**Java 3D**

[Interacción en Java 3D](about:blank)

* + [Comportamiento: la Base para Interacción y Animación](about:blank)
    - [Aplicaciones de Behavior](about:blank)
    - [Introducción a la clases Behavior](about:blank)
  + [Behavior Básico](about:blank)
    - [Escribir una Clase Behavior](about:blank)
    - [Usar una Clase Behavior](about:blank)
    - [API de la Clase Behavior](about:blank)
  + [Condiciones de Disparo: Cómo se Disparan los Comportamientos](about:blank)
    - [WakeupCondition](about:blank)
    - [WakeupCriterion](about:blank)
    - [Clases WakeupCriterion Específicas](about:blank)
    - [WakeupCondition Composition](about:blank)
  + [Clases de Comportamientos Útiles para la Navegación por Teclado](about:blank)
    - [Programa de Ejemplo de KeyNavigatorBehavior](about:blank)
    - [Clases KeyNavigatorBehavior y KeyNavigator](about:blank)
  + [Clases de Utilidad para Interactuar con el Ratón](about:blank)
    - [Usar las Clases de Comportamiento del Ratón](about:blank)
    - [Fundamentos del Comportamiento del Ratón](about:blank)
    - [Clases Específicas de Comportamientos de Ratón](about:blank)
    - [MouseNavigation](about:blank)
  + [Picking](about:blank)
    - [Usar las Clases de Utilidad de Picking](about:blank)
    - [El API Corazón de Clases Picking de Java 3D](about:blank)
    - [Clases Generales del Paquete Picking](about:blank)
    - [Clases de Comportamientos Picking Específicas](about:blank)

**Interacción en Java 3D**

**Comportamiento: la Base para Interacción y Animación**

La interacción y la animación se especifican con objetos **Behavior**. La clase **Behavior** es una subclase abstracta que proporciona el mecanismo para incluir código que modifique el escenario gráfico. La clase **Behavior**, y sus descendientes, son enlaces a código del usuario que proporciona las modificaciones para los gráficos y los sonidos del universo virtual.

El propósito del objeto **Behavior** en un escenario gráfico es modificar el propio escenario gráfico, o los objetos que hay dentro de él, en respuesta a algunos estímulos. Un estímulo puede ser una pulsación de tecla, un movimiento del ratón, la colisión de objetos, el paso del tiempo, algún otro evento, o una combinación de estos. Los cambios producidos incluyen la adicción de objetos al escenario gráfico, la eliminación de objetos, cambio de atributos de los objetos del escenario gráfico, reordenación de los objetos del escenario gráfico, o una combinación de estos. Las posibilidades sólo están limitadas por las capacidades de los objetos del escenarios gráfico.

**Aplicaciones de Behavior**

Como un comportamiento (Behavior) es un enlace entre un estímulo y una acción, si consideramos todas las combinaciones posibles entre estímulos y acciones podremos obtener todas las aplicaciones de los objetos **Behavior**. La siguiente tabla muestra algunas de las posibilidades de **Behavior**, listando los posibles estímulos hacia abajo, y los posibles cambios hacia la derecha.

La tabla no lista todas las combinaciones posibles, sólo las más simples (un estímulo resulta en un cambio). Algunas combinaciones de estímulos y cambios sólo tienen sentido en un entorno específico; estas se listan como "específicas de la aplicación".

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Estímulo** | **Objeto** | **del** | **cambio** |  |
| **(razón para el cambio)** | **TransformGroup** (los objetos visuales cambian la orientación o la localización) | **Geometry** (los objetos visuales cambian la forma o el color) | **Scene Graph** (añadir, eliminar o intercambiar objetos) | **View** (cambiar la localización o dirección de la vista) |
| usuario | interacción | específico de la aplicación | específico de la aplicación | navegación |
| colisiones | Los objetos visuales cambian su orientación o posición | Los objetos visuales cambian su apariencia con la colisión | Los objetos visuales desaparecen con la colisión | La vista cambia con la colisión |
| tiempo | animación | animación | animación | animación |
| Posición de la Vista | cartelera | nivel de detalles (LOD) | específico de la aplicación | específico de la aplicación |

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

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

El comportamiento de nivel de detalle (LOD) tiene una aplicación relacionada. Con LOD, los objetos visualmente complejos son representados por múltiples objetos visuales variando los niveles de detales (de ahí su nombre). La representación del objeto visual con menor nivel de detalle se usa cuando el espectador está lejos. La representación con más nivel de detalle se usa cuando el espectador está muy cerca. El comportamiento LOD cambia automáticamente entre las representación del objeto basándose en la distancia al espectador.

Los comportamientos de cartelera y de nivel de detalle corresponden a clases extendidas desde **Behavior** que implementan estas aplicaciones comunes. Son posibles otros comportamientos especilizados y varios de ellos se pueden ver en la Figura 4-1. Por ejemplo, hay varias clases **MouseBehavior** que manipulan una transformación en respuesta a movimientos del ratón. Normalmente la transformación de la vista se cambia por el comportamiento del ratón para cambiar la vista en respuesta a una acción del ratón.

Observa también como los comportamientos pueden encadenarse. Por ejemplo, los movimientos del ratón o las pulsaciones de teclas pueden usarse para cambiar la vista. En respuesta al movimiento de la vista, podrían tener lugar otros comportamientos como la cartelera, o el nivel de detalles. Afortunadamente, cada comportamiento se especifica de forma separada.

**Animación contra Interacción**

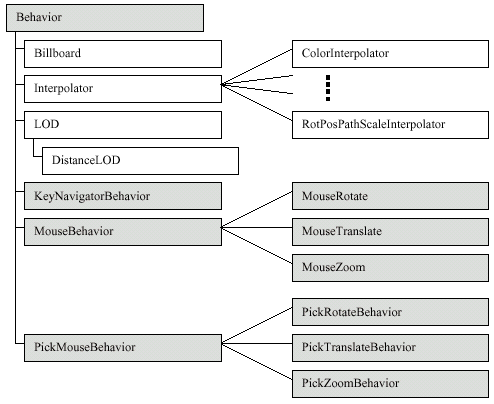
Como la distinción entre animación e interacción usada en este tutorial está bastante bien, aquí hay un ejemplo para clarificar esta distinción. Si un usuario navega en un programa donde se proporciona un comportamiento, la vista se moverá en respuesta a eventos del teclado y/o ratón. El movimiento de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

**Introducción a la clases Behavior**

La Figura 4-1 muesta especializaciones de la clase **Behavior** creadas en el API de Java 3D. También son posibles las clases especializadas de **Behavior** definidas por el usuarios y están sólo limitadas por la imaginación del programador. Este módulo del tutorial cubre cada una de las clases de la Figura 4-1. Este capítulo cubre las clases sombreadas, el siguiente capítulo cubre el resto.

**Behavior Básico**

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



**Escribir una Clase Behavior**

Esta sección explica cómo escribir una clase de comportamiento personalizado. Ya sabemos que hay clases de comportamiento que podemos usar. Sin embargo, al ver cómo crear una clase **Behavior** aprenderemos como funciona. Por eso, incluso si no planeas usar una clase comportamiento, podrías querer leer esta sección. Las clases escritas en esta sección se usan en la siguiente.

**Mecánismo de Behaviors**

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

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

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

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

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

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

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

**Ejemplo de Clase Behavior Personalizada: SimpleBehavior**

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

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

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

Cómo sólo hay una posible condición de disparo, el método processStimulus no la descodifica. Es posible posteriormente descodificar el evento de pulsación de tecla para determinar qué tecla, o combinación de teclas, se pulsó.

El método processStimulus siempre incrementa la variable del ángulo, entonces lo usa para ajustar el objeto **TransformGroup**. El último trabajo de este método es resetear el disparo. En este ejemplo, el disparo siempre se resetea a una pulsación de tecla. Los comportamientos pueden cambiar el evento de disparo en el tiempo para cambiar comportamientos (otra razón para tener que descodificar el evento de disparo), o no seleccionar otro disparo para comportamientos de una sóla vez.

El [Fragmento de Código 4-1](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento4-1) presenta la clase **SimpleBehavior** ([SimpleBehaviorApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/interaction/SimpleBehaviorApp.java)) que es una implementación de la clase de comportamiento personalizada. Las sentencias import son necesarias para esta clase. java.awt.event es necesaria para la interacción con el teclado. java.util.eumeration es necesaria para decodificar el **WakeupCondition**; y por lo tanto necesaria virtualmente para casi cualquier clase de comportamiento personalizado. También son necesarias las sentencias import normales del API Java 3D.

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

1. import java.awt.event.\*;

2. import java.util.Enumeration;

3.

4. // SimpleBehaviorApp renders a single, rotated cube.

5.

6. public class SimpleBehaviorApp extends Applet {

7.

8. public class SimpleBehavior extends Behavior{

9.

10. private TransformGroup targetTG;

11. private Transform3D rotation = new Transform3D();

12. private double angle = 0.0;

13.

14. // create SimpleBehavior - set TG object of change

15. SimpleBehavior(TransformGroup targetTG){

16. this.targetTG = targetTG;

17. }

18.

19. // initialize the Behavior

20. // set initial wakeup condition

21. // called when behavior becomes live

22. public void initialize(){

23. // set initial wakeup condition

24. this.wakeupOn(new WakeupOnAWTEvent(KeyEvent.KEY\_PRESSED));

25. }

26.

27. // called by Java 3D when appropriate stimulus occurs

28. public void processStimulus(Enumeration criteria){

29. // do what is necessary in response to stimulus

30. angle += 0.1;

31. rotation.rotY(angle);

32. targetTG.setTransform(rotation);

33. this.wakeupOn(new WakeupOnAWTEvent(KeyEvent.KEY\_PRESSED));

34. }

35.

36. } // end of class SimpleBehavior

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

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

**Riesgos de Programación al Escribir Clases Behavior**

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

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

Obviamente, si no se selecciona el disparo inicial en el método initialization(), el comportamiento nunca será invocado. Un poco menos obvio es que el disparo debe seleccionarse de nuevo en el método processStimulus() si se desea un comportamiento repetido.

Como estos dos métodos (initialization() y processStimulus()) son llamados por el sistema Java 3D, deben volver para permitir que continúe el renderizado. Por ejemplo, si se desea un ejemplo peonza, el ángulo y el **TransformGroup** necesitan actualizarse periódicamente. Si nuestro comportamiento implementa este comportamiento sin deshilar un thread, no se renderizará nada más. También, hay una forma mucho mejor para conseguir este tipo de comportamiento.

**Usar una Clase Behavior**

Encontrar o escribir la clase **behavior** apropiada para nuestra aplicación es el principio para escribir un programa Java 3D interactivo. Esta sección cubre los problemas de programación en la adicción de objetos **behavior** a los programas.

El primer paso implica el asegurarnos de que el escenario gráfico hace provisiones para el **behavior**. Por ejemplo, para usar la clase **SimpleBehavior** de la sección anterior debe haber un **TransformGroup** en el escenario gráfico sobre el objeto a rotar. Muchos comportamientos sólo necesitan un único objeto **TransformGroup**; sin embargo, los requerimientos de un escenario gráfico para una comportamiento dependen de la aplicación y del propio comportamiento y podrían ser más complejos.

Habiendo establecido el soporte para un comportamiento, se debe añadir un ejemplar de la clase al escenario gráfico. Sin ser una parte de un escenario gráfico vivo, no hay forma de poder inicializar un comportamiento. De hecho, un objeto **behavior** que no es parte de un escenario gráfico se convertirá en basura y será eliminado en la próxima recolección.

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

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

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

El [Fragmento de Código 4-2](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento4-2) es un extracto del programa de ejemplo [SimpleBehaviorApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/interaction/SimpleBehaviorApp.java) y es la continuación del [Fragmento de Código 4-1](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento4-1)

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

37. public BranchGroup createSceneGraph() {

38. // Create the root of the branch graph

39. BranchGroup objRoot = new BranchGroup();

40.

41. TransformGroup objRotate = new TransformGroup();

42. objRotate.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);

43.

44. objRoot.addChild(objRotate);

45. objRotate.addChild(new ColorCube(0.4));

46.

47. SimpleBehavior myRotationBehavior = new SimpleBehavior(objRotate);

48. myRotationBehavior.setSchedulingBounds(new BoundingSphere());

49. objRoot.addChild(myRotationBehavior);

50.

51. // Let Java 3D perform optimizations on this scene graph.

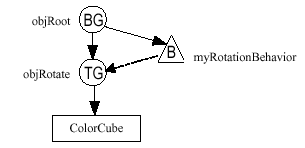
52. objRoot.compile();

53.

54. return objRoot;

55. } // end of CreateSceneGraph method of SimpleBehaviorApp

Se necesita muy poco código para completar el programa de los fragmentos de código 4-1 y 4-2. El programa completo está en: [SimpleBehaviorApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/interaction/SimpleBehaviorApp.java). La aplicación completa renderiza un objeto **ColorCube** en una escena estática hasta que se pulsa una tecla. En respuesta a la pulsación de la tecla, el **ColorCube** rota 0,1 radianes (unos 6°). La Figura 4-4 muestra el diagrama del escenario gráfico para la rama de contenido gráfico de esta aplicación.



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

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

**Riesgos de Programación al usar Objetos Behavior**

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

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

La intersección de los límites de un **behavior** con el volumen de activación de una vista determina si el evento Java 3D considera el disparo del estímulo para el **behavior**. Java 3D no avisará si no ponemos los límites -- el comportamiento nunca se disparará. También debemos mantener los límites de cada objeto **behavior** tan pequeños como sea posible para una mejora global del rendimiento.

Como se mencionó arriba, un objeto **behavior** que no forma parte de un escenario gráfico será considerado basura y será eliminado en el siguiente ciclo del recolector de basura. Esto, también sucederá sin errores ni avisos.

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

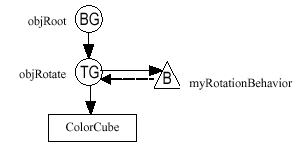
Los comportamientos pueden situarse en cualquier lugar del escenario gráfico. Los problemas para esta localización son: 1) el efecto de los límites, y 2) el mantenimiento del código.

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

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

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

Hay dos soluciones a este problema. Una es cambiar el escenario gráfico para mantener los límites del comportamiento con el objeto visual. Esto se consigue fácilmente como se demuestra en la Figura 4-5. La solución alternativa usa un objeto **BoundingLeaf** para los límites.



**Recomendaciones de Diseño para la Clase Behavior**

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

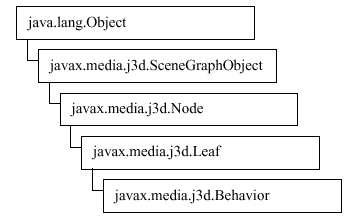
'Quemar la Memoria' es el término para la creacción de objetos innecesarios en Java. La quema de memoria excesiva causará la recolección de basura. Las pausas ocasionales en el renderizado son típicas de la quema de memoria ya que durante la recolección de basura, el renderizado se parará.

Los métodos de la clase **Behavior** frecuentemente son responsables de crear problemas de quema de memoria. Por ejemplo, en el [Fragmento de Código 4-1](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento4-1) el **processStimulus** usa un 'new' en la invocación de **wakeupOn** (línea 24). Esto causa que se cree un nuevo objeto cada vez que se invoca a este método. El objeto se convierte en basura cada vez que se dispara el comportamiento.

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

**API de la Clase Behavior**

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



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

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

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

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

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

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

|  |
| --- |
| Sumario de Métodos de la Clase **Behavior**  Behavior es una clase abstracta que contiene el marco de trabajo para los componentes de comportamiento en Java 3D.  View getView()  Devuelve la vista primaria asociada con este comportamiento.  void initialize()  Inicializa este comportamiento.  void postId(int postId)  Postea la identidad especificada.  void processStimulus(java.util.Enumeration criteria)  Procesa un estímulo para este comportamiento.  void setEnable(boolean state)  Activa o desactiva este comportamiento.  void setSchedulingBoundingLeaf(BoundingLeaf region)  Selecciona la región de límites del comportamiento con los límites del leaf especificado.  void setSchedulingBounds(Bounds region)  Selecciona la región de límites del comportamiento con los límites especificados.  void wakeupOn(WakeupCondition criteria)  Define este criterio de disparo del comportamiento. |

**API ViewPlatform**

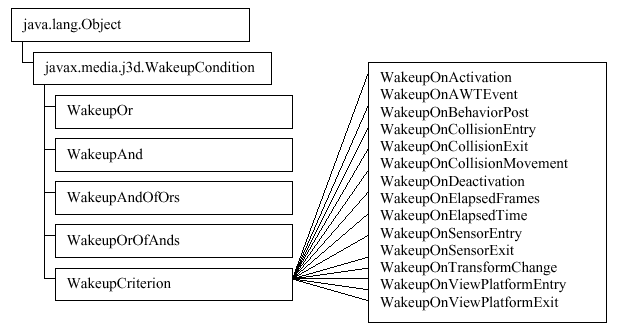
Los Comportamientos están activos (dispuestos para ser disparados) sólo cuando sus límites (o BoundingLeaf) intersecciona con el volumen de activación de una **ViewPlatform**.

|  |
| --- |
| Lista Parcial de Métodos de la Clase **ViewPlatform**  Estos métodos de la clase **ViewPlatform** obtienen y seleccionan el radio del volumen de activación (esfera). El valor por defecto es 62.  float getActivationRadius()  Obtiene el radio de activación del ViewPlatform.  void setActivationRadius(float activationRadius)  Selecciona el radio de activación del ViewPlatform que define un volumen de activación alrededor de la plataforma. |

**Condiciones de Disparo: Cómo se Disparan los Comportamientos**

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

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



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

**WakeupCondition**

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

|  |
| --- |
| Sumario de Métodos de **WakeupCondition**  La clase abstracta **WakeupCondition** es la base para todas las clases **wakeup**. Proporciona los siguientes métodos:  Enumeration allElements()  Devuelve una enumeración con todos los objetos WakeupCriterion en esta condición.  Enumeration triggeredElements()  Devuelve una enumeración de todos los objetos WakeupCriterion disparados en esta condición. |

**WakeupCriterion**

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

|  |
| --- |
| Sumario de Métodos de **WakeupCriterion**  boolean hasTriggered()  Devuelve **true** si el criterio disparó el comportamiento. |

**Clases WakeupCriterion Específicas**

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

|  |  |
| --- | --- |
| **Clase Criterio** | **Disparo** |
| WakeupOnActivation | en la primera detección de una intersección del volumen de activación de un **ViewPlatform** con la región límite del objeto. |
| WakeupOnAWTEvent | cuando ocurre un evento AWT específico |
| WakeupOnBehaviorPost | cuando un objeto **behavior** envía un evento específico |
| WakeupOnCollisionEntry | en la primera detección de colisión del objeto especificaco con otro objeto del escenario gráfico. |
| WakeupOnCollisionExit | cuando el objeto especifico no colisiona con ningún otro objeto del escenario gráfico. |
| WakeupOnCollisionMovement | cuando el objeto especificado se mueve mientras colisiona con otro objeto del escenario gráfico |
| WakeupOnDeactivation | cuando el volumen de activación de un **ViewPlatform** deja de intereseccionar con los límites del objeto |
| WakeupOnElapsedFrames | cuando ha pasado un número determinado de frames |
| WakeupOnElapsedTime | cuando ha pasado un número de segundos determinado |
| WakeupOnSensorEntry | en la primera detección de cualquier sensor que intersecciona con los límites especificados |
| WakeupOnSensorExit | cuando un sensor que interseccionava con los límites del objeto deja de interseccionar con los límites especificados |
| WakeupOnTransformChange | cuando cambia la transformación dentro de un **TransformGroup** especificado |
| WakeupOnViewPlatformEntry | en la primera detección de intersección del volumen de activación de un **ViewPlatform** con los límites especificados |
| WakeupOnViewPlatformExit | cuando el volumen de activación de una vista deja de intereseccionar con los límites especificados |

**Comentarios Generales sobre WakeupCriterion**

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

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

**WakeupOnActivation**

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

|  |
| --- |
| Sumario de Constructores de **WakeupOnActivation**  Extiende: WakeupCriterion  Esta clase especifica la condición de disparo en la primera detección de una interesección del volumen de activación de un **ViewPlatform** con la región límite de su objeto. **WakeupOnActivation** está emparejado con **WakeupOnDeactivation** que veremos más adelante.  WakeupOnActivation()  Construye un nuevo criterio **WakeupOnActivation**. |

**WakeupOnAWTEvent**

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

|  |
| --- |
| Sumario de Constructores de **WakeupOnAWTEvent**  Extiende: WakeupCriterion  Esta clase especifica el disparo de un **Behavior** cuando ocurre un evento AWT específico.  WakeupOnAWTEvent(int AWTId)  Construye un nuevo objeto **WakeupOnAWTEvent**, donde **AWTId** es uno de KeyEvent.KEY\_TYPED, KeyEvent.KEY\_PRESSED, KeyEvent.KEY\_RELEASED, MouseEvent.MOUSE\_CLICKED, MouseEvent.MOUSE\_PRESSED, MouseEvent.MOUSE\_RELEASED, MouseEvent.MOUSE\_MOVED, MouseEvent.MOUSE\_DRAGGED, o uno de los otros muchos valores de eventos.  WakeupOnAWTEvent(long eventMask)  Construye un nuevo objeto **WakeupOnAWTEvent** usando valores ORed EVENT\_MASK. Estos valores son: KEY\_EVENT\_MASK, MOUSE\_EVENT\_MASK, MOUSE\_MOTION\_EVENT\_MASK, u otros valores. |

|  |
| --- |
| Sumario de métodos de **WakeupOnAWTEvent**  AWTEvent[] getAWTEvent()  Recupera el array de eventos AWT consecutivos que ocasionaron el disparo. |

**WakeupOnBehaviorPost**

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

|  |
| --- |
| Sumario de Constructores de **WakeupOnBehaviorPost**  Extiende: WakeupCriterion  Esta clase especifica un disparo de un objeto **Behavior** cuando un comportamiento específico postea un evento específico.  WakeupOnBehaviorPost(Behavior behavior, int postId)  Construye un nuevo criterio **WakeupOnBehaviorPost**. |

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

|  |
| --- |
| Sumario de Métodos de **WakeupOnBehaviorPost**  Behavior getBehavior()  Devuelve el comportamiento especificado en este constructor.  int getPostId()  Recupera el **posId** especificado en el **WakeupCriterion**.  Behavior getTriggeringBehavior()  Devuelve el comportamiento que disparo este evento.  int getTriggeringPostId()  Devuelve el **postId** que causó el disparo del comportamiento. |

El [Fragmento de Código 4-3](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento4-3) y el [Fragmento de Código 4-4](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento4-4) muestran un código parcial para un programa de ejemplo que usa posteo de comportamientos para coordinar comportamientos. El ejemplo abre y cierra una puerta. El código incluye una clase: **OpenBehavior**, y el código que crea los dos objetos **behavior**. El segundo objeto es un ejemplar de **CloseBehavior**, que es casi un duplicado exacto de **OpenBehavior**. En **CloseBehavior**, la condición es compartida en el método initialization (y el comportamiento opuesto completado).

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

1. public class OpenBehavior extends Behavior{

2.

3. private TransformGroup targetTG;

4. private WakeupCriterion pairPostCondition;

5. private WakeupCriterion AWTEventCondition;

6.

7. OpenBehavior(TransformGroup targetTG){

8. this.targetTG = targetTG;

9. AWTEventCondition = new WakeupOnAWTEvent(KeyEvent.KEY\_PRESSED);

10. }

11.

12. public void setBehaviorObjectPartner(Behavior behaviorObject){

13. pairPostCondition = new WakeupOnBehaviorPost(behaviorObject, 1);

14. }

15.

16. public void initialize(){

17. this.wakeupOn(AWTEventCondition);

18. }

19.

20. public void processStimulus(Enumeration criteria){

21. if (AWTEventCondition.hasTriggered()){

22. // make door open – code excluded

23. this.wakeupOn(pairPostCondition);

24. postId(1);

25. } else {

26. this.wakeupOn(AWTEventCondition);

27. }

28. }

29.

30. } // end of class OpenBehavior

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

1. // inside a method to assemble the scene graph ...

2.

3. // create the relevant objects

4. TransformGroup doorTG = new TransformGroup();

5. OpenBehavior openObject = new OpenBehavior(doorTG);

6. CloseBehavior closeObject = new CloseBehavior(doorTG);

7.

8. //prepare the behavior objects

9. openObject.setBehaviorObjectPartner(closeObject);

10. closeObject.setBehaviorObjectPartner(openObject);

11.

12. // set scheduling bounds for behavior objects – code excluded

13.

14. // assemble scene graph – code excluded

15.

Los objetos de estas dos clases responderán en estricta alternancia a los eventos de pulsación de teclas. El comportamiento **OpenBehavior** se disparará en respuesta a la primera pulsación. En su respuesta, señala el comportamiento **CloseBehavior** y selecciona su condición de disparo para que sea una señal para este objeto. El objeto **CloseBehavior** selecciona su condición de disparo para que sea una pulsación de tecla en respuesta a la señal desde el objeto **OpenBehavior**. Puedes encontrar un programa de ejemplo en [DoorApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/interaction/DoorApp.java).

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

**WakeupOnCollisionEntry**

Java 3D puede detectar la colisión de objetos en el mundo virtual. Hay tres clases **WakeupCriterion** útiles para procesar la colisión de objetos: **WakeupOnCollisionEntry**, **WakeupOnCollisionMovement**, y **WakeupOnCollisionExit**.

Un Criterio **WakeupOnCollisionEntry** se disparará cuando un objeto colisione por primera vez. Luego, el criterio **WakeupOnCollisionMovement** disparará (potencialmente varios disparos) mientras dos objetos están en colisión hay un movimiento relativo entre los objetos. Finalmente, un sólo **WakeupOnCollisionExit** se diparará cuando finalice la colisión.

Java 3D sólo puede manejar una colisión por cada objeto a la vez. Una vez que se ha detectado una colisión de un objeto, las colisiones con otros objetos no se detectarán hasta que finalice la primera colisión. También puede ocurrir que una colisión sea tan breve que no sea detectada y por lo tanto no se disparará ninguna condición.

La detección de colisiones es más compleja que esta discusión sobre las condiciones de disparo. Sin embargo este tutorial no cubre la detección de colisiones en detalle, para esto puedes referirte a la Especificación del API Java 3D.

|  |
| --- |
| Sumario de Constructores de **WakeupOnCollisionEntry**  Extiende: WakeupCriterion  Esta clase especifica un disparo en la primera detección de colisión de un objeto especificado con otro objeto en el escenario gráfico. También puedes ver: **WakeupOnCollisionMovement**, y **WakeupOnCollisionExit**.  WakeupOnCollisionEntry(Bounds armingBounds)  Construye un nuevo criterio WakeupOnCollisionEntry.  WakeupOnCollisionEntry(Node armingNode)  Construye un nuevo cirterio WakeupOnCollisionEntry.  WakeupOnCollisionEntry(Node armingNode, int speedHint)  Construye un nuevo criterio WakeupOnCollisionEntry, donde speedHint es:   * USE\_BOUNDS - Usa límites geométircos como una aproximación al cálculo de colisiones. * USE\_GEOMETRY - Usa geometría en el cálculo de colisiones.   WakeupOnCollisionEntry(SceneGraphPath armingPath)  Construye un nuevo criterio WakeupOnCollisionEntry con USE\_BOUNDS como velocidad de choque.  WakeupOnCollisionEntry(SceneGraphPath armingPath, int speedHint)  Construye un nuevo criterio WakeupOnCollisionEntry, donde speedHint es USE\_BOUNDS o USE\_GEOMETRY. |

**WakeupOnCollisionExit**

|  |
| --- |
| Sumario de Constructores de **WakeupOnCollisionExit**  Extiende: WakeupCriterion  Esta clase especifica un disparo cuando se termina la colisión de un objeto especificado con otro objeto en el escenario gráfico. También puedes ver: **WakeupOnCollisionMovement**, y **WakeupOnCollisionExit**.  WakeupOnCollisionExit(Bounds armingBounds)  Construye un nuevo criterio WakeupOnCollisionExit.  WakeupOnCollisionExit(Node armingNode)  Construye un nuevo criterio WakeupOnCollisionExit.  WakeupOnCollisionExit(Node armingNode, int speedHint)  Construye un nuevo criterio WakeupOnCollisionExit, donde speedHint es:   * USE\_BOUNDS - Usa límites geométircos como una aproximación al cálculo de colisiones. * USE\_GEOMETRY - Usa geometría en el cálculo de colisiones.   WakeupOnCollisionExit(SceneGraphPath armingPath)  Construye un nuevo criterio WakeupOnCollisionExit.  WakeupOnCollisionExit(SceneGraphPath armingPath, int speedHint)  Construye un nuevo criterio WakeupOnCollisionExit, donde speedHint es USE\_BOUNDS, o USE\_GEOMETRY. |

|  |
| --- |
| Sumario de Métodos de **WakeupOnCollisionExit**  Bounds getArmingBounds()  Devuelve los límites del objeto usado en la especificación de la condición de colisión.  SceneGraphPath getArmingPath()  Devuelve el path usado en la especificación de la condición de colisión.  Bounds getTriggeringBounds()  Devuelve el objeto Bounds que causó la colisión.  SceneGraphPath getTriggeringPath()  Devuelve el path que describe el objeto que causó la colisión. |

**WakeupOnCollisionMovement**

|  |
| --- |
| Sumario de Constructores de **WakeupOnCollisionMovement**  Extiende: WakeupCriterion  Esta clase especifica un disparo cuando el objeto especificado se mueve durante la colisión con otro objeto en el escenario gráfico. También puedes ver: **WakeupOnCollisionEntry**, y **WakeupOnCollisionExit**.  WakeupOnCollisionMovement(Bounds armingBounds)  Construye un nuevo criterio WakeupOnCollisionMovement.  WakeupOnCollisionMovement(Node armingNode)  Construye un nuevo criterio WakeupOnCollisionMovement.  WakeupOnCollisionMovement(Node armingNode, int speedHint)  Construye un nuevo criterio WakeupOnCollisionMovement, donde speedHint es:   * USE\_BOUNDS - Usa límites geométircos como una aproximación al cálculo de colisiones. * USE\_GEOMETRY - Usa geometría en el cálculo de colisiones.   WakeupOnCollisionMovement(SceneGraphPath armingPath)  Construye un nuevo criterio WakeupOnCollisionMovement.  WakeupOnCollisionMovement(SceneGraphPath armingPath, int speedHint)  Construye un nuevo criterio WakeupOnCollisionMovement, donde speedHint es USE\_BOUNDS, o USE\_GEOMETRY. |

|  |
| --- |
| Sumario de Métodos de **WakeupOnCollisionMovement**  Bounds getArmingBounds()  Devuelve el objeto **Bounds** usado para especificar la condición de colisión.  SceneGraphPath getArmingPath()  Devuelve el path usado en la especificación de la condición de colisión.  Bounds getTriggeringBounds()  Devuelve el objeto **Bounds** que causó la colisión.  SceneGraphPath getTriggeringPath()  Devuelve el path que describe el objeto que causó la colisión. |

**WakeupOnDeactivation**

|  |
| --- |
| Sumario de Constructores de **WakeupOnDeactivation**  Extiende: WakeupCriterion  Esta clase especifica un disparo para la primera detección de que el volumen de activación deja de interseccionar con la región de límites de este objeto. También puedes ver **WakeupOnActivation**.  WakeupOnDeactivation()  Construye un nuevo criterio WakeupOnDeactivation. |

**WakeupOnElapsedFrames**

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

|  |
| --- |
| Sumario de Constructores de **WakeupOnElapsedFrames**  Extiende: WakeupCriterion  Esta clase especifica un disparo cuando han pasado un número especificado de frames.  WakeupOnElapsedFrames(int frameCount)  Construye un nuevo criterio WakeupOnElapsedFrames. |

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

**WakeupOnElapsedTime**

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

|  |
| --- |
| Sumario de Constructores de **WakeupOnElapsedTime**  Extiende: WakeupCriterion  Esta clase especifica un disparo después de que hayan pasado un número de milisegundos especificado.  WakeupOnElapsedTime(long milliseconds)  Construye un nuevo criterio WakeupOnElapsedTime. |

|  |
| --- |
| Sumario de Métodos de **WakeupOnElapsedTime**  long getElapsedFrameTime()  Devuelve el valor de tiempo que se utilizó en la construcción de este objeto. |

**WakeupOnSensorEntry**

En Java 3D, cualquier dispositivo de entrada distinto del teclado o el ratón es un **sensor**. Un sensor es un concepto abstracto para un dispositivo de entrada. Cada sensor tiene un punto caliente definido en el sistema de coordenadas del sensor. La intersección del punto caliente de un sensor con una región puede detectarse con las clases **WakeupOnSensorEntry** y **WakeupOnSensorExit**.

Es posible que un sensor entre y salga de una región armada tan rápidamente que ninguna de las condiciones se dispare.

|  |
| --- |
| Sumario de Constructores de **WakeupOnSensorEntry**  Extiende: WakeupCriterion  Esta clase especifica un disparo en la primera detección de la intersección de cualquier sensor con los límites especificados.  WakeupOnSensorEntry(Bounds region)  Construye un nuevo criterio WakeupOnEntry. |

|  |
| --- |
| Sumario de Métodos de **WakeupOnSensorEntry**  Bounds getBounds()  Devuelve la especificación de límites de este objeto. |

**WakeupOnSensorExit**

|  |
| --- |
| Sumario de Constructores de **WakeupOnSensorExit**  Extiende: WakeupCriterion  Esta clase especifica un disparo en la primera detección de que un sensor que previamente intereseccionaba con los límites deja de intereseccionar con los límites especificados. También puedes ver **WakeupOnSensorEntry**.  WakeupOnSensorExit(Bounds region)  Construye un nuevo criterio WakeupOnExit. |

|  |
| --- |
| Sumario de Métodos de **WakeupOnSensorExit**  Bounds getBounds()  Devuelve la especificación de límites de este objeto. |

**WakeupOnTransformChange**

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

|  |
| --- |
| Sumario de Constructores de **WakeupOnTransformChange**  Extiende: WakeupCriterion  Esta clase especifica un disparo cuando cambia la transformación dentro de un **TransformGroup** especificado.  WakeupOnTransformChange(TransformGroup node)  Construye un nuevo criterio WakeupOnTransformChange. |

|  |
| --- |
| Sumario de Métodos de **WakeupOnTransformChange**  TransformGroup getTransformGroup()  Devuelve el nodo **TransformGroup** usado en la creacción de este WakeupCriterion |

**WakeupOnViewPlatformEntry**

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

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

|  |
| --- |
| Sumario de Constructores de **WakeupOnViewPlatformEntry**  Extiende: WakeupCriterion  Esta clase especifica un disparo en la primera intersección del **ViewPlatform** con los límites especificados.  WakeupOnViewPlatformEntry(Bounds region)  Construye un nuevo criterio WakeupOnEntry. |

|  |
| --- |
| Sumario de Métodos de **WakeupOnViewPlatformEntry**  Bounds getBounds()  Devuelve la especificación de límites de este objeto. |

**WakeupOnViewPlatformExit**

|  |
| --- |
| Sumario de Constructores de **WakeupOnViewPlatformExit**  Extiende: WakeupCriterion  Esta Class especifica un disparo en la primera detección de un Viewplatform que deja de interseccionar con el límite especificado. También puedes ver WakeupOnViewPlatformEntry.  WakeupOnViewPlatformExit(Bounds region)  Construye un nuevo criterio WakeupOnExit. |

|  |
| --- |
| Sumario de Métodos de **WakeupOnViewPlatformExit**  Bounds getBounds()  Devuelve la especificación de límites de este objeto |

**WakeupCondition Composition**

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

**WakeupAnd**

|  |
| --- |
| Sumario de Constructores de **WakeupAnd**  Extiende: WakeupCondition  Esta clase especifica cualquier número de criterios de disparo que son (AND) juntos de forma lógica.  WakeupAnd(WakeupCriterion[] conditions)  Construye una nueva condición WakeupAnd. |

**WakeupOr**

|  |
| --- |
| Sumario de Consctructores de **WakeupOr**  Extiende: WakeupCondition  Esta clase especifica cualquiere número de criterios de disparo que son (OR) juntos de forma lógica.  WakeupAnd(WakeupCriterion[] conditions)  Construye una nueva condición WakeupOr. |

**WakeupAndOfOrs**

|  |
| --- |
| Sumario de Consctructores de **WakeupAndOfOrs**  Extiende: WakeupCondition  Esta clase especifica cualquier número de criterios de disparo WakeupOr que son (AND) juntos de forma lógica.  WakeupAndOfOrs(WakeupOr[] conditions)  Construye una nueva condición WakeupAndOfOrs. |

**WakeupOrOfAnds**

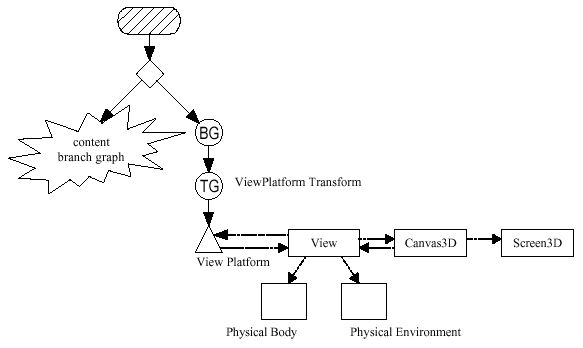
|  |
| --- |
| Sumario de Consctructores de **WakeupOrOfAnds**  Extiende: WakeupCondition  Esta clase especifica cualquier número de criterios de disparo WakeupAnd que son (OR) juntos de forma lógica.  WakeupOrsOfAnds(WakeupAnd[] conditions)  Construye una nueva condición WakeupOrOfAnds. |

**Clases de Comportamientos Útiles para la Navegación por Teclado**

Hasta este momento, el espectador ha estado en una localización fija con una orientación fija. El poder mover el espectador es una capacidad importante en muchas aplicaciones de los gráficos 3D. Java 3D es capaz de mover el espectador. De echo hay clases utilitarias de Java 3D que implementan esta funcionalidad.

La Figura 4-8 muestra la rama gráfica básica para un universo virtual de Java 3D. En esta figura, se considera la plataforma de la visión **transform**. Si se cambia la transformación, el efecto es mover, o reorientar, o ambas, al espectador. De esto, podemos ver que el diseño básico de la navegación del teclado es simple: hacemos que un objeto **behavior** cambie la transformación de la vista de la plataforma en respuesta a los movimientos dominantes.

Este diseño simple es exactamente el modo de trabajo de las clases utilitarias del teclado de Java 3D. Por supuesto podríamos construir nuestro propio comportamiento de navegación del teclado. El resto de esta sección explica cómo utilizar las clases de la navegación del teclado de Java 3D.



**Cómo Navegar en un SimpleUniverse**

Podría ser que pensaramos que necesitar el acceso a los grupos de objeto **Transform** de la plataforma significa abandonar la utilidad **SimpleUniverse**. Sin embargo, **SimpleUniverse**, y las clases relacionadas, proporcionan una combinación de métodos para extraer el objeto **ViewPlatformTransform**. Por lo tanto, podemos tener nuestro **SimpleUniverse** y navegar en él también!

Específicamente, la siguiente línea de código extrae el **ViewPlatformTransform** de un objeto de SimpleUniverse, **su**.

TransformGroup vpt = su.getViewingPlatform().getViewPlatformTransform();

**Programa de Ejemplo de KeyNavigatorBehavior**

Es fácil utilizar la clase de utilidad **KeyNavigatorBehavior** en un programa de Java 3D. Esta sección demuestra el uso de la clase en el programa del ejemplo de [KeyNavigatorApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/interaction/KeyNavigatorApp.java). En este programa podemos ver que los pasos necesarios para usar la clase **KeyNavigatorBehavior** son esencialmente idénticos a los de usar cualquier clase de comportamiento. Los pasos para usar **KeyNavigatorBehavior** se resumen en la siguiente lista.

1. crear un objeto **KeyNavigatorBehavior**, seleccionado el grupo de transformación
2. añadir el objeto **KeyNavigatorBehavior** al escenario gráfico
3. proporcionar unos límites (o BoundingLeaf) para el objeto **KeyNavigatorBehavior**

Como cualquier problema de programación, hay una variedad de maneras de implementar los pasos de esta receta. Un acercamiento es incorporar estos pasos en el método de createSceneGraph. El [Fragmento de Código 4-5](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento4-5) muestra los pasos de la receta según la implementación para el programa del ejemplo de [KeyNavigatorApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/interaction/KeyNavigatorApp.java).

**Fragmento de Código 4-5, usar la clase KeyNavigatorBehavior (parte 1)**

1. public BranchGroup createSceneGraph(SimpleUniverse su) {

2. // Create the root of the branch graph

3. TransformGroup vpTrans = null;

4.

5. BranchGroup objRoot = new BranchGroup();

6.

7. objRoot.addChild(createLand());

8.

9. // create other scene graph content

10.

11.

12. vpTrans = su.getViewingPlatform().getViewPlatformTransform();

13. translate.set( 0.0f, 0.3f, 0.0f); // 3 meter elevation

14. T3D.setTranslation(translate); // set as translation

15. vpTrans.setTransform(T3D); // used for initial position

16. KeyNavigatorBehavior keyNavBeh = new KeyNavigatorBehavior(vpTrans);

17. keyNavBeh.setSchedulingBounds(new BoundingSphere(

18. new Point3d(),1000.0));

19. objRoot.addChild(keyNavBeh);

20.

21. // Let Java 3D perform optimizations on this scene graph.

22. objRoot.compile();

23.

24. return objRoot;

25. } // end of CreateSceneGraph method of KeyNavigatorApp

La ejecución del paso 1 de la receta en el método de createSceneGraph requiere el acceso al grupo de transformación de **ViewPlatform**. Esta implementación pasa el objeto **SimpleUniverse** (línea 34 del [Fragmento de Código 4-6](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento4-6)) al método createSceneGraph que lo hace disponible para tener acceso a la transformación de **ViewPlatform** (la línea 12 del [Fragmento de Código 4-5](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento4-5)).

Pasar el objeto **SimpleUniverse** al método de createSceneGraph permite acceder a otras características de la rama gráfica de la vista de **SimpleUniverse**, tales como **PlatformGeometry**, **ViewerAvatar**, o de agregar un **BoundingLeaf** a la rama gráfica de la vista.

La líneas 13 a 15 del [Fragmento de Código 4-5](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento4-5) proporcionan una posición inicial para el espectador. En este caso, el espectador se mueve a una posición 0,3 metros sobre el origen del mundo virtual. Esto es solamente una posición inicial, y de ninguna manera limita la posición futura o la orientación del espectador.

**Fragmento de Código 4-6, usar la clase KeyNavigatorBehavior (parte 2)**

26. public KeyNavigatorApp() {

27. setLayout(new BorderLayout());

28. Canvas3D canvas3D = new Canvas3D(null);

29. add("Center", canvas3D);

30.

31. // SimpleUniverse is a Convenience Utility class

32. SimpleUniverse simpleU = new SimpleUniverse(canvas3D);

33.

34. **BranchGroup scene = createSceneGraph(simpleU);**

35.

36. simpleU.addBranchGraph(scene);

37. } // end of KeyNavigatorApp (constructor)

**Cómo Crear una Aplicación Universal de un Comportamiento**

Como con cualquier objeto comportamiento, el objeto **KeyNavigtorBehavior** está solo activo cuando sus límites interseccionan con el volumen de activación de un **ViewPlatform**. Esto puede estar particularmente limitado para un comportamiento de navegación, donde el comportamiento debe siempre estar activado. El [Capítulo 3](http://www.labvis.unam.mx/elio/J3D/cursos/3d/cap_3.html) discute una solución a este problema usando un **BoundingLeaf**.

**Clases KeyNavigatorBehavior y KeyNavigator**

La utilidad de navegación del teclado se implementa como dos clases. En el tiempo de ejecución hay dos objetos. El primer objeto es el objeto **KeyNavigatorBehavior**, el segundo es un objeto **KeyNavigator**. La segunda clase no se documenta aquí ya que ni el programador ni el cliente deben saber que existe la segunda clase u objeto.

El objeto **KeyNavigatorBehavior** realiza todas las funciones típicas de una clase de comportamiento, excepto que llama al objeto **KeyNavigator** para realizar la función de processStimulus. La clase **KeyNavigator** toma el **AWTEvent** y lo procesa bajo al nivel de pulsaciones de teclas individuales. La Tabla siguiente muestra el efecto de las pulsaciones de teclas individuales. **KeyNavigator** implementa el movimiento con aceleración.

Movimientos de **KeyNavigatorBehavior**

|  |  |  |
| --- | --- | --- |
| **Tecla** | **Movimiento** | **Movimiento Alt-tecla** |
| <- | rotar a la izquierda | traslación lateral izquierda |
| -> | rotar a la derecha | traslación lateral derecha |
| ^ | mover hacia adelante |  |
| v | mover hacia atrás |  |
| PgUp | rotar arriba | translación hacia arriba |
| PgDn | rotar abajo | translación hacia abajo |
| + | aumenta la distancia de salto (y vuelve al origen) |  |
| - | reduce la distancia de salto |  |
| = | vuelve al centro del universo |  |

|  |
| --- |
| Sumario de Constructores de **KeyNavigatorBehavior**  Paquete: com.sun.j3d.utils.behaviors.keyboard  Extiende: Behavior  Esta clase es un sencillo comportamiento que invoca el **KeyNavigator** para modificar la transformación de la vista de la plataforma.  KeyNavigatorBehavior(TransformGroup targetTG)  Construye un nuevo comportamiento de navegación por teclado que opera sobre el grupo de transformación especificado. |

|  |
| --- |
| Sumario de Métodos de **KeyNavigatorBehavior**  void initialize()  Sobreescribe el método initialize de **Behavior** para configurar los criterios de disparo.  void processStimulus(java.util.Enumeration criteria)  Sobreescribe el método stimulus de **Behavior** para manejar el evento. |

**Clases de Utilidad para Interactuar con el Ratón**

El paquete de comportamientos de ratón (com.sun.j3d.utils.behaviors.mouse) contiene las clases del comportamiento en las cuales el ratón se utiliza como entrada de información para la interacción con los objetos visuales. Incluyendo las clases para traslaciones (moviéndose en un plano paralelo a la placa de la imagen), enfocando (que mueve hacia atrás y adelante), y los objetos visuales que rotan en respuesta a los movimientos del ratón.

La siguiente tabla resume las tres clases específicas del comportamiento del ratón incluidas en el paquete. Además de estas tres clases, está la clase abstracta **MouseBehavior**, y el interface **MouseCallback**. Esta clase abstracta y el interface se utilizan en la creación de las clases específicas del comportamiento del ratón y son útiles para crear comportamientos personalizados del ratón.

Sumario de las clases espécificas de **MouseBehavior**

|  |  |  |
| --- | --- | --- |
| **Clase MouseBehavior** | **Acción en Respuesta a la Acción del Ratón** | **Acción del ratón** |
| MouseRotate | rota el objeto visual sin moverlo | botón izquierdo pulsado con movimiento del ratón |
| MouseTranslate | translada el objeto visual en un plano paralelo al plato de imagen | boton derecho pulsado con movimiento del ratón |
| MouseZoom | translada el objeto visual en un plano orthogonal al plato de imagen | botón central pulsado con movimiento del ratón |

**Usar las Clases de Comportamiento del Ratón**

La clases de comportamientos específicos del ratón son fáciles de usar; es esencialmente lo mismo que el de otras clases de comportamientos. La siguiente lista representa la receta para usarlas:

1. proporcionar capacidades de lectura y escritura para el **transformGroup** fuente
2. crear uno bjeto MouseBehavior
3. seleccionar el **transformGroup** fuente
4. proporcionar unos límites (o BoundingLeaf) para el objeto MouseBehavior
5. añadir el objeto **MouseBehavior** al escenario gráfico

Como con algunas otras recetas, los pasos no tienen que ser realizados en el orden dado. El paso dos se debe realizar antes del tres, del cuatro, y del cinco; los otros pasos se pueden realizar en cualquier orden. También, los pasos dos y tres se pueden combinar usando un constructor diferente.

El [Fragmento de Código 4-7](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento4-7) presenta el método createSceneGraph del programa del ejemplo de [MouseRotateApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/interaction/MouseRotateApp.java). El escenario gráfico incluye el objeto **ColorCube**. El usuario puede rotar el **ColorCube** usando el ratón debido a la inclusión de un objeto **MouseRotate** en el escenario gráfico.

**Fragmento de Código 4-7, usar la clase de utilidad MouseRotate**

1. public class MouseRotateApp extends Applet {

2.

3. public BranchGroup createSceneGraph() {

4. // Create the root of the branch graph

5. BranchGroup objRoot = new BranchGroup();

6.

7. TransformGroup objRotate = new TransformGroup();

8. objRotate.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);

9. objRotate.setCapability(TransformGroup.ALLOW\_TRANSFORM\_READ);

10.

11. objRoot.addChild(objRotate);

12. objRotate.addChild(new ColorCube(0.4));

13.

14. MouseRotate myMouseRotate = new MouseRotate();

15. myMouseRotate.setTransformGroup(objRotate);

16. myMouseRotate.setSchedulingBounds(new BoundingSphere());

17. objRoot.addChild(myMouseRotate);

18.

19. // Let Java 3D perform optimizations on this scene graph.

20. objRoot.compile();

21.

22. return objRoot;

23. } // end of CreateSceneGraph method of MouseRotateApp

La misma receta funcionará para las otras clases de comportamiento del ratón. De hecho los tres comportamientos se pueden utilizar en la misma aplicación que funciona en el mismo objeto visual. Puesto que cada uno de los comportamientos del ratón lee el **transform** fuente antes de escribirlo, sólo se necesita un objeto **TransformGroup** incluso con tres comportamientos de ratón. El programa del ejemplo de [MouseRotateApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/interaction/MouseRotateApp.java) hace apenas eso.

El siguiente ejemplo muestra cómo dos comportamientos del ratón trabajan en un solo mundo virtual. El programa [MouseRotate2App.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/interaction/MouseRotate2App.java) del ejemplo crea un escenario gráfico con dos objetos **ColorCube** uno junto al otro en el mundo virtual. Cada uno de los **ColorCubes** tiene un objeto **MouseRotate** asociado a él. Puesto que ambos objetos de comportamiento del ratón están activos, cuando el usuario hace clic y mueve el ratón, ambos **ColorCubes** rotan.

Si no quisieramos que ambos objetos rotaran, hay dos soluciones: 1) cambiar la posición del espectador, o cambiar los límites del comportamiento, de modo que solamente un comportamiento sesté activo, o 2) usar un mecanismo de selección para aislar el comportamiento.

**Fundamentos del Comportamiento del Ratón**

Las clases específicas de comportamiento del ratón (**MouseRotate**, **MouseTranslate**, y **MouseZoom**) son extensiones de la clase abstracta **MouseBehavior** e implementan el interface **MouseCallback**.

**La Clase Abstracta MouseBehavior**

Esta clase abstracta se presenta aquí en el evento que deseamos ampliarlo para escribir una clase personaliza de comportamiento del ratón. El método SetTransformGroup() es probablemente el único que utilizarán los usuarios de un ejemplar de **MouseBehavior**. Los otros métodos se crearón para los autores de las clases de comportamientos personalizados del ratón.

|  |
| --- |
| Sumario de Métodos de **MouseBehavior**  La clase base para todos los manipuladores de ratón (puedes ver **MouseRotate** y **MouseZoom** para ejemplos).  void initialize()  Inicializa el comportamiento.  void processMouseEvent(java.awt.event.MouseEvent evt)  Maneja eventos del ratón.  void processStimulus(java.util.Enumeration criteria)  Todos los manipuladores de ratón deben implementar este método de **Behavior** (para responder a los estimulos).  void setTransformGroup(TransformGroup transformGroup)  Selecciona el TransformGroup para el comportamiento.  void wakeup()  Dispara manualmente el comportamiento. |

**Interface MouseCallback**

Una clase que implementa este interfaz proporciona al método transformChanged que será llamado cuando cambie el **transform** fuente de la manera especificada. Cada uno de los tres comportamientos específicos del ratón implementa esta clase. Un programador simplemente puede reemplazar el método transformChanged de una de esas clases para especificar un método que se llamará cuando se modifique el **transform**.

|  |
| --- |
| Sumario de Métodos del **Interface MouseBehaviorCallback**  Paquete: com.sun.j3d.utils.behaviors.mouse  void transformChanged(int type, Transform3D transform)  Las clases que implementan este interface que se registran con un **MouseBehaviors** serán llamadas cada vez que el comportamient actualice el **Transform**. El tipo es uno de **MouseCallback.ROTATE**, **MouseCallback.TRANSLATE**, o **MouseCallback.ZOOM**. |

**Clases Específicas de Comportamientos de Ratón**

**MouseRotate**

Un escenario gráfico que incluye un objeto **MouseRotate** permite que el usuario rote objetos visuales en el mundo virtual. Los programas de ejemplo **MouseRotateApp**, **MouseRotate2App**, y **MouseBehaviorApp** demuestran el uso de esta clase.

|  |
| --- |
| Sumario de Constructores de **MouseRotate**  Paquete: com.sun.j3d.utils.behaviors.mouse  Extiende: MouseBehavior  MouseRotate es un objeto de comportamiento de Java3D que deja a los usuarios controlar la rotación de un objeto mediante una pulsación del botón izquierdo del ratón. Para utilizar esta utilidad, primero creamos un **TransformGroup** sobre el que operará este comportamiento. El usuario puede rotar cualquier objeto hijo del **TransformGroup** fuente.  MouseRotate()  Crea un comportamiento mouseRotate por defecto.  MouseRotate(TransformGroup transformGroup)  Crea un comportamiendo dando el transformgroup.  MouseRotate(int flags)  Crea un comportamiento con las banderas seleccionadas, donde las banderas son:   * MouseBehavior.INVERT\_INPUT. Invierte la entradas. * MouseBehavior.MANUAL\_WAKEUP. Dispara manualmente el comportamiento. |

|  |
| --- |
| Sumario de Métodos de **MouseRotate**  void setFactor(double factor)  Selecciona el factor multiplicador para los movimientos en los ejes x e y al valor **factor**  void setFactor(double xFactor, double yFactor)  Selecciona el factor multiplicador para los movimientos en los ejes x e y a los valores **xFactor** e **yFactor** respectivamente.  void setupCallback(MouseBehaviorCallback callback)  El método que se llama cada vez que se actualiza el transformgroup  void transformChanged(Transform3D transform)  Los usuarios pueden sobreescribir este método que es llamado cada vez que el comportamiento actualiza el transformgroup. La implementación por defecto no hace nada. |

**MouseTranslate**

Un escenario gráfico que incluye un objeto **MouseTranslate** permite que el usuario mueva objetos visuales en un plano paralelo a la placa de la imagen en el mundo virtual.

|  |
| --- |
| Sumario de Constructores de **MouseTranslate**  Paquete: com.sun.j3d.utils.behaviors.mouse  Extiende: MouseBehavior  MouseTranslate es un objeto comportamiento de Java3D que permite a los usuarios controlar la traslación (X,Y) de un objeto mediante un movimiento de arrastre del ratón con el botón derecho.  MouseTranslate()  Crea un comportamiento de movimiento por defecto.  MouseTranslate(TransformGroup transformGroup)  Crea un comportamiento de movimiento dando un transformgroup.  MouseTranslate(int flags)  Crea un comportamiento de movimiento con banderas, donde las banderas son:   * MouseBehavior.INVERT\_INPUT. Invierte la entradas. * MouseBehavior.MANUAL\_WAKEUP. Dispara manualmente el comportamiento. |

|  |
| --- |
| Sumario de Métodos de **MouseTranslate**  void setFactor(double factor)  Selecciona el factor multiplicador para los movimientos en los ejes x e y al valor **factor**  void setFactor(double xFactor, double yFactor)  Selecciona el factor multiplicador para los movimientos en los ejes x e y a los valores **XFactor** e **yFactor** respectivamente.  void setupCallback(MouseBehaviorCallback callback)  El método que se llama cada vez que se actualiza el transformgroup  void transformChanged(Transform3D transform)  Los usuarios pueden sobreescribir este método que es llamado cada vez que el comportamiento actualiza el transformgroup. La implementación por defecto no hace nada. |

**MouseZoom**

Un escenario gráfico que incluye un objeto **MouseZoom** permite a los usuarios mover objetos visuales en un plano orthogonal al plato de imagen en un mundo virtual.

|  |
| --- |
| Sumario de Constructores de **MouseZoom**  Paquete: com.sun.j3d.utils.behaviors.mouse  Extiende: MouseBehavior  MouseZoom es un objeto de comportamiento Java3D que permite a los usuarios controlar la traslación en el eje Z de un objeto mediante un movimiento de arrastre del ratón con el botón central (alt-tecla en el PC con el ratón de dos botones).  MouseZoom()  Crea un comportamiento de zoom con ratón por defecto.  MouseZoom(TransformGroup transformGroup)  Crea un comportamiento de zoom dando el transformgroup.  MouseZoom(int flags)  Crea un comportamiento de zoom con banderas, donde las banderas son:   * MouseBehavior.INVERT\_INPUT. Invierte la entradas. * MouseBehavior.MANUAL\_WAKEUP. Dispara manualmente el comportamiento. |

|  |
| --- |
| Sumario de Métodos de **MouseZoom**  void setFactor(double factor)  Selecciona el factor multiplicador del movimiento sobre el eje Z al valor **factor**.  void setupCallback(MouseBehaviorCallback callback)  El método que se llama cada vez que se actualiza el transformgroup  void transformChanged(Transform3D transform)  Los usuarios pueden sobreescribir este método que es llamado cada vez que el comportamiento actualiza el transformgroup. La implementación por defecto no hace nada. |

**MouseNavigation**

Las tres clases específicas de comportamiento del ratón se pueden utilizar para crear un universo virtual en el cual el ratón se utilice para la navegación. Cada una de las clases específicas del comportamiento del ratón tiene un constructor que toma un solo parámetro entero para las banderas. Cuando se utiliza **MouseBehavior.INVERT\_INPUTS** como argumento a este constructor, el comportamiento del ratón responde en la dirección opuesta. Este comportamiento inverso es apropiado para cambiar el **transform** ViewPlatform. Es decir las clases del comportamiento del ratón se pueden utilizar para el control navegacional.

El programa de ejemplo [MouseNavigatorApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/interaction/MouseNavigatorApp.java) utiliza casos de las tres clases específicas del comportamiento del ratón para la interacción navegacional. El [Fragmento de Código 4-8](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento4-8) muestra el método createSceneGraph de este programa del ejemplo.

El **TransformGroup** fuente para cada uno de los objetos del comportamiento del ratón es el ViewPlatform transform. El objeto **SimpleUniverse** es un argumento al método createSceneGraph de modo que se puedan alcanzar los objetos **transform** de ViewPlatform.

**Fragmento de Código 4-8 ,Usar clases de Comportamientos del Ratón para Navegación Interactiva en un Mundo Virtual.**

1. public BranchGroup createSceneGraph(SimpleUniverse su) {

2. // Create the root of the branch graph

3. BranchGroup objRoot = new BranchGroup();

4. TransformGroup vpTrans = null;

5. BoundingSphere mouseBounds = null;

6.

7. vpTrans = su.getViewingPlatform().getViewPlatformTransform();

8.

9. objRoot.addChild(new ColorCube(0.4));

10. objRoot.addChild(new Axis());

11.

12. mouseBounds = new BoundingSphere(new Point3d(), 1000.0);

13.

14. MouseRotate myMouseRotate = new

MouseRotate(MouseBehavior.INVERT\_INPUT);

15. myMouseRotate.setTransformGroup(vpTrans);

16. myMouseRotate.setSchedulingBounds(mouseBounds);

17. objRoot.addChild(myMouseRotate);

18.

19. MouseTranslate myMouseTranslate = new

MouseTranslate(MouseBehavior.INVERT\_INPUT);

20. myMouseTranslate.setTransformGroup(vpTrans);

21. myMouseTranslate.setSchedulingBounds(mouseBounds);

22. objRoot.addChild(myMouseTranslate);

23.

24. MouseZoom myMouseZoom = new

MouseZoom(MouseBehavior.INVERT\_INPUT);

25. myMouseZoom.setTransformGroup(vpTrans);

26. myMouseZoom.setSchedulingBounds(mouseBounds);

27. objRoot.addChild(myMouseZoom);

28.

29. // Let Java 3D perform optimizations on this scene graph.

30. objRoot.compile();

31.

32. return objRoot;

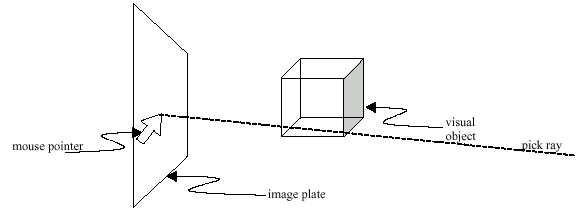
33. } // end of createSceneGraph method of MouseNavigatorApp

Los objetos **bounds** para los objetos de comportamientos de ratón se especifican como un **BoundingSphere** con un radio de 1000 metros. Si el usuario se sale de esta esfera, los objetos comportamiento se desactivarán.

**Picking**

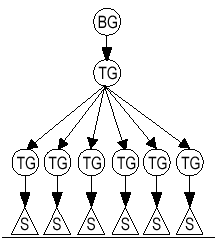
En el programa de ejemplo [MouseNavigatorApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/interaction/MouseNavigatorApp.java), ambos objetos **ColorCube** giran en respuesta a acciones del usuario. En esta aplicación, no hay forma de manipular los cubos de forma separada. "Picking" (Elección) le da al usuario una forma de interactuar con objetos visuales individuales en la escena.

Picking (elección) está implementado por un comportamiento tipicamente disparado por eventos de botones del ratón. En la selección de un objeto visual, el usuario sitúa el puntero del ratón sobre el objeto elegido y pulsa el botón del ratón. El objeto **behavior** se dispara por la pulsación de este botón y empieza la operación de selección. Se proyecta un rayo dentro del mundo virtual desde la posición del puntero del ratón paralela con la proyección. Se calcula la intersección de este rayo con los objetos del mundo virtual. El objeto visual que interseccione más cerca al plato de la imagen se selecciona para interacción. La Figura 4-11 muestra un rayo de selección proyectado en un mundo virtual.



En algunos casos la interacción no se hace directamente con el objeto seleccionado, sino con un objeto en el camino del escenario gráfico hasta el objeto. Por ejemplo, al seleccionar un objeto **ColorCube** para rotación, este objeto no se manipula, se manipula el objeto **TransformGroup** que hay sobre el **ColorCube** en el path del escenario gráfico. Por otro lado, su la operación de selección selecciona un objeto visual para el que se pensó un cambio de color, entonces el objeto visual seleccionado es requerido.

La determinación del objeto para un posterior procesamiento no siempre es sencilla. Si in objeto visual cúbico que va a ser rotado está compuesto por seis objetos **Shape3D** individuales junto con seis objetos **TransformGroup**, como en el escenario gráfico de la Figura 4-12, no es el objeto **TransformGroup** sobre el objeto **Shape3D** interseccionado el que necesita ser modificado. El 'cubo' se rota por la manipulación del objeto **TransformGroup** que es hijo del objeto **BranchGroup** en el escenario gráfico. Por esta razón, el resultado de algunas operaciones de selección es devolver el path del escenario gráfico para su posterior procesamiento.



La comprobación de intersecciones es necesita mucho cálculo. Por lo tanto, la selección es cara y se vuelve más cara con la complejidad del escena. El API Java 3D proporciona varias formas para que un programador pueda limitar la cantidad de cálculos realizados por la selección. Una forma importante es através de las capacidades y atributos de los nodos del escenario gráfico. Si un nodo es o no elegible se selecciona con el método setPickable() de la clase. Un nodo con setPickable() seleccionado a false no es elegible ni ninguno de sus hijos tampoco. Consecuentemente, estos nodos no se tienen en cuenta cuando se calculan las intersecciones.

Otra característica relacionada con la selección en la clase **Node** es la capacidad ENABLE\_PICK\_REPORTING. Esta capacidad sólo se aplica a nodos **Group**. Cuando se selecciona para un grupo, este objeto **group** siempre será incluido en el escenario gráfico devuelto por una operación de selección. Los nodos **Group** no se necesitan para unidades en un escenario gráfico que serán excluidas cuando la capacidad no está seleccionada. No tener seleccionado correctamente los nodos del escenario gráfico es un fuente común de frustacciones en el desarrollo de aplicaciones que utilizan operaciones de selección.

|  |
| --- |
| Lista Parcial de Métodos de **Node**  Extiende: SceneGraphObject  Subclases: Group, Leaf  La clase **Node** proporciona una clase abstracta para todos los nodos **Group** y **Leaf**. Porporciona un marco de trabajo común para construir un escenario gráfico Java 3D, especificamente volúmenes, y las capacidades de selección y colisión.  void setBounds(Bounds bounds)  Selecciona los límites geométricos de un nodo.  void setBoundsAutoCompute(boolean autoCompute)  Activa/desactiva el cálculo automático de los límites geométricos de un nodo.  setPickable(boolean pickable)  Cuando se selecciona a true este nodo puede ser elegido. Cuando se selecciona a false indica que este nodo y sus hijos no son elegibles. |

|  |
| --- |
| Lista Parcial de Capacidades de **Node**  **ENABLE\_PICK\_REPORTING**  Especifica que este nodo será reportado en el **SceneGraphPath** si ocurre una selección. Esta capacidad es sólo aplicable para nodos **Group**; es ignorado para nodos **leaf**. El valor por defecto para nodos **Group** es false. Los nodos interiores no necesitan ser únicos en un **SceneGraphPath** que no tiene seleccionado ENABLE\_PICK\_REPORTING serán excluidos del **SceneGraphPath**.  **ALLOW\_BOUNDS\_READ | WRITE**  Especifica que este nodo permite leer (escribir) la información de sus límites.  **ALLOW\_PICKABLE\_READ | WRITE**  Especifica que este nodo permite leer (escribir) su estado de selección. |

Otra forma en la que un programador puede reducir el cáculo de selección es usar pruebas de interesección de límites en vez de pruebas de intersecciones geométricas. Varias clases relacionadas con la selección (pick) tiene constructores y/o métodos con un parámetro que se selecciona a uno de: **USE\_BOUNDS** o **USE\_GEOMETRY**. Cuando se selecciona **USE\_BOUNDS**, la selección está determinada usando los límites de los objetos visuales, no la geometría real. La determinación de una selección usando los límites es significativamente más sencilla (computacionalmente) para todo excepto para las formas geométricas sencillas y por lo tanto, resulta en un mejor rendimiento. Por supuesto, la pérdida es que la selección no es tan precisa cuando se utilizan límites para su determinación.

Una tercera técnica de programación para reducir el coste de cálculo para la selección es limitar el ámbito de la prueba de selección a la porcion relevante del escenario gráfico. En cada clase de utilidad de selección se selecciona un nodo como el raíz para el gráfico a testear. Este nodo no es necesariamente el raíz de la rama de contenido gráfico. Por el contrario, el nodo pasado debería ser el raíz de la subrama de contenido que sólo contiene objetos elegibles, si es posible. Esta consideración pordría ser una mayor factor de determinación en la construcción de un escenario gráfico para algunas aplicaciones.

**Usar las Clases de Utilidad de Picking**

Hay dos aproximaciones básicas para usar las características de selección de Java 3D, usar objetos de clases **picking**, o crear clases **picking** personalizadas y usar ejemplares de estas clases. El paquete **picking** incluye clases para pick/rotate, pick/translate, y pick/zoom. Es decir, un usuario puede elegir y rotar un objeto presionando el botón del ratón cuando el puntero está sobre el objeto deseado y entonces arrastra el ratón (mientras mantiene pulsado el botón). Cada una de estas clases de **picking** usa un botón diferente del ratón haciendo posible el uso de objeto para las tres clases de **picking** en la misma aplicación simultáneamente.

Cómo un objeto comportamiento **picking** operará sobre cualquier objeto del escenario gráfico (con las capacidades apropiadas), sólo se necesita proporcionar un objeto **picking**. Las dos siguientes líneas de código son todo lo que necesitamos incluir en un programa Java 3D para usar las clases de selección:

PickRotateBehavior behavior = new PickRotateBehavior(root, canvas, bounds);

root.addChild(behavior);

El objeto behavior monitorizará cualquier evento de selección en el escenario gráfico (bajo el nodo raíz) y meneja los arrastres y pulsaciones del ratón. El root proporciona la porción del escenario gráfico a chequear para la selección, el canvas se situa donde está el ratón, y bounds son los límites del objeto de comportamiento **picking**.

Receta para usar las clases de utilidades de **picking**.

1. Crear nuestro escenario gráfico.
2. Crear un objeto behavior **picking** con la especificación de root, canvas, y bounds.
3. Añadir el objeto **behavior** al escenario gráfico.
4. Activar las capacidades apropiadas para los objetos del escenario gráfico.

**Riesgos de Programación cuando se usan Objetos Picking**

Los Riesgos más comunes incluyen; olvidarse de incluir el objeto **behavior** en el escenario gráfico, y no seleccionar los límites apropiados del objeto.

Otro problema común es no seleccionar las capacidades apropiadas para los objetos del escenario gráfico. Hay otros dos problemas menores, que deberíamos chequear si nuestra aplicación no funciona. Uno es no seleccionar apropiadamente el raíz del escenario gráfico. Otro problema potencial es no seleccionar apropiadamente el canvas. Ninguno de estos errores de programación generarán un aviso o mensaje de error.

**Programa de Ejemplo MousePickApp**

El [Fragmento de Código 4-9](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento4-9) muestra el método createSceneGraph de la aplicación [MousePickApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/interaction/MousePickApp.java). Este programa usa un objeto **PickRotate** para proporcionar interacción.

Observa que como la construcción del objeto **picking** requiere un objeto **Canvas3D**, el método createSceneGraph difiere de versiones anteriores por la inclusión del parámetro canvas. Por supuesto, también cambia la correspondiente invocación a createSceneGraph.

**Fragmento de Código 4-9, Método createSceneGraph de la aplicación MousePickApp.**

1. public BranchGroup createSceneGraph(Canvas3D canvas) {

2. // Create the root of the branch graph

3. BranchGroup objRoot = new BranchGroup();

4.

5. TransformGroup objRotate = null;

6. PickRotateBehavior pickRotate = null;

7. Transform3D transform = new Transform3D();

8. BoundingSphere behaveBounds = new BoundingSphere();

9.

10. // create ColorCube and PickRotateBehavior objects

11. transform.setTranslation(new Vector3f(-0.6f, 0.0f, -0.6f));

12. objRotate = new TransformGroup(transform);

13. objRotate.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);

14. objRotate.setCapability(TransformGroup.ALLOW\_TRANSFORM\_READ);

15. objRotate.setCapability(TransformGroup.ENABLE\_PICK\_REPORTING);

16.

17. objRoot.addChild(objRotate);

18. objRotate.addChild(new ColorCube(0.4));

19.

20. pickRotate = new PickRotateBehavior(objRoot,canvas, behaveBounds);

21. objRoot.addChild(pickRotate);

22.

23. // add a second ColorCube object to the scene graph

24. transform.setTranslation(new Vector3f( 0.6f, 0.0f, -0.6f));

25. objRotate = new TransformGroup(transform);

26. objRotate.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);

27. objRotate.setCapability(TransformGroup.ALLOW\_TRANSFORM\_READ);

28. objRotate.setCapability(TransformGroup.ENABLE\_PICK\_REPORTING);

29.

30. objRoot.addChild(objRotate);

31. objRotate.addChild(new ColorCube(0.4));

32.

33. // Let Java 3D perform optimizations on this scene graph.

34. objRoot.compile();

35.

36. return objRoot;

37. } // end of createSceneGraph method of MousePickApp

Este código es similar al de [MouseRotate2App.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/interaction/MouseRotate2App.java), pero es distinto en muchas cosas. Primero, en este programa sólo se usa un objeto **behavior**, mientras que MouseRotate2App usaba dos objetos **behavior** - uno por cada objeto visual. Aunque el código es similar, el comportamiento es diferente. Este programa permite al usuario seleccionar un objeto e interactuar con él. MouseRotate2App rotaba los dos objetos o ninguno.

**El API Corazón de Clases Picking de Java 3D**

Hay tres niveles de clases **picking** en Java 3D. El API corazón de Java 3D proporciona la menor funcionalidad. El paquete de utilidad **picking** proporciona clases de comportamientos generales, elegibles para personalización. El paquete **picking** también proporciona clases **picking** específicas que pueden usarse directamente en programas Java 3D.

Las clases corazón incluyen **PickShape** y **SceneGraphPath**, y métodos de **BranchGroup** y **Locale**. Estas clases proporcionan el mecanismo para especificar una forma usada en la comprobación de intersecciones con objetos visuales. Esta sección presenta el API de las clases **PickShape** y **SceneGraphPath**, y las clases y métodos relacionados.

**Clases PickShape**

Esta clase abstracta no proprociona ni constructores ni métodos. Proporciona abstracción para cuatro subclases: **PickBounds**, **PickRay**, **PickSegment**, y **PickPoint**.

|  |
| --- |
| **PickShape**  Subclases Conocidas: PickBounds, PickRay, PickSegment, PickPoint  Una clase general para describir un forma de selección que puede usarse con métodos de selección de **BranchGroup** y **Locale**. |

**PickBounds**

Los objetos **PickBounds** representan un límite para testear elecciones. Como una subclase de **PickShape**, los objetos **PickBounds** se usan con **BranchGroup** y **Locale** así como con clases del paquete **picking**.

|  |
| --- |
| Sumario de Constructores de **PickBounds**  Extiende: PickShape  Un límite para suministrar a los métodos de selección de **BranchGroup** y **Locale**.  PickBounds()  Crea un PickBounds.  PickBounds(Bounds boundsObject)  Crea un PickBounds con los límites especificados. |

|  |
| --- |
| Sumario de Métodos de **PickBounds**  Bounds get()  Obtiene el **boundsObject** desde este PickBounds.  void set(Bounds boundsObject)  Selecciona el **boundsObject** dentro de este PickBounds. |

**PickPoint**

Los objetos **PickPoint** representan un punto para selección. Como una subclase de **PickShape**, los objetos **PickBounds** se usan con **BranchGroup** y **Locale** así como con clases del paquete **picking**.

|  |
| --- |
| Sumario de Constructores de **PickPoint**  Extiende: PickShape  Suministra un punto a los métodos de selección de **BranchGroup** y **Locale**  PickPoint()  Crea un PickPoint en (0, 0, 0).  PickPoint(Point3d location)  Crea un PickPoint en location. |

|  |
| --- |
| Sumario de Métodos de **PickPoint**  void set(Point3d location)  Selecciona la posición de este PickPoint. Existe un método get correspondiente. |

**PickRay**

Los objetos **PickRay** representan un rayo (un punto y una dirección) para selección. Como una subclase de **PickShape**, los objetos **PickBounds** se usan con **BranchGroup** y **Locale** así como con clases del paquete **picking**.

|  |
| --- |
| Sumario de Constructores de **PickRay**  Extiende: PickShape  PickRay es una encapsulación de un rayo para pasarlo a los métodos de selección en **BranchGroup** y **Locale**  PickRay()  Crea un PickRay con origen y dirección de (0, 0, 0).  PickRay(Point3d origin, Vector3d direction)  Crea un rayo desde origin con dirección a direction. |

|  |
| --- |
| Sumario de Métodos de **PickRay**  void set(Point3d origin, Vector3d direction)  Selecciona el rayo que apunte desde origin en dirección direction. Existe el correspondiente método get. |

**PickSegment**

Los objetos **PickSegment** representan un segmento de línea (definida por dos puntos) para selección. Como una subclase de **PickShape**, los objetos **PickBounds** se usan con **BranchGroup** y **Locale** así como con clases del paquete **picking**.

|  |
| --- |
| Sumario de Constructores de **PickSegment**  Extiende: PickShape  PickRay es una encapsulación de un segmento pasado a los métodos de selección de **BranchGroup** y **Locale**  PickSegment()  Crea un PickSegment.  PickSegment(Point3d start, Point3d end)  Crea un PickSegment desde el punto start hata el punto end. |

|  |
| --- |
| Sumario de métodos de **PickSegment**  void set(Point3d start, Point3d end)  Selecciona el segmento desde el punto start hasta el punto end. Existe el correspondiente método get. |

**SceneGraphPath**

La clase **SceneGraphPath** se usa en la mayoría de las aplicaciones de selección. Esto es porque normalmente la selección implica encontrar un camino de escenario gráfico en el que se encuentra un objeto para permite la manipulación del objeto o de un objeto **TransformGroup** en el path.

Un objeto **SceneGraphPath** representa el camino del escenario gráfico hacia el objeto elegido permitiendo la manipulación del objeto o de un objeto **TransformGroup** en el camino del objeto.

|  |
| --- |
| Introducción a **SceneGraphPath**  Un objeto **SceneGraphPath** representa el camino desde un **Locale** hasta un nodo temrinal en el escenario gráfico. Este camino consiste en un **Locale**, un nodo terminal, y un array de nodos internos que están en el path desde el **Locale** hasta el nodo terminal. El nodo terminal podría ser un nodo **Leaf** o un nodo **Group**. Un **SceneGraphPath** válido debe identificar únicamente un ejemplar de un nodo terminal. Para nodos que no están bajo un **SharedGroup**, el **SceneGraphPath** mínimo consiste en el **Locale** y el propio nodo terminal. Para nodos que están bajo un **SharedGroup**, el **SceneGraphPath** mínimo consiste en el **Locale**, el nodo terminal, y una lista de todos los nodos **Link** en el camino desde el **Locale** hacia el nodo terminal. Un **SceneGraphPath** opcionalmente podría contener otros nodos interiores que están en el camino. Un **SceneGraphPath** se verifica contra errores cuando se envía como argumento a otros métodos de Java 3D.  En el array de nodos internos, el nodo en el índice 0 es el nodo más cercano al **Locale**. El índice se incrementa a lo largo del camino hacia el nodo terminal, con el nodo de índice longitud-1 siendo el nodo más cercano la nodo terminal. El array de nodos no contiene ni el **Locale** (que no es un nodo) ni el nodo terminal. |

|  |
| --- |
| Sumario de Constructores de **SceneGraphPath**  Cuando un **SceneGraphPath** es devuelto desde métodos de selección o colisión de Java 3D, también contiene el valor del objeto **transform** **LocalToVworld** del nodo terminal que era en efecto en el momento en que ocurrió la colisión o la selección. Obherva que **ENABLE\_PICK\_REPORTING** y **ENABLE\_COLLISION\_REPORTING** están desactivados por defecto. Esto significa que los métodos de selección y colisión devolverán el **SceneGraphPath** mínimo por defecto.  SceneGraphPath()  Construye un objeto SceneGraphPath con parámetros por defecto.  SceneGraphPath(Locale root, Node object)  Construye un nuevo objeto SceneGraphPath.  SceneGraphPath(Locale root, Node[] nodes, Node object)  Construye un nuevo objeto SceneGraphPath. |

|  |
| --- |
| Lista Parcial de Métodos de **SceneGraphPath**  boolean equals(java.lang.Object o1)  Devuelve true si el objeto o1 es del tipo **SceneGraphPath** y todos los datos miembros de o1 son iguales a los miembros de datos correspondientes en este **SceneGraphPath** y si los valores de transformación son iguales.  Transform3D getTransform()  Devuelve una copia del transform asociado con este SceneGraphPath; devuelve null si no hay transform.  int hashCode()  Devuelve un número 'hash' basado en los valores de los datos de este objeto.  boolean isSamePath(SceneGraphPath testPath)  Determina si dos objetos **SceneGraphPath** representan el mismo path del escenario grafico; algún objeto podría incluir un subconjunto diferente de nodos internos; sólo los nodos links internos, **Locale**, y el propio nodo son comparados.  int nodeCount()  Recupera el número de nodos de este path.  void set(SceneGraphPath newPath)  Selecciona los valores del path al path especificado.  void setLocale(Locale newLocale)  Selecciona el **Locale** de este path a los Locale especificado.  void setNode(int index, Node newNode)  Reemplaza el nodo en el índice especificado con newNode.  void setNodes(Node[] nodes)  Selecciona el objeto nodo de este path con los objetos nodos especificados.  void setObject(Node object)  Selecciona el nodo terminal de este path al objeto nodo especificado.  void setTransform(Transform3D trans)  Selecciona el componente transform de este **SceneGraphPath** al valor del transform pasado.  java.lang.String toString()  Devuelve una representación string de este objeto; el string contiene los nombres de las clases de todos los nodos en el **SceneGraphPath**, el método toString() de cualquier usuario asociado, también imprime el transform si no es nulo. |

**Métodos de Selección de BranchGroup y Local**

En los siguientes bloques de referencia están los métodos de las clases **BranchGroup** y **Local** para chequeo de intersección con objetos **PickShape**. Este es el nivel de cálculo de selección más bajo proporcionado por el API Java 3D.

|  |
| --- |
| Métodos de selección de **BranchGroup** y **Locale** para su uso con **PickShape**  SceneGraphPath[] pickAll(PickShape pickShape)  Devuelve un array que referencia todos los ítems que son elegibles bajo este **BranchGroup** que intereseccionan con **PickShape**. El array resultante no está ordenado.  SceneGraphPath[] pickAllSorted(PickShape pickShape)  Devuelve un array ordenado de referencias a todos los ítems elegibles que interseccionan con el **pickShape**. **Element [0]** referencia el ítem más cercano al origen de **PickShape**, con los elementos siguientes alejándose del origen. Nota: si **pickShape** es del tipo **PickBounds**, el array resultante no está ordenado.  SceneGraphPath pickClosest(PickShape pickShape)  Devuelve un **SceneGraphPath** que referencia el ítem elegible que está más cercano al origen de **pickShape**. Nota: si **pickShape** es del tipo **PickBounds**, la respuesta es cualquier nodo elegible debajo de este **BranchGroup**.  SceneGraphPath pickAny(PickShape pickShape)  Devuelve una referencia a cualquir ítem elegible debajo de este **BranchGroup** que intersecciona con **pickShape**. |

**Clases Generales del Paquete Picking**

Incluidas en el paquete com.sun.j3d.utils.behaviors.picking hay varias clases de comportamientos generales y específicos. Las clases generales son útiles para crear nuevos comportamientos de selección, entre las que se incluyen **PickMouseBehavior**, **PickObject**, y **PickCallback**. Las clases específicas de comportamiento del ratón, presentadas en la siguiente sección, son subclases de **PickMouseBehavior**.

**Clase PickMouseBehavior**

Esta es la clase base para los comportamientos de selección específicos proporcionados en el paquete. También es útil para extender clases de comportamientos de selección personalizados.

|  |
| --- |
| Sumario de Métodos de **PickMouseBehavior**  Paquete: com.sun.j3d.utils.behaviors.picking  Extiende: Behavior  Clase base que permite a los programadores añadir selección y manipulación del ratón en un escenario gráfico (puedes ver **PickDragBehavior** para un ejemplo de cómo extender esta clase base).  void initialize()  Este método debería ser sobreescrito para proporcionar estado inicial y la condición de disparo inicial.  void processStimulus(java.util.Enumeration criteria)  Este método debería sobreescribirse para proporcionar el comportamiento en respuesta a una condición de disparo.  void updateScene(int xpos, int ypos)  Las subclases deberían implementar esta función update. |

**Clase PickObject**

La clase **PickObject** proporciona métodos para determinar qué objeto fué seleccionado por una operación de selección del usuario. Una amplia variedad de métodos resulta de las distitnas formas posibles de aplicaciones de selección. Es útil crear clases de selección personalizadas.

|  |
| --- |
| Sumario de Constructores de **PickObject**  Paquete: com.sun.j3d.utils.behaviors.picking  Extiende: java.lang.Object  Contiene métodos para ayudar en la selección. Un **PickObject** se crea dando un **Canvas3D** y un **BranchGroup**. **SceneGraphObjects** bajo el **BranchGroup** especificado pueden chequearse para determinar si han sido seleccionados.  PickObject(Canvas3D c, BranchGroup root)  Crea un PickObject. |

|  |
| --- |
| Lista Parcial de Métodos de **PickObject**  **PickObject** tiene numerosos métodos para el cálculo de intersecciones de un **pickRay** con objeto del escenario gráfico. Algunos de los métodos sólo difieren en un parámetro. Por ejemplo el segundo método pickAll (no listado) existe con el firma de método: SceneGraphPath[] pickAll(int xpos, int ypos, int flag), donde **flag** es uno de: **PickObject.USE\_BOUNDS**, o **PickObject.USE\_GEOMETRY**.  Esta lista ha sido ordenada para excluir los métodos con parámetros de bandera. Estos métodos son idénticos a los incluidos en esta lista con la diferencia del parámetro bandera. Estos métodos son: **pickAll**, **pickSorted**, **pickAny**, y **pickClosest**.  PickShape generatePickRay(int xpos, int ypos)  Crea un PickRay que empieza en la posición del espectador y apunta dentro de la escena en dirección a (xpos, ypos) especificados en el espacio de la ventana.  SceneGraphPath[] pickAll(int xpos, int ypos)  Devuelve un array que referencia todos los ítems que son elegibles debajo del **BranchGroup** (especificado en el constructor de PickObject) que interseccionan con un rayo que empieza en la posición del espectador y apunta dentro de la escena en dirección (xpos, ypos) especificados en el espacio de la ventana.  SceneGraphPath[] pickAllSorted(int xpos, int ypos)  Devuelve un array ordenado de referencias a todos los ítems Pickable bajo el **BranchGroup** (especificado en el constructor de PickObject) que interesecciona con el rayo que empieza en la posición del espectador y apunta a la dirección de (xpos, ypos) en el espacio de la ventana.  SceneGraphPath pickAny(int xpos, int ypos)  Devuelve una referencia a cualquier ítem que sea elegible debajo del **BranchGroup** (especificado en el constructor de PickObject) que interesecciona con el rayo que empieza en la posición del espectador y apunta a la dirección (xpos, ypos) en el espacio de la ventana.  SceneGraphPath pickClosest(int xpos, int ypos)  Devuelve una referencia al ítem que está más cercano al espectador y es elegible bajo el **BranchGroup** (especificado en el constructor de PickObject) que interesecciona con el rayo que empieza en la posición del espectador y apunta a la dirección (xpos, ypos) en el espacio de la ventana.  Node pickNode(SceneGraphPath sgPath, int node\_types)  Devuelve una referencia a un nodo elegible que es del tipo especificado que está contenido en el **SceneGraphPath** especificado. Donde **node\_types** es la OR lógica de uno o más: **PickObject.BRANCH\_GROUP**, **PickObject.GROUP**, **PickObject.LINK**, **PickObject.MORPH**, **PickObject.PRIMITIVE**, **PickObject.SHAPE3D**, **PickObject.SWITCH**, **PickObject.TRANSFORM\_GROUP**.  Node pickNode(SceneGraphPath sgPath, int node\_types, int occurrence)  Devuelve una referencia a un nodo elegible que es del tipo especificado que está contenido en el **SceneGraphPath** especificado. Donde **node\_types** está definido en el método anterior. El parámetro **occurrence** indica qué objeto devolver. |

**Interface PickingCallback**

El interface **PickingCallback** proporciona un marco de trabajo para extender una clase de selección existente. En particular cada una de las clases específicas implementa este interface permitiendo al programador proporcionar un método que sea llamado cuando la operación de selección tenga lugar.

|  |
| --- |
| Sumario de Métodos del **Interface PickingCallback**  Paquete:  com.sun.j3d.utils.behaviors.picking  void transformChanged(int type, TransformGroup tg)  Llamado por el **Behavior** **Pick** que es retro-llamado que es registrado cada vez que se intenta la selección. Los valores de tipos válidos son: **ROTATE**, **TRANSLATE**, **ZOOM** o **NO\_PICK** (el usuario hace una selección pero no hay nada seleccionado realmente). |

**Clase Intersect**

La clase **Intersect** proporciona varios métodos para comprobar la intersección de un objeto **PickShape** (clase corazón) y geometrías primitivas. Esta clases es útil para la creacción de clases picking personalizadas.

|  |
| --- |
| Sumario de Constructores de **Intersect**  Paquete: com.sun.j3d.utils.behaviors.picking  Extiende: java.lang.Object  Contiene métodos estáticos para ayudar a las comprobaciones de intersección entre varias clases **PickShape** y geometrías primitivas (como **quad**, **triangle**, **line** y **point**).  Intersect()  Crea un objeto intersect. |

|  |
| --- |
| Lista Parcial de Métodos de **Intersect**  Esta clase tiene vaios métodos de intersección, algunos de los cuales sólo se diferencian por un tipo de parámetro. Por ejemplo el método: boolean pointAndPoint(PickPoint point, Point3f pnt) se diferencia del segundo método listado aquí en el tipo del parámetro pnt. La mayoría de los métodos listado aquí con un parámetro del tipo **Point3d** tienen un método correspondiente con un parámetro del tipo **Point3f**.  boolean pointAndLine(PickPoint point, Point3d[] coordinates, int index)  Devuelve true si el **PickPoint** y el objeto **Line** interseccionan. coordinates[index] y coordinates[index+1] definen la línea.  boolean pointAndPoint(PickPoint point, Point3d pnt)  Devuelve true si el **PickPoint** y el objeto **Point3d** interseccionan.  boolean rayAndLine(PickRay ray, Point3d[] coordinates, int index,  double[] dist)  Devuelve true si el **PickPoint** y el objeto **Line** interseccionan. coordinates[index] y coordinates[index+1] definen la línea.  boolean rayAndPoint(PickRay ray, Point3d pnt, double[] dist)  Devuelve true si el **PickPoint** y el objeto **Point3d** interseccionan.  boolean rayAndQuad(PickRay ray, Point3d[] coordinates, int index,  double[] dist)  Devuelve true si el **PickPoint** y el objeto **cuadrilátero** interseccionan.  boolean rayAndTriangle(PickRay ray, Point3d[] coordinates, int index,  double[] dist)  Devuelve true si el triángulo intersecciona con el rayo, la distancia desde el origen del rayo al punto de intersección, se almacena en dist[0]. coordinates[index], coordinates[index+1], ycoordinates[index+2] definen el triángulo.  boolean segmentAndLine(PickSegment segment, Point3d[] coordinates, int index,  double[] dist)  Devuelve true si la lína intersecciona con el segmento; la distanta desde el inicio del segmento a la intersección se almacena dist[0]. coordinates[index] y coordinates[index+1] definen la línea.  boolean segmentAndPoint(PickSegment segment, Point3d pnt, double[] dist)  Devuelve true si el **PickSegment** y el objeto **Point3d** interseccionan.  boolean segmentAndQuad(PickSegment segment, Point3d[] coordinates, int index,  double[] dist)  Devuelve true si el quad intersecciona con el segmento; la distancia desde el inicio del segmento al punto de intersección se almacena en dist[0].  boolean segmentAndTriangle(PickSegment segment, Point3d[] coordinates,  int index, double[] dist)  Devuelve true si el triangulo intersecciona con el segmento; la distancia desde el inicio del segmento al punto de intersección se almacena en dist[0]. |

**Clases de Comportamientos Picking Específicas**

Incluidas en el paquete com.sun.j3d.utils.behaviors.picking hay clases de comportamientos específicas: **PickRotateBehavior**, **PickTranslateBehavior**, y **PickZoomBehavior**. Estas clases permiten al usuario interactuar con un objeto seleccionado con el ratón. Los comportamientos individuales responden a los diferentes botones del ratón (izquierdo=rotar, derecho=trasladar, central=zoom). Estas clases son subclases de **PickMouseBehavior**.

Los objetos de estas clases pueden incorporarse en mundos virtuales de Java 3D para proporcionar interacción siguiendo la receta anterior. Como todas estas clases implementan el interface **PickingCallback**, la operación de elección pueden mejorarse con una llamada a un método definido por el usuario.

**PickRotateBehavior**

Esta clase permite al usuario seleccionar y rotar interactivamente un objeto visual. El usuario usa el botón izquierdo del ratón para seleccionar y rotar. Se puede usar un ejemplar de **PickRotateBehavior** en conjunción con otras clases de selección específicas.

|  |
| --- |
| Sumario de Constructores de **PickRotateBehavior**  Paquete: com.sun.j3d.utils.behaviors.picking  Extiende: PickMouseBehavior  Implementa: PickingCallback  Un comportamiento de ratón que permite al usuario seleccionar y rotar objetos de un escenario gráfico; expándible a través de retro-llamada.  PickRotateBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds)  Crea un comportamiento que espera eventos del ratón en el escenario gráfico.  PickRotateBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds,  int pickMode)  Crea un comportamiento que espera eventos del ratón en el escenario gráfico. El parámetro pickMode se especifica como uno de **PickObject.USE\_BOUNDS** o **PickObject.USE\_GEOMETRY**. Nota: si pickMode se selecciona a **PickObject.USE\_GEOMETRY**, todos los objetos geométricos del escenario gráfico están disponibles para su selección y deben tener activado su **ALLOW\_INTERSECT**. |

|  |
| --- |
| Sumario de Métodos de **PickRotateBehavior**  void setPickMode(int pickMode)  Selecciona el componente pickMode de este PickRotateBehavior a uno de **PickObject.USE\_BOUNDS** o **PickObject.USE\_GEOMETRY**. Nota: si pickMode se selecciona a **PickObject.USE\_GEOMETRY**, todos los objetos geométricos del escenario gráfico están disponibles para su selección y deben tener activado su **ALLOW\_INTERSECT**.  void setupCallback(PickingCallback callback)  Registra la clase retrollamada a llamar cada vez que el objeto seleccionado se mueve.  void transformChanged(int type, Transform3D transform)  Método de retrollamda desde MouseRotate. Se usa cuando al selección con retrollamada está activa.  void updateScene(int xpos, int ypos)  Actualiza la escena para manipular cualquier nodo. |

**PickTranslateBehavior**

Esta clase permite al usuario seleccionar y trasladar interactivamente un objeto visual. El usuario usa el botón derecho del ratón para seleccionar y trasladar. Se puede usar un ejemplar de **PickTranslateBehavior** en conjunción con otras clases de selección específicas.

|  |
| --- |
| Sumario de Constructores de **PickTranslateBehavior**  Paquete: com.sun.j3d.utils.behaviors.picking  Extiende: PickMouseBehavior  Implementa: PickingCallback  Un comportamiento de ratón que permite al usuario seleccionar y trasladar objetos de un escenario gráfico; expándible a través de retro-llamada.  PickTranslateBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds)  Crea un comportamiento que espera eventos del ratón para el escenario gráfico  PickTranslateBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds,  int pickMode)  Crea un comportamiento que espera eventos del ratón en el escenario gráfico. El parámetro pickMode se especifica como uno de **PickObject.USE\_BOUNDS** o **PickObject.USE\_GEOMETRY**. Nota: si pickMode se selecciona a **PickObject.USE\_GEOMETRY**, todos los objetos geométricos del escenario gráfico están disponibles para su selección y deben tener activado su **ALLOW\_INTERSECT**. |

|  |
| --- |
| Sumario de Métodos de **PickTranslateBehavior**  void setPickMode(int pickMode)  Selecciona el componente pickMode de este PickTranslateBehavior al valor pasado en pickMode.  void setupCallback(PickingCallback callback)  Registra la clase de retrollamada que será llamada cada vez que el objeto seleccionado se mueva.  void transformChanged(int type, Transform3D transform)  Método de retrollamada desde MouseTranslate. Se usa cuando la selección por retrollamada está activa.  void updateScene(int xpos, int ypos)  Actualiza la escena para manipular cualquier nodo. |

**PickZoomBehavior**

Esta clase permite al usuario seleccionar y hacer zoom interactivamente un objeto visual. El usuario usa el botón central del ratón para seleccionar y hacer zoom. Se puede usar un ejemplar de **PickZoomBehavior** en conjunción con otras clases de selección específicas.

|  |
| --- |
| Sumario de Constructores de **PickZoomBehavior**  Paquete: com.sun.j3d.utils.behaviors.picking  Extiende: PickMouseBehavior  Implementa: PickingCallback  Un comportamiento de ratón que permite al usuario seleccionar y hacer zoom a objetos de un escenario gráfico; expándible a través de retro-llamada.  PickZoomBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds)  Crea un comportamiento que espera eventos del ratón para el escenario gráfico.  PickZoomBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds,  int pickMode)  Crea un comportamiento que espera eventos del ratón en el escenario gráfico. El parámetro pickMode se especifica como uno de **PickObject.USE\_BOUNDS** o **PickObject.USE\_GEOMETRY**. Nota: si pickMode se selecciona a **PickObject.USE\_GEOMETRY**, todos los objetos geométricos del escenario gráfico están disponibles para su selección y deben tener activado su **ALLOW\_INTERSECT**. |

|  |
| --- |
| Sumario de Métodos de **PickZoomBehavior**  void setPickMode(int pickMode)  Selecciona el componente pickMode de este PickZoomBehavior al valor pasado en pickMode.  void setupCallback(PickingCallback callback)  Registra la clase de retrollamada a llamar cada vez que el objeto se seleccione.  void transformChanged(int type, Transform3D transform)  Metodo de retrollamada desde MouseZoom. Se usa cuando la selección con retrollamada está activa.  void updateScene(int xpos, int ypos)  Actualiza la escena para manipular cualquier nodo. |