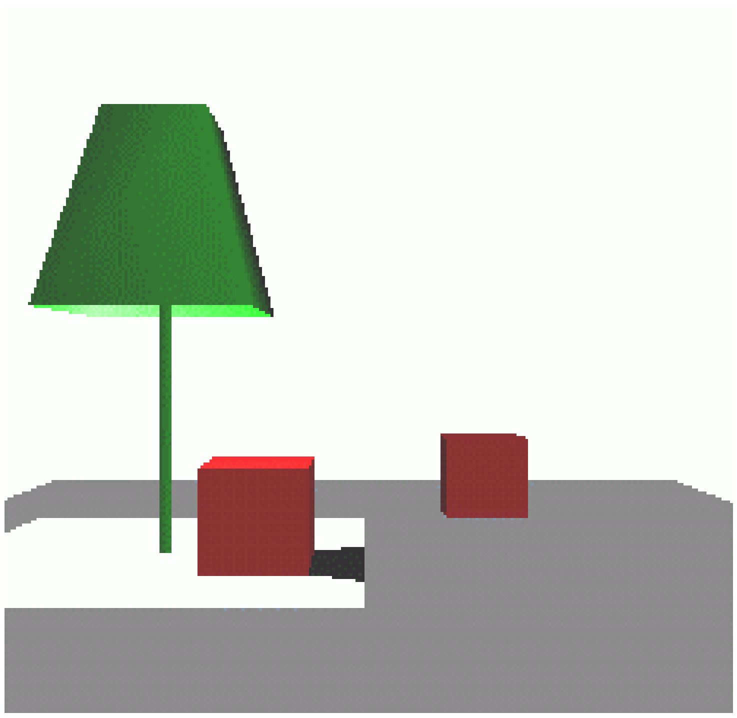
Getting Started with the Java 3D™ API

Chapter 6
Lights



Dennis J Bouvier

K Computing

tutorial v1.5 (Java 3D API v1.1.2)

Table of Contents

List of Figures

List of Tables

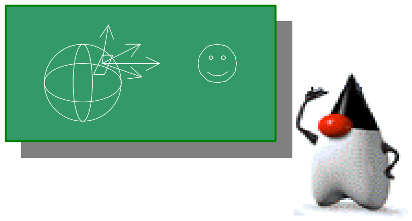
List of Code Fragments

List of Reference Blocks

Preface to Chapter 6

This document is one part of a tutorial on using the Java 3D API. You should be familiar with Java 3D API basics to fully appreciate the material presented in this Chapter. Additional chapters and the full preface to this material is presented in the Module 0 document available at:
`http://java.sun.com/products/javamedia/3d/collateral`

CHAPTER 6
Lights



Chapter Objectives

After reading this chapter, you'll be able to:

- Understand the Java 3D lighting model
- Specify light objects and attributes
- Specify visual objects with material properties
- Specify the influence of lights in a variety of ways

The images Java 3D renders from the virtual worlds in "Getting Started with the Java 3D API" (Module 1 of this tutorial) lack visual detail. Those images, like "paint by numbers art ", lack the variation in color and shading present in the real world. This module presents techniques for providing visual detail through *shading* and texturing of visual objects. Chapter 6, Lights, explains the lighting model and how to use lights in Java 3D to achieve shading. Chapter 7, Texturing, shows how to use textures to add detail to a visual object without adding more geometry.

Java 3D *shades* visual objects based on the combination of their *material properties* and the lights in the virtual universe. *Shading* results from applying a lighting model to a visual object in the presence of light sources. Section 6.1 gives an overview of the lighting model used in the Java 3D renderer and how the light interacts with material properties to provide shading. Each of the subsequent sections explains the Java 3D API features relevant to the lighting model.

6.1 Shading in Java 3D

The shading of visual objects in Java 3D depends on many factors. This section provides a brief overview of the Java 3D Lighting Model, Color Model, and Shading Models. The Java 3D API Specification presents more detailed information on the Java 3D Lighting Model. Since much of the Java 3D lighting and shading model is based on OpenGL, more information can also be found in OpenGL references.

Lighting Model

In the real world, the colors we perceive are a combination of the physical properties of the object, the characteristics of the light sources, the objects' relative positions to light sources, and the angle from which the object is viewed. Java 3D uses a lighting model to approximate the physics of the real world. The rest of this section explains the Java 3D lighting model in general terms. Section E.2 of The Java 3D API Specification presents the mathematical equations of the Java 3D lighting model.

The lighting model equation depends on three vectors: the surface normal (N), the light direction (L), and the direction to the viewer's eye (E) in addition to the material properties of the object and the light characteristics. Figure 6-1 shows the three vectors for two vertices of a spherical surface. The vectors for each vertex have may different directions depending on scene specifics. When the light and eye vectors vary, they are computed at rendering time. Therefore, each vertex of the sphere potentially renders as a different shade.

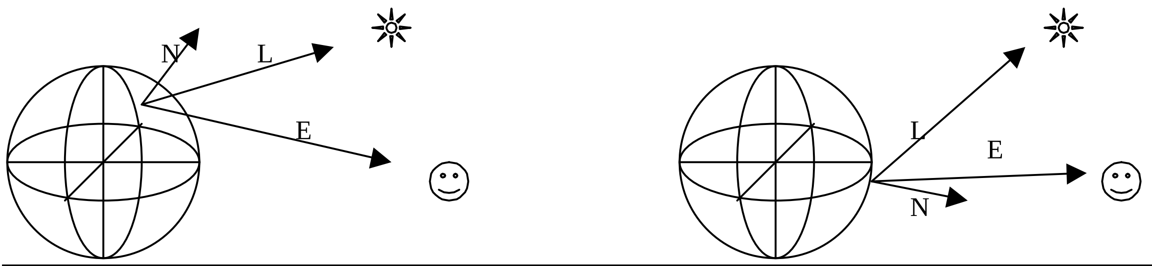


Figure 6-1 Light, Surface Normal, and Eye Vectors used to Shade Vertices.

The lighting model incorporates three kinds of real world lighting reflections: ambient, diffuse, and specular. Ambient reflection results from ambient light, constant low level light, in a scene. Diffuse reflection is the normal reflection of a light source from a visual object. Specular reflections are the highlight reflections of a light source from an object, which occur in certain situations.

Figure 6-2 shows a sphere and a plane rendered by Java 3D. The three types of reflection are seen in the sphere in Figure 6-2. The darkest part of the sphere exhibits ambient reflection alone. The center of the sphere is lit by diffuse and ambient light. With a blue sphere and a white light, the diffuse reflection is blue. The brightest part of the sphere is the result of specular reflection with ambient and diffuse reflections.

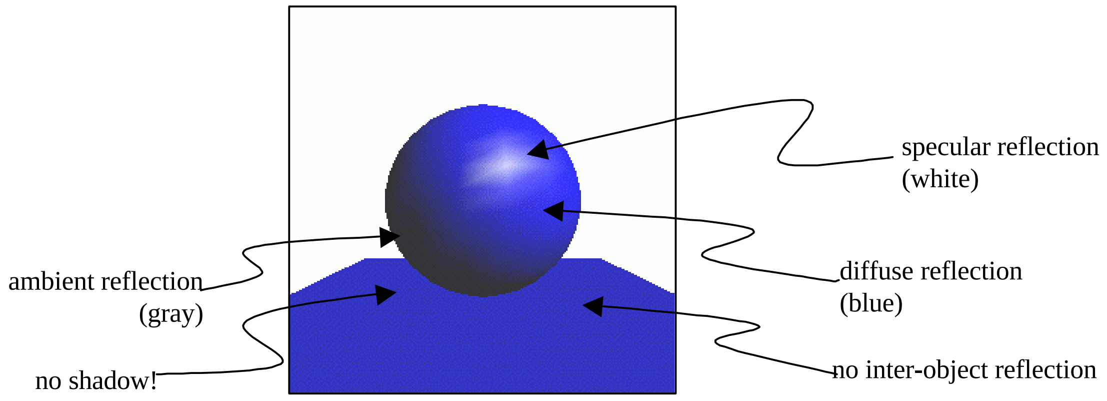


Figure 6-2 Shaded Sphere and Plane

Local Eye versus Infinite Eye Vectors

If each vertex of each visual object in a scene requires a light vector, an eye vector, and shade calculation, a significant portion of the rendering computation is used in shading vertices. The amount of computation can be reduced if the light vector, or eye vector, or both vectors are constant. The light vector is constant when using a directional light (see section 6.3.2 for more information). The eye vector is constant by default, although you can specify a variable eye vector using a method of the View object (see section 6.7.5).

Inter-object Effects Not Considered

While the lighting model is based on physics, complex physical phenomena are not modeled. Obviously, the shadow cast by the sphere on the plane is missing from Figure 6-2. Not as obvious, the light reflected from the sphere onto the plane is also missing. Also missing is the light reflected from the plane onto the sphere which is reflected back to the plane … and so on.

It is often difficult to comprehend the complexity of calculating the action of light. Consider the difficulty of calculating how each drop of water behaves in a shower. Drops come from the showerhead in a variety of directions. When they encounter an object, the resulting collision produces many smaller drops travelling in a variety of directions. The process repeats many times before the water runs down the drain. The complexity of light interactions with visual objects is similarly complex. Some differences between the behaviors of water and light are that light has no adhesion (light doesn't stick to visual objects) and the effect of gravity is negligible for light.

To reduce the complexity of computation, the lighting model considers only one visual object at a time. As a result, shadows and inter-object reflections are not rendered by the lighting model. Both of these effects require considering all objects together with their relative positions at rendering time. Significantly more computation is required to render a single scene with inter-object effects. Java 3D, and all other real time graphics systems, ignore inter-object effects in rendering. Some of the ignored real world effects can be added to scenes where necessary. Section 6.7 discusses the complexity of shadows and techniques for producing shadows in a Java 3D program.

Color Model

The color model is not physics based. Java 3D models the color of lights and materials as the combination of red, green, and blue. The color white, whether as a light or a material color, is the combination of all three components at maximum intensity. Each light produces a single color light specified by a RGB tuple. The lighting model is applied for each of the RGB color components. For example, a red ball in the presence of a blue light will not be visible since the blue light will not reflect from a red object. In reality, color is a combination of many wavelengths of light, not just three. The RGB color model represents many colors, but not all.

Influence of Lights

In Java 3D, the portion of a scene where visual objects are illuminated by a particular light source is called that light object's *region of influence*. The simplest region of influence for a light source uses a Bounds object and the `setInfluencingBounds()` method of the light. When a light source object's influencing bounds intersects the bounds of a visual object, the light is used in shading the entire object. The influencing bounds of a light determines which objects to light, not which portions of objects to light. Section 6.2.2 explains this concept in more detail. Section 6.6 presents alternative methods for controlling the influence of lights.

Shading Model

The lighting model *shades*[1] each vertex of a visual object for each influencing light. The shade of a vertex is the sum of the shades provided by each light source for the vertex. The rest of a visual object is shaded based on the shade of the vertices. The Shade Model of a visual object, specified as an attribute in the Appearance NodeComponent, determines how the shading is done for the rest of the visual object.

The ColoringAttributes NodeComponent specifies the Shade Model for visual objects where the shade model is specified as one of SHADE_GOURAUD, SHADE_FLAT, FASTEST, NICEST. Refer to Section 2.6.3 of Module 1 for additional information on the ColoringAttributes class. Be aware the color set in a ColoringAttributes object is never used in shading.

In Gouraud shading, each pixel is shaded with a value derived from trilinear interpolation of the shade value of each vertex of its enclosing polygon. In flat shading, all pixels for a polygon are assigned the shade value from one vertex of the polygon[2]. Figure 6-3 shows one flat shaded sphere and one Gouraud shaded sphere. The advantage to flat shading is speed in software rendering[3]. Gouraud shading has the advantage of visual appeal.

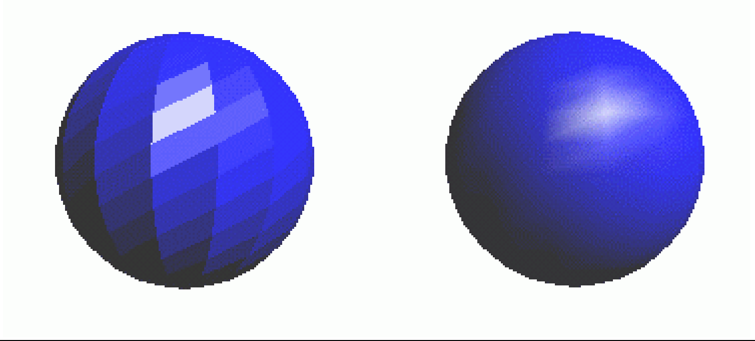


Figure 6-3 Flat and Gouraud Shaded Spheres.

One last point before going on to an example; light source objects are not visual objects. Even if a light source is placed within the view, it is not rendered.

6.2 Recipe for Lit Visual Objects

A number of steps are necessary to enable lighting for visual objects in the virtual universe. In addition to creating and customizing light objects, each light object must be added to the scene graph and have a bounds object specified. Each visual object to be shaded must have surface normals and material properties. Figure 6-4 lists these requirements.

[1] The term *shade* means to calculate the color value for a vertex or pixel in a rendered scene. In this context, shade has nothing to do with shadows. See the glossary for more information.

[2] The selection of the vertex to use for a polygon is implementation dependent and can be neither determined nor controlled.

[3] When using graphics hardware, the difference in rendering time for flat versus Gouraud shading varies considerably depending on the hardware. Gouraud shading may be faster than flat shading for some hardware.

1. Light Source specification
    a. set bounds
    b. add to scene graph
2. Visual object
    a. normals
    b. material properties

Figure 6-4 Items Necessary for Lit Scenes in Java 3D

Missing any one of these items will prevent lighting from being used. The presence of the Material object in an Appearance bundle of a visual object enables the lighting model for that object. Without the Material object the visual object will be colored, but not shaded by either the ColoringAttribute or vertex colors of the Geometry object. If neither is present, then the object will be solid white. Figure 6-5 shows the exception given when a visual object has a Material object but no surface normals.

```
javax.media.j3d.IllegalRenderingStateException: Cannot do lighting without
specifying normals in geometry object
```

Figure 6-5 Exception When a Visual Object With Material is Missing Normals.

Mistakes with light sources may not be as easy to find. There is no warning for leaving light out of a scene graph. There is no warning for not setting a light source's influencing bounds. In either case, the light object will have no influence on visual objects in the scene graph. A visual object properly specified for shading (i.e., one with a Material object) in a live scene graph but outside the influencing bounds of all light source objects renders black.

It is possible to properly specify a scene in which a visual object with material properties influenced by a light object that renders black. The relative orientation of the light, the visual object, and the viewing direction all come into play in rendering. Periodically this chapter addresses rendering issues. In particular, Sections 6.3.6 and 6.5 present possible programming pitfalls.

6.2.1 Simple Lights Example

As mentioned above, creating shaded renderings involves the proper specification of the light source and visual objects. Thus far, neither the Light classes nor Material objects have been discussed in any detail. However, taking advantage of API defaults and features, we can proceed in lighting virtual worlds. Geometric primitives generate surface normals when requested (refer to Section 2.3.8 of Module 1 for more information). Material Object defaults specify a reasonable visual object. Light source constructors' defaults specify usable light sources.

Using the SimpleUniverse class with the two methods of Code Fragment 6-1 produces a virtual universe including a single sphere with default material properties lit by a single AmbientLight light source object. The first method of the code fragment assembles a Material object with an Appearance object for the Sphere. The second method creates a BranchGroup object to serve as the root of the content branch graph, then adds Sphere and AmbientLight objects to the graph. The Material object in the appearance bundle is added to the Sphere object in the construction of the Sphere (lines 12 and 13). A default BoundingSphere provides the region of influence for the AmbientLight object (lines 15 through 17). The scene graph diagram of this virtual world appears in Figure 6-6.

```
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.  }
```

Code Fragment 6-1 Creating a Scene with a Lit Sphere.

Lines 4 and 5 of Code Fragment 6-1 could be replaced with the following line creating and using an anonymous Material object.

```
      Appear.setMaterial(new Material());
```

Material objects are fully customizable using the parameterized constructor, simplifying the use of anonymous Material objects (see Section 6.4). By contrast, creating an anonymous Light object makes adding the influencing bounds for the light much harder (see Section 6.3). Keep in mind that naming the Material object may make it easier to share the object among various appearance bundles, resulting in better performance.

The SimpleUniverse provides the VirtualUniverse and Locale objects along with the View Branch Graph for the Scene Graph Diagram shown in Figure 6-6. Without a transformation, the Sphere object and the BoundingSphere object are centered at the origin, and thus they intersect. The Sphere object is shaded by the AmbientLight source. Figure 6-7 shows the resulting image with a white background. The background specification is not shown in the code.

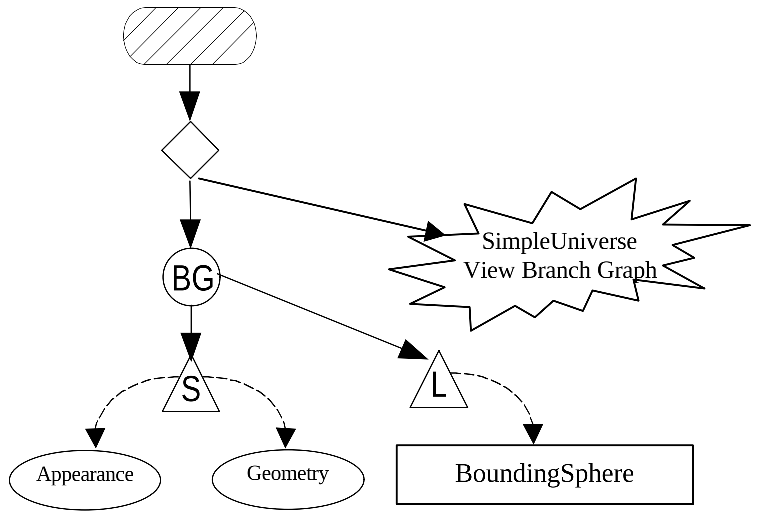


Figure 6-6 Scene Graph Diagram of Simple Example (Code Fragment 6-1)

The sphere of Figure 6-7 is uniform gray in color, which is the ambient material property default value.

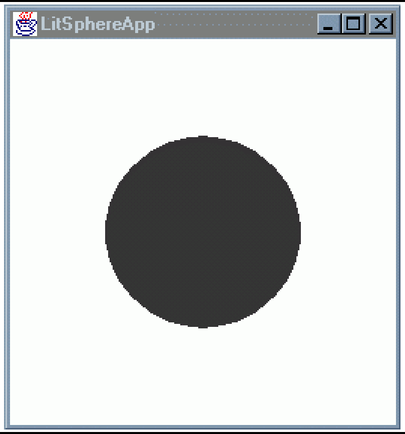


Figure 6-7 A Sphere with Ambient Light.

Figure 6-7 demonstrates that scenes lit with ambient light alone are dull. Because Ambient lighting is uniform, it produces uniform shade. Ambient light is intended as fill light in the scene where other sources do not light. Adding a DirectionalLight source will make this scene more interesting.

Inserting Code Fragment 6-2 into Code Fragment 6-1 adds a DirectionalLight to the content branch graph of the scene graph. Again, the defaults are used for the light source, and a default BoundingSphere is used for the region of influence. Figure 6-8 shows the resulting scene graph diagram without the objects provided by the SimpleUniverse.

```
1.   DirectionalLight lightD1 = new DirectionalLight();
2.   lightD1.setInfluencingBounds(new BoundingSphere());
3.   // customize DirectionalLight object
4.   scene.addChild(lightD1);
```

Code Fragment 6-2 Adding a Directional Light to the Scene.

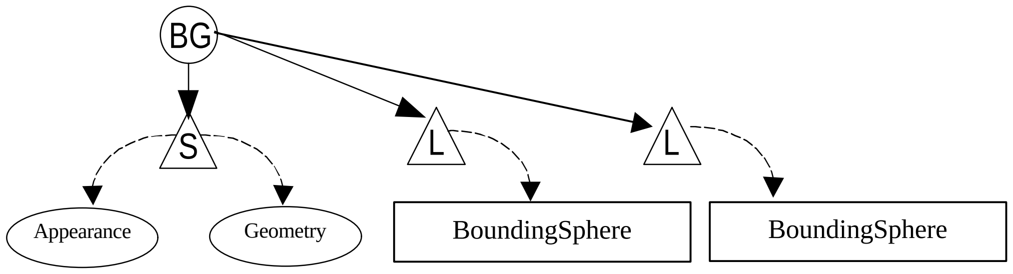


Figure 6-8 Abbreviated Scene Graph Diagram of Simple Example (Code Fragment 6-1 and 6-2).

Figure 6-9 shows the image produced by the combination of two code fragments. The influence of the AmbientLight object can hardly be seen with the DirectionalLight source. Obviously, customization of the light objects and/or the material properties of the visual object is necessary to create interesting scenes. These topics are covered in Sections 6.3 and 6.4, respectively. The next section presents a discussion of the location of light objects in the scene graph.

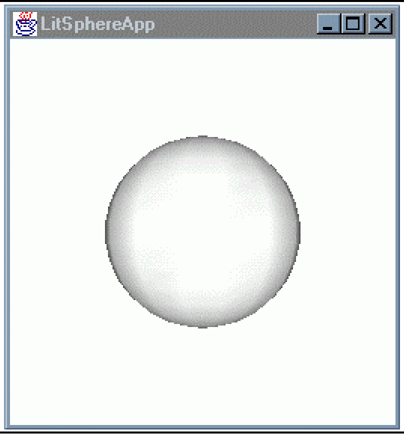


Figure 6-9 A Sphere with Ambient and Directional Light Sources.

6.2.2 Where to Add a Light Object in the Scene Graph

The influence of a light object on the world is not affected by the light object's position in the scene graph; however, the bounds object referenced by the light is. The bounds object is subject to the local coordinates of the scene graph where the light object is inserted. Consider Figure 6-10 as an example. The same BoundingSphere object referenced by two light sources provided two different regions of influence due the translation provided by the TransformGroup object. The origin of the local coordinate system of the scene graph below the TransformGroup is 2 meters below the origin of the world (Locale) and the other region of influence sphere.

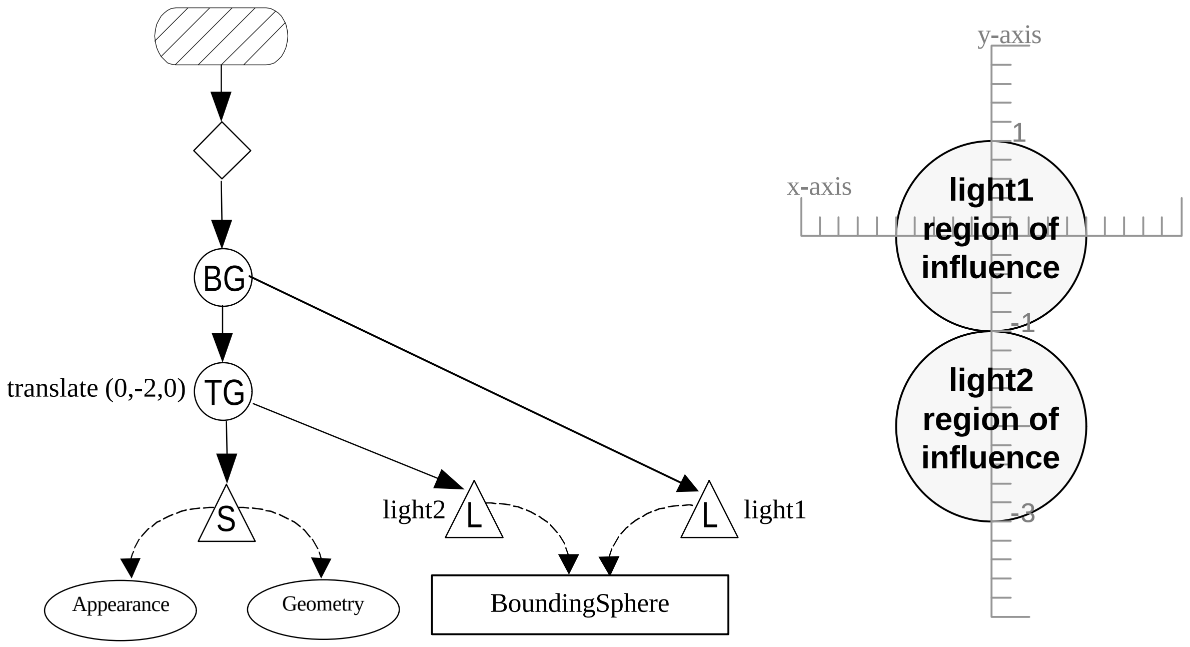


Figure 6-10 BoundingSphere Affected by Transformation

Whether or not either, or both, light source objects of the scene graph in Figure 6-10 influence the shading (light) of the visual object depends on whether the bounds of the visual object intersects the region of influence of the light objects. Specifying a light's region of influence as a single bounds object may not work for all applications. Section 6.6 presents alternative methods for controlling the influence of lights.

6.3 Light Classes

The Java 3D API provides four classes for lights. Each are derived from the Light class. Figure 6-11 shows the Java 3D class hierarchy related to lights. Light, an abstract class, provides the methods and associated capability constants for manipulating the state, color, and bounds of a Light object. The state of light is a Boolean turning the light on or off. Sections 6.3.1 through 6.3.4 give details for each of the light classes. Section 6.6 presents alternative methods for controlling the influence of lights.

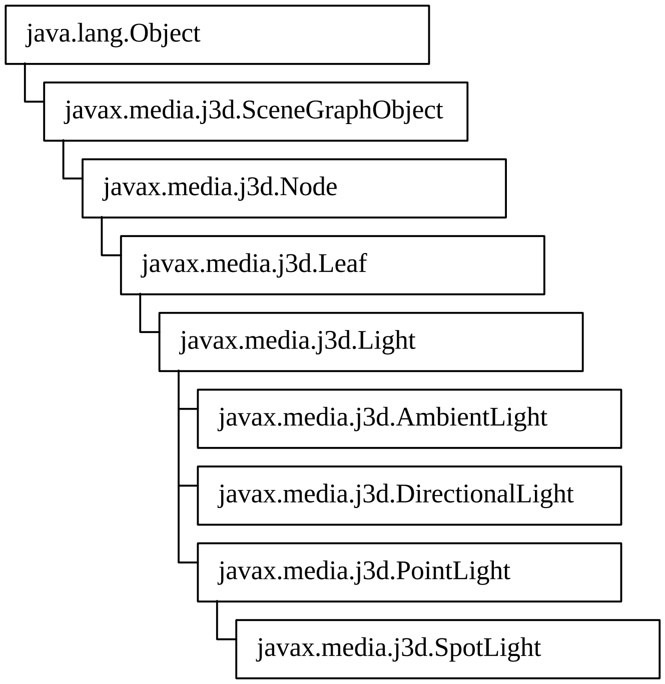

java.lang.Object
javax.media.j3d.SceneGraphObject
javax.media.j3d.Node
javax.media.j3d.Leaf
javax.media.j3d.Light
javax.media.j3d.AmbientLight
javax.media.j3d.DirectionalLight
javax.media.j3d.PointLight
javax.media.j3d.SpotLight

Figure 6-11 Java 3D API Class Hierarchy for Light Classes

The following reference block lists the methods and constants of the Light class. Recall that bounds set with `setInfluencingBounds()` enables a light when the referenced bounds object intersects the view frustum.

Light Method and Capability Summary (partial list)

Light is an abstract class containing instance variables common to all lights. Additional methods appear in a reference block in Section 6.6 (page 6-29).

`void setColor(Color3f color)`
Sets the Light's current color.

`void setEnable(boolean state)`
Turns the light on or off.

`void setInfluencingBounds(Bounds bounds)`
Sets the light's influencing bounds.

Light Capabilities Summary

Refer to Section 1.8.2 for more information on Capabilities.

`ALLOW_INFLUENCING_BOUNDS_READ | WRITE`

`ALLOW_STATE_READ | WRITE`

`ALLOW_COLOR_READ | WRITE`

6.3.1 Ambient Light

Ambient light objects provide light of the same intensity in all locations[4] and in all directions. Ambient light objects model the light reflected from other visual objects. If you look up at the underside of your desk, you will see the bottom surface of the desk although no light source is directly shining on that surface (unless you have a lamp under your desk). The light shining up on the bottom surface of the desk reflected off the floor and other objects. In natural environments with many objects, light reflects off many objects to provide ambient light. The AmbientLight class in Java 3D simulates this effect.

The following reference block lists the constructors for the AmbientLight class. The Light abstract class provides the methods and capabilities for this class (listed in the previous reference block).

AmbientLight Constructor Summary

A light source object providing the same intensity of light at all locations in all directions. Models the complex inter-object reflection of light present in natural scenes. See the Light Method Summary on page 6-10 for methods.

`AmbientLight()`
Constructs and initializes an ambient light source using the following default values:

```
      lightOn         true
      color           (1, 1, 1)
```

`AmbientLight(Color3f color)`
Constructs and initializes an ambient light using the specified parameters.

`AmbientLight(boolean lightOn, Color3f color)`
Constructs and initializes an ambient light using the specified parameters.

While it may be natural to think of an ambient light source as globally applicable, it is not necessarily true in a Java 3D program. AmbientLight source influence is controlled by its bounds just like other Java 3D light sources. Multiple AmbientLight source objects can be used in a Java 3D program. There is no limit on the number of AmbientLight source objects in Java 3D programs[5].

As mentioned in Section 6.1 (Shading in Java 3D), the shade of a vertex is the result of the light sources, the material properties of the visual object, and their relative geometry (distance and orientation). For ambient reflections, the geometry is not a factor. The ambient material property is only used in calculating the ambient reflection. The lighting model calculates the ambient reflection of light as the product of AmbientLight intensity and the ambient material properties of visual object. Section 6.4 presents material properties of visual objects in more detail.

6.3.2 Directional Light

A DirectionalLight source approximates very distant light sources such as the sun. Unlike AmbientLight sources, DirectionalLight sources provide light shining in one direction only. For objects lit with a DirectionalLight source, the Light vector is constant.

[4] Of course, the influence of the light source is limited by its bounds. "In all locations" means under the visual object as well as over it when the light source's bounds intersects the ambient light source's region of influence.

[5] While the Java 3D API imposes no limit on the number of light sources, the implementation may. See page 6-20 for more information on the Limit on the Number of Lights.

Figure 6-12 shows two vertices of the same sphere being lit by a DirectionalLight source. The Light Vector is the same for these two and all vertices. Compare Figure 6-12 with Figure 6-1 to see the difference. Since all light vectors from a DirectionalLight source are parallel, the light does not attenuate. In other words, the intensity of a DirectionalLight source does not vary by the distance between the visual object and the DirectionalLight source.

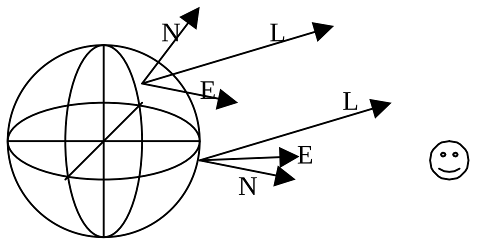


Figure 6-12 Light Vector is Constant for DirectionalLight Source.

The next two reference block list the constructors and methods for DirectionalLight object, respectively.

DirectionalLight Constructor Summary

DirectionalLight objects model very distant light sources having a constant light direction vector.

`DirectionalLight()`

Constructs and initializes a directional source using the following default values:

```
      lightOn        true
      color          (1, 1, 1)
      direction      (0, 0, -1)
```

`DirectionalLight(Color3f color, Vector3f direction)`

Constructs and initializes a directional light with specified color and direction. By default the state is true (on).

`DirectionalLight(boolean lightOn, Color3f color, Vector3f direction)`

Constructs and initializes a directional light with specified state, color, and direction.

The following reference block lists the methods and capability bits the DirectionalLight class adds to the methods of Light class.

DirectionalLight Method Summary

See the Light Method Summary on page 6-10 for more methods.

`void setDirection(Vector3f direction)`

Set light direction.

`void setDirection(float x, float y, float z)`

Set light direction.

DirectionalLight Capabilities Summary

In addition to the Capabilities inherited from Light (listed on page 6-11), DirectionalLight objects have the following Capability.

`ALLOW_DIRECTION_READ | WRITE`

DirectionalLights only participate in diffuse and specular reflection portions of the lighting model. For diffuse and specular reflections, the geometry is a factor (unlike ambient reflections). Varying the direction of the light source will change the shading of visual objects. Only diffuse and specular material properties are used in calculating the diffuse and specular reflections. Section 6.4 presents material properties of visual objects in more detail.

6.3.3 Point Light

The PointLight is the conceptual opposite of a DirectionalLight. It is an omni-directional light source whose intensity attenuates with distance and has location. (A DirectionalLight source has no location, just a direction.) PointLight objects approximate bare light bulbs, candles, or other light sources without reflectors or lenses.

A quadratic equation models the attenuation of PointLight sources. The equation is found in Section E.2 of The Java 3D API Specification. Figure 6-13 illustrates the relationship of a PointLight object with a sphere. Note that the light vectors are not parallel.

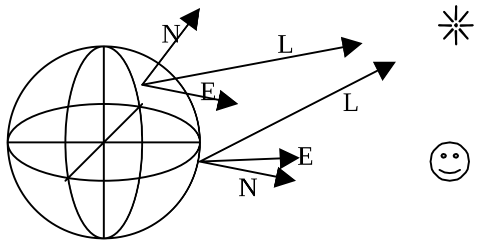


Figure 6-13 Light Vector Varies for a PointLight Source.

The following two reference blocks list the constructors and methods of PointLight, respectively.

PointLight Constructor Summary

The PointLight object specifies an attenuated light source at a fixed point in space that radiates light equally in all directions away from the light source.

`PointLight()`
Constructs and initializes a point light source using the following default values:

```
      lightOn        true
      color          (1, 1, 1)
      position       (0, 0, 0)
      attenuation    (1, 0, 0)
```

`PointLight(Color3f color, Point3f position, Point3f attenuation)`
Constructs and initializes a point light. By default, the light is on.

`PointLight(boolean lightOn, Color3f color, Point3f position, Point3f attenuation)`
Constructs and initializes a point light.

PointLight Method Summary

See the Light Method Summary on page 6-10 for more methods.

`void setAttenuation(Point3f attenuation)`
Sets this Light's current attenuation values and places it in the parameter specified. The three values specified in the Point3f object specify the *constant*, *linear*, and *quadratic* coefficients, respectively.

$$attenuation = \frac{1}{constant + linear * distance + quadratic * distance^2}$$

where *distance* is measured from the light source to the vertex being shaded.

`void setAttenuation(float constant, float linear, float quadratic)`
Sets this Light's current attenuation values and places it in the parameter specified. See the above equation.

`void setPosition(Point3f position)`
Set light position.

`void setPosition(float x, float y, float z)`
Set light position.

PointLight Capabilities Summary

In addition to the Capabilities inherited from Light (listed on page 6-11), PointLight objects have the following Capabilities.

`ALLOW_POSITION_READ | WRITE`

`ALLOW_ATTENUATION_READ | WRITE`

Like DirectionalLights, PointLights only participate in diffuse and specular reflection portions of the lighting model. For diffuse and specular reflections, the geometry is a factor. Varying the location of a PointLight object will change the shading of visual objects in a scene.

6.3.4 Spot Light

SpotLight is a subclass of PointLight. SpotLight class adds direction and concentration to the position and attenuation parameters of PointLight. SpotLight objects model man-made light sources such as flash lights, lamps, and other sources with reflectors and/or lenses.

The intensity of light produced from a SpotLight source produces light within a specified spread angle from the direction of the light. If the vertex lies outside the spread angle of the light, then no light is produced. Inside the spread angle, the intensity varies by the angle and distance to the vertex. Again, a quadratic equation models the attenuation due to distance. The concentration parameter and a different equation governs the variation of intensity due to angle. The equations governing these relationships are found in Section E.2 of The Java 3D API Specification. Figure 6-14 illustrates in 2D how light intensity varies from a PointLight source in 3D.

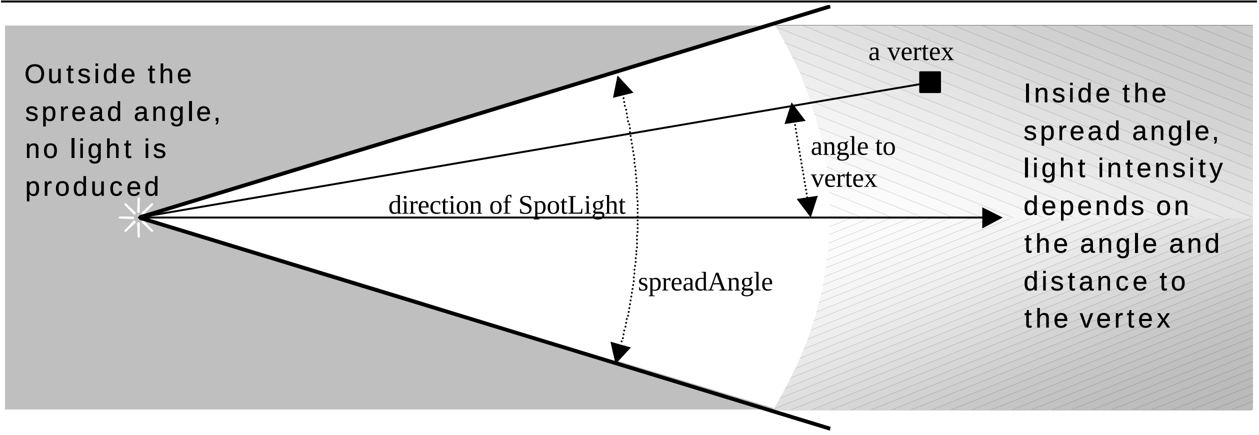

Outside the spread angle, no light is produced
a vertex
angle to vertex
direction of SpotLight
spreadAngle
Inside the spread angle, light intensity depends on the angle and distance to the vertex

Figure 6-14 Light Intensity Varies with Distance and Orientation for a PointLight Source.

The spread angle of a SpotLight object may cause the light to illuminate part of a visual object. This is the only light capable of lighting part of a visual object.

The following two reference blocks list the constructors and methods of PointLight, respectively.

SpotLight Constructor Summary

SpotLight is a subclass of a point light object with the addition of direction, spread angle, and concentration attributes.

`SpotLight()`
Constructs and initializes a spot light source using the following default values:

```
      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)`
Constructs and initializes a spot light. See the PointLight Method Summary on page 6-14 for information on attenuation. By default the light is on.

`SpotLight(boolean lightOn, Color3f color, Point3f position, Point3f attenuation, Vector3f direction, float spreadAngle, float concentration)`
Constructs and initializes a spot light. See the PointLight Method Summary on page 6-14 for information on attenuation.

SpotLight Method Summary

See the PointLight Method Summary on page 6-14, and the Light Method Summary on page 6-10 for more methods.

`void setConcentration(float concentration)`
Set spot light concentration.

`void setDirection(float x, float y, float z)`
Set light direction.

`void setDirection(Vector3f direction)`
Sets the light's direction.

`void setSpreadAngle(float spreadAngle)`
Set spot light spread angle.

SpotLight Capabilities Summary

In addition to the Capabilities inherited from Light (listed on page 6-11), SpotLight objects have the following Capabilities.

`ALLOW_SPREAD_ANGLE_READ | WRITE`

`ALLOW_CONCENTRATION_READ | WRITE`

`ALLOW_DIRECTION_READ | WRITE`

Like DirectionalLight and PointLight objects, SpotLights only participate in diffuse and specular reflection portions of the lighting model. For diffuse and specular reflections, the geometry is a factor. Changing the location or orientation of a SpotLight source changes the shading of the vertices within the light's region of influence.

6.3.5 Applications of Light Sources

With all of the types of light sources, and variety of ways to use them, I will give a little guidance on their typical use in this section. In general, you want to use as few light sources as you can for a given application. How many is enough will vary depending on the lighting effect desired for the application. The number of lights and the setting of attributes is much more an artistic consideration than a scientific one.

From an artistic point of view, it is often sufficient to have only two lights for a given scene. One light provides the main lighting, the other is used to fill in the darker side of the objects. The main light is typically positioned to the viewer's right, the fill to the viewer's left. Again, these are simply general guidelines for what can be a complex artistic design.

Including Directional light sources is preferred for most applications since the computation required in rendering is significantly less than for point and spot lights. Point light sources are rarely used due to the high computational complexity.

Including a single Ambient light source with a large region of influence is normal. This will light the backsides of objects (like the "dark side of the moon"). The default color will work reasonably well. The time required to include the Ambient light is small compared to other light sources. Leaving out an Ambient light may be very noticeable in some scenes, and not noticed at all in other scenes.

6.3.6 Examples of Lighting

The interaction of light with objects is very complex in nature. Even in the virtual world where the lighting model is less complex, the light sources are simplistic, and the surfaces are less detailed, the effect of a light source on a visual object is rather complex. This section presents a few lighting examples to help clarify the characteristics, capabilities, and limitations of the lighting model in Java 3D.

Two Colored Lights

Figure 6-15 shows a single white sphere illuminated by two directional light sources, one red and one blue. Although it may surprise you, the resulting shade is magenta. Mixing red and blue watercolor results in purple, which is the result in the subtractive color system. Mixing red and blue light results in magenta, the results of an additive color system.

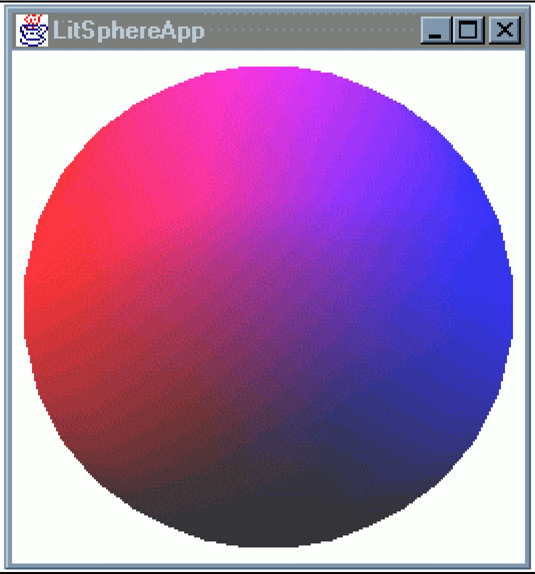


Figure 6-15 Spheres Lit by Two Directional Light Sources

In the absence of light, the sphere is black. If the only light source is red, then the sphere appears red, or some shade of red. In adding a blue light source, only red, blue, and mixtures of red and blue are possible.

Different Lighting Patterns

The next application illustrates the differences among light sources. In `LightsNPlanesApp.java` three planes are lit by one light source each. From left to right, DirectionalLight, PointLight, and SpotLight objects light the planes. Figure 6-16 shows the image rendered by this application.

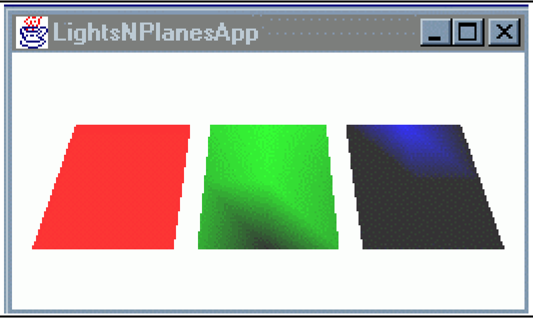


Figure 6-16 Planes Lit by Directional, Point, and Spot Light Sources (best viewed in color).

The DirectionalLight illuminates the plane evenly. The PointLight, located directly above the upper edge of the center plane, illuminates the center plane unevenly due to the variable direction of light with respect to the normals, and, to a lesser extent, attenuation of the light. The SpotLight, also located directly above the center of its plane, only illuminates a small part of the third plane.

Figure 6-17 illustrates the geometry involved in the lighting of the first two planes. In the left illustration, the constant light vectors of the DirectionalLight source in combination with the constant normal vectors of a plane results in constant reflection vectors, and even illumination of the plane. In the right illustration the variable light vectors of the PointLight source combine with the constant normal vectors of the plane resulting in various directions for reflection vectors, and uneven illumination of the plane. The SpotLight is a special case of the PointLight source where the influence of the light source is limited by the spread angle.

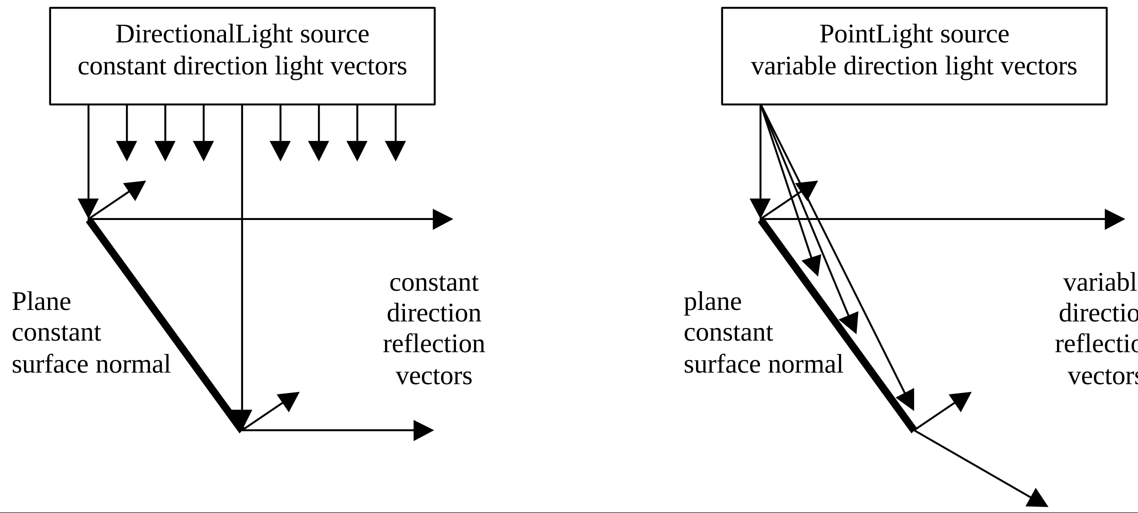


Figure 6-17 Geometry of Lighting Planes with Directional and Point Light Sources

Concentration and Spread Angle of SpotLights

Figure 6-18 shows images rendered from two different versions of the `ConcentrationApp.java` program. One plane is lit by nine spot lights in the program. The spread angle and concentration values for the spot lights vary by position. The spread angle varies by row with values of .1, .3. and .5 (radians) for the upper to lower row, respectively. The concentration varies by column with values of 1.0, 50.0, and 100.0 for the left to right columns, respectively.

The concentration values have no effect for the upper row, the spread angle is the only factor. On the lower row, the concentration has an effect for each of the spread angles. The blue in the images is the diffuse color of the material.

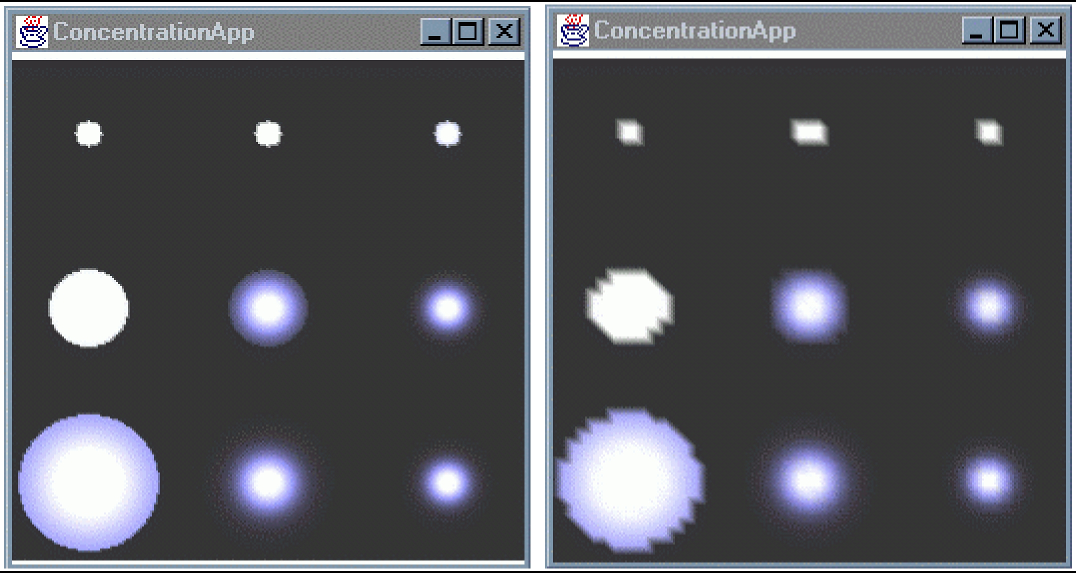


Figure 6-18 A Plane Lit by SpotLights with Different Concentrations and Spread Angles.

ConcentrationApp demonstrates two limitations of the lighting model. The first being the rendering artifacts shown in Figure 6-18. Similar artifacts are visible in Figure 6-16. The uneven lighting patterns for both the green and red planes is due to the small number of vertices used to represent the planes. Recall the lighting model is only applied per vertex. So the more vertices, the smoother the lighting effect and the longer it will take to render.

The difference in the left and right images of Figure 6-18 is due to the difference in the number of vertices used to represent the plane. The version of the program that generated the left image used 16 times more vertices than the one on the right (2,500 vertices versus 40,000). The artifacts in the right image is a result of the reduction of vertex density on the surface and the triangulation imposed by the Java 3D rendering system.

Limit on the Number of Lights

The second limitation demonstrated in ConcentrationApp is not seen in the rendering. The plane in ConcentrationApp is really four plane objects next to each other. This was done to overcome a potential limitation of underlying rendering system. The OpenGL specification only requires support for eight simultaneous light sources[6]. If the plane in ConcentrationApp were one visual object, then OpenGL would limit the number of lights to eight for some machines.

Using the influencing bounds to pick only the relevant light sources for a visual object, Java 3D dynamically creates light specifications for lights as visual objects are rendered. As long as no single object is lit by more than eight lights, Java 3D programs are not limited in the number of lights in a virtual world.

So in providing four smaller planes and the appropriate bounds to ensure that no plane is influenced by more than eight lights in the example, it appears there are nine lights (actually ten, with the ambient light) illuminate one plane. It takes a little more programming, but the resulting program is more portable. While many implementations of OpenGL support more than eight simultaneous lights, if you plan distribute your programs, you should be aware of this potential limitation.

In this section, a few examples show some of the characteristics and limitation of lighting in Java 3D. The intention of this section is to give readers basic usage examples and some example figures to compare to their own programs. It is not possible to provide examples of every possible lighting situation, as the factors in rendering are too varied.

One last thing, the point and spot light support the specification of attenuation. The attenuation is specified by the constant terms in the inverse quadratic equation based on the distance between the light and the vertex (see the reference block on page). Finding the appropriate attenuation for a specific application is an artistic issue. No attenuation example programs are included in this tutorial.

6.4 Material Object

The material properties of a visual object are specified in the Material object of an appearance bundle. Material is a subclass of NodeComponent. Figure 6-19 shows the Java 3D API hierarchy for Material.

[6] As you should be aware, the Java 3D API uses low level graphics systems for rendering (currently OpenGL or DirectX). The limitations of a particular low level graphics system may affect the results of a Java 3D program.

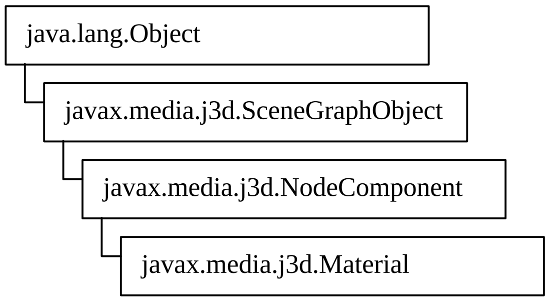

java.lang.Object
javax.media.j3d.SceneGraphObject
javax.media.j3d.NodeComponent
javax.media.j3d.Material

Figure 6-19 API Hierarchy for Material

The Material object specifies ambient, diffuse, specular, and emissive colors and a shininess value. Each of the first three colors are used in the lighting model to calculate the corresponding reflection. The emissive color allows visual objects to "glow in the dark". The shininess value is only used in calculating specular reflections.

The following two reference blocks list the constructors and methods for Material.

Material Constructor Summary

The Material object defines the appearance of an object under illumination.

`Material()`

Constructs and initializes a Material object using the following default values:

```
      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)`

Constructs and initializes a new Material object using the specified parameters.

Material Method Summary (partial list)

`void setAmbientColor(Color3f color)`
Sets this material's ambient color.

`void setAmbientColor(float r, float g, float b)`
Sets this material's ambient color.

`void setDiffuseColor(Color3f color)`
Sets this material's diffuse color.

`void setDiffuseColor(float r, float g, float b)`
Sets this material's diffuse color.

`void setDiffuseColor(float r, float g, float b, float a)`
Sets this material's diffuse color plus alpha.

`void setEmissiveColor(Color3f color)`
Sets this material's emissive color.

`void setEmissiveColor(float r, float g, float b)`
Sets this material's emissive color.

`void setLightingEnable(boolean state)`
Enables or disables lighting for visual objects referencing this appearance node component object.

`void setShininess(float shininess)`
Sets this material's shininess.

`void setSpecularColor(Color3f color)`
Sets this material's specular color.

`void setSpecularColor(float r, float g, float b)`
Sets this material's specular color.

`java.lang.String toString()`
Returns a String representation of this Materials values.

Material Capabilities Summary

In addition to the Capabilities inherited from NodeComponent, Material objects have the following Capability.

`ALLOW_COMPONENT_READ | WRITE` allows reading/writing of individual component field information.

6.4.1 Simple Material Examples

Specular reflections occur naturally with smooth objects. In general, the smoother a surface is, the more defined and intense the specular reflection. When a surface is sufficiently smooth, it acts like a mirror reflecting the light without changing the color of the light. Consequently, the specular color of an object is normally white. Change the specular color of a material to alter the intensity of a specular reflection (e.g., `Color3f(0.8f, 0.8f, 0.8f)`).

The shininess value controls the spread range of viewing angle for which a specular reflection can be seen. Higher shininess values result in smaller specular reflections. Figure 6-20 shows nine different spheres illuminated by one light source. Each sphere has a different shininess value.

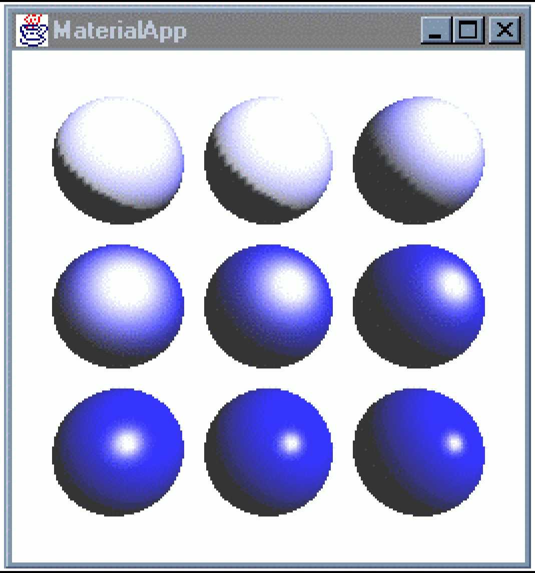


Figure 6-20 Different Material Properties

A Material object is associated with a visual object through an Appearance object much the same way appearance attributes objects are. The `setMaterial()` method of Appearance class references a Material object for that Appearance object. Refer to Code Fragment 6-1 (lines 1-4) for an example.

6.4.2 Geometry color, ColoringAttributes, and Material Properties

There are three ways to specify color for a visual object: per-vertex color specified in the geometry with `getColor()` methods (Section 2.5.1), ColoringAttributes of an Appearance node (Section 2.6.3), and Material Object (Section 6.3). Java 3D allows you to create visual objects using none, some, or all three of the ways to specify color.

When more than one color specification has been made, two simple rules determine which color specification takes precedence.

- Material color is only used when rendering lit objects and ColoringAttributes color is only used when rendering unlit objects.
- Per-vertex geometry color always takes precedence over ColoringAttributes or Material color.

The rules may be clearer when the problem is divided into lit and unlit objects. Lighting in enabled for an object when a Material object is referenced. Conversely, when no Material object is associated with the visual object, lighting is disabled for that object. Note that a scene may have both lit and unlit objects.

When lighting is enabled for an object (i.e., a Material object is referenced), Material color or per-vertex Geometry color is used in shading. If present, per-vertex color overrides the diffuse and ambient Material colors. Note that the ColoringAttributes color is never used for lit objects. Table 6-1 summarizes the relationships.

Table 6-1 Coloring for Objects when Lighting is Enabled (Material object is referenced)

per-vertex Geometry color	ColoringAttributes color	Result
NO	NO	Material color
YES	NO	Geometry color
NO	YES	Material color
YES	YES	Geometry color

When lighting is disabled for an object (i.e., a Material object is not referenced), ColoringAttributes color or per-vertex color is used for coloring. If present, per-vertex Geometry color overrides the ColoringAttributes color. Table 6-2 summarizes the relationships.

Table 6-2 Coloring for Objects when Lighting is Disabled (no Material object is referenced)

per-vertex Geometry color	ColoringAttributes color	Result
NO	NO	flat white
YES	NO	Geometry color
NO	YES	ColoringAttributes
YES	YES	Geometry color

6.5 Surface Normals

As mentioned in Section 6.2, normals are required for shaded visual objects. When creating visual objects using Geometry classes, use one of the `setNormal()` methods from Section 2.5.1 of Module 1 to specify per vertex normal vectors.

The NormalGenerator included with the Java 3D utilities generates normals when specifying visual objects using GeometryInfo objects. To generate normals, put your visual object geometry into a GeometryInfo object and call `NormalGenerator.generateNormals()`.

Geometric primitives generate their own normals when specified. Refer to Section 2.3.8 of Module 1 for more information.

No matter how the normals are specified (or generated), only one normal is specified per vertex. This leads to some interesting problems. For example, when both surfaces of polygons are viewable, the normal is only correct for one of the surfaces. The result is for the back faces to be rendered (if they are rendered) only using the ambient material properties. Both the diffuse and specular reflections require proper normal specification.

This common problem is solved by specifying back face normals as the inverse of the front face normals. (See Section 2.6.4 of Module 1 for a discussion of front and back faces.) Use the `setBackFaceNormalFlip()` method of a PolygonAttributes object for this purpose.

Figure 6-21 shows two images of the shaded twist strip. The left image was rendered with front facing normals only, the right with back face flipped normals.

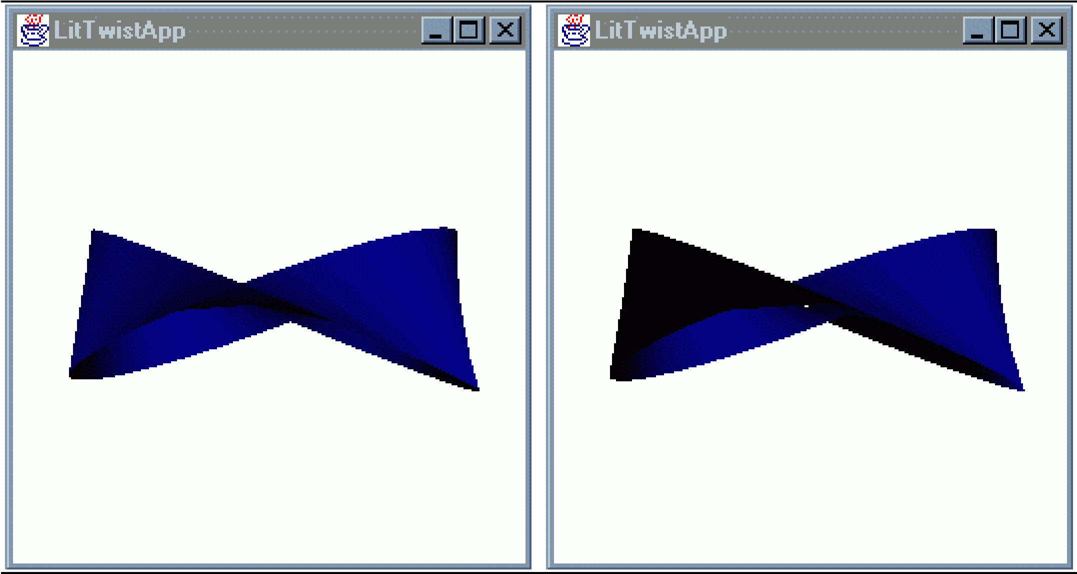


Figure 6-21 Twist Strip with (left) and without (right) Back Face Flip Normals (best viewed in color)

When a vertex is shared by surfaces of widely varying orientations, having only one normal per vertex can result in problems. Consider the examples illustrated in Figure 6-22. The geometry illustrated on the left hand side of Figure 6-22 shows the cross section of a surface where each polygon is oriented at a 90° angle to its neighbors. If the normal is selected as the actual normal for one surface, it is very wrong for its neighbor. If the normals are specified as shown, then the surface will be shaded consistently between vertices with parallel normals. A similar problem occurs with the cube geometry shown on the right hand side in Figure 6-22. The solution to both problems is to increase the number of vertices in order to increase the number of normals. This, of course, increases memory usage and rendering time.

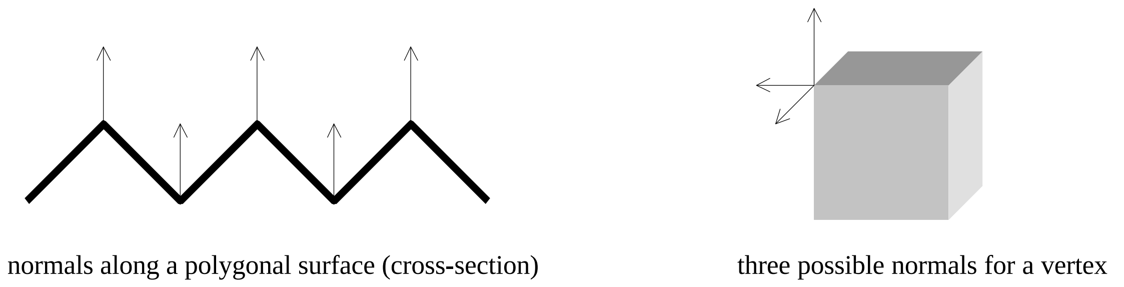

normals along a polygonal surface (cross-section)
three possible normals for a vertex

Figure 6-22 Geometry with Normal Specification Problems

6.6 Specifying the Influence of Lights

In previous examples the specification of a light object's influencing bounds is accomplished through referencing a Bounds object. This links the location of the influencing bounds with the location of the light. (Section 6.2.2 explained how the transformations in the scene graph affect the bounding volumes used to specify influencing bounds of lights.) While this makes it trivial to move lights together with the visual objects they light, other applications need more flexible specification of the influence of lights. Fortunately, the Java 3D API provides an alternative method for specifying the influencing bounds and a way to limit the scope in addition to the bounds.

6.6.1 Influencing Bounds Alternative: BoundingLeaf

A BoundingLeaf object is one alternative to an influencing bounds object. A BoundingLeaf object is referenced by other leaf nodes to define a region of influence. As a descendant of SceneGraphObject, instances of BoundingLeaf object are added to the scene graph. The BoundingLeaf object is subject to the local coordinate system of its position in the scene graph, which could be independent of the coordinate system of the light object. In other words, using a BoundingLeaf allows a light and its influencing bounds to move independently.

A call to `setInfluencingBoundingLeaf()` for a light object specifies the BoundingLeaf argument as the influencing bounds for the light. This specification overrides any regional influencing bounds specification. A BoundingLeaf object can be shared by multiple light objects.

Figure 6-23 shows the scene graph diagram for an example application of a BoundingLeaf object with light objects. In this scene, two lights are moved together using a TransformGroup object (on the right). These lights could be instances of PointLight or SpotLight. However, the influence of these lights does not change when the lights move. The influence of the lights moves when the left TransformGroup changes the location of the BoundingLeaf object. Compare this scene graph diagram to the one in Figure 6-10.

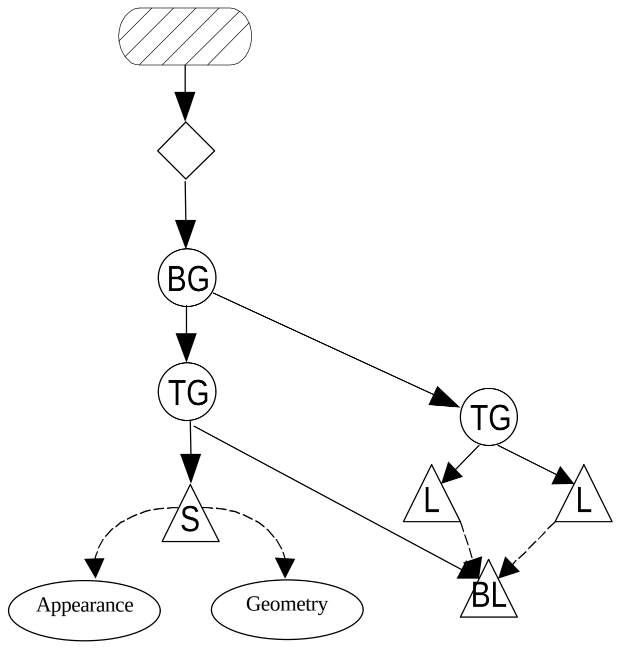


Figure 6-23 Moving Light Objects Independently of Their Influence using a BoundingLeaf.

In Figure 6-10, if the light is moved it's region of influence moves. Also, as demonstrated in Figure 6-10, the region of influence of two lights sharing the same Bounds object may or may not have the same region of influence. When two or more lights share the same BoundingLeaf object, they always have the same region of influence.

6.6.2 Scoping Limits Lights' Influence

A bounding region, whether a Bounds or a BoundingLeaf object, specifies the region of influence of a light object. A specified scope can further limit the influence of a light to a portion of the scene graph. By default, all lights have the scope of the virtual world in which it resides. Adding a scope specification further reduces the influence of a light to the visual objects in the scene graph below the group(s) specified. For example, consider the following application.

Light Scoping Example

The scene consists of a lamp and some visual objects on a table. The lamp has a shade, so not all of the objects, nor all of the table, should be lit by the lamp. The interior (but not the exterior) of the lamp shade should be lit by the lamp as well (in this example, the lamp shade is fully opaque). We know that Java 3D will not provide the occlusion for us. Using just a bounding volume, the influence of the light can be controlled, but it could be very difficult, especially if the lit and unlit objects are near to each other, or move.

Specifying scope limitations for the light allows you to control complex influence limitations more easily. The only consideration is to keep the lit and unlit visual objects in separate parts of the scene graph. Your initial thought might be to start building the scene graph with BranchGroups for the lit and unlit items, but that is not often necessary nor recommended for most applications.

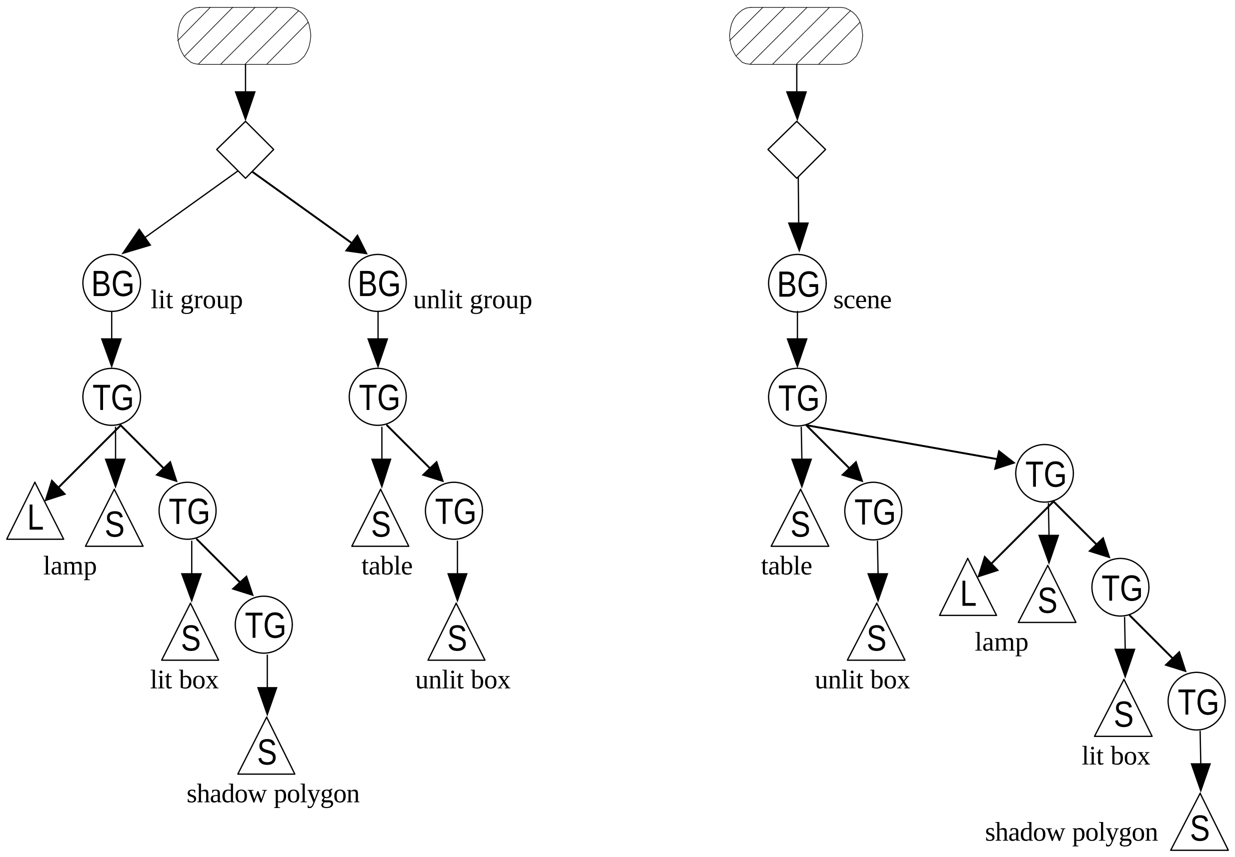


Figure 6-24 Two Possible Scene Graphs for the Light Scoping Example.

The scene graph diagram on the left of Figure 6-24 shows a naïve approach to scene graph construction. The organization is not natural and will be difficult to manipulate in an animated application. For example, if the table moves, the lamp and other objects should move with it. In the left scene graph, moving the table (via TransformGroup manipulation) will not move the lamp or the lit box; only the unlit box moves with the table.

The scene graph diagram on the right represents a more natural organization for the scene. The objects on the table are children of the TransformGroup that positions the table. If the table moves (via TransformGroup manipulation) the objects on the table move with it.

The example scene is created in `examples/light/LightScopeApp.java`. Figure 6-25 shows two images rendered from the example program. The left image uses light scoping to limit the influence of the lamp light to the lamp and the lit box. The right image does not use scoping; therefore, the lamp light illuminates the 'unlit box'.

The bright area under the lamp (not represented in either scene graph diagram) is a polygon placed just above the table top. This bright polygon represents the part of the table that is lit by the lamp. The bright area appears lighter than the rest of the table (even in the right image of Figure 6-9) because its normals are more closely aligned with the point light of the lamp.

The shadow does not appear lit in either image of Figure 6-25 because its diffuse material property is black. The shadow can be created through the use of scoping only if an additional group node is used in the scene graph.

The shadow in this scene was created by hand. Techniques for automatically (even dynamically) creating shadows are discussed in Section 6.7.1. Note that the techniques discussed in Section 6.7.1 may apply to automatically creating the bright area of the table.

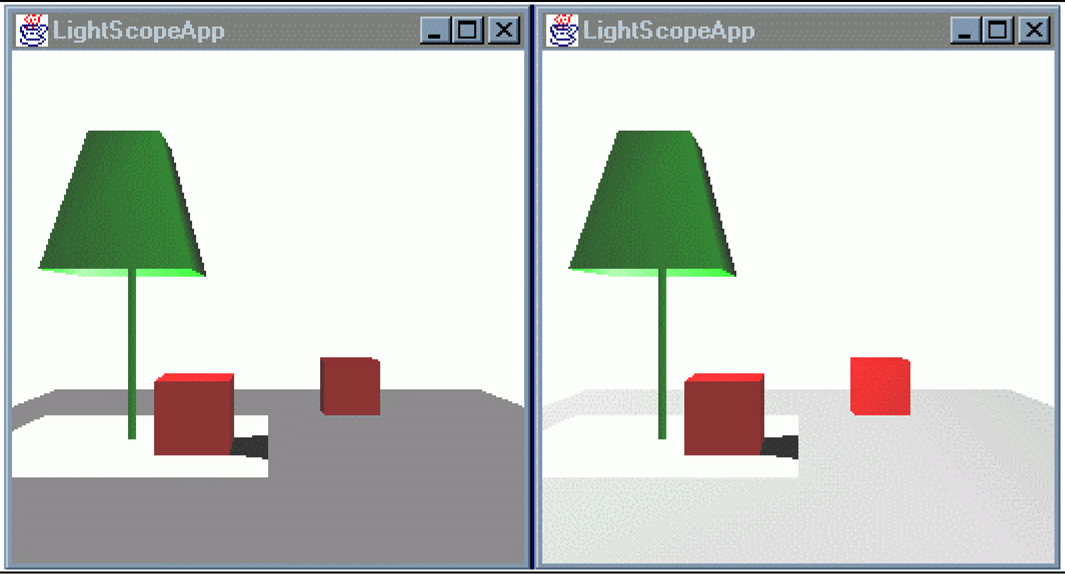


Figure 6-25 Light Scoping Example Scene With (left) and Without (right) Scoping of the Lamp Light.

Also not represented in either scene graph diagram are three additional light sources: two directional light sources and an ambient light source. These are necessary to simulate the light of a natural scene (see "Inter-object Effects Not Considered" subsection of section 6.1 on page 6-3, and Section 6.3.5).

The following reference block shows the methods of Light used to specify scoping limitations and the use of BoundingLeaf objects to specify an influence bounds.

Light Method Summary (partial list)

More Light class methods appear in Section 6.3 (page 6-9).

`void addScope(Group scope)`
Appends the specified scope to this node's list of scopes.

`java.util.Enumeration getAllScopes()`
Returns an Enumeration object of all scopes.

`void insertScope(Group scope, int index)`
Inserts the scope specified by the group node at the specified index location.

`int numScopes()`
Returns a count of this lights' scopes.

`void removeScope(int index)`
Removes the node's scope at the specified index location.

`void setInfluencingBoundingLeaf(BoundingLeaf region)`
Sets the Light's influencing region to the specified bounding leaf. Setting a BoundingLeaf overrides an influencing bounds object.

`void setScope(Group scope, int index)`
Sets the Light's hierarchical scope at the index specified. By default, lights are scoped only by their region of influence bounds.

One other benefit of using scopes to limit a light's influence: it may reduce rendering time. Calculating the bounds intersection for a visual object with the influencing bounds of a light is more complex than determining the scope of a light. Be aware that neither using influencing bounds nor scopes will limit a light's influence to part of a visual object.

6.7 Creating Glow-in-the-Dark Objects, Shadows, and Other Lighting Issues

The previous sections cover the typical applications of lights in Java 3D applications. This section covers some of the less used features and techniques.

6.7.1 Glow-in-the-Dark Objects

The Material object allows the specification of an emissive color. This can be used to create the effect of a glow-in-the-dark object. Having an emissive color does not make the visual object a light source; it will not illuminate other visual objects. Emissive material is also useful in special applications, such as indicating a special object or an object that has been picked (Chapter 5).

Figure 6-26 shows the scene from the light scoping example program where the unlit box has been given emissive color (most noticeable in color). Compare this image to the left image of Figure 6-25. As you can see, the use of emissive color only changes the visual object that has it. This example also demonstrates that the effective use of emissive color, as with most of the lighting parameters, is more of an artistic than programming problem.

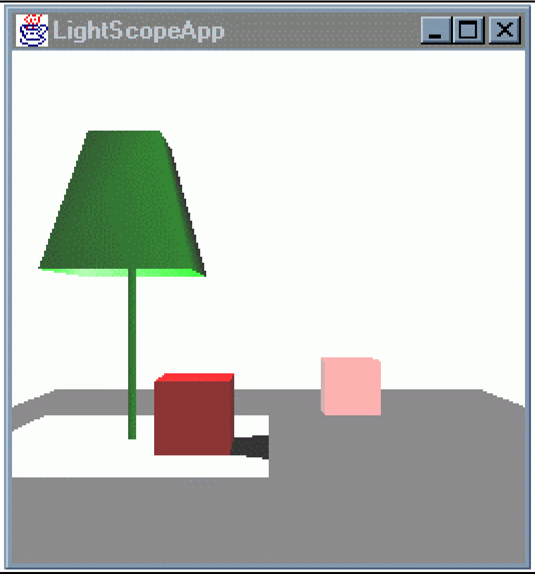


Figure 6-26 A Glow-in-the-Dark Object Example.

6.7.2 Computing Shadows

The complexity of computing shadows is so great it is not part of any real-time graphics system. The complexity comes from computing whether or not light from a light source reaches a vertex. Every other polygon from every other visual object must be considered in computing the answer.

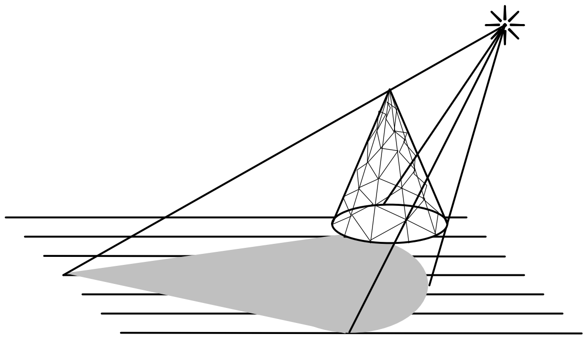


Figure 6-27 Projecting the Shadow of a Visual Object to a Plane for One Light

Shadowing is much more complex in reality. Light sources are not purely directional nor perfect point sources. Consequently, shadows do not have sharp edges. Ignoring reality, as we often do in graphics, let's take a look at ways for simulating shadows.

6.7.3 Creating Shadows

There are two basic parts to simulating (or faking) shadows: computing where the shadows are, and creating geometry (or textures) to serve as the shadow. There are several ways of computing the location of shadow, but the details of the various shadowing techniques are beyond the scope of this tutorial. The next two sections cover two general techniques for creating shadow content. Section 6.7.4 presents a simple example program that calculates the position of shadows.

Shadow Polygons

A polygon specified without material properties can be used as a shadow polygon, called a colored shadow polygon. The color of the shadow polygon, specified by either geometry or with a ColoringAttributes object, is chosen to appear as the object in shade. Shadow polygons specified in this way may look fake in complex scenes.

Shadow polygons specified with material properties but outside of the influence of one or more light objects are called shaded shadow polygons. Shaded shadow polygons shaded by light objects which influence it which appear more realistic. Obviously, specifying a shaded shadow polygon is more complex than specifying a colored shadow polygon.

No matter how a shadow polygon is specified, the position of the shadow polygon is just above, or in front of, the polygon that is shadowed. While adding shadow polygons normally does not result in more polygons to render (because of the occlusion of other polygons) it does create more objects in the virtual universe which can degrade rendering performance.

Instead of creating shadow polygons, shadows can be created by changing the influence of lights to exclude polygons 'in the shade'. Scoping of lights is particularly useful for this purpose. However, since influence is determined on a per object basis, it can be complex to calculate how to subdivide visual objects which are partially shaded.

Shadow Textures

As mentioned earlier, natural shadows are complex. A natural shadow rarely has a straight edge and a constant shade. Texturing can be used to make more realistic shadows. There are two basic ways of using texturing in creating shadows: applying texture to shadow polygons, or applying textures to visual objects.

Since texturing has not been covered yet (Chapter 7), and calculating shadow textures (even off-line) is difficult (and beyond the scope of this tutorial) this is a subject left for another book.

Moving Object, Moving Shadows

Keep in mind that adding shadows to an application makes the application much more complex. For example, when a cube with a shadow rotates, the shadow rotates and changes shape. For that matter, moving lights make shadows move too. In either case, movement adds another level of complexity to the programming of shadows.

6.7.4 Shadow Example Program

The program `example/light/ShadowApp.java` gives an example of how simple shadow polygons can be created. The program defines a class to create the shadow polygons. The shadow class creates one shadow polygon for each geometry given as input. Code Fragment 6-3 shows the `SimpleShadow` class used to create shadow polygons in `ShadowApp.java`. Figure 6-28 shows the rendered scene with a shadow.

```
public class SimpleShadow extends Shape3D {
    SimpleShadow(GeometryArray geom, Vector3f direction,
                 Color3f col, float height) {

        int vCount = geom.getVertexCount();
        QuadArray poly = new QuadArray(vCount, GeometryArray.COORDINATES
                                           | GeometryArray.COLOR_3
                                           );

        int v;
        Point3f vertex = new Point3f();
        Point3f shadow = new Point3f();
        for (v = 0; v < vCount; v++) {
            geom.getCoordinate(v, vertex);
            shadow.set( vertex.x + (vertex.y-height) * direction.x,
                        height + 0.0001f,
                        vertex.z + (vertex.y-height) * direction.y);
            poly.setCoordinate(v, shadow);
            poly.setColor(v, col);
        }

        this.setGeometry(poly);
    }
```

Code Fragment 6-3 Shadow Class for Making Shadow Polygons.

A number of assumptions made in SimpleShadow class (to make it easy) limit the applications of this class. SimpleShadow is limited in that it: only projects to planes, only considers one light, only does some orientations, doesn't consider the dimensions of the plane it is projecting on to. Writing a general purpose shadow calculation class is a significant undertaking.

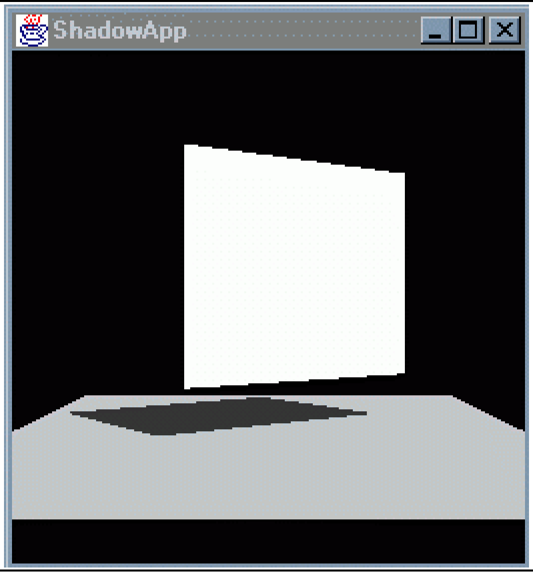


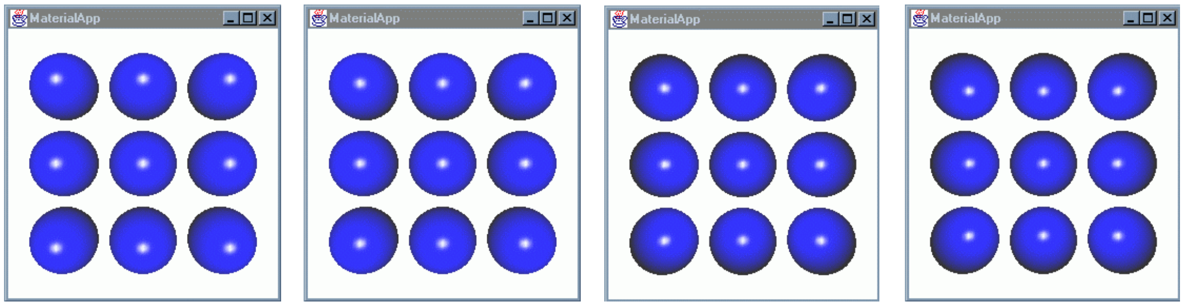
Figure 6-28 Scene Produced by ShadowApp.java Demonstrating Automatic Shadowing in Java 3D

6.7.5 Advanced Topic: The Role of the View Object in Shading

The view (or views) associated with a scene graph plays a variety of roles in how a scene is rendered. This section does not explain all of the roles of the View object. The Java 3D API Specification provides a complete reference to the View class. This section only mentions two methods of the View class useful in understanding the shading of visual objects.

As mentioned in the "Local Eye versus Infinite Eye Vectors" section (subsection of 6.1 on page 6-3), the eye vector is constant by default. This is known as an infinite eye vector. That is, the scene is rendered as if it were viewed from infinity. Having an infinite eye significantly reduces the rendering computation.

However, the resulting image may look incorrect. Figure 6-29 shows images rendered from one scene using an infinite eye and local eye viewing using different light sources.



(a)directional light, infinite eye | (b) directional light, local eye | (c) point light infinite eye | (d) point light local eye

Figure 6-29 Infinite Eye Versus Local Eye Rendering with Directional and Point Light Sources.

To fully appreciate the images of Figure 6-29 you need to know the geometry of the scene. The scene is of nine spheres in a planar organization. Each of the images are viewed with the same field of view from the same position. The only variables are whether the light is a DirectionalLight or a PointLight, and whether the eye is infinite or local. The DirectionalLight has direction (0, 0, -1), the PointLight is positioned at (0, 0, 1).

In Figure 6-29 images (a) and (c) are rendered with an infinite eye. In these images, the eye vectors are constant, so the specular reflections are basically in the same position for each sphere. Images (b) and (d) in Figure 6-29 are rendered with a local eye. The eye vectors vary in these images, so the specular reflections are in different position for each sphere. Note also, the diffuse reflection (blue) on the sphere varies with the light source only. The eye vector only plays a role in the specular reflection calculation.

Again, the infinite eye viewing feature is used to reduce computation, and therefore time, in rendering. Image (a) of Figure 6-29 takes the least time to render and image (d) takes the most time. Images (b) and (c) take about the same amount of time, which is less that the time for (d), but more that the time for (a). The actual time for rendering varies with the system used. The difference is most pronounced on systems that render in software.

View Method Summary (partial list, methods related to shading)

The View object contains all parameters needed in rendering a three dimensional scene from one viewpoint.

`void setLocalEyeLightingEnable(boolean flag)`
Sets a flag that indicates whether the local eye point is used in lighting calculations for perspective projections.

`void setWindowEyepointPolicy(int policy)`
Sets the view model's window eye point policy to one of: `RELATIVE_TO_FIELD_OF_VIEW, RELATIVE_TO_SCREEN, RELATIVE_TO_WINDOW`

The View object can be gotten from a SimpleUniverse using the appropriate methods. Then the view object can be manipulated as necessary as in the following example.

```
      SimpleUniverse su = new SimpleUniverse(canvas);

      su.getViewer().getView().setLocalEyeLightingEnable(true);
```

6.8 Chapter Summary

This chapter presents the Java 3D features for rendering objects using lights. Section 6.1 explains the lighting model, gives a recipe for constructing lit scenes, and presents a simple example using a light. Section 6.2 shows an example program consisting of a single sphere and two light sources. Section 6.3 provides details of the light classes available in the Java 3D API. Section 6.3.5 illustrates the differences among the light source classes through several example programs. Section 6.4 shows the Material object API interface. Section 6.5 discussed various aspects of creating surface normals for geometric content. Section 6.6 presents advanced topics on specifying the influence of lights including scene graph hierarchical scoping. Section 6.7 discusses advanced topics such as creating shadows. You are reading Section 6.8, the chapter summary.

This chapter explains the classes which are shaded in the following Java 3D API class hierarchy diagram.

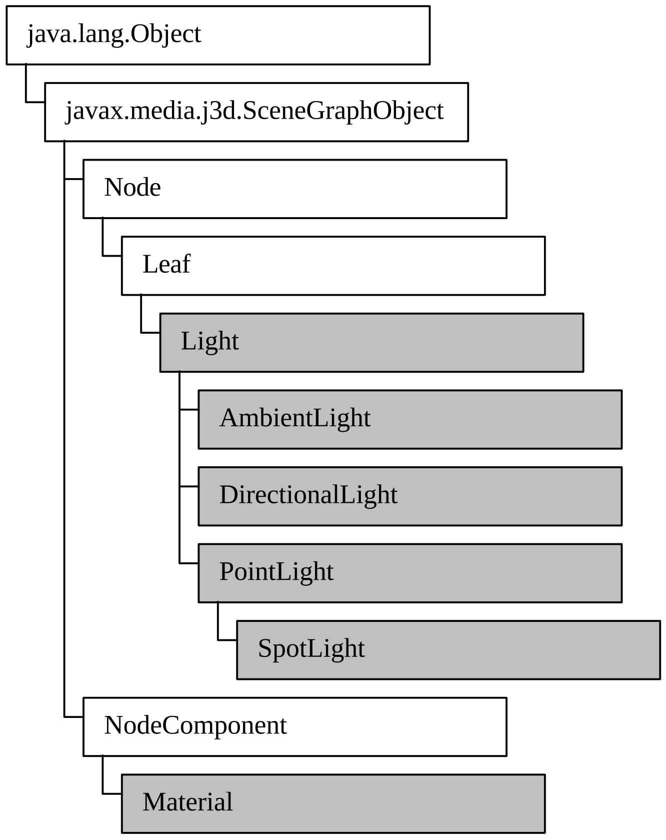


Figure 6-30 Java 3D API Hierarchy for Classes Covered in this Chapter.

6.9 Self Test

1. Add a green DirectionalLight pointing up to the `LitSphereApp` to illustrate additive color mixing with all three primary colors. Don’t forget to add the light to the scene graph and set the influencing bounds for the new light source object. Which two primary colors make yellow?
2. To learn more about the interaction between material colors and light colors, create a scene with red, green, and blue visual objects. Light the objects with a single color light. What did you see? You may use LitSpereApp.java or MaterialApp.java as a starting point.

3. Using LightScopeApp.java as a starting point (see Section 6.6.2), change the program to create the shadow of the lit box through the use of scoping only.
4. Using `ShadowApp.java` as a starting point, find situations where the shadow polygon produced by `SimpleShadow` is not correct, then fix `SimpleShadow` to make it work. (You could spend many hours working on this.)