**Java 3D**

* [Crear Geometrías en Java 3D](about:blank)
  + [Sistema de Coordenadas del Mundo Virtual](about:blank)
  + [Definición Básica de Objeto Visual](about:blank)
    - [Un Ejemplar de Shape3D Define un Objeto Visual](about:blank)
    - [NodeComponent](about:blank)
    - [Definir Clases de Objetos Visuales](about:blank)
  + [Clases de Utilidades Geométricas](about:blank)
    - [Box](about:blank)
    - [Cone](about:blank)
    - [Cylinder](about:blank)
    - [Sphere](about:blank)
    - [Más Sobre los Geométricos Primitivos](about:blank)
    - [ColorCube](about:blank)
    - [Ejemplo: Crear un Simple Yo-Yo desde dos Conos](about:blank)
    - [Geometrías Primitivas](about:blank)
  + [Clases Matemáticas](about:blank)
    - [Clases Point](about:blank)
    - [Clases Color](about:blank)
    - [Clases Vector](about:blank)
    - [Clases TexCoord](about:blank)
  + [Clases Geometry](about:blank)
    - [Clase GeometryArray](about:blank)
    - [Paso 1: Construcción de un objeto GeometryArray vacío](about:blank)
    - [Paso 2: Rellenar con Datos el Objeto GeometryArray](about:blank)
    - [Paso 3: Hacer que los Objetos Shape3D Referencien a los Objetos GeometryArray](about:blank)
    - [Subclases de GeometryArray](about:blank)
    - [Subclases de GeometryStripArray](about:blank)
    - [Subclases de IndexedGeometryArray](about:blank)
    - [Axis.java es un ejemplo de IndexedGeometryArray](about:blank)
  + [Atributos y Apariencia](about:blank)
    - [NodeComponent Appearance](about:blank)
    - [Compartir Objetos NodeComponent](about:blank)
    - [Clases Attribute](about:blank)
    - [Ejemplo: Recortar la cara trasera](about:blank)

**Crear Geometrías en Java 3D**

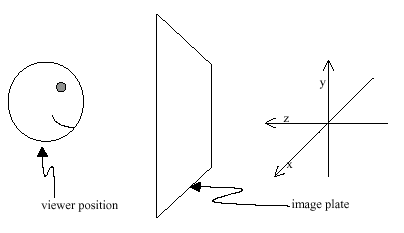
Hay tres formas principales de crear contenidos geométricos. Una forma es usar las clases de utilidades geométricas para **box**, **cone**, **cylinder**, y **sphere**. Otra forma es especificar coordenadas de vértices para puntos, segmentos de líneas y/o superficies poligonales. Una tercera forma es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

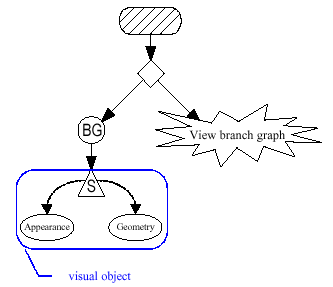


**Definición Básica de Objeto Visual**

**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. |

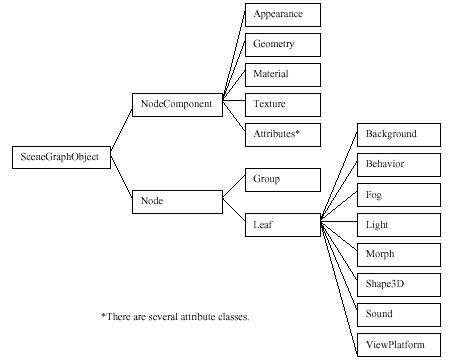
Mientras que el objeto **Shape3D** no esté vivo o compilado, las referencias a los componentes pueden modicarse con los métodos del siguiente recuadro. Estos métodos pueden usarse sobre objetos **Shape3D** vivos o compilados si se configuran las capacidades del objeto primero. El otro recuadro lista 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](http://www.labvis.unam.mx/elio/J3D/#fragmento2-1) 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](http://java.sun.com/products/java-media/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.

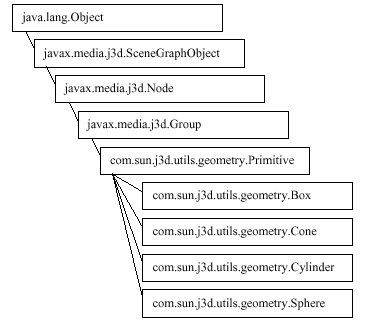
**Clases de Utilidades Geométricas**

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. |

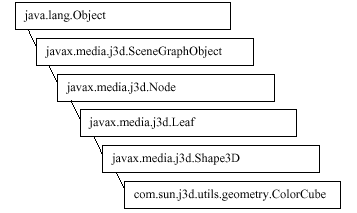
**Más Sobre los Geométricos Primitivos**

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.

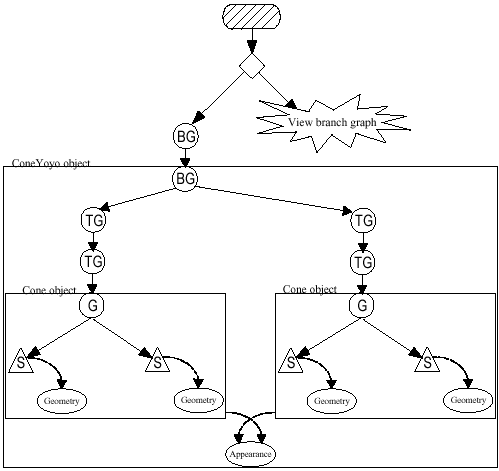


**ColorCube** es la única clase distribuida con el API Java 3D que permite a los programadores ignorar los problemas con los colores y las luces. Por esta razón, la clase **ColorCube** es útil para ensamblar rápidamente escenario gráficos para probar prototipos.

**Ejemplo: Crear un Simple Yo-Yo desde dos Conos**

Esta sección presenta un sencillo ejemplo que usa la clase **Cone**: [ConeYoyoApp.java](http://www.labvis.unam.mx/elio/J3D/.clases/geometry/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)

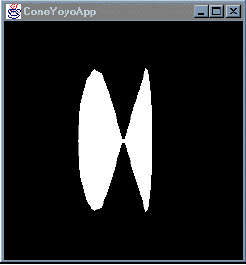
at javax.media.j3d.Group.addChild(Group.java:241)

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](http://www.labvis.unam.mx/elio/J3D/#fragmento2-2) 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](http://www.labvis.unam.mx/elio/J3D/#figura2-4) 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. |

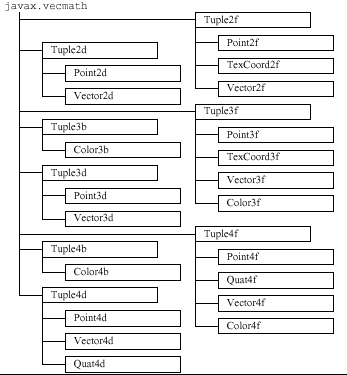
Otros constructores adicionales de **Box**, **Cone**, **Cylinder**, y **Sphere** permiten la especificación de banderas **Primitive** en el momento de la creacción. Puedes consultar la especificación del API Java 3D para más información.

**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](http://www.labvis.unam.mx/elio/J3D/#fragmento2-1).

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)  Seleccciona 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.

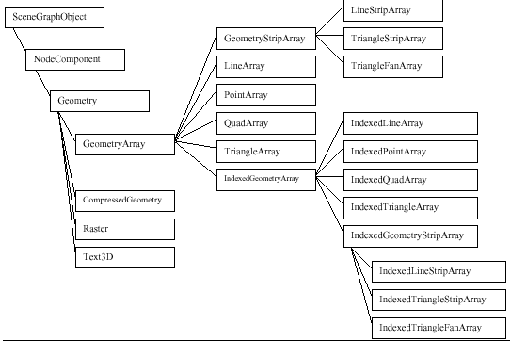
**Clases Geometry**

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](http://www.labvis.unam.mx/elio/J3D/#fragmento2-4) muestra la clase **Axis** del programa [AxisApp.java](http://www.labvis.unam.mx/elio/J3D/.clases/geometry/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](http://www.labvis.unam.mx/elio/J3D/.clases/geometry/AxisApp.java) es diferente de la clase **Axis** definida en [Axis.java](http://www.labvis.unam.mx/elio/J3D/.clases/geometry/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](http://www.labvis.unam.mx/elio/J3D/#fragmento2-5) 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](http://www.labvis.unam.mx/elio/J3D/#fragmento2-6) 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.**

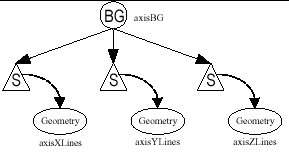
1. axisBG = new BranchGroup();

2.

3. axisBG.addChild(new Shape3D(axisYLines));

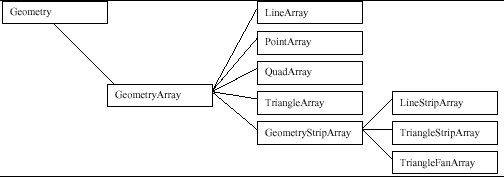
4. axisBG.addChild(new Shape3D(axisZLines));

La Figura 2-11 muestra el escenario gráfico parcial creado por la clase **Axis** en [AxisApp.java](http://www.labvis.unam.mx/elio/J3D/.clases/geometry/AxisApp.java).

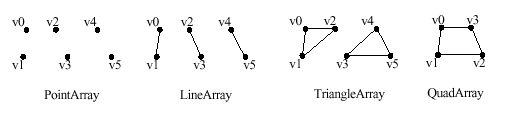


**Subclases de GeometryArray**

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) |

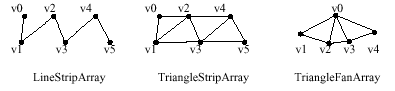
Para ver el uso de estos constructores y métodos, podemos volver a cualquiera de los fragmentos de código [2-4](http://www.labvis.unam.mx/elio/J3D/#fragmento2-4), [2-5](http://www.labvis.unam.mx/elio/J3D/#fragmento2-5) o [2-6](http://www.labvis.unam.mx/elio/J3D/#fragmento2-6), que usan objetos **LineArray** .

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[])) |

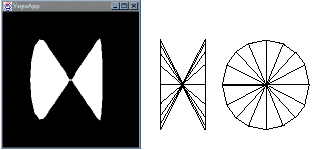
Observa que Java 3D no soporta primitivos rellenos con más de cuatro lados. El programador es responsable de usar mosaicos para descomponer polígonos más complejos en objetos Java 3D. La clases de utilidad **Triangulator** convierte polígonos complejos en triángulos

|  |
| --- |
| 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](http://www.labvis.unam.mx/elio/J3D/.clases/geometry/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](http://www.labvis.unam.mx/elio/J3D/#fragmento2-7) 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](http://www.labvis.unam.mx/elio/J3D/#fragmento2-8). Las líneas 23-26, 36-39 y 46 especifician 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);

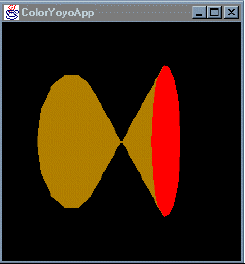
46. tfa.setColors(0,colors);

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**

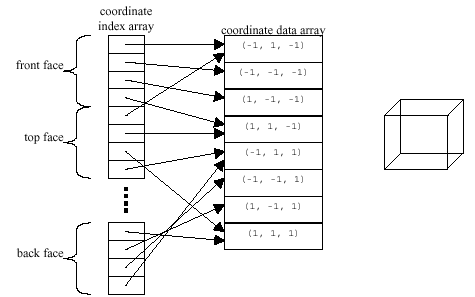
Las subclases de **GeometryArray** descritas anteriormente declaran los vértices de forma borrosa. Solo las subclases de **GeometryStripArray** tienen incluso limitada la reutilización e vértices. Muchos objetos geométricos invitan a la reutilización de vértices. Por ejemplo, para definir un cubo, cada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

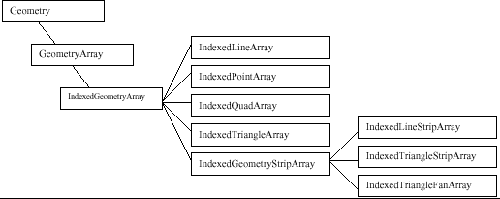
Sin embargo, los objetos **IndexedGeometryArray** también necesitan arrays adicionales ("arrays de índices") que contienen índices a los "arrays de datos". Hay hasta cuatro "arrays de índice": índices de coordenadas, índices de colores, índices de superficies normales, e índices de coordenadas de textura, 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 **IndexedGeometryArray**.



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 **IndexedGeometryArray**  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 **IndexedGeometryStripArray**  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[])) |

**IndexedGeometryArray**, **IndexedGeometryStripArray**, 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**

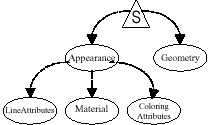
El fichero [Axis.java](http://www.labvis.unam.mx/elio/J3D/.clases/geometry/Axis.java) define el objeto visual **Axis** muy útil para dibujar los ejes y el origen de un universo virtual. También sirve como ejemplo de geometría indexada.

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();

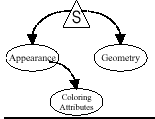
4. app.setColoringAttributes(ca);

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**.

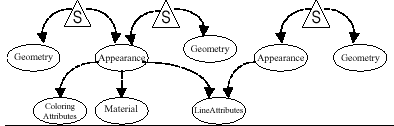
|  |
| --- |
| Constructor de **Appearance**  El constructor por defecto de **Appearance** crea un objeto con todas las referencias a objetos inicializadas a null. Los valores por defecto, para componentes con referencias nulas, normalmente son predecibles: puntos y líneas, se dibujan con un tamaño y 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**.

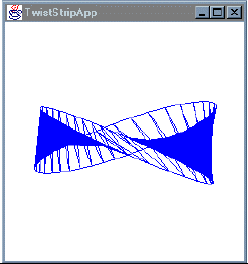
|  |  |
| --- | --- |
| color | white (1, 1, 1) |
| texture environment mode | TEXENV\_REPLACE |
| texture environment color | white (1, 1, 1) |
| depth test enable | true |
| shade model | SHADE\_GOURAUD |
| polygon mode | POLYGON\_FILL |
| transparency enable | false |
| transparency mode | FASTEST |
| 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](http://www.labvis.unam.mx/elio/J3D/.clases/geometry/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);

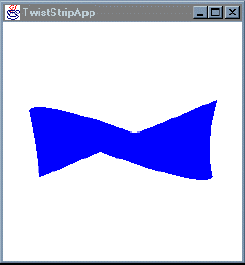
5. // several lines later, after the twistStrip TriangleStripArray has

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.

