|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **Java 3D**   * [Animación en Java 3D](about:blank)   + [Los Interpoladores y los Objetos Alpha Proporcionan Animaciones Basadas en el Tiempo](about:blank)     - [Alpha](about:blank)     - [Usar Objetos Interpolator y Alpha](about:blank)     - [Ejemplo de uso de Alpha y RotationInterpolator](about:blank)     - [El API Alpha](about:blank)     - [Clases de Comportamiento Interpolator](about:blank)     - [API Corazón de Interpolator](about:blank)     - [Clases PathInterpolator](about:blank)   + [La Clase Billboard](about:blank)     - [Usar un Objeto Billboard](about:blank)     - [Programa de Ejemplo de Billboard](about:blank)     - [El API Billboard](about:blank)   + [Animaciones de Nivel de Detalle (LOD)](about:blank)     - [Usar un Objeto DistanceLOD](about:blank)     - [Ejemplo de uso de DistanceLOD](about:blank)     - [El API DistanceLOD](about:blank)     - [API de LOD (Level of Detail)](about:blank)   + [Morph](about:blank)     - [Usar un Objeto Morph](about:blank)     - [Ejemplo de Aplicación Morph: Walking](about:blank)     - [El API Morph](about:blank)     **Animación en Java 3D**  La igual que las interacciones, las animaciones Java 3D se implementan usando ojetos **Behavior**. Como podrás imaginar, se puede crear cualquier animación personalizada usando objetos **Behavior**. Sin embargo, el API Java 3D proporciona varias clases útiles para crear animaciones sin tener que crear una nueva clase. No debería sorprendernos que estas clases estén basadas en la clase **Behavior**.  Un conjunto de clases de animación es conocido como interpoladores. Un objeto **Interpolator**, junto con un objeto **Alpha**, manipula algún parámetro de un objeto del escenario gráfico para crear animaciones basadas en el tiempo. El objeto **Alpha** proporciona el temporizado.  Otro conjunto de clases de animación animan objetos visuales en respuestas a cambios en la vista. Este conjunto de clases incluye los comportamientos **Billboard** y **Level of Detail (LOD)** que no están dirigidos por el paso del tiempo, sino por la posición u orientación de la vista. La Figura 5-1 muestra las clases de alto nivel del árbol de clases para animación.  Figura 5-1, Algunas Clases Usadas en Animaciones Java 3D  **Los Interpoladores y los Objetos Alpha Proporcionan Animaciones Basadas en el Tiempo**  Un objeto **Alpha** produce un valor entre cero y uno, inclusives, dependiendo de la hora y los parámetros del objeto **Alpha**. Los **Interpolators** son objetos **behavior** personalizados que usan un objeto **Alpha** para proporcionar animaciones de objetos visuales. Las acciones de **Interpolator** incluyen el cambio de localización, orientación, tamaño, color o transpariencia de un objeto visual. Todos los comportamientos interpoladores podrían implementarse creando una clase **behavior** personalizada; sin embargo, usar un **interpolador** hacer más sencilla la creacción de estas animaciones. Las clases **Interpolator** existen para otras acciones, incluyendo algunas combinaciones de estas acciones.  **Alpha**  Un objeto **Alpha** produce un valor, llamado **valor alpha**, entre 0,0 y 1,0; ámbos inclusives. El **valor alpha** cambia con el tiempo según los parámetros especificados en el objeto **alpha**. Para unos parámetros especificos en un momento particular, sólo hay un **valor alpha** que el objeto **alpha** producirá. Dibujando el **valor alpha** sobre el tiempo veremos la forma de onda que produce el objeto **alpha**.  La forma de onda del objeto **alpha** tiene cuatro fases: incremento, alpha a uno, decremento, y alpha a cero. La colección de las cuatro fases es un ciclo de la forma de onda **alpha**. estas cuatro fases corresponden con cuatro parámetros del objeto **Alpha**. La duración de las cuatro fases se especifica por un valor entero expresando su duración en milisegundos. La Figura 5-2 muestra las cuatros fases de la forma de onda **Alpha**.  Todos los tiempos **alpha** son relativos al momento de inicio del objeto **Alpha**. El momento de arranque para todos los objetos **Alpha** se toma de la hora de arranque del sistema. Consecuentemente, los objetos **Alpha** creados en diferentes momentos tienen el mismo momento de inicio. Como resultado, todos los objetos interpoladores, incluso aquellos basados en diferentes objetos **Alpha**, están sincronizados.  Los objetos **Alpha** pueden iniciar sus formas de onda en diferentes momentos. El inicio de la primera forma de onda de un objeto **alpha** puede retrasarse usando alguno de los otros dos parámetros TriggerTime y PhaseDelayDuration. El parámetro TriggerTime especifica un tiempo después de **StartTime** para empezar la operación del objeto **Alpha**. Un tiempo especificado por el parámetros PhaseDelayDuration después de TriggerTime, empezará el primer ciclo de la forma de onda. La Figura 5-2 muestra el StartTime, TriggerTime y PhaseDelayDuration.  Una forma de onda **alpha** podría realizarse sólo una vez, repetirse un número determinado de veces, o hacer un ciclo contínuo. El número de ciclos se especifica en el parámetro loopCount. Cuando loopCount es positivo, especifica un número de ciclos. Si es -1 especifica un bucle contínuo. Cuando la forma de onda **alpha** se repite más de una vez, se repiten las cuatro fases del ciclo, pero no se repite el retardo de fase.  Figura 5-2, Fases de la Forma de Onda Alpha.  Una forma de onda **alpha** no siempre usa las cuatro fases. Podría estar formada por una, dos, tres o las cuatro fases de la forma de onda **Alpha**. La Figura 5-3 muestra seis de las quince formas de ondas posibles.  Figura 5-3, Algunas Formas de Ondas Básicas Creadas con un Objeto Alpha.  El objeto **alpha** tiene dos modos que especifican un subconjunto de fases a usar. El modo **INCREASING\_ENABLE** indica que se usan las fases de incremento alpha y alpha a uno . El modo **DECREASING\_ENABLE** indica que se usan las fases decremento alpha y alpha a cero. Un tercer modo es la combianción de los dos anteriores, lo que indica que se usarán las cuatro fases.  La especificación de modo sobreescribe las selecciones de los parámetros de duración. Por ejemplo, con el modo **INCREASING\_ENABLE**, se ignoran los parámetros DecreasingAlphaDuration, DecreasingAlphaRampDuration, y AlphaAtZeroDuration. También, se puede especificar cualquier forma de onda especificando las duracciones de las fases no necesarias como cero, la especificación apropiada del modo incrementa la eficiciencia del objeto **Alpha**.  **Usar Objetos Interpolator y Alpha**  La receta para usar objetos **Interpolator** y **Alpha** es muy similar a la usada para cualquier objeto **behavior**. La diferencia principal es la inclusión del objeto **Alpha**.   1. crear el objeto fuente con las capacidades apropiadas 2. crear el objeto **Alpha** 3. crear el objeto **Interpolator** que referencia al objeto **Alpha** y al objeto fuente 4. añadir límites al objeto **Interpolator** 5. añadir el objeto **Interpolator** al escenario gráfico   **Ejemplo de uso de Alpha y RotationInterpolator**  [ClockApp.java](http://www.labvis.unam.mx/clases/Animation/ClockApp.java) es un ejemplo de uso de la clase **RotationInterpolator**. La escena es la esfera de un reloj. El reloj es rotado por unos objetos **RotationInterpolator** y **Alpha** una vez por minuto.  En esta aplicación, el objeto fuente es un **TransformGroup**, que necesita la capacidad **ALLOW\_TRANSFORM\_WRITE**. Algunos otros interpoladores actúan sobre diferentes objetos fuente. Por ejemplo, la fuente para un objeto **ColorInterpolator** es un objeto **Material**. Un objeto **interpolator** selecciona los valores de su objeto fuente basándose en el valor **alpha** y en los valores que el propio objeto **interpolator** contiene.  El **interpolator** define los puntos finales de la animación. En el caso de **RotationInterpolator**, el objeto especifica los ángulos inicial y final de la rotación. El **alpha** controla la animación con respecto al tiempo y cómo se moverá el interpolador desde un punto definido hasta el otro especificando las fases de la forma de onda **apha**  Esta aplicación usa las selecciones por defecto de **RotationInterpolator** con un ángulo inicial de cero y un ángulo final de 2P (una rotación completa). El eje por defecto de rotación es el eje y. El objeto **alpha** se selecciona para rotar contínuamente (loopCount = -1) con un periodo de un minuto (60.000 milisegundos). La combinación de estos dos objetos creará el efecto visual de una rotación completa cada minuto. El ciclo se repite continuamente. El resultado parece que el reloj está moviendose continuamente, no como si se parara y empezara de nuevo.  El [Fragmento de Código 5-1](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento5-1) muestra el método createSceneGraph de la aplicación [ClockApp.java](http://www.labvis.unam.mx/elio/J3D/clases/Animation/ClockApp.java).  **Fragmento de Código 5-1, usar un RotationInterpolator y un Alpha en un Reloj (de ClockApp)**  1. public BranchGroup createSceneGraph() {  2. // Create the root of the branch graph  3. BranchGroup objRoot = new BranchGroup();  4.  5. // create target TransformGroup with Capabilities  6. TransformGroup objSpin = new TransformGroup();  7. objSpin.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);  8.  9. // create Alpha that continuously rotates with a period of 1 minute  10. Alpha alpha = new Alpha (-1, 60000);  11.  12. // create interpolator object; by default: full rotation about y-axis  13. RotationInterpolator rotInt = new RotationInterpolator(alpha, objSpin);  14. rotInt.setSchedulingBounds(new BoundingSphere());  15.  16. //assemble scene graph  17. objRoot.addChild(objSpin);  18. objSpin.addChild(new Clock());  19. objRoot.addChild(rotInt);  20.  21. // Let Java 3D perform optimizations on this scene graph.  22. objRoot.compile();  23.  24. return objRoot;  25. } // end of CreateSceneGraph method of ClockApp  La Figura 5-5 es una escena renderizada por **ClockApp** a las 4:30. La cara del reloj es oblicua al espectador porque todo el reloj está rotando.  Figura 5-5, Escena renderizada a las 4:30 por el programa de ejemplo ClockApp.  El programa ClockApp muestra una sencilla aplicación de **RotationInterpolator**. El objeto **Clock**, está definido en el fichero [Clock.java](http://www.labvis.unam.mx/elio/J3D/clases/Animation/Clock.java), muestra una aplicación más avanzada del objeto **RotationInterpolator**. El objeto **clock** usa un objeto **RotationInterpolator** para animar cada manecilla del reloj. Sin embargo, sólo se usa un objeto **alpha**. No es necesario utilizar otro objeto **alpha** para coordinar las manecillas, como se mencionó anteriormente, todos los objetos **Alpha** se sicronizan con el momento de arranque del programa. Sin embargo, compartir un objeto **Alpha** ahorra memoria del sistema.  Algunas de las características especiales de la clase **Clock** son:   * La selección del ángulos inicial y final de las manecillas, * la selección de los ejes de rotación, y * la selección del recorte poligonal de varios componentes del reloj.   **Suavizar la Forma de Onda Alpha**  Además de la duración de las cuatro fases, el programador puede especificar una duración de la rampa para las fases de incremento y decremento **alpha**. Durante la duración de la rampa, el valor de **alpha** cambia gradualmente. En el caso de interpoladores del movimiento, parecerá como si el objeto visual acelerara y decelerara de un forma más natural, como en el mundo real.  El valor de la duración de la rampa se utiliza para las porciones inicial y final de la fase y por lo tanto la duración de la rampa está limitada a la mitad de la duración de la fase. La Figura 5-6 muestra una forma de onda **alpha** con IncreasingAlphaRampDuration y DecreasingAlphaRampDuration. Observa que el valor de **alpha** cambia linearmente entre los dos períodos de la rampa.  Figura 5-6, Suavizar la forma de onda producida por Alpha 9  El programa del ejemplo, [AlphaApp.java](http://www.labvis.unam.mx/elio/J3D/clases/Animation/AlphaApp.java), demuestra el efecto de IncreasingAlphaRampDuration en una forma de onda **alpha**. En este programa hay tres objetos visuales car. Los tres coches comienzan al mismo tiempo en la misma coordenada X y viajan en paralelo. El coche superior no tiene ninguna rampa (duración de la rampa = 0), el coche inferior tiene una duración máxima de la rampa (mitad de la duración del incremento o decremento **alpha**), y el coche del medio tiene la mitad de la duración máxima de la rampa (un cuarto de la duración del incremento o decremento **alpha**). Cada coche tarda dos segundos en cruzar la vista. La Figura 5-7 muestra cuatro escenas renderizadas de esta aplicación.  Figura 5-7, Cuatro escenas renderizadas por AlphaApp mostrando el efecto de IncreasingAlphaRampDuration.  Unos 0,4 segundos después de que los coches comiencen, la primera imagen (izquierda) de la Figura 5-7 fue capturada mostrando las posiciones de los coches. El coche superior, que procederá en un ratio constante en ausencia de una rampa, ha viajado la mayor parte de la distancia en el primer marco. Los otros dos coches comienzan más lentamente y aceleran. Un segundo después (no mostrado), todos los coches han viajado la misma distancia. Las posiciones relativas se invierten durante la segunda mitad de la fase. Al final de los dos segundos, cada uno de los coches ha viajado la misma distancia.  **El API Alpha**  El API de la clase de la **Alpha** es correcto. Cuatro constructores cubren las aplicaciones más comunes. Una plétora de métodos, enumerada en el bloque de referencia siguiente, hace el trabajo fácil de modificar un objeto **alpha** para personalizar un objeto **alpha** para cualquier aplicación.   |  | | --- | | Sumario de Constructores de **Alpha**  Extiende: Object  La clase **Alpha** convierte un valor de tiempo en un valor **alpha** (un valor en el rango 0 a 1, inclusivo). El objeto **alpha** es efectivamente una función de tiempo que genera valores en el rango [ 0.1 ]. Un uso común de **alpha** proporciona valores **alpha** para los comportamientos del interpolador. Las características del objeto **alpha** están determinadas por parámetros definibles por el usuario.  Alpha()  Construye un objeto **Alpha** con el mode = INCREASING\_ENABLE, loopCount = -1, increasingAlphaDuration = 1000, y todos los demás parámetros = 0, excepto StartTime. StartTime se selecciona en el momento de comienzo del programa.  Alpha(int loopCount, long increasingAlphaDuration)  Este constructor sólo toma loopCount e increasingAlphaDuration como parámetros, selecciona el modo a INCREASING\_ENABLE y asigna 0 a todos los demás parámetros (excepto a StartTime).  Alpha(int loopCount, long triggerTime, long phaseDelayDuration,  long increasingAlphaDuration, long increasingAlphaRampDuration,  long alphaAtOneDuration)  Construye un nuevo objeto Alpha y selecciona el modo a INCREASING\_ENABLE.  Alpha(int loopCount, int mode, long triggerTime, long phaseDelayDuration,  long increasingAlphaDuration, long increasingAlphaRampDuration,  long alphaAtOneDuration, long decreasingAlphaDuration,  long decreasingAlphaRampDuration, long alphaAtZeroDuration)  Este constructor toma todos los parámetros definidos por el usuario. |  |  | | --- | | Lista Parcial de Métodos de **Alpha**  Cada uno de estos métodos tiene su correspondiente método get que devuelve un valor del tipo que corresponda al parámetro del método set.  boolean finished()  Comprueba si este objeto alpha ha finalizado totalmente su actividad.  void setAlphaAtOneDuration(long alphaAtOneDuration)  Selecciona el valor de alphaAtOneDuration al valor especificado.  void setAlphaAtZeroDuration(long alphaAtZeroDuration)  Selecciona el valor de alphaAtZeroDuration al valor especificado.  void setDecreasingAlphaDuration(long decreasingAlphaDuration)  Selecciona el valor de decreasingAlphaDuration al valor especificado.  void setDecreasingAlphaRampDuration(long decreasingAlphaRampDuration)  Selecciona el valor de decreasingAlphaRampDuration al valor especificado.  void setIncreasingAlphaDuration(long increasingAlphaDuration)  Selecciona el valor de increasingAlphaDuration al valor especificado.  void setIncreasingAlphaRampDuration(long increasingAlphaRampDuration)  Selecciona el valor de increasingAlphaRampDuration al valor especificado.  void setLoopCount(int loopCount)  Selecciona el valor de loopCount al valor especificado.  void setMode(int mode)  Selecciona el modo al especificado en este argumento. Este puede ser INCREASING\_ENABLE, DECREASING\_ENABLE, o un valor OR de los dos   * DECREASING\_ENABLE - Especifica que se usan las fases 3 y 4 * INCREASING\_ENABLE - Especifica que se usan las fases 1 y 2.   void setPhaseDelayDuration(long phaseDelayDuration)  Selecciona el valor de phaseDelayDuration al valor especificado.  void setStartTime(long startTime)  Selecciona el startTime al valor especificado; startTime selecciona la base (o cero) para todos los cálculos relativos al tiempo, el valor por defecto es la hora de arranque del sistema.  void setTriggerTime(long triggerTime)  Selecciona el valor triggerTime al valor especificado.  float value()  Esta función devuelve un valor entre 0,0 y 1,0 inclusives, basandose en la hora actual y todos los parámetros establecidos para este objeto **alpha**.  float value(long atTime)  Esta función devuelve un valor entre 0,0 y 1,0 inclusives, basandose en la hora actual y todos los parámetros establecidos para este objeto **alpha**. |   **Clases de Comportamiento Interpolator**  La Figura 5-8 muestra las clases del **Interpolator** en los paquetes base y de utilidad. En esta figura, se puede ver que hay unas 10 clases **Interpolator**, y eso son todas las subclases de la clase **Interpolator**. También, esta clase es una extensión de **Behavior**. Los dos rectángulos sombreados representan clases **Interpolator** de utilidad, los otros rectángulos representan clases **Interpolator** básicas.  Figura 5-8, Árbol de clases de la clase Interpolator  Cada **Interpolator** es un **Behavior** personalizado con un disparador para despertar cada marco. En el método processStimulus, de un objeto **Interpolator** controla su objeto **alpha** asociado para saber si el valor actual **alpha**, ajusta la fuente basándose en el valor **alpha**, entonces reajusta su disparador al marco siguiente (a menos que se haya terminado el **alpha**). Alguna de estas funcionalidades se proporcionan en la clase **Interpolator**. La mayoría de este comportamiento se implementa en cada clase individual **Interpolator**.  La mayoría de los objetos del **Interpolator** almacenan dos valores que se utilizan como los puntos finales para la acción interpolada. Por ejemplo, el **RotationInterpolator** guarda dos ángulos que son los extremos de la rotación proporcionada por este **Interpolator**. Por cada marco, el objeto **Interpolator** controla el valor **alpha** de su objeto **alpha** y hace el ajuste rotatorio apropiado a su objeto **TransformGroup** fuenete. Si el valor **alpha** es 0, entonces se usa uno de los valores; si el valor **alpha** es 1, se utiliza el otro valor. Para los valores **alpha** entre 0 y 1, el **Interpolator** interpola linearmente entre los dos valores basándose en el valor **alpha** y utiliza el valor que resulta para el ajuste del objeto fuente  Esta descripción general de **Interpolator** no describe bien las clases **SwitchValueInterpolator** ni **PathInterpolator**. El **SwitchValueInterpolator** elige uno entre los nodos hijos del objeto **Switch** fuente basándose en el valor **alpha**; por lo tanto, no se hace ninguna interpolación en esta clase.  Mientras que varias de las clases **Interpolator** son similares, también se diferencian en algunos detalles. La siguiente tabla muestra algunas de las diferencias entre clases **Interpolator**.   |  |  |  | | --- | --- | --- | | **Clase Interpolator** | **usada para** | **tipo de objeto fuente** | | ColorInterpolator | cambia el color difuso de un objeto(s) | Material | | PathInterpolator | Clase Abstracta | TransformGroup | | PositionInterpolator | cambia la posición de un objeto(s) | TransformGroup | | RotationInterpolator | cambia la rotación (orientación) de un objeto(s) | TransformGroup | | ScaleInterpolator | cambia el tamaño de un objeto(s) | TransformGroup | | SwitchValueInterpolator | elige uno (cambia) entre una colección de objetos | Switch | | TransparencyInterpolator | cambia la transparencia de un objeto(s) | TransparencyAttributes |   El programa del ejemplo, [InterpolatorApp.java](http://www.labvis.unam.mx/elio/J3D/clases/Animation/InterpolatorApp.java), demuestra seis clases no-abstractas **Interpolator** de la tabla anterior. En este programa, cada objeto **Interpolator** está dirigido por un solo objeto **alpha**. La Figura 5-9 muestra dos escenas renderizadas de **InterpolatorApp**. Los cambios en la posición, la rotación, la escala, el color, la transparencia, y el objeto visual (de arriba a abajo) son realizados por los objetos **PositionInterpolator**, **RotationInterpolator**, **ScaleInterpolator**, **ColorInterpolator**, **TransparencyInterpolator**, y **SwitchValueInterpolator**, respectivamente.  Figura 5-9, dos escenadas de InterpolatorApp mostrando varios Interpolators.  **Riesgos de Programación de la Clase Interpolator**  Los objetos **Interpolator** se derivan, y se relacionan de cerca, a objetos **Behavior**. Por lo tanto, usar objetos **Interpolator** da lugar a los mismos riesgos de programación que usar objeto **behavior**. Además de éstos, hay riesgos de programación **Interpolator** en general, y riesgos específicos para algunas clases **Interpolator**.  Un riesgo potencial de programación de **Interpolator** es no darse cuenta de que el objeto **Interpolator** contiene el valor de sus objetos fuente. Podríamos pensar que el **TransformGroup** fuente de un **RotationInterpolator** se puede utilizar para trasladar el objeto visual además de la rotación proporcionada por el **Interpolator**. Esto no es verdad. El objeto **transform** seleccionado en el objeto **TransformGroup** fuente se reescribe en cada marco en que el objeto **alpha** está activo. Esto también significa que dos **Interpolator** no pueden tener el mismo objeto fuente.  Otro riesgo general del **Interpolator** es no seleccionar apropiadamente las capacidades del objeto fuente. Fallar al hacer esto resultará en un error de tiempo de ejecución.  **API Corazón de Interpolator**  Como una clase abstracta, **Interpolator** sólo se usa para crear una subclase nueva. La clase **Interpolator** proporciona solamente un método para los usuarios de sus subclases. Los métodos útiles para la escritura de subclases no se enumeran aquí. La mayoría de la información necesaria para escribir una subclase del **Interpolator** puede obtenerse del [Capítulo anterior](http://www.labvis.unam.mx/elio/J3D/cursos/3d/tutorial.php?id=3d&pag=5)   |  | | --- | | Lista Parcial de Métodos de la Clases **Interpolator**  Extiende: Behavior  Subclases conocidas: ColorInterpolator, PathInterpolator, PositionInterpolator, RotationInterpolator, ScaleInterpolator, SwitchValueInterpolator, TCBSplinePathInterpolator, TransparencyInterpolator  El comportamiento **Interpolator** es un clase abstracta que proporciona bloques de construcción usados por varios interpoladores especializados.  void setAlpha(Alpha alpha)  Selecciona el objeto alpha de este interpolador al objeto alpha especificado. |   **ColorInterpolator**  Un objeto **ColorInterpolator** tiene un objeto **Material** como su fuente. Este **Interpolator** cambia el color difuso del **Material** de la fuente. Esto hace del **ColorInterpolator** tan poderoso como limitado. La potencia viene de la capacidad de tener más de un objeto visual del mismo objeto **Material**. Así pues, un **ColorInterpolator** con una fuente **Material** puede afectar a más de un objeto visual. La limitación es que los objetos visuales con un **Material** NodeComponent son solamente visibles cuando se iluminan.  La mayoría de los riesgos de programación potenciales son el resultado de la complejidad de escenas (iluminadas) sombreadas. La iluminación es tan compleja que es el tema de un capítulo entero, el [Capítulo 6](http://www.labvis.unam.mx/elio/J3D/cursos/3d/cap_6,html). Por ejemplo, el color de un objeto visual sombreado es la combinación de componentes specular, difuso, y ambiente. El **ColorInterpolator** cambia solamente uno de los tres componentes, el color difuso, así que en ciertas situaciones es enteramente posible que parezca que el **ColorInterpolator** no afecta al objeto visual.  Otro riesgo de programación potencial menos exótico es no agregar el objeto **Material** fuente del objeto **Shape3D**. La Figura 5-10 muestra un diagrama parcial del escenario gráfico de un **ColorInterpolator** y de su **Material** NodeComponent fuente.  Figura 5-10, Escenario gráfico parcial de un objeto ColorInterpolator y su objeto fuente Material NodeComponent.  El **ColorInterpolator** es diferente de otros **Interpolator** en el formato de sus métodos get. Los métodos get de **ColorInterpolator** tienen parámetros; por lo tanto se listan con los métodos set   |  | | --- | | Sumario de Constructores de **ColorInterpolator**  Extiende: Interpolator  Esta clase define un comportamiento que modifica el color difuso de su objeto **Material** fuente interpolandolo linearmente entre una pareja de colores especificados (usando el valor generado por el objeto **Alpha**).  ColorInterpolator(Alpha alpha, Material target)  Construye un colorInterpolator trivial con una fuente especificada, un color inicial negro y un color final blanco.  ColorInterpolator(Alpha alpha, Material target, Color3f startColor,  Color3f endColor)  Construye un colorInterpolator con la fuente especificada, y los colores inicial y final. |  |  | | --- | | Lista Parcial de Métodos de **ColorInterpolator**  Los métodos get no siguen las convenciones de los otros interpoladores.  void setEndColor(Color3f color)  Selecciona el color final de este interpolator.  Correspondiente método get: void getEndColor(Color3f color)  void setStartColor(Color3f color)  Selecciona el color inicial de este interpolator.  Correspondiente método get: void getStartColor(Color3f color)  void setTarget(**Material** target)  Seleccion el componente fuente de este interpolator.  Correspondiente método get: Material getTarget() |   **PositionInterpolator**  El **PositionInterpolator** varía la posición de un objeto visual a lo largo de un eje. La especificación de los puntos finales de la interpolación se hace con dos valores de coma flotante y un eje de traslación. El valor por defecto del eje de traslación es el eje X   |  | | --- | | Sumario de Constructores de **PositionInterpolator**  Extiende: Interpolator  Esta clase define un comportamiento que modifica el componente de translación de su **TransformGroup** fuente interpolandolo linearmente entre un par de posiciones especificadas (que usan el valor generado por el objeto **alpha** especificado). La posición interpolada se utiliza para generar una traslación a lo largo del eje X local (o del eje especificado en la traslació) de este **Interpolator**.  PositionInterpolator(Alpha alpha, TransformGroup target)  Construye un positionInterpolator trivial con la fuente especificada, con el eje de traslación por defecto (X), una posición inicial de 0.0f, y una posición final de 1.0f.  PositionInterpolator(Alpha alpha, TransformGroup target,  Transform3D axisOfTranslation, float startPosition, float endPosition)  Construye un nuevo positionInterpolator que varía el componente translacional del TransformGroup (startPosition y endPosition) a lo largo del eje de traslación especificado. |  |  | | --- | | Lista Parcial de Métodos de **PositionInterpolator**  Cada uno de estos métodos tiene un correspondiente método get sin parámetros que devuelve un valor del tipo correspondiente al parámetro del método set.  void setAxisOfTranslation(Transform3D axisOfTranslation)  Selecciona el eje de traslación de este interpolator.  void setEndPosition(float position)  Selecciona la posición final de este interpolator.  void setStartPosition(float position)  Selecciona la posición inicial de este interpolator.  void setTarget(TransformGroup target)  Selecciona la fuente de este interpolator. |   **RotationInterpolator**  El RotationInterpolator varía la orientación rotacional de un objeto visual sobre un eje. La especificación de los puntos finales de la interpolación se hace con dos valores de ángulo en coma flotante y un eje de la rotación. El valor por defecto del eje de rotación es el eje Y positivo.   |  | | --- | | Sumario de Constructores de **RotationInterpolator**  Extiende: Interpolator  Esta clase define un comportamiento que modifica el componente rotacional de su **TransformGroup** fuente interpolandolo linearmente entre un par de ángulos especificados (que usan el valor generado por el objeto **alpha** especificado). El ángulo interpolado se utiliza para generar una rotación sobre el eje Y local de este **Interpolator**, o el eje especificado de rotación.  RotationInterpolator(Alpha alpha, TransformGroup target)  Construye un rotationInterpolator trivial con una fuente especificada el eje de rotación por defecto es (+Y), un ángulo mínimo de 0.0f, y un ángulo máximo de 2\*pi radianes.  RotationInterpolator(Alpha alpha, TransformGroup target,  Transform3D axisOfRotation, float minimumAngle, float maximumAngle)  Construye un nuevo rotationInterpolator que varía el componente rotacional del componente. |  |  | | --- | | Lista Parcial de Métodos de **RotationInterpolator**  Cada uno de estos métodos tiene un correspondiente método get sin parámetros que devuelve un valor del tipo correspondiente al parámetro del método set.  void setAxisOfRotation(Transform3D axisOfRotation)  Selecciona el eje de rotación de este interpolator.  void setMaximumAngle(float angle)  Selecciona el ángulo máximo de este interpolator, en radianes.  void setMinimumAngle(float angle)  Selecciona el ángulo mínimo de este interpolator, en radianes.  void setTarget(TransformGroup target)  Selecciona el nodo TransformGroup para este interpolator. |   **ScaleInterpolator**  El **ScaleInterpolator** varia el tamaño de un objeto visual. La especificación de los puntos finales de la interpolación se hace con dos valores en coma flotante.   |  | | --- | | Sumario de Constructores de **ScaleInterpolator**  Extiende: Interpolator  Esta clase define un comportamiento que modifica el componente de la escala de su **TransformGroup** fuente interpolandolo linearmente entre un par de valores de escala especificados (que usan el valor generado por el objeto **alpha** especificado). El valor de escala interpolado se utiliza para generar una escala en el sistema de coordenadas local de este **Interpolator**.  ScaleInterpolator(Alpha alpha, TransformGroup target)  Construye un scaleInterpolator que varía el nodo TransformGroup de su fuente entre los dos valores **alpha**, una matriz de identidad, una escala mínima de 0.1f, y una escala máxima de 1.0f.  ScaleInterpolator(Alpha alpha, TransformGroup target, Transform3D axisOfScale,  float minimumScale, float maximumScale)  Construye un nuevo scaleInterpolator que varía el componente de escala de su nodo TransformGroup entre dos valores de escala (minimumScale y maximumScale). |  |  | | --- | | Sumario de Métodos de **ScaleInterpolator**  Cada uno de estos métodos tiene un correspondiente método get sin parámetros que devuelve un valor del tipo correspondiente al parámetro del método set.  void setAxisOfScale(Transform3D axisOfScale)  Selecciona el eje de escala para este interpolator.  void setMaximumScale(float scale)  Selecciona la escala máxima para este interpolator.  void setMinimumScale(float scale)  Selecciona la escala mínima para este interpolator.  void setTarget(TransformGroup target)  Selecciona el TransformGroup fuente para este interpolator. |   **SwitchValueInterpolator**  **SwitchValueInterpolator** no interpola entre dos valores como otros **Interpolator**. Selecciona uno de los hijos de un objeto **Switch** para renderizarlo. Los valores de umbral para cambiar a un hijo diferente se determinan uniformemente dividiendo el rango 0,0 a 1,0 por el número de hijos que teine el objeto **Switch**.  Un riesgo potencial específico de programación de **SwitchValueInterpolator** miente en el hecho de que el **Interpolator** no es actualizado cuando cambia el número hijos del objeto **Switch**. Más importante, se determinan los valores de umbral de la conmutación cuando se crea el objeto de **SwitchValueInterpolator**. Así pues, si el **Switch** no tiene ningún hijo antes de que se cree el **Interpolator**, o si el número de hijos cambia después de que se cree el objeto **Interpolator**, entonces el número de hijos en el objeto **Interpolator** debe ser actualizado. La ventaja es que podemos especificar un subconjunto de índices que el **Interpolator** utilizará. El subconjunto se limita a un conjunto secuencial de índices.   |  | | --- | | Sumario de Constructores de **SwitchValueInterpolator**  Extiende: Interpolator  Esta clase define un comportamiento que modifica el hijo seleccionado del nodo **switch** interpolandolo linearmente entre un par hijos especificados (usando el valor generado por el objeto **alpha**).  SwitchValueInterpolator(Alpha alpha, Switch target)  Construye un SwitchValueInterpolator que varía su índice de nodo Switch fuente entre 0 y n-1, donde n es es el número de hijos del nodo Switch fuente.  SwitchValueInterpolator(Alpha alpha, Switch target, int firstChildIndex,  int lastChildIndex)  Construye un SwitchValueInterpolator que varía el índice de nodo entre los dos valores proporcionados. |  |  | | --- | | Lista Parcial de Métodos de **SwitchValueInterpolator**  Cada uno de estos métodos tiene un correspondiente método get sin parámetros que devuelve un valor del tipo correspondiente al parámetro del método set.  void setFirstChildIndex(int firstIndex)  Selecciona el primer índice de hijo para este interpolator.  void setLastChildIndex(int lastIndex)  Selecciona el último índice de hijo para este interpolator.  void setTarget(Switch target)  Selecciona la fuente de este interpolator. |   **Switch**  La clase **Switch** se muestra aquí porque se utiliza en **SwitchValueInterpolator** (y más adelante en **DistanceLOD**). **Switch** se deriva de **Group** y es el padre de cero o más ramas del escenario gráfico. Un objeto **Switch** puede seleccionar cero, uno, o más, incluyendo todos, sus hijos que se renderizarán. Por supuesto un objeto **Switch** se puede utilizar sin un **Interpolator** u objeto **LOD**. El método más comúnmente usado es addChild() derivado de la clase **Group**.   |  | | --- | | Sumario de Constructores de **Switch**  Extiende: Group  El nodo **Switch** controla cuál de sus hijos será renderizado. Define un valor de selección de hijo (un valor switch) que puede seleccionar un sólo hijo, o puede seleccionar cero o más hijos usando una máscara para indicar que hijos son seleccionados para renderización.  Switch()  Construye un nodo Switch con los parámetros por defecto.  Switch(int whichChild)  Construye e inicializa un nodo Switch usando los índices de hijos especificados.   * CHILD\_ALL todos los hijos son renderizados * CHILD\_MASK se usa la máscada childMask para seleccionar los hijos a renderizar * CHILD\_NONE no se renderiza ningún hijo.   Switch(int whichChild, java.util.BitSet childMask)  Construye e inicializa un nodo Switch usando la máscara y el índice de hijos especificados. |  |  | | --- | | Lista Parcial de Métodos de **Switch**  Cada uno de estos métodos tiene un correspondiente método get sin parámetros que devuelve un valor del tipo correspondiente al parámetro del método set.  void setChildMask(java.util.BitSet childMask)  Selecciona la máscara de selección de hijos.  void setWhichChild(int child)  Selecciona el índice de selección de hijos que especifica qué hijo será renderizado. |  |  | | --- | | Sumaro de Capacidades de **Switch**  ALLOW\_SWITCH\_READ | WRITE  Especifica que este nodo permite leer sus valores de selección de hijos de máscara y de hijo actual |   **TransparencyInterpolator**  Un objeto **TransparencyInterpolator** tiene un NodeComponent **TransparencyAttributes** como su fuente. Este **Interpolator** cambia el valor de la transparencia del objeto fuente. Más de un objeto visual pueden compartir un objeto **TransparencyAttributes**. Así pues, un **TransparencyInterpolator** puede afectar a más de un objeto visual. También, debemos tener cuiddo con distintos modos de transparencia que pueden afectar el funcionamiento y el aspecto de representación del objeto visual. Puedes refierte a la especificación de Java 3d API para más información sobre la clase **TransparencyAttributes**.  Un riesgo potencial de programación de **TransparencyInterpolator** es no agregar al objeto fuente **TransparencyAttributes** el manojo de aspecto del objeto visual. Esto es similar a un problema potencial de **ColorInterpolator**.   |  | | --- | | Sumario de Constructores de **TransparencyInterpolator**  Extiende: Interpolator  Esta clase define un comportamiento que modifica la transpariencia de su objeto **TransparencyAttributes** fuente interpolandolo linearmente entre un par de valores de transpariencia especificados (usando el valor generado por el objeto **alpha**).  TransparencyInterpolator(Alpha alpha, TransparencyAttributes target)  Construye un transparencyInterpolator con una fuente especificada, una transpariencia mínima de 0.0f y una transpariencia máxima de 1.0f.  TransparencyInterpolator(Alpha alpha, TransparencyAttributes target,  float minimumTransparency, float maximumTransparency)  Construye un nuevo objeto transparencyInterpolator que varía al transparencia del **Material** fuente entre dos valores de transpariencia. |  |  | | --- | | Sumario de Métodos de **TransparencyInterpolator**  Cada uno de estos métodos tiene un correspondiente método get sin parámetros que devuelve un valor del tipo correspondiente al parámetro del método set.  void setMaximumTransparency(float transparency)  Selecciona el valor máximo de transpariencia para este interpolator.  void setMinimumTransparency(float transparency)  Selecciona el valor mínimo de transpariencia para este interpolator.  void setTarget(TransparencyAttributes target)  Selecciona el objeto TransparencyAttributes fuente para este interpolator. |   **Clases PathInterpolator**  Las clases **PathInterpolator** se diferencian de los otros **Interpolator** en que pueden guardar dos o más valores para la interpolación. El corazón de Java 3D proporciona las clases **PathInterpolator** para la interpolación de la posición, de la rotación, de la traslación y de la escala. La fuente de un objeto **PathInterpolator** es un objeto **TransformGroup** que cambia la posición, la orientación, y la escala, según sea apropiado, para sus objetos hijos.  Los objetos **PathInterpolator** almacenan un conjunto de valores, o de los nudos, que son utilizados por parejas para la interpolación. El valor **alpha** determina que dos valores son utilizados. Los valores de nudo están en el rango de 0,0 a 1,0 inclusivos, que corresponde al rango de los valores del objeto **alpha**. El primer nudo debe tener un valor de 0,0 y el último nudo debe tener un valor de 1,0. Los nudos restantes se deben grabar en orden creciente en el objeto **PathInterpolator**.  Los valores del nudo corresponden con los valores para las variables de los parámetros (por ejemplo, posición o rotación) usado en la interpolación. Hay un valor de parámetro especificado para cada valor del nudo. El nudo con el mayor valor igual o menor que el valor **alpha**, y el nudo siguiente, serán utilizados. Los nudos se especifican en orden, para que en los cambios del valor **alpha**, se utilicen parejas adyacentes.  El panel izquierdo de la Figura 5-11 muestra los valores del nudo para un **PathInterpolator** de posición. Para propósitos ilustrativos, solamente se utilizan las posiciones 2D. El panel central de la figura asocia la posición del objeto visual sobre los valores **alpha**, 0,0 a 1,0. El panel derecho de la figura muestra los valores del nudo usados para los distintos valores **alpha** de este ejemplo. La combinación de los valores del nudo y de los parámetros del **alpha** determinan la animación.  Figura 5-11, la relación entre Nudos y Valores Alpha para un ejemplo de posición 2D.  **Aplicación de Ejemplo de PathInterpolator**  Usar un objeto **PathInterpolator** sigue la misma receta que otros objetos **Interpolator**. La única diferencia es el número de valores usados para inicializar el objeto **PathInterpolator**   1. crear el objeto fuente con las capacidades apropiadas 2. crear el objeto Alpha 3. crear arrays de nudos y otros valores 4. crear el objeto pathInterpolator que referencia al objeto Alpha, al objeto fuente, y al array de selecciones 5. añadir límites al objeto Interpolator 6. añadir el objeto pathInterpolator al escenario gráfico   El programa del ejemplo [RotPosPathApp.java](http://www.labvis.unam.mx/elio/J3D/clases/Animation/RotPosPathApp.java) utiliza un objeto **RotPosPathInterpolator** para animar un objeto **ColorCube** con un número de valores de posición y de rotación. El **RotPosPathInterpolator** graba conjuntos de rotaciones (como un array de Quat4f), de posiciones (como un array de Point3f), y de valores de nudo (como un array de float). [El fragmento de código 5-2](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento5-2) muestra un extracto del ejemplo.  **Fragmento de Código 5-2, un Fragmento del método CreateSceneGraph de RotPosPathApp.java**  1. public BranchGroup createSceneGraph() {  2. BranchGroup objRoot = new BranchGroup();  3.  4. TransformGroup target = new TransformGroup();  5. Alpha alpha = new Alpha(-1, 10000);  6. Transform3D axisOfRotPos = new Transform3D();  7. float[] knots = {0.0f, 0.3f, 0.6f ,1.0f};  8. Quat4f[] quats = new Quat4f[4];  9. Point3f[] positions = new Point3f[4];  10.  11. target.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);  12.  13. AxisAngle4f axis = new AxisAngle4f(1.0f,0.0f,0.0f,0.0f);  14. axisOfRotPos.set(axis);  15.  16. quats[0] = new Quat4f(0.0f, 1.0f, 1.0f, 0.0f);  17. quats[1] = new Quat4f(1.0f, 0.0f, 0.0f, 0.0f);  18. quats[2] = new Quat4f(0.0f, 1.0f, 0.0f, 0.0f);  19.  20. positions[0]= new Point3f( 0.0f, 0.0f, -1.0f);  21. positions[1]= new Point3f( 1.0f, -1.0f, -2.0f);  22. positions[2]= new Point3f( -1.0f, 1.0f, -3.0f);  23.  24. RotPosPathInterpolator rotPosPath = new RotPosPathInterpolator(  25. alpha, target, axisOfRotPos, knots, quats, positions);  26. rotPosPath.setSchedulingBounds(new BoundingSphere());  27.  28. objRoot.addChild(target);  29. objRoot.addChild(rotPosPath);  30. target.addChild(new ColorCube(0.4));  31.  32. return objRoot;  33. } // end of createSceneGraph method of RotPosPathApp  [El fragmento de código 5-2](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento5-2) se basa en el método createSceneGraph de [RotPosPathApp.java](http://www.labvis.unam.mx/elio/J3D/clases/Animation/RotPosPathApp.java). La diferencia está en el número de nudos mostrados en el fragmento del código y los usados en el programa del ejemplo. [RotPosPathApp.java](http://www.labvis.unam.mx/elio/J3D/clases/Animation/RotPosPathApp.java) define nueve nudos mientras que el [fragmento de código 5-2](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento5-2) muestra solamente tres. La Figura 5-13 muestra una imagen de RotPosPathApp. En la imagen, un punto rojo se visualiza por cada uno de las nueve posiciones de nudo. Se reutiliza una de las posiciones, por eso en la figura se ven solo ocho puntos.  Figura 5-13, una escena renderizada por RotPosPathApp mostrando la interpolación de la Rotación y Posición de un ColorCube. Los puntos rojos muestras los nudos de la aplicación de ejemplo.  Cuando se ejecuta el programa del ejemplo **RotPosPathApp**, el **ColorCube** se mueve desde la posición de nudo a otra posición de nudo mientras que rota para alcanzar las distintas rotaciones de nudo. Como con todos los **Interpolator**, la animación que resulta depende de la combinación de los valores del **Interpolator** y de los parámetros **alpha** usados.  Según lo mencionado antes, hay una variedad de subclases de la clase de **PathInterpolator**. Además de estas subclases en el corazón Java 3D, hay pares de clases relacionadas en el paquete de utilidades. La clase **TCBPathSplineInterpolator** es una clase similar a **PathInterpolator**. Tiene una subclase en el paquete de utilidades.  En el ejemplo **RotPosPathApp**, la animación no parece natural debido a la combinación de las posiciones de nudo elegidas. El **ColorCube** se mueve a cada posición de nudo especificada y tan pronto como se alcanza esa posición, el movimiento cambia repentinamente para alcanzar la posición siguiente. Esto no parece natural puesto que este tipo de acción no sucede en el mundo real donde todos los objetos tienen cierta inercia.  **TCBPathSplineInterpolator** es una clase de utilidad que proporciona comportamiento y funciones similares a las de la clase **PathInterpolator**, pero suaviza el camino del objeto visual basándose en un tira en la posición del nudo. El camino de la tira imita el movimiento de objetos del mundo real. En el camino del movimiento de la tira, el objeto visual puede no pasar por todas las (o cualesquiera) posiciones especificadas de nudo. En el subdirectorio jdk1.2/demo/java3d/SplineAnim de la distribución de Java 3D puedes encontrar un ejemplo de utilización de la clase Splineanim.java.  **PathInterpolator**  **PathInterpolator** es una clase abstracta que proporciona el interfaz y las funciones básicas a sus subclases. Los objetos **PathInterpolator** graban los valores de nudo y calculan el índice de los valores de nudo que se utilizarán basándose en el valor **alpha** actual.   |  | | --- | | PathInterpolator  Extiende: Interpolator  Subclases Directas Conocidas: PositionPathInterpolator, RotationPathInterpolator,  RotPosPathInterpolator, RotPosScalePathInterpolator  Esta clase abstracta define la clase base para todos los **PathInterpolator**. Las subclases tienen acceso al método para calcular currentInterpolationValue dando el tiempo actual y el valor **alpha**. El método también calcula el currentKnotIndex, que se basa en el currentInterpolationValue. El currentInterpolationValue se calcula interpolandolo linearmente entre una serie de nudos predefinidos (que usan el valor generado por el objeto **alpha** especificado).  El primer nudo debe tener un valor de 0,0 y el último nudo debe tener un valor de 1,0. Un nudo intermedio con el índice k debe tener un valor terminantemente mayor que cualquier nudo con índice menor que k. |  |  | | --- | | Lista Parcial de Métodos de **PathInterpolator**  int getArrayLengths()  Este método recupera la longitud del array de nudos.  void setKnot(int index, float knot)  Este métodos selecciona el nudo en el índice especificado para este interpolator. |   **RotPosPathInterpolator**  Un objeto **RotPosPathInterpolator** varía la rotación y la posición de un objeto visual basándose en un conjunto de valores de nudo. El constructor es el más importante de las características del API de esta clase. En el constructor todos los valores y objetos relacionados se especifican. Debemos tener cuidado de que cada uno de los arrays debe ser de la misma longitud en este y en todos los objetos **PathInterpolator**.   |  | | --- | | Sumario de Constructores de **RotPosPathInterpolator**  Extiende: PathInterpolator  Esta clase define un comportamiento que modifica los componentes rotacionales y de translación de su **TransformGroup** fuente interpolandolo linearmente entre una serie de pares predefinidos de nudo/position y de nudo/orientation (que usan el valor generado por el objeto **alpha** especificado). La posición y la orientación interpoladas se utilizan para generar una transformación en el sistema de coordenadas local de este **Interpolator**.  RotPosPathInterpolator(Alpha alpha, TransformGroup target,  Transform3D axisOfRotPos, float[] knots, Quat4f[] quats,  Point3f[] positions)  Construye un nuevo interpolador que varía la rotación y la traslación del TransformGroup fuente. |  |  | | --- | | Sumario de Métodos de **RotPosPathInterpolator**  void setAxisOfRotPos(Transform3D axisOfRotPos)  Selecciona el valor del eje de rotación para este interpolator.  void setPosition(int index, Point3f position)  Selecciona la posición del índice especificado para este interpolator.  void setQuat(int index, Quat4f quat)  Selecciona el "quaternion" en el índice especificaco para este interpolator.  void setTarget(TransformGroup target)  Selecciona el TransformGroup fuente para este interpolator. |   **La Clase Billboard**  El término "cartelera" (Billboard) usado en el contexto de los gráficos de ordenador se refiere a la técnica de rotar automáticamente un objeto visual plano para que esté siempre de frente hacia el espectador. La motivación original para el comportamiento de cartelera era permitir el uso de un plano texturado como reemplazo de bajo costo para la geometría compleja. El comportamiento de cartelera todavía se utiliza comúnmente para esta aplicación, pero también se utiliza para otros propósitos, tales como mantener la información textual visible desde cualquier ángulo el ambiente virtual. En Java 3D, la técnica de cartelera se implementa en una subclase de la clase **Behavior**, de ahí la frase "comportamiento de cartelera" usado en la literatura de Java 3D.  La aplicación clásica de ejemplo del comportamiento de cartelera es representar árboles como texturas 2D. Por supuesto, si las texturas se orientan estáticamente, cuando el espectador se mueve, la naturaleza 2D de las texturas se revela. Sin embargo, si los árboles se reorientan de tal forma que siempre están paralelos a su superficie normal, aparecen como objetos 3D. Esto es especialmente cierto si los árboles están en el fondo de una escena o vistos en la distancia.  **Usar un Objeto Billboard**  El comportamiento **Billboard** funciona con los árboles porque los árboles parecen básicamente iguales cuando se ven de frente, de la parte posterior, o de cualquier ángulo. Puesto que el comportamiento **Billboard** hace que un objeto visual parezca exactamente igual cuando se ve desde cualquier ángulo, es apropiado utilizar **Billboard**s y las imágenes 2D para representar los objetos 3D geométricos que son simétricos sobre el eje Y tal como edificios cilíndricos, silos de grano, torres de agua, o cualquier objeto cilíndrico. El comportamiento **Billboard** también se puede utilizar para objetos no simétricos cuando se ven desde suficiente distancia como para ocultar los detalles 2D.  Usar un objeto **Billboard** es similar a usar un **Interpolator** excepto en que no hay un objeto **alpha** para conducir la animación. La animación del objeto **Billboard** la dirige su posición relativa al espectador en el mundo virtual. Abajo podemos ver una receta para el uso **Billboard**.   1. crear un objeto TransformGroup con la capacidad ALLOW\_TRANSFORM\_WRITE 2. crear un objeto Billboard que referencie al objeto fuente TransformGroup 3. suministrar límites para el objeto Billboard 4. ensamblar el escenario gráfico   **Riesgos de Programación de Billboard**  Aunque el uso de un objeto **Billboard** es directo, hay un par de potenciales errores de programación. La primera cosa a observar es que el **TransformGroup** fuente se limpia cada vez que se actualiza. Por lo tanto, este **TransformGroup** no se puede utilizar para posicionar el objeto visual. Si intentamos utilizar la fuente para el posicionamiento, el **Billboard** funcionara, pero en la primera actualización de la rotación, se perderá la información de la posición en la fuente y el objeto visual aparecerá en el origen.  Sin la capacidad de **ALLOW\_TRANSFORM\_WRITE** seleccionada para la fuente, obtendremos un error de tiempo de ejecución. También, si no se fijan los límites, o no se hace correctamente, el objeto de **Billboard** no animará el objeto visual. Los límites se especifican típicamente por un **BoundingSphere** con un radio lo bastante grande para incluir el objeto visual. Justo igual otros objetos **behavior**, dejar el objeto **Billboard** fuera del escenario gáfico lo eliminará del mundo virtual sin ningún error o la alerta.  Hay un problema con la clase **Billboard** que no puede ser superado. En aplicaciones con más de una vista, cada objeto **Billboard** animará correctamente solamente una de las vistas. Esto es una limitación en el diseño de Java 3D y será tratada en una versión posterior.  **Programa de Ejemplo de Billboard**  El programa del ejemplo [BillboardApp.java](http://www.labvis.unam.mx/elio/J3D/clases/Animation/BillboardApp.java) crea un mundo virtual con árboles de un comportamiento **Billboard**. Aunque los árboles se crean a lo bruto (desde las aspas de un ventilador) no parecen como objetos 2D en el fondo.  Hay objetos **TransformGroup** para cada árbol en este ejemplo. Un **TransformGroup**, TGT, traslada simplemente el árbol a la posición para la aplicación. Los Transform TGT no se modifican en tiempo de ejecución. El segundo **TransformGroup**, TGR, proporciona la rotación para el árbol. El TGR es la fuente de **Billboard**.  **Fragmento de Código 5-3, Extracto del método createSceneGraph de BillboardApp.java.**  1. public BranchGroup createSceneGraph(SimpleUniverse su) {  2. BranchGroup objRoot = new BranchGroup();  3.  4. Vector3f translate = new Vector3f();  5. Transform3D T3D = new Transform3D();  6. TransformGroup TGT = new TransformGroup();  7. TransformGroup TGR = new TransformGroup();  8. Billboard billboard = null;  9. BoundingSphere bSphere = new BoundingSphere();  10.  11. translate.set(new Point3f(1.0f, 1.0f, 0.0f));  12. T3D.setTranslation(translate);  13. TGT.set(T3D);  14.  15. // set up for billboard behavior  16. TGR.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);  17. billboard = new Billboard(TGR);  18. billboard.setSchedulingBounds(bSphere);  19.  20. // assemble scene graph  21. objRoot.addChild(TGT);  22. objRoot.addChild(billboard);  23. TGT.addChild(TGR);  24. TGR.addChild(createTree());  25.  26. // add KeyNavigatorBehavior(vpTrans) code removed;  27.  28. return objRoot;  29. } // end of CreateSceneGraph method of BillboardApp  La Figura 5-15 muestra el diagrama del escenario gráfico de los objetos ensamblados en el Fragmento de Código 5-3.  Figura 5-15, Diagrama del Escenario Gráfico que Usa un Objeto Billboard Creado en el Fragmento de Codigo 5-3.  La Figura 5-16 muestra una imagen renderizada del programa del ejemplo [BillboardApp.java](http://www.labvis.unam.mx/elio/J3D/clases/Animation/BillboardApp.java). El [Fragmento de Código 5-3](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento5-3) muestra el código para colocar un árbol animado **Billboard** en un mundo virtual. El programa de BillboardApp coloca varios árboles en el paisaje virtual que es por lo que se ven cuatro árboles en la Figura 5-16.  Figura 5-16, Imagen de BillboardApp con 'Árboles' 2D de cara al Espectador.  El programa de ejemplo **BillboardApp** proporciona un **KeyNavigatorBehavior** de modo que el espectador pueda moverse alrededor y observar los árboles desde varias posiciones y orientaciones.  **El API Billboard**  El ejemplo muestra el uso del valor por defecto del objeto **Billboard**, que es rotar sobre un eje. En este modo de valor por defecto, el objeto visual se rotará solamente sobre el eje Y. Así pues, si los árboles en el programa **BillboardApp** se ven desde arriba o desde abajo, su geometría 2D sería revelada.  El modo alternativo es rotar alrededor de un punto. En este modo, la rotación sería alrededor de un punto tal que el objeto visual se vea siempre orthogonalmente desde cualquier posición de la vista. Es decir nunca será obvio que el objeto visual es de dos dimensiones. Una aplicación posible es representar la luna, u otros objetos esféricos distantes como círculos. Los objetos esféricos aparecen como círculos cuando se ven desde cualquier ángulo.   |  | | --- | | Sumario de Constructores de **Billboard**  Extiende: Behavior  El nodo de comportamiento **Billboard** funciona sobre el nodo **TransformGroup** para hacer que el eje local +z del **TransformGroup** apunte a la posición del ojo del espectador. Esto se hace sin importar las transformaciones sobre el nodo **TransformGroup** especificado en el escenario gráfico. Los nodos **Billboard** proporcionan a la mayoría de las ventajas para los objetos complejos, ásperos-simétricos. Un uso típico puede consistir en un cuadrilátero texturado con la imagen de un árbol.  Billboard()  Construye un nodo Billboard con los parámetros por defecto: mode = ROTATE\_ABOUT\_AXIS, axis =(0,1,0).  Billboard(TransformGroup tg)  Construye un nodo Billboard con los parámetros por defecto que opera sobre el nodo TransformGroup especificado.  Billboard(TransformGroup tg, int mode, Vector3f axis)  Construye un nodo Billboard con el eje y el modo especificados que opera sobre el nodo TransformGroup especificado.  Puedes ver el método setMode() para una explicación del parámetro mode.  Billboard(TransformGroup tg, int mode, Point3f point)  Construye un nodo Billboard con el punto de rotación especificado y el modo que opera sobre el nodo TransformGroup especificado.  Puedes ver el método setMode() para una explicación del parámetro mode. |  |  | | --- | | Lista Parcial de Métodos de **Billboard**  void setAlignmentAxis(Vector3f axis)  Selecciona el eje de alineamiento.  void setAlignmentAxis(float x, float y, float z)  Selecciona el eje de alineamiento.  void setAlignmentMode(int mode)  Selecciona el modo de alineamiento, donde mode es uno de:   * ROTATE\_ABOUT\_AXIS - Especifica qué rotación debería sobre el eje especificado. * ROTATE\_ABOUT\_POINT - Especifica qué rotación denería ser sobre el punto especificado y cual de los ejes Y del hijo debería corresponder con el eje Y de la vista del objeto.   void setRotationPoint(Point3f point)  Selecciona el punto de rotación.  void setRotationPoint(float x, float y, float z)  Selecciona el punto de rotación.  void setTarget(TransformGroup tg)  Selecciona el objeto TransformGroup fuente para este objeto Billboard. |   **Animaciones de Nivel de Detalle (LOD)**  El nivel del detalle (**LOD**) es un término general para una técnica que varía la cantidad de detalle en un objeto visual basándose en un cierto valor del mundo virtual. La aplicación típica es variar el nivel del detalle basándose en la distancia al espectador. Cuanto mayor sea la distancia a un objeto visual, menos detalles aparecerán en la representación. Así pues, la reducción de la complejidad del objeto visual puede no afectar el resultado visual. Sin embargo, disminuir la cantidad de detalle en el objeto visual cuando está lejos del espectador reduce la cantidad de cálculo de renderizado. Si se hace bien, el ahorro de cálculos es significativo y se pueden hacer sin la pérdida de contenido visual.  La clase **DistanceLOD** proporciona comportamiento **LOD** basado en la distancia al espectador. Otras aplicaciones posibles de **LOD** incluyen variar el nivel del detalle basándose en la velocidad de representación (marcos por segundo) con la esperanza de mantener un ratio mínimo de marcos, la velocidad del objeto visual, o el nivel del detalle se podría controlar por las configuraciones del usuario.  Todo objeto **LOD** tiene uno o más objetos **Switch** como fuente. Según lo mencionado antes, un objeto **Switch** es un **group** especial que incluye cero, uno, o más, de sus hijos en el escenario gráfico para renderizar. Con un objeto **DistanceLOD**, la selección del hijo del objeto **Switch** fuente se controla por la distancia del objeto **DistanceLOD** a la vista basada en un conjunto de distancias de umbral.  Las distancias de umbral se especifican en un arrray que comienza con la distancia máxima que utilizará el primer hijo del **Switch** fuente. El primer hijo típicamente es el objeto visual más detallado. Cuando la distancia del objeto **DistanceLOD** a la vista es mayor que este primer umbral, se utiliza el segundo hijo de **Switch**. Cada umbral siguiente de la distancia debe ser mayor que el anterior y especifica la distancia en la cual se utiliza el hijo siguiente del **Switch** fuente.  Si se agrega más de un **Switch** como fuente del objeto **LOD**, cada **Switch** fuente se utiliza en paralelo. Es decir, seleccionan al hijo del mismo índice simultáneamente para cada uno de los **Switch** fuente. De esta manera, un objeto visual complejo puede ser representado por objetos geométricos múltiples que son hijos de diversos nodos **Switch**.  **Usar un Objeto DistanceLOD**  Usar un objeto **DistanceLOD** es similar a usar un **Interpolator** excepto en que no hay un objeto **alpha** para conducir la animación. La animación del objeto **LOD** la dirige su distancia relativa a la vista en el mundo virtual; de esta manera usar un objeto **DistanceLOD** es muy similar a usar un objeto **Billboard**. Usar un objeto de **DistanceLOD** también requiere fijar las distancias del umbral. Abajo tenemos uns receta que nos muestra los pasos para usar un **LOD**.   1. crear un objeto Switch fuente con la capacidad ALLOW\_SWITCH\_WRITE 2. crear un array de distancias de umbral para el objeto DistanceLOD 3. crear el objeto DistanceLOD usando el array de distancias de umbral 4. seleccionar el objeto Switch fuente para el objeto DistanceLOD 5. suministrar límites para el objeto DistanceLOD 6. ensamblar el escenario gráfico, incluyendo la adición de hijos al objeto Switch fuente   **Riesgos de Programación de LOD**  Incluso aunque que el uso de un objeto **LOD** es directo, hay un par de potenciales errores de programación. El error más común es no incluir el objeto **Switch** fuente en el escenario gráfico. Fijar el objeto **Switch** como la fuente del objeto **DistanceLOD** no lo incluye automáticamente en el escenario gráfico.  Sin la capacidad **ALLOW\_SWITCH\_WRITE** fijada para el objeto **Switch** fuente, se generará un error en tiempo de ejecución. También, si no se fijan los límites, o no se hace correctamente, el objeto **LOD** no animará el objeto visual. Los límites se especifican típicamente con un **BoundingSphere** con un radio lo bastante grande como para incluir el objeto visual. Igual que con otros objetos **behavior**, dejar el objeto **LOD** fuera del escenario gráfico lo eliminará del mundo virtual sin ningún error o alerta.  Hay un problema con las clases de **LOD** que no puede ser superada. Igual que con aplicaciones de **Billboard**, en las aplicaciones que tienen más de una vista, el objeto **LOD** animará correctamente sólo una de ellas.  **Ejemplo de uso de DistanceLOD**  El [Fragmento de Código 5-4](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento5-4) muestra un extracto del método createSceneGraph de [DistanceLODApp.java](http://www.labvis.unam.mx/elio/J3D/clases/Animation/DistanceLODApp.java).  **Fragmento de Código 5-4, Extracto del método createSceneGraph en DistanceLODApp.**  1. public BranchGroup createSceneGraph() {  2. BranchGroup objRoot = new BranchGroup();  3. BoundingSphere bounds = new BoundingSphere();  4.  5. // create target TransformGroup with Capabilities  6. TransformGroup objMove = new TransformGroup();  7.  8. // create DistanceLOD target object  9. Switch targetSwitch = new Switch();  10. targetSwitch.setCapability(Switch.ALLOW\_SWITCH\_WRITE);  11.  12. // add visual objects to the target switch  13. targetSwitch.addChild(new Sphere(.40f, 0, 25));  14. targetSwitch.addChild(new Sphere(.40f, 0, 15));  15. targetSwitch.addChild(new Sphere(.40f, 0, 10));  16. targetSwitch.addChild(new Sphere(.40f, 0, 4));  17.  18. // create DistanceLOD object  19. float[] distances = { 5.0f, 10.0f, 20.0f};  20. DistanceLOD dLOD = new DistanceLOD(distances, new Point3f());  21. dLOD.addSwitch(targetSwitch);  22. dLOD.setSchedulingBounds(bounds);  23.  24. // assemble scene graph  25. objRoot.addChild(objMove);  26. objMove.addChild(dLOD); // make the bounds move with object  27. objMove.addChild(targetSwitch); // must add switch to scene graph  28.  29. return objRoot;  30. } // end of CreateSceneGraph method of DistanceLODApp  La Figura 5-18 muestra el diagrama del escenario gráfico creado en el fragmento de código 5-4. Observa que el objeto **Switch** fuente es hijo de un objeto **TransformGroup** y es referido por el objeto **DistanceLOD**. Ambos lazos son necesarios.  Figura 5-18, Diagrama Parcial del Escenario Gráfico del Programa DistanceLODApp.  La Figura 5-19 muestra dos escenas renderizadas por **DistanceLODApp**. Cada escena tiene dos esferas estáticas y una esfera que se mueve. (en la escena derecha, se oculta la esfera de la izquierda.) La esfera móvil se representa por un objeto **DistanceLOD** con cuatro esferas de distinta complejidad geométrica. La esfera pequeña verde es la esfera más detallada usada por el objeto **DistanceLOD** en la distancia máxima. La esfera grande roja es la menos detallada del objeto **DistanceLOD** en la distancia mínima. Las dos esferas estáticas se incluyen para propósitos de comparación.  En esta aplicación el objeto **DistanceLOD** se representa por diversas esferas de color para ilustrar la conmutación. Cada objeto visual usado por un objeto **LOD** debería parecer tan normal como sea apropiado.  Se utiliza un **PositionInterpolator** para mover el objeto **DistanceLOD** hacia adelante y hacia atrás en la escena. Cuanto más lejos de la vista se mueva el objeto **DistanceLOD**, más cambiaran los objetos visuales. Sin el cambio del color en esta aplicación, no sería fácil decir cuándo ocurre la conmutación.  Figura 5-19, Dos escenas Renderizadas desde DistanceLODApp.  En la práctica, normalmente necesitaremos experimentar con las distancias de umbral y las distintas representaciones para conseguir los efectos visuales deseados.  **El API DistanceLOD**  En Java 3D, la clase **LOD** proporciona las funciones básicas para todas las aplicaciones **LOD**. La clase **DistanceLOD** extiende la clase **LOD** para agregar cálculos '**Switch** según la distancia al espectador '. Varios métodos de la clase **LOD** son necesarios para el uso de un objeto **DistanceLOD**.   |  | | --- | | Sumario de Constructores de **DistanceLOD**  Esta clase define un nodo de comportamiento basado en la distancia que funciona sobre un nodo **Switch** para seleccionar uno de los hijos de ese nodo **Switch** basándose en la distancia de este nodo **LOD** hasta el espectador. Un array de n elementos que aumenta monotónicamente según la distancia especificada, tal que distances[0 ] está asociada al nivel más alto de detalle y distances[n-1 ] está asociado al nivel más bajo de detalle. De acuerdo con la distancia real desde el espectador a este nodo **DistanceLOD**, estos valores de la distancia de n [ 0, n-1 ] seleccionan entre de los niveles n+1 de detalle [ 0, n ]. Si d es la distancia del espectador al nodo de **LOD**, entonces la ecuación para determinar el nivel del detalle (hijo del nodo **Switch**) seleccionado es:  0, if d <= distances[0]  i, if distances[i-1] < d <= distances[i]  n, if d > distances[n-1]  Observa que tanto la posición como el array de distancias están especificados en el sistema de coordenadas local de este nodo.  DistanceLOD()  Construye e inicializa un nodo DistanceLOD con los valores por defecto.  DistanceLOD(float[] distances)  Construye e inicializa un nodo DistanceLOD con el array de distancias especificado y la posición por defecto de (0,0,0).  DistanceLOD(float[] distances, Point3f position)  Construye e inicializa un nodo DistanceLOD con el array de distancias y la posición especificados. |  |  | | --- | | Sumario de Métodos de **DistanceLOD**  int numDistances()  Devuelve un contador del número de parámetros de distancia del LOD.  void setDistance(int whichDistance, double distance)  Selecciona una distancia de umbral particular.  void setPosition(Point3f position)  Selecciona la posición de este nodo LOD. |   **API de LOD (Level of Detail)**  Como clase abstracta, la clase **LOD** no se utiliza directamente en los programas de Java 3D. Los métodos de la clase **LOD** se utilizan para manejar el objeto **Switch** fuente de un objeto **DistanceLOD**. También, otras aplicaciones de **LOD** podían crearse extendiendo esta clase según sea apropiado.   |  | | --- | | Sumario de Constructores de **LOD**  Un nodo hoja **LOD** es una clase de comportamiento abstracto que opera sobre una lista de nodos **Switch** para seleccionar uno de los hijos de los nodos **Switch**. La clase **LOD** se extiende para implentar distintos crierios de selección.  LOD()  Construye e inicializa un nodo LOD. |  |  | | --- | | Sumario de Métodos de **LOD**  void addSwitch(Switch switchNode)  Añade el switch especificado a la lista de switches de este nodo LOD.  java.util.Enumeration getAllSwitches()  Devuelve el objeto enumeration de todos los switches.  void insertSwitch(Switch switchNode, int index)  Inserta el nodo switch especificado en el índice especificado.  int numSwitches()  Devuelve un contador de switches de este LOD.  void removeSwitch(int index)  Elimina el nodo switch del índice especificado.  void setSwitch(Switch switchNode, int index)  Reemplaza el nodo switch especificado con el nodo switch proporcionado. |   **Morph**  Las clases **Interpolator** modifican varios atributos visuales del mundo virtual. Sin embargo, no hay un **Interpolator** para cambiar la geometría de un objeto visual. Esto es exactamente lo que hace la clase **Morph**. Un objeto **Morph** crea la geometría para un objeto visual con la interpolación de un conjunto de objetos **GeometryArray**. De esta forma la clase **Morph** es como las clases **Interpolator**. Pero, **Morph** no es un **Interpolator**; no es ni siquiera una extensión de la clase **Behavior**. La clase **Morph** extiende la clase **Node**.  El [Capítulo 4](http://www.labvis.unam.mx/elio/J3D/cursos/3d/tutorial.php?id=5) explica que todos los cambios en una escena viva o los objetos en un gráfico vivo de la escena se hacen normalmente con el método processStimulus de los objetos **Behavior**. Puesto que no hay una clase específica de comportamiento para el uso con un objeto **Morph**, se debe escribir un clase personalizada para aplicarla con **Morph**. Si la clase **Morph** se considera una clase de animación o de interacción depende del estímulo para el comportamiento que diriga el objeto **Morph**.  Los objetos **Morph** se pueden utilizar para convertir las pirámides en cubos, gatos en perros, o cambiar cualquier geometría en cualquier otra geometría. La única limitación es que los objetos de la geometría usados para la interpolación sean de la misma clase, una subclase de **GeometryArray**, y con el mismo número de vértices. La restricción en el número de vértices no es un límite como parece a primera vista. En el Java 3D se distibuye un programa de ejemplo que convierte una pirámide en un cubo, Pyramid2Cube.java.  Los objetos **Morph** también se pueden utilizar para animar un objeto visual (por ejemplo., hacer que una persona corra). En el API Java 3D también puedes encontrar un programa que anima una mano, Morphing.java. Un tercer ejemplo de **Morph** que hace caminar a una figura de alambre es el tema de la sección siguiente.  **Usar un Objeto Morph**  Entender el uso del objeto de **Morph** requiere saber cómo funcionan los objetos **Morph**. Afortunadamente, un objeto **Morph** no es muy complejo. Un objeto **Morph** graba un array de objetos **GeometryArray**. Podemos recordar del [Capítulo 2](http://www.labvis.unam.mx/elio/J3D/cursos/3d/cap_2.html) que **GeometryArray** es la superclase de **TriangleArray**, de **QuadStripArray**, de **IndexedLineStripArray**, y de **TriangleFanArray** (entre otros muchos).  El **GeometryArray** individual define completamente una especificación geométrica completa para el objeto visual incluyendo color, superficies normales, y coordenadas de la textura. Los objetos **GeometryArray** se pueden imaginar como marcos de una animación, o más correctamente, como las constantes en una ecuación para crear un nuevo objeto **GeometryArray**.  Además del array de objetos **GeometryArray**, un objeto **Morph** tiene un array de valores del peso- éstas son las variables en la ecuación. Usando el **GeometryArray** y los pesos, un objeto **Morph** construye un nuevo objeto geometría usando el promedio de las coordenadas, el color, las superficies normales, y la información de coordenadas de la textura de los objetos de **GeometryArray**. Modificar los pesos cambia la geometría resultante.  Todo lo que se requiere para utilizar un objeto **Morph** es crear el array de objetos **GeometryArray** y fijar los valores de carga. Abajo podemos ver una receta para usar un objeto **Morph**.   1. crear un array de objetos GeometryArray 2. crear un objeto Morph con ALLOW\_WEIGHTS\_WRITE 3. ensamblar el escenario gráfico, incluyendo la adición de los hijos de los objetos Switch fuentes   Como se puede ver, usar un objeto **Morph** no es duro; sin embargo, estos pasos de la receta no proporcionan ni animación ni interacción. La animación o la interacción se proporciona a través de un objeto **Behavior**. Por lo tanto, usar un objeto **Morph** significa generalmente escribir una clase **Behavior**.  Un objeto **Morph** se puede referir a un paquete de aspecto. El manojo de aspecto se utiliza con el objeto **GeometryArray** creado por el objeto **Morph**. Debemos tener cuidado ya que el objeto **Morph** crea siempre un objeto **GeometryArray** con colores-por-vertice. Por consiguiente, se ignorarán las especificaciones de **ColoringAttributes** y de color difuso de **Material**. Puedes ver el [Capítulo 6](http://www.labvis.unam.mx/elio/J3D/cursos/3d/cap_6.html) para más información sobre colorear y sombrear objetos visuales.  **Riesgos de Programación de Morph**  Incluso tan simple como es el uso de **Morph**, hay un riesgo potencial de programación asociado (no mencionado todavía). Pesos que no suman 1,0 resultan en un error en tiempo de ejecución. Ya hemos mencionado la limitación del aspecto.  **Ejemplo de Aplicación Morph: Walking**  Esta aplicación de **Morph** utiliza un objeto **Behavior** personalizado para proporcionar la animación. El primer paso de la receta es escribir el comportamiento personalizado.  En un comportamiento usado para animar un objeto **Morph**, el método processStimulus cambia los pesos del objeto **Morph**. Este proceso es solo tan complejo como necesario para alcanzar el efecto deseado de la animación o de la interacción. En este programa, el método processStimulus fija los valores de los pesos basándose en el valor de un objeto **alpha**. Esto sucede en cada marco de renderizado donde la condición del disparador se ha cumplido. El [Fragmento de Código 5-5](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento5-5) demuestra el código para el comportamiento de personalizado del programa [MorphApp.java](http://www.labvis.unam.mx/elio/J3D/clases/Animation/MorphApp.java). En este código, solamente el método processStimulus es interesante.  **Fragmento de Código 5-5, Clase MorphBehavior de MorphApp.**  1. public class MorphBehavior extends Behavior{  2.  3. private Morph targetMorph;  4. private Alpha alpha;  5. // the following two members are here for efficiency  6. private double[] weights = {0, 0, 0, 0};  7. private WakeupCondition trigger = new WakeupOnElapsedFrames(0);  8.  9. // create MorphBehavior  10. MorphBehavior(Morph targetMorph, Alpha alpha){  11. this.targetMorph = targetMorph;  12. this.alpha = alpha;  13. }  14.  15. public void initialize(){  16. // set initial wakeup condition  17. this.wakeupOn(trigger);  18. }  19.  20. public void processStimulus(Enumeration criteria){  21. // don't need to decode event since there is only one trigger  22. weights[0] = 0; weights[1] = 0; weights[2] = 0; weights[3] = 0;  23.  24. float alphaValue = 4f \* alpha.value(); // get alpha  25. int alphaIndex = (int) alphaValue; // which Geom obj  26. weights[alphaIndex] = (double) alphaValue - (double)alphaIndex;  27. if(alphaIndex < 3) // which other obj  28. weights[alphaIndex + 1] = 1.0 - weights[alphaIndex];  29. else  30. weights[0] = 1.0 - weights[alphaIndex];  31.  32. targetMorph.setWeights(weights);  33.  34. this.wakeupOn(trigger); // set next wakeup condition  35. }  36. } // end of class MorphBehavior  La clase **MorphBehavior** crea una animación de marcos usando dos objetos **GeometryArray** al mismo tiempo en un modelo cíclico. Esta clase es conveniente para cualquier animación de **morph** de cuatro marcos y se puede modificar fácilmente para acomodar otro número de marcos.  Con el comportamiento personalizado escrito, todo que lo resta es desarrollar los marcos para la animación. La Figura 5-21 muestra los dibujos a mano usados como los marcos para esta aplicación de ejemplo. Se podrían haber creado mejores marcos usando algún paquete 3D.  Figura 5-21, Imágenes para la aplicación MorphApp con el Trazo de un vértice.  Las figuras negras pueden parecer dos marcos, cada uno repetido una vez, pero en realidad, son cuatro marcos únicos. La diferencia está en el orden que se especifican los vértcies.  El [Fragmento de Código 5-6](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento5-6) presenta un extracto del método createSceneGraph. En este método se crean un objeto **MorphBehavior**, el objeto **alpha**, y un objeto **Morph** y se ensamblan en el escenario gráfico. Se crean los objetos del marco **GeometryArray** usando algunos otros métodos (no mostrados aquí). El código completo lo tienes en [MorphApp.java](http://www.labvis.unam.mx/elio/J3D/clases/Animation/MorphApp.java).  **Fragmento de Código 5-6, un extracto del método createSceneGraph de MorphApp.**  1. public BranchGroup createSceneGraph() {  2. // Create the root of the branch graph  3. BranchGroup objRoot = new BranchGroup();  4.  5. Transform3D t3d = new Transform3D();  6. t3d.set(new Vector3f(0f, -0.5f, 0f));  7. TransformGroup translate = new TransformGroup(t3d);  8.  9. // create GeometryArray[] (array of GeometryArray objects)  10. GeometryArray[] geomArray = new GeometryArray[4];  11. geomArray[0] = createGeomArray0();  12. geomArray[1] = createGeomArray1();  13. geomArray[2] = createGeomArray2();  14. geomArray[3] = createGeomArray3();  15.  16. // create morph object  17. Morph morphObj = new Morph(geomArray);  18. morphObj.setCapability(Morph.ALLOW\_WEIGHTS\_WRITE);  19.  20. // create alpha object  21. Alpha alpha = new Alpha(-1, 2000); // continuous 2 sec. period  22. alpha.setIncreasingAlphaRampDuration(100);  23.  24. // create morph driving behavior  25. MorphBehavior morphBehav = new MorphBehavior(morphObj, alpha);  26. morphBehav.setSchedulingBounds(new BoundingSphere());  27.  28. //assemble scene graph  29. objRoot.addChild(translate);  30. translate.addChild(morphObj);  31. objRoot.addChild(morphBehav);  32.  33. return objRoot;  34. } // end of CreateSceneGraph method of MorphApp  Es interesante observar que son posibles varias animaciones con los marcos creados para esta aplicación del ejemplo con diversas clases de comportamiento. La Figura 5-22 muestra una escena renderizada por Morph3App.  En este programa, otras tres clases de comportamiento crean animaciones basadas en alguno, o todos los objetos **GeometryArray** de **MorphApp**. Se llaman (de izquierda a derecha en la figura) "In Place", "Tango", y "Broken". No todas las animaciones son buenas. Por supuesto, para apreciar de verdad las animaciones, tenemos que ejecutar el programa.  Figura 5-22, una escena renderizada desde Morph3App mostrando las animaciones de tres clases de comportamientos alternativas (no todas buenas).  **El API Morph**  Con la simplicidad de uso de la receta anterior, podríamos esperar un API sencillo -- y así es.   |  | | --- | | Sumario de Constructores de **Morph**  Extiende: Node  Los objetos **Morph** crean un nuevo objeto **GeometryArray** usando el promedio de peso de los objetos **GeometryArray**. Si se proporciona un objeto de apariencia, se utiliza con la geometría resultante. Los pesos se especifican con el método setWeights. Un objeto **Morph** se utiliza generalmente con un objeto **Behavior** personalizado para ajustar los pesos en el tiempo de ejecución para proporcionar la animación (o la interacción).  Morph(GeometryArray[] geometryArrays)  Construye e inicializa un objeto Morph con el array de objetos GeometryArray especificado y un objeto Appearance null.  Morph(GeometryArray[] geometryArrays, Appearance appearance)  Construye e inicializa un objeto Morph con el array de objetos GeometryArray especificado y el objeto de apariencia especificado. |  |  | | --- | | Lista Parcial de Métodos de **Morph**  void setAppearance(Appearance appearance)  Selecciona el componente de apariencia de este nodo Morph.  void setGeometryArrays(GeometryArray[] geometryArrays)  Selecciona el componente geometryArrays de este nodo Morph.  void setWeights(double[] weights)  Selecciona el vector de pesos de ese nodo Morph. |  |  | | --- | | Sumario de Capacidades de **Morph**  ALLOW\_APPEARANCE\_READ | WRITE  Especfica que el nodo permite el acceso de lectura/escritura a su información de apariencia.  ALLOW\_GEOMETRY\_ARRAY\_READ | WRITE  Especfica que el nodo permite el acceso de lectura/escritura a su información de geometría.  ALLOW\_WEIGHTS\_READ | WRITE  Especfica que el nodo permite el acceso de lectura/escritura a su vector de pesos. | |  |  | |