatos de fichero.
Una palabra más sobre los programas del ejemplo antes de pasar al paso siguiente. Los fragmentos del código y los programas de ejemplo de este capítulo utilizan los nombres del archivo para algunos ficheros de imagen que están incluidos en los ficheros Jar de ejemplo. No hay nada especial en estos ficheros de imagen a excepción de que cumplen con la restricción de las dimensiones. Cualquier fichero de imagen se puede utilizar en los programas simpre que las dimensiones de las imágenes sean potencia de dos . Podemos compilar y ejecutar los programas del ejemplo con nuestros propios ficheros de a imagen. Ahora, con las imágenes de texturas listas, podemos empezar la programación.

Texturado Paso 2: Cargar la Textura

El siguiente paso es conseguir la imagen preparada en un objeto imagen. Esto se conoce como cargar la textura. Las texturas se pueden cargar desde ficheros o URLs usando de mismo proceso básico. Cargar una textura se puede lograr con muchas líneas del código, o con dos líneas de código que utilicen un objeto **TextureLoader**. De cualquier forma, el resultado es conseguir la imagen en un objeto **ImageComponent2D**. El Fragmento de Código 7-1 muestra un ejemplo de dos líneas que utilizan un **TextureLoader**. El resultado de estas dos líneas es cargar la imagen del fichero stripe.gif en un objeto **Image2DComponent** que se pueda utilizar para crear el manojo de apariencia necesario para el paso 4.

Fragmento de Código 7-1, usar un objeto TextureLoader Object para Cargar el fichero de imagen STRIPE.GIF

```
1. TextureLoader loader = new TextureLoader("stripe.gif", this);
2. ImageComponent2D image = loader.getImage();
```

Antes de pasar al paso 4 de la receta, echemos una ojeada más cercana el uso del objeto **TextureLoader**. El segundo argumento del constructor especifica un objeto que sirva como image observer. La clase **TextureLoader** utiliza el paquete

java.awt.image para cargar las imágenes. Este paquete carga las imágenes de forma asíncrona, lo que es particularmente útil cuando una imagen se carga de una URL. Para facilitar el manejo de cargas asíncronas de imágenes, los componentes de AWT están capacitados para ser observadores de imagen, que es observar el proceso de la carga de la imagen. A un observador de imagen se le puede preguntar por los detalles de la carga de la imagen.

Con el fin escribir programas Java 3D todo lo que necesitamos saber es que cualquier componente del AWT puede servir como un observador de imagen. Puesto que **Applet** es una extensión del componente **Panel** del AWT, el objeto **Applet** de un programa de Java 3D puede ser el observador de imagen para el objeto **TextureLoader**.

Texturado Paso 3: Crer el Manojo de Appearance

Para ser utilizada como textura para un objeto visual, la imagen de textura cargada en el paso de 2a se debe asignar como la textura de un objeto **Texture**, que entonces se utiliza en un manojo de apariencia referenciado por el objeto visual. Específicamente, un objeto **Texture2D** contiene la imagen de la textura. La imagen **ImageComponent2D** cargada en el paso 2a es el centro de la creación del manojo de apariencia del paso 2b.

El Fragmento de Código 7-2 muestra dos líneas del código del paso 2a seguidas por el código para formar un sencillo manojo de apariencia texturado. Cargando la textura (líneas 1 y 2), la imagen entonces se asigna al objeto **Texture2D** (línea 4). Luego el objeto **Texture2D** se agrega al objeto **Appearance**(línea 6).

Fragmento de Código 7-2, Crear un Appearance con un objeto Texture.

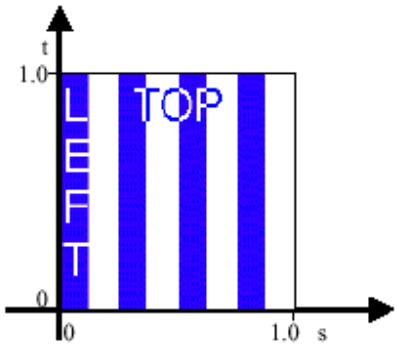
```
1.  TextureLoader loader = new TextureLoader("stripe.jpg", this);
2.  ImageComponent2D image = loader.getImage();
3.  Texture2D texture = new Texture2D();
4.  texture.setImage(0, image);
5.  Appearance appear = new Appearance();
6.  appear.setTexture(texture);
```

El manojo de apariencia creado en el Fragmento de Código 7-2 podría tener otros nodos componentes, la más notable de las posibilidades es el nodo componente **TextureAttributes**. Para este ejemplo, no se utiliza ningún objeto **TextureAttributes**.

Texturado Paso 4: Especificar TextureCoordinates

Además de cargar la textura en un manojo de apariencia, el programador también especifica la colocación de la textura en la geometría a través de la especificación de las coordenadas de textura. Las especificaciones de coordenadas de textura se hacen por cada vértice de la geometría. Cada coordenada de textura especifica un punto de textura que se aplicará al vértice. Con la especificación de algunos puntos de la imagen que se aplicarán a los vértices de la geometría, la imagen será rotada, estirada, aplastada, y/o duplicada para hacer que quepa en la especificación.

TextureCoordinates se especifica en las dimensiones s (horizontal) y t (verticales) de la imagen de textura según lo mostrado en la Figura 7-3. Esta figura muestra las coordenadas de la textura en el espacio de la imagen de textura.



El siguiente bloque de referencia muestra sólo uno de los métodos de **GeometryArray** disponible para fijar coordenadas de textura.

Método setTextureCoordinate de **GeometryArray**

Las coordenadas de textura se especfican por cada vértice de la geometría mediante uno de los distintos métodos setTextureCoordinate de la clase **GeometryArray**.

void setTextureCoordinate(int index, Point2f texCoord)

Selecciona las coordenadas de textura asociadas con el vértice del índice especificado para este objeto.

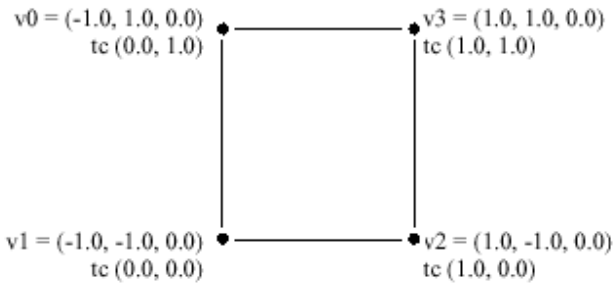
El Fragmento de Código 7-3 crea un solo plano usando un objeto de geometría **QuadArray**. Los coordenadas de la textura se asignan para cada vértice. En el código, las líneas tres a once establecen las cuatro esquinas de un cuadrángulo en 3-espacios. Las líneas 13 a 21 establecen la localización de la textura en la geometría. Este fragmento determinado de código crea un plano de 2 metros en una cara y pone la imagen de la textura en la orientación normal (hacia arriba, no invertido) a lo largo de la cara del plano.

Fragmento de Código 7-3, Aplicar Coordenadas de Texturas a un Quad.

```
1.  QuadArray plane = new QuadArray(4, GeometryArray.COORDINATES
2.                                  | GeometryArray.TEXTURE_COORDINATE_2);
3.  Point3f p = new Point3f();
4.  p.set(-1.0f, 1.0f, 0.0f);
5.  plane.setCoordinate(0, p);
6.  p.set(-1.0f, -1.0f, 0.0f);
7.  plane.setCoordinate(1, p);
8.  p.set( 1.0f, -1.0f, 0.0f);
9.  plane.setCoordinate(2, p);
10. p.set( 1.0f, 1.0f, 0.0f);
11. plane.setCoordinate(3, p);
12.
13. Point2f q = new Point2f();
14. q.set(0.0f, 1.0f);
15. plane.setTextureCoordinate(0, q);
16. q.set(0.0f, 0.0f);
17. plane.setTextureCoordinate(1, q);
18. q.set(1.0f, 0.0f);
19. plane.setTextureCoordinate(2, q);
20. q.set(1.0f, 1.0f);
21. plane.setTextureCoordinate(3, q);
```

La Figura 7-4 muestra la relación entre las coordenadas de vértice y las coordenadas de la textura para el cuadrángulo del ejemplo creado en el

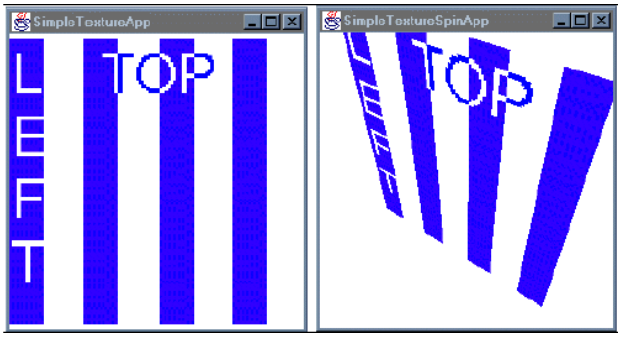
Fragmento de Código 7-3. La imagen izquierda de la Figura 7-5 muestra la aplicación de la textura **stripe.gif** a la geometría del ejemplo.



Ahora que hemos completado los tres pasos del texturado, el objeto texturado puede añadirse a un escenario gráfico. La siguiente sección presenta una serie de programas de ejemplo que muestra algunas opciones de texturado.

- **Sencillos Ejemplos de Programas de Textura**

Siguiendo la receta anterior, se ha desarrollado un sencillo programa de ejemplo de texturado. SimpleTexturaApp.java. Este programa es poco más que un programa de visualización de imágenes.



Se ha creado otro programa de ejemplo SimpleTexturaSpinApp.java utilizando un objeto **RotationInterpolator**. En este programa se hace girar el mismo plano texturado para demostrar la naturaleza 3D del programa. La Figura 7-5 muestra una renderización de este programa a la derecha. Una cosa a observar cuando se ve el programa, la cara posterior del plano está en blanco.

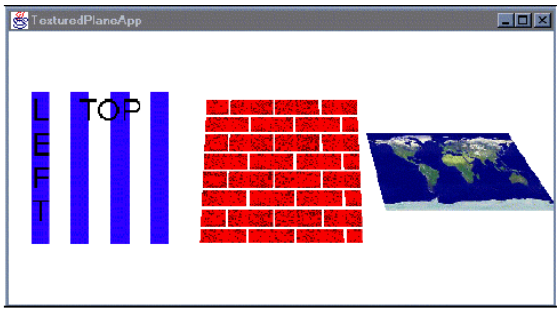
La Clase NewTextureLoader

Los programas Java 3D que usan texturas pueden tener una gran cantidad de líneas sólo para cargar las texturas y crear los manojos de apariencia. Se puede ahorrar algo de programación y, más importante, memoria en tiempo de ejecución compartiendo manojos de apariencia cuando sea apropiado. Sin embargo, esto no reduce mucho la cantidad de programación. Se pueden conseguir otras reducciones de programación creando una clase para crear los manojos de apariencia de textura. El desafío de crear esta clase consiste en el requisito del observador de imagen para el objeto **TextureLoader**.

El objeto **Canvas3d** o un **Applet** pueden servir como el observador de imagen, pero tener una referencia a un cierto componente por todas partes en el programa puede ser fastidioso. Para tratar esta inconveniencia, se ha extendido la clase **TextureLoader** que elimina la necesidad de un componente observador de imagen. En su lugar se utiliza un solo método para especificar un observador de imagen para todas las aplicaciones futuras del cargador de textura.

Los constructores de **NewTextureLoader** son iguales a los de **TextureLoader** excepto en que ninguno requiere un componente observador de imagen. Los métodos de **NewTextureLoader** son los mismos de **TextureLoader** con el método adicional para fijar un observador de imagen.

Otro programa del ejemplo,TexturaPlaneApp.java, carga tres texturas y las visualiza en los planos según lo mostrado en la Figura 7-6. Lo importante de este programa es que las texturas se cargan usando la clase **TexturedPlane** definida externamente al resto del programa que se hace más fácilmente con la clase **NewTextureLoader**. Esta clase **TexturedPlane** no es lo bastante flexible para ser utilizada en muchas aplicaciones, pero sirve como demostración para clases similares.



El Fragmento de Código 7-4 es un extracto de TexturaPlaneApp.java y es casi todo el código necesario para crear los tres planos texturadps de esta aplicación. El objeto observador de imagen se proporciona al objeto de **NewTextureLoader** del **TexturedPlane**.

Fragmento de Código 7-4, Añadir tres objetos Texturados a un Escenario Gráfico.

```
1.  scene.addChild(tg0);
2.  tg0.addChild(tg1);
3.  tg1.addChild(new TexturedPlane("stripe.gif"));
4.
5.  tg0.addChild(tg2);
6.  tg2.addChild(new TexturedPlane("brick.gif"));
7.
8.  tg0.addChild(tg3);
9.  tg3.addChild(new TexturedPlane("earth.jpg"));
```

Más sobre las Coordenadas de Textura

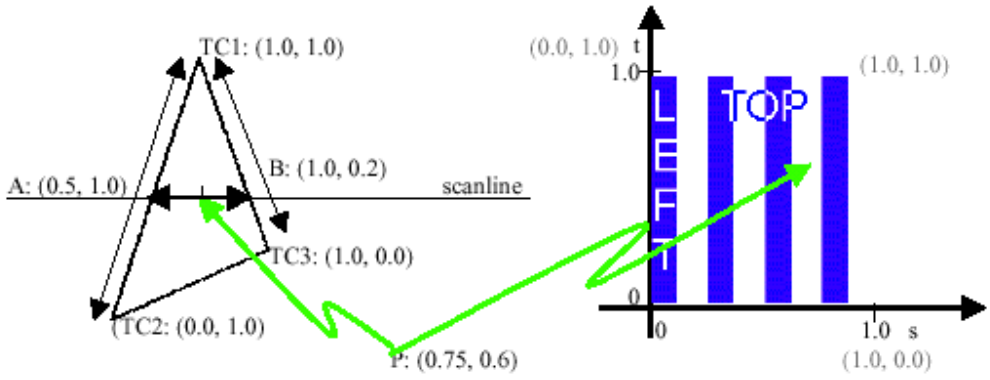
Según lo mencionado en "Texturado Paso 3: Especificar TextureCoordinates", la imagen de la textura se crea para caber en la geometría basándose en la especificación de las coordenadas de textura. El proceso real es asociar los texels de la textura a los pixeles de la geometría cuando es renderizada. Cada pixel de una textura se llama un texel, o un 'elemento de textura'. Éste es el proceso de mapeado de la textura.

El mapeo de textura comienza con la especificación de las coordenadas de textura para los vértices de la geometría. Mientras se renderiza cada pixel del triángulo texturado, se calculan las coordenadas de la textura en el pixel desde los vértices del triángulo. La interpolación Trilinear de las coordenadas de textura de los

vertices determina las coordenadas de textura para el pixel y por lo tanto, el texel de la imagen de la textura usada en el color final del pixel.

La Figura 7-7 ilustra el proceso de interpolación trilinear para un pixel del ejemplo. La representación se hace en orden de scan. El pixel, P, para el mapeo de textura está justo en el centro del scan actual en el triángulo de la izquierda de la ilustración. Se han asignado las coordenadas de textura para cada uno de los vértices del triángulo. Se han etiquetado TC1, TC2, y TC3. Estas coordenadas de textura son el punto de partida para la interpolación trilinear (cada una de las interpolaciones lineares se muestra como flechas dos-cabezas en la figura). Las primeras dos interpolaciones lineares determinan las coordenadas de la textura a lo largo de las caras del triángulo en el scan (puntos etiquetados A y B en la figura). La tercera interpolación se hace entre estos dos puntos.

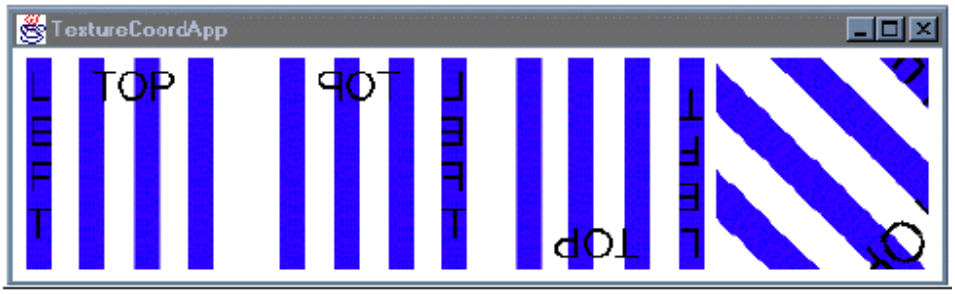
Las coordenadas de textura que resultan para P son (0,75, 0,6). En la derecha de la figura está la textura. Usando las coordenadas de textura calculadas para P se selecciona el texel.



La selección del Texel no se explica completamente en el ejemplo anterior. La especificación de la sección de filtración (ver secciones posteriores) da más detalles sobre la selección del texel. Otro detalle todavía no explicado es la interacción entre el color del texel, otras fuentes del color, y el color final del pixel. El modo de valor por defecto es 'substituye' con el cuál se utiliza el color del texel como el color del pixel, pero hay otros modos. Antes de pasar a otros asuntos, ya está en orden la discusión adicional de las coordenadas y el mapeo de textura.

En este punto del capítulo se han utilizado todas las texturas en su orientación ordinaria. La Figura 7-8 muestra planos con algunas de las posibles orientaciones

de textura sólo seleccionando las coordenadas de textura en los vértices. El programa de ejemplo TextureCoordApp.java produce esta imagen.



Debemos observar que en el programa de ejemplo TextureCoordinatesApp.java la textura **stripe.gif** se carga solamente una vez. Se crea solamente un manojo de apariencia de la textura que es compartido por los cuatro planos texturados. Esto es posible porque no hay nada en el manojo de la textura que sea único para cualquiera de los planos. Cargar la textura una vez ahorra tiempo y memoria.

Por supuesto, se pueden cometer errores al especificar las coordenadas de textura. Cuando sucede esto, el sistema de renderizado de Java 3D hace que se pregunte por él. Cuando las coordenadas de textura no se especifican para un espacio regularmente mapeado, entonces la triangulación de la geometría llega a ser obvia pues las 'costuras' de la textura se verán a lo largo de los bordes de triángulos.

La Figura 7-9 muestra la imagen renderizada de los planos texturados donde no se especifican las coordenadas de textura para hacer una presentación uniforme de la textura. El programa que genera esta imagen, TexturaRequestApp.java, es solamente un manojo de textura compartido por tres objetos visuales. Las variaciones en el aspecto de los planos sólo son debidas a la especificación de las coordenadas de textura. Ésta es una representación de la frase "en texturado, se consigue lo que se pide".

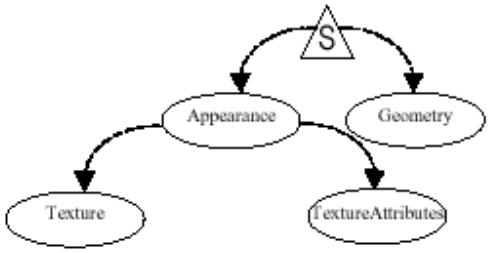
Este programa muestra algunos de los posibles renderizados para un plano usando la misma textura. Las asignaciones de textura hechas en este programa son ejemplos de posibles errores mientras que todas son aplicaciones legítimas. La imagen de la izquierda es una aplicación de solo una sola fila de texels - las mismas coordenadas de textura se asignan a las parejas de vértices. La imagen de la derecha es la aplicación de un solo texel - las cuatro coordenadas de textura son iguales. Las dos imágenes del centro muestran la asignación de las coordenadas de textura de maneras no uniformes. El cambio de la textura a lo largo de la diagonal es debido a la triangulación del polígono.
El Fragmento de Código 7-5 muestra las coordenadas de textura asignadas en la aplicación TexturaRequestApp.java. Estas asignaciones usadas con la textura **stripe.gif** resultan en las imágenes de la Figura 7-9.

Fragmento de Código 7-5, Asignaciones de Coordenadas de Textura para Planos en TextureRequestApp.

```
1.  // texture coordinate assignments fof the first plane
2.  texturedQuad.setTextureCoordinate(0, new Point2f( 1.0f, 0.0f));
3.  texturedQuad.setTextureCoordinate(1, new Point2f( 1.0f, 0.0f));
4.  texturedQuad.setTextureCoordinate(2, new Point2f( 0.0f, 0.0f));
5.  texturedQuad.setTextureCoordinate(3, new Point2f( 0.0f, 0.0f));
6.  // texture coordinate assignments for the second plane
7.  texturedQuad.setTextureCoordinate(0, new Point2f( 0.0f, 1.0f));
8.  texturedQuad.setTextureCoordinate(1, new Point2f( 1.0f, 0.5f));
9.  texturedQuad.setTextureCoordinate(2, new Point2f( 0.5f, 0.5f));
10. texturedQuad.setTextureCoordinate(3, new Point2f( 0.0f, 1.0f));
11. // texture coordinate assignments for the third plane
12. texturedQuad.setTextureCoordinate(0, new Point2f( 1.0f, 0.0f));
13. texturedQuad.setTextureCoordinate(1, new Point2f( 1.0f, 1.0f));
14. texturedQuad.setTextureCoordinate(2, new Point2f( 0.0f, 0.0f));
15. texturedQuad.setTextureCoordinate(3, new Point2f( 1.0f, 1.0f));
16. // texture coordinate assignments for the forth plane
17. texturedQuad.setTextureCoordinate(0, new Point2f( 0.0f, 0.0f));
18. texturedQuad.setTextureCoordinate(1, new Point2f( 0.0f, 0.0f));
19. texturedQuad.setTextureCoordinate(2, new Point2f( 0.0f, 0.0f));
20. texturedQuad.setTextureCoordinate(3, new Point2f( 0.0f, 0.0f));
```

Preview de Algunas Opciones de Texturado

Hay mucho más en el texturado que sólo especificar las coordenadas de textura para los vértices de la geometría. En este punto, la discusión de texturado no ha incluido ninguna de las opciones disponibles en aplicaciones de textura. Por ejemplo, el objeto **Texture2D** se puede configurar para diversos modos de límites y filtros de mapeo. Pero hay inlcuso más que esto.
La configuración adicional de una textura se hace a través de un componente del nodo **TextureAttributes**. La Figura 7-10 muestra un objeto visual con un manojo de apariencia con los componentes **Texture** y **TextureAttributes**.



Otras opciones de texturado van más allá de las configuraciones **Texture** y **TextureAttributes**. Por ejemplo, una textura puede ser tridimensional. En secciones posteriores veremos el API para la clase **Texture3d**, que, como **Texture2D**, es una extensión de la clase **Texture**.

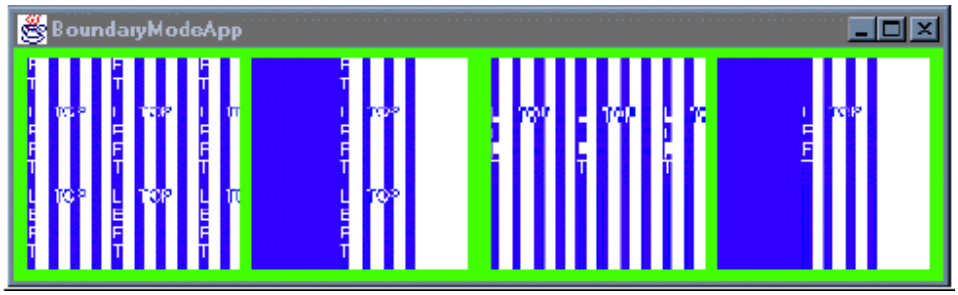
Opciones de Textura

Texture2D, la clase usada en los ejemplos anteriores, es una extensión de **Texture**. Algunas de las opciones básicas para el texturado se implementan en esta clase. Puesto que **Texture** es una clase abstracta, sus configuraciones se harán a través de un objeto **Texture2D** o **Texture3d**. Las configuraciones son el modo de límites, filtros, y el formato de la textura.

Modo de Límites: Envolver o Abrazar

En todos los programas anteriores, se han asociado las texturas de una manera tal que una copia de la imagen se ha utilizado para cubrir el plano. El problema es qué hacer cuando una coordenada de textura está más allá del rango 0 a 1 del espacio de la textura y no fue direccionada.

La configuración del modo de límites determina lo que ocurre cuando el mapeo tiene lugar si las coordenadas de textura van más allá del rango 0 a 1 del espacio de la imagen. Las opciones son envolver la imagen, o abrazar la imagen. Envolver, significa repetir la imagen según sea necesario, es el valor por defecto. Abrazar utiliza el color del borde de la imagen en cualquier lugar fuera del rango 0 a 1. Estas configuraciones se hacen independientemente en las dimensiones s y t. En el programa del ejemplo BoundaryModeApp la textura se asocia sobre aproximadamente la novena parte de cada uno de los planos. La Figura 7-11 muestra la escena según lo renderizado por este programa. Las variaciones en las imágenes se deben solamente a la configuración de los modos de límites para los planos. De izquierda a derecha las configuraciones son (s luego t) WRAP y WRAP, CLAMP y WRAP, WRAP y CLAMP, y CLAMP y CLAMP.



Observa que al contrario que las aplicaciones anteriores que comparten el mismo objeto **Texture** entre cuatro objetos visuales, la textura se carga cuatro veces en esta aplicación. Esto es necesario puesto que cada uno de los objetos **Texture** tiene diversas combinaciones de las configuraciones del modo de límites.

Especificación de Filtrado

En el cálculo de las coordenadas de textura para cada pixel, raramente hay una correspondencia del pixel directamente a un sólo texel. Normalmente un pixel es del tamaño de varios texels o más pequeño que un texel. En el primer caso se utiliza un filtro de ampliación para asociar varios texels a un pixel. En el segundo caso se utiliza un filtro de reducción para asociar el texel o los texels a un pixel. Hay opciones para manejar cada uno de estos casos.

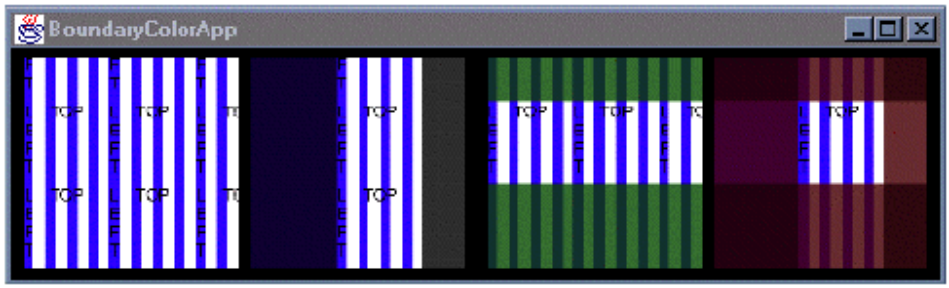
El filtro de ampliación especifica qué hacer cuando un pixel es más pequeño que un texel. En este caso la textura será ampliada como si se aplicara sobre la geometría. Cada texel aparecerá como varios pixels y es posibles que la imagen resultante exhiba el "texelization" donde se verían los texels individuales para la renderización. Las opciones para el filtro de ampliación son hacer el punto de muestreo, que es seleccionar el texel más cercano y utilizar su color, o interpolarlo entre texels vecinos. El punto de muestreo, o muestreo del vecino más cercano, generalmente tiene menor coste de cálculo; mientras que el muestreo linear de la interpolación cuesta más (en cálculo y por lo tanto en tiempo de renderización) pero reduce el aspecto de cualquier texelization.
El filtro de reducción especifica qué hacer cuando un pixel es más grande que un texel. En este caso los texels deben ser "reducidos" para caber en el pixel. El problema trata en que un pixel puede solamente tener un valor de color y varios texels podrían suministrarlo. Las opciones para el filtro de reducción son hacer el punto de muestreo, que es seleccionar el texel más cercano y utilizar su color, o interpolarlo entre texels vecinos.
No está siempre claro qué filtro será utilizado. Consideremos una textura estirada en una dirección pero aplastada en otra. Dependiendo de qué dimensión see considera, se escogerá un filtro diferente. No hay nada que el programador pueda hacer para determinar cuál será utilizado. Sin embargo, el sistema de ejecución normalmente escoge el filtro que resulte en una imagen mejor.
La selección de un filtro tiene otras implicaciones con respecto al uso del color de límite (siguiente sección). También, las opciones del filtro de reducción son más complejas cuando se usan varios niveles de texturado.

Color de Limites

El comportamiento del modo de límites es además configurable con un color del límite. Cuando el modo de límite es CLAMP y se especifica color de límite, se utiliza el color del límite cuando las coordenadas de la textura están fuera del rango 0 a 1. Solo se puede especificar un color de límite para utilizar un color en

cada dimensión para la cual el modo de límite se fije a CLAMP. El programa de ejemplo BoundaryColorApp demuestra esta característica.

En la Figura 7-12 se han fijado colores de límite para los cuatro planos. El plano más a la izquierda no utiliza su color de límite porque sus modos de límite son ambos WRAP. Para el plano siguiente utiliza el color negro de límite sólo en la dimensión vertical debido a los modos de límite. Podemos ver la mezcla entre el azul y el negro a la izquierda; en el lado derecho de la imagen el color negro del límite se mezcla con el borde blanco de la textura. Los colores de límite para los dos planos restantes son verde y rojo. Ambos modos de límite son CLAMP para el plano de la derecha de la imagen.



Observa que el color de límite no se utiliza si el filtro es BASE_LEVEL_POINT. Para que el color del límite sea utilizado, el filtro necesita ser por lo menos BASE_LEVEL_LINEAR. El corolario es que siempre que el filtro no sea BASE_LEVEL_POINT se utiliza el **BoundaryColor**.

También observa que la misma textura se carga cuatro veces en esta aplicación. Un objeto **Texture** no se puede compartir entre los cuatro planos en esta aplicación puesto que cada objeto de textura se configura con una combinación distinta de modos de límite.

Formato de Textura

La última configuración de la clase **Texture** es la del formato de textura. El formato de textura es una declaración de cuántos valores hay por texel y cómo esos valores afectan a los pixels. Por ejemplo, una configuración del formato de textura de INTENSITY indica que solo el valor del texel será utilizado para rojo, verde, azul, y los valores de alpha del pixel. Una configuración del formato de textura de

RGB indica que los tres valores del texel serán utilizados para los valores rojo, verde, y azul del pixel mientras que el valor de alpha del pixel sigue siendo igual.

Formato de Textura	Valores por Texel	Modifica Color del Pixel	Modifica alpha del Pixel
INTENSITY	1	si, R=G=B	si, R=G=B=A
LUMINANCE	1 (sólo color)	si, R=G=B	no
ALPHA	1 (sólo alpha)	no	si
LUMINANCE_ALPHA	2	si, R=G=B	si
RGB	3	si	no
RGBA	4	si	si

Texture3d

Como el nombre implica, un objeto **Texture3d** contiene una imagen tridimensional de la textura. Puede ser que pensemos en él como un volumen de color. La clase **Texture3d** es una extensión de **Texture**, así que todas las características de la clase **Texture** se aplican a **Texture3d**. La única característica que **Texture3d** tiene y que **Texture2D** no tiene, es una especificación para el modo de límite de la tercera dimensión, o la dimensión r.

Algunas Aplicaciones de Texturado

Créamoslo o no, hay muchas más características texturado a explicar. Sin embargo, podemos utilizar las características ya discutidas en muchas aplicaciones. Esta sección hace una parada en la discusión de los detalles de texturado para mostrar dos aplicaciones de texturado. Una aplicación aplica una textura a un primitivo geométrico (véase el Capítulo 2). Otra textura las líneas de polígonos no-rellenos. Una tercera aplicación utiliza la textura creada por un **Text2D** (véase el Capítulo 3) a otro objeto visual.

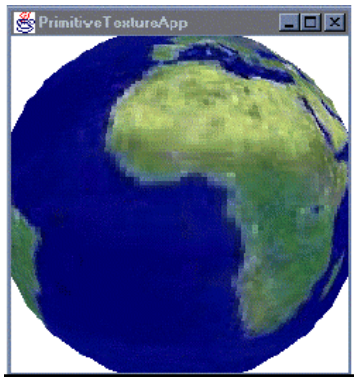
Texturado de Geométricos Primitivos

Una forma para simplificar el proceso de presentar una textura es utilizar un primitivo geométrico. Se puede utilizar una bandera para asignar automáticamente las coordenadas de textura al crear primitivos geométricos. El Fragmento de Código 7-6 muestra el uso de un constructor para una esfera primitiva con la generación de coordenadas.

Fragmento de Código 7-6, Crear un Esfera Primitiva con Pre-asignación de Coordenadas de Textura.

```
1.  objRoot.addChild(new Sphere(1.0f, Primitive.GENERATE_TEXTURE_COORDS, appear));
```

La línea del Fragmento de Código 7-6 se utiliza en el programa de ejemplo PrimitiveTextureApp.java. Este programa textura una esfera con la imagen **earth.jpg**, que también está en el fichero jar de ejemplo, dando por resultado la imagen de al Figura 7-13.



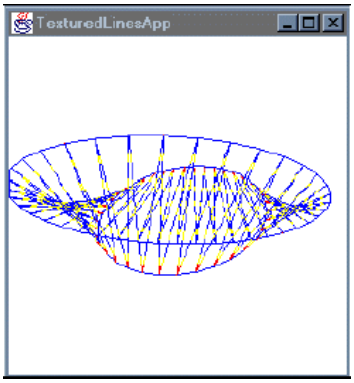
Texturado de Líneas

Los polígonos no son los únicos elementos gráficos que pueden ser texturados; las líneas también se pueden texturar. El programa TexturedLinesApp.java lo demuestra, usando una textura 1-D para texturar líneas de un objeto visual. En esta aplicación, el objeto visual es una tira torcida creada en el capítulo 2. El único 'truco' para texturar líneas es crear el manojo de apariencia apropiado para visualizar las líneas de la geometría y de los polígonos no rellenos. El Fragmento de Código 7-7 muestra las líneas del código para agregar el componente **PolygonAttributes** a un manojo de apariencia para visualizar las líneas.

Fragmento de Código 7-7, Crear un Manojo de Appearance para Mostrar Líneas en un Geometry Array.

```
1. PolygonAttributes polyAttrib = new PolygonAttributes();
2. polyAttrib.setCullFace(PolygonAttributes.CULL_NONE);
3. polyAttrib.setPolygonMode(PolygonAttributes.POLYGON_LINE);
4. twistAppear.setPolygonAttributes(polyAttrib);
```

Una textura unidimensional es realmente un objeto **Texture2D** con una dimensión (generalmente t) con tamaño 1. Para el programa de ejemplo, la textura tiene 16 texels por 1 texel. Las coordenadas de dos dimensiones de la textura se asignan al objeto visual. El valor-t de cada coordenada de textura se fija a 0.0f. Sin embargo, se podría utilizar cualquier valor-t y el resultado sería igual. La Figura 7-14 muestra la geometría de la tira torcida visualizada como líneas texturadas. El código fuente está en TexturedLinesApp.java

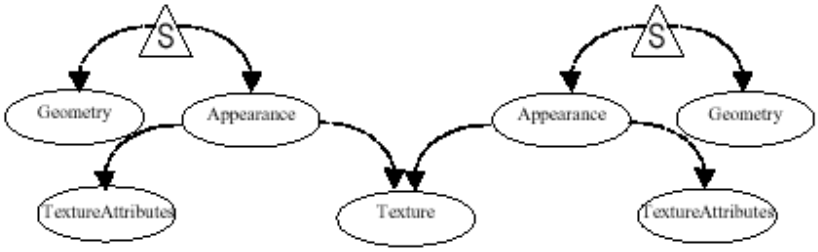


Usar Texturas Text2D

Los objetos **Text2D** crean texturas de texto especificado y aplican la textura a un polígono. A esta textura se puede acceder fácilmente desde **Text2D** y aplicarla a otro objeto visual. La Figura 7-15 muestra una imagen producida por el programa Text2DTextureApp.java, un programa que aplica la textura creada por otro objeto **Text2D** mostrándola en el fondo de la geometría de otro objeto visual.

Atributos de Textura

En secciones anteriores vimos algunas de las opciones disponibles en texturado. El componente **TextureAttributes** permite la personalización posterior del texturado. Las configuraciones de los atributos de textura incluyen el modo de textura, color de mezcla, modo de corrección de perspectiva, y una correspondencia del mapeo de la textura. Los valores por defecto para estas configuraciones son **REPLACE**, **black**, **FASTEST**, y **NONE**, respectivamente. Además, el método setEnable permite activar y desactivar el mapeo de la textura. Todas las configuraciones se explican en esta sección.
Una ventaja de tener controladas las características del texturado por un componente diferente del nodo es la capacidad de compartir una textura entre objetos visuales pero aún así poder personalizarla para cada objeto visual. La Figura 7-10 muestra dos objetos visuales que comparten un solo objeto **Texture**. Cada uno de los objetos visuales personaliza la textura con el componente **TextureAttributes** en su manojo de apariencia.



Los objetos **TextureAttributes** se añaden al escenario gráfico como miembros de una manejo de apariencia. El método setTextureAttributes de **Appearance** se muestra en el siguiente bloque de referencia.

Método setTextureAttributes de la Clase **Appearance**

void setTextureAttributes(TextureAttributes textureAttributes)
Selecciona el objeto textureAttributes en un objeto appearance.

Modo de Textura

Para apreciar el rol del modo de textura debemos entender la secuencia de las operaciones implicadas en la determinación del color de un pixel. Indicado brevemente, primero se calcula el color de la no-textura de un pixel, y luego se aplica la textura. El color de no-textura lo determina el color de la geometría por vértice, **ColoringAttributes**, o la combinación de las características materiales y las condiciones de iluminación. Como hay varias formas de determinar el color de la no-textura, hay varias maneras posibles de combinar el color de la no-textura y el color de la textura.

La configuración del modo de textura es un factor importante en la determinación de cómo afecta el valor del texel (color y/o alpha) a los valores del color y del alpha del pixel de la no-textura. La operación de texturado real depende de la combinación del formato de la textura y del modo de textura. Puedes referirte a la especificación de Java 3D API (apéndice E) para más información.

El modo de textura por defecto es **REPLACE**, las otras opciones son **BLEND**, **DECAL**, y **MODULATE**. Cada uno de los modos se describe en las siguientes secciones. También podemos ver la tabla que resume los modos de la textura.

Blend

En el modo **BLEND**, el color de la textura se mezcla con el color de la no-textura. El color de la textura determina la cantidad del color de la no-textura a utilizar. La transparencia que resulta es la combinación de la transparencia de la textura y del material. Este modo determinado de la textura tiene la flexibilidad agregada de incluir opcionalmente un color de mezcla.

Decal

En el modo **DECAL**, el color de la textura se aplica como etiqueta encima del color de la no-textura. La transparencia de la textura determina la cantidad de color material a utilizar. La transparencia del pixel se deja sin cambios. Esto es totalmente igual que aplicar una etiqueta a un objeto del mundo real. El formato de textura debe ser RGB o RGBA para el modo de textura de **DECAL**.

Modulate

En el modo **MODULATE** el color de la textura se combina con el color de la no-textura. La transparencia que resulta es la combinación de la transparencia de la textura y del material. Puesto que el color que resulta depende de la no-textura y de los colores de la textura, este modo es útil en la aplicación de la misma textura a una gran variedad de objetos visuales sin hacer que todos parezcan iguales. Este modo se utiliza a menudo en escenas de iluminación.

Replace

En el modo **REPLACE** la textura proporciona el color y la transparencia para el pixel, no haciendo caso del resto de los colores excepto del color specular (si se permite la iluminación). Éste es el modo de textura por defecto incluso cuando no hay componente **TextureAttributes** en el manojo de apariencia.

Sumario de Modos de Textura

La Siguiente tabla ofrece un sumario de cada uno de los modos de textura. Esta tabla se ha pensado como una guía general para entender los distintos modos de textura disponibles. Los cálculos del color real se basan en una combinación del modo de textura y del formato de textura.

Cuando se permite la iluminación, los componentes de colores ambiente, difuso, y emisivo se ven afectados por la operación de textura; el color specular se ve afectado. El color specular se calcula basándose en las condiciones del material y de la iluminación, luego después de que la operación de la textura se aplique a otros componentes del color (no-texturados) el color specular se agrega a los otros colores que renderizan el color final del pixel.

Modo de Textura	Color de Pixel Derivado desde	Determinado por	Aplicación a...
BLEND	Color de Textura, no-textura, y color mezcla opcional	Color de textura	Escenas Iluminadas con Color de Mezcla
DECAL	Color de Textura y color no-textura	alpha de la textura	detalles de superficie
MODULATE	Colores de Textura y de no-textura	n/a	Escenas de iluminación

REPLACE	Sólo color de textura (modo de textura por defecto)	n/a	Escenas sin iluminación

Textura con Color de Mezcla

El color de mezcla se utiliza en texturado sólo cuando el modo de textura es **BLEND**. El color del pixel resultante es una combinación del color del texel y del color de mezcla. Con el color de mezcla se puede aplicar la misma textura con diferentes sombras a diferentes objetos visuales. El color de mezcla se expresa como un valor **RGBA**. El color de mezcla por defecto es (0,0,0,0) negro con un alpha de 0.

Modo de Corrección de Perspectiva

El mapeo de textura ocurre en el espacio de la imagen. Por esta razón los planos texturados pueden parecer incorrectos cuando se ven desde un lateral. Es decir, parecen incorrectos a menos que se haga una corrección de la perspectiva. En Java 3D la corrección de la perspectiva se hace siempre. La única opción es cómo hacer esta corrección de la perspectiva.
Las dos opciones son **FASTEST** y **NICEST**. Obviamente, el dilema es la velocidad clásica contra la calidad de la imagen. Para esta opción, la configuración del valor por defecto es **NICEST**.

Transformación del Mapeo de Textura

Dentro del componente **Attributes** de una textura se puede especificar un objeto **Transform3d** para alterar la función de mapeo de la textura. Esta correspondencia de transformación del mapeo de textura se puede utilizar para mover una textura sobre un objeto visual en tiempo de ejecución. La transformación traslada, rota, y escala las coordenadas de textura (s, t, r) antes de que los texels sean seleccionados desde la imagen de textura.
Una traslacción en la transformación de la textura desplazaría la textura a través del objeto visual. Se puede utilizar una transformación de rotación para reorientar

la textura en un objeto visual. Se pueden utilizar una transformación de escala para repetir la textura a través de un objeto visual. Por supuesto, como un objeto **transform** puede contener una combinación de éstos, se puede animar la textura de un objeto visual manipulando este objeto.

API TextureAttributes

Los siguientes bloques de referencia listan los constructores, métodos y capacidades del componente **TextureAttributes**.

Sumario de Constructores de la Clase **TextureAttributes**

Extiende: NodeComponent

El objeto **TextureAttributes** define los atributos que se aplican a un mapeo de textura.

TextureAttributes()
Construye un objeto TextureAttributes con estos valores por defecto:
texture mode : REPLACE, transform : null, blend color : black (0,0,0,0), perspective correction: NICEST
TextureAttributes(int textureMode, Transform3d transform,
Color4f textureBlendColor, int perspCorrectionMode)
Construye un objeto TextureAttributes con los valores especificados.

Constantes de la Clase **TextureAttributes**

Estas constantes se usan en los constructores y métodos para seleccionar los modos de textura y de corrección de la perspectiva.

Constantes de Modo de Textura

- **BLEND** Mezcla el color de mezcla con el color del objeto.
- **DECAL** Aplica el color de la textura al objeto como una etiqueta.
- **MODULATE** Modula el color del objeto con el color de la textura.
- **REPLACE** Reemplaza el color del objeto con el color de la textura.

Constantes del Modo de Corrección de Perspectiva

- **FASTEST** Usa el método más rápido disponible para la correción de perspectiva del mapeo de textura.
- **NICEST** Usa el mejor método (de mayor calidad) disponible para la correción de perspectiva del mapeo de textura.

Sumario de Métodos de la Clase **TextureAttributes**

void getTextureBlendColor(Color4f textureBlendColor)

Obtiene el color de mezcla de la textura para este objeto appearance.

void getTextureTransform(Transform3d transform)

Recupera una copia del objeto transformation de la textura.

void setPerspectiveCorrectionMode(int mode)

Selecciona el modo de corrección de la perspectiva a usar para la interpolación de cooredndas de color y/o textura a uno de :

- **FASTEST** Usa el método más rápido disponible para la correción de perspectiva del mapeo de textura.
- **NICEST** Usa el mejor método (de mayor calidad) disponible para la correción de perspectiva del mapeo de textura.

void setTextureBlendColor(Color4f textureBlendColor)

void setTextureBlendColor(float r, float g, float b, float a)

Selecciona el color de mezcla para este objeto TextureAttributes.

void setTextureMode(int textureMode)

Selecciona el parámetro del modo de textura a uno de:

- **BLEND** Mezcla el color de mezcla con el color del objeto.
- **DECAL** Aplica el color de la textura al objeto como una etiqueta.
- **MODULATE** Modula el color del objeto con el color de la textura.
- **REPLACE** Reemplaza el color del objeto con el color de la textura.

void setTextureTransform(Transform3d transform)

Selecciona el objeto transform usado para transformar las coordenadas de textura.

Sumario de capacidades de la Clase **TextureAttributes**

- **ALLOW_BLEND_COLOR_READ | WRITE**

Permite leer (escribir) el color de mezcla de la textura

- **ALLOW_MODE_READ | WRITE**
  Permite leer (escribir) los modos de textura y de correción de perspectiva.
- **ALLOW_TRANSFORM_READ | WRITE**
  Permite leer (escribir) el objeto transform de la textura.

Generación Automática de Coordenadas de Textura

Según se explicó anteriormente, asignar coordenadas de textura a cada vértice de la geometría es un paso necesario en el texturado de objetos visuales. Este proceso puede comsumir mucho tiempo así como es difícil para objetos visuales grandes y/o complejos. Debemos tener presente que esto es un problema para el programador y una vez solucionado, no es un problema que se repita.
A menudo las coordenadas de textura se asignan con el código específico de un objeto visual. Sin embargo, otra solución es automatizar la asignación de las coordenadas de textura mediante algún método. Este método se podía utilizar para cualquier objeto visual tanto si es grande o pequeño, complejo o simple. Este acercamiento es exactamente lo que lo hace un objeto **TexCoordGeneration** (generación de coordenadas de textura). Siempre que un objeto se cargue desde un fichero o sea creado en el código del programa, se puede utilizar un objeto **TexCoordGeneration** para asignar coordenadas de textura.
TexCoordGeneration es una clase del corazón del API Java 3D usada para generar coordenadas de textura. Para generar automáticamente coordenadas de textura, el programador especifica los parámetros de la coordenada de textura en un objeto **TexCoordGeneration** y agrega este objeto al manojo de apariencia del objeto visual. Las coordenadas de textura se calculan basándose en los parámetros de especificación de coordenadas en el tiempo de ejecución. Los parámetros se explican en las secciones siguientes.

- **Formato de Generación de Textura**

Esta selección simplemente especifica si las coordenadas de textura serán generadas para una textura de dos o tres dimensiones. Las selecciones posibles son **TEXTURE_COORDINATE_2** y **TEXTURE_COORDINATE_3** que generan coordenadas de textura 2D (S y T) y coordenadas de textura 3D (S, T, y R), respectivamente.

- **Modo de Generación de Textura**

Hay dos aproximaciones básicas para la generación de textura: proyección linear y mapeo esférico:

Poyección Linear

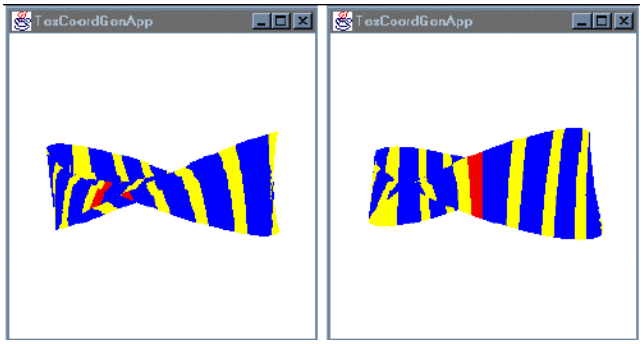
Con la proyección linear, las coordenadas de textura se especifican con planos. Para las coordenadas de textura de dos dimensiones (s,t), se utilizan dos planos. La distancia desde un vértice a un plano es la coordenada de textura en una dimensión; la distancia desde el otro plano a un vértice es la coordenada de textura en la otra dimensión. Para las texturas tridimensionales, se utilizan tres planos.

Los tres parámetros planos posibles se nombran planeS, planeT, y planeR, donde el nombre corresponde a la dimensión para la cual se utiliza. Cada plano se especifica como 4-tuple (ecuación plana). Los primeros tres valores son el vector normal superficial para el plano. El cuarto valor especifica la distancia desde el origen al plano a lo largo de un vector paralelo al vector normal superficial del plano.

Hay dos variaciones en este método automático de generación de coordenadas de textura. El primero, llamado objeto linear, produce coordenadas de textura estáticas. Con coordenadas de textura generadas linearmente, si el objeto visual se mueve, los coordenadas de textura no cambian. La segunda opción, llamada ojo linear, produce coordenadas de textura relativas a las coordenadas del ojo

dando como resultado coordenadas de textura variables. Con coordenadas de textura linear del ojo los objetos que se mueven parecen moverse con la textura. La Figura 7-17 muestra las imágenes producidas por un programa de ejemplo que utiliza un objeto **TexCoordGeneration** para asignar coordenadas de textura a una tira torcida. En esta aplicación se usa una textura unidimensional. La textura tiene un solo texel rojo en un extremo. Cuando la aplicación se ejecuta, la tira torcida rota.

La imagen de la izquierda de la Figura 7-17 muestra el texturado con modo de generación **OBJECT_LINEAR**. En este caso la textura rota con el objeto y se puede ver el texel rojo rotar con la tira. La imagen de la derecha de la Figura 7-17 muestra la textura que resulta cuando el modo la generación es **EYE_LINEAR** para la tira torcida. En este caso, el texel rojo permanece en el centro de la vista mientras que el objeto rota.



TexCoordGenApp.java es el programa que produce estás imágenes.

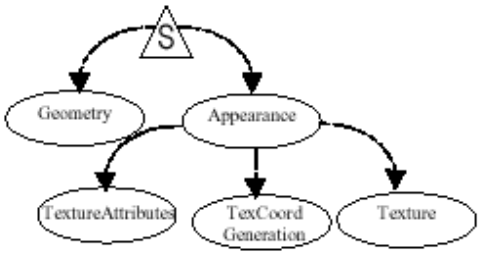
Mapeo Esférico

Si un objeto brillante está en el centro de una habitación real, probablemente reflejaría la imagen de muchos de los otros objetos de la habitación. Las reflexiones dependerían de la forma del objeto y de la orientación de las cosas en la habitación. El modo de generación de las coordenadas del mapeo esférico está diseñado para asignar coordenadas de textura para aproximar las reflexiones de otros objetos sobre el objeto visual como sucedería para el objeto brillante en el mundo real del ejemplo.

Cuando se usa un objeto **TexCoordGeneration** en el modo de mapeo esférico el cálculo de las coordenadas de textura se basa en las superficies normales y en la dirección de la vista.
La textura usada para este efecto debe estar especialmente preparada. Si el ambiente virtual del objeto brillante existe en el mundo real, una fotografía de la escena tomada con una lente de ojo de pez creará una imagen de textura conveniente. Si no existe la escena, entonces la textura se debe crear para parecer que la imagen es una fotografía tomada con una lente de ojo de pez.

- **Cómo usar un Objeto TexCoordGeneration**

Para usar un objeto **TexCoordGeneration**, lo seleccionamos como un componente del manojo de apariencia del objeto visual a texturar. La Figura 7-18 muestra el diagrama de una manojo de apariencia con un objeto **TexCoordGeneration** junto con un objeto **Texture** y otro objeto **TextureAttributes**.



Método setTexCoordGeneration de la Clase **Appearance**

void setTexCoordGeneration(TexCoordGeneration texCoordGeneration)
Selecciona el objeto texCoordGeneration al objeto especificado.

- **API TexCoordGeneration**

Los siguientes bloques de referencia listan los constructores, constantes, métodos y capacidades de los objetos de la clase **TexCoordGeneration**.
Sumario de Constructores de la Clase **TexCoordGeneration**

El objeto **TexCoordGeneration** contiene todos los parámetros necesarios para

generar coordenadas de textura. Está incluido como parte de un objeto **Appearance**.

TexCoordGeneration()

Construye un objeto TexCoordGeneration usando los valores por defecto para todas las edades.

TexCoordGeneration(int genMode, int format)

Construye un objeto TexCoordGeneration con genMode y format especificados.

TexCoordGeneration(int genMode, int format, Vector4f planeS)

TexCoordGeneration(int genMode, int format, Vector4f planeS, Vector4f planeT)

TexCoordGeneration(int genMode, int format, Vector4f planeS, Vector4f planeT, Vector4f planeR)

Construyen un objeto TexCoordGeneration con genMode, format, y las ecuaciones de los planos especificados.

Sumario de Campos de la Clase **TexCoordGeneration**

Constantes de Modo de Generación

- **EYE_LINEAR** Genera coordenadas de textura como una función linear en coordenadas de ojo (por defecto).
- **OBJECT_LINEAR** Genera coordenadas de textura como una función linear en coordenadas del objeto .
- **SPHERE_MAP** Genera coordenadas de textura usando un mapeo de reflexión esférica en coordenadas de ojo.

Constantes de Formato

- **TEXTURE_COORDINATE_2** Genera coordenadas de textura 2D (S y T) (por defecto)
- **TEXTURE_COORDINATE_3** Genera coordenadas de textura 3D (S, T, y R)

Sumario de Métodos de la Clase **TexCoordGeneration**

void setEnable(boolean state)

Activa o desactiva la generación de coordenadas para este objeto appearance.

void setFormat(int format)

Selecciona el formato TexCoordGeneration al valor especificado.

void setGenMode(int genMode)

Selecciona el modo de generación de TexCoordGeneration al valor especificado.

void setPlaneR(Vector4f planeR)

Selecciona la ecuación plana de la coordenada R.

void setPlaneS(Vector4f planeS)

Selecciona la ecuación plana de la coordenada S.

void setPlaneT(Vector4f planeT)

Selecciona la ecuación plana de la coordenada T.

Sumario de Capacidades de la Clase **TexCoordGeneration**

- **ALLOW_ENABLE_READ | WRITE**
  Permite leer/escribir su bandera de enable.
- **ALLOW_FORMAT_READ**
  Permite leer escribir su información de formato.
- **ALLOW_MODE_READ**
  Permite leer su información de modo.
- **ALLOW_PLANE_READ**
  Permite leer la información de componentes planeS, planeR, y planeT.

Múltiples Niveles de Textura (Mipmaps)

Para entender la razón de los múltiples niveles de textura, consideremos una aplicación que contenga un objeto visual texturado que se mueva alrededor la escena (o que lo mueva el espectador). Cuando este objeto visual está cerca del espectador aparecen demasiados pixels en la imagen. Para este caso, se debería utilizar una textura de buen tamaño para evitar la visión de texels individuales; esto es especialmente cierto cuando se utiliza el punto de muestreo como filtro de ampliación.

Sin embargo, cuando este objeto visual se ve en la distancia, la textura será demasiado grande para el objeto visual y la textura se reducirá durante la representación. (Recordamos que el mapeo de textura tiene lugar durante la renderización en la imagen, la pantalla, o el espacio). El punto de muestreo para el filtro de reducción problablemente no dará resultados satisfactorios cuando el

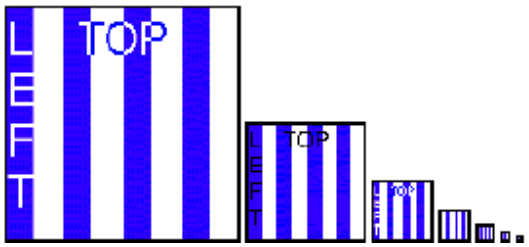
objeto visual sea 1/32 o más pequeño (tamaño del pixel) que la resolución de la textura. El dilema es calidad de la imagen contra rendimiento de renderización.
Si en vez de usar un gran mapeo de textura (porque el objeto visual aparecerá grande) se usa uno pequeño para hacer que la vista del objeto sea mejor cuando es pequeño, existe el problema inverso. Para las imágenes de buena calidad el filtro de ampliación implicará la interpolación linear dando por resultado más cálculo. De nuevo, el dilema está entre la calidad de imagen contra el rendimiento de la renderización. La única ventaja de usar un mapeo de textura más pequeño es un menor requerimiento de memoria para almacenar la textura.
Lo que se necesita es un mapeo de textura pequeño cuando el objeto visual aparece pequeño y un mapeo de textura grande cuando el objeto visual aparece grande. Las técnicas de texturado actuales que usan una imagen de textura, llamada texturado base, no pueden hacer esto. Y esto es exactamente lo que proporcionan los múltiples niveles de textura.

¿Qué es el Texturado Multi-Nivel (MIPmap)?

Los múltiples niveles de textura se refieren a una técnica de texturado donde se utilizan juntas una serie de imágenes de textura como textura para los objetos visuales. La serie de imágenes es (generalmente) la misma textura en una variedad de resoluciones. Cuando un objeto visual se está renderizando con múltiples niveles de textura, se utiliza la imagen de textura que está más cercana al tamaño de pantalla del objeto visual.
El funcionamiento del renderizador depende de los filtros de reducción y ampliación usados. Sin embargo, con **MIPmaps** tenemos más control sobre la apariencia de los objetos visuales y podemos conseguir objetos visuales de mejor aspecto con un mejor rendimiento.
Usar múltiples niveles de textura es como usar un objeto **DistanceLOD** (véase el Capítulo 5) para aplicar diversas texturas a un objeto visual cuando se ve desde diversas distancias. Las excepciones son que con el **Mipmap** el objeto visual siempre es texturado mientras que con el objeto **DistanceLOD**, el objeto podría no

ser texturado a algunas distancias. Y, para los objetos visuales texturados en todas las distancias, el **MIPmap** es más eficiente y ha agregado posibilidades de filtrado con respecto a un objeto **DistanceLOD** usado para una aplicación similar. Los múltiples niveles la textura son referidos comúnmente como mipmap. El término "**MIPmap**" viene de las siglas del Latin **"multum in parvo"**, que significa muchas cosas en un lugar pequeño. El término **MIPMap** realmente se refiere a una técnica específica de almacenaje para almacenar una serie de imágenes para el uso en texturado de múltiples niveles El término **MIPmap** se utiliza comúnmente para significar texturado de múltiples niveles.

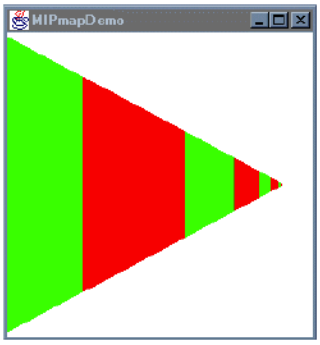
Con la técnica de almacenaje **MIPmap**, el tamaño de una imagen de textura es ¼ del tamaño anterior (½ del tamaño en cada cada dimensión). Esto continúa hasta que el tamaño de la imagen más pequeña es de 1 texel por 1 texel. Por ejemplo, si tamaño máximo de la textura es 16x4, las texturas restantes son 8x2, 4x1, 2x1, y 1x1. La Figura 7-19 muestra los niveles de textura para la textura **stripe.gif**, Cada una de estas imágenes de textura fue preparada usando software de edición de imágenes.



La Figura 7-20 muestra una imagen de un solo plano texturado con una textura múltiple donde cada nivel de textura tiene un color diferente. El plano se orienta en ángulo al espectador para que el lado izquierdo esté más cercano al espectador que el derecho. Debido a la proyección de la perspectiva el lado izquierdo del plano parece más grande en coordenadas de la imagen que el derecho.

Debido a la orientación y a la proyección del plano, los pixels representan menos área superficial (en el sistema virtual de coordenadas del objeto) a la izquierda y progresivamente más área superficial hacia a la derecha, dando como resultado que el nivel de textura cambia. A la izquierda del plano en la imagen, se utiliza el

nivel base de la textura. Los cambios del color en la imagen indican donde ocurrieron los cambios del nivel de textura durante la renderización.
Tener una textura para cada nivel de un color diferene no es la aplicación típica de texturado múltiple. Esta aplicación simplemente iluistra la operación de una textura múltiple.



La Figura 7-20 está generada por el programas MIPmapDemo.java.

- **Ejemplos de Texturas Multi-Nivel**

En lo que concierte a programación con el API Java 3D, crear una textura de niveles múltiples es casi igual que crearla de un solo nivel, o nivel base. Mirando hacia la receta texturado simple la única diferencia es que se necesitan varias imágenes de textura para la textura de niveles múltiples. Hay dos maneras de crear los niveles múltiples de las imágenes de la textura. Una forma es crear cada imagen a mano con las aplicaciones apropiadas de edición/creacción de imágenes, la otra es utilizar una característica del cargador de textura para crear esas imágenes desde la imagen base.
Las dos técnicas de texturado de nivel múltiple ocupan una cantidad casi igual de código. La de menos cantidad de trabajo es la de generar las imágenes de los niveles desde la imagen base. El Fragmento de Código 7-8 presenta el código de carga de las texturas de MIPmapApp.java. Esta aplicación es un ejemplo de como generar niveles múltiples de textura desde una imagen base.

Fragmento de Código 7-8, Crear Texturado Multi-nivel desde una sola imagen base.

```
1.  Appearance appear = new Appearance();
2.
3.  NewTextureLoader loader = new NewTextureLoader("stripe.gif",
4.  TextureLoader.GENERATE_MIPMAP);
5.  ImageComponent2D image = loader.getImage();
6.
7.  imageWidth = image.getWidth();
8.  imageHeight = image.getHeight();
9.
10. Texture2D texture = new Texture2D(Texture.MULTI_LEVEL_MIPMAP,
11. Texture.RGB, imageWidth, imageHeight);
12. imageLevel = 0;
13. texture.setImage(imageLevel, image);
14.
15. while (imageWidth > 1 || imageHeight > 1){ // loop until size: 1x1
16.     imageLevel++; // compute this level
17.
18.     if (imageWidth > 1) imageWidth /= 2; // adjust width as necessary
19.     if (imageHeight > 1) imageHeight /= 2; // adjust height as necessary
20.
21.     image = loader.getScaledImage(imageWidth, imageHeight);
22.     texture.setImage(imageLevel, image);
23.     }
24.
25. texture.setMagFilter(Texture.BASE_LEVEL_POINT);
26. texture.setMinFilter(Texture.MULTI_LEVEL_POINT);
27.
28. appear.setTexture(texture);
```

El Fragmento de Código 7-8 empieza siguiendo los mismos pasos que son utilizados para cualquier aplicación de textura cargando la imagen base. Una diferencia es que el **TextureLoader** se crea con la bandera **GENERATE_MIPMAP** (líneas 3-4). Entonces se extrae la imagen base del cargador de la forma usual. Las dimensiones de esta imagen son necesarias no sólo para crear el objeto **Texture2D**, sino también para calcular los tamaños de las siguientes imágenes.

Por esta razón se guardan en dos variables (líneas 7 y 8). Estas variables serán utilizadas durante la generación y carga de las imágenes restantes.
El objeto **Texture2D** se crea usando el modo MIPmap **MULTI_LEVEL_MIPMAP** y la dimensión de la imagen base (líneas 10 y 11). El nivel base es el nivel 0. Entonces el número de nivel se graba y se selecciona la imagen base como la imagen para el nivel 0 (líneas 12 y 13).
El bucle itera hasta que el tamaño de la imagen sea de 1 pixel por 1 pixel (línea 15). El número de nivel se incrementa en cada iteración (línea 16) y se calcula la dimensión de la imagen (líneas 18 y 19). La imagen apropiadamente escalada se consigue desde **TextureLoader** (línea 21) y se selecciona para el nivel actual en el objeto **Texture2D** (línea 22).
Cuando se crea un mapeo de textura de múltiples niveles debemos asegurarnos de seleccionar filtros de nivel como se hace en las líneas 25 y 26 del Fragmento de Código 7-8. Las selecciones de filtrado desactivan el nivel de texturado múltiple.
Crea las imágenes a mano permite una calidad superior y/o poder añadir efectos especiales. Las imágenes generadas se producen filtrando la imagen base.

Fragmento de Código 7-9, Múltiples Niveles de Textura Cargados desde Ficheros de Imagenes Individuales.

```
1.  Appearance appear = new Appearance();
2.
3.  String filename = "stripe.gif"; // filename for level 0
4.  NewTextureLoader loader = new NewTextureLoader(filename);
5.  ImageComponent2D image = loader.getImage();
6.
7.  imageWidth = image.getWidth();
8.  imageHeight = image.getHeight();
9.
10. Texture2D texture = new Texture2D(Texture.MULTI_LEVEL_MIPMAP,
11. Texture.RGBA,imageWidth, imageHeight);
12. imageLevel = 0;
13. texture.setImage(imageLevel, image);
14.
```

```
15.  while (imageWidth > 1 || imageHeight > 1){ // loop until size: 1x1
16.      imageLevel++; // compute this level
17.
18.      if (imageWidth > 1) imageWidth /= 2; // adjust width as necess.
19.      if (imageHeight > 1) imageHeight /= 2;// adjust height as necess.
20.      filename = "stripe"+imageWidth+".gif";// file to load
21.
22.      loader = new NewTextureLoader(filename);
23.      image = loader.getImage();
24.
25.      texture.setImage(imageLevel, image);
26.      }
27.
28.  texture.setMagFilter(Texture.BASE_LEVEL_POINT);
29.  texture.setMinFilter(Texture.MULTI_LEVEL_POINT);
30.
31.  appear.setTexture(texture);
```

Filtros de Reducción para Múltiples Niveles de Textura

Además de los dos filtros de nivel base, hay dos opciones de múltiples filtros para la configuración del filtro de reducción. Estas configuraciones adicionales son **MIPMAP_POINT**, y **MIPMAP_LINEAR**. Como con las otras configuraciones de filtro, el filtro de punto es probable que cree imágenes más rápidas pero de una calidad más baja con respecto al filtro linear.
Recuerda, cuando usamos múltiples niveles de textura, debemos seleccionar uno de los filtros de múltiples niveles para el filtro de reducción para utilizar otros niveles distintos del nivel base. Estas configuraciones adicionales de filtro no se aplican a las configuraciones de filtro de ampliación puesto que la ampliación de la textura se haría solamente en el nivel base.

Modo Mipmap

El modo **MIPmap** de la clase **Texture** es realmente una elección entre varios niveles de textura y un sólo nivel de textura. Las dos selecciones son **BASE_LEVEL** y **MULTI_LEVEL_MIPMAP**.

API de Texture, Texture2D, y Texture3d

Muchas de las secciones precedentes presentan algunas porciones de las clases **Texture**, **Texture2D**, o **Texture3d**. Puesto que estas clases se han descrito en muchas secciones, el API de estas clases se presenta en esta sección. **Texture** es la clase base para **Texture2D** y **Texture3d**. La clase **Texture** proporciona la mayoría del interfaz para las clases **Texture2D** y **Texture3d** incluyendo texturado multi-nivel. La siguiente tabla presenta un resumen de las características de estas tres clases. Para cada opción de texturado la tabla lista la clase que proporciona al interface, el método (set) para cambiar la configuración y el valor por defecto

Característica/Selección	Clase	Métodos set	Valor Defecto
Texture Image	Texture	setImage()	null
Image Format	Texture	(ver constructores)	none
Mipmap Mode	Texture	setMipMapMode()	BASE_LEVEL
Minification Filter	Texture	setMinFilter()	BASE_LEVEL_POINT
Magnification Filter	Texture	setMagFilter()	BASE_LEVEL_POINT
Boundary Modes	Texture Texture2D Texture3d	setBoundaryModeS() setBoundaryModeT() setBoundaryModeR()	WRAP WRAP WRAP
BoundaryColor	Texture	setBoundaryColor()	black

Filtros de Reducción y Ampliación

Según lo discutido anteriormente hay configuraciones separadas de filtros para reducción y ampliación. Las opciones de ampliación son: **BASE_LEVEL_POINT**, **BASE_LEVEL_LINEAR**, **FASTEST**, o **NICEST**. El filtro será **BASE_LEVEL_POINT** cuando se especifique **FASTEST** y **BASE_LEVEL_LINEAR** cuando se especifique **NICEST**.
Las opciones de reducción son: **BASE_LEVEL_POINT**, **BASE_LEVEL_LINEAR**, **MULTI_LEVEL_POINT**, **MULTI_LEVEL_LINEAR**, **FASTEST**, o **NICEST**. Las opciones de filtro de nivel base se pueden utilizar para las texturas de un sólo nivel o texturas de varios niveles. Los filtros reales usados cuando se especifica **FASTEST** o **NICEST** se implementan dependiendo de si se elige un filtro multi-nivel o una textura de múltiples niveles.

API Texture

Ahora que se han presentado todas las características de la textura, presentamos el API de la clase **Texture**. Como esta es una clase abtracta no hay bloque de referencia sobre sus constructores.

Sumario de Campos de la Clase **Texture**

El objeto **Texture** es un componente de un objeto **Appearance** que define las propiedades de la textura usada cuando se activa el mapeo de texturas. El objeto **Texture** es una clase abstracta y todos sus objetos se deben crear usando objetos **Texture2D** o **Texture3d**.

Constantes de Formato

- **ALPHA** Especifica la textura que sólo contiene valores Alpha.
- **INTENSITY** Especifica la textura que sólo contiene valores Intensity.
- **LUMINANCE** Especifica la textura que sólo contiene valores Luminance.
- **LUMINANCE_ALPHA** Especifica la textura que contiene valores Luminance y Alpha.
- **RGB** Especifica la textura que contiene valores de color Red, Green y Blue.
- **RGBA** Especifica la textura que contiene valores de color Red, Green, Blue y valor

Alpha.

Constantes de modo MIPMap

- **BASE_LEVEL** Indica que el objeto Texture sólo tiene un nivel.
- **MULTI_LEVEL_MIPMAP** El objeto Texture tiene varios niveles - uno por cada nivel mipmap.

Constantes de Filtro

- **BASE_LEVEL_LINEAR** Realiza interpolación bilinear sobre los cuatro texels más cercanos en el nivel 0 del mapa de textura.
- **BASE_LEVEL_POINT** Selecciona el texel más cernano en el nivel 0 del mapa de textura.
- **MULTI_LEVEL_LINEAR** Realiza interpolación tri-linear sobre los cuatro texels más cercanos de los dos niveles de mipmap más cercanos.
- **MULTI_LEVEL_POINT** Selecciona el texel más cercano en el mipmap más cercano.

Constantes de Modo de Límites

- **CLAMP** Encierra las coordenadas de textura para que estén en el rango [0, 1].
- **WRAP** Repite la envoltura envolviendo las coordenadas de textura que están fuera del rango [0,1].

Constantes del Modo de Corrección de la Perspectiva

- **FASTEST** Usa el método más rápido disponible para procesar la geometría.
- **NICEST** Usa el método de mejor apariencia disponible para procesar la geometría.

Sumario de Métodos de la Clase **Texture**

El objeto **Texture** es un componente de un objeto **Appearance** que define las propiedades de la textura usada cuando se activa el mapeo de texturas. El objeto **Texture** es una clase abstracta y todos sus objetos se deben crear usando objetos **Texture2D** o **Texture3d**.

ImageComponent getImage(int level)

Obtiene el nivel de mipmap especificado.

void setBoundaryColor(Color4f boundaryColor)

void setBoundaryColor(float r, float g, float b, float a)

Selecciona el color límite de la textura para este objeto.

void setBoundaryModeS(int boundaryModeS)

Selecciona el modo de límites para la coordenada S en este objeto texture.

void setBoundaryModeT(int boundaryModeT)

Selecciona el modo de límites para la coordenada T en este objeto texture.

void setEnable(boolean state)

Activa o desactiva el mapeo de textura para este objeto appearance.

void setImage(int level, ImageComponent image)

Selecciona un nivel de mipmap especificado.

void setMagFilter(int magFilter)

Selecciona la función de filtro de ampliación.

void setMinFilter(int minFilter)

Selecciona la función de filtro de reducción.

void setMipMapMode(int mipmapMode)

Selecciona el modo mipmap para el mapeo de textura de este objeto texture.

Sumario de Capacidades de la Clase **Texture**

- **ALLOW_BOUNDARY_COLOR_READ**
  Permite leer su información de color de límite.
- **ALLOW_BOUNDARY_MODE_READ**
  Permite leer su información de modo de límite.
- **ALLOW_ENABLE_READ | WRITE**
  Permite leer/escribir su bandera de enable.
- **ALLOW_FILTER_READ**
  Permite leer su información de filtro.
- **ALLOW_IMAGE_READ**
  Permite leer su información de componente imagen.
- **ALLOW_MIPMAP_MODE_READ**
  Permite leer su información de modo de mipmap.

API de Texture2D

Texture2D es una extensión concreta de la clase abstracta **Texture**. **Texture2D** sólo porporciona un constructor de interés. Todos los métodos usados con objetos **Texture2D** son métodos de **Texture**.

Sumario de Constructores de la Clase **Texture2D**

Texture2D es una subclase de la clase **Texture**. Extiende la clase **Texture** añadiendo un constructor.

Texture2D(int mipmapMode, int format, int width, int height)
Construye un objeto **Texture2D** vacío con los valores especificados de mipmapMode, format, width, y height. La imagen del nivel 0 la debe seleccionar la aplicación usando el método setImage. Si mipmapMode se selecciona a **MULTI_LEVEL_MIPMAP**, se deben seleccionar las imágenes para TODOS los niveles.
Parámetros:

- **mipmapMode** - tipo mipmap para este Texture: Uno de **BASE_LEVEL**, **MULTI_LEVEL_MIPMAP**.
- **format** - formato de datos de las texturas grabadas en el objeto. Uno de **INTENSITY**, **LUMINANCE**, **ALPHA**, **LUMINANCE_ALPHA**, **RGB**, **RGBA**.
- **width** - anchura de la imagen del nivel 0. Debe ser una potencia de 2.
- **height** - altura de la imagen del nivel 0. Debe ser una potencia de 2.

API de Texture3d

Texture3D es una extensión concreta de la clase abstracta **Texture**. **Texture3D** sólo porporciona un constructor de interés. Todos los métodos usados con objetos **Texture3D** son métodos de **Texture**.

Sumario de Constructores de la Clase **Texture3d**

Texture3d es una subclase de la clase **Texture**. Extiende la clase **Texture** añadiendo una tercera coordenada, un constructor y un método mutador para

seleccionar una imagen de textura 3D.

Texture3d(int mipmapMode, int format, int width, int height, int depth)

Construye un objeto **Texture2D** vacío con los valores especificados de mipmapMode, format, width, height y depth. La imagen del nivel 0 la debe seleccionar la aplicación usando el método setImage. Si mipmapMode se selecciona a **MULTI_LEVEL_MIPMAP**, se deben seleccionar las imágenes para TODOS los niveles.

Parámetros:

- **mipmapMode** - tipo mipmap para esta Texture: Uno de **BASE_LEVEL**, **MULTI_LEVEL_MIPMAP**.
- **format** - formato de datos de las texturas grabadas en el objeto. Uno de **INTENSITY**, **LUMINANCE**, **ALPHA**, **LUMINANCE_ALPHA**, **RGB**, **RGBA**.
- **width** - anchura de la imagen del nivel 0. Debe ser una potencia de 2.
- **height** - altura de la imagen del nivel 0. Debe ser una potencia de 2.
- **depth** - profundidad de la imagen del nivel 0. Debe ser una potencia de 2.

Sumario de Métodos de la Clase **Texture3d**

void setBoundaryModeR(int boundaryModeR)

Selecciona el modo de límite para la coordenada R de este objeto texture.

Parámetro:

- **boundaryModeR** - el modo de límite para la coordenada R, uno de: **CLAMP** o **WRAP**.

API de TextureLoader y NewTextureLoader

Esta sección lista los bloques de referencia de las clases **TextureLoader** y **NewTextureLoader**.

La clase **NewTextureLoader** extiende la clase **TextureLoader** proporcionando una utilidad de cargador de texturas más fácil de utilizar -- una que no requiere un observador de imagen del AWT para cada constructor.

API de TextureLoader

Sumario de Campos de **TextureLoader**

GENERATE_MIPMAP

Bandera opcional - especifica los mipmaps generados para todos los niveles.

El siguiente bloque de referencia lista algunos constructores de la clase **TextureLoader**. Hay más constructores que no se listan en este bloque de referencia que permiten la carga de imágenes de textura desde otras fuentes. Puedes consultar la especificación del API Java 3D para ver una lista completa de todos los constructores.

Lista Parcial de Constructores de la Clase **TextureLoader**

Extiende: java.lang.Object

Paquete: com.sun.j3d.utils.image

Esta clase se usa para cargar una textura desde un objeto **Image** o **BufferedImage**. Se proporcionan métodos para recuperar el objeto **Texture** y el objeto **ImageComponent** asociado o una versión escalada del objeto **ImageComponent**.

El formato por defecto es RGBA.

Otros formatos legales son: RGBA, RGBA4, RGB5_A1, RGB, RGB4, RGB5, R3_G3_B2, LUM8_ALPHA8, LUM4_ALPHA4, LUMINANCE y ALPHA

TextureLoader(java.lang.String fname, java.awt.Component observer)
TextureLoader(java.lang.String fname, int flags, java.awt.Component observer)
Contruye un objeto TextureLoader usando el fichero especificado, la opción flags y el formato por defecto RGBA.
TextureLoader(java.net.URL url, java.awt.Component observer)
TextureLoader(java.net.URL url, int flags, java.awt.Component observer)

Construye un objeto TextureLoader usando la URL especificada, opción flags y el formato por defecto RGBA.

Sumario de Métodos de la Clase **TextureLoader**

ImageComponent2D getImage()
Devuelve el objeto ImageComponent2D asociado.
ImageComponent2D getScaledImage(float xScale, float yScale)
Devuelve el objeto ImageComponent2D escalado.
ImageComponent2D getScaledImage(int width, int height)
Devuelve el objeto ImageComponent2D escalado.
Texture getTexture()
Devuelve el objeto Texture asociado.

- **API de NewTextureLoader**

La razón de utilizar **NewTexureLoader** es evitar la necesidad de un observador de imagen para construir un cargador de textura. El siguiente boque de referencia enumera algunos constructores de la clase **NewTextureLoader**.
NewTextureLoader tiene los mismos constructores que **TextureLoader** excepto en que ninguno requiere un componente del awt para servir como el observador de imagen.

Lista Parcial de Constructores de la Clase **NewTextureLoader**

Extiende: com.sun.j3d.utils.image.TextureLoader

Esta clase se usa para cargar una textura desde un fichero o una URL. Esta clase se diferencia de com.sun.j3d.util.image.TextureLoader sólo en la ausencia de un observador de imagen en el constructor y en el método para selecccionar un sólo observador de imagen para todos los usos posteriores.

NewTextureLoader(java.lang.String fname)
NewTextureLoader(java.lang.String fname, int flags)
Construye un objeto TextureLoader usando el fichero especificado, opción flags y

el formato po defecto RGBA.

NewTextureLoader(java.net.URL url)

NewTextureLoader(java.net.URL url, int flags)

Construye un objeto TextureLoader usando la URL especificada, opción flags y el formato por defecto RGBA.

El siguiente bloque de referencia lista los dos métodos definidos en la clase **NewTextureLoader**. Todos los demás métodos están definidos por la clase **TextureLoader**. Para usar un objeto **NewTextureLoader** se debe seleccionar primero un observador de imagen. Esto se hace normalmente cuando se crea el objeto **Canvas3d**.

Lista Parcial de Métodos de la Clase **NewTextureLoader**

java.awt.component getImageObserver()

Devuelve el objeto awt.component usado como el observador de imagen para los objetos NewTextureLoader.

void setImageObserver(java.awt.component imageObserver)

Selecciona un objeto awt.component como el objeto a usar como observador de imagen en la construcción de los siguientes objetos NewTextureLoader.