**Java 3D**

[Crear Contenidos Sencillos en Java 3D](about:blank)

* + [Cargadores](about:blank)
    - [Ejemplo de Usos de un Loader](about:blank)
    - [Cargadores Disponibles Públicamente](about:blank)
    - [Interfaces y Clases Base del Paquete Loader](about:blank)
    - [Escribir un Loader](about:blank)
  + [GeometryInfo](about:blank)
    - [Sencillo Ejemplo de GeometryInfo](about:blank)
    - [Clases para GeometryInfo](about:blank)
  + [Texto 2D](about:blank)
    - [Ejemplo de Text2D](about:blank)
    - [Clases Usadas para Crear Objetos Text2D](about:blank)
  + [Texto 3D](about:blank)
    - [Ejemplo de Text3D](about:blank)
    - [Clases Usadas en la Creación de Objetos Text3D](about:blank)
  + [Fondo](about:blank)
    - [Ejemplos de fondos](about:blank)
    - [La Clase BoundingLeaf](about:blank)
  + [Datos de Usuario](about:blank)

**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](http://www.labvis.unam.mx/cursos/3d/#fragmento3-1) 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](http://www.labvis.unam.mx/elio/J3D/cursos/3d/cap_4,html)) y luces ([Capítulo 6](http://www.labvis.unam.mx/elio/J3D/cursos/3d/cap_6,html)) 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**.

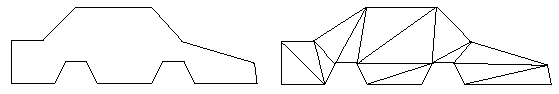
En la extensión del clase **loader** base, la mayoría del trabajo será escribir métodos que reconozcan los distintos tipos de contenidos que se pueden representar en el formato del fichero. Cada uno de esos métodos crea el correspondiente componente Java 3D del escenario gráfico y lo almacena en un objeto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

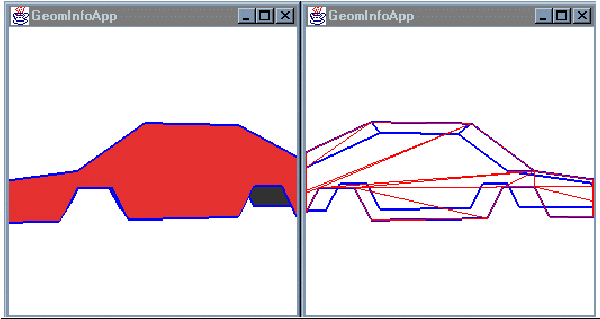
Si no tenemos acceso a los ficheros de modelos geométricos o a software de modelado geométrico, tenemos que crear nuestra geometría a mano. Como se mencionó en capítulos anteriores, esta codificación de geometría a mano requiere mucho tiempo y es una actividad muy propensa a errores. Como sabemos, cuando 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](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/easyContent/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](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento3-2) 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.

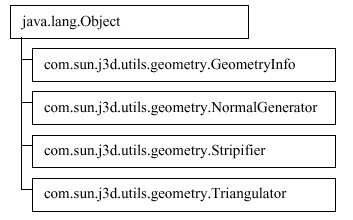
14. Shape3D part = new Shape3D();

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](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento3-3) 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.

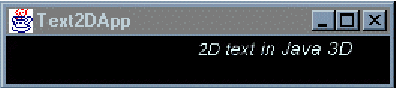
21. Text2D text2D = new Text2D("2D text is a textured polygon",

22. new Color3f(0.9f, 1.0f, 1.0f),

23. "Helvetica", 18, Font.ITALIC));

24. objRoot.addChild(text2D);

[Text2DApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/easyContent/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](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento3-4) muestra el código que modifica el objeto **text2d**, creado en el [Fragmento de Código 3-3](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento3-3).

**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);

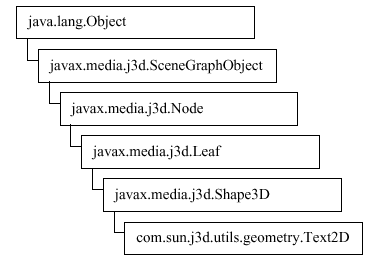
30. polyAttrib.setBackFaceNormalFlip(true);

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](http://www.labvis.unam.mx/elio/J3D/pagina37.html) 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](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento3-5) 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);

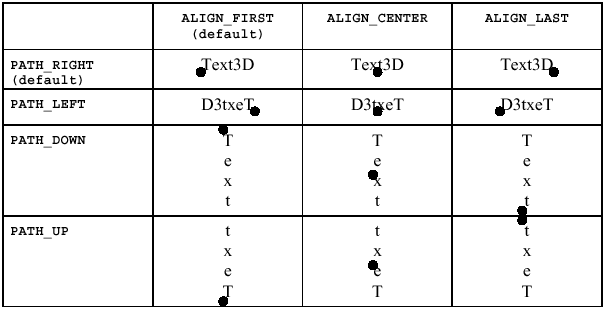
La Figura 3-9 muestra un objeto **Text3D** que ilustra la extrusión del tipo. En la figura, la extrusión se muestra en gris mientras que el tipo se muestra en negro. Para recrear esta figura en Java 3D, son necesarios un objeto **Material** y otro **DirectionalLight**. No podemos seleccionar el color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

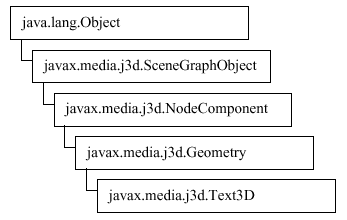
La situación del punto de referencia puede definirse explícitamente sobreescribiendo el camino y la alineación.



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](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento3-6) 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](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento3-7) 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](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/easyContent/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));

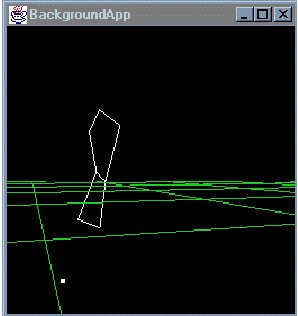
4. objRoot.addChild(backg);

**BackgroundApp.java**

Para apreciar un fondo, necesitamos experimentarlo. [BackgroundApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/easyContent/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.

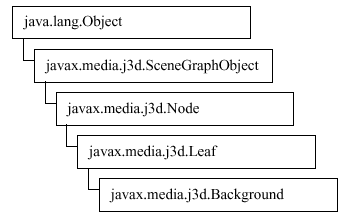
[BackgroundApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/easyContent/BackgroundApp.java) usa la clase **KeyNavigatorBehavior** proprocionada por la librería de utilidades para visores de movimiento.

**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](http://www.labvis.unam.mx/elio/J3D/cursos/3d/cap_4.html). 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. |

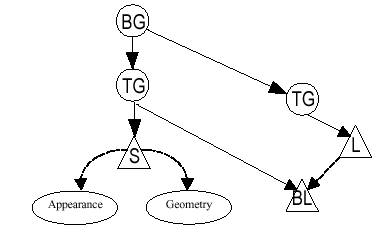
|  |
| --- |
| Sumario de Capacidades de la Clase **Background**   * **ALLOW\_APPLICATION\_BOUNDS\_READ | WRITE** permite leer (escribir) al acceso a los límites de aplicación. * **ALLOW\_COLOR\_READ | WRITE** permite leer (escribir) al acceso a su color * **ALLOW\_GEOMETRY\_READ | WRITE** permite leer (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](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento3-8) presenta un ejemplo de la aplicación **BoundingLeaf** usada con un **Background**.

El [Fragmento de Código 3-8](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento3-8) 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.

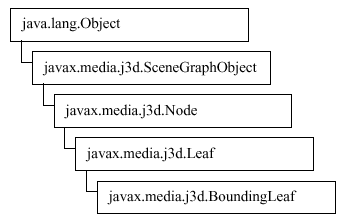
10. Background backg = new Background(1.0f, 1.0f, 1.0f);

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. |