|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **Java 3D**   * [El API Java 3D](about:blank)   + [Construir un Escenario Gráfico](about:blank)   + [Árbol de Clases de Alto Nivel del API Java 3D](about:blank)   + [Receta para Escribir Programas Java 3D](about:blank)   + [Una Sencilla Receta para Escribir Programas Java 3D](about:blank)   + [Alguna Terminología Java 3D](about:blank)   + [Ejemplo de la Receta Sencilla: HelloJava3Da](about:blank)   + [Clases Java 3D Usadas en HelloJava3Da](about:blank)   + [Rotar el Cubo](about:blank)     - [Ejemplo de Combinación de Transformaciones: HelloJava3Db](about:blank)   + [Capacidades y Rendimiento](about:blank)     - [Compilar Contenidos](about:blank)     - [Capacidades](about:blank)   + [Añadir Comportamiento de Animación](about:blank)     - [Especificar un Comportamiento de Animación](about:blank)     - [Funciones de Variación de Tiempo: Mapear un Comportamiento en el Tiempo](about:blank)     - [Región Programada](about:blank)     - [Ejemplo de Comportamiento: HelloJava3Dc](about:blank)     - [Ejemplo de Combinación de Transformation y Behavior: HelloJava3Dd](about:blank)     **El API Java 3D**  Todo programa Java 3D está, al menos, parcialmente ensamblado por objetos del árbol de clases Java 3D. Esta colección de objetos describe un universo virtual, que va a ser renderizado. El API define unas 100 clases presentadas en el paquete javax.media.j3d  Hay cientos de campos y métodos en las clases del API Java 3D. Sin embargo, un sencillo universo virtual que incluya animación puede construirse con unas pocas clases. Este capítulo describe un conjunto mínimo de objetos y sus interacciones para renderizar un universo virtual.  Esta página incluye el desarrollo de un sencillo pero completo programa Java 3D, llamado [HelloJava3Dd.java](http://www.labvis.unam.mx/elio/J3D/clases/HelloJava3D/HelloJava3Dd.java), que muestra un cubo giratorio. El programa de ejemplo se desarrolla de forma incremental, y se presenta en varias versiones, para demostrar cada parte del proceso de programación Java 3D.  Además del paquete corazón de Java 3D, se usan otros paquetes para escribir programas Java 3D. Uno de estos paques es com.sun.j3d.utils el que normalmente se referiere como clases de utilidades de Java 3D. El paquete de las clases corazón incluye sólo las clases de menor nivel necesarias en la programación Java 3D.  Las clases de utilidades son adiciones convenientes y poderosas al corazón. Estas clases se dividen en cuatro categorías: cargadores de contenidos, ayudas a la construcción del escenario gráfico, clases de geometría y utilidades de conveniencia.  Al utilizar las clases de utilidades se reduce significativamente el número de líneas de código en un programa Java 3D. Además de las clases de los paquetes corazón y de utilidades de Java 3D, todo programa 3D usa clases de los paquetes java.awt y javax.vecmath. El paquete java.awt define el "Abstract Windowing Toolkit" (AWT). Las clases AWT crean una ventana para mostrar el renderizado. El paquete javax.vecmath define clases de vectores matemáticos para puntos, vectores, matrices y otros objetos matemáticos.  En el resto del texto, el término **objeto visual** se utilizará para referirnos a un "objeto del escenario gráfico" (por ejemplo, un cubo o una esfera). El término **objeto** sólo se usará para referirse a un ejemplar de una clase. El término **contenido** se usará para referirnos a objetos visuales en un escenario gráfico como un todo.  **Construir un Escenario Gráfico**  Un universo virtual Java 3D se crea desde un escenario gráfico. Un escenario gráfico se crea usando ejemplares de clases Java 3D. El escenario gráfico está ensamblado desde objetos que definen la geometría, los sonidos, las luces, la localización, la orientación y la apariencia de los objetos visuales y sonoros.  Una definición común de un escenario gráfico es una estructura de datos compuesta de nodos y arcos. Un nodo es un elemento dato y un arco es una relación entre elementos datos. Los nodos en un escenario gráfico son los ejemplares de las clases Java 3D. Los arcos representan dos tipos de relaciones entre ejemplares Java 3D.  La relación más común es padre-hijo. Un nodo **Group** puede tener cualquier número de hijos, pero sólo un padre. Un nodo hoja sólo puede tener un padre y no puede tener hijos. La otra relación es una **referencia**. Una referencia asocia un objeto **NodeComponent** con un nodo del escenario gráfico. Los objetos **NodeComponent** definen la geometría y los atributos de apariencia usados para renderizar los objetos visuales.  Un escenario gráfico Java 3D está construido de objetos nodos con relaciones padre-hijo formando una estructura de árbol. En una estructura de árbol, un nodo es el raíz. Se peude acceder a otros nodos siguiendo los arcos desde el raíz. Los nodos de un árbol no forman bucles. Un escenario gráfico está formado desde los árboles con raíces en los objetos **Locale**. Los **NodeComponents** y las referencias a arcos no forman parte del escenario gráfico.  Sólo existe un camino desde la raíz de un árbol a cada una de las hojas; por lo tanto, sólo hay un camino desde la raíz hasta el escenario gráfico de cada nodo hoja. El camino desde la raíz de un escenario gráfico hasta una hoja especificada es el camino al escenario gráfico del nodo hoja. Como un camino de un escenario gráfico trata exactamente con un sola hoja, hay un camino de escenario gráfico para cada hoja en el escenario.  Todo camino de escenario gráfico en un escenario gráfico Java 3D especifica completamente la información de estado de su hoja. Esta información incluye, la localización, la orientación y el tamaño del objeto visual. Consecuentemente, los atributos visuales de cada objeto visual dependen sólo de su camino de escenario gráfico. El renderizador Java 3D se aprovecha de este echo y renderiza las hojas en el orden que él determina más eficiente. El programador Java 3D normalmente no tiene control sobre el orden de renderizado de los objetos.  Las representaciones gráficas de un escenario gráfico pueden servir como herramienta de diseño y/o documentación para los programas Java 3D. Los escenarios gráficos se dibujan usando símbolos gráficos estándards como se ve en la Figura 1-1. Los programas Java 3D podrían tener más objetos que los que hay en su escenerio gráfico.  Para diseñar un universo virtual Java 3D se dibuja un escenario gráfico usando un conjunto de símbolos estándard. Después de completar el diseño, este escenerio gráfico es la especificación para el programa. Después de completar el programa, el mismo escenario gráfico es una representación concisa del programa (asumiendo que se siguió la especificación).  Figura 1-1. Símbolos que Representan Objetos en un Escenario Gráfico  Cada uno de los símbolos mostrados al lado izquierdo de la Figura 1.1 representa un sólo objeto cuando se usa en un escenario gráfico. Los dos primeros símbolos representan objetos de clases específicas: **VirtualUniverse** y **Locale**. Lo siguientes tres símbolos de la izquierda representan objetos de las clases **Group**, **Leaf**, y **NodeComponent**. Estos tres símbolos normalmente tienen anotaciones para indicar la subclase del objeto específico. El último símbolo se usa para representar otras clases de objetos.  El símbolo de la flecha sólida representa una relación padre-hijo entre dos objetos. La flecha punteada es una referencia a otro objeto. Los objetos referenciados pueden ser compartidos entre diferentes ramas de una escenario gráfico. En la Figura 1-2, podemos ver un sencillo escenario gráfico:  Figura 1-2. Primer Ejemplo de Escenario Gráfico  Es posible crear un escenario gráfico ilegal. Podemos ver uno en la Figura 1-3. Este escenario es ilegal porque viola las propiedades de un DAG. El problema son los dos objetos **TransformGroup** que tienen al mismo objeto **Shape3D** como hijo. Recuerda que una hoja sólo puede tener un padre. En otras palabras, sólo puede haber un camino desde el objeto **Locale** hasta la hoja (o un camino desde la hoja hasta el objeto **Loale**).  Podríamos pensar que la estructura mostrada en la figura 1-3 define tres objetos visuales en un universo virtual. Pero el escenario gráfico define dos objetos visuales que re-usan el objeto visual (Shape3D) del lado derecho de la figura. Conceptualmente, cada objeto **TransformGroup** que apadrina al ejemplar compartido de **Shape3D** podría situar una imagen en el objeto visual en diferentes localizaciones. Sin embargo, es un escenario gráfico ilegal porque el arco padre-hijo no forma un árbol. En este ejemplo, el resultado es que el objeto **Shape3D** tiene más de un padre.  Las explicaciones del árbol y de las estructuras DAG son correctas. Sin embargo, el sistema de ejecución Java 3D reporta el error en términos de la relación hijo-padre. Un resultado de la limitación de la estructura de árbol es que cada objeto **Shape3D** está limitado a un sólo padre. Para el ejemplo de la Figura 1-3, se lanzará una excepción **'multiple parent'** en el momento de la ejecución. La Figura 1-4, con un padre para cada objeto **Shape3D**, muestra una posible solución para este escenario gráfico.  Figura 1-3. Un Escenario Gráfico Ilegal  Un programa Java 3D que define un escenario gráfico ilegal podría compilarse, pero no se renderiza. Cuando se ejecuta un programa Java 3D que define un escenario gráfico ilegal, el sistema Java 3D detecta el problema y lanza una excepción. El programa podría estár ejecutandose y consecuentemente deberíamos pararlo. Sin embargo, no se renderizará ninguna imagen.  Figura 1-4. Posible solución al escenario gráfico ilegal de la Figura 1-3  Cada escenario gráfico tiene un sólo **VirtualUniverse**. Este objeto tiene una lista de objetos **Locale**. Un objeto **Locale** proporciona una referencia a un punto en el universo virtual. Podemos pensar en los objetos **Locale** como marcas de tierra que determinan la localización de los objetos visuales en el universo virtual.  Es técnicamente posible para un programa Java 3D tener más de un objeto **VirtualUniverse**, y así definir más de un universo virtual. Sin embargo, no hay ninguna forma de comunicación entre los universos virtuales. Además, un objeto de un escenario gráfico no puede existir en más de un universo virtual. Es altamente recomendable usar uno y sólo un ejemplar de **VirtualUniverse** en cada programa Java 3D.  Mientras que un objeto **VirtualUniverse** podría referenciar muchos objetos **Locale**, la mayoría de los programas Java 3D tiene un sólo objeto **Locale**. Cada objeto**Locale** puede servir de raíz para varios sub-gráficos del escenario gráfico. Por ejemplo, si nos referimos a la Figura 1-2 podremos observar las dos ramas sub-gráficas que salen desde el objeto **Locale**.  Un objeto **BranchGroup** es la raíz de un sub-gráfico, o rama gráfica. Hay dos categorias de escenarios sub-gráficos: la rama de vista gráfica y la rama de contenido gráfico. La rama de contenido gráfico especifica el contenido del universo virtual - geometría, apariencia, comportamiento, localización, sonidos y luces. La rama de vista gráfica especifica los parámetros de visualización, como la posición de visualización y la dirección. Juntas, las dos ramas especifican la mayoría del trabajo que el renderizador tiene que hacer.  **Árbol de Clases de Alto Nivel del API Java 3D**  En la Figura 1-5 podemos ver los tres primeros niveles del árbol de clases del API Java 3D. En esta parte del árbol aparecen las clases **VirtualUniverse**, **Locale**, **Group**, y **Leaf**.  **SceneGraphObject** es la superclase de casi todas las clases corazón y de utilidad de Java 3D. Tiene dos subclases: **Node** y **NodeComponent**. Las subclases de **Node** proporcionan la mayoría de los objetos de un escenario gráfico. Un objeto **Node** es un objeto nodo **Group** o un objeto nodo **Leaf**.  **Clase Node**  La clase **Node** es una superclase abstracta de las clases **Group** y **Leaf**. Esta clase define algunos de los métodos importantes de sus subclases. Las subclases de **Node** componen escenarios gráficos.  **Clase Group**  La clase **Group** es la superclase usada en especificación de localización y orientación de objetos visuales en el universo virtual. Dos de las subclases de **Group** son: **BranchGroup** y **TransformGroup**. En la representación gráfica de un escenario gráfico, los simbolos de **Group** (círculos) normalmente se anotan con **BG** para **BranchGroups**, **TG** para **TransformGroups**, etc. La Figura 1-2 muestra algunos ejemplos de esto.  **Clase Leaf**  La clase **Leaf** es la superclase usada para especificar la forma, el sonido y comportamiento de los objetos visuales en el universo virtual. Algunas de las subclases de **Leaf** son: **Shape3D**, **Light**, **Behavior**, y **Sound**. Estos objetos podrían no tener hijos pero podrían referenciar a **NodeComponents**.  **Clase NodeComponent**  La clase **NodeComponent** es la superclase usada para especificar la geometría, la apariencia, la textura y las propiedades de material de un nodo **Shape3D** (Leaf). Los **NodeComponent**s no forman parte del escenario gráfico, pero son referenciados por él. un **NodeComponent** podría ser referenciado por más de un objeto **Shape3D**.  Figura 1-5. Introdución al Árbol de Clases de API Java 3D  **Receta para Escribir Programas Java 3D**  Las subclases de **SceneGraphObject** son los ladrillos que se ensamblan en los escenarios gráficos. La línea básica de desarrollo de un programa Java 3D consiste en siete pasos (a los que la especificación del API Java 3D se referiere como un **Receta**) presentados a continuación. Esta receta puede usarse para ensamblar muchos útiles programas Java 3D.   1. Crear un Objeto **Canvas3D** 2. Crear un objeto **VirtualUniverse** 3. Crear un objeto **Locale**, adjuntarlo al objeto **VirtualUniverse** 4. Construir la rama de vista gráfica    * Crear un objeto **View**    * Crear un objeto **ViewPlatform**    * Crear un objeto **PhysicalBody**    * Crear un objeto **PhysicalEnvironment**    * Adjuntar los objetos **ViewPlatform**, **PhysicalBody**, **PhysicalEnvironment**, y **Canvas3D** al objeto **View** 5. Construir la(s) rama(s) gráfica(s) de contenido 6. Compilar la(s) rama(s) gráfica(s) 7. Insertar los subgráficos dentro del objeto **Locale**   Esta receta ignora algunos detalles pero ilustra el concepto fundamental para toda la programación Java 3D: crear la rama gráfica del escenario gráfico es la programación principal. En vez de ampliar esta receta, la siguiente sección explica una forma sencilla de construir un escenario gráfico muy similar con menos programación.  **Una Sencilla Receta para Escribir Programas Java 3D**  Los programas Java 3D escritos usando la receta básica tienen ramas de vista gráfica con idéntica estructura. La regularidad de la estructura de las ramas de vista gráfica tambien se encuentra en la clase de utilidad **SimpleUniverse**. Los ejemplares de esta clase realizan los pasos 2, 3 y 4 de la receta básica. Usando la clase **SimpleUniverse** en programación Java 3D se reduce significativamente el tiempo y el esfuerzo necesario para crear las ramas de vista gráfica. Consecuentemente, el programador tiene más tiempo para concentrarse en el contenido. Esto de lo que se trada el escribir programas Java 3D.  La clase **SimpleUniverse** es un buen punto de inicio en la programación Java 3D, porque permite al programador ignorar las ramas de vista gráfica. Sin embargo, usar **SimpleUniverse** no permite tener varias vistas de un universo virtual.  La clase **SimpleUniverse** se usa en todos los ejemplos de programación de este tutorial.  **La clase SimpleUniverse**  El constructor de **SimpleUniverse** crea un escenario gráfico que incluye un objeto **VirtualUniverse** y **Locale**, y una rama de vista gráfica completa. Esta rama gráfica creada usa un ejemplar de las clases de conveniencia **ViewingPlatform** y **Viewer** en lugar de las clases corazón usadas para crear una rama de vista gráfica. Observa que **SimpleUniverse** sólo usa indirectamente los objetos **View** y **ViewPlatform** del corazón Java 3D. Los objetos **SimpleUniverse** suministran la funcionalidad de todos los objetos que hay dentro del recuadro azul de la Figura 1-7.  El paquete com.sun.j3d.utils.universe contiene **SimpleUniverse**, **ViewingPlatform**, y clases **Viewer** de conveniencia.  Figura 1-7. La clase SimpleUniverse proporciona un universo virtual mínimo, indicado por la línea punteada azul  Al usar los objetos **SimpleUniverse** la receta básica se simplifica:   1. Crear un objeto **Canvas3D** 2. Crear un objeto **SimpleUniverse** que referencia al objeto **Canvas3D** anterior    * Personalizar el objeto **SimpleUniverse** 3. Construir la rama de contenido 4. Compilar la rama de contenido gráfico 5. Insertar la rama de contenido gráfico dentro del objeto **Locale** de **SimpleUniverse**  |  | | --- | | **Constructores de SimpleUniverse**  Paquete: **com.sun.j3d.utils.universe**  Esta clase configura un entorno de usuario mínimo para obtener rápida y fácilmente un programa Java 3D y ejecutarlo.  Esta clase de utilidad crea todos los objetos necesarios para la rama de vista gráfica. Especificamente crea los objetos **Locale**, **VirtualUniverse**, **ViewingPlatform**, y **Viewer** (todos con sus valores por defecto). Los objetos tiene las relaciones apropiadas para formar la rama de vista gráfica.  SimpleUniverse proporciona toda la funcionalidad necesaria para muchas aplicaciones Java 3D básicas. **Viewer** y **ViewingPlatform** son clases de conveniencia. estas clases usan las clases **View** y **ViewPlatform** del corazón Java.  SimpleUniverse()  Construye un sencillo universo virtual.  SimpleUniverse(Canvas3D canvas3D)  Construye un sencillo universo virtual con una referencia al objeto **Canvas3D** nombrado. |   El objeto **SimpleUniverse** crea una rama de vista gráfica completa para un universo virtual. Esta rama incluye un plato de imagen. Un plato de imagen es el rectángulo conceptual donde se proyecta el contenido para formar la imagen renderizada. El objeto **Canvas3D**, que proporciona una imagen en una ventana de nuestra pantalla, puede ser el plato de imagen.  La Figura 1-9 muestra la relación entre el plato de imagen, la posición del ojo, y el universo virtual. La posición del ojo está detrás del plato de imagen. Los objetos visuales delante del plato de imagen son renderizados en el plato de imagen. El renderizado puede ser como una proyección de los objetos visuales sobre el plato de imagen. Esta idea se ilustra con los cuatro proyectores de la imagen (líneas punteadas).  Figura 1-9. Dibujo Conceptual de un Plato de Imagen y la Posición del Ojo en un Universo Virtual  Por defecto, el plato de imagen está centrado en el origen de **SimpleUniverse**. La orientación por defecto es mirando hacia abajo el eje Z. Desde esta posición, el eje X es una línea horizontal que atraviesa el plato de imagen con los valores positivos hacia la derecha. El eje Y es una línea vertical que atraviesa el centro del plato de imagen, con los valores positivos arriba. Consecuentemente, el punto (0,0,0) es el centro del plato de imagen.  Los típicos programas Java 3D mueven la vista hacía atrás (z positivo) para hacer que los objetos se acerquen, al origen dentro de la vista. La clase **SimpleUniverse** tiene un miembro que es un objeto de la clase **ViewingPlatform**. Esta clase tiene un método **setNominalViewingTransform** que selecciona la posición del ojo para que esté centrado en (0, 0, 2.41) buscando en dirección z negativa hacia el origen.   |  | | --- | | El Método **ViewingPlatform setNominalViewingTransform()**  Paquete: com.sun.j3d.utils.universe  La clase **ViewingPlatform** se usa para configurar la rama de vista gráfica de un escenario gráfico Java 3D en un objeto **SimpleUniverse**. Este método normalmente se usa en conjunción con el método **getViewingPlatform** de la clase **SimpleUniverse**.  void setNominalViewingTransform()  Selecciona la distancia nominal de la vista a una distancia de aproximadamente 2,42 metros en la vista de transformación de un SimpleUniverse. Desde esta distancia y con el campo de vista por defecto, los objetos con 2 metros de altura o de anchura generalmente entran en el plato de imagen. |   Después de crear los objetos **Canvas3D** y **SimpleUniverse**, el siguiente paso es la creacción de la rama de contenido gráfico. La regularidad de estructura encontrada en la rama de vista gráfica no existe para la rama de contenido gráfico. La rama de contenido varia de un programa a otro haciendo imposible obtener los detalles de su construcción en una receta. Esto también significa que no hay una clase de "contenido sencillo" para ningún universo que podamos querer ensamblar.  Después de crear la rama de contenido gráfico, se inserta dentro del universeo usando el método **addBranchGraph** de **SimpleUniverse**. Este método toma un ejemplar de **BranchGroup** como único argumento. Este **BranchGroup** se añade como hijo del objeto **Locale** creado por **SimpleUniverse**.   |  | | --- | | Lista Parcial de Métodos de **SimpleUniverse**  Paquete: com.sun.j3d.utils.universe  void addBranchGraph(BranchGroup bg)  Se usa para añadir Nodos al objeto **Locale** del escenario gráfico creado por el **SimpleUniverse**. Se usa para añadir una rama de contenido gráfico al universo virtual.  ViewingPlatform getViewingPlatform()  Se usa para recuperar el objeto **ViewingPlatform** del **SimpleUniverse** ejemplarizado. Este método se usa con el método **setNominalViewingTransform()** de **ViewingPlatform** para ajustar la localización de la posición de vista. |   **Alguna Terminología Java 3D**  Insertar una rama gráfica dentro de un **Locale** la hace **viva**, y consecuentemente, cada uno de los objetos de esa rama gráfica también están vivos. Hay algunas consecuencias cuando un objeto se convierte en **vivo**. Los objetos vivos estan sujetos s renderización. Los parámetros de los objetos vivos no pueden ser modificados a menos que la capacidad correspondiente haya sido seleccionada especificamente antes de que el objeto esté vivo.  Los objetos **BranchGroup** pueden ser compilados. Compilar un **BranchGroup** lo convierte a él y a todos sus ancestros en una forma más eficiente para el renderizado. Compilar los objetos **BranchGroup** está recomendado como el último paso antes de hacerlo vivir. Es mejor compilar solo los objetos **BranchGroup** insertados dentro de objetos **Locale**   |  | | --- | | Método **BranchGroup compile()**  void compile()  compila la fuente **BranchGroup** asociada con este objeto creado y cacheando un escenario gráfico compilado. |   Los conceptos de **compilado** y **vivo** se implementan en la clase **SceneGraphObject**. Abajo podemos ver los dos métodos de la clase **SceneGraphObject** que se relacionan con estos conceptos.   |  | | --- | | Lista Parcial de Métodos de **SceneGraphObject**  **SceneGraphObject** es la superclase usada para crear un escenario gráfico incluyendo **Group**, **Leaf**, y **NodeComponent**. **SceneGraphObject** proporciona varios métodos y campos comunes para sus subclases:  boolean isCompiled()  Devuelve una bandera indicando si el nodo forma parte de un escenario gráfico que ha sido compilado.  boolean isLive()  Devuelve una bandera que indica si el nodo forma parte de un escenario gráfico vivo. |   Observa que no hay un paso "Empezar a renderizar" en ninguna de las recetas anteriores. El renderizador Java 3D empieza a funcionar en un bucle infinito cuando una rama gráfica que contiene un ejemplar de **View** se vuelve vivo en un universo virtual. Una vez arrancado, el renderizador Java 3D realiza las operaciones mostradas en el siguiente listado:  while(true) {  Procesos de entrada  If (petición de salida) break  Realiza comportamientos  Atraviesa el escenario gráfico  y renderiza los objetos visuales  }  Limpieza y salida  Las secciones anteriores explicaban la construcción de un sencillo universo virtual sin una rama de contenido gráfico. La creacción de esta rama es el objetivo de las siguientes secciones.  **Ejemplo de la Receta Sencilla: HelloJava3Da**  El programa Java 3D típico empieza definiendo una nueva clase que extiende la clase **Applet**. El ejemplo [HelloJava3Da.java](http://www.labvis.unam.mx/elio/J3D/clases/HelloJava3D/HelloJava3Da.java) es una clase definida para extender la clase **Applet**. Los programas Java 3D podrían escribirse como aplicaciones, pero usar Applets nos ofrece una forma más sencilla de producir una aplicación con ventanas.  La clase principal de un programa Java 3D normalmente define un método para construir la rama de contenido gráfico. En el ejemplo **HelloJava3Da** dicho método está definido como **createSceneGraph()**. Los pasos de la receta sencilla se implementan en el constructor de la clase **HelloJava3Da**. El paso 1, crear un objeto **Canvas3D**, se completa en la línea 4. El paso 2, crear un objeto **SimpleUniverse**, se hace en la línea 11. El paso 2a, personalizar el objeto **SimpleUniverse**, se realiza en la línea 15. El paso 3, construir la rama de contenido, se realiza en la llamada al método **createSceneGraph()**. El paso 4, compilar la rama de contenido gráfico, se hace en la línea 8. Finalmente el paso 5, insertar la rama de contenido gráfico en el objeto **Locale** del **SimpleUniverse**, se completa en la línea 16:  **Fragmento de código 1-1. La clase HelloJava3D**  1. public class HelloJava3Da extends Applet {  2. public HelloJava3Da() {  3. setLayout(new BorderLayout());  4. Canvas3D canvas3D = new Canvas3D(null);  5. add("Center", canvas3D);  6.  7. BranchGroup scene = createSceneGraph();  8. scene.compile();  9.  10. // SimpleUniverse is a Convenience Utility class  11. SimpleUniverse simpleU = new SimpleUniverse(canvas3D);  12.  13. // This moves the ViewPlatform back a bit so the  14. // objects in the scene can be viewed.  15. simpleU.getViewingPlatform().setNominalViewingTransform();  16.  17. simpleU.addBranchGraph(scene);  18. } // end of HelloJava3Da (constructor)  El paso 3 de esta sencilla receta es crear la rama de contenido gráfico. Esta rama se crea en el Fragmento de código 1-2. Probablemente sea la rama de contenido gráfico más sencilla posible. Contiene un objeto gráfico estático, un **ColorCube**. Éste está localizado en el origen del sistema de coordenadas del universo virtual. Con la localización y orientación dadas de la dirección de la vista del cubo, el cubo aparece como un rectángulo cuando es renderizado. La imagen que mostrará este programa la podemos ver en la Figura 1-12:  **Fragmento de código 1-2. Método createSceneGraph de la clase HelloJava3D**  1. public BranchGroup createSceneGraph() {  2. // Create the root of the branch graph  3. BranchGroup objRoot = new BranchGroup();  4.  5. // Create a simple shape leaf node, add it to the scene graph.  6. // ColorCube is a Convenience Utility class  7. objRoot.addChild(new ColorCube(0.4));  8.  9. return objRoot;  10. } // end of createSceneGraph method of HelloJava3Da  11. } // end of class HelloJava3Da  La clase **HelloJava3Da** está derivada de Applet pero el programa puede ejecutarse como una aplicación con el uso de la clase **MainFrame**. La clase Applet se usa como clase base para hacer más fácil la escritura de un programa Java 3D que se ejecuta en una ventana. **MainFrame** proporciona un marco AWT (ventana) para un applet permitiendo que el applet se ejecute como una aplicación. El tamaño de la ventana de la aplicación resultante se especifica en la construcción de la clase **MainFrame**. El Fragmento de Código 1-3 muestra el uso de la clase **MainFrame** en **HelloJava3Da.java**.   |  | | --- | | Lista Parcial de Constructores de **MainFrame**  paquete: com.sun.j3d.utils.applet  **MainFrame** crea un applet en una aplicación. Una clase derivada de Applet podría tener un método **main()** que llame al constructor **MainFrame**. **MainFrame** extiende java.awt.Frame e implementa java.lang.Runnable, java.applet.AppletStub, y java.applet.AppletContext. La clase **MainFrame** es Copyright © 1996-1998 de Jef Poskanzer email: [jef@acme.com](mailto:jef@acme.com) en <http://www.acme.com/java/>.  MainFrame(java.applet.Applet applet, int width, int height)  Crea un objeto **MainFrame** que ejecuta un applet como una aplicación.  Parámetros:   * applet - el constructor de una clase derivada de Applet. MainFrame proporciona un marco AWT para este applet. * width - la anchura de la ventana en pixels * height - la altura de la ventana en pixels |   **Fragmento de código 1-3. Método Main() de HelloJava3Da Invoca a MainFrame**  1. // The following allows this to be run as an application  2. // as well as an applet  3.  4. public static void main(String[] args) {  5. Frame frame = new MainFrame(new HelloJava3Da(), 256, 256);  6. } // end of main (method of HelloJava3Da)  Los tres fragmentos de código anteriores (1-1, 1-2, y 1-3) forman un programa Java 3D completo cuando se usan las sentencias import adecuadas. Aquí podemos ver las sentencias import necesarias para compilar la clase **HelloJava3Da**. Las clases más comunmente usadas en Java 3D se encuentran en los paquetes javax.media.j3d, o javax.vecmath. En este ejemplo, sólo la clase de utilidad **ColorCube** se encuentra en el paquete com.sun.j3d.utils.geometry. Consecuentemente, la mayoría de los programas Java 3D tienen las sentencias **import** mostradas en el Fragmento de Código 1-4 con la excepción de **ColorCube**.  **Fragmento de código 1-4. Sentencias Import para HelloJava3Da**  1. import java.applet.Applet;  2. import java.awt.BorderLayout;  3. import java.awt.Frame;  4. import java.awt.event.\*;  5. import com.sun.j3d.utils.applet.MainFrame;  6. import com.sun.j3d.utils.universe.\*;  7. import com.sun.j3d.utils.geometry.ColorCube;  8. import javax.media.j3d.\*;  9. import javax.vecmath.\*;  En el programa de ejemplo **HelloJava3Da.java**, sólo se sitúo un objeto gráfico en una única localización. En la figura 1-11 podemos ver el escenario gráfico resultante:  Figura 1-11 Escenario Gráfico del ejemplo HelloJava3Da  Compilamos el programa con el comando javac HelloJava3Da.java. Y lo ejecutamos con el comando: java HelloJava3Da. La imagen producida por el programa **HelloJava3Da** se puede ver en la Figura 1-12.  Figura 1-12 Imagen Producida por HelloJava3Da  Como no se explica cada línea de código del ejemplo **HelloJava3Da**, las ideas básicas de ensamblar un programa Java 3D deberían estar claras habiendo leído el ejemplo. La siguiente sección presenta cada una de las clases usadas en el programa.  **Clases Java 3D Usadas en HelloJava3Da**  Para añadir un poco de entendimiento del API Java 3D y el ejemplo **HelloJava3Da** aquí presentamos una sinopsis de las clases del API Java 3D usadas en **HelloJava3Da**.  **Clase BranchGroup**  Los objetos de este tipo se usan para formar escenarios gráficos. Los ejemplares de **BranchGroup** son la raíz de los sub-gráficos. Los objetos **BranchGroup** son los únicos que pueden ser hijos de los objetos **Locale**. Los objetos **BranchGroup** pueden tener varios hijos. Los hijos de un objeto **BranchGroup** pueden ser otros objetos **Group** o **Leaf**.   |  | | --- | | Constructor por defecto de **BranchGroup**  BranchGroup()  Los ejemplares de **BranchGroup** sirven como raíz para las ramas del escenario gráfico; los objetos **BranchGroup** son los únicos objetos que pueden insertarse en un conjunto de objetos **Locale**. |   **Clase Canvas3D**  La clase **Canvas3D** deriva de la clase **Canvas** del AWT. Al menos un objeto **Canvas3D** debe ser referenciado en la rama de vista gráfica del escenario gráfico.   |  | | --- | | Constructor de **Canvas3D**  Canvas3D(GraphicsConfiguration graphicsconfiguration)  Construye e inicializa un nuevo objeto **Canvas3D** que el Java 3D puede renderizar dando un objeto **GraphicsConfiguration** válido. Es una extensión de la clase **Canvas** del AWT. |   **Clase Transform3D**  Los objetos **Transform3D** representan transformaciones de geometrías 3D como una traslación o una rotación. Estos objetos normalmente sólo se usan en la creacción de un objeto **TransformGroup**. Primero, se construye el objeto **Transform3D**, posiblemente desde una combinación de objetos **Transform3D**. Luego se construye el objeto **TransformGroup** usando el objeto **Transform3D**.   |  | | --- | | Constructor por Defecto de **Transform3D**  Un objeto de transformación generalizado se representa internamente como una matriz de 4x4 doubles de punto flotante. La representación matemática es la mejor forma. Un objeto **Transform3D** no se usa en un escenario gráfico. Se usa para especificar la transformación de un objeto **TransformGroup**.  Transform3D()  Construye un objeto **Transform3D** que representa la matriz de identidad (no la transformación). |   Un objeto **Transform3D** puede representar una traslación, una rotación, un escalado, o una combinación de éstas. Cuando se especifica una rotación, el ángulo se expresa en radianes. Una rotación completa es **2 PI radianes**. Una forma de especificar ángulos es usar la constante Math.PI. Otra forma es especificar los valores directamente. Algunas aproximaciones son: 45º es 0.785, 90º es 1.57, y 180º es 3.14.   |  | | --- | | Lista Parcial de Métodos de **Transform3D**  Los objetos **Transform3D** representan transformaciones geométricas como una rotación, traslación y escalado. **Transform3D** es una de las pocas clases que no se usan directamente en un escenario gráfico. Las transformaciones representadas por objetos **Transform3D** se usan para crear objetos **TransformGroup** que si se usan en escenarios gráficos.  void rotX(double angle)  Selecciona el valor de esta transformación a una rotación en contra del sentido del reloj sobre el eje-x. El ángulo se especifica en radianes.  void rotY(double angle)  Selecciona el valor de esta transformación a una rotación en contra del sentido del reloj sobre el eje-y. El ángulo se especifica en radianes.  void rotZ(double angle)  Selecciona el valor de esta transformación a una rotación en contra del sentido del reloj sobre el eje-z. El ángulo se especifica en radianes.  void set(Vector3f translate)  Selecciona el valor transacional de esta matriz al valor del parámetro **Vector3f**, y selecciona los otros componentes de la matriz como si ésta transformación fuera una matriz idéntica. |   **Clase TransformGroup**  Como una subclase de la clase **Group**, los ejemplares de **TransformGroup** se usan en la creacción de escenarios gráficos y tienen una colección de objetos nodos como hijos. Los objetos **TransformGroup** contienen transformaciones geométricas como traslaciones y rotaciones. La transformación normalmente se crea en un objeto **Transform3D**, que no es un objeto del escenario gráfico.   |  | | --- | | Constructores de **TransformGroup**  Los objetos **TransformGroup** son contenedores de transformaciones en el escenario gráfico.  TransformGroup()  Construye e inicializa un **TransformGroup** usando una identidad de transformación.  TransformGroup(Transform3D t1)  Construye e inicializa un **TransformGroup** desde un objeto **Transform3D** pasado:  Parámetros:   * t1 - el objeto transform3D |   La transformación contenida en un objeto **Transform3D** se copia a un objeto **TransformGroup** o cuando se crea el **TransformGroup**, o usando el método **setTransform()**.   |  | | --- | | Método **setTransform()** de TransformGroup  void setTransform(Transform3D t1)  Selecciona el componente de transformación de este **TransformGroup** al valor de la transformación pasada.  Parámetros:   * t1 - la transformación a copiar. |   **Clase Vector3f**  **Vector3f** es una clase matemática que se encuentra en el paquete javax.vecmath para especificar un vector usando tres valores de punto flotante. Los objetos **Vector** se usan frecuentemente para especificar traslaciones de geometrías. Los objetos **Vector3f** no se usan directamente en la construcción de un escenario gráfico. Se usan para especificar la traslaciones, superficies normales, u otras cosas.   |  | | --- | | Constructores de **Vector3f**  Un vector de 3 elementos que es representado por puntos flotantes de precisión sencilla para las coordenadas x, y, y z.  Vector3f()  Construye e inicializa un **Vector3f** a (0,0,0).  Vector3f(float x, float y, float z)  Construye e inicializa un **Vector3f** desde las coordenadas x, y, z especificadas. |   **Clase ColorCube**  **ColorCube** es una clase de utilidad que se encuentra en el paquete com.sun.j3d.utils.geometry que define la geometría y colores de un cubo centrado en el origen y con diferentes colores en cada cara. El objeto **ColorCube** es un cubo que tiene 2 métros de arista. Si un cubo sin rotar se sitúa en el origen (como en **HelloJava3Da**), se verá la cara roja desde la localización de visión nominal. Los otros colores son azul, magenta, amarillo, verde y cian.   |  | | --- | | Constructores de **ColorCube**  Paquete: com.sun.j3d.utils.geometry  Un **ColorCube** es un objeto visual, un cubo con un color diferente en cada cara. **ColorCube** extiende la clase **Shape3D**; por lo tanto, es un nodo hoja. **ColorCube** es fácil de usar cuando se pone junto a un universo virtual.  ColorCube()  Construye un cubo de color del tamaño por defecto. Por defecto, una esquina está situada a 1 metro de cada uno de los ejes desde el origen, resultando un cubo que está centrado en el origen y tiene 2 metros de alto, de ancho y de profundo.  ColorCube(double scale)  Construye un cubo de color escalado por el valor especificado. El tamaño por defecto es 2 metros de lado. El **ColorCube** resultante tiene esquinas en (scale, scale, scale) y (-scale, -scale, -scale). |   **Rotar el Cubo**  Una simple rotación del cubo puede hacer que se vea más de una de sus caras. El primer paso es crear la transformación deseada usando un objeto **Transform3D**.  El Fragmento de Código 1-5 incorpora un objeto **TransformGroup** en el escenario gráfico para rotar el cubo sobre el eje x. Primero se crea la transformación de rotación usando el objeto **rotate** de **Transform3D**. Este objeto se crea en la línea 6. La rotación se especifica usando el método **rotX()** de la línea 8. Entonces se crea el objeto **TransformGroup** en la línea 10 para contener la transformación de rotación.  Dos parámetros especifican la rotación: el eje de revolución, y el ángulo de rotación. El eje se elige seleccionando el método apropiado. El ángulo de rotación es el valor que se le pasa como argumento. Como el ángulo de rotación se especifica en radianes, el valor **PI/4** es **1/8** de una rotación completa, o 45 grados.  Después de crear el objeto Transform3D, **rotate**, se usa en la creacción del objeto TransformGroup **objRotate** (línea 10). El objeto Transform3D se usa en el escenario gráfico. Entonces el objeto **objRotate** hace que **ColorCube** sea su hijo (línea 11). A su vez, el objeto **objRoot** hace a **objRotate** como su hijo (línea 12).  **Fragmento de código 1-5. Una Rotación en la Rama de Contenido Gráfico**  1. public BranchGroup createSceneGraph() {  2. // Create the root of the branch graph  3. BranchGroup objRoot = new BranchGroup();  4.  5. // rotate object has composite transformation matrix  6. Transform3D rotate = new Transform3D();  7.  8. rotate.rotX(Math.PI/4.0d);  9.  10. TransformGroup objRotate = new TransformGroup(rotate);  11. objRotate.addChild(new ColorCube(0.4));  12. objRoot.addChild(objRotate);  13. return objRoot;  14. } // end of createSceneGraph method  La rama de contenido gráfico ahora incluye un objeto **TransformGroup** en el camino del escenario gráfico hacia el objeto **ColorCube**.Cada uno de los caminos del escenario gráfico es necesario. El objeto **BranchGroup** es el único que puede ser hijo de un **Locale**. El objeto **TransformGroup** es el único que puede cambiar la localización, la orientación, o el tamaño de un objeto visual. En este caso el objeto **TransformGroup** cambia la orientación. Por supuesto, el objeto **ColorCube** es necesario para suministrar el objeto visual.  Aquí podemos ver la imagen producida por el Fragmento de Código 1-5.  Figura 1-13 Imagen creada con el Fragmento de Código 1-5  **Ejemplo de Combinación de Transformaciones: HelloJava3Db**  Frecuentemente un objeto visual se traslada y se rota, o se rota sobre dos ejes. En cualquier caso, se especifican dos transformaciones diferentes para un sólo objeto visual. Las dos transformaciones pueden combinarse en una matriz de transformaciones y contenerse en un sólo objeto **TransformGroup**. Podemos ver un ejemplo en el Fragmento de Código 1-6.  En el programa [HelloJava3Db.java](http://www.labvis.unam.mx/elio/J3D/clases/HelloJava3D/HelloJava3Db.java) se combinan dos rotaciones. Crear estas dos rotaciones simultáneas requiere combinar dos objetos **Transform3D** de rotación. El ejemplo rota el cubo sobre los ejes x e y. Se crean dos objetos **Transform3D**, uno por cada rotación (líneas 6 y 7). Las rotaciones individuales se especifican para los dos objetos **TransformGroup** (líneas 9 y 10). Luego las rotaciones se combinan mediante la multiplicación de los objetos **Transform3D** (línea 11). La combinación de las dos transformaciones se carga en el objeto **TransformGroup** (línea 12).  **Fragmento de código 1-6. Dos Transformaciones de Rotación en HelloJava3Db**  1. public BranchGroup createSceneGraph() {  2. // Create the root of the branch graph  3. BranchGroup objRoot = new BranchGroup();  4.  5. // rotate object has composite transformation matrix  6. Transform3D rotate = new Transform3D();  7. Transform3D tempRotate = new Transform3D();  8.  9. rotate.rotX(Math.PI/4.0d);  10. tempRotate.rotY(Math.PI/5.0d);  11. rotate.mul(tempRotate);  12. TransformGroup objRotate = new TransformGroup(rotate);  13.  14. objRotate.addChild(new ColorCube(0.4));  15. objRoot.addChild(objRotate);  16. return objRoot;  Tanto el Fragmento de Código 1-5 como el Fragmento de código 1-6 podrían reemplazar al Fragmento de Código 1-2. El Fragmento de código 1-6 se usa en **HelloJava3Db.java**. Aquí puedes encontrar el ejemplo completo: [HelloJava3Db.java](http://www.labvis.unam.mx/elio/J3D/clases/HelloJava3D/HelloJava3Db.java)  En la Figura 1.-14 podemos ver el escenario gráfico creado en **HelloJava3Db.java**. La rama de vista gráfica es la misma producida en **HelloJava3Da**, que está construida por un **SimpleUniverse** y representada por una gran estrella. La rama de contenido gráfico ahora incluye un **TransformGroup** en el camino del escenario gráfico hacia el objeto **ColorCube**.  Figura 1-14 Escenario Gráfico del Ejemplo HelloJava3Db  La imagen de la figura 1-15 muestra el **ColorCube** girado del **HelloJava3Db**.  Figura 1-15 Imagen del ColorCube rotada por el programa HelloJava3D  **Capacidades y Rendimiento**  El escenario gráfico construido por un programa Java 3D podría usarse directamente para renderizar. Sin embargo, la representación no es muy eficiente. La flexibilidad construida dentro de cada objeto escenario gráfico (que no se van a discutir en este tutorial) crean un representación sub-optima del universo virtual. Para mejorar el rendimiento de la renderización se usa una representación más eficiente del universo virtual.  Java 3D tiene una representación interna para una universo virtual y los métodos para hacer la conversión. Hay dos formas para hacer que el sistema Java 3D haga la conversión de la representación interna. Una forma es compilar todas las ramas gráficas. La otra forma es insertar una rama gráfica en un universo virtual para darle vida.  **Compilar Contenidos**  El objeto **BranchGroup** tiene un método compilador. Llamando a este método se convierte la rama gráfica completa que hay debajo del **BranchGroup** a la representación interna de Java 3D de la rama gráfica. Además de la conversión, la representación interna podría optimizarse de una o varias maneras.  Las posibles optimizaciones no se especifican en el API Java 3D. Sin embargo, se puede ganar en eficiencia de varias formas. Una de las posibles optimizaciones es combinar **TransformGroups** con caminos de escenario gráfico. Por ejemplo, si un escenario gráfico tiene dos objetos **TransformGroup** en una relacion padre-hijo pueden ser representados por un objeto **TransformGroup**. Otra posibilidad es combinar objetos **Shape3D** que tienen una relación estática física. Estos tipos de optimizaciones se hacen posibles cuando las capacidades no se configuran.  La Figura 1-16 presenta una representación conceptual de la conversión a una representación más eficiente. El escenario gráfico del lado izquierdo es compilado y transformado en la representación interna mostrada en el lado derecho. La figura sólo representa el concepto de representación interna, no como Java 3D realmente lo hace.  Figura 1-16 Ejemplo Conceptual del Resultado de Compilar una Escenario Gráfico  **Capacidades**  Una vez que una rama gráfica empieza a vivir o es compilada el sistema de renderizado Java 3D la convierte a una representación interna más eficiente. El efecto más importante de esta conversión es la mejora del rendimiento de renderizado.  Pero también tiene otros efectos, uno de ellos es fijar el valor de transformaciones y otros objetos en el escenario gráfico. A menos que especificamente lo proporcionemos en el programa, este no tendrá la capacidad de cambiar los valores de los objetos del escenario gráfico una vez que estén vivos.  Hay casos en que un programa necesita la capacidad de cambiar estos valores después de que estén vivos. Por ejemplo, cambiar el valor de un objeto **TransformGroup** crea animaciones. Para que esto suceda, la transforamción debe poder cambiar después de estar viva. La lista de parámetros a los que se puede acceder, y de que forma, se llama capacidades del objeto.  Cada **SceneGraphObject** tiene un conjunto de bits de capacidad. Los valores de estos bits determinan que capacidades existen para el objeto después de compilarlo o de darle vida. El conjunto de capacidades varía con la clase.   |  | | --- | | Lista Parcial de Métodos de **SceneGraphObject**  **SceneGraphObject** es la superclase de casi cualquier clase usada para crear un escenario gráfico, incluyendo **Group**, **Leaf**, y **NodeComponent**.  void clearCapability(int bit)  Borra el bit de capacidad especificado.  boolean getCapability(int bit)  Recupera el bit de capcidad especificado.  void setCapability(int bit)  Configura el bit de capacidad especificado. |   Como ejemplo, para poder leer el valor de la transformación representada por un objeto **TransformGroup**, esta capacidad debe activarse antes de compilarlo o darle vida. De forma similar, para poder cambiar el valor de la transformación en un objeto **TransformGroup**, su capacidad de escribir transformación debe configurarse antes de compilarlo o darle vida. Intentar hacer un cambio en un objeto vivo o compilado para el que la propiedad adecuada no se ha configurado resultará en una excepción.  En la siguiente sección, las animaciones se crean usando una transformación de rotación que varía con el tiempo. Para que esto sea posible, el objeto **TransformGroup** debe tener su capacidad **ALLOW\_TRANSFORM\_WRITE** activada antes de que sea compilado o se le de vida.   |  | | --- | | Lista Parial de Capcidades de **TransformGroup**  Las dos capacidades listadas aquí son las únicas definidas por **TransformGroup**. Éste hereda varias capacidades de sus clases ancestros: **Group** y **Node**. La configuración de capacidades se puede seleccionar, eliminar o recuperar usando los métodos definidos en **SceneGraphObject**.  ALLOW\_TRANSFORM\_READ  Especifa que el nodo **TransformGroup** permite acceder a la información de transformación de su objeto.  ALLOW\_TRANSFORM\_WRITE  Especifica que el nodo **TransformGroup** permite escribir la información de transformación de su objeto. |   Las capacidades también controlan el acceso a otros aspectos de un objeto **TransformGroup**. Los objetos **TransformGroup** heredan configuración de capacidades de sus clases ancestros: **Group** y **Node**. En el siguiente bloque de referencia podemos ver algunas de esas capacidades.   |  | | --- | | Lista Parcial de Capacidades de **Group**  **TransformGroup** hereda varios bits de capacidades de sus clases ancestros.  ALLOW\_CHILDREN\_EXTEND  Permite que se puedan añadir hijos al nodo **Group** después de que esté compilado o vivo.  ALLOW\_CHILDREN\_READ  Permite que se puedan leer las referencias a los hijos del nodo **Group** después de que esté compilado o vivo.  ALLOW\_CHILDREN\_WRITE  Permite que se puedan escribir las referencias a los hijos del nodo **Group** después de que esté compilado o vivo. |   **Añadir Comportamiento de Animación**  En Java 3D, **Behavior** es una clase para especificar animaciones o interacciones con objetos visuales. El comportamiento puede cambiar virtualmente cualquier atributo de un objeto visual. Un programador puede usar varios comportamientos predefinidos o especificar un comportamiento personalizado. Una vez que se ha especificado un comportamiento para un objeto visual, el sistema Java 3D actualiza automáticamente la posición, la orientación, el color, u otros atributos del objeto visual.  La distinción entre animación e interacción es si el comportamiento es activado en respuesta al paso del tiempo o en respuesta a actividades del usuario, respectivamente.  Cada objeto visual del universo virtual puede tener sus propio comportamiento predefinido. De echo, un objeto visual puede tener varios comportamientos. Para especificar un comportamiento para un objeto visual, el programador crea objetos que especifiquen el comportamiento, añade el objeto visual al escenario gráfico y hace las referencias apropiadas entre los objetos del escenario gráfico y los objetos **Behavior**.  En un universo virtual con muchos comportamientos, se necesita una significante potencia de cálculo para calcular los comportamientos. Como tanto el renderizador como el comportamiento usan el mismo procesador, es posible que la potencia de cálculo que necesita el comportamiento degrade el rendimiento del renderizado.  Java 3D permite al programador manejar este problema especificando un límite espacial para que el comportamiento tenga lugar. Este límite se llama región programada. Un comportamiento no está activo a menos que el volumen de activación de **ViewPlatform** intereseccione con una región progamada del **Behavior**. En otras palabras, si nadie en el bosque ve el árbol caer, éste no cae. La característica de región programada hace más eficiente a Java 3D en el manejo de universos virtuales con muchos comportamientos.  Un **Interpolator** es uno de las muchas clases de comportamientos predefinidos en el paquete corazón de Java 3D. Basado en una función de tiempo, el objeto **Interpolator** manipula los parámetros de un objeto del escenario gráfico. Por ejemplo, para el **RotationInterpolator**, manipula la rotación especificada por un **TransformGroup** para afectar la rotación de los objetos visuales que son ancestros de **TransformGroup**.  La siguiente lista enumera los pasos envueltos para especificar una animación con un objeto **interpolator**. Los cinco pasos forman una receta para crear un comportamiento de animación con interpolación:   1. Crear un **TransformGroup** fuente.  Selecciona la capacidad **ALLOW\_TRANSFORM\_WRITE**. 2. Crear un objeto **Alpha** (función de tiempo en Java 3D)  Especifica los parámetros de tiempo para el **alpha** 3. Crear el objeto **interpolator**.  Tiene referencias con los objetos **Alpha** y **TransformGroup**.  Personalizar los parámetros del comportamiento. 4. Especificar la región programada.  Configura la región programada para el comportamiento. 5. Hacer el comportamiento como hijo del **TransformGroup**   **Especificar un Comportamiento de Animación**  Una acción de comportamiento puede ser cambiar la localización (PositionInterpolator), la orientación (RotationInterpolator), el tamaño (ScaleInterpolator), el color (ColorInterpolator), o la transpariencia (TransparencyInterpolator) de un objeto visual. Como se mencionó antes, los **Interpolators** son clases de comportamiento predefinidas. Todos los comportamientos mencionados son posibles sin usar un **interpolator**; sin embargo, los **interpolators** hacen mucho más sencilla la cracción de comportamientos. Las clases **Interpolators** existen para proporcionar otras acciones, incluyendo combinaciones de estas acciones.  **Clase RotationInterpolator**  Esta clase se usa para especificar un comportamiento de rotación de un objeto visual o de un grupo de objetos visuales. Un objeto **RotationIterpolator** cambia un objeto **TransformGroup** a una rotación especififca en repuesta a un valor de un objeto **Alpha**. Como el valor de este objeto cambia cada vez, la rotación también cambia. Un objeto **RotationInterpolator** es flexible en la especificación del eje de rotación, el ángulo de inicio y el ángulo final.  Para rotaciones constantes sencillas, el objeto **RotationInterpolator** tiene el siguiente constructor que puede usarse para eso:   |  | | --- | | Lista Parcial de Constructores de **RotationInterpolator**  Esta clase define un comportamiento que modifica el componente rotacional de su **TransformGroup** fuente linearizando la interpoalización entre un par de ángulos especificados (usando el valor generado por el objeto **Alpha** especificado). El ángulo interpolado se usa para generar una transformación de rotación.  RotationInterpolator(Alpha alpha, TransformGroup target)  Este constructor usa valores por defecto de algunos parámetros del interpolador para construir una rotación completa sobre el eje y, usando el **TransformGroup** especificado.  Parámetros:   * alpha - la función de variación de tiempo para referencia. * target - el objeto **TransformGroup** a modificar. |   El objeto **TransformGroup** de un interpolador debe tener la capacidad de escritura activada.  **Funciones de Variación de Tiempo: Mapear un Comportamiento en el Tiempo**  Mapear una acción en el tiempo se hace usando un objeto **Alpha**. La especificación de este objeto puede ser compleja.  **Clase Alpha**  Los objetos de la clase **Alpha** se usan para crear una función que varía en el tiempo. La clase **Alpha** produce un valor entre cero y uno, inclusives. El valor que produce depende de la hora y de los parámetros del objeto **Alpha**. Los objetos **Alpha** se usan comunmente con un comportamiento **Interpolator** para proporcionar animaciones de objetos visuales.  **Alpha** tiene diez parámetos, haciendo la programación tremendamente flexible. Sin entrar en detalles de cada parámetros, saber que un ejemplar de **Alpha** puede combinarse fácilmente con un comportamiento para proporcionar rotaciones sencillas, movimientos de péndulo, y eventos de una vez, como la apertura de puertas o el lanzamiento de cohetes.   |  | | --- | | Constructor de **Alpha**  La clase **Alpha** proporciona objetos para convertir la hora en un valor alpha (un valor entre 0 y 1). El objeto **Alpha** es efectivamente una función de tiempo que genera valores alpha entre cero y uno. La función "f(t)" y las características del objeto **Alpha** están determinadas por parámetros definidos por el usuario:  Alpha()  Bucle continuo con un periodo de un segundo.  Alpha(int loopCount, long increasingAlphaDuration)  Este constructor toma sólo **loopCount** e **increasingAlphaDuration** como parámetros y asigna los valores por derecto a todos los demás parámetros, resultando un objeto **Alpha** que produce valores desde cero a uno crecientes. Esto se repite el número de veces especificado por **loopCount**. Si **loopCount** es -1, el objeto alpha se repite indefinidamente. El tiempo que tarde en ir desde cero hasta uno está especificando en el segundo parámetro usando una escala de milisegundos.  Parámetros:   * loopCount - número de veces que se ejecuta este objeto alpha; un valor de -1 especifica un bucle indefinido.. * increasingAlphaDuration - tiempo en milisegundos que tarda el objeto alpha en ir de cero a uno. |   **Región Progamada**  Como se mencionó anteriormente, cada comportamiento tiene unos límites programados. Estos límites se configuran usando el método **setSchedulingBounds** de la clase **Behavior**.  Hay varias formas de especificar una región programada, la más sencilla es crear un objeto **BoundingSphere**. Otras opciones incluyen **BoundingBox** y **BoundingPolytope**.   |  | | --- | | Método **setSchedulingBounds** de Behavior  void setSchedulingBounds(Bounds region)  Selecciona la región programada del **Behavior** a unos límites especificados.  Parámetros:   * region - Los límites que contienen la región programada del **Behavior**. |   **Clase BoundingSphere**  Especificar un límite esférico se consigue especificando un punto central y un rádio para la esfera. El uso normal de este tipo de límites es usar el centro a (0, 0, 0). Entonces el radio se selecciona lo suficientemente grande como para contener el objeto visual, incluyendo todas las posibles localizaciones del objeto.   |  | | --- | | Lista Parcial de Constructores de **BoundingSphere**  Esta clase define una región de límites esférica que está definida por un punto central y un rádio.  BoundingSphere()  Este constructor crea una límite esférico centrado en el origen (0, 0, 0) con un radio de 1.  BoundingSphere(Point3d center, double radius)  Construye e inicializa un **BoundingSphere** usando el punto central y el rádio especificados. |   **Ejemplo de Comportamiento: HelloJava3Dc**  El Fragmento de Código 1-7 muestra un ejemplo completo del uso de las clases interpoladoras para crear una animación. La animación creada con este código es una rotación continúa con un tiempo de rotación total de 4 segundos.  El paso 1 de la receta es crear el objeto **TransformGroup** para modificarlo durante la ejecución. El objeto **TransformGroup** fuente de un interpolador debe tener activada la capacidad de escritura. El objeto **TransformGroup** llamado **objSpin** se crea en la línea 7. La capacidad de escritiura de **objSpin** se selecciona en la línea 8.  El paso 2 es crear un objeto alpha para dirigir la interpolación. Los dos parámetros especificados en la línea 16 del fragmento de código son el número de interacciones del bucle y el tiempo de un ciclo. El valor de "-1" especifica un bucle contínuo. El tiempo se especifica en milisegundos por lo que el valor de 4000 significa 4 segundos. Por lo tanto, el comportamiento es rotar cada cuatro segundos.  El paso 3 de la receta es crear el objeto **interpolator**. El objeto **RotationInterpolator** se crea en las líneas 21 y 22. El interpolador debe tener referencias a la transformación fuente y al objeto alpha. Esto se consigue en el constructor. En este ejemplo se usa el comportamiento por defecto del **RotationInterpolator** para hacer una rotación completa sobre el eje y.  El paso 4 es especificar una región programada. Se usa un objeto **BoundingSphere** con sus valores por defecto. El objeto **BoundingSphere** se crea en la líena 25. La esfera se configura como los límites del comportamiento en la línea 26.  El paso final, el 5, hace del comportamiento un hijo del **TransformGroup**. Esto se consigue en la línea 27.  **Fragmento de código 1-7. Método createSceneGraph con Comportamiento RotationInterpolator**  1. public BranchGroup createSceneGraph() {  2. // Create the root of the branch graph  3. BranchGroup objRoot = new BranchGroup();  4.  5. // Create the transform group node and initialize it to the  6. // identity. Add it to the root of the subgraph.  7. TransformGroup objSpin = new TransformGroup();  8. objSpin.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);  9. objRoot.addChild(objSpin);  10.  11. // Create a simple shape leaf node, add it to the scene graph.  12. // ColorCube is a Convenience Utility class  13. objSpin.addChild(new ColorCube(0.4));  14.  15. // create time varying function to drive the animation  16. Alpha rotationAlpha = new Alpha(-1, 4000);  17.  18. // Create a new Behavior object that performs the desired  19. // operation on the specified transform object and add it into  20. // the scene graph.  21. RotationInterpolator rotator =  22. new RotationInterpolator(rotationAlpha, objSpin);  23.  24. // a bounding sphere specifies a region a behavior is active  25. BoundingSphere bounds = new BoundingSphere();  26. rotator.setSchedulingBounds(bounds);  27. objSpin.addChild(rotator);  28.  29. return objRoot;  30. } // end of createSceneGraph method  Este fragmento de código se usa con otros fragmentos anteriores para crear el programa de ejemplo [HelloJava3Dc.java](http://www.labvis.unam.mx/elio/J3D/clases/HelloJava3D/HelloJava3Dc.java). Al ejecutar la aplicación veremos como se renderiza el **ColorCube** con un comportamiento de rotación cada cuatro segundos.  El programa **HelloJava3Dc** crea el escenario gráfico de la Figura 1-18. El objeto **rotation** es tanto hijo del **TransformGroup** como una referencia a él. Aunque esta relación parece violar la restricciones de bucles dentro del escenaio gráfico, no lo hace. Recuerda que los arcos de referencia (flecha punteada) no son parte del escenario gráfico. La línea punteada desde el **Behavior** hacia el **TransformGroup** es esta referencia.  Figura 1-18 Escenario Gráfico del Ejemplo HelloJava3Dc  La imagen de la Figura 1-19 muestra un marco de la ejecución del programa HelloJava3Dc.  Figura 1-19 Una imagen del ColorCube en Movimiento.  **Ejemplo de Combinación de Transformation y Behavior: HelloJava3Dd**  Por supuesto, podemos combinar comportamientos con las transformaciones de rotación de los ejemplos anteriores. [HelloJava3Dd.java](http://www.labvis.unam.mx/elio/J3D/clases/HelloJava3D/HelloJava3Dd.java) hace esto. En la rama de contenido gráfico, hay objetos llamados **objRotate** y **objSpin**, que distinguen entre la rotación estática y el comportamiento de rotación (bucle continuo) del objeto cube respectivamente. El escenario resultante de este fragmento de código podemos verlo en la figura 1.20.  **Fragmento de código 1-8. Rama de Contenido Gráfico para un ColorCube giratorio**  1. public BranchGroup createSceneGraph() {  2. // Create the root of the branch graph  3. BranchGroup objRoot = new BranchGroup();  4.  5. // rotate object has composite transformation matrix  6. Transform3D rotate = new Transform3D();  7. Transform3D tempRotate = new Transform3D();  8.  9. rotate.rotX(Math.PI/4.0d);  10. tempRotate.rotY(Math.PI/5.0d);  11. rotate.mul(tempRotate);  12.  13. TransformGroup objRotate = new TransformGroup(rotate);  14.  15. // Create the transform group node and initialize it to the  16. // identity. Enable the TRANSFORM\_WRITE capability so that  17. // our behavior code can modify it at runtime. Add it to the  18. // root of the subgraph.  19. TransformGroup objSpin = new TransformGroup();  20. objSpin.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);  21.  22. objRoot.addChild(objRotate);  23. objRotate.addChild(objSpin);  24.  25. // Create a simple shape leaf node, add it to the scene graph.  26. // ColorCube is a Convenience Utility class  27. objSpin.addChild(new ColorCube(0.4));  28.  29. // Create a new Behavior object that performs the desired  30. // operation on the specified transform object and add it into  31. // the scene graph.  32. Transform3D yAxis = new Transform3D();  33. Alpha rotationAlpha = new Alpha(-1, 4000);  34.  35. RotationInterpolator rotator =  36. new RotationInterpolator(rotationAlpha, objSpin, yAxis,  37. 0.0f, (float) Math.PI\*2.0f);  38.  39. // a bounding sphere specifies a region a behavior is active  40. // create a sphere centered at the origin with radius of 1  41. BoundingSphere bounds = new BoundingSphere();  42. rotator.setSchedulingBounds(bounds);  43. objSpin.addChild(rotator);  44.  45. return objRoot;  46. } // end of createSceneGraph method of HelloJava3Dd  Figura 1-20 Escenario Gráfico del Ejemplo HelloJava3Dd  La imagen de la Figura 1-21 muestra un marco del ColorCube en movimiento del programa **HelloJava3Dd**.  Figura 1-21 Imagen del ColorCube en movimiento |  |  | |

[Siguiente](http://www.labvis.unam.mx/elio/J3D/pagina33.html) [Anterior](http://www.labvis.unam.mx/elio/J3D/pagina31.html) [Inicio](http://www.labvis.unam.mx/elio/J3D/tutorialJ3D.html)