```
            Double.MAX_VALUE));
        root.addChild(light);

        root.compile();

        return root;
    }
}
```

This program creates two spheres of radius 40 cm and positions them 50 cm behind the origin. The sphere with the red material is moved 50 cm in the negative x direction and the sphere with the blue material is moved 50 cm in the positive x direction. A single point light with the default parameters, white colour (1.0, 1.0, 1.0) and no attenuation, is placed at the origin in order to illuminate the two spheres. The output generated when this program is executed is illustrated in Figure 2.32.

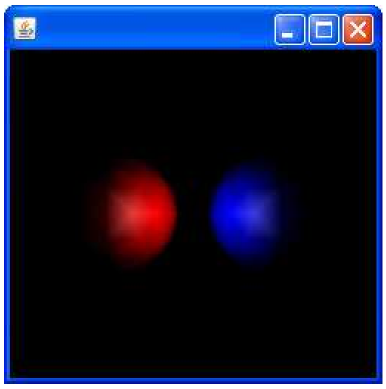


Figure 2.32: An examples of two spheres located left and right of the origin illuminated by a `PointLight` that is located at the origin.

2.7.9.3 `SpotLight`

The `SpotLight` class specifies an attenuated light source located at a fixed point in space that radiates light in a specified direction. The `SpotLight` class extend `PointLight` to include this additional functionality. A `SpotLight` node has the same attributes as a `PointLight` node, with the addition of the following:

- Direction - The axis of the cone of light. The default direction is (0.0, 0.0, -1.0), i.e. into the screen. The direction of the spotlight is only significant when the spread angle is not π radians (the default value for this attribute).
- Spread angle - The angle in radians between the direction axis and a ray along the edge of the cone of light.
  - Note that the angle of the cone at the apex is twice this angle.
  - The range of values for the spread angle is $[0.0, \frac{\pi}{2}]$ radians, with a special value of π radians.
  - Spread angle values lower than 0 are clamped to 0 and values greater than $\frac{\pi}{2}$ are clamped to $\frac{\pi}{2}$.

- Concentration - Specifies how quickly the light intensity attenuates as a function of the angle of radiation as measured from the direction of radiation. The light's intensity is highest at the centre of the cone and is attenuated towards the edges of the cone by the cosine of the angle between the direction of the light and the direction from the light to the object being lit, raised to the power of the spot concentration exponent. The higher the concentration value, the more focused the light source. The range of values is [0.0, 128.0]. The default concentration is 0.0, which provides uniform light distribution.

A `SpotLight` contributes to diffuse and specular reflections, which depend on the orientation and position of an object's surface. A `SpotLight` does not contribute to ambient reflections. An illustration of the spread angle and the concentration for a `SpotLight` node are illustrated in Figure 2.33.

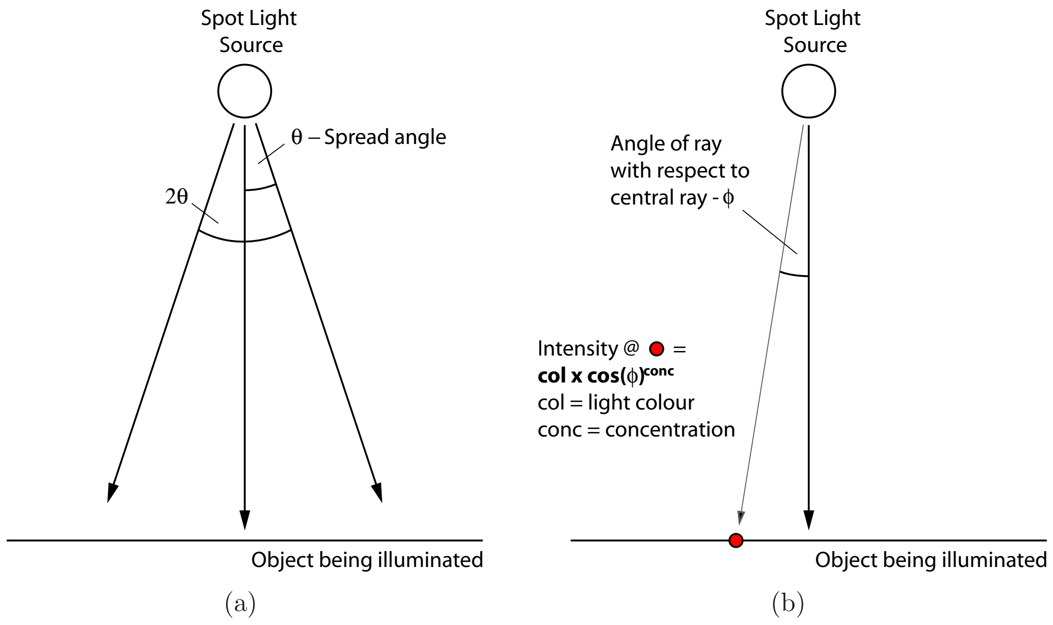


Figure 2.33: An illustration of two of the main attributes associated with a spot light. The spread angle (a) and the concentration (b).

The most comprehensive constructor for a `SpotLight` has the following format:

- `SpotLight(boolean on, Color3f color, Point3f position, Point3f attenuation, Vector3f dir, float spreadAngle, float conc)`
  Creates a new instance of a `SpotLight` object with the specified attributes.

The `SpotLight` class also defines methods to set or retrieve its various attributes:

- `void setConcentration(float concentration)`
  Sets the concentration for this `SpotLight` object.

- `float getConcentration()`
  Retrieves the concentration for this `SpotLight` object.

- `void setDirection(Vector3f direction)`
  Sets the direction for this `SpotLight` object to the specified vector.

- `void getDirection(Vector3f direction)`
  Retrieves the direction for this `SpotLight` and stores it in the specified vector.

- `void setSpreadAngle(float spreadAngle)`
  Sets the spread angle for this `SpotLight` object. The minimum spread angle is 0 radians and the maximum spread angle is $\frac{\pi}{2}$ radians.

- `float getSpreadAngle()`
  Retrieves the spread angle for this `SpotLight` object.

Note: If you are more comfortable specifying angles in degrees then the following method may be useful:

- `double Math.toDegrees(double radians)`
  Converts the specified angle from radians to degrees.

- `double Math.toRadians(double degrees)`
  Converts the specified angle from degrees to radians.

The `SpotLight` class also defines a series of capability bits that can be used to enable access to its attributes after the scene graph has gone live, these include:

- `ALLOW_CONCENTRATION_READ`
  Indicates that this `SpotLight` allows read access to its concentration information after the scene graph has gone live.

- `ALLOW_CONCENTRATION_WRITE`
  Indicates that this `SpotLight` object allows write access to its concentration information after the scene graph has gone live.

- `ALLOW_DIRECTION_READ`
  Indicates that this `SpotLight` object allows read access to its direction information after the scene graph has gone live.

- `ALLOW_DIRECTION_WRITE`
  Indicates that this `SpotLight` object allow write access to its direction information after the scene graph has gone live.

- `ALLOW_SPREAD_ANGLE_READ`
  Indicates that this `SpotLight` object allows read access to its spread angle information after the scene graph has gone live.

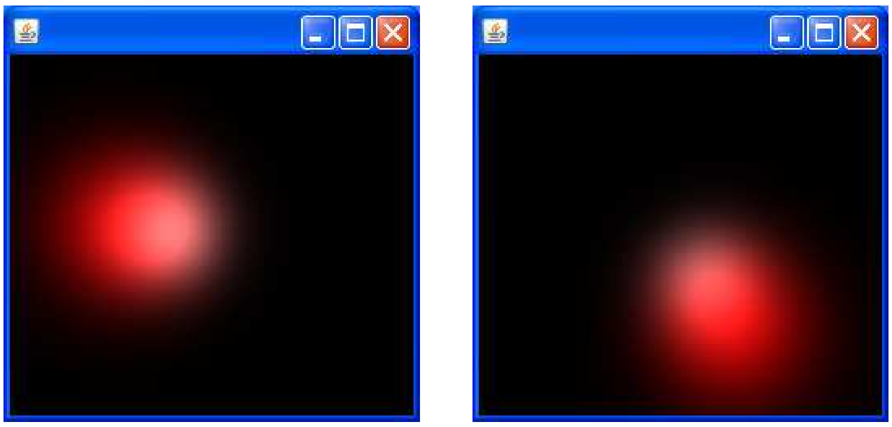
- `ALLOW_SPREAD_ANGLE_WRITE`
  Indicates that this `SpotLight` object allows write access to its spread angle information after the scene graph has gone live.

The following example demonstrates how a `SpotLight` can be used to illuminate an object.

```
import javax.media.j3d.*;
import javax.vecmath.*;
import java.lang.*;

import com.sun.j3d.utils.behaviors.mouse.MouseRotate;
```

```java
import com.sun.j3d.utils.geometry.*;

public class SpotLightExample extends BasicScene
{
  public static void main(String args[]){new SpotLightExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    Transform3D transform1 = new Transform3D();
    transform1.setTranslation(new Vector3f(0.0f, 0.0f, -20.0f));
    TransformGroup tg1 = new TransformGroup(transform1);
    root.addChild(tg1);

    // Red appearance
    Appearance appearance1 = new Appearance();
    Material material1 = new Material();
    material1.setDiffuseColor(new Color3f(1.0f, 0.0f, 0.0f));
    appearance1.setMaterial(material1);

    // A detailed sphere with a large radius
    Sphere sphere1 = new Sphere(15.0f, Primitive.GENERATE_NORMALS,
        500, appearance1);
    tg1.addChild(sphere1);

    TransformGroup tg2 = new TransformGroup();
    tg2.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    root.addChild(tg2);

    // A mouse rotate behaviour to control the direction of the light
    MouseRotate rotate = new MouseRotate();
    rotate.setTransformGroup(tg2);
    tg2.addChild(rotate);
    rotate.setSchedulingBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));

    // A spot light
    SpotLight light = new SpotLight();
    light.setConcentration(45f);
    light.setSpreadAngle((float)Math.toRadians(30));
    light.setInfluencingBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));
    tg2.addChild(light);

    root.compile();

    return root;
  }
}
```

This program begins by creating a large detailed sphere of radius 15 metres that is located 20 metres behind the origin. The creation of a detailed sphere is achieved by specifying a high number of subdivisions. The repositioning of the sphere with respect to the origin is achieved using a `TransformGroup`. A `SpotLight` is then created with a white colour, a concentration of 45.0 and a spread angle of 30 degrees. The `SpotLight` is attached to a `TransformGroup` which is associated with a `MouseRotation` behaviour. This setup enables the direction that the `SpotLight` is shining to be controlled by the mouse. The type of renderings obtained when this program is executed are illustated in Figure 2.34.



(a) (b)

Figure 2.34: An illustration of a highly detailed `Sphere` primitive illuminated by a `SpotLight` with a concentration of 45.0 and a spread angle of 30 degrees.

2.7.9.4 `AmbientLight`

An `AmbientLight` is used to represent light that appears to come from all directions. The `AmbientLight` class extends the `Light` class and consequently has the same attributes including colour, influencing bounds, scopes and a flag indicating whether the light source is on or off. Ambient reflections do not depend on the orientation or position of a surface. Ambient light only has an ambient reflection component. It does not have either diffuse or specular reflection components.

An `AmbientLight` object can be created using the following constructor:

- `AmbientLight(boolean lightOn, Color3f color)`
  Creates a new `AmbientLight` object with the specified state and colour information.

The `AmbientLight` class does not define any additional methods or capabilities as all of its required functionality is defined by the `Light` class.

The following example demonstrates how an `AmbientLight` can be used to illuminate an object.

```java

import javax.vecmath.*;
import javax.media.j3d.*;
import com.sun.j3d.utils.geometry.*;
```

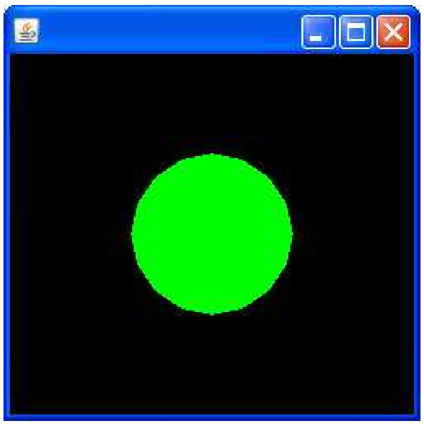
```
public class AmbientLightExample extends BasicScene
{
  public static void main(String args[]){new AmbientLightExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    // Create an yellow ambient light
    AmbientLight light = new AmbientLight(new Color3f(1.0f, 1.0f, 0.0f));
    BoundingSphere bounds = new BoundingSphere(new Point3d(0.0, 0.0, 0.0),
        Double.POSITIVE_INFINITY);
    light.setInfluencingBounds(bounds);
    root.addChild(light);

    // Create a cyan material
    Appearance appearance = new Appearance();
    Material material = new Material();
    material.setAmbientColor(new Color3f(0.0f, 1.0f, 1.0f));
    appearance.setMaterial(material);

    // Create a sphere of radius 40 cm
    Sphere sphere = new Sphere(0.4f, appearance);
    root.addChild(sphere);

    root.compile();

    return root;
  }
}
```

The operation of this program is quite straightforward. An `AmbientLight` is created and positioned at the origin (the default location). The colour of the `AmbientLight` is set to yellow in the constructor. A `Sphere` primitive is created with a radius of 40 cm. The sphere has a `Material` with an ambient colour of cyan. Consequently, the colour of the sphere under the ambient light will be green. This is illustrated in Figure 2.35.

2.7.10 Texture Mapping

Texture mapping changes the appearance of a shape by wrapping an image around the structure of the shape. The use of an image, or texture, in this way enables the creation of extremely detailed content in a relatively straightforward manner. For example, a table textured with a wood grain texture would look more realistic than if a solid brown colour were used. Textures are applied, or mapped, to a surface using data that relates each vertex in the geometry to a location in the texture. The locations in the texture are specified using texture coordinates and the texture is considered to be a rectangular array of colour values called texels.

Note: A texel is to a texture what a pixel is to a picture.



(a)

Figure 2.35: A sphere with a material that has an ambient colour of cyan illuminated by yellow ambient light. This results in a sphere that appears to have a green colour.

2.7.10.1 Texture Coordinates

The position of a pixel in an image is represented using x and y values. In a similar way, the position of a texel in a texture is represented using s and t values. These values are known as texture coordinates. The s coordinate corresponds to the horizontal axis of the texture and the t coordinate corresponds to the vertical axis of the texture. The lower left hand corner of the texture is at $(0, 0)$ and the upper right hand corner of the image is at $(1, 1)$. This coordinate system is illustrated in Figure 2.36.

Note: If a texture image is non-square the texture coordinate still have the same range. For example, if the texture is a 128×256 image the top right hand corner of the texture will have the coordinates $(1, 1)$ and not (0.5, 1).

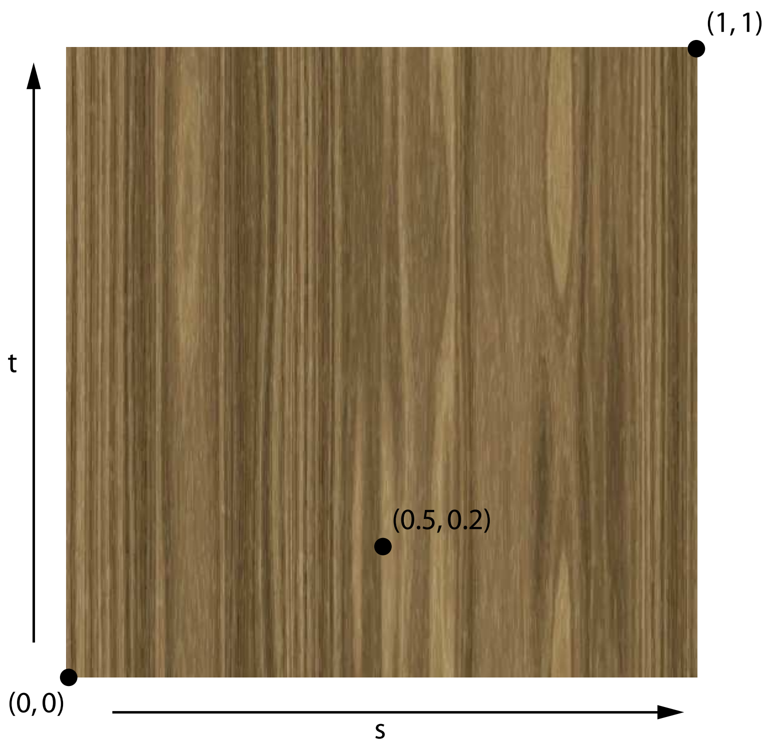
Texture mapping stretches the texture to make the texture locations specified by the texture coordinates line up with the texture coordinates assigned to the vertices of the geometry being texture mapped. Texture mapping is controlled by several components:

- The `Texture` appearance component controls the texture image.
- The `TextureAttributes` appearance component control how the texture is applied to the surface of the geometry that is being texture mapped.
- The texture coordinates are specified by the `Geometry`. If the `Geometry` does not have texture coordinates, a `TextureCoordGeneration` appearance component can be used to generate texture coordinates from the geometric coordinates.

2.7.10.2 `Texture`

The base class for textures is `Texture`. The `Texture` class has two main subclasses:

- `Texture2D` - specifies a 2D image that is to be mapped to the exterior of a particular geometry.



(a)

Figure 2.36: An illustration of the texture coordinate system. The location of each texel is represented by a pair of texture coordinates (s, t). The s coordinate represents the location of the texel in the horizontal direction and the t coordinate represents the location of the texel in the vertical direction. The origin of the texture coordinate system is located at the bottom left hand corner of the texture.

- `Texture3D` - specified a 3D or volumetric texture that can be used in volume rendering.

The most straightforward method for the generation of a `Texture2D` object is to use a `TextureLoader`. The `TextureLoader` loads the texture and sets up several of its basic properties. A `TextureLoader` object can be created using one of the following constructors:

- `TextureLoader(BufferedImage image)`
  Constructs a `TextureLoader` object using the specified `BufferedImage` and the default format RGBA.

- `TextureLoader(Image image, Component observer)`
  Constructs a `TextureLoader` object using the specified `Image` and the default format RGBA.

- `TextureLoader(String filename, Component observer)`
  Constructs a `TextureLoader` object that will load a texture from the specified file location.

- `TextureLoader(URL url, Component observer)`
  Constructs a `TextureLoader` object that will load a texture from the specified URL.

Note: In the last three versions of the constructor listed here, an `ImageObserver` object must be specified. This is required to monitor the progress of `Image` object that are in the process of being loaded.

It is clear that a `TextureLoader` object can be used to load a texture image from a variety of sources. The texture loader also supports the image formats supported by the JDK i.e. GIF and JPEG. If the optional Java Advanced Imaging (JAI) package is installed then JAI will be used to load the texture image and consequently the loader will support BMP, FlashPix, PNG, PNM and TIFF file formats.

It is also possible to specify a series of flags in the constructor, for example:

- `TextureLoader(BufferedImage image, int flags)`
  Constructs a `TextureLoader` object using the specified `BufferedImage` object, the specified flag options and the default format RGBA.

The flags are used to specify options for the loader. The options that can be specified by the flags are integers that can be OR'd together. The possible flag values are:

- `GENERATE_MIPMAP` - Tells the `TextureLoader` to create the texture with multiple levels of resolution called mipmaps that are used when the texture is viewed at a variety of scales.
- `BY_REFERENCE` - Specifies that the `ImageComponent2D` object representing the texture will access the image data by reference.
- `Y_UP` - Indicates that the `ImageComponent2D` object representing the texture will have a y-orientation of y up, meaning that the origin of the image is in the lower left hand corner (i.e. so that the image coordinate system corresponds to the texture coordinate system).

It is also possible to specify an image format in the constructor for a `TextureLoader` object, for example:

- `TextureLoader(BufferedImage image, int format)`
  Constructs a `TextureLoader` object with the specified `BufferedImage` and the specified image format.

The format parameter is an advanced option that is used to specify the internal format of the image. The default format is RGBA, i.e. this is the format that is used if the version of the constructor being used does not have a format parameter. The RGBA format indicates that each texel has red, green, blue and alpha components. A variety of other formats are also available, for example ALPHA, indicates that only the transparency values of the loaded image are to be used.

Once a `TextureLoader` object has been created the `Texture` object that it represents can be obtained using the following method:

- `Texture getTexture()`
  Returns the relevant `Texture` object or null if the texture image failed to load.

Note: If the width or height of the image is not a power of 2 (i.e. 32, 64, 128, 256, etc.), then the image is scaled so that its dimensions are a power of two.

The other methods provided by the `TextureLoader` class can be used to load an `ImageComponent2D` representation of the texture image. This can be used in conjunction with the `Raster` geometry and `Background` environment node. These methods have the following format:

- `ImageComponent2D getImage()`
  Returns an `ImageComponent2D` representation of the texture image.

- `ImageComponent2D getScaledImage(float xScale, float yScale)`
  Returns a scaled `ImageComponent2D` representation of the texture image that has been scaled by the specified horizontal and vertical scale factors.

- `ImageComponent2D getScaledImage(int width, int height)`
  Returns a scaled `ImageComponent2D` representation of the texture image that has the specified dimensions.

The following program demonstrates how a texture can be applied to a simple triangle:

```
import javax.media.j3d.*;
import com.sun.j3d.utils.image.*;

public class TextureCoordinateExample extends BasicSceneWithMouseControl
{
  public static void main(String args[]){new TextureCoordinateExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    Texture woodTexture = null;

    try
    {
      // Load the wood texture from the local file syste,
      TextureLoader loader = new TextureLoader("wood.jpg", this);
      woodTexture = loader.getTexture();
    }
    catch(Exception e){System.out.println(e.toString());}

    Appearance appearance = new Appearance();
    appearance.setTexture(woodTexture);

    float [] coordinates = {-0.5f, -0.5f, 0.0f,
        0.5f, -0.5f, 0.0f,
        0.0f, 0.5f, 0.0f};

    // Define the texture coordinates for the vertices
    float [] texCoords = {0.0f, 0.0f,
        1.0f, 0.0f,
      0.5f,  1.0f};

    // Create a geometry array from the specified coordinates
    GeometryArray geometryArray = new TriangleArray(3,
    GeometryArray.COORDINATES|GeometryArray.TEXTURE_COORDINATE_2);

    geometryArray.setCoordinates(0, coordinates);
    geometryArray.setTextureCoordinates(0,0,texCoords);

    // Create a Shape3D object using the GeometryArray
    Shape3D shape = new Shape3D(geometryArray, appearance);
    root.addChild(shape);

    root.compile();

    return root;
  }
}
```

The program begins by creating a `Texture2D` object that represents the wood texture stored in the file `wood.jpg`. This is achieved using a `TextureLoader` object. An `Appearance` object is then created and its texture is set to be the loaded `Texture2D` object. A set of coordinates are then defined that represent a simple triangle located at the origin. A set of texture coordinates are subsequently defined. These associate the vertices of the triangle with points in the texture image. The first vertex is associated with the the bottom left corner of the texture image, the second vertex is associated with the bottom right corner of the image and the third vertex is associated with the central point at the top of the texture image. A `TriangleArray` object is then created and the `COORDINATE` and `TEXTURE_COORDINATE_2` flags are set to indicate that both coordinates and 2D texture coordinates will be specified for the vertices of the triangle array. A `Shape3D` object is then created using the previously discussed appearance and geometry. Finally, the resulting shape is added to the root of the scene graph. A rendering of the texture mapped triangle is illustrated in Figure 2.37.

The following example demonstrates how texture coordinates can be generated to example texture mapping for a primitive shape.

```

import javax.media.j3d.*;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.image.*;

public class TexturePrimitiveExample extends BasicSceneWithMouseControl
{
  public static void main(String args[]){new TexturePrimitiveExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

```

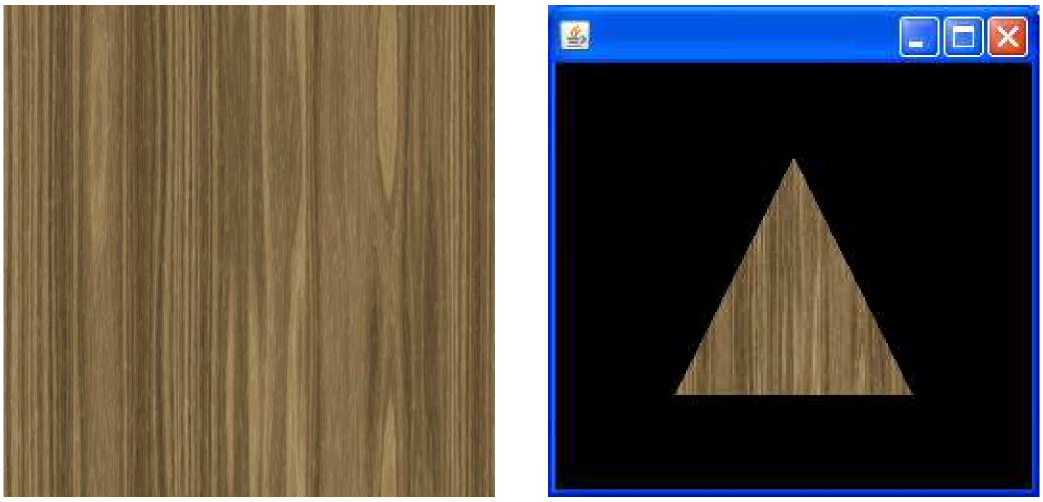


Figure 2.37: The original wood texture (a) and a triangle that has been texture mapped using the wood texture (b).

```
    Appearance appearance = new Appearance();

    Texture woodTexture = null;
    try
    {
      // Load the wood texture from the local file system
      TextureLoader loader = new TextureLoader("wood.jpg", this);
      woodTexture = loader.getTexture();
    }
    catch(Exception e){System.out.println(e.toString());}

    appearance.setTexture(woodTexture);

    // Create a Box primitive with texture coordinates
    Box box = new Box(0.2f, 0.05f, 0.6f,
        Box.GENERATE_TEXTURE_COORDS,
        appearance);
    root.addChild(box);

    root.compile();

    return root;
  }
}
```

The operation of this example is similar to the operation of the previous example. The wood texture is loaded and associated with the `Appearance` in the same way. A `Box` primitive is then created with a width of 20 cm, a height of 5 cm and a depth of 60 cm. The `GENERATE_TEXTURE_COORDS` is specified to indicate that the `Box` primitive should have texture coordinates associated with its vertices. Finally, the appearance with the associated wood texture is specified for use with the `Box` primitive. A rendering of the texture mapped `Box` is illustrated in Figure 2.38.

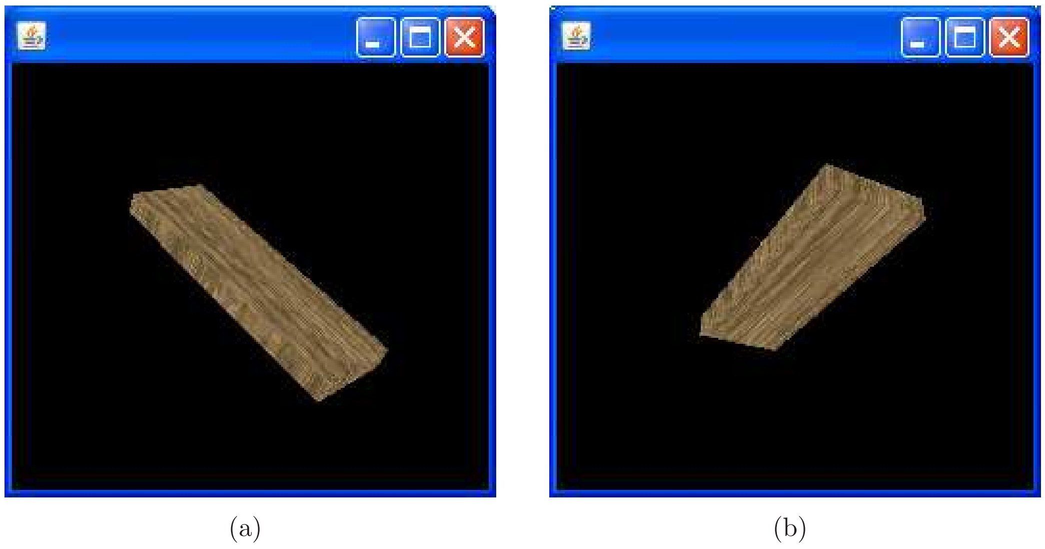


Figure 2.38: Sample renderings of the texture mapped `Box` primitive.

The texture image is the most important attribute of the `Texture` class. However, the `Texture` class has a variety of other attributes that define how the texture appears when it is viewed, for example:

- The state attribute allows the texture to be enabled or disabled.
- The boundary mode specifies how the texture appears for texture coordinates outside the range [0, 1].
- The texture filtering mode specifies how the texture is drawn when it is larger or smaller than its original size.

Texture mapping is enabled if all three of the following are true:

- The shape has texture coordinates
- The appearance has a texture image associated with it
- The texture is enabled

The texture is enabled using the following method:

- `void setEnable(boolean state)`
  Sets the state of the texture to the specified value. A value of true indicates that the texture is enabled and a value of false indicates that the texture is disabled.

The state information for a texture can be accessed after the scene graph has gone live provided that the relevant capabilities are enabled, these are:

- `ALLOW_ENABLE_READ`
  Indicates that this `Texture` object allows read access to its state information after the scene graph has gone live.

- `ALLOW_ENABLE_WRITE`
  Indicates that this `Texture` object allows write access to its state information after the scene graph has gone live.

The `Texture` class also defines similar capabilities for the other attributes that it supports.

The coordinates of a texture image always have values in the range [0, 1], however, the texture coordinates associated with the vertices of a geometry can have texture coordinates outside this range. The boundary mode of a texture specifies how texture coordinates outside the range [0, 1] are dealt with. There are two main types of boundary mode:

- `CLAMP` - clamps texture coordinate to be in the range [0, 1]. Texture boundary texels are used for values that fall outside this range.
- `WRAP` - repeat the texture by wrapping texture coordinates that are outside the range [0, 1]. Only the fractional portion of the texture coordinates will be used here. The integer portion in discarded, e.g. 1.5 would become 0.5.

The boundary mode for the horizonal and vertical directions are specified separately using the following methods:

- `void setBoundaryModeS(int mode)`
  Sets the horizontal boundary mode of this `Texture` object to the specified value.
- `void setBoundaryModeT(int mode)`
  Sets the vertical boundary mode of this `Texture` object to the specified value.

The following example demonstrated the operation of the two different boundary modes.

```
import javax.media.j3d.*;
import com.sun.j3d.utils.image.*;

public class TextureBoundaryModeExample extends BasicSceneWithMouseControl
{
  public static void main(String args[]){new TextureBoundaryModeExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    Texture earthTexture = null;

    try
    {
      // Create the texture and set it horizontal and vertical
      // boundary modes
      TextureLoader loader = new TextureLoader("earth.jpg", this);
      earthTexture = loader.getTexture();
      earthTexture.setBoundaryModeS(Texture.CLAMP);
      earthTexture.setBoundaryModeT(Texture.WRAP);
    }
    catch(Exception e){System.out.println(e.toString());}

    Appearance appearance = new Appearance();
    appearance.setTexture(earthTexture);

    float [] coordinates = {-0.5f, -0.5f, 0.0f,
        0.5f, -0.5f, 0.0f,
        0.5f, 0.5f, 0.0f,
        -0.5f, 0.5f, 0.0f};

    // Specify the texture coordiantes for the vertices
    float [] texCoords = {-1.0f, -1.0f,
        2.0f, -1.0f,
        2.0f, 2.0f,
        -1.0f, 2.0f};

    // Create a geometry array from the specified coordinates
    GeometryArray geometryArray = new QuadArray(4,
    GeometryArray.COORDINATES|GeometryArray.TEXTURE_COORDINATE_2);

    geometryArray.setCoordinates(0, coordinates);
    geometryArray.setTextureCoordinates(0,0,texCoords);

    // Create a Shape3D object using the GeometryArray
    Shape3D shape = new Shape3D(geometryArray, appearance);
    root.addChild(shape);

    root.compile();

    return root;
  }
}
```

This example defines a `QuadArray` geometry consisting of a single quadrilateral facing the viewer. The quadrilateral has sides that are one meter in length and it is centred at the origin. The horizontal and vertical texture coordinates assigned to the vertices of the quadrilateral are in the range [-1.0, 2.0], i.e. they extend outside the range defined for the texture image. The horizontal boundary mode for the texture image is set to `CLAMP` and the vertical boundary mode for the texture image is set to `WRAP`. The output obtained when this program is executed is illustrated in Figure 2.39.

The `Texture` class also defines filtering modes that specify how the resolution for the texture image is increased or decreased during rendering. When a texture is mapped to a piece of geometry there is rarely a one-to-one correspondence between the pixels of the rendered geometry and the pixels of the texture image. Instead one of the following situations occurs:

- **Magnification** - where a single pixel of the rendered geometry corresponds to a small portion of a texel.

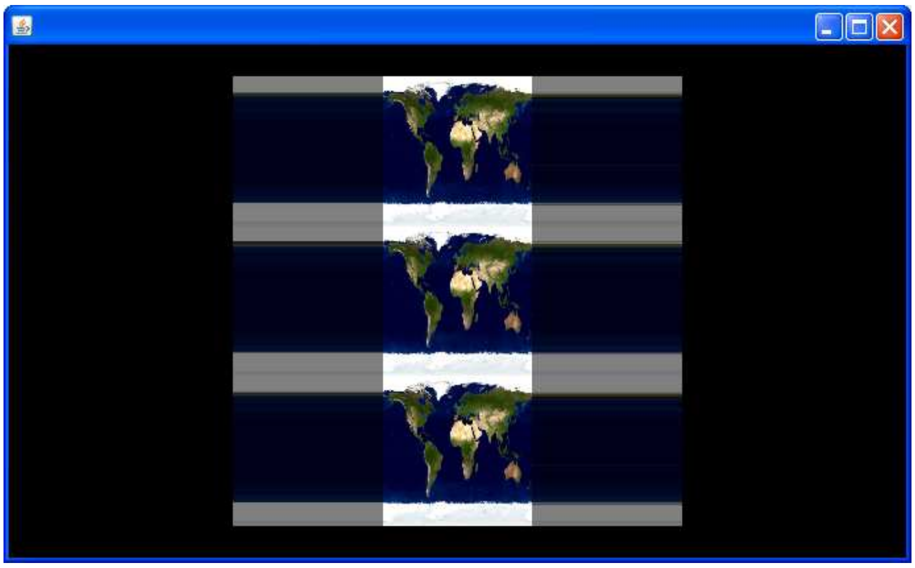


Figure 2.39: A rendering of a texture mapped quadrilateral where the horizontal texture mode has been set to `CLAMP` and the vertical texture mode has been set to `WRAP`.

- **Minification** - where a single pixel of the rendered geometry corresponds to an area of the texture, i.e. several texels.

The texture filtering mode specifies the quality of the process used to magnify or minify the texture. The better the quality the less "blocky" the texture mapped geometry will appear. The possible the minification filter modes include:

- `BASE_LEVEL_POINT`
  Selects the nearest point in the base level texture image.

- `BASE_LEVEL_LINEAR`
  Performs bilinear interpolation on the four nearest texels in the base level texture image.

- `FILTER4`
  Applies a used defined weight function to the nearest 4×4 texels in the base level texture image.
  Note: The weight function is set using the `setFilter4Func()` method of the `Texture` class.

- `FASTEST`
  Uses the fastest minification filter.

- `NICEST`
  Uses the minification filter that generates the most visually appealing results.

The minification filter mode associated with a `Texture` object can be set or retrieved using the following methods:

- `void setMinFilter(int minFilter)`
  Sets the minification filter for this `Texture` object to the specified value.

- `int getMinFilter()`
  Returns the minification filter value for this `Texture` object.

The magnification filter can also use the modes listed above. The minification filter and the magnification filter mode associated with a `Texture` object cam be set or retrieved using the following methods:

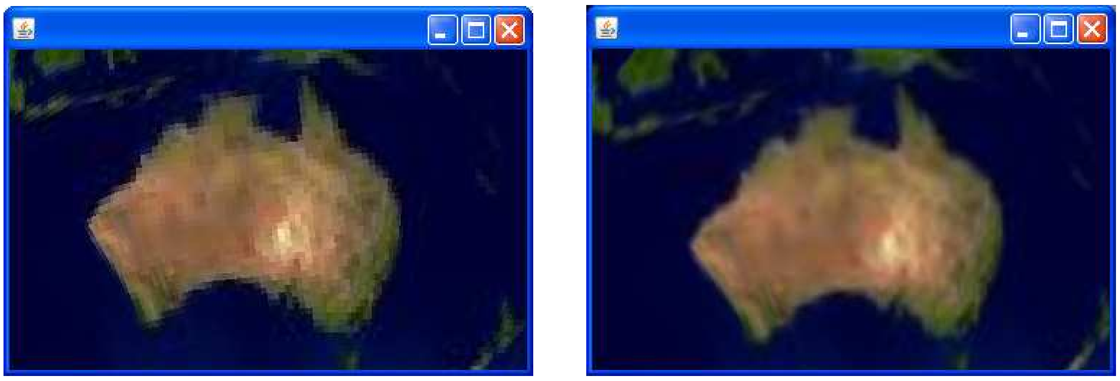
- `void setMagFilter(int magFilter)`
  Sets the magnification filter for this `Texture` object to the specified value.

- `int getMagFilter()`
  Returns the magnification filter value for this `Texture` object.

The following example demonstrates how a specific magnification filter mode can be used in conjunction with a `Texture` object.

```

import javax.media.j3d.*;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.image.*;

public class MagnificationFilterExample extends BasicSceneWithMouseControl
{
  public static void main(String args[]){new MagnificationFilterExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    Appearance appearance = new Appearance();

    Texture earthTexture = null;
    try
    {
      // Load the texture and set its magnification filter
      TextureLoader loader = new TextureLoader("earth.jpg", this);
      earthTexture = loader.getTexture();
      earthTexture.setMagFilter(Texture.BASE_LEVEL_LINEAR);

    }
    catch(Exception e){System.out.println(e.toString());}

    appearance.setTexture(earthTexture);

    // Use the texture in conjucntion with a sphere geometry
    Sphere sphere = new Sphere(0.5f,
        Sphere.GENERATE_TEXTURE_COORDS,
        appearance);
    root.addChild(sphere);

    root.compile();

    return root;
  }
}
```

This program begins by loading a texture image that represents the surface of the planet earth using a suitably constructed `TextureLoader` object. The magnification filter mode of the loaded texture is set to `BASE_LEVEL_LINEAR`. The `Texture` object is then associated with an `Appearance` object and the `Appearance` object is used to construct a `Sphere` primitive which is ultimately added to the root of the scene graph. Examples of renderings obtained with different version of this program are illustrated in Figure 2.40.



(a) (b)

Figure 2.40: A sphere texture mapped with an image of the earth where Australia is visual rendered using two magnification modes: `BASE_LEVEL_POINT` (a) and `BASE_LEVEL_LINEAR` (b)

It should be noted that there are a variety of other attributes associated with the `Texture` class. These other attributes are discussed in detail in the Java 3D API specification.

2.7.10.3 `TextureAttributes`

The `TextureAttribute` class defines a range of attributes that apply to the texture mapping process. One of the main attributes defined in the `TextureAttributes` class is the texture mode. The texture mode can have one of the following values:

- `MODULATE` - mixes the colour of the texture with the colour of the underlying surface. This texture mode make it possible to use lighting with a texture.
- `DECAL` - applies the texture to the shape being mapped in the form of a decal. This means that the transparency defined in the texture image is preserved.
- `BLEND` - blends the texture blend colour with the colour of the underlying surface. This is an advanced mode where the blending that occurs at each point depends on the values of the pixels in the texture image.
- `REPLACE` - applies the texture directly onto the object overriding the underlying colour of the shape. This is the default texture mode. Textures applied using this mode are not affected by light.

- COMBINE - combines the object colour with the texture color or texture blend colour according to the combine operation specified by the texture combine mode. Possible values for the combine mode include:
  - COMBINE_REPLACE
  - COMBINE_MODULATE
  - COMBINE_ADD
  - COMBINE_SUBTRACT
  - COMBINE_INTERPOLATE

It is possible to set and retrieve the texture mode of a `TextureAttributes` object using the following methods:

- `void setTextureMode(int textureMode)`
  Sets the texture mode of this `TextureAttributes` object to the specified value.
- `int getTextureMode()`
  Returns the texture mode of this `TextureAttributes` object.

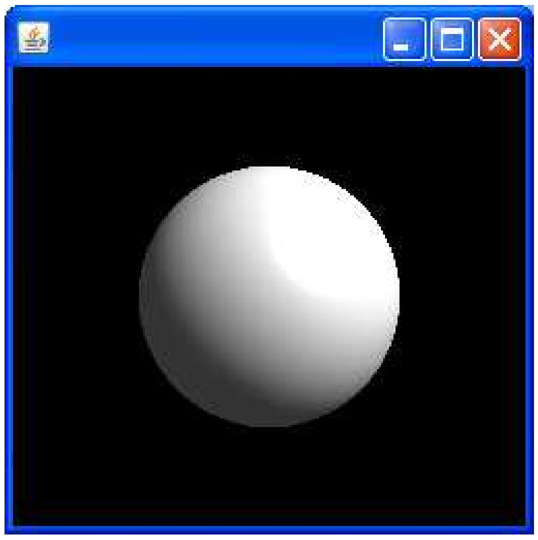
The following example demonstrates how the `TextureAttrubtes` appearance component can be used to make a texture appear to respond to lighting.

```

import javax.media.j3d.*;

import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.image.*;
import javax.vecmath.*;

public class TextureModeExample extends BasicSceneWithMouseControl
{
  public static void main(String args[]){new TextureModeExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    Appearance appearance = new Appearance();

    // Create a white material
    Material material = new Material();
    material.setAmbientColor(new Color3f(1.0f, 1.0f, 1.0f));
    material.setDiffuseColor(new Color3f(1.0f, 1.0f, 1.0f));
    appearance.setMaterial(material);

    Texture earthTexture = null;
    try
    {
      // Load the earth texture
      TextureLoader loader = new TextureLoader("earth.jpg", this);
      earthTexture = loader.getTexture();
    }
    catch(Exception e){System.out.println(e.toString());}
    appearance.setTexture(earthTexture);

    // Set the texture mode to modulate
    TextureAttributes textureAttributes = new TextureAttributes();
    textureAttributes.setTextureMode(TextureAttributes.MODULATE);
    appearance.setTextureAttributes(textureAttributes);

    // Create a sphere primtive with texture coordinates and normals
    Sphere sphere = new Sphere(0.5f,
        Sphere.GENERATE_TEXTURE_COORDS|Sphere.GENERATE_NORMALS,
        50, appearance);
    root.addChild(sphere);

    // Create a bright white directional light
    DirectionalLight light = new DirectionalLight(new Color3f(1.0f, 1.0f, 1.0f),
        new Vector3f(-1.0f, -1.0f, -1.0f));
    light.setInfluencingBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));
    root.addChild(light);

    // Create a dark white ambient light
    AmbientLight ambientLight = new AmbientLight(new Color3f(0.2f, 0.2f, 0.2f));
    ambientLight.setInfluencingBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));
    root.addChild(ambientLight);

    root.compile();

    return root;
  }
}
```

The program begins by creating an `Appearance` object. A `Material` object is then associated with the `Appearance` object. The ambient colour of the `Material` is white (1.0, 1.0, 1.0) and the diffuse colour of the `Material` is also white (1.0, 1.0, 1.0). The "earth" texture is loaded using a suitably constructed `TextureLoader` object. A `TextureAttributes` object is then created and its texture mode is set to `MODULATE`. The `TextureAttributes` object is also associated with the previously constructed `Appearance` object. The `Appearance` object is ultimately used in the construction of a `Sphere` primitive. Two light sources are also included in the scene: a directional light with a bright white colour (1.0, 1.0, 1.0) and an ambient light with a dim white colour (0.2, 0.2, 0.2). Renderings generated by variation of this program are illustrated in Figure 2.41.

It is possible to transform the texture coordinates for a piece of geometry using the a `TextureAttributes` object. This is achieved by specifying a suitable `Transform3D` object using the following method:

- `void setTextureTransform(Transform3D transform)`
  Transforms the texture coordinates using the transform represented by the


(a)

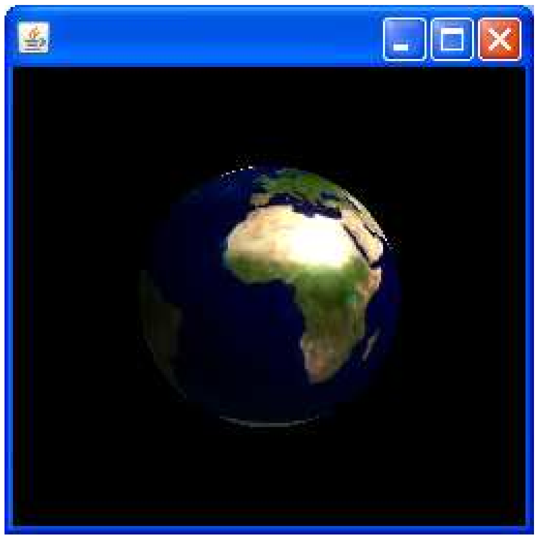

(b)

Figure 2.41: Making textures respond to lighting. A white `Sphere` primitive illuminated by ambient and directional light sources (a). A texture applied to the same `Sphere` using the `MODULATE` texture mode appears to respond to both light sources (b).

specified `Transfrom3D` object.

The `TextureAttributes` class also support other attributes and these are discussed in detail in the Java 3D API specification.

2.7.10.4 `TexCoordGeneration`

The `TexCoordGeneration` class defines all of the parameters required for automatic texture coordinate generation and it is included as a part of an `Appearance` object.

Texture coordinates determine which texel in the texture map is assigned to a given vertex. Texture coordinates are interpolated between vertices in a similar method to the way colors are interpolated between vertices.

Rather than the programmer having to explicitly assign texture coordinates to a particular piece of geometry, Java 3D can automatically generate the texture coordinates to achieve texture mapping. The `TexCoordGeneration` class defines attributes that specify the functions for automatically generating texture coordinates. The attributes defines by the `TexCoordGeneration` class include:

- **The texture format -** defines whether the generated texture coordinates are 2D, 3D, or 4D. In the case of 2D texture coordinates this attribute has a value of `TEXTURE_COORDINATE_2`.
- **Texture generation mode -** defines how the texture coordinates are generated. The possible values for this attribute are:
  - `OBJECT_LINEAR` - texture coordinates are generated as a linear function of the object coordinates, i.e. in the case of 2D texture coordinates the

(s, t) texture coordinates are obtained directly from the (x, y) vertex coordinates.

  - `EYE_LINEAR` - texture coordinates are generated as a linear function in eye coordinates. Note that this mode transforms the shapes coordinates to the viewers coordinate system before the texture coordinates are generated.
  - `SPHERE_MAP` - texture coordinates are generated using spherical reflection mapping in eye coordinates. This mode is used to simulate the reflected image of a spherical environment onto a polygon.
  - `NORMAL_MAP` - texture coordinates are generated to match vertices' normals in eye coordinates.
  - `REFLECTION_MAP` - texture coordinates are generated to match vertices' reflection vectors in eye coordinates.

- **Plane equation coefficients** - defines the coefficients for the plane equations used to generate the coordinates in the `OBJECT_LINEAR` and `OBJECT_EYE` texture coordinate generation modes. The coefficients define a reference plane in either object coordinates or in eye coordinates, depending on the texture generation mode.

The following example demonstrates how texture coordinates can be automatically generated for a simple triangular polygon.

```java
import javax.media.j3d.*;
import com.sun.j3d.utils.image.*;

public class TexCoordGenerationExample extends BasicSceneWithMouseControl
{
  public static void main(String args[]){new TexCoordGenerationExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    Texture woodTexture = null;

    try
    {
      // Load the wood texture
      TextureLoader loader = new TextureLoader("wood.jpg", this);
      woodTexture = loader.getTexture();
    }
    catch(Exception e){System.out.println(e.toString());}

    Appearance appearance = new Appearance();
    appearance.setTexture(woodTexture);

    // Defines 2D texture coordinates generated by a linear mapping
    TexCoordGeneration texCoordGeneration = new TexCoordGeneration();
    texCoordGeneration.setFormat(TexCoordGeneration.TEXTURE_COORDINATE_2);
    texCoordGeneration.setGenMode(TexCoordGeneration.OBJECT_LINEAR);
    appearance.setTexCoordGeneration(texCoordGeneration);

    float [] coordinates = {-0.5f, -0.5f,  0.0f,
        0.5f, -0.5f,  0.0f,
        0.0f,  0.5f,  0.0f};

    // Create a geometry array from the specified coordinates
    GeometryArray geometryArray = new TriangleArray(3,
    GeometryArray.COORDINATES);

    geometryArray.setCoordinates(0, coordinates);

    // Create a Shape3D object using the GeometryArray
    Shape3D shape = new Shape3D(geometryArray, appearance);
    root.addChild(shape);

    root.compile();

    return root;
  }
}
```

This program begins by loading the "wood" texture using a suitably constructed `TextureLoader` object. An `Appearance` object is then created and a `TexCoordGeneration` object is associated with the `Appearance` object. The format of the `TexCoordGeneration` object is set to `TEXTURE_COORD_2` and the texture coordinate generation mode is set to `OBJECT_LINEAR`. The vertices are specified for a `TriangleArray` geometry and a `Shape3D` object is created using the appearance and the geometry. Finally, the `Shape3D` object is added to the scene graph to be displayed. The output obtained when this program is executed is illustrated in Figure 2.42.

2.7.10.5 Using Multiple Textures

The texture options that have been discussed so far have dealt with applying a single texture to a surface. It is also possible to apply several layers of textures to a surface using a process referred to as multitexturing. This is an advanced approach to texturing that can be used to implement shadows and special kinds of lighting.

Multilayered textures are created by specifying a series of `TextureUnitState` objects for each layer of texture mapping. A `TextureUnitState` object holds the `Texture`, `TextureAttributes` and `TextCoordGeneration` objects that represent a specific texture. A `TextureUnitState` object can be associated with an `Appearance` object using one of the following methods:

- `void setTextureUnitState(int index, TextureUnitState state)`
  Set the `TextureUnitState` object for this `Appearance` object at the specified index to the specified value.

- `void setTextureUnitState(TextureUnitState[] stateArray)`
  Set the array of `TextureUnitState` objects for this `Appearance` object to the specified array.

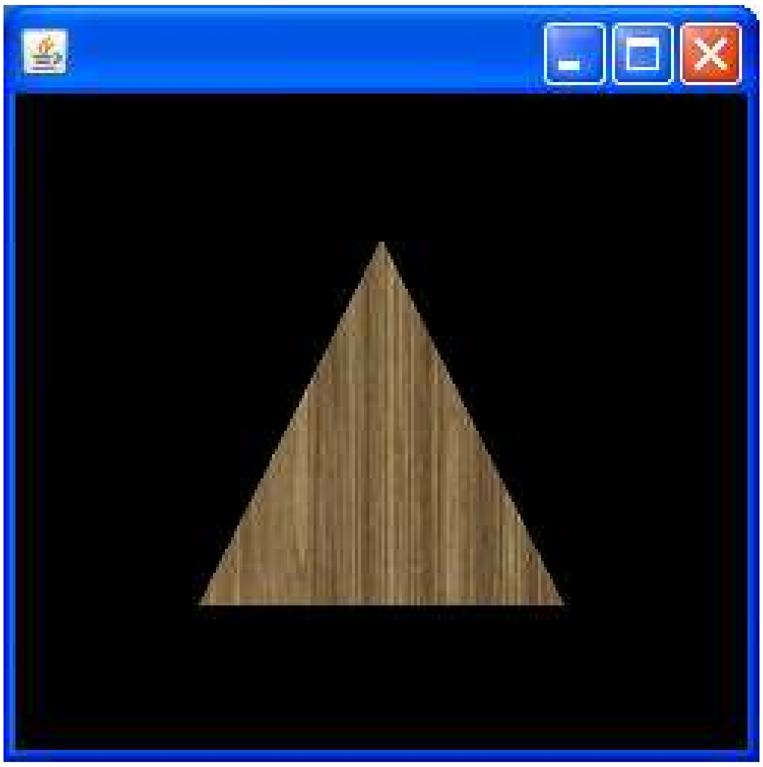


Figure 2.42: A simple triangular geometry where the texture coordinates have been automatically generated using a `TexCoordGeneration` appearance component.

It is also possible to specify several sets of texture coordinates with an instance of `GeometryArray`. The mapping of the texture coordinates to a specific `TextureUnitState` object is defined in the constructor for the relevant subclass of `GeometryArray`, for example `TriangleArray`:

- `TriangleArray(int vertexCount, int vertexFormat, int texCoordSetCount, int[] texCoordSetMap)`
  Constructs a `TriangleArray` object that support multiple sets of texture coordinates. The number of sets is specified by the `texCoordSetCount` argument and the mapping between the individual sets of texture coordinates and the associated `TextureUnitState` object is indicated in the `texCoordSetMap` argument.

The texture coordinate associated with a particular `TextureUnitState` object can be specified using the following method:

- `void setTextureCoordinates(int texCoordSet, int index, TexCoord2f[] texCoords)`
  Sets the texture coordinates associated with the vertices starting at the specified index in the specified texture coordinate set for this `GeometryArray` object.

2.7.11 Environment Nodes

Java 3D provides a range of environment nodes that affect the affect the environment of a virtual world. Environment nodes can be used to control lighting, sound and the background of a scene.

2.7.11.1 Bounding Regions

Environment nodes typically affect the a particular region of a virtual world. For example, lights shine on shapes and sounds create audible content. Light and sound sources should not affect the entire virtual universe. Consequently, Java 3D requires bounding regions to be specified for environment nodes to define the region where they are active. There are two main types of bounding region and they are constructed as follows:

- `BoundingBox(Point3d lower, Point3d upper)`
  Creates a new `BoundingBox` object within the specified bounds.

- `BoundingSphere(Point3d centre, double radius)`
  Creates a new `BoundingSphere` object at the specified location with the specified radius.

If a bounding region is not specified for a particular environment node then it is considered to be inactive.

If a environment node is required to be active throughout the entire virtual universe then it is possible to approximate infinite bounds by creating a `BoundingSphere` with a radius of `Double.MAX_VALUE`.

2.7.11.2 Background

The `Background` environment node defines a solid background colour and a background image that are used to fill the window at the beginning of each new frame. The `Background` environment node also allows background geometry to be specified.

A `Background` environment node that represents a colour can be created using the following constructor:

- `Background(Color3f backgroundColour)`
  Creates a new `Background` environment node with the specified colour.

Alternatively, a background environment node that represents a image can be created:

- `Background(ImageComponent2D backgroundImage)`
  Creates a new `Background` environment node using the specified image.

An `ImageComponent2D` object representing an image resource can be obtained by calling the `getImage()` method of a suitably constructed `TextureLoader` object.

The following example demonstrates how a background image can be created and added to a 3D scene.

```

import javax.media.j3d.*;

import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.image.*;
import javax.vecmath.*;

public class BackgroundImageExample extends
   BasicSceneWithMouseControlAndLights
{
  public static void main(String args[]){new BackgroundImageExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    ImageComponent2D starImage = null;
    try
    {
      // Load the stars image
      TextureLoader loader = new TextureLoader("stars.gif", this);
      starImage = loader.getImage();
    }
    catch(Exception e){System.out.println(e.toString());}

    // Create a background node from the stars image
    Background bg = new Background(starImage);
    bg.setApplicationBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));
    root.addChild(bg);

    Appearance appearance = new Appearance();

    Material material = new Material();
    appearance.setMaterial(material);

    Texture earthTexture = null;
    try
    {
      // Load the earth texture
      TextureLoader loader = new TextureLoader("earth.jpg", this);
      earthTexture = loader.getTexture();
    }
    catch(Exception e){System.out.println(e.toString());}
    appearance.setTexture(earthTexture);

    // Allow the texture to respond to lighting
    TextureAttributes textureAttributes = new TextureAttributes();
    textureAttributes.setTextureMode(TextureAttributes.MODULATE);
    appearance.setTextureAttributes(textureAttributes);

    // Create a details sphere and map the earth texture
    Sphere sphere = new Sphere(0.5f,
        Sphere.GENERATE_TEXTURE_COORDS|Sphere.GENERATE_NORMALS,
        100, appearance);
    root.addChild(sphere);

    root.compile();
    return root;
  }
}
```

This program is an modified version of the `TextureModeExample.java` example that was discussed earlier. The main difference here is that a `Background` environment node is added to the scene. This is achieved by loading the required background image using a suitably constructed `TextureLoader` object and then calling the `getImage()` method to obtain a `ImageComponent2D` object that represents the loaded image. This image is subsequently used to construct a `Background` object. The application bounds for the `Background` environment node are set to the maximum value and the `Background` is added to the root of the scene graph. The rendering obtained when this program is executed is illustrated in Figure 2.43

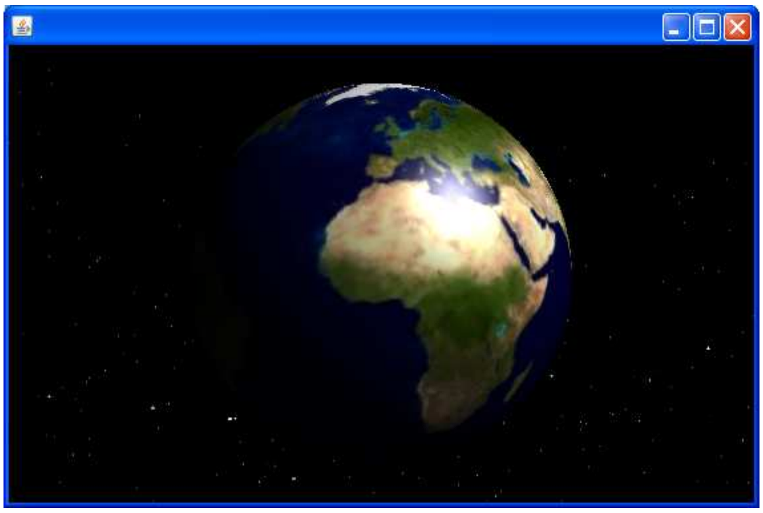


Figure 2.43: A texture mapped sphere representing the planet earth and a background image representing a backdrop of stars.

It is also possible to use a geometry rather than a flat image the create a background. This involves rendering a texture mapped `Sphere` at an infinite distance. The normals of the sphere must be flipped inwards so that the texture is applied to the interior of the `Sphere`.

The following example demonstrates how a background geometry can be specified for a scene.

```

import javax.media.j3d.*;

import com.sun.j3d.utils.behaviors.mouse.MouseRotate;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.image.*;
import javax.vecmath.*;

public class BackgroundGeometryExample extends
```

```
  BasicSceneWithMouseControlAndLights
{
  public static void main(String args[]){new BackgroundGeometryExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    Texture starTexture = null;
    try
    {
      // Load the stars texture
      TextureLoader loader = new TextureLoader("stars.gif", this);
      starTexture = loader.getTexture();
    }
    catch(Exception e){System.out.println(e.toString());}

    // Create a backgound node with maximum bounds
    Background background = new Background();
    background.setApplicationBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));

    BranchGroup backgroundGroup = new BranchGroup();

    // Set the texture for the background appearance
    Appearance backgroundAppearance = new Appearance();
    backgroundAppearance.setTexture(starTexture);

    // Create a detailed sphere with normal pointing inwards
    Sphere backgroundSphere = new Sphere(0.5f,
        Sphere.GENERATE_TEXTURE_COORDS|
        Sphere.GENERATE_NORMALS_INWARD,
        100, backgroundAppearance);
    backgroundGroup.addChild(backgroundSphere);
    background.setGeometry(backgroundGroup);
    root.addChild(background);

    // Create the earth sphere
    Appearance earthAppearance = new Appearance();

    Material earthMaterial = new Material();
    earthAppearance.setMaterial(earthMaterial);

    Texture earthTexture = null;
    try
    {
      TextureLoader loader = new TextureLoader("earth.jpg", this);
      earthTexture = loader.getTexture();
    }
    catch(Exception e){System.out.println(e.toString());}
    earthAppearance.setTexture(earthTexture);

    Sphere earthSphere = new Sphere(0.5f,
        Sphere.GENERATE_TEXTURE_COORDS|
        Sphere.GENERATE_NORMALS,
        100, earthAppearance);
    root.addChild(earthSphere);

    root.compile();

    return root;
  }
}
```

The main difference between this example and the previous background example is that a background geometry is used rather than at flat background. The background geometry is represented by a `BranchGroup` with a single `Sphere` child. The `Sphere` generates normals that are projected inwards so that the texture is mapped to the inside of the sphere rather than the outside. The `BranchGroup` representing the background geometry is associated with a `Background` object using the `setGeometry()` method and the `Background` object is ultimately added to the root of the scene graph using the `addChild()` method. The output obtained when this program is executed is illustrated in Figure 2.44.

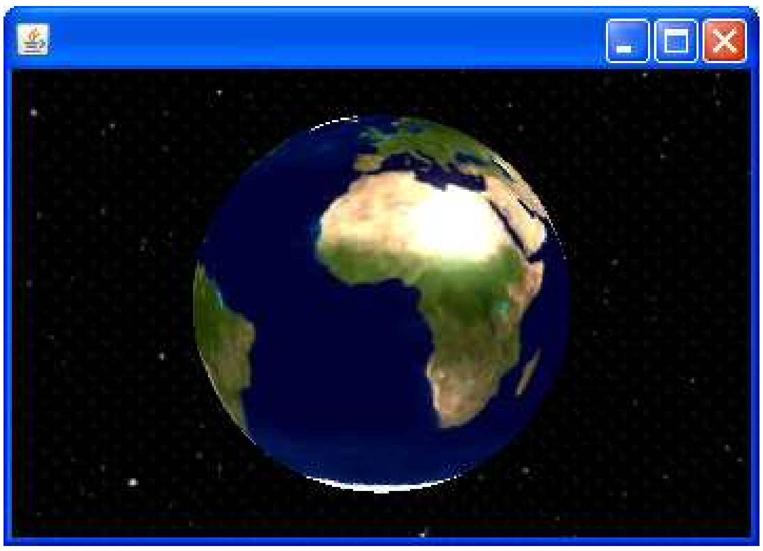


Figure 2.44: A scene where the background is represented by a spherical geometry rendered at infinity. The normals of the sphere point inwards so that the background texture is mapped to the interior of the sphere rather than the exterior.

2.7.11.3 `Fog`

The `Fog` environment node simulates the way that object appear to fade into the background when viewed from a distance. Fog is a useful feature and can be used to add a great deal of realism to a scene. The effect caused by fog is sometimes called “depth cueing” because it gives the brain a visual cue as to the depth of an object in the scene. Fog is implemented in Java 3D by blending the fog colour with the colour of the scene objects based on their distance from the viewer. Java 3D provides support for two types of `Fog` environment nodes:

- `LinearFog` - has a constant density, so the level of obscurity generated by the fog increases linearly as the viewer moves away from the object being viewed.
- `ExponentialFog` - the fog density increases exponentially so that the level of obscurity increases exponentially as the view moves away from the object being viewed.

`LinearFog` has three main attributes:

- **Colour -** defines the colour of the fog.
- **Front distance -** anything closer to the viewer than the front distance is not affected by the fog.
- **Back distance -** anything further away from the view than the back distance is completely obscured by the fog.

A `LinearFog` object can be created using the following constructor:

- `LinearFog(Color3f colour, double frontDistance, double backDistance)` Creates a `LinearFog` object with the specified colour, front distance and back distance.

It should be noted that the influencing bounds for a `Fog` node must be set, otherwise the `Fog` node will be considered to be disabled.

The following example demonstrates how a `Fog` node can be used to obscure an object in a scene.

```java
import javax.media.j3d.*;
import javax.vecmath.*;

import com.sun.j3d.utils.behaviors.mouse.*;
import com.sun.j3d.utils.geometry.*;

public class LinearFogExample extends BasicScene{

  public static void main(String args[]){new LinearFogExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    // Define a backgound with a constant mid grey colour
    Background background = new Background(new Color3f(0.5f, 0.5f, 0.5f));
    background.setApplicationBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));
    root.addChild(background);

    // Define a linear fog node with the same colour as the background
    LinearFog linearFog = new LinearFog(new Color3f(0.5f, 0.5f, 0.5f), 1.0, 10.0);
    linearFog.setInfluencingBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));
    root.addChild(linearFog);

    TransformGroup tg = new TransformGroup();
    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    root.addChild(tg);

    // Include support for mouse zoom
    MouseZoom zoom = new MouseZoom();
    zoom.setTransformGroup(tg);
    tg.addChild(zoom);
    zoom.setSchedulingBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));

    // Include support for mouse rotate
    MouseRotate rotate = new MouseRotate();
    rotate.setTransformGroup(tg);
    tg.addChild(rotate);
    rotate.setSchedulingBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));

    ColorCube colorCube = new ColorCube(0.4f);
    tg.addChild(colorCube);

    return root;
  }
}
```

This example begins by creating a background with a constant colour of mid grey (0.5, 0.5, 0.5). A `LinearFog` environment node of the same colour is then created and added to the root of the scene. The front distance for the `LinearFog` node is 1 metre and the back distance is 10 metres. This means that objects less than one metre away from the viewer are not affected by the fog and objects greater than 10 metres away from the viewer are completely obscured by the fog. `MouseZoom` and `MouseRotate` behaviours are added to the scene in order to demonstrate the effects of the fog. Examples of the types of renderings that are obtained when this program is executed are illustrated in Figure 2.45.

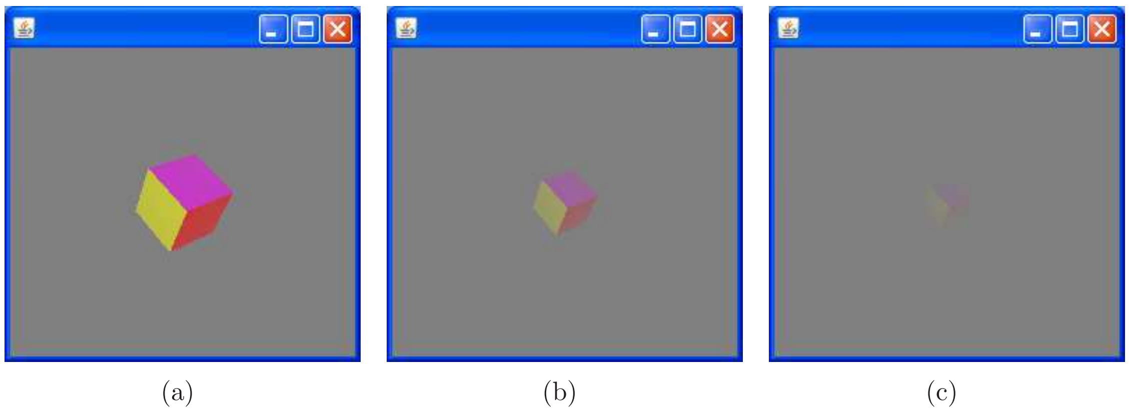


Figure 2.45: Examples of a `ColorCube` viewed from different distances in the presence of `LinearFog`.