|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **Java 3D**   * [Iluminación en Java 3D](about:blank)   + [Sombreado en Java 3D](about:blank)   + [Receta para Iluminar Objetos Visuales](about:blank)     - [Ejemplos de Luces Sencillas](about:blank)     - [Dónde Añadir un Objeto Light en un Escenario Gráfico](about:blank)   + [Clase Light](about:blank)     - [Luz Ambiente](about:blank)     - [Luz Direccional](about:blank)     - [Punto de Luz](about:blank)     - [SpotLight](about:blank)     - [Aplicaciones de Fuentes de Luz](about:blank)     - [Ejemplos de Iluminación](about:blank)   + [Objetos Material](about:blank)     - [Ejemplos sencillos de Material](about:blank)     - [Propiedades Geometry color, ColoringAttributes, y Material](about:blank)   + [Superficies Normales](about:blank)   + [Especificar la Influencia de las Luces](about:blank)     - [Alternativa a los Límites de Influencia: BoundingLeaf](about:blank)     - [Ámbito de Límites de Influencia de las Luces](about:blank)   + [Crear Objetos Brillantes-en-la-Oscuridad, Sombras y Otros Problemas de Iluminación](about:blank)     - [Objetos Brillantes-en-la-Oscuridad](about:blank)     - [Calcular Sombras](about:blank)     - [Crear Sombras](about:blank)     - [Programa de Ejemplo de Sombras](about:blank)     - [Tópico Avanzado: El Papel del Objeto View en el Sombreado](about:blank)     **Iluminación en Java 3D**  Este módulo presenta las técnicas para proporcionar detalles de los objetos visuales a través de sombras y texturas. Esta página explica el modelo de iluminación y cómo utilizar luces en Java 3D para conseguir sombras.  Java 3D sombrea los objetos visuales basándose en la combinación de sus características materiales y en las luces del universo virtual. El sombreado resulta de aplicar un modelo de iluminación a un objeto visual en presencia de fuentes de luz. La siguiente sección da una descripción del modelo de iluminación usado en el renderizador de Java 3D y cómo la luz interactúa con las características materiales para proporcionar sombreado. Cada una de las siguientes secciones explica las características relevantes del API Java 3D para el modelo de iluminación.  **Sombreado en Java 3D**  Sombrear objetos visuales en Java 3D depende de muchos factores. Esta sección proporciona una descripción abreviada del modelo de iluminación de Java 3D, del modelo de color, y del modelo de sombreado. La especificación del API Java 3D presenta una información más detallada sobre el modelo de iluminación de Java 3D. Como la mayor parte de la iluminación de Java 3D y del modelo de sombreado se basa en **OpenGL**, se puede encontrar más información en páginas sobre **OpenGL**.  **Modelo de Iluminación**  En el mundo real, los colores que percibimos son una combinación de las características físicas del objeto, de las características de las fuentes de luz, de las posiciones relativas de los objetos a las fuentes de luz, y del ángulo desde el cual se ve el objeto. Java 3D utiliza un modelo de iluminación para aproximar la física del mundo real. El resto de esta sección explica el modelo de iluminación de Java 3D en términos generales. La sección E.2 de la especificación del API Java 3D presenta las ecuaciones matemáticas para el modelo de iluminación Java 3D.  La ecuación del modelo de iluminación depende de tres vectores: la superficie normal (n), la dirección de la luz (l), y la dirección al ojo del espectador (e) además de las características materiales del objeto y de las características de la luz. La Figura 6-1 muestra los tres vectores para dos vértices de una superficie esférica. Los vectores para cada vértice pueden tener distintas direcciones dependiendo de específicidades de la escena. Cuando los vectores de la luz y del ojo varían, se cálculan en tiempo de ejecución. Por lo tanto, cada vértice de la esfera potencialmente se renderiza como una sombra diferente.  Figura 6-1, Vectores de Luz, Superficie Normal y Ojo del espectador usados para vértices sombreados.  El modelo de iluminación incorpora tres tipos de reflexiones de luz del mundo real: ambiente, difuso, y especular. La reflexión ambiente resulta de la luz ambiente, luz constante de bajo nivel, en una escena. La reflexión difusa es la reflexión normal de una fuente de luz desde un objeto visual. Las reflexiones especulares son las reflexiones sobreiluminadas de una fuente de luz sobre un objeto, que ocurren en ciertas situaciones.  La Figura 6-2 muestra una esfera y un plano renderizados por Java 3D. Los tres tipos de reflexión se pueden ver en la esfera de la Figura 6-2. La parte más oscura de la esfera exhibe sólo la reflexión ambiente. El centro de la esfera está iluminado por la luz difusa y ambiente. Con una esfera azul y una luz blanca, la reflexión difusa es azul. La parte más brillante de la esfera es el resultado de la reflexión especular con reflexiones ambiente y difusa.  Figura 6-2, Esfera sobreada y un Plano.  **Ojo Local contra Vectores de Ojos Infinitos**  Si cada vértice de cada objeto visual en una escena requiere un vector de luz, un vector del ojo, y el cálculo de la sombra, una porción significativa del cálculo de representación se utiliza en los vértices sombreados. La cantidad de cálculo puede reducirse si el vector de luz, o el vector del ojo, o ambos vectores son constantes. El vector de luz es constante cuando usa una luz direccional.. El vector del ojo es constante por defecto, aunque podemos especificar un vector variable del ojo usando un método del objeto **View**.  **Efectos inter-objetos no Considerados**  Mientras que el modelo de iluminación se basa en la física, los fenómenos físicos complejos no se modelan. Obviamente, la sombra echada por la esfera sobre el plano no está en la Figura 6-2. No tan obvio, también falta la luz reflejada de la esfera sobre el plano. También falta la luz reflejada desde el plano sobre la esfera que de nuevo se refleja en el plano... etcétera.  A menudo es difícil comprender la complejidad del cálculo de la acción de la luz. Consideremos la dificultad de calcular cómo cada gota del agua se comporta en una ducha. Las gotas vienen de la cabeza de la ducha en distintas direcciones. Cuando encuentran un objeto, la colisión que resulta produce muchas gotas más pequeñas que viajan en distintas direcciones. El proceso se repite muchas veces antes de que el agua se vaya por el desague. La complejidad de interacciones de la luz con los objetos visuales es muy semejante. Algunas de las diferencias entre los comportamientos del agua y la luz son que la luz no tiene ninguna adherencia (luz no se pega a los objetos visuales) y el efecto de la gravedad es insignificante para la luz.  Para reducir la complejidad del cálculo, el modelo de iluminación considera solamente un objeto visual al mismo tiempo. Consecuentemente, las sombras y las reflexiones inter-objetos no son renderizadas por el modelo de iluminación. Estos dos efectos requieren la consideración de todos los objetos junto con sus posiciones relativas en el momento de la representación. Se necesita considerablemente más cálculo para renderizar una sola escena con efectos inter-objetos. Java 3D, y el resto de los sistemas gráficos en tiempo real, no hacen caso de efectos inter-objetos en la representación. Algunos de los efectos ignorados del mundo real se pueden agregar a las escenas cuando sea necesario.  **Modelo de Color**  El modelo del color no está basado en la física. Java 3D modela el color de las luces y los materiales como una combinación de rojo, verde, y azul. El color blanco, como el color de la luz o del material, es la combinación de los tres componentes con la intensidad máxima. Cada luz produce un solo color de luz especificado por un tuple RGB. El modelo de iluminación se aplica a cada uno de los componentes del color RGB. Por ejemplo, una bola roja en presencia de una luz azul no será visible puesto que la luz azul no se refleja desde un objeto rojo. En realidad, el color es una combinación de muchas longitudes de onda de la luz, no solo tres. El modelo de color RGB representa muchos colores, pero no todos.  **Influencia de las Luces**  En Java 3D, la porción de una escena donde los objetos visuales son iluminados por una fuente de luz determinada se llama la región de influencia de ese objeto de luz. La región de influencia más simple para una fuente de luz usa un objeto **Bounds** y el método setInfluencingBounds() de **Light**. Cuando un objeto fuente de luz con límites de influencia intersecciona con los límites de un objeto visual, se utiliza la luz para sombrear todo el objeto. Los límites de influencian de una luz determinan qué objetos iluminar, no qué porciones de objetos se iluminan.  **Modelo de Sombreado**  El modelo de iluminación sombrea todos los vértice de un objeto visual por cada luz de influencia. La sombra de un vértice es la suma de las sombras proporcionadas por cada fuente de luz para la vértice. El resto de un objeto visual se sombrea basándose en la sombra de los vértices. El modelo de sombra de un objeto visual, especificado como atributo **Appearance NodeComponent**, determina cómo se hace el sombreado para el resto del objeto visual.  El **ColoringAttributes NodeComponent** especifica el modelo de sombra para los objetos visuales donde el modelo de sombra se especifica como uno de **SHADE\_GOURAUD**, **SHADE\_FLAT**, **FASTEST**, **NICEST**. Debemos tener cuidado ya que el color de un objeto **ColoringAttributes** nunca se utiliza en el sombreado.  En el sombrado **Gouraud**, cada pixel se sombrea con un valor derivado de la interpolación trilinear del valor de la sombra de cada vértice del polígono que lo encierra. En el sombreado plano, todos los pixeles de un polígono se asignan el valor de la sombra a partir de un vértice del polígono. La Figura 6-3 muestra una esfera sombreada plana y una esfera sombrada **Gouraud**. La ventaja del sombreado plano es la velocidad en la representación del software. El sombreado de **Gouraud** tiene la ventaja de la apariencia visual.  Figura 6-3, Esferas con Sombreado Plano y Gouraud.  Un último punto antes de ir a un ejemplo; los objetos fuentes de luz no son objetos visuales. Incluso si una fuente de luz se situa dentro de la vista, no será renderizada.  **Receta para Iluminar Objetos Visuales**  Se necesita una serie de pasos para permitir la iluminación de los objetos visuales en el universo virtual. Además de crear y de personalizar objetos para requisitos particulares de luz, cada objeto de luz debe ser agregado al escenario gráfico y haber especificado un objeto **Bound**. Cada objeto visual que se va a sombrear debe tener superficies normales y características de material. Abajo podemos ver la receta con los pasos necesarios:   1. Especificación de la fuente de Luz    1. seleccionar los límites    2. añadirla al escenario gráfico 2. Objeto Visual    1. superficies    2. propiedades de material   Si falta algunos de estos elementos no se podrá usar la iluminación. La presencia del objeto **Material** en un manojo **Appearance** de un objeto visual permite el modelo de iluminación para ese objeto. Sin el objeto **Material** el objeto visual será coloreado, pero no sombreado por el **ColoringAttribute** o los colores de vértice del objeto **Geometry**. Si ni uno ni otro están presentes, el objeto será blanco sólido. La Figura 6-5 muestra la excepción lanzada cuando un objeto visual tiene un objeto **Material** pero no tiene superficies normales.  javax.media.j3D.IllegalRenderingStateException: Cannot do lighting without  specifying normals in geometry object  Los errores con fuentes de luz pueden no ser fáciles de encontrar. No hay ningún aviso de que dejamos sin luz un escenario gráfico. Ni hay ninguna alerta por no fijar los límites de influencia de una fuente de luz. En cualquiera de estos casos, el objeto de luz no tendrá ninguna influencia sobre los objetos visuales en el escenario gráfico. Un objeto visual especificado correctamente para sombreado (es decir, uno con un objeto **Material**) en un escenario gráfico vivo pero fuera de los límites de influencia de todos los objetos fuente de luz se renderizará a negro.  Es posible especificar correctamente una escena en la cual un objeto visual con las características materiales influenciadas por un objeto de luz y que se renderice a negro. La orientación relativa de la luz, el objeto visual, y la dirección de la visión entran en juego en el renderizado.  **Ejemplos de Luces Sencillas**  Según lo mencionado arriba, crear renderizados con sombras implica la especificación apropiada de la fuente de luz y de los objetos visuales. Hasta el momento, ni la clase **Light** ni los objetos **Material** se han discutido en detalle. Sin embargo, aprovechándo los valores por defecto del API y sus características, podemos proceder a iluminar mundos virtuales. Los primitivos geométricos generan superficies normales cuando se solicitan. Los valores por defecto del objeto **Material** especifican un objeto visual razonable. Los valores por defecto de los constructores de la fuente de luz especifican fuentes de luz utilizables.  Usando la clase **SimpleUniverse** con los dos métodos del [Fragmento de código 6-1](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento6-1) se produce un universo virtual que incluye una sola esfera con las características materiales con sus valores por defecto iluminada por un solo objeto fuente de luz **AmbientLight**. El primer método del fragmento del código ensambla un objeto **Material** con un objeto **Apeareance** para la esfera. El segundo método crea un objeto **BranchGroup** para servir como la raíz de la rama de contenido gráfico, después agrega los objetos **Sphere** y **AmbientLight** al escenario gráfico. El objeto **Material** en el manojo del aspecto se agrega al objeto **Sphere** en la construcción de la esfera (líneas 12 y 13). Un valor por defecto **BoundingSphere** proporciona la región de influencia para el objeto **AmbientLight** (líneas 15 a 17). El diagrama del escenario gráfico de este mundo virtual aparece en La Figura 6-6.  **Fragmento de Código 6-1, Crear un Escena con un Esfera Iluminada.**  1. Appearance createAppearance() {  2. Appearance appear = new Appearance();  3. Material material = new Material();  4. appear.setMaterial(material);  5.  6. return appear;  7. }  8.  9. BranchGroup createScene (){  10. BranchGroup scene = new BranchGroup();  11.  12. scene.addChild(new Sphere(0.5f, Sphere.GENERATE\_NORMALS,  13. createAppearance()));  14.  15. AmbientLight lightA = new AmbientLight();  16. lightA.setInfluencingBounds(new BoundingSphere());  17. scene.addChild(lightA);  18.  19. return scene;  20. }  Las líneas 4 y 5 del [Fragmento de código 6-1](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento6-1) podrían ser reemplazadas por la siguiente línea que crea y usa un objeto **Material** anónimo.  Appear.setMaterial(new Material());  Los objetos **Material** son completamente personalizables con los parámetros de su constructor, simplificando el uso de objetos **Material** anónimos. Por el contrario, crear un objeto **Light** anónimo hace mucho más díficil la adicción de límites de influencia. Debemos tener en cuenta que el nombramiento del objeto **Material** puede hacer el objeto sea más fácil de compartir entre varios manojos del aspecto, dando como resultado un funcionamiento mejor.  El **SimpleUniverse** proporciona los objetos **VirtualUniverse** y **Locale** junto con el la rama de vista gráfica para el diagrama de escenario gráfico mostrado en la Figura 6-6. Sin una transformación, el objeto **Sphere** y el objeto **BoundingSphere** estarán centrados en el origen, y se interseccionarán. El objeto **Sphere** se sombrea por la fuente **AmbientLight**. La Figura 6-7 muestra la imagen que resulta con un fondo blanco. La especificación del fondo no se muestra en el código.  Figura 6-6, Diagrama de Escenario Gráfico del Fragmento de Código 6-1  La esfera de la Figura 6-7 es de color gris uniforme, que es el valor por defecto de la propiedad ambiente del material.  Figura 6-7, Una esfera con luz Ambiente.  La Figura 6-7 muestra que las escenas iluminadas unicamente con luz ambiente son opacas. Como la iluminación ambiente es uniforme, produce la sombra uniforme. La luz ambiente está pensada para llenar de luz una escena donde otras fuentes no iluminan. La adición de una fuente **DirectionalLight** hará esta escena más interesante.  Insertando el [Fragmento de código 6-2](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento6-2) en el [Fragmento de código 6-1](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento6-1) se añade un **DirectionalLight** a la rama de contenido gráfico de la escena. Una vez más los valores por defecto se utilizan para la fuente de luz, y se utiliza un **BoundingSphere** por defecto para la región de influencia. La Figura 6-8 muestra el diagrama del escenario gráfico que resulta sin los objetos proporcionados por el **SimpleUniverse**.  **Fragmento de Código 6-2, Añadir una Luz Direccional a la Escena.**  1. DirectionalLight lightD1 = new DirectionalLight();  2. lightD1.setInfluencingBounds(new BoundingSphere());  3. // customize DirectionalLight object  4. scene.addChild(lightD1);  Figura 6-8, Diagrama Abreviado del ejemplo de los fragmentos de código 6-1 y 6-2.  La Figura 6-9 muestra la imagen producida por la combinación de los dos fragmentos del código. La influencia del objeto **AmbientLight** apenas se puede ver con la fuente **DirectionalLight**. Obviamente, es necesaria la personalización de los requisitos particulares de los objetos de luz y/o las características materiales del objeto visual para crear escenas interesantes.  Figura 6-9, Una Esfera con fuentes de Luz ambiente y direccional.  **Dónde Añadir un Objeto Light en un Escenario Gráfico**  La influencia de un objeto de luz en el mundo no está afectada por la posición del objeto de luz en el escenario gráfico; sin embargo, el objeto **bounds** referenciado por la luz si lo está. El objeto **bounds** está sujeto a las coordenadas locales del escenario gráfico donde se inserta el objeto de luz. Consideremos la Figura 6-10 como ejemplo. El mismo objeto **BoundingSphere** referenciado por dos fuentes de luz proporciona dos regiones de influencia distintas debidas la traslación proporcionada por el objeto **TransformGroup**. El origen del sistema de coordenadas local del escenario gráfico debajo del **TransformGroup** está 2 metros por debajo del origen del mundo (Locale) y la otra región de la esfera de influencia.  Figura 6-10, un BoundingSphere Afectado por un Transformation  Si dependen o no, los objetos de la fuente de luz del escenario gráfico en la Figura 6-10 influencian el sombreado (luz) del objeto visual iluminado si los límites del objeto visual interseccionan con la región de influencia de los objetos de luz. Especificar la región de influencia de una luz como un sólo **bounds** podría no funcionar para todas las aplicaciones.  **Clase Light**  El API Java 3D proporciona cuatro clases para luces. Todas se derivan de la clase **Light**. La Figura 6-11 muestra la jerarquía de clases de Java 3D relacionada con las luces. **Light**, una clase abstracta, proporciona los métodos y las constantes de capacidades asociadas para manipular el estado, color, y los límites de un objeto **Light**. El estado de la luz es un boleano que activa y desactiva la luz.  Figura 6-11, El Árbol de Clases del API Java 3D con las clases Light  El siguiente bloque de referencia lista los métodos y las constantes de la clase **Light**. Debemos recordar que los límites seleccionados con setInfluencingBounds() activan una luz cuando el objeto **bounds** referenciado intersecciona con la vista.   |  | | --- | | Lista Parcial de Métodos de la Clase **Light**  **Light** es una clase abstracta que contiene variables de ejemplar comunes a todas las luces.  void setColor(Color3f color)  Selecciona el color actual de la luz.  void setEnable(boolean state)  Activa y desactiva la luz.  void setInfluencingBounds(Bounds bounds)  Selecciona los límites de influencia de la luz. |  |  | | --- | | Sumario de Capacidades de la Clase **Light**  ALLOW\_INFLUENCING\_BOUNDS\_READ | WRITE  ALLOW\_STATE\_READ | WRITE  ALLOW\_COLOR\_READ | WRITE. |   **Luz Ambiente**  Los objetos de luz ambiente proporcionan luz de la misma intensidad en todas las localizaciones y en todas las direcciones. Los objetos de luz ambiente modelan la luz reflejada desde otros objetos visuales. Si miramos la superficie inferior de nuestro escritorio, veremos la parte inferior del escritorio aunque ninguna fuente de luz esté dando directamente en esa superficie (a menos que tengamos una lámpara bajo el escritorio). La luz que brillaba hacia arriba en el fondo del escritorio se reflejó en el suelo y en otros objetos. En ambientes naturales con muchos objetos, la luz se refleja desde muchos objetos para proporcionar la luz ambiente. La clase **AmbientLight** de Java 3D simula este efecto.  El siguiente bloque de referencia enumera los constructores de la clase **AmbientLight**. La clase abstracta **Light** proporciona los métodos y las capacidades para esta clase (enumerada en el bloque de referencia anterior).   |  | | --- | | Sumario de Constructores de la clase **AmbientLight**  Un objeto fuente de luz ambiente proporciona la misma intensidad de luz en todas las localización y direcciones. Modela la compleja reflexión inter-objetos de la luz presente en escenas naturales.  AmbientLight()  Construye e inicializa un objeto fuente de luz ambiente usando los siguientes valores por defecto:   * lightOn true * color (1, 1, 1)   AmbientLight(Color3f color)  Construye e inicializa una luz ambiente usando los parámetros especificados.  AmbientLight(boolean lightOn, Color3f color)  Construye e inicializa una luz ambiente usando los parámetros especificados. |   Mientras que podría ser natural pensar que una fuente de luz ambiente se puede aplicar globalmente, esto no es necesariamente cierto en un programa Java 3D. La influencia de la fuente **AmbientLight** está controlada por sus límites igual que otras fuentes de luz Java 3D. Se pueden utilizar varios objeto fuente **AmbientLight** en un programa de Java 3D. No hay límite en el número de los objetos fuentes **AmbientLight** que se pueden utilizar.  Según lo mencionado en secciones anteriores, la sombra de un vértice es el resultado de las fuentes de luz, de las características materiales del objeto visual, y de su geometría relativa (distancia y orientación). Para las reflexiones ambiente, la geometría no es un factor. La propiedad ambiente del material sólo se utiliza para calcular la reflexión ambiente. El modelo de iluminación calcula la reflexión ambiente de la luz como el producto la intensidad del **AmbientLight** y la propiedad ambiente del material del objeto visual.  **Luz Direccional**  Una fuente **DirectionalLight** aproxima fuentes de luz muy distantes tales como el sol. Al contratrio que las fuentes **AmbientLight**, las fuentes **DirectionalLight** proporcionan una sola dirección al brillo de luz. Para los objetos iluminados con una fuente **DirectionalLight**, el vector de luz es constante.  La Figura 6-12 muestra dos vértices de la misma esfera que están siendo iluminados por una fuente **DirectionalLight**. El vector de luz es igual para estos dos y para todos los vértices. Compara La Figura 6-12 con la Figura 6-1 para ver la diferencia. Puesto que todos los vectores de luz de una fuente **DirectionalLight** son paralelos, la luz no se atenúa. En otras palabras, la intensidad de una fuente **DirectionalLight** no varía con la distancia al objeto visual y la fuente **DirectionalLight**.  Figura 6-12, El Vector de Luz es Constante para Fuentes DirectionalLight.  Los siguientes bloques de referencia listan los constructores y los métodos de **DirectionalLight**, respectivamente.   |  | | --- | | Sumario de Constructores de la Clase **DirectionalLight**  Los objetos **DirectionalLight** modelan fuentes de luz muy distantes teniendo una dirección del vector de luz constante  DirectionalLight()  Construye e inicializa una fuente direccional usando los siguientes valores por defecto:   * lightOn true * color (1, 1, 1) * direction (0, 0, -1)   DirectionalLight(Color3f color, Vector3f direction)  Construye e inicializa una luz direccional con el color y la dirección especificados. Por defecto el estado es true (on).  DirectionalLight(boolean lightOn, Color3f color, Vector3f direction)  Construye e inicializa una luz direccional con el estado, el color y la dirección especificados. |  |  | | --- | | Sumario de Métodos de la Clase **DirectionalLight**  void setDirection(Vector3f direction)  Selecciona la dirección de la luz.  void setDirection(float x, float y, float z)  Selecciona la dirección de la luz. |  |  | | --- | | Sumario de Capacidades de la Clase **DirectionalLight**  Además de las Capacidades heredadas de la clase Light, los objetos DirectionalLight tienen la siguiente capacidad:  ALLOW\_DIRECTION\_READ | WRITE |   Los **DirectionalLights** sólo participan en las porciones difusas y specular de la reflexión del modelo de la iluminación. Para las reflexiones difusas y specular, la geometría es un factor (al contrario que las reflexiones ambiente). Variar la dirección de la fuente de luz cambiará el sombreado de los objetos visuales. Solo las características materiales difusas y specular se utilizan para calcular las reflexiones difusas y specular.  **Punto de Luz**  Un **PointLight** es el contrario de un **DirectionalLight**. Es una fuente de luz omnidireccional cuya intensidad se atenúa con la distancia y tiene una localización. (un **DirectionalLight** no tiene ninguna localización, solo una dirección). Los objetos **PointLight** se aproximan a bombillas, velas, u otras fuentes de luz sin reflectores o lentes.  Un modelo de ecuación cuadrática modela la atenuación de las fuentes **PointLight**. La ecuación se encuentra en la sección E.2 de la especificación del API Java 3D. La Figura 6-13 ilustra la relación de un objeto **PointLight** con una esfera. Observa que los vectores de luz no son paralelos.  Figura 6-13, El vector de Luz varía para una fuente PointLight.  Los siguientes bloques de referencia listan los constructores y los métosos de **PointLight**, respectivamente.   |  | | --- | | Sumario de Constructores de la Clase **PointLight**  El objeto **PointLight** especifica una fuente de luz atenuada en el espacio que irradia la luz igualmente en todas las direcciones desde la fuente de luz.  PointLight()  Construye e inicializa una fuente de punto de luz usando los siguientes valores por defecto:   * lightOn true * color (1, 1, 1) * position (0, 0, 0) * attenuation (1, 0, 0)   PointLight(Color3f color, Point3f position, Point3f attenuation)  Construye e inicializa un punto de luz. Por defecto la luz está activa.  PointLight(boolean lightOn, Color3f color, Point3f position, Point3f  attenuation)  Construye e inicializa un punto de luz. |  |  | | --- | | Sumario de Métodos de la Clase **PointLight**  void setAttenuation(Point3f attenuation)  Selecciona los valores de atenuación actuales de la luz y los sitúa en el parámetro especificado. Los tres valores especificados en el objeto **Point3f** especifican los coeficientes constante, linear, y cuadrático, respectivamente.  1  atenuación = --------------------------------------------------------------------  constate+linear+cuadrático\*distancia2  donde distancia es la medida desde la fuente de luz al vértice que está siendo sombreado.  void setAttenuation(float constant, float linear, float quadratic)  Selecciona los valores de atenuación actuales de la luz y los sitúa en el parámetro especificado. Ver la ecuación anterior.  void setPosition(Point3f position)  Selecciona la posición de la Luz.  void setPosition(float x, float y, float z)  Selecciona la posición de la Luz. |  |  | | --- | | Sumario de Capacidades de la Clase **PointLight**  Además de las capacidades heredadas de la clase **Light**, los objetos **PointLight** tienen las siguientes capacidades.  ALLOW\_POSITION\_READ | WRITE  ALLOW\_ATTENUATION\_READ | WRITE |   Como **DirectionalLight**, **PointLight** participa solamente en las porciones difusa y de reflexión especular del modelo de iluminación. Para las reflexiones difusas y especular, la geometría es un factor. Variando la localización de un objeto **PointLight** se cambiará el sombreado de los objetos visuales en una escena.  **SpotLight**  **SpotLight** es una subclase de **PointLight**. La clase **SpotLight** agrega dirección y concentración a los parámetros de posición y de atenuación de **PointLight**. Los objetos **SpotLight** crean manualmente modelos de fuentes de luz artificiales como flashes , lámparas, y otras fuentes con reflectores y/o lentes.  La intensidad de la luz producida por una fuente **SpotLight** produce la luz dentro de un ángulo especificado desde la dirección de la luz. Si el vértice exterior se sale del ángulo de la extensión de la luz, entonces no se produce ninguna luz. Por dentro del ángulo de extensión, la intensidad varía mediante el ángulo y la distancia al vértice. Una vez más una ecuación cuadrática modela la atenuación debido a la distancia. El parámetro concentración y una ecuación diferente gobiernan la variación de la intensidad debido al ángulo. Las ecuaciones que gobiernan estos lazos se encuentran en la sección E.2 de la especificación del API Java 3D. La Figura 6-14 ilustra en 2D cómo la intensidad de luz varía desde una fuente **PointLight** en 3D.  Figura 6-14, La intensidad de Luz varía con la Distancia y la Orientación desde una fuente PointLight.  El ángulo de extensión de un objeto **SpotLight** podría hacer que la luz iluminara parte de un objeto visual. Esta es la única luz capaz de iluminar sólo una parte de un objeto visual.  Los siguientes bloques de referencia listan los constructores y métodos de **PointLight**, respectivamente.   |  | | --- | | Sumario de Constructores de la Clase **SpotLight**  **SpotLight** es una suclase de **PointLight** con los atributos de dirección, ángulo de extensión y concentración.  SpotLight()  Construye e inicializa una fuente de luz usando los siguientes valores por defecto:   * lightOn true * color (1, 1, 1) * position (0, 0, 0) * attenuation (1, 0, 0) * direction (0, 0, -1) * spreadAngle PI (180 degrees) * concentration 0.0   SpotLight(Color3f color, Point3f position, Point3f attenuation, Vector3f  direction, float spreadAngle, float concentration)  Construye e inicializa un punto de luz. Puedes ver el sumario de métodos de **PointLight** para más infromación sobre la atenuación. Por defecto la luz está activa.  SpotLight(boolean lightOn, Color3f color, Point3f position, Point3f  attenuation, Vector3f direction, float spreadAngle, float concentration)  Construye e inicializa un punto de luz. Puedes ver el sumario de métodos de **PointLight** para más infromación sobre la atenuación. |  |  | | --- | | Sumario de Métodos de la Clase **SpotLight**  Además de los métodos listados anteriormente para **PointLight**, la clase **SpotLight** tiene los siguientes métodos:  void setConcentration(float concentration)  Selecciona la concentración del punto de luz.  void setDirection(float x, float y, float z)  Selecciona la dirección de la luz.  void setDirection(Vector3f direction)  Selecciona la dirección de la luz.  void setSpreadAngle(float spreadAngle)  Selecciona el ángulo de exposición de la luz. |  |  | | --- | | Sumario de Capacidades de la Clase **SpotLight**  Además de las capacidades heredadas de la clase **Light**, los objetos **SpotLight** tienen las siguientes capacidades:  ALLOW\_SPREAD\_ANGLE\_READ | WRITE  ALLOW\_CONCENTRATION\_READ | WRITE  ALLOW\_DIRECTION\_READ | WRITE |   Como los objetos **DirectionalLight** y **PointLight**, los **SpotLights** participan solamente en las porciones difusas y reflexión specular del modelo de iluminación. Para las reflexiones difusas y specular, la geometría es un factor. Cambiar la localización o la orientación de una fuente de **SpotLight** cambiará el sombreado de los vértices dentro de la región de influencia de la luz.  **Aplicaciones de Fuentes de Luz**  Con todos los tipos de fuentes de luz, y la variedad de maneras de utilizarlas, veremos una pequeña guía de su uso típico en esta sección. En general, desearemos utilizar tan pocas fuentes de luz como se pueda para una aplicación dada. Cuántas son suficientes dependerá del efecto de iluminación deseado para la aplicación. El número de luces y la configuración de atributos es más una consideración artística que científica.  Desde un punto de vista artístico, a menudo es suficiente tener solo dos luces para una escena dada. Una luz proporciona la iluminación principal, la otra se utiliza para completar la cara más oscura de los objetos. La luz principal normalmente se coloca a la derecha del espectador, el relleno a la izquierda del espectador. Una vez más éstas son pautas generales para lo que pueda ser un diseño artístico complejo.  Normalmente se prefiere incluir fuentes de luz direccionales para la mayoría de las aplicaciones puesto que el cálculo requerido en la representación es perceptiblemente menor que para los puntos de luz. Las fuentes de puntos de luz se usan muy raramente debido a la alta complejidad de cálculo.  Es normal incluir una sola fuente de luz ambiente con una gran región de influencia. Esto iluminará las partes posteriores de los objetos (como la "cara oscura de la luna"). El valor por defecto del color funcionará razonablemente bien. El tiempo requerido para incluir la luz ambiente es pequeño comparado con otras fuentes de luz. Dejar fuera una luz ambiente puede ser muy sensible en algunas escenas, y no ser notado en absoluto en otras.  **Ejemplos de Iluminación**  La interacción de la luz con los objetos es muy compleja en la naturaleza. Incluso en el mundo virtual donde es menos complejo el modelo de la iluminación, las fuentes de luz son simplistas, y las superficies son menos detalladas, el efecto de una fuente de luz en un objeto visual es algo complejo. Esta sección presenta algunos ejemplos de la iluminación para ayudar a clarificar las características, capacidades, y las limitaciones del modelo de iluminación en Java 3D.  **Dos Luces Coloreadas**  La Figura 6-15 muestra una sola esfera blanca iluminada por dos fuentes de luz direccionales, una roja y una azul. Aunque puede sorprendernos, la sombra que resulta es magenta. Mezclar rojo y azul da lugar a la púrpura, que es el resultado en el sistema de color sustractivo. Mezclar luces rojas y azules resulta en magenta, los resultados de un sistema de color aditivo.  Figura 6-15 Un Esfera iluminada por dos fuentes de Luz Direccionales  En ausencia de luz, la esfera es negra. Si la única fuente de luz es roja, entonces la esfera aparecerá roja, o algo sombreada en rojo. Con la adicción de una fuente de luz azul, sólo son posibles el rojo, el azul y las mezclas de estos dos.  **Diferentes Patrones de Iluminación**  La siguiente aplicación ilustra las diferencia entre las fuentes de luz. En [LightsNPlanesApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/light/LightsNPlanesApp.java) se iluminan tres planos con una fuente de luz distinta. De izquierda a derecha, los objetos **DirectionalLight**, **PointLight**, y **SpotLight** iluminan los planos. La Figura 6-16 muestra la imagen renderizada por la aplicación.  Figura 6-16, Planos iluminados por diferentes fuentes de luz.  El **DirectionalLight** ilumina el plano uniformemente. El **PointLight**, situado directamente sobre el borde superior del plano del centro, ilumina el plano de forma irregular debido a la dirección variable de la luz con respecto a las superficies, y, en un grado inferior, a la atenuación de la luz. El **SpotLight**, también situado directamente sobre el centro de su plano, ilumina solamente una parte pequeña del tercer plano.  La Figura 6-17 ilustra la geometría implicada en la iluminación de los primeros dos planos. En la ilustración izquierda, los vectores de luz constantes de la fuente **DirectionalLight** en conjunción con los vectores normales constantes de un plano dan lugar a vectores constantes de la reflexión, e incluso de la iluminación del plano. En la ilustración derecha los vectores de luz variables de la fuente **PointLight** se combinan con los vectores normales constantes del plano dando por resultado las distintas direcciones para los vectores de reflexión, y una iluminación desigual del plano. El **SpotLight** es un caso especial de la fuente **PointLight** donde la influencia de la fuente de luz está limitada por el ángulo de la extensión.  Figura 6-17, Geometría de Planos Iluminados con Fuentes de Luces Direccionales y de Punto  **Concentración y Ángulo de Extensión de SpotLights**  La Figura 6-18 muestra las imágenes renderizadas a partir de versiones distintas del programa [ConcentrationApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/light/ConcentrationApp.java). Un plano es iluminado por nueve puntos de luz. Los valores del ángulo y de la concentración de la extensión para las luces de los puntos varían con la posición. El ángulo de la extensión varía por cada fila con valores de .1, .3 y .5 (radianes) desde la fila superior a la inferior, respectivamente. La concentración varía por cada columna con valores de 1,0, 50,0, y 100,0 desde la columna de la izquierda a la de la derecha, respectivamente.  Los valores de concentración no tienen ningún efecto para la fila superior, el ángulo de la extensión es el único factor. En la fila inferior, la concentración tiene un efecto para cada uno de los ángulos de la extensión. El azul en las imágenes es el color difuso del material.  Figura 6-18, Un Plano Iluminado por SpotLights con diferentes Concentraciones y Ángulos de Extensión.  [ConcentrationApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/light/ConcentrationApp.java) demuestra dos limitaciones del modelo de iluminación. La primera es el renderizado de los artefactos representados en la Figura 6-18. Artefactos similares son visibles en la Figura 6-16. Los modelos desiguales de iluminación para los planos verdes y rojos son debidos al pequeño número de vértices usados para representar los planos. Recordamos que el modelo de iluminación se aplica solamente en los vértices. Cuantos más vértices, mayor es el efecto de sombreado y más tardará en renderizarse.  La diferencia entre las imágenes izquierda y derecha de la Figura 6-18 es debido a la diferencia en el número de las vértices usados para representar el plano. La versión del programa que generó la imagen izquierda utilizó 16 veces más vértices que la que está a la derecha (2.500 vértices contra 40.000). Los artefactos de la imagen derecha son un resultado de la reducción de la densidad de vértices en la superficie y la triangulación impuesta por el sistema de renderizado de Java 3D.  **Limitar el Número de Luces**  La segunda limitación demostrada en [ConcentrationApp](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/light/ConcentrationApp.java) no se ve en la representación. El plano de **ConcentrationApp** son realmente cuatro objetos planos uno al lado de otro. Esto se hizo para superar una potencial limitación del sistema de representación subyacente. La especificación de **OpenGL** requiere soporte para ocho fuentes de luz simultáneas. Si el plano de **ConcentrationApp** fuera un objeto visual, entonces **OpenGL** limitaría el número de luces a ocho en algunas máquinas.  Usando los límites de influencia para seleccionar solamente las fuentes de luz relevantes para un objeto visual, Java 3D crea dinámicamente las especificaciones de iluminación para las luces mientras que se renderizan los objetos visuales. Mientras que ningún objeto sea iluminado por más de ocho luces, los programas de Java 3D no están limitados en el número de luces en un mundo virtual.  Por eso proporcionar al cuadro planos más pequeños y los límites apropiados para asegurarse de que ningún plano se ve influenciado por más de ocho luces, en el ejemplo parece que hay nueve luces (realmente diez, con la luz ambiente) iluminando un plano. Necesita un poco más programación, pero el programa que resulta es más portable. Mientras que muchas implementaciones de **OpenGL** utilizan más de ocho luces simultáneas, si estamos planeando distribuir nuestros programas, debemos tener en cuante esta limitación potencial.  En esta sección, algunos ejemplos muestran alguna de las características y las limitaciones de la iluminación Java 3D. La intención de esta sección es dar a los lectores algunos ejemplos de programas básicos y algunas figuras de ejemplo para comparar con sus propios programas. No es posible proporcionar ejemplos de cada posible situación de iluminación, pues los factores en la representación son demasiado diferentes.  Una última cosa, **PointLight** y **SpotLight** utilizan la especificación de atenuación. La atenuación se especifica por los términos constantes en la ecuación cuadrática inversa basada en la distancia entre la luz y el vértice (véase el bloque de la referencia anterior). Encontrar la atenuación apropiada para una aplicación específica es un problema artístico. No se incluye ningun programa de ejemplo de atenuación en este tutorial.  **Objetos Material**  Las características materiales de un objeto visual se especifican en el objeto **Material** de un manojo de aspecto. **Material** es una subclase de **NodeComponent**. La Figura 6-19 muestra la jerarquía de clases del API Java 3D para **Material**.  Figura 6-19, Árbol de Clases de la Clase Material  El objeto **Material** especifica colores ambiente, difusos, especular, y emisivo y un valor de brillantez. Cada uno de los tres primeros colores se utiliza en el modelo de iluminación para calcular la reflexión correspondiente. El color emisivo permite que los objetos visuales "brillen intensamente en la oscuridad". El valor de brillantez se utiliza solamente para calcular reflexiones especulares.  Los siguientes bloques de referencia enumeran los constructores y los métodos de la clase **Material**.   |  | | --- | | Sumario de Constructores de la Clase **Material**  El objeto **Material** define la aparienia de un objeto bajo la iluminación.  Material()  Construye e inicializa un objeto **Material** usando los siguientes valores por defecto:   * ambientColor (0.2, 0.2, 0.2) * emissiveColor (0, 0, 0) * diffuseColor (1, 1, 1) * specularColor (1, 1, 1) * shininess 0.0   Material(Color3f ambientColor, Color3f emissiveColor, Color3f diffuseColor,  Color3f specularColor, float shininess)  Construye e inicializa un nuevo objeto **Material** usando los parámetros especificados. |  |  | | --- | | Lista Parcial de Métodos de la Clase **Material**  void setAmbientColor(Color3f color)  Selecciona el color ambiente de este **Material**.  void setAmbientColor(float r, float g, float b)  Selecciona el color ambiente de este **Material**.  void setDiffuseColor(Color3f color)  Selecciona el color difuso de este **Material**.  void setDiffuseColor(float r, float g, float b)  Selecciona el color difuso de este **Material**.  void setDiffuseColor(float r, float g, float b, float a)  Selecciona el color difuso más alpha de este **Material**.  void setEmissiveColor(Color3f color)  Selecciona el color emisivo de este **Material**.  void setEmissiveColor(float r, float g, float b)  Selecciona el color emisivo de este **Material**.  void setLightingEnable(boolean state)  Activa o desactiva la iluminación de objetos visuales que referencian este objeto.  void setShininess(float shininess)  Selecciona la brillantez de este **Material**.  void setSpecularColor(Color3f color)  Selecciona el color especular de este **Material**.  void setSpecularColor(float r, float g, float b)  Selecciona el color especular de este **Material**.  java.lang.String toString()  Devuelve una representación String de los valores de este **Material**. |  |  | | --- | | Sumario de Capacidades de la Clase **Material**  Además de las Capacidades heredadas de **NodeComponent**, los objetos **Material** tienen la siguiente capacidad:  ALLOW\_COMPONENT\_READ | WRITE  Permite ller/escribir información de los campos individuales del componente. |   **Ejemplos sencillos de Material**  Las reflexiones especulares ocurren naturalmente en los objetos lisos. En general, cuanto más lisa sea una superficie, más definida e intensa es la reflexión especular. Cuando una superficie es suficientemente lisa, actúa como un espejo que refleja la luz sin cambiar el color de la luz. Por lo tanto, el color especular de un objeto normalmente es blanco. Cambiamos el color specular de un **Material** para alterar la intensidad de una reflexión specular (por ejemplo, Color3f(0.8f, 0.8f, 0.8f)).  El valor de brillantez controla el rango de la extensión del ángulo de la visión para el cual se puede ver una reflexión especular. Una brillantez más alta resulta en reflexiones especulares más pequeñas. La Figura 6-20 muestra nueve esferas distintas iluminadas por una fuente de luz. Cada esfera tiene un valor de brillantez distinto.  Figura 6-20, Diferentes Propiedades de Material  Un objeto **Material** se asocia a un objeto visual a través de un objeto **Apearance** de la misma manera que lo hacen los atributos del aspecto. El método SetMaterial() de la clase **Appearance** referencia un objeto **Material** para ese objeto **Appearance**.  **Propiedades Geometry color, ColoringAttributes, y Material**  Hay tres maneras de especificar el color para un objeto visual: color por-vértice especificado en la geometría con los métodos getColor(), **ColoringAttributes** de un nodo **Appearance**, y el objeto **Material**. Java 3D permite que creemos objetos visuales sin usar ninguna, alguna, o las tres formas de especificar color.  Cuando se ha hecho más de una especificación del color, dos sencillas reglas determinan qué especificación del color toma la precedencia.   * Color **Material** se utiliza solamente cuando la representación ilumina objetos y color de **ColoringAttributes** sólo se utiliza cuando se renderizan objetos no iluminados. * Geometría por-vértice siempre tiene precedencia sobre **ColoringAttributes** o **Material**.   Las reglas pueden ser más claras cuando el problema se divide en objetos iluminados o apagados. La iluminación está activa para un objeto cuando se referencia un objeto **Material**. Inversamente, cuando no se asocia ningún objeto **Material** al objeto visual, la iluminación está desactivada para ese objeto. Observa que una escena puede tener tanto objetos iluminados como apagados.  Cuando la iluminación está activa para un objeto (es decir, se referencia un objeto **Material**), se utilizan el color material o el color de la geometría por-vértice para sombrear. Si esta presente, el color por-vértice reemplaza los colores de **Material** difusos y ambiente. Observa que el color de **ColoringAttributes** nunca se utiliza para la iluminación de objetos. La siguiente Tabla resume las relaciones:   |  |  |  | | --- | --- | --- | | **Color de Geometry por Vértice** | **Color ColoringAttributes** | **Resultado** | | NO | NO | Color Material | | SI | NO | Color Geometry | | NO | SI | Color Material | | SI | SI | Color Geometry |   Cuando la iluminación está desactivada para un objeto (es decir, no se referencia un objeto **Material**), se usan el color de **ColoringAttributes** o el color de por-vértice para colorear. Si está presente, el color de la geometría por-vértice reemplaza el color de **ColoringAttributes**. La siguiente Tabla resume las relaciones.   |  |  |  | | --- | --- | --- | | **Color Geometry por Vértice** | **Color ColoringAttributes** | **Resultado** | | NO | NO | blanco plano | | SI | NO | Color Geometry | | NO | SI | ColoringAttributes | | SI | SI | Color Geometry |   **Superficies Normales**  Según lo mencionado en secciones anteriores, las superficies normales son necesarias para sombrear los objetos visuales. Al crear objetos visuales usando clases **Geometry**, utilizamos uno de los métodos setNormal() para especificar los vectores de los vértices.  El **NormalGenerator** incluido con los utilidades de Java 3D genera superficies normales al especificar los objetos visuales que usan objetos **GeometryInfo**. Para generar superficies normales, ponemos nuestro objeto visual **Geometry** y llamamos a NormalGenerator.generateNormals().  Los primitivos geométricos generan sus propias superficies normalea cuando son especificados.  No importa cómo se especifican (o se generan) las superficies normales, sólo se especifica una superficie normal por vértice. Esto conduce a algunos problemas interesantes. Por ejemplo, cuando las dos superficies normales de polígonos son visibles, la normal es solamente correcta para una de las superficies normales. El resultado es que las caras posteriores sean renderizadas (si se renderizan) solamente con las características materiales de ambiente. Las reflexiones difusa y especular requieren la especificación normal apropiada.  Este problema común se soluciona especificando caras normales detrás al contrario que las superficies normales delanteras. Utilizamos el método setBackFaceNormalFlip() de un objeto **PolygonAttributes** para este propósito.  La Figura 6-21 muestra dos imágenes sombreadas de una tira doblada. La imagen de la izquierda fue renderizada desde la superficie frontal, y la derecha muestra las superficies normales traseras.  Figura 6-21, Tira Doblada con y son superfices traseras  Cuando un vértice es compartido por las superficies normales o varian las orientaciones, tener solamente una superficie normal por vértice puede dar lugar a problemas. Consideremos los ejemplos ilustrados en la Figura 6-22. La geometría ilustrada en la cara del lado izquierdo de la Figura 6-22 muestra la sección transversal de una superficie donde cada polígono se orienta a un ángulo de 90° de sus vecinos. Si se selecciona la superficie normal como el normal actual para una superficie, es muy incorrecto para su vecino. Si las superficies normales se especifican según lo mostrado, entonces la superficie estará sombreada constantemente entre los vértices con superficies paralelas. Un problema similar ocurre con la geometría del cubo mostrada en la cara derecha en la Figura 6-22. La solución a ambos problemas es aumentar el número de vértices para aumentar el número de superficies normales. Esto, por supuesto, aumenta el uso de la memoria y el tiempo de la renderización.  Figure 6-22 , Geometry con Problemas de Especificación de Normal  **Especificar la Influencia de las Luces**  En ejemplos anteriores, la especificación de los límites que influencian un objeto de luz se consigue al referirse a un objeto **Bounds**. Esto conecta la localización de los límites que influencian a la localización de la luz. (En secciones anteriores se explicó cómo las transformaciones en el escenario gráfico afectan a los volúmenes de limites usados para especificar los límites de influencia de las luces.) Mientras que esto se hace trivial para mover luces junto con los objetos visuales que se iluminan, otras aplicaciones necesitan una especificación más flexible de la influencia de luces. Afortunadamente, el API Java 3D proporciona un método alternativo para especificar los límites de influencia y una manera de limitar el ámbito en adición de los límites.  **Alternativa a los Límites de Influencia: BoundingLeaf**  Un objeto **BoundingLeaf** es un alternativa a un objeto **Bounds** de influencia. Un objeto **BoundingLeaf** es referido por otros nodos de la hoja para definir una región de influencia. Como descendiente de **SceneGraphObject**, los ejemplares de **BoundingLeaf** se agregan al escenario gráfico. El objeto **BoundingLeaf** está sujeto al sistema de coordenadas local de su posición en el escenario gráfico, que podría ser independiente del sistema de coordenadas del objeto de luz. Es decir, usar un **BoundingLeaf** permite a una luz y a sus límites de influencia moverse independientemente.  Una llamada a setInfluencingBoundingLeaf() para un objeto de luz especifica el argumento **BoundingLeaf** como los límites de influencia de la luz. Esta especificación reemplaza cualquier especificación regional de los límites de influencia. Un objeto **BoundingLeaf** puede ser compartido por varios objetos de luz.  La Figura 6-23 muestra el diagrama del escenario gráfico para una aplicación de ejemplo de un objeto **BoundingLeaf** con objetos de luz. En esta escena, se mueven dos luces junto con un objeto **TransformGroup** (a la derecha). Estas luces podían ser ejemplares de **PointLight** o de **SpotLight**. Sin embargo, la influencia de estas luces no cambia cuando las luces se mueven. La influencia de las luces se mueve cuando el **TransformGroup** izquierdo cambia la localización del objeto **BoundingLeaf**. Podemos comparar este diagrama del escenario gráfico con el que está en la Figura 6-10.  Figura 6-23, Mover un Objeto Light Independientemente de su Influencia usando un BoundingLeaf.  En la Figura 6-10, si se mueve la luz, su región de influencia también se mueve. También, según lo demostrado en la Figura 6-10, la región de influencia de dos luces que comparten el mismo objeto **Bounds** pueden o no pueden tener la misma región de influencia. Cuando dos o más luces comparten el mismo objeto **BoundingLeaf**, tienen siempre la misma región de la influencia.  **Ámbito de Límites de Influencia de las Luces**  Una región de limites, con un objeto **Bounds** o un objeto **BoundingLeaf**, especifica la región de influencia de un objeto de luz. Un ámbito especificado puede además limitar la influencia de una luz a una porción del escenario gráfico. Como valor por defecto, todas las luces tienen el alcance del mundo virtual en el cual reside. La adición de una especificación del alcance reduce además la influencia de una luz a los objetos visuales en el escenario gráfico debajo del **group(s)** especificado. Por ejemplo, consideremos la aplicación siguiente.  **Ejemplo de Ámbito de Iluminación**  La escena consiste en una lámpara y algunos objetos visuales en una mesa. La lámpara tiene una sombra, por eso no todos los objetos, ni toda la mesa, debe ser iluminada por la lámpara. El interior (pero no el exterior) de la lámpara también se debe iluminar (en este ejemplo, la sombra de la lámpara es completamente opaca). Sabemos que Java 3D no proporcionará la obstrucción por nosotros. Usando sólo un volumen de limitación, la influencia de la luz puede ser controlada, pero podría ser muy difícil, especialmente si se iluminan y apagan objetos que están uno cerca del otro, o se mueven.  Especificar un ámbito de limitaciones para la luz nos permite controlar limitaciones complejas de la influencia más fácilmente. La única consideración es mantener los objetos iluminados y apagados en partes separadas del escenario gráfico. Nuestro pensamiento inicial pudo ser comenzar a construir el escenario gráfico **BranchGroups** para los objetos iluminados y apagados, pero éso no es a menudo necesario ni recomendado para la mayoría de las aplicaciones.  Figura 6-24, Dos posibles Escenarios Gráficos para el Ejemplo de Ámbito de Luz.  El diagrama del escenario gráfico de la izquierda de la Figura 6-24 muestra un acercamiento nativo a la construcción del escenario gráfico. La organización no es natural y será difícil de manipular en una aplicación animada. Por ejemplo, si la mesa se mueve, la lámpara y otros objetos deben moverse con ella. En el escenario gráfico de la izquierda, mover la mesa (mediante la manipulación de **TransformGroup**) no moverá la lámpara o el rectángulo ilumiando; solamente el rectángulo apagado se moverá con la mesa.  El diagrama del escenario gráfico de la derecha representa una organización más natural para la escena. Los objetos en la mesa son hijos del **TransformGroup** que coloca la mesa. Si la mesa se mueve (mediante la manipulación de **TransformGroup**) los objetos de la mesa se moveran con ella.  La escena de ejemplo se crea en [LightScopeApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/light/LightScopeApp.java). La Figura 6-25 muestra dos imágenes renderizadas del programa de ejemplo. La imagen izquierda utiliza ámbito de luz para limitar la influencia de la luz de la lámpara a la lámpara y el rectángulo iluminado. La imagen derecha no utiliza scoping; por lo tanto, la luz de la lámpara ilumina el ' rectángulo no iluminado'.  El área brillante debajo de la lámpara (no representada en ningún diagrama del escenario gráfico) es un polígono situado justo sobre la tapa de la mesa. Este polígono brillante representa la parte de la mesa que es iluminada por la lámpara. El área brillante aparece más ligera que el resto de la mesa (incluso en la imagen derecha de la Figura 6-9) porque sus superficies se alinean más cercanas al punto de luz de la lámpara.  La sombra no aparece iluminada en ninguna imagen de la Figura 6-25 porque su característica **Material** difusa es negra. La sombra puede crearser con el uso del scoping solamente si un nodo adicional del grupo que se utiliza en el escenario gráfico.  La sombra en esta escena fue creada a mano. Las técnicas para crear sombras automáticamente (incluso dinámicamente) se discuten en la siguiente sección.  Figura 6-25, Ejemplo de Escena con y sin ámbito de Luz.  Tampoco se representan en ningún diagrama del escenario gráfico las tres fuentes de luz adicionales: dos fuentes de luz direccionales y una fuente de luz ambiente. Éstas son necesarias para simular la luz de una escena natural.  El siguiente bloque de referencia muestra los métodos de la clase **Light** usados para especificar limitaciones del scoping y el uso de los objetos **BoundingLeaf** para especificar los límites de influencia.   |  | | --- | | Lista Parcial de Métodos de la Clases **Light**  Otros métodos de la clase **Light** aparecieron en secciones anteriores.  void addScope(Group scope)  Añade el ámbito especificado a la lista de ámbitos de este nodo.  java.util.Enumeration getAllScopes()  Devuelve un objeto Enumeration con todos los ámbitos.  void insertScope(Group scope, int index)  Inserta el ámbito especificado por el nodo grupo en el índice especificado.  int numScopes()  Devuelve un contador con los ámbitos de luces.  void removeScope(int index)  Elimina el ámbito del nodo en la posición de índice especificada.  void setInfluencingBoundingLeaf(BoundingLeaf region)  Selecciona la región de influencia de la luz al **BoundingLeaf** especificado. Seleccionar un **BoundingLeaf** sobreescribe un objeto **Bounds**.  void setScope(Group scope, int index)  Selecciona el ámbito de herencias en el índice especifiado. Por defecto las luces tienen ámbitos sólo para los límites de su región de influencia. |   Otra ventaja de usar alcances para limitar la influencia de una luz: puede reducir el tiempo de renderizado. Calcular la intersección de los límites para un objeto visual con los límites que influencian de una luz es más complejo que determinar el alcance de una luz. Debemos tener cuidado con que ni el uso de los límites de influencia ni los alcances limitará la influencia de una luz a una parte de un objeto visual.  **Crear Objetos Brillantes-en-la-Oscuridad, Sombras y Otros Problemas de Iluminación**  Las secciones anteriores cubren las aplicaciones típicas de iluminación en Java 3D. Esta sección cubre algunas de las características y técnicas menos utilizadas.  **Objetos Brillantes-en-la-Oscuridad**  El objeto **Material** permite la especificación de un color emisivo. Esto se puede utilizar para crear el efecto de un objeto brillante en la oscuridad. Tener un color emisivo no hace del objeto visual una fuente de luz; no iluminará otros objetos visuales. El **Material** Emisivo es también útil en aplicaciones especiales, tales como indicar un objeto especial o un objeto que se ha escogido.  La Figura 6-26 muestra la escena del programa del ámbito de luz donde se le ha dado color emisivo al rectángulo no iluminado. Comparemos esta imagen con la imagen izquierda de la Figura 6-25. Como podemos ver, el uso del color emisivo sólo modifica al objeto visual que lo tiene. Este ejemplo también demuestra que el uso eficaz del color emisivo, como con la mayoría de los parámetros de la iluminación, es más un problema artístico que de programación.  Figura 6-26, Un ejemplo de objeto brillante en la oscuridad.  **Calcular Sombras**  La complejidad de calcular sombras es tan grande que no forma parte de ningún sistema de gráficos en tiempo real. La complejidad viene de cálcular si la fuente de luz alcanza o no un vértice. Todo polígono de otro objeto visual debe ser considerado al calcular la respuesta.  Figura 6-27, Proyectar la Sombra de un Objeto visual hacia un Plano con una Luz  El sombreado es mucho más complejo en realidad. Las fuentes de luz no son fuentes puramente direccionales ni perfectas. Por lo tanto, las sombras no tienen bordes sostenidos. Ignorando la realidad, como hacemos a menudo en gráficos, echemos una ojeada a las formas de simular sombras.  **Crear Sombras**  Hay dos partes básicas al simular (o al falsificar) sombras: calcular donde están las sombras, y crear geometrías (o texturas) para servir como sombras. Hay varias maneras de calcular la localización de la sombra, pero los detalles de las distintas técnicas de sombreado están más allá del alcance de esta guía. Las dos secciones siguientes cubren dos técnicas generales para crear el contenido de la sombra.  **Sombrear Polígonos**  Un polígono especificado sin propiedades **Material** se puede utilizar como polígono de sombra, llamado un polígono sombra coloreado. El color del polígono sombra, especificado por geometría o con un objeto **ColoringAttributes**, se elige para aparecer como el objeto en sombra. Los polígonos sombra especificados de esta manera pueden parecer falsos en escenas complejas.  Los polígonos sombra especificados con las características **Material** pero fuera de la influencia de uno o más objetos de luz se llaman polígonos sombra sombreados. Los polígonos sombra son sombreados por los objetos de luz que los influencian tal que parecen más realistas. Obviamente, especificar un polígono sombra sombreado es más complejo que especificar un polígono sombra coloreado.  No importa cómo se especifique un polígono sombra, la posición del polígono sombra es justo arriba, o en frente de, el polígono al que da sombra. Mientras que la adición de polígonos sombra no da lugar normalmente a más polígonos para renderizar (debido a la obstrucción de otros polígonos) crea más objetos en el universo virtual lo que puede degradar el funcionamiento de la renderización.  En vez de crear los polígonos sombra, las sombras pueden crearse cambiando la influencia de luces para excluir polígonos 'en la sombra'. El ámbito de luces es útil para este propósito. Sin embargo, puesto que la influencia se determina en base al objeto, puede ser complejo calcular cómo subdividir los objetos visuales que se sombrean parcialmente.  **Sombrear Texturas**  Como las sombras anteriores, las sombras naturales son complejas. Una sombra natural raramente tiene un borde recto y una sombra constante. Se puede usar el texturado para hacer sombras más realistas. Hay dos maneras básicas de usar texturado para crear sombras: aplicando textura a los polígonos sombra, o la aplicación de texturas a los objetos visuales.  Como el texturado no se ha cubierto todavía ([Capítulo 7](http://www.labvis.unam.mx/elio/J3D/cursos/3d/cap_7.html)), y el cálculo de las texturas de la sombra (incluso off-line) es difícil (y va más allá del alcance de esta guía) este es un tema pendiente para otro libro.  **Mover Objetos, Mover Sombras**  Debemos tener presente que la adición de sombras a una aplicación hace la aplicación mucho más compleja. Por ejemplo, cuando un cubo con una sombra gira, la sombra gira y se deforma. Para esa materia, las luces móviles hacen que las sombras se muevan también. En cualquier caso, el movimiento agrega otro nivel de complejidad a la programación de sombras.  **Programa de Ejemplo de Sombras**  El programa [ShadowApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/light/ShadowApp.java) da un ejemplo de cómo se pueden crear polígonos sombra sencillos. El programa define una clase para crear los polígonos sombra. La clase sombra crea un polígono sombra para cada geometría dada como entrada de información. El [Fragmento de Código 6-3](http://www.labvis.unam.mx/elio/J3D/cursos/3d/#fragmento6-3) muestra la clase **SimpleShadow** usada para crear polígonos sombra en [ShadowApp.java](http://www.labvis.unam.mx/elio/J3D/cursos/3d/clases/light/ShadowApp.java). La Figura 6-28 muestra la escena renderizada con una sombra.  **Fragmento de Código 6-3 Clase Shadow para Crear Polígonos Sombreados.**  1. public class SimpleShadow extends Shape3D {  2. SimpleShadow(GeometryArray geom, Vector3f direction,  3. Color3f col, float height) {  4.  5. int vCount = geom.getVertexCount();  6. QuadArray poly = new QuadArray(vCount, GeometryArray.COORDINATES  7. | GeometryArray.COLOR\_3  8. );  9.  10. int v;  11. Point3f vertex = new Point3f();  12. Point3f shadow = new Point3f();  13. for (v = 0; v < vCount; v++) {  14. geom.getCoordinate(v, vertex);  15. shadow.set( vertex.x + (vertex.y-height) \* direction.x,  16. height + 0.0001f,  17. vertex.z + (vertex.y-height) \* direction.y);  18. poly.setCoordinate(v, shadow);  19. poly.setColor(v, col);  20. }  21.  22. this.setGeometry(poly);  23. }  Varias asunciones hechas en la clase **SimpleShadow** (para hacerlo fácil) limitan las aplicaciones de esta clase. **SimpleShadow** está limitada en que: proyecta solamente a los planos, sólo considera una luz, sólo hace algunas orientaciones, no considera las dimensiones del plano sobre el que se está proyectando. Escribir una clase de fines generales para el cálculo de la sombra es una empresa importante.  Figura 6-28, Escena Producida por ShadowApp.java Demostrando el sombreado automático en Java 3D  **Tópico Avanzado: El Papel del Objeto View en el Sombreado**  La vista (o las vistas) asociadas a un escenario gráfico juegan una gran variedad de papeles en cómo se renderiza una escena. Esta sección no explica todos los papeles del objeto **View**. La especificación del API Java 3D proporciona una referencia completa a la clase **View**. Esta sección menciona solamente dos métodos de la clase **View** útiles para entender el sombreado de objetos visuales.  Según lo mencionado en la sección "Vectores de ojo local contra ojos infinito", el vector del ojo es constante como valor por defecto. Esto se conoce como un vector de ojo infinito. Es decir, la escena se renderiza como si fuera vista desde el infinito. Tener un ojo infinito reduce perceptiblemente el cálculo de renderización.  Sin embargo, la imagen que resulta puede parecer incorrecta. La Figura 6-29 muestra las imágenes renderizadas a partir de una escena usando un ojo infinito y un ojo local usando diversas fuentes de luz.  Figura 6-29, Ojo Infinito Frente a Ojo Local en el Renderizado con Fuentes de Luz Puntuales y Direccionales.  Para apreciar completamente las imágenes de la Figura 6-29 necesitamos conocer la geometría de la escena. La escena son nueve esferas en una organización planar. Cada una de las imágenes se ve con el mismo campo visual desde la misma posición. Las únicas variables son si la luz es un **DirectionalLight** o un **PointLight**, y si el ojo es infinito o local. El **DirectionalLight** tiene dirección (0, 0, -1), el **PointLight** se coloca en (0, 0, 1).  Las imágenes (a) y (c) de la Figura 6-29 se renderizan con un ojo infinito. En estas imágenes, los vectores del ojo son constantes, así que las reflexiones especulares están básicamente en la misma posición para cada esfera. Las imágenes (b) y (d) de la Figura 6-29 se renderizan con un ojo local. Los vectores del ojo varían en estas imágenes, así que las reflexiones especulares están en distinta posición para cada esfera. Observemos también que la reflexión difusa (azul) en las esferas varía sólo con la fuente de luz. El vector del ojo sólo desempeña un papel en el cálculo de la reflexión especular.  Una vez más la característica de la visión del ojo infinito se utiliza para reducir el cálculo, y por lo tanto el tiempo de la renderización. La imagen (a) de la Figura 6-29 tarda un menor tiempo para renderizarse y la imagen (d) tarda el mayor tiempo. Las imágenes (b) y (c) tardan una cantidad casi igual de tiempo, que es menor que el tiempo de la imagen (d), pero mayor que el de la imagen (a). El tiempo real para renderizar varía con el sistema utilizado. La diferencia es más pronunciada en los sistemas que renderizan por software.   |  | | --- | | Lista Parcial de Métodos (Relacionados con el Sombreado) de la Clase **View**  El objeto **View** contiene todos los parámetros necesarios para rendereizar una escena tridimensional desde un punto de vista.  void setLocalEyeLightingEnable(boolean flag)  Selecciona una bandera que india si se usa el ojo local para calacular las proyecciones de perspectivas.  void setWindowEyepointPolicy(int policy)  Selecciona la política del modelo de vista de ojo de la ventana a uno de :   * RELATIVE\_TO\_FIELD\_OF\_VIEW, * RELATIVE\_TO\_SCREEN, RELATIVE\_TO\_WINDOW |   El objeto **View** se puede conseguir desde un **SimpleUniverse** usando los métodos apropiados. Entonces el objeto **View** se puede manipular como en el siguiente ejemplo: |  |  | |