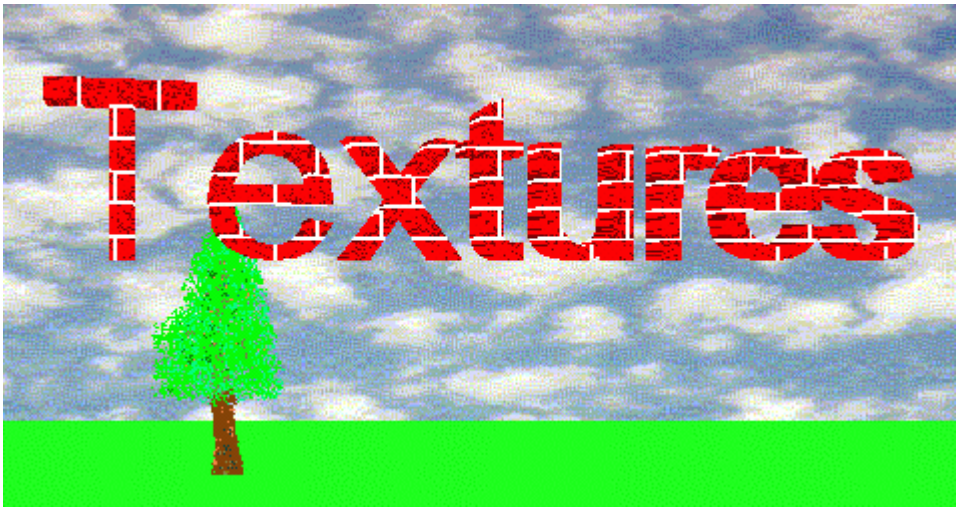
Getting Started with the Java 3D™ API

Chapter 7
Textures



Dennis J Bouvier

tutorial v1.6 (Java 3D API v1.2)

This documentation was prepared for Sun Microsystems by Dennis J Bouvier (`djb@raven-red.com`). For further information about course development or course delivery, please contact either Sun Microsystems or Dennis.

Table of Contents
Chapter 7:

List of Figures

List of Tables

List of Code Fragments

List of Reference Blocks

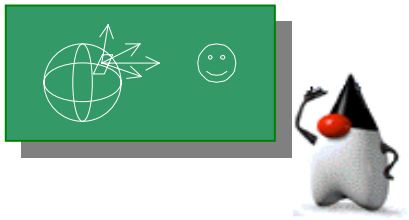
Preface to Chapter 7

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/java-media/3D/collateral`

New for Java 3D API version 1.2

This chapter of the tutorial has been updated to include new features in Java 3D API release version 1.2. You may notice the tag **`<new in 1.2>`** to the right of some section headings and in some reference blocks in this chapter. This tag indicates the that tutorial topic or API feature is new in the API release version 1.2. Note that since chapters are updated and released individually not all of the tutorial chapters may reflect the latest version of the Java 3D API.

CHAPTER 7
Textures



Chapter Objectives

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

- Add visual richness to simple geometry with textures
- Load textures easily with the TextureLoader utility
- Customize the use of textures with TextureAttributes objects
- Automatically generate texture coordinates to simplify texturing
- Apply multiple textures to a single visual object

The appearance of many real world objects depends on its texture. The texture of an object is really the relatively fine geometry at the surface of an object. To appreciate the role surface textures play in the appearance of real world objects, consider carpet. Even when all of the fibers of a carpet are the same color the carpet does not appear as a constant color due to the interaction of the light with geometry of the fibers. Even though Java 3D is capable of modeling the geometry of the individual carpet fibers, the memory requirements and rendering performance for a room size piece of carpet modeled to such detail would make such a model useless. On the other hand, having a flat polygon of a single color does not make a convincing replacement for the carpet in the rendered scene.

Up to this point in the tutorial, the detail of visual objects has been provided by the geometry. As a result, visually rich objects, such as trees, can require a great deal of geometry which in turn requires the appropriate memory and rendering computation. At some level of detail, the performance may become unacceptable. This chapter shows how to add the appearance of surface detail to a visual object without adding more geometry through the use of textures.

7.1 What is Texturing

Carpet may be the extreme example in terms of the complexity and density of the surface geometry, but it is far from the only object for which we perceive texture. Bricks, concrete, wood, lawns, walls, and paper are just some of the objects for which flat geometry (e.g., polygons) can be used to represent the general shape, but not the fine detail. Just as with carpet, the cost of representing surface texture in geometric primitives for these objects is quite high.

A possible alternative to modeling the fiber of the carpet is to model the carpet as a flat polygon with many vertices, assigning colors to the vertices to give variations in color. If the vertices are sufficiently close, then the image of the carpet can be produced. This requires significantly less memory than the model that includes the fibers of the carpet; however, the model would still require too much memory for a reasonable size room. This idea, that of representing the image of the object on a flat surface, is the basic idea of texturing. However, with texturing, the geometry can be very simple.

Texturing, also called *texture mapping*, is a way to add the visual richness of a surface without adding the fine geometric details. The visual richness is provided by an image, also called a *texture*[1], which gives the appearance of surface detail for the visual object. The image is mapped on to the geometry of the visual object at rendering time. Thus the term *texture mapping*.

Figure 7-1 shows some of the textures used in example programs for this chapter. As you can see from this figure, a texture can provide the visual richness of objects of various sizes. The striped texture with words is used in a variety of examples to demonstrate the flexibility of texturing objects, the words are there to easily distinguish the edges of the texture image.

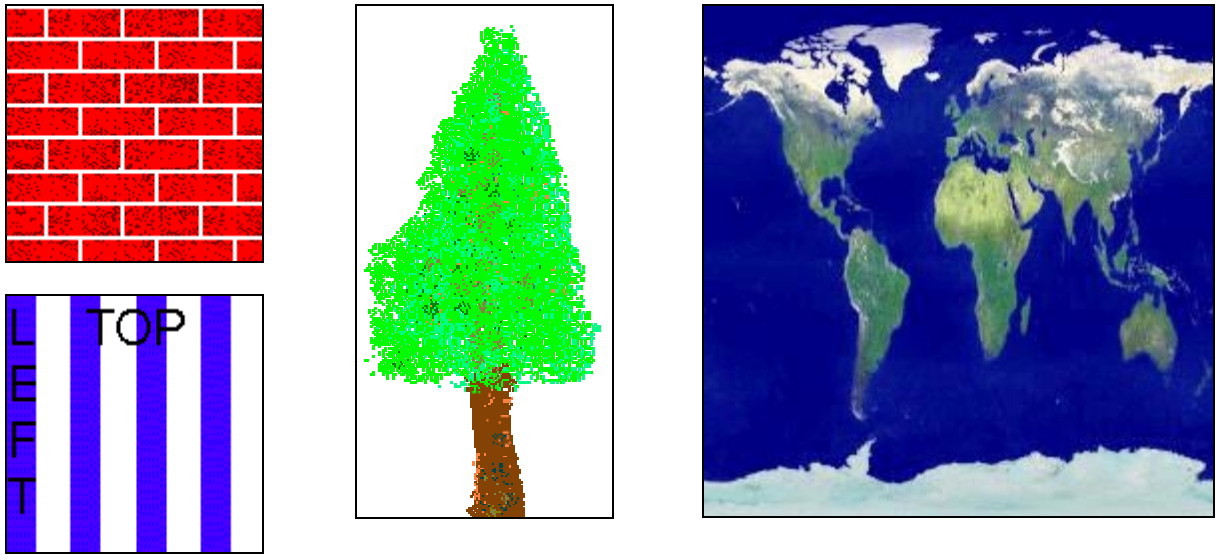


Figure 7-1 Some of the Images (outlined) used as Textures in Example Programs.

7.2 Basic Texturing

Texturing of polygons in a Java 3D program is achieved though creating the appropriate appearance bundle and loading the texture image into it, specifying the location of the texture image on the geometry, and setting texturing attributes. As you will see, specifying textures can be complex. Fortunately, there are utility classes to help with the process and the default settings for texturing options are appropriate for basic texturing applications.

To explain texturing, Section 7.2.1 presents a simple recipe; then Section 7.2.2 develops an example program based on the recipe, further explaining texturing. The remaining subsections present additional example programs to further explain details of texture specification. The texturing options not discussed in the context of an example will be discussed with the API details in Section 7.7.

[1] Even though texture images are referred to as 'textures' and they visually represent geometric structures, they are neither representations of geometry nor do they alter the geometry of a visual object in any way.

7.2.1 Simple Texturing Recipe

Due to the flexibility of texturing in the Java 3D API, the number of texturing related options can be a bit overwhelming at first. Even so, texturing need not be difficult. To make easy work of texture specifications, follow the simple recipe of Figure 7-2.

The recipe only outlines the steps directly related to texturing. The reader should realize that the geometry and appearance are set in a Shape3D object which is added to the scene graph. Previous chapters of this tutorial, Chapters 1 and 2 in particular, cover the implied steps of the recipe

1. Prepare texture images
2a. Load the texture
2b. Set the texture in Appearance bundle
3. Specify TextureCoordinates of Geometry

Figure 7-2 Simple Texturing Recipe

As with several of the recipes in the tutorial, some steps of the recipe may be performed out of the order they are presented. In fact, the steps of this recipe may be performed in any order (provided steps 2a and 2b are done together).

Texturing Step 1: Prepare the texture image

This recipe begins with a non-programming step: "prepare texture images". Creating and editing texture images is something that is normally done external to Java 3D programs. In fact, most texture images are prepared before the program is begun. There are two essential tasks in texture image preparation: 1. ensuring the images are of acceptable dimensions, and 2. ensuring the images are saved in a file format which can be read. Of course the image could be edited to achieve the desired color, transparency, and tiling characteristics.

For rendering efficiency, Java 3D requires the size of the texture image to be a mathematical power of two (1, 2, 4, 8, 16, …) in each dimension. Failing to meet this restriction will result in a runtime exception.

If an image is not of acceptable dimensions, it must be modified (scaled or cropped) to meet the dimension requirements before it is used. Image editing can be done in a wide variety of programs including the Java Advanced Imaging API[2]. In Figure 7-1, the two smaller images are 128 by 128, the tree is 256 by 128, and the earth is 256 by 256.

As far as the file formats are concerned, a texture (image) may be stored in any image file format provided there is a method to load it. The programs of this chapter load textures using the TextureLoader utility class (there is more information on texture loaders in the next step, API details are in Section 7.7). A TextureLoader object loads JPEG, GIF, and other file formats.

One more word about the example programs before moving to the next step. The code fragments and example programs of this chapter use file names for some image files that are included in the example programs jar. There is nothing special about these image files other than that they comply with the power of two dimension restriction. Any image file can be used in the programs provided the images have dimensions that are a power of two. Feel free to compile and run the example programs with your own image files. Now, with texture images ready, the programming can begin.

[2] The Java Advanced Imaging API (`http://java.sun.com/products/java-media/jai`) enables Java programmers to easily create and edit 2D imagery.

Texturing Step 2a: Load the Texture

The next step is to get the prepared image into an image object. This is known as loading the texture. Textures can be loaded from files or URLs using the same basic process[3]. Loading a texture can be accomplished with many lines of code, or with two lines of code that use a TextureLoader utility object. Either way, the result is to get the image into a ImageComponent2D object. Code Fragment 7-1 shows an example of two lines that use a TextureLoader. The result of these two lines is to load the image of the file `stripe.gif` into a Image2DComponent object which can be used to create the necessary appearance bundle of step 3.

```
1. TextureLoader loader = new TextureLoader("stripe.gif", this);
2. ImageComponent2D image = loader.getImage();
```

Code Fragment 7-1 Using a TextureLoader Object to Load the STRIPE.GIF Image File.

Before moving on to step 3 of the recipe, let's take a closer look at the use of the TextureLoader object. The second argument of the constructor specifies an object which serves an the *image observer*. The TextureLoader class uses the `java.awt.image` package for loading the images. This package loads images asynchronously, which is particularly useful when an image is loaded from a URL. To facilitate managing asynchronous loads of images, AWT components are capable of being image observers, which is to observe the image load process. An image observer can be queried for the details of the image load.

For the purpose of writing Java 3D programs all that you need to know is that any AWT component can serve as an image observer. Since Applet is an extension of the AWT component Panel, the Applet object of a Java 3D program can be the image observer for the TextureLoader object. Further details on image observers and other AWT topics is beyond the scope of this tutorial. Refer to an AWT reference for more information.

Texturing Step 2b: Create the Appearance Bundle

To be used as a texture for a visual object, the texture image loaded in Step 2a must be set as the texture in a Texture object, which is then used in an appearance bundle referenced by the visual object. Specifically, a Texture2D[4] object holds the texture image. The ImageComponent2D image loaded in the Step 2a is central to the appearance bundle creation of step 2b [5].

Code Fragment 7-2 shows the two lines of code from Step 2a followed by the code to form a simple texturing appearance bundle. Having loaded the texture (lines 1 and 2), the image is then set in the Texture2D object (line 4). The Texture2D object is then added to the Appearance object (line 6).

```
1.    TextureLoader loader = new TextureLoader("stripe.jpg", this);
2.    ImageComponent2D image = loader.getImage();
3.    Texture2D texture = new Texture2D();
4.    texture.setImage(0, image);
5.    Appearance appear = new Appearance();
6.    appear.setTexture(texture);
```

Code Fragment 7-2 Creating an Appearance with a Texture.

[3] Other sources for Textures include Text2D objects and procedurally created images.

[4] There is a Texture3D object discussed in Section 7.7.4. The Texture3D object is used when a volume of color, or a stack of images, is the texture.

[5] The discussion of the image observer is a major reason why loading a texture (step 2a) is identified as a step separate from the creation of the appearance bundle (step 2b).

The appearance bundle created in Code Fragment 7-2 could have other node components, most notably of the possibilities is the TextureAttributes node component. For the simple example, no TextureAttributes object is used. Section 7.4 discusses TextureAttributes.

Texturing Step 3: Specify TextureCoordinates

In addition to loading the texture into an appearance bundle, the programmer also specifies the placement of the texture on the geometry through the specification of the texture coordinates. Texture coordinate specifications are made per geometry vertex. Each texture coordinate specifies a point of the texture to be applied to the vertex. With the specification of some points of the image to be applied to vertices of the geometry, the image will then be rotated, stretched, squashed, and/or duplicated to make it fit the specification[6].

TextureCoordinates are specified in the `s` (horizontal) and `t` (vertical) dimensions of the texture image space in the range of 0.0 to 1.0, as shown in Figure 7-3.

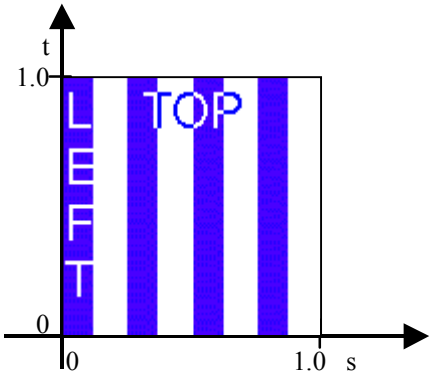


Figure 7-3 Texture Mapping Coordinates

The reference block below shows just some of the many GeometryArray methods available for setting texture coordinates. This reference block also lists a deprecated setTextureCoordinate() method, more is said about this on page 7-7 in the section called "A word about API changes". Refer to the Java 3D API Specification for additional setTextureCoordinate methods. Additional information on the GeometryArray class is available in Chapter 2 of the tutorial.

[6] Specifically, the texture coordinates are linearly interpolated from the vertices to map the texture to the geometry. See Section 7.2.3 for more details.

GeometryArray setTextureCoordinate Methods (partial list)

Texture coordinates are specified per vertex in the geometry via one of several setTextureCoordinate methods which are methods of the GeometryArray class. Beginning with Java 3D API version 1.2 the methods for managing texture coordinates changed. More is said about this on page 7-7 in "A word about API changes".

This reference block lists only two of the new methods of which there are nine. See The Java 3D API Specification for additional information on setting texture coordinates.

void setTextureCoordinate(int texCoordSet, int index, TexCoord* texCoord) **<new in 1.2>**

Sets the texture coordinate associated with the vertex at the specified index in the specified texture coordinate set for this object. There are numerous similar methods.

texCoordSet - texture coordinate set in this geometry array
index - destination vertex index in this geometry array
texCoord – any one of the TexCoord* class objects (such as TexCoord2f or TexCoord3f) containing the new texture coordinate

void setTexCoordRefFloat(int texCoordSet, float[] texCoords) **<new in 1.2>**

Sets the float texture coordinate array reference for the specified texture coordinate set to the specified array.

texCoordSet - texture coordinate set in this geometry array
texCoords - an array of 2*n or 3*n values to which a reference will be set.

void setTextureCoordinate(int index, Point2f texCoord) **<deprecated>**

Sets the texture coordinate associated with the vertex at the specified index for this object.

Code Fragment 7-3 creates a single plane using a QuadArray geometry object. Texture coordinates are assigned to each vertex. In Code Fragment 7-3, lines three through eleven establish the four corners of a quad in 3-space. Lines 13 through 21 establish the texture's location on the geometry. This particular code fragment creates a plane of 2 meters on a side and places the texture image in the normal (upright, not reversed) orientation across the face of the plane.

```
1.  QuadArray plane = new QuadArray(4, GeometryArray.COORDINATES
2.                                   | GeometryArray.TEXTURE_COORDINATE_2);
3.  Point3f p = new Point3f();
4.  p.set(-1.0f,  1.0f,  0.0f);
5.  plane.setCoordinate(0, p);
6.  p.set(-1.0f, -1.0f,  0.0f);
7.  plane.setCoordinate(1, p);
8.  p.set( 1.0f, -1.0f,  0.0f);
9.  plane.setCoordinate(2, p);
10. p.set( 1.0f,  1.0f,  0.0f);
11. plane.setCoordinate(3, p);
12.
13. TexCoord2f q = new TexCoord2f();
14. q.set(0.0f, 1.0f);
15. plane.setTextureCoordinate(0, 0, q);
16. q.set(0.0f, 0.0f);
17. plane.setTextureCoordinate(0, 1, q);
18. q.set(1.0f, 0.0f);
19. plane.setTextureCoordinate(0, 2, q);
20. q.set(1.0f, 1.0f);
21. plane.setTextureCoordinate(0, 3, q);
```

Code Fragment 7-3 Applying Texture Coordinates to a Quad.

Figure 7-4 shows the relationship between the vertex coordinates and the texture coordinates for the example quad created in Code Fragment 7-3. The left image of Figure 7-5 shows the application of the `stripe.gif` texture to the example geometry.

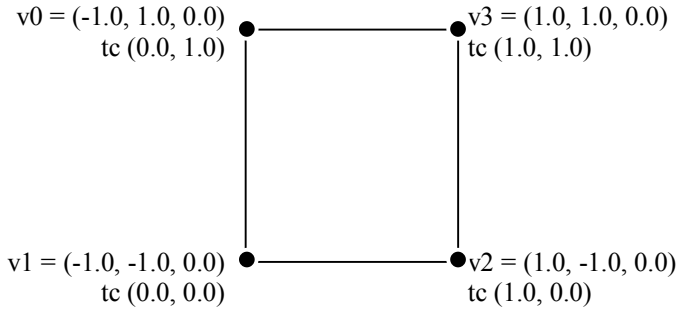


Figure 7-4 The Orientation of Texture Coordinates in Texture Image Space.

Having now completed the three texturing steps, the textured object can be added to a scene graph. The following section presents a series of example programs demonstrating some options in texturing.

A word about API changes …. `<new in 1.2>`

… and about Multitexture

The previous reference block lists has the only deprecated method listed in the tutorial. You should be aware that deprecated API features (e.g., constructors, methods, classes, …) should be avoided. However, the `setTextureCoordinate(int index, Point2f texCoord)` method of GeometryArray is listed here for two reasons.

The first reason is to point out one of the most heavily revised classes in v1.2 of the Java 3D API. All of the setTextureCoordinate() methods of class GeometryArray have changed. In previous versions of the API, the setTextureCoordinate() methods used Point* classes instead of TexCoord* classes. The switch to TexCoord classes is made in v1.2 and all setTextureCoordinate() methods now use TexCoord* parameters, as was intended.

The second reason for showing the deprecated method next to the new methods is to introduce multitexturing by way of explaining the new parameter in the setTextureCoordinate methods. You may have noticed the additional parameter in the new setTextureCoordinate() methods. The first parameter of the setTextureCoordinate() method selects the texture coordinate set to be used. Which only makes sense if a GeometryArray object can have multiple sets of texture coordinates. This is exactly the case; a GeometryArray object can have multiple sets of texture coordinates.

Java 3D API v1.1 introduces multitextures. **Multitexture** is a feature allowing one visual object to have multiple textures applied. If multiple textures are applied to a single visual object then the visual object may need multiple texture coordinate sets so that each texture may have its own coordinate set. Hence, the need for multiple texture coordinate sets.

Many applications may benefit from multitexture. For example, a texture may give a brick appearance to an object. However, the brick objects may appear fake due to the regular nature of the same texture image applied repeatedly to a large object. A second texture, one of a weathering or dirty pattern may be applied over the brick to give a more natural appearance to the visual object. Another application involving multitexture is to texture a shadow over an already textured visual object.

Since there are many basic texture details left to explain, the topic of multitexture is put off until Section 7.8. The reason for introducing it now is to explain purpose of the `texCoordSet` parameter of the setTextureCoordinate() methods. But, also, knowing multiple textures can be applied to one visual object may inspire you to think more, and experiment more with textures.

7.2.2 Simple Texture Example Programs

Following the recipe of Figure 7-2, a simple texturing example program has been developed. A complete example is found in `SimpleTextureApp.java` of the `examples/texture` directory in the examples jar available with this tutorial. Figure 7-5 shows the scene rendered by SimpleTextureApp on the left. This program appears as little more than an image display program.

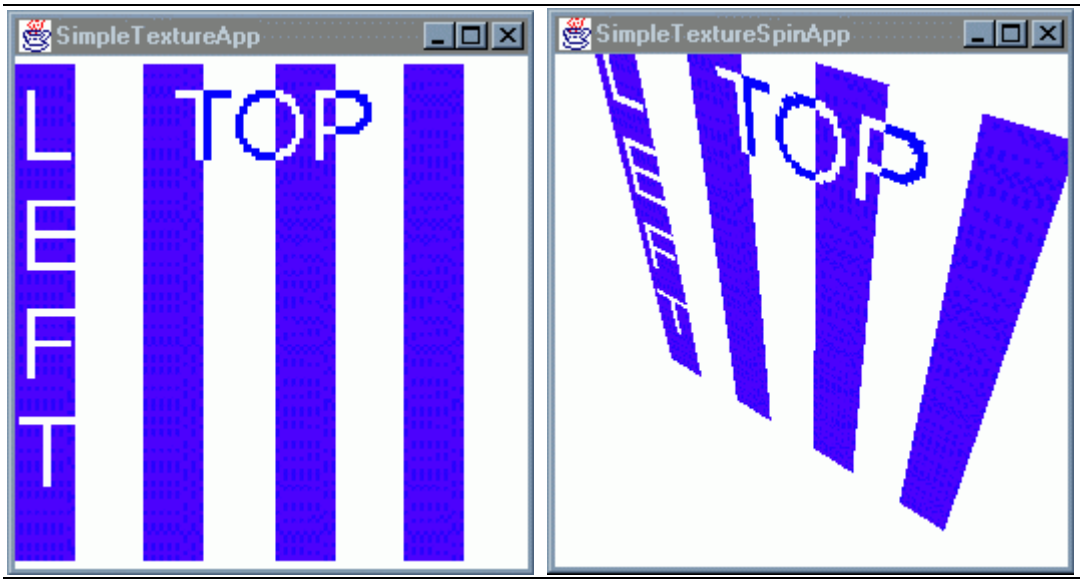


Figure 7-5 Scenes Rendered by the `SimpleTextureApp` (left) and `SimpleTextureSpinApp` (right) programs.

Using a RotationInterpolator object[7] another example program, `SimpleTextureSpinApp`, was created. In this program the same textured plane spins to demonstrate the 3D nature of the program. Figure 7-5 shows a rendering from this program on the right. One thing to notice when viewing the program, the back side of the plane is blank.

The NewTextureLoader Class

Java 3D programs using textures can have a large number of lines just for loading textures and creating the appearance bundles. Some programming and, more importantly, runtime memory can be conserved by sharing appearance bundles when appropriate. However, this does not reduce the amount of programming a great deal. Further reductions in programming can be achieved by creating a class to create the texture appearance bundles. The challenge in creating such a class lies in the image observer requirement for the TextureLoader object.

The Canvas3D object or an Applet can server as the image observer, but having a reference to some component everywhere in the program can be bothersome. To address this inconvenience, I have

[7] The interpolator code used is similar to that of the HelloJava3Dd example from Chapter 1. Interpolators are explained in Chapter 5 of the tutorial.

extended the TextureLoader class eliminating the need for an image observer component. Instead a single method is used to specify an image observer for all future uses of the texture loader.

The constructors for NewTextureLoader are the same as those for TextureLoader except none require a image observer component. The methods for NewTextureLoader are those of TextureLoader with the additional method for setting an image observer. See Section 0 for API information for both classes.

Another example program, `TexturedPlaneApp`, loads three textures and displays them on planes as shown in Figure 7-6. The significance of this program is that the textures are loaded using the TexturedPlane class defined external to the rest of the program which is more easily done with the NewTextureLoader class. This TexturedPlane class is not flexible enough to be used in very many applications, but serves as a demonstration for similar classes.

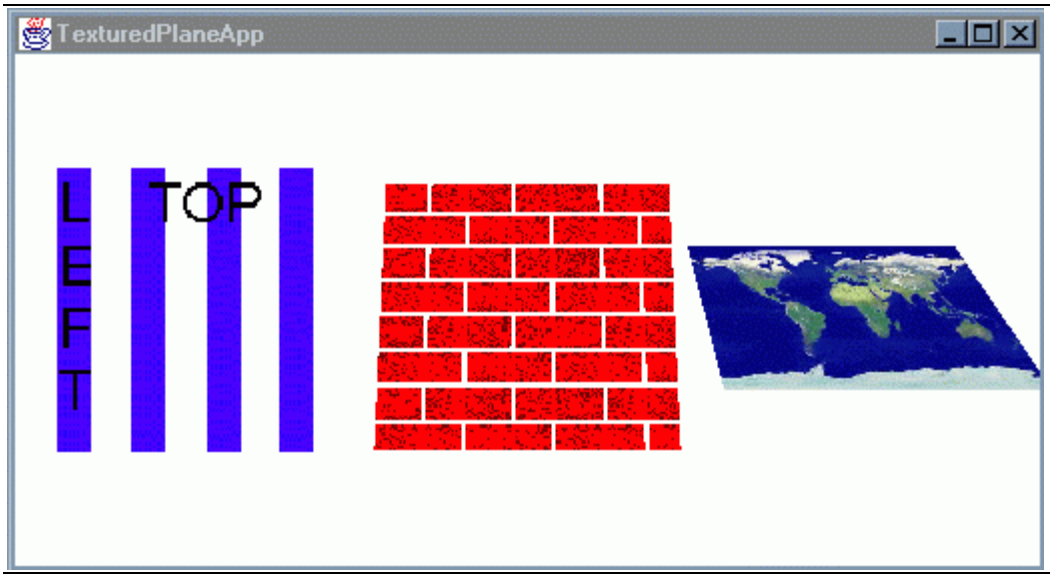


Figure 7-6 Three Textured Planes as Rendered By TexturedPlaneApp.java

Code Fragment 7-4 is an excerpt from `TexturedPlaneApp` and is nearly all the code required to create the three textured planes of that application. The image observer object is provided to the NewTextureLoader object of the TexturedPlane.

```
1.     scene.addChild(tg0);
2.     tg0.addChild(tg1);
3.     tg1.addChild(new TexturedPlane("stripe.gif"));
4.
5.     tg0.addChild(tg2);
6.     tg2.addChild(new TexturedPlane("brick.gif"));
7.
8.     tg0.addChild(tg3);
9.     tg3.addChild(new TexturedPlane("earth.jpg"));
```

Code Fragment 7-4 Adding Three New TexturedPlane Objects to the Scene Graph.

7.2.3 More about Texture Coordinates

As mentioned in "Texturing Step 3: Specify TextureCoordinates" (page 7-5), the texture image is made to fit the geometry based on the specification of the texture coordinates. The actual process is to map the *texels* of the texture to the pixels of the geometry as it is rendered. Each pixel of a texture is called a texel, or a 'texture element'. This is the process of texture mapping.

Texture mapping begins with the specification of texture coordinates for the vertices of the geometry. As each pixel of textured triangle is rendered, the texture coordinates for the pixel is calculated from the vertices of the triangle. Trilinear interpolation of the vertices' texture coordinates determine the texture coordinates for the pixel and therefore, the *texel* of the texture image used in the final color of the pixel.

Figure 7-7 illustrates the process of trilinear interpolation for an example pixel. Rendering is done in *scanline* order. The pixel, P, to texture map is roughly in the center of the current scanline in the triangle on the left of the illustration. Texture coordinates have been assigned to each of the vertices of the triangle. They are labeled TC1, TC2, and TC3. These texture coordinates are the starting point for the trilinear interpolation (each of the linear interpolations are shown as two-headed arrows in the figure). The first two linear interpolations determine the texture coordinates along the sides of the triangle at the scanline (labeled points A and B in the figure). The third interpolation is done between these two points. The resulting texture coordinates for P are (0.75, 0.6). On the right of the figure is the texture. Using the calculated texture coordinates for P the texel is selected.

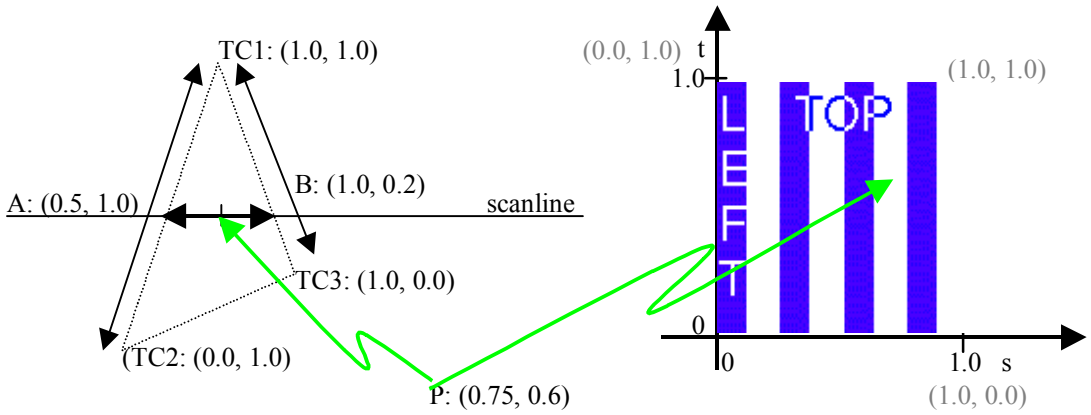


Figure 7-7 Picture of Texture Mapping

Texel selection is not fully explained in the above example. The Specification of Filtering section (page 7-13) gives more details on texel selection. Another detail not yet explained is the interaction between the texel color, other sources of color, and the final pixel color. The default mode is 'replace' in which the texel color is used as the color of the pixel, but there are other modes as explained in Section 7.4.1. Before moving on to other topics, further discussion of texture coordinates and mapping is in order.

To this point in the chapter all of the textures have been used in their ordinary orientation. Figure 7-8 shows planes with a few of the texture orientations possible just by setting the texture coordinates of the vertices. The `TextureCoordApp` example program produces this image.

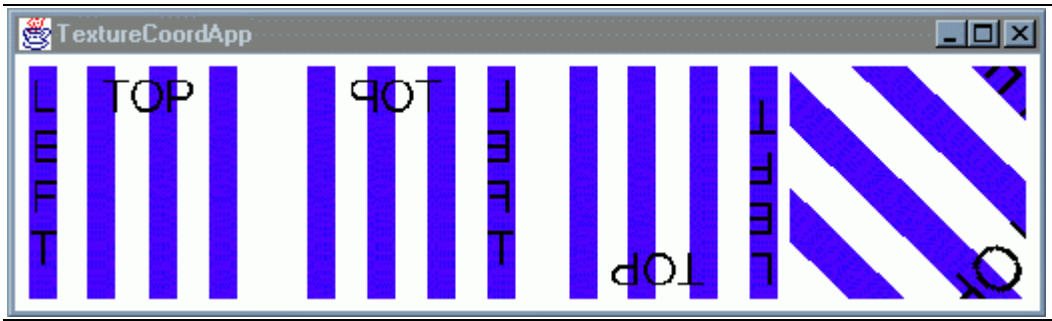


Figure 7-8 Some of the Possible Orientations for a Texture on a Plane.

You should note that in the TextureCoordinatesApp example program the stripe.gif texture is loaded only once. Only one texture appearance bundle is created which is shared by all four textured planes. This is

possible because there is nothing in the texture bundle that is unique for any of the planes. Loading the texture only once saves time and memory.

Of course, mistakes can be made in specifying the texture coordinates. When this happens, the Java 3D rendering system does what is asked of it. When the texture coordinates are not specified for regularly spaced mapping, then the triangulation of the geometry becomes obvious as the 'seams' of the texture will occur along the edges of triangles.

Figure 7-9 shows the image rendered for textured planes where the texture coordinates are not specified to make a uniform presentation of the texture. In the program that generates this image, `TextureRequestApp`, there is only one texture bundle shared by the three visual objects. The variations in the appearance of the planes is due only to the specification of the texture coordinates. This is a rendering of the phrase "In texturing, you get what you ask for."

Figure 7-9 In Texturing, You Get What You Ask For.

This program shows some of the possible renderings for a plane using the same texture. The texture assignments made in this program are examples of possible mistakes while all are legitimate applications. The left-most image is an application of only a single row of texels – the same texture coordinates are assigned to pairs of vertices. The right-most image is the application of a single texel – all four texture coordinates are the same. The middle two images demonstrate the assignment of texture coordinates in non-uniform ways. The change of the texture along the diagonal is due to the triangulation of the polygon.

Code Fragment 7-5 shows the texture coordinate assignments made in TextureRequestApp. These assignments used with the stripe.gif texture result in the images of Figure 7-9.

```
1.      // texture coordinate assignments fof the first plane
2.      texturedQuad.setTextureCoordinate(0, 0, new TexCoord2f( 1.0f, 0.0f));
3.      texturedQuad.setTextureCoordinate(0, 1, new TexCoord2f( 1.0f, 0.0f));
4.      texturedQuad.setTextureCoordinate(0, 2, new TexCoord2f( 0.0f, 0.0f));
5.      texturedQuad.setTextureCoordinate(0, 3, new TexCoord2f( 0.0f, 0.0f));

6.      // texture coordinate assignments for the second plane
7.      texturedQuad.setTextureCoordinate(0, 0, new TexCoord2f( 0.0f, 1.0f));
8.      texturedQuad.setTextureCoordinate(0, 1, new TexCoord2f( 1.0f, 0.5f));
9.      texturedQuad.setTextureCoordinate(0, 2, new TexCoord2f( 0.5f, 0.5f));
10.     texturedQuad.setTextureCoordinate(0, 3, new TexCoord2f( 0.0f, 1.0f));

11.     // texture coordinate assignments for the third plane
12.     texturedQuad.setTextureCoordinate(0, 0, new TexCoord2f( 1.0f, 0.0f));
13.     texturedQuad.setTextureCoordinate(0, 1, new TexCoord2f( 1.0f, 1.0f));
14.     texturedQuad.setTextureCoordinate(0, 2, new TexCoord2f( 0.0f, 0.0f));
15.     texturedQuad.setTextureCoordinate(0, 3, new TexCoord2f( 1.0f, 1.0f));
```

```
16.     // texture coordinate assignments for the forth plane
17.     texturedQuad.setTextureCoordinate(0, 0, new TexCoord2f( 0.0f, 0.0f));
18.     texturedQuad.setTextureCoordinate(0, 1, new TexCoord2f( 0.0f, 0.0f));
19.     texturedQuad.setTextureCoordinate(0, 2, new TexCoord2f( 0.0f, 0.0f));
20.     texturedQuad.setTextureCoordinate(0, 3, new TexCoord2f( 0.0f, 0.0f));
```

Code Fragment 7-5 Texture Coordinate Assignments for Planes in TextureRequestApp.

The complete source for the `TextureRequestApp` and `TextureCoordApp` example programs is available in the examples jar available with the tutorial.

7.2.4 A Preview of Some Texturing Choices

There is much more to texturing than just specifying the texture coordinates for the vertices of the geometry. To this point, the discussion of texturing has not included any of the options available in texture applications. For example, the Texture2D object can be configured for different boundary modes and mapping filters. Section 7.2.5 presents these options. But there is even more than this.

Additional configuration of a texture is done through a TextureAttributes node component. Figure 7-10 shows a visual object with an appearance bundle with both Texture and TextureAttributes components. Section 7.4 presents the details of the TextureAttributes components.

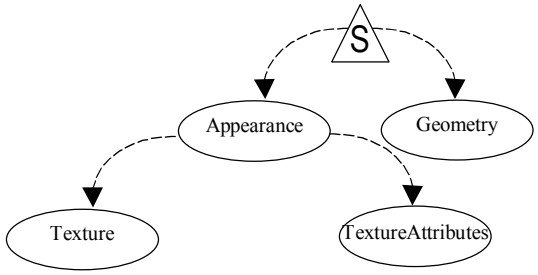


Figure 7-10 An Appearance Bundle with Texture and TextureAttributes Components.

Other options in texturing are beyond the settings in Texture and TextureAttributes. For example, a texture can be three dimensional. Section 7.7 presents the API for the Texture3D class, which, like Texture2D, is an extension of the Texture class. Section 7.6 presents Multiple level textures, commonly called MIPmaps, and their applications. Section 7.5 presents a utility for automatic texture coordinate generation.

Since many of these options are intertwined in the API, the API details appear at the end of the chapter after all of the various options have been discussed.

7.2.5 Texture Options

Texture2D, the class used in the previous examples, is an extension of Texture. Some of the basic choices for texturing are implemented in the Texture class. Since Texture is an abstract class, your settings will be made through either a Texture2D or Texture3D object. The settings are Boundary Mode, Filters, and Texture Format.

Boundary Mode: Wrap or Clamp

In all of the previous programs, the textures have been mapped in such a way that one copy of the image has been used to cover the plane. The issue of what to do when a texture coordinate is beyond the 0 to 1 range of the texture space was not addressed.

The boundary mode setting determines what mapping takes place when the texture coordinates go beyond the 0 to 1 range of the image space. The choices are to wrap the image, or to clamp it. Wrapping, which means to repeat the image as needed, is the default. Clamping the image uses the color from the edge of the image anywhere outside of the 0 to 1 range. These settings are made independently in the s and t dimensions.

In the `BoundaryModeApp` example program the texture is mapped onto approximately the middle ninth of each of the planes. Figure 7-11 shows the scene as rendered by this program. The variations in the images are due only to the setting of the boundary modes for the planes. From left to right the settings are (s then t) WRAP and WRAP, CLAMP and WRAP, WRAP and CLAMP, and CLAMP and CLAMP. Consult the source code of this example for specific details of this program. Section 7.7.2 presents the API for this topic in more detail.

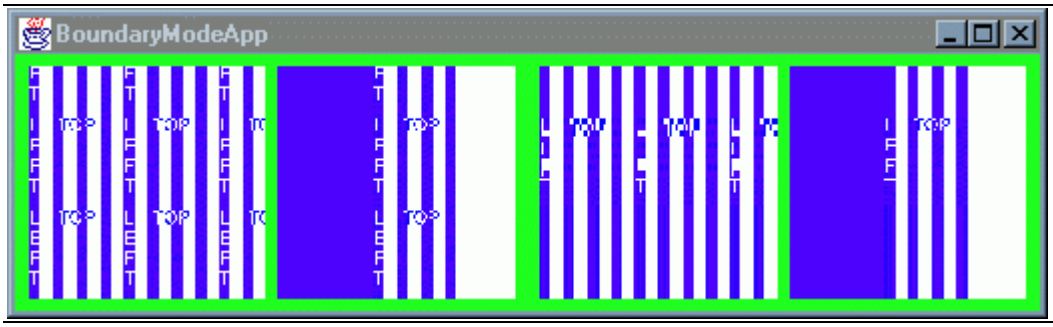


Figure 7-11 Comparing the Combinations of Boundary Mode Settings for a Textured Plane.

Note that unlike the previous applications which share the same texture object among four visual objects, the texture is loaded four times in this application. This is necessary since each of the texture objects has different combinations of Boundary Mode settings.

Specification of Filtering

In the computation of texture coordinates for each pixel, rarely does a pixel map directly to just one texel. Usually a pixel is either the size of multiple texels or smaller than one texel. In the former case a magnification filter is used to map the multiple texels to one pixel. In the later case a minification filter is used to map the texel or texels to the pixel. There are choices for how to handle each of these cases.

The magnification filter specifies what to do when a pixel is smaller than a texel. In this case the texture will be magnified as it is applied to the geometry. Each texel will appear as several pixels and it is possible for the resulting image to exhibit "texelization" where the individual texels would be seen in the rendering. The choices for magnification filter are to do point sampling, which is to select the nearest texel and use its color, or to interpolate among neighboring texels. The point sampling, or nearest neighbor sampling, filter usually has the least computational cost; while the linear interpolation sampling typically costs more (in computation and therefore rendering time) but reduces the appearance of any texelization[8].

The minification filter specifies what to do when a pixel is larger than a texel. In this case the texels must be "minified" (opposite of magnified) to fit the pixel. The problem lies in that a pixel can only have one color value and yet several texels could supply the color. The choices for the minification filter are to do point sampling, which is to select the nearest texel and use its color, or to interpolate among neighboring texels.

[8] Performance differences will vary significantly among different platforms.

It is not always clear which filter will be used. Consider a texture stretched in one direction but squashed in another. Depending on which dimension is considered, a different filter would be picked. There is nothing the programmer can do to determine which will be used. However, the runtime system usually picks the filter that results in the better image.

The selection of a filter has other implications with respect to the use of the boundary color (next section). Also, the minification filter choices are more complex when a multiple level texture is used. Multi level texturing is discussed in Section 7.6. Section 7.7.1 gives the API details for filter settings.

Boundary Color

The boundary mode behavior is further configurable by a boundary color. When the boundary mode is CLAMP and boundary color is specified, the boundary color is used when texture coordinates are outside of the 0 to 1 range. Only one boundary color can be specified so that one color is used in each dimension for which the boundary mode is set to clamp. The example program `BoundaryColorApp` demonstrates this feature.

In Figure 7-12 boundary colors are set for all four planes. The left most plane does not use its boundary color as the boundary modes are both WRAP. For the next plane the black boundary color is used in the vertical dimension only due to the boundary modes. You can see the blend between the blue and black on the left; on the right side of the image the black boundary color blends with the white edge of the texture. The boundary colors for the remaining two planes are green and red. Both boundary modes are CLAMP for the rightmost plane in the image.

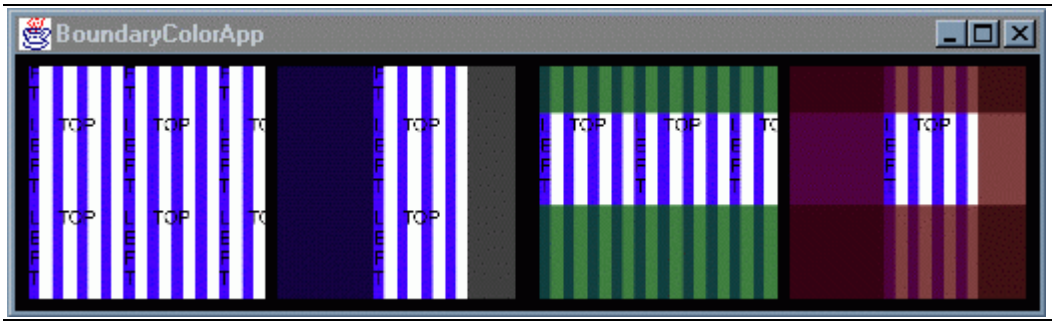


Figure 7-12 Image Produced by BoundaryColorApp.

Note that the Boundary Color is not used if the filter is BASE_LEVEL_POINT. For the Boundary Color to be used, the filter needs to be at least BASE_LEVEL_LINEAR. The corollary is that anytime the filter is not BASE_LEVEL_POINT the BoundaryColor will be used.

Also note that the same texture is loaded four times in this application. One texture object can not be shared among the four planes in this application since each texture object is configured with a different combination of Boundary Mode settings.

Texture Format

The last setting of the Texture class is that of the texture format. The texture format is both a statement of how many values there are per texel and how those values effect pixels. For example, a texture format setting of INTENSITY states that the single texel value will be used for red, green, blue, and alpha values of the pixel. A texture format setting of RGB states that the three texel values will be used for red, green, and blue values of the pixel while the alpha value of the pixel remains the same.

Table 7-1 How Texture Format Affect Pixels

Texture Format	values per texel	modify pixel color	modify pixel alpha
INTENSITY	1	yes, R=G=B	yes, R=G=B=A
LUMINANCE	1 (color only)	yes, R=G=B	no
ALPHA	1 (alpha only)	no	yes
LUMINANCE_ALPHA	2	yes, R=G=B	yes
RGB	3	yes	no
RGBA	4	yes	yes

The texture mode, or how the texel values are used to change the pixel value, is a setting of the texture attribute object.

7.2.6 Texture3D

As the name implies, a Texture3D object holds a three dimensional texture image. You might think of it as being a volume of color. The Texture3D class is an extension of Texture, so all of the features of the Texture class as explained in the previous section (Section 7.2.5) applies to Texture3D. The only feature that Texture3D has that Texture2D does not is a specification for the boundary mode in the third dimension, or r dimension.

7.3 Some Texturing Applications

Believe it or not, there are many more texturing features to be discussed. However, you can use the features already discussed in many applications. This section takes a break from discussing the details of texturing to demonstrate two applications of texturing. One application is to apply a texture to a geometric primitive (see Chapter 2). Another is to texture the lines of non-filled polygons. A third application uses the texture created by a Text2D (see Chapter 3) object to another visual object.

7.3.1 Texturing Geometric Primitives

One way to simplify the process of presenting a texture is to use a geometric primitive. A flag can be used to have texture coordinates automatically assigned when creating geometric primitives. Code Fragment 7-6 shows the use of a constructor for a Sphere primitive with the coordinate generation.

```
1. objRoot.addChild(new Sphere(1.0f, Primitive.GENERATE_TEXTURE_COORDS, appear));
```

Code Fragment 7-6 Creating a Sphere Primitive with Pre-Assigned Texture Coordinates.

The line of Code Fragment 7-6 is used in the `PrimitiveTextureApp` example program. The complete source for this application is available in the `examples/texture` subdirectory of the example program jar distributed with this tutorial. This program textures the sphere with the `earth.jpg` image, also in the examples jar, resulting in the image of Figure 7-13.

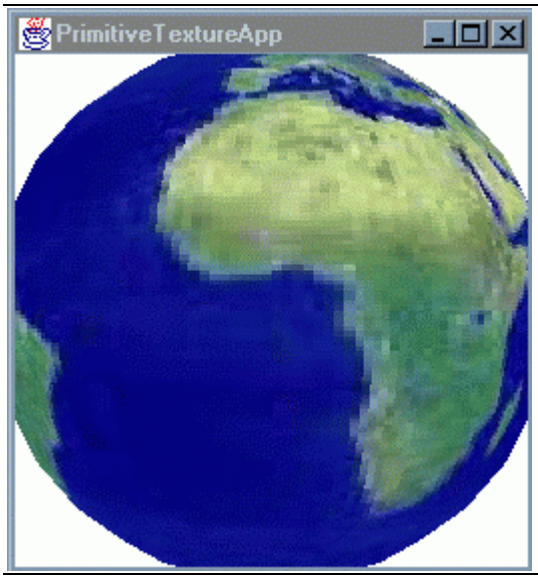


Figure 7-13 Texturing the Sphere Geometric Primitive.

7.3.2 Texturing Lines

Polygons are not the only graphic elements that can be textured; lines can be textured too. The `TexturedLinesApp` demonstrates using a 1-D texture to texture lines of a visual object. In this application, the twisted strip visual object created in Chapter 2 is used. The only 'trick' to texturing the lines is to create the appropriate appearance bundle to display the lines of the geometry and not filled polygons. Code Fragment 7-7 shows the lines of code to add the PolygonAttributes component to an appearance bundle to display lines.

```
1.      PolygonAttributes polyAttrib = new PolygonAttributes();
2.      polyAttrib.setCullFace(PolygonAttributes.CULL_NONE);
3.      polyAttrib.setPolygonMode(PolygonAttributes.POLYGON_LINE);
4.      twistAppear.setPolygonAttributes(polyAttrib);
```

Code Fragment 7-7 Creating an Appearance Bundle to Display the Lines of a Geometry Array.

A one dimensional texture is really a Texture2D object with one dimension (usually t) having size 1. For the example program, the texture is 16 texels by 1 texel. Two dimensional texture coordinates are assigned to the visual object. The t-value of every texture coordinate is set to 0.0f. However, any t-value could be used and the result would be the same. Figure 7-14 shows the twisted strip[9] geometry displayed as textured lines. The complete source code and texture for `TexturedLinesApp` appears in the `examples/texture` subdirectory of the examples jar distributed with the tutorial.

[9] The twisted strip geometry used in this example first appears in Chapter 2 of the tutorial.

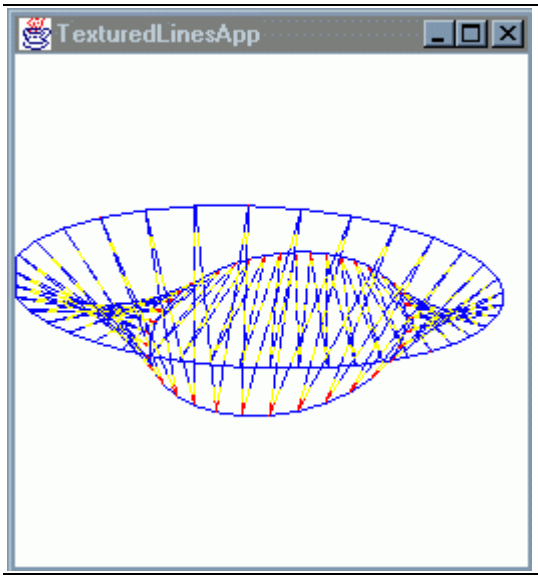


Figure 7-14 Textured Lines in Java 3D.

7.3.3 Using Text2D Textures

Text2D objects create textures of specified text and apply the texture to a polygon. This texture can be easily accessed from the Text2D and applied to another visual object. Figure 7-15 shows the image produced by `Text2DTextureApp`, a program in the examples jar, which applies the texture created by the Text2D object shown in the background to geometry of another visual object as seen in the foreground.

Figure 7-15 The Texture of a Text2D Object Applied to Another Visual Object.

7.4 Texture Attributes

Section 7.2 presented some of the options available in texturing. The TextureAttributes node component allows further customization of the texturing. Texture attribute settings include the texture mode, blend color, perspective correction mode, and a texture map transform. The default values for these settings are REPLACE, black, FASTEST, and NONE, respectively. In addition, the setEnable method allows enabling and disabling of texture mapping. Each of the settings are explained in this section.

One benefit of having texturing features controlled by a different node component is the ability to share a texture among visual objects but still be able to customize it for each visual object. Figure 7-10 shows two visual objects sharing a single texture object. Each of the visual objects customize the texture with the TextureAttributes component in its appearance bundle.

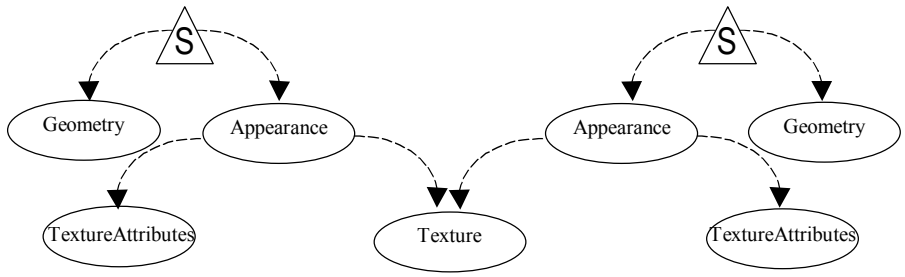


Figure 7-16 Two Visual Objects Sharing a Texture Customized by TextureAttributes Components.

TextureAttributes objects are added to the scene graph as members of an appearance bundle. The Appearance method is setTextureAttributes, as shown in the following reference block.

Appearance setTextureAttributes method

```
void setTextureAttributes(TextureAttributes textureAttributes)
```
Sets the textureAttributes object in an appearance object.

7.4.1 Texture Mode

To appreciate the role of the texture mode you must understand the sequence of operations involved in determining the color of a pixel. Very simply stated, a pixel's non-texture color is calculated first, then the texture is applied. The non-texture color is determined from geometry per-vertex color, ColoringAttributes, or by the combination of material properties and lighting conditions. Just as there are several ways to determine the non-texture color, there are several possible ways to combine the non-texture color and the texture color.

The texture mode setting is a major factor in determining how the texel value (color and/or alpha) affects the non-texture pixel color and alpha values. The actual texturing operation depends on the combination of the texture format (see Section 7.2.5) and the texture mode. Refer to the Java 3D API Specification (Appendix E) for more information.

The default texture mode is REPLACE, the other options are BLEND, DECAL, and MODULATE. Each of the modes is described in the following subsections. Also note Table 7-2 (page 7-19) which summarizes the texture modes.

Blend

In BLEND mode, the texture color blends with the non-texture color. The texture color determines the amount of the non-texture color to use. The resulting transparency is the combination of the texture and material transparency. This particular texture mode has the added flexibility of optionally including a blend color. Section 7.4.2 for more information on blend color.

Decal

In DECAL mode, the texture color is applied as a decal on top of the non-texture color. The transparency of the texture determines the amount of material color to use. The transparency of the pixel

is left unchanged. This is completely analogous to applying a decal to a real world object. The texture format must be RGB or RGBA for DECAL texture mode.

Modulate

In MODULATE mode the Texture color is combined with the non-texture color. The resulting transparency is the combination of the texture and material transparency. Since the resulting color depends on both the non-texture and the texture colors, this mode is useful in applying the same texture to a variety of visual objects without having them all look the same. This mode is often used in lit scenes.

Replace

In REPLACE mode the texture provides the color and transparency for the pixel, ignoring all other colors except the specular color (if lighting is enabled). This is the default texture mode even when there is no TextureAttributes component in the appearance bundle.

Texture Modes Summary

Table 7-2 gives summary information for each of the texture modes. This table is meant as a general guide to understanding the variety of texture modes available. The actual color calculations are based on a combination of texture mode and texture format[10].

When lighting is enabled, the ambient, diffuse, and emissive color components are all affected by the texture operation; specular color is not. The specular color is calculated normally based on material and lighting conditions, then after the texture operation is applied to the other color components the (untextured) specular color is added to the other colors yielding the final pixel color.

Table 7-2 Summary of Texture Modes

Texture Mode	Pixel Color Derived From	Determined By	Application Hint
BLEND	Texture color, non-texture, and optional blend color	texture color	lit scenes with blending color
DECAL	Texture color and non-texture color	alpha of texture	detailing a surface
MODULATE	Texture color and non-texture color	n/a	lit scenes
REPLACE	texture color only (default texture mode)	n/a	non-lit scenes

7.4.2 Texture Blend Color

The blend color is used in texturing only when the texture mode is BLEND. The resulting pixel color is a combination of the texel color and the blend color. With the blend color the same texture can be applied in a variety of shades to different visual objects. The blend color is expressed as an RGBA value. The default blend color is (0,0,0,0) black with and alpha of 0.

[10] The actual texture operation may vary by implementation of Java 3D.

7.4.3 Perspective Correction Mode

Texture mapping takes place in the image space. For this reason the textured planes may appear incorrect when viewed from an edge. That is, they appear incorrect unless a perspective correction is made. In Java 3D the perspective correction is always made. The choice is how this perspective correction is made.

The two options are FASTEST and NICEST. Obviously, the choice is the classic speed versus image quality tradeoff. For this option, the default setting is NICEST.

7.4.4 Texture Map Transform

Within the Texture Attributes component a Transform3D object can be specified to alter the texture mapping function. This texture map transform can be used to move a texture on a visual object at run time. The transform translates, rotates, and scales texture coordinates (s,t,r) before the texels are selected from the texture image.

A translation in the texture transform would slide the texture across the visual object. A rotation transformation could be used to reorient the texture on a visual object. A scale transformation can be used to repeat the texture across a visual object. Of course, since a transform object can contain a combination of these, the texture can be animated on a visual object by manipulating the texture transform.

7.4.5 TextureAttributes API

The following reference blocks list the constructors, methods, and capabilities of the Texture Attributes node component.

TextureAttributes Constructor Summary

extends: `NodeComponent`

The TextureAttributes object defines attributes that apply to texture mapping. See the following reference block for a list of texture mode and perspective correction mode constants. Consult the text for more information.

`TextureAttributes()`
Constructs a TextureAttributes object with default settings:
texture mode : REPLACE, transform : null, blend color : black (0,0,0,0), perspective correction: NICEST

```
TextureAttributes(int textureMode, Transform3D transform,
                  Color4f textureBlendColor, int perspCorrectionMode)
```
Construct a TextureAttributes with specified values.

TextureAttributes Capabilities Summary

`ALLOW_COLOR_TABLE_READ` **`<new in 1.2>`**
Specifies that this TextureAttributes object allows reading its texture color table component information.

`ALLOW_COLOR_TABLE_WRITE` **`<new in 1.2>`**
Specifies that this TextureAttributes object allows writing its texture color table component information.

TextureAttributes Constants

These constants are used in constructors and methods for setting the texture and perspective correction modes.

Texture Mode Constants

`BLEND`	Blend the texture blend color with the object color.
`DECAL`	Apply the texture color to the object as a decal.
`MODULATE`	Modulate the object color with the texture color.
`REPLACE`	Replace the object color with the texture color.

Perspective Correction Mode Constants

`FASTEST`	Use the fastest available method for texture mapping perspective correction.
`NICEST`	Use the nicest (highest quality) available method for texture mapping perspective correction.

TextureAttributes method summary (partial list)

`void setTextureColorTable(int[][] table)` **`<new in 1.2>`**

Sets the texture color table from the specified table. The individual integer array elements are copied. The array is indexed first by color component (r, g, b, and a, respectively) and then by color value; table.length defines the number of color components and table[0].length defines the texture color table size. If the table is non-null, the number of color components must either be 3, for rgb data, or 4, for rgba data. The size of each array for each color component must be the same and must be a power of 2. If table is null or if the texture color table size is 0, the texture color table is disabled. If the texture color table size is greater than the device-dependent maximum texture color table size for a particular Canvas3D, the texture color table is ignored for that canvas.

When enabled, the texture color table is applied after the texture filtering operation and before texture application. Each of the r, g, b, and a components are clamped to the range [0,1], multiplied by textureColorTableSize-1, and rounded to the nearest integer. The resulting value for each component is then used as an index into the respective table for that component. If the texture color table contains 3 components, alpha is passed through unmodified.

Parameters:
table - the new texture color table

`int getNumTextureColorTableComponents()` **`<new in 1.2>`**

Retrieves the number of color components in the current texture color table. A value of 0 is returned if the texture color table is null.

Returns: the number of color components in the texture color table, or 0 if the table is null

`int getTextureColorTableSize()` **`<new in 1.2>`**

Retrieves the size of the current texture color table. A value of 0 is returned if the texture color table is null.

Returns: the size of the texture color table, or 0 if the table is null

TextureAttributes Method Summary

See the following reference block for a list of texture mode and perspective correction mode constants. Consult the text of this chapter for more information.

void getTextureBlendColor(Color4f textureBlendColor)
Gets the texture blend color for this appearance component object.

void getTextureTransform(Transform3D transform)
Retrieves a copy of the texture transformation object.

void setPerspectiveCorrectionMode(int mode)
Sets perspective correction mode to be used for color and/or texture coordinate interpolation to one of:
`FASTEST` Use the fastest available method for texture mapping perspective correction.
`NICEST` Use the nicest (highest quality) available method for texture mapping perspective correction.

void setTextureBlendColor(Color4f textureBlendColor)
void setTextureBlendColor(float r, float g, float b, float a)
Sets the texture blend color for this TextureAttributes object.

void setTextureMode(int textureMode)
Sets the texture mode parameter to one of:
`BLEND` Blend the texture blend color with the object color.
`DECAL` Apply the texture color to the object as a decal.
`MODULATE` Modulate the object color with the texture color.
`REPLACE` Replace the object color with the texture color.

void setTextureTransform(Transform3D transform)
Sets the texture transform object used to transform texture coordinates.

TextureAttributes Capabilities Summary

ALLOW_BLEND_COLOR_READ | WRITE	Allow reading (writing) texture blend color
ALLOW_MODE_READ | WRITE	Allow reading (writing) texture and perspective correction modes.
ALLOW_TRANSFORM_READ | WRITE	Allow reading (writing) texture transform.

7.5 Automatic Texture Coordinate Generation

As previously discussed, assigning texture coordinates to each vertex of the geometry is a necessary step in texturing visual objects. This process can be time consuming as well as difficult for large and/or complex visual objects. Keep in mind that this is a problem for the programmer and once solved, it is not a recurring problem.

Texture coordinates are often assigned with code specific to a visual object. However, another solution is to automate the assignment of texture coordinates via some method. This method could be used for any visual object whether large or small, complex or simple. This approach is exactly what a TexCoordGeneration (texture coordinate generation) object does. Whether an object is loaded from a file or created in the program code, a texture coordinate generation object can be used to assign texture coordinates.

TexCoordGeneration is a Java 3D API core class used to generate texture coordinates. To automatically generate texture coordinates, the programmer specifies the texture coordinate parameters in a TexCoordGeneration object and adds that object to the appearance bundle of the visual object. The texture coordinates are calculated based on the coordinate specification parameters at runtime. The parameters are explained in the following sections.

7.5.1 Texture Generation Format

This setting simply specifies if the texture coordinates will be generated for a two or three dimensional texture. The possible settings are `TEXTURE_COORDINATE_2` and `TEXTURE_COORDINATE_3` which generates 2D texture coordinates (S and T) and 3D texture coordinates (S, T, and R), respectively.

7.5.2 Texture Generation Mode

There are two basic texture generation approaches: linear projection or sphere mapping. The following two subsections explain these options.

Linear Projection

With linear projection, the texture coordinates are specified with planes. For texture coordinates of two dimensions (s,t), two planes are used. The distance from a vertex to one plane is the texture coordinate in one dimension; distance to the other plane to a vertex is the texture coordinate in the other dimension. For three dimensional textures, three planes are used.

The three possible plane parameters are named planeS, planeT, and planeR, where the name corresponds to the dimension for which it is used. Each plane is specified as a 4-tuple (plane equation). The first three values are the surface normal vector for the plane. The fourth value specifies the distance from the origin to the plane along a vector parallel to the plane's surface normal vector.

There are two variations on this automatic texture coordinate generation method. The first, called object linear, produces static texture coordinates. With object linear generated texture coordinates, if the visual object moves, the texture coordinates do not change. The second option, called eye linear, produces texture coordinates relative to the eye coordinates resulting in variable texture coordinates for the object. With eye linear texture coordinates moving objects appear to move through the texture.

Figure 7-17 shows images produced by an example program that uses a `TexCoordGeneration` object to assign texture coordinates to a twisted strip. A one dimensional texture is used for this application. The texture has a single red texel at one end. When the application runs, the twisted strip rotates.

The image on the left of Figure 7-17 shows the texturing with the OBJECT_LINEAR generation mode. In this case the texture rotates with the object and you can see the red texel rotate with the strip. The image on the right of Figure 7-17 shows the resulting texture when the EYE_LINEAR generation mode is used for the twisted strip. In this case, the red texel stays in the center of the view as the object rotates.

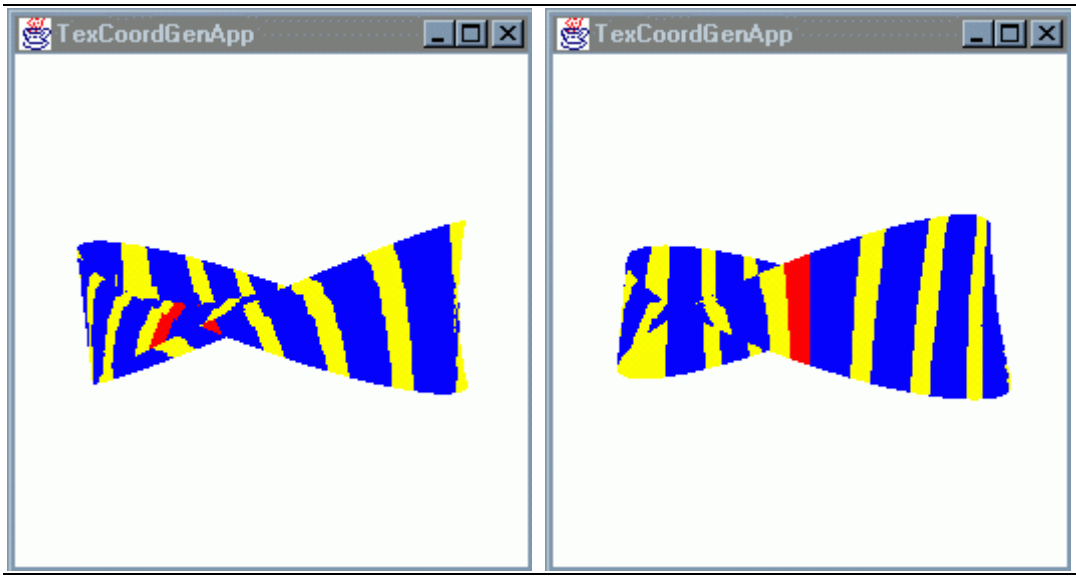


Figure 7-17 Comparing Generation Modes of the TexCoordGeneration Object.

`TexCoordGenApp` is the program that produces these images and is available in the texture subdirectory of the examples jar distributed with this tutorial. This is one application you should run to see the difference. The example program is written with the Generation Mode set to `EYE_LINEAR`. Line 100 is the place to change to `OBJECT_LINEAR` generation mode.

Sphere Map

If a shiny object is in the middle of a real room, the shiny object would likely reflect the image of many of the other objects in the room. The reflections would depend on the shape of the object and orientation of things in the room. The sphere map coordinate generation mode is designed to assign texture coordinates to approximate the reflections of other objects onto the visual object as would happen for the shiny object in the example real world.

When a TexCoordGeneration object is used in sphere map generation mode the texture coordinates are calculated based on the surface normals and the viewing direction.

The texture used for this effect must be specially prepared. If the virtual environment of the shiny object exists in the real world, a photograph of the scene taken with a fisheye lens will create a suitable texture image. If the scene does not exist, then the texture must be created to look like the image is a photograph taken with a fisheye lens.

7.5.3 How to use a TexCoordGeneration Object

To use a TexCoordGeneration Object, set it as a component of an appearance bundle for the visual object to be textured. Figure 7-18 shows the diagram of an appearance bundle with an TexCoordGeneration object along with a Texture and TextureAttributes object.

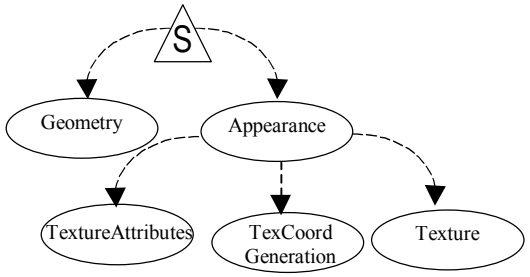


Figure 7-18 Appearance Bundle with Texture, TextureAttributes, and TexCoodGeneration.

The following reference block shows the Appearance method for setting a TexCoordGeneration object as a component of an appearance bundle.

Appearance setTexCoordGeneration method

```
void setTexCoordGeneration(TexCoordGeneration texCoordGeneration)
```
Sets the texCoordGeneration object to the specified object.

7.5.4 TexCoordGeneration API

The following reference blocks list the constructors, constants, methods, and the capabilities for TexCoordGeneration class objects.

TexCoordGeneration Constructor Summary

The TexCoordGeneration object contains all parameters needed for texture coordinate generation. It is included as part of an Appearance component object.

```
TexCoordGeneration()
```
Constructs a TexCoordGeneration object using defaults for all state variables.

```
TexCoordGeneration(int genMode, int format)
```
Constructs a TexCoordGeneration object with the specified genMode and format.

```
TexCoordGeneration(int genMode, int format, Vector4f planeS)
TexCoordGeneration(int genMode, int format, Vector4f planeS, Vector4f planeT)
TexCoordGeneration(int genMode, int format, Vector4f planeS, Vector4f planeT,
                   Vector4f planeR)
```
Constructs a TexCoordGeneration object with the specified genMode, format, and the coordinate plane equation(s).

TexCoordGeneration Field Summary

The TexCoordGeneration object contains all parameters needed for texture coordinate generation. It is included as part of an Appearance component object.

Generation Mode Constants

EYE_LINEAR	Generates texture coordinates as a linear function in eye coordinates. (default)
OBJECT_LINEAR	Generates texture coordinates as a linear function in object coordinates.
SPHERE_MAP	Generates texture coordinates using a spherical reflection mapping in eye coordinates.

Format Constants

TEXTURE_COORDINATE_2	Generates 2D texture coordinates (S and T) (default)
TEXTURE_COORDINATE_3	Generates 3D texture coordinates (S, T, and R)

TexCoordGeneration Method Summary

void setEnable(boolean state)
Enables or disables texture coordinate generation for this appearance component object.

void setFormat(int format)
Sets the TexCoordGeneration format to the specified value.

void setGenMode(int genMode)
Sets the TexCoordGeneration generation mode to the specified value.

void setPlaneR(Vector4f planeR)
Sets the R coordinate plane equation.

void setPlaneS(Vector4f planeS)
Sets the S coordinate plane equation.

void setPlaneT(Vector4f planeT)
Sets the T coordinate plane equation.

TexCoordGeneration Capabilities Summary

ALLOW_ENABLE_READ | WRITE	allows reading/writing its enable flag.
ALLOW_FORMAT_READ	allows reading its format information.
ALLOW_MODE_READ	allows reading its mode information.
ALLOW_PLANE_READ	allows reading its planeS, planeR, and planeT component information.

7.6 Multiple Levels of Texture (Mipmaps)

To understand the reason for multiple levels of texture, consider an application which contains a textured visual object which moves about in the scene (or the viewer moves). When this visual object is near the viewer it appears as many pixels in the image. For this case, a texture of good size should be used to avoid viewing individual texels; this is especially true when point sampling is used for the magnification filter.

However, when this visual object is viewed as a distance, the texture will be much too large for the visual object and the texture will be mininified during the rendering. (Recall that texture mapping takes

place at render time in the image, or screen, space.) Point sampling for the minification filter will probably not yield satisfactory results when the visual object appears 1/32 or smaller (in pixel size) than the texture resolution. The tradeoff is image quality for rendering performance.

If instead of using a large texture map (because the visual object will appear large) a small one is used to make the visual object look better when it is small, the reverse problem exists. For good quality images the magnification filter will involve linear interpolation resulting in more computation. Once again, the tradeoff is for image quality versus rendering performance. The only advantage that using a smaller texture map has is a reduced memory requirement for storing the texture.

What is needed is a small texture map when the visual object is appears small and a large texture map when the visual object appears large. The current texturing technique using one texture image, called base level texturing, can not do this. That is exactly what multiple levels of texture provides.

7.6.1 What is Multi Level Texturing (MIPmap)

Multiple Levels of Texture refers to a texturing technique where a series of texture images together are used as the texture for visual objects. The series of images is (usually) the same texture at a variety of resolutions. When a visual object is being rendered with multiple levels of texture, the texture image that is closest to the screen size of the visual object is used.

The performance of the renderer depends on the minification and magnification filters used (see Sections 7.2.5 and 0). However, with MIPmaps you have more control over the appearance of the visual objects and can get better looking visual objects with better performance[11].

Using multiple levels of texture is like using a DistanceLOD object (see Section 5.4) to apply different textures to a visual object when it is viewed from different distances. The exceptions are that with the Mipmap the visual object will always be textured whereas with the DistanceLOD object, the object could be untextured at some distances. And, for visual objects textured at all distances, the MIPmap is more efficient and has added filtering possibilities as compared with a DistanceLOD object used for a similar application.

Multiple levels of texture is commonly referred to as a mipmap. The term "MIPmap" comes from an acronym of the Latin phrase *multum in parvo,* which means many things in a small place. The term MIPMap truly refers to a specific storage technique for storing a series of images for use in multilevel texturing. The term MIPmap is commonly used to mean multilevel texturing.

With the MIPmap storage technique, the size of a texture image is ¼ the size of the previous (½ the size in each dimension). This continues until the size of the smallest image is 1 texel by 1 texel. For example, if the full size texture is 16x4, the remaining textures are 8x2, 4x1, 2x1, and 1x1. Figure 7-19 shows the multiple levels of texture for the `stripe.gif` texture, each outlined. Each of these texture images was prepared using image editing software.

[11] The quality of appearance versus rendering performance tradeoff depends on the execution environment, choices of texture filters, the texture image, and the range of distances the visual object is viewed from.

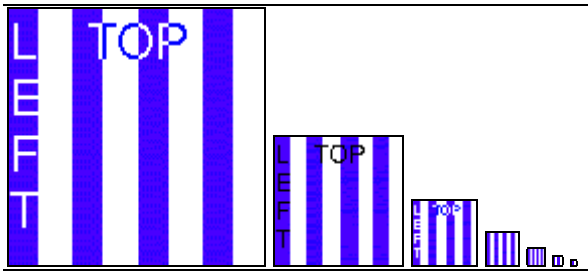


Figure 7-19 Multiple Levels of a Texture (outlined). (Image Sizes: 128x128, 64x64, 32x32, …, 1x1)

Figure 7-20 shows an image of a single plane textured with a multiple level texture where each level of the texture is a different color. The plane is oriented at an angle to the viewer such that the left side is much nearer to the viewer than the right side. Due to the perspective projection the left side of the plane appears larger in image coordinates than the right side.

Due to the orientation and projection of the plane, the pixels represent less of the surface area (in the virtual object coordinate system) on the left and progressively more visual object surface area proceeding to the right, resulting in the texture level changes. At the left of the plane in the image, the base level of the texture is used. The color changes in the image indicate where texture level changes occurred while rendering.

Having a texture where each level is a different color is not the typical application of multiple level texturing. This application simply illustrates the operation of a multiple level texture.

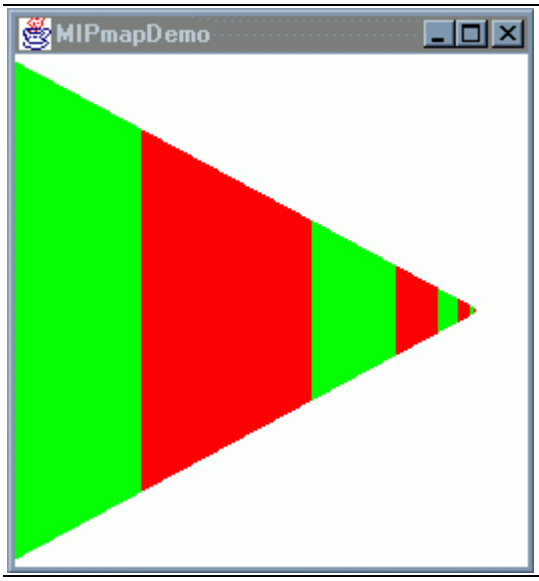


Figure 7-20 The Image Generated for a Plane Textured with a Multi Color Mipmap Texture.

Figure 7-20 is generated by `MIPmapDemo`, an example program available in the examples jar[12]. The texture in this program is created from the files named color<number>.gif (e.g., `color128.gif`, `color64.gif`, `color32.gif`, …) also in the examples jar.

[12] The example `MIPmapDemo.java` example application was inspired by a similar OpenGL example application in the OpenGL Programming Guide, third edition, by Mason Woo, et al.

7.6.2 Multiple Levels of Texture Examples

As far as programming with the Java 3D API is concerned, creating a multilevel texture is nearly the same as creating the single level, or base level, texture. Looking back at the simple texturing recipe (Figure 7-2) the only thing that differs is that multiple texture images are required for the multilevel texture. There are two ways to create the multiple levels of texture images. One way is to create each image by hand using the appropriate image editing/creation applications, the other uses a texture loader feature to create those images from the base image.

The two multiple level texturing techniques take about the same amount of code. The least amount of overall work is to generate the levels' images from the base image. Code Fragment 7-8 presents the texture loading code from `MIPmapApp.java`. This application is an example of generating multiple levels of texture from a base image. The complete code for this application is available in the tutorial's examples jar `example/texture` subdirectory.

```
1.   Appearance appear = new Appearance();
2.
3.   NewTextureLoader loader = new NewTextureLoader("stripe.gif",
4.                                      TextureLoader.GENERATE_MIPMAP);
5.   ImageComponent2D image = loader.getImage();
6.
7.   imageWidth = image.getWidth();
8.   imageHeight = image.getHeight();
9.
10.  Texture2D texture = new Texture2D(Texture.MULTI_LEVEL_MIPMAP,
11.                                    Texture.RGB, imageWidth, imageHeight);
12.  imageLevel = 0;
13.  texture.setImage(imageLevel, image);
14.
15.  while (imageWidth > 1 || imageHeight > 1){  // loop until size: 1x1
16.      imageLevel++;                          // compute this level
17.
18.      if (imageWidth > 1) imageWidth /= 2;   // adjust width as necessary
19.      if (imageHeight > 1) imageHeight /= 2; // adjust height as necessary
20.
21.      image = loader.getScaledImage(imageWidth, imageHeight);
22.      texture.setImage(imageLevel, image);
23.  }
24.
25.  texture.setMagFilter(Texture.BASE_LEVEL_POINT);
26.  texture.setMinFilter(Texture.MULTI_LEVEL_POINT);
27.
28.  appear.setTexture(texture);
```

Code Fragment 7-8 Creating Multiple Level Texturing from a Base Level Image Only.

Code Fragment 7-8 begins by following the same steps as are used for any texture application by loading the base image. One difference is that the TextureLoader is created with the `GENERATE_MIPMAP` flag set (lines 3-4). Then the base image is retrieved from the loader in the usual way.

The dimensions of this image are needed not only to create the Texture2D object, but also to calculate the sizes of the images that follow. For this reason they recorded in two variables (lines 7 and 8). These variables will be used while generating and loading the remaining images for the texture.

The Texture2D object is created using the MIPmap Mode `MULTI_LEVEL_MIPMAP` and the dimension of the base image. (lines 10 and 11). The base level is level 0. Then the level number is recorded and the base image set as the image for level 0 (lines 12 and 13).

The loop interates until the size of the image is 1 pixel by 1 pixel (line 15). The level number is incremented for each iteration (line 16) and the dimension of the image is calculated (lines 18 and 19). The appropriately scaled image is gotten from the TextureLoader (line 21) and set for the current level in the Texture2D object (line 22).

When creating a multiple level texture map be sure to set a multiple level filters as is done on lines 25 and 26 of Code Fragment 7-8. (Section 7.7.1 presents more information on filter choices.) The default filter settings disable multiple level texturing.

Creating the images by hand allows for superior image quality and/or special effects. The generated images are produced by filtering the base image.

```
1. Appearance appear = new Appearance();
2.
3.       String filename = "stripe.gif";          // filename for level 0
4.       NewTextureLoader loader = new NewTextureLoader(filename);
5.       ImageComponent2D image = loader.getImage();
6.
7.       imageWidth = image.getWidth();
8.       imageHeight = image.getHeight();
9.
10.      Texture2D texture = new Texture2D(Texture.MULTI_LEVEL_MIPMAP,
11.                                        Texture.RGBA,imageWidth, imageHeight);
12.      imageLevel = 0;
13.      texture.setImage(imageLevel, image);
14.
15.      while (imageWidth > 1 || imageHeight > 1){ // loop until size: 1x1
16.          imageLevel++;                          // compute this level
17.
18.          if (imageWidth > 1) imageWidth /= 2;  // adjust width as necess.
19.          if (imageHeight > 1) imageHeight /= 2;// adjust height as necess.
20.          filename = "stripe"+imageWidth+".gif";// file to load
21.
22.          loader = new NewTextureLoader(filename);
23.          image = loader.getImage();
24.
25.          texture.setImage(imageLevel, image);
26.      }
27.
28.      texture.setMagFilter(Texture.BASE_LEVEL_POINT);
29.      texture.setMinFilter(Texture.MULTI_LEVEL_POINT);
30.
31.      appear.setTexture(texture);
```

Code Fragment 7-9 Multiple Levels of Texture Loaded from Individual Image Files.

Section 0 presents the API for the TextureLoader and NewTextureLoader classes.

7.6.3 Multiple Levels of Texture Minification Filters

In addition to the two base level filter options, there are two multiple level filter options for the minification filter setting. These additional settings are MIPMAP_POINT, and MIPMAP_LINEAR. As with the other filter settings, the point filter is likely to be faster but yield images of lower quality as compared to the linear filter.

Remember, when using a multiple level texture, you must select one of the multiple level filters for the minification filter to utilize the levels other than the base level. These additional filter settings do not apply to the magnification filter settings since magnification of the texture would only be done at the base level. Consult section 7.7.1 for further filter information.

7.6.4 Mipmap Mode

The MIPmap Mode of the Texture class is really a choice between multiple levels of texture and a single level texture, called base level texturing. The two settings are BASE_LEVEL and MULTI_LEVEL_MIPMAP. Of course, for multiple levels of texture the latter setting is used.

7.7 Texture, Texture2D, and Texture3D API

Many of the preceding sections present some portion of the Texture, Texture2D, or Texture3D classes. Since these classes are described over many sections, the API for these classes is presented in this section.

Texture is the base class for Texture2D and Texture3D. The Texture class provides the majority of the interface for the Texture2D and Texture3D classes including multi level texturing. Table 7-3 presents a summary of the features of these three classes. For each texturing option the table lists the class which provides the interface, the set-Method for changing the setting, the default value, and sections of the tutorial which discuss the feature.

Table 7-3 Directory of Texture Features

Feature/Setting	Class	set-Methods	Default	Sections
Texture Image	Texture	`setImage()`	null	7.2
Image Format	Texture	(see constructors)	none	7.2
Mipmap Mode	Texture	`setMipMapMode()`	BASE_LEVEL	7.6
Minification Filter	Texture	`setMinFilter()`	BASE_LEVEL_POINT	7.2.5, 7.6.3, 7.7.1
Magnification Filter	Texture	`setMagFilter()`	BASE_LEVEL_POINT	7.2.5, 7.6.3, 7.7.1

Boundary Modes	Texture
	Texture
	Texture3D
		`setBoundaryModeS()`
		`setBoundaryModeT()`
		`setBoundaryModeR()`
			WRAP
			WRAP
			WRAP
				7.2.5

BoundaryColor	Texture	`setBoundaryColor()`	black	7.2.5

7.7.1 Minification and Magnification Filters

Sections 7.2.5 and 7.6.3 both discuss texture filters. Since neither of these sections discuss texture filters in detail, this section presents texture filters in a little more generality.

As previously discussed there are separate filter settings for minification and magnification. The magnification choices are: `BASE_LEVEL_POINT`, `BASE_LEVEL_LINEAR`, `FASTEST`, or `NICEST`. The filter will be `BASE_LEVEL_POINT` when `FASTEST` is specified and `BASE_LEVEL_LINEAR` when NICEST is specified.

The minification choices are: `BASE_LEVEL_POINT`, `BASE_LEVEL_LINEAR`, `MULTI_LEVEL_POINT`, `MULTI_LEVEL_LINEAR`, `FASTEST`, or `NICEST`. The base level filter choices can be used for single or multiple level textures. The actual filters used when FASTEST or NICEST is specified is implementation dependant and may not choose a multi level filter for a multiple level texture.

7.7.2 Texture API

Now that all texture features have been presented, the Texture API is presented. The Texture class is abstract so there is no Texture class constructor reference block. The next reference block lists the fields of the Texture class which are used as settings. The method and capabilities reference blocks follow.

Texture Field Summary

The Texture object is a component object of an Appearance object that defines the texture properties used when texture mapping is enabled. Texture object is an abstract class and all texture objects must be created as either a Texture2D object or a Texture3D object.

Format Constants

ALPHA	Specifies Texture contains only Alpha values.
INTENSITY	Specifies Texture contains only Intensity values.
LUMINANCE	Specifies Texture contains only luminance values.
LUMINANCE_ALPHA	Specifies Texture contains Luminance and Alpha values.
RGB	Specifies Texture contains Red, Green and Blue color values.
RGBA	Specifies Texture contains Red, Green, Blue color values and Alpha value.

MIP Map Mode Constants

BASE_LEVEL	Indicates that Texture object only has one level.
MULTI_LEVEL_MIPMAP	Texture object has multiple images- one for each mipmap level

Filter Constants

BASE_LEVEL_LINEAR	Performs bilinear interpolation on the four nearest texels in level 0 texture map.
BASE_LEVEL_POINT	Selects the nearest texel in level 0 texture map.
MULTI_LEVEL_LINEAR	Performs tri-linear interpolation between four texels each from two nearest mipmap levels.
MULTI_LEVEL_POINT	Selects the nearest texel in the nearest mipmap.

Boundary Mode Constants

CLAMP	Clamps texture coordinates to be in the range [0, 1].
WRAP	Repeats the texture by wrapping texture coordinates that are outside the range [0,1].

Perspective Correction Mode Constants

FASTEST	Uses the fastest available method for processing geometry.
NICEST	Uses the nicest available method for processing geometry.

Texture Method Summary

The Texture object is a component object of an Appearance object that defines the texture properties used when texture mapping is enabled. Texture object is an abstract class and all texture objects must be created as either a Texture2D object or a Texture3D object.

`ImageComponent getImage(int level)`
Gets a specified mipmap level.

`void setBoundaryColor(Color4f boundaryColor)`
`void setBoundaryColor(float r, float g, float b, float a)`
Sets the texture boundary color for this texture object.

`void setBoundaryModeS(int boundaryModeS)`
Sets the boundary mode for the S coordinate in this texture object.

`void setBoundaryModeT(int boundaryModeT)`
Sets the boundary mode for the T coordinate in this texture object.

`void setEnable(boolean state)`
Enables or disables texture mapping for this appearance component object.

`void setImage(int level, ImageComponent image)`
Sets a specified mipmap level.

`void setMagFilter(int magFilter)`
Sets the magnification filter function.

`void setMinFilter(int minFilter)`
Sets the minification filter function.

`void setMipMapMode(int mipmapMode)`
Sets mipmap mode for texture mapping for this texture object.

`void setImages(ImageComponent[] images)` **`<new in 1.2>`**
Sets the array of images for all mipmap levels.

Parameters:
images - array of ImageComponent objects containing the texture images for all mipmap levels

`int getFormat()` **`<new in 1.2>`**
Retrieves the format of this Texture object.

`public int getWidth()` **`<new in 1.2>`**
Retrieves the width of this Texture object.

`public int getHeight()` **`<new in 1.2>`**
Retrieves the height of this Texture object.

`int numMipMapLevels()` **`<new in 1.2>`**
Retrieves the number of mipmap levels needed for this Texture object.

Returns: log2(max(width,height))+1 if mipMapMode is MULTI_LEVEL_MIPMAP; otherwise it returns 1.

Texture Capabilities Summary

ALLOW_BOUNDARY_COLOR_READ	allows reading its boundary color information.	
ALLOW_BOUNDARY_MODE_READ	allows reading its boundary mode information.	
ALLOW_ENABLE_READ | WRITE	allows reading its enable flag.	
ALLOW_FILTER_READ	allows reading its filter information.	
ALLOW_IMAGE_READ	allows reading its image component information.	
ALLOW_MIPMAP_MODE_READ	allows reading its mipmap mode information.	
ALLOW_IMAGE_WRITE	allows writing its image component information	**<new in 1.2>**
ALLOW_FORMAT_READ	allows reading its format information	**<new in 1.2>**
ALLOW_SIZE_READ	allows reading its size information (e.g., width, height)	**<new in 1.2>**

7.7.3 Texture2D API

Texture2D is a concrete extension of the abstract Texture class. Texture2D provides only one constructor of interest. All of the methods used with Texture2D objects are methods of Texture. The following reference block presents the Texture2D constructor.

Texture2D Constructor Summary

Texture2D is a subclass of Texture class. It extends Texture class by adding a constructor.

Texture2D(int mipmapMode, int format, int width, int height)
Constructs an empty Texture2D object with specified mipmapMode, format, width, and height. Image at level 0 must be set by the application using 'setImage' method. If mipmapMode is set to MULTI_LEVEL_MIPMAP, images for ALL levels must be set.

Parameters:

mipmapMode - type of mipmap for this Texture: One of BASE_LEVEL, MULTI_LEVEL_MIPMAP.
format - data format of Textures saved in this object. One of INTENSITY, LUMINANCE, ALPHA, LUMINANCE_ALPHA, RGB, RGBA.
width - width of image at level 0. Must be power of 2.
height - height of image at level 0. Must be power of 2.

7.7.4 Texture3D API

Texture3D is a concrete extension of the abstract Texture class. Texture3D provides only one constructor and a method to set the boundary mode in the r dimension. All other methods used with Texture3D objects are methods of Texture. The following two reference blocks present the Texture3D constructor and the Texture 3D methods.

Texture3D Constructor Summary

Texture3D is a subclass of Texture class. It extends Texture class by adding a third coordinate, a constructor, and a mutator method for setting a 3D texture image.

`Texture3D(int mipmapMode, int format, int width, int height, int depth)`
Constructs an empty Texture3D object with specified mipmapMode, format, width, height, and depth. Image at level 0 must be set by the application using 'setImage' method. If mipmapMode is set to MULTI_LEVEL_MIPMAP, images for ALL levels must be set.
Parameters:
mipmapMode - type of mipmap for this Texture: One of BASE_LEVEL, MULTI_LEVEL_MIPMAP.
format - data format of Textures saved in this object. One of INTENSITY, LUMINANCE, ALPHA, LUMINANCE_ALPHA, RGB, RGBA.
width - width of image at level 0. Must be power of 2.
height - height of image at level 0. Must be power of 2.
depth - depth of image at level 0. Must be power of 2.

Texture3D Method Summary

`void setBoundaryModeR(int boundaryModeR)`
Sets the boundary mode for the R coordinate in this texture object.
boundaryModeR - the boundary mode for the R coordinate, one of: CLAMP or WRAP.

`int getDepth()` **`<new in 1.2>`**
Retrieves the depth of this Texture3D object.

7.8 Multitexture **`<new in 1.2>`**

Before tackling mutltitexuring, you should be familiar with how texturing is accomplished in Java 3D. If you skipped the previous 7 sections of this chapter, here is an outline of what you missed.

The previous sections explain how to combine Texture, TextureAttributes, and TextureCoordGeneration objects to texture visual objects. Not all texture applications use objects of all classes, but some do. Table 7-1 lists these texturing classes.

Table 7-4 Review of Fundamental Texture Related Classes

Class	Feature/Setting	Sections
Texture	Texture Image, Image Format, MIPmap Mode, Minification and Magnification Filters, and Boundary Modes	7.2, 7.6, 7.7
TextureAttributes	TextureMode, Texture Transform, Blend Color, Perspective Correction	7.4
TextureCoordGeneration	automatic generation of texture coordinates (optional)	7.5

Also recall that Texture, TextureAttributes, and TextureCoordGeneration objects are associated with a visual object through an Appearance object. Figure 7-21 illustrates an appearance bundle with a complete complement of texturing objects.

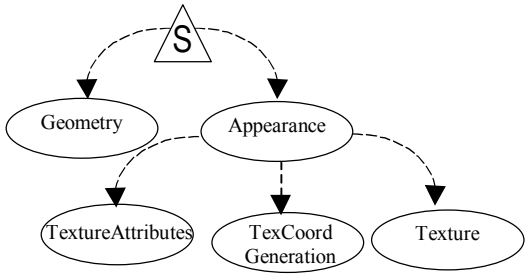


Figure 7-21 A Visual Object with Texture, TextureAttributes, and TexCoodGeneration objects.

With the review now behind us (feel free to refer back to previous sections as necessary) lets move on to multitexturing

7.8.1 Multitexture, Texture Units, and TextureUnitStates `<new in 1.2>`

Multitexturing allows the programmer to apply multiple textures to a single visual object. As we have just seen, texturing may be specified by the combination of Texture, TextureAttributes, and TexCoordGeneration objects. In addition, the GeometryArray has texture coordinates specified on a per vertex basis. To achieve multitexturing, multiple Texture, TextureAttributes, and TexCoordGeneration objects are needed. This is where the texture unit and the TextureUnitState class comes in.

With the exception of the texture coordinates, a **texture unit** completely defines texturing of a visual object. The texture coordinates are defined with the object's geometry. That is, a texture unit references Texture, TextureAttributes, and TexCoordGeneration objects. In some graphics cards, there are hardware implementations of texture units, otherwise, software handles its functions.

Figure 7-22 shows a visual object set for texturing by two texture units. The Appearance node component references two TextureUnitState objects. Each of the TextureUnitState objects has TextureAttributes, TexCoordGeneration, and Texture objects. Although not done in this example, TextureAttributes, TexCoordGeneration, and/or Texture objects can be shared by TextureUnitState objects.

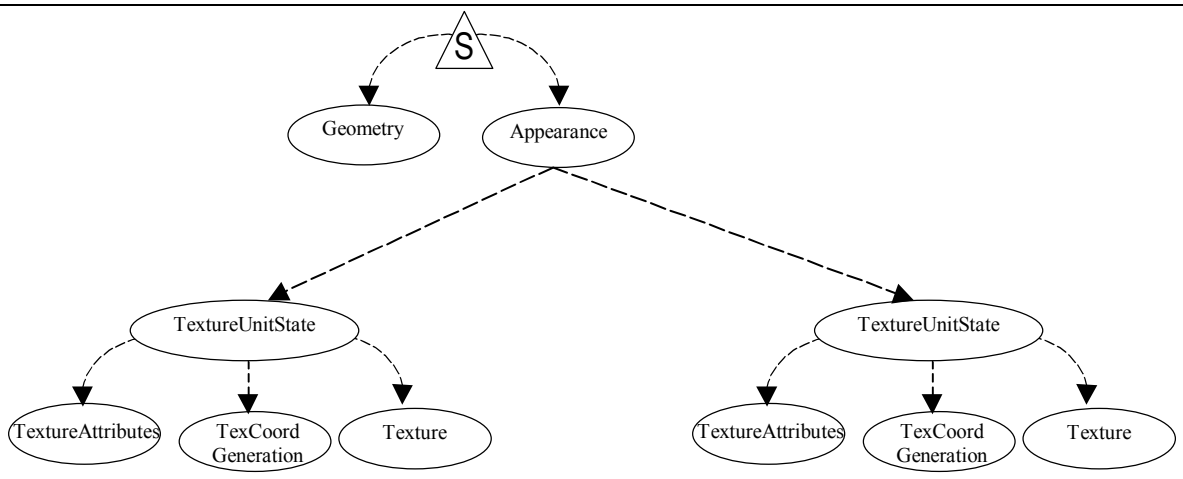


Figure 7-22 Appearance Bundle with multiple TextureUnitState entries.

Two things should be obvious from this section. First, multitexturing is not difficult. Multitexturing is simply texturing multiple times. Second, to accomplish multitexturing requires knowledge of

TextureUnitState and Appearance classes. The relevant API features of TextureUnitState and Appearance appear in sections 7.8.2 and 7.8.3, respectively.

As mentioned on page 7-7, multitexturing may require multiple texture coordinate sets. This final piece of the multitexture puzzle is explained in section 7.8.4.

7.8.2 TextureUnitState API <new in 1.2>

Sections 7.1 to 7.7 texture visual objects without obviously using a texture unit. In this section the texture unit is introduced in explaining multitexturing. With the discussion of texture units, you may have expected to find a TextureUnit class in the Java 3D API. This is not the case. Think of a texture unit as some hardware in your computer for handling texturing. Since you have texture unit hardware, you need a way to control it – the way to control the hardware is by changing its state, thus, the TextureUnitState class.

The TextureUnitState object defines all texture mapping state for a single texture unit. An appearance object contains an array of texture unit state objects to define the state for multiple texture mapping units. The texture unit state consists of a Texture object, a TexturingAtttributes object, and a TexCoordGeneration object. Previous sections describe the functions of these classes. Refer to Table 7-4 (page 7-35) for a summary of the function of these classes.

TextureUnitState Constructor Summary <new in 1.2>

The TextureUnitState object defines all texture mapping state for a single texture unit. An appearance object contains an array of texture unit state objects to define the state for multiple texture mapping units. The texture unit state consists of the following: a Texture object, a TextureAttributes object, and a TexCoordGeneration object.

`TextureUnitState()` <new in 1.2>
Constructs a TextureUnitState component object using defaults for all state variables.

`TextureUnitState(Texture texture, TextureAttributes textureAttributes, TexCoordGeneration texCoordGeneration)` <new in 1.2>
Constructs a TextureUnitState component object using the specified component objects.

The methods of TextureUnitState allow the setting (and getting) of the Texture, TextureAttributes, and TexCoordGeneration objects referred to by a TextureUnitState object.

TextureUnitState Method Summary

`void set(Texture texture, TextureAttributes textureAttributes, TexCoordGeneration texCoordGeneration)` <new in 1.2>
Sets the texture, texture attributes, and texture coordinate generation components in this TextureUnitState object to the specified component objects.

`void setTexCoordGeneration(TexCoordGeneration texCoordGeneration)` <new in 1.2>
Sets the texCoordGeneration object to the specified object.

`void setTexture(Texture texture)` <new in 1.2>
Sets the texture object to the specified object.

`void setTextureAttributes(TextureAttributes textureAttributes)` <new in 1.2>
Sets the textureAttributes object to the specified object.

Without setting the appropriate capabilities the texture node components objects referred to by a live or compiled TextureUnitState object can neither be read nor written. The following reference block lists the related capabilities.

TextureUnitState Field Summary

`ALLOW_STATE_READ | WRITE` **`<new in 1.2>`**
Specifies that this TextureUnitState object allows reading (writing) its texture, texture attribute, or texture coordinate generation component information.

7.8.3 Appearance API for Multitexture `<new in 1.2>`

As previously noted, Appearance objects refer to multiple TextureUnitState objects to achieve multitexturing. Appearance class reference blocks appear in nearly every chapter of this tutorial. This section and the accompanying reference block explains the Appearance API related to multitexturing.

In the sections prior to 7.8 (this section), an Appearance object could refer to Texture, TextureAttributes, and TextureCoordGeneration objects directly (as shown in Figure 7-21, page 7-36). To achieve multitexturing, an Appearance object indirectly refers to Texture, TextureAttributes, and TextureCoordGeneration objects through TextureUnitState objects (as shown in Figure 7-22, page 7-36). It is not legal for an Appearance object to do both. Either use the texture objects directly, or use them indirectly (through a TextureUnitState object). To do otherwise will cause an exception to be thrown at runtime.

The Appearance object may have an array of references to TextureUnitState objects. The array is created when the first method is called. The TextureStateUnit array internal to the Appearance node component will have references to any TextureUnitState objects referred to by the parameter array. The second method can be used to change the TextureUnitState objects referred to by an Appearance object.

Appearance multitexture related methods

`void setTextureUnitState(TextureUnitState[] stateArray)` **`<new in 1.2>`**
Sets the texture unit state array for this appearance object to the specified array. A shallow copy of the array of references to the TextureUnitState objects is made. If the specified array is null or if the length of the array is 0, multi-texture is disabled. Within the array, a null TextureUnitState element disables the corresponding texture unit.
stateArray - array of TextureUnitState objects that specify the desired texture state for each unit. The length of this array specifies the maximum number of texture units used by this Appearance object. The texture units are numbered from 0 through stateArray.length-1.

`void setTextureUnitState(int index, TextureUnitState state)` **`<new in 1.2>`**
Sets the texture unit state object at the specified index within the texture unit state array to the specified object. If the specified object is null, the corresponding texture unit is disabled. The index must be within the range [0, stateArray.length-1].
index - the array index of the object to be set
state - new texture unit state object

`int getTextureUnitCount()` **`<new in 1.2>`**
Retrieves the length of the texture unit state array from this appearance object. The length of this array specifies the maximum number of texture units used by this Appearance object. If the array is null, a count of 0 is returned.

Without setting the appropriate capabilities TextureStateUnit objects of a live or compiled Appearance object can neither be read nor written. The following reference block lists the multitexture-related Appearance capabilities.

Appearance Capabilities (partial list)

ALLOW_TEXTURE_UNIT_STATE_READ | WRITE **<new in 1.2>**
Specifies that this Appearance object allows writing its texture unit state component information.

7.8.4 GeometryArray API for Multitexture **<new in 1.2>**

A texture unit provides all texture information except the texture coordinates of the geometry. As texture coordinates are defined per vertex, it is necessary that this information is stored with the geometry. While texture units can share the same set of texture coordinates, with multitexture it may be desirable to have different texture coordinates for each texture unit. For this reason Java 3D API v1.2 includes the ability to store multiple sets of texture coordinates in a GeometryArray.

Some setTextureCoordinates() methods are presented in the reference block on page 7-6. There are more setTextureCoordinate() methods than listed in that reference block, refer to the Java 3D API Specification for more information.

With the possibility of having multiple texture units and multiple texture coordinate sets there is a need to create associations among them. For this purpose a GeometryArray object may have a map, which is an array whose contents determines which set of texture coordinates is used by which texture unit.

The array, `texCoordSetMap`, contains integers. The array is indexed by texture unit number for each texture unit in the associated Appearance object. The values in the array specify the texture coordinate set within this GeometryArray object that maps to the corresponding texture unit. For example, the integer in the first location of the array, `texCoordSetMap[0]`, contains the number of the texture coordinate set to be used by the first texture unit.

All elements within the array must be less than `texCoordSetCount`, the number of texture coordinate sets defined in the geometry object. A negative value in the `texCoordSetMap[]` array specifies that no texture coordinate set is to be used for the texture unit corresponding to the index. If there are more texture units in any associated Appearance object than elements in the mapping array, the extra elements are assumed to be -1. The same texture coordinate set may be used for more than one texture unit. The following table shows an example usage of the texCoordSetMap.

Table 7-5 Example Usage of the texCoordSetMap

Index (texture unit)	Element Value (texture coordinate set)	Description
0	1	Use texture coord set 1 for texture unit 0
1	-1	Use no texture coord set for texture unit 1
2	0	Use texture coord set 0 for texture unit 2
3	1	Reuse texture coord set 1 for texture unit 3

Each texture unit in every associated Appearance must have a valid source of texture coordinates: either a non-negative texture coordinate set must be specified in the mapping array or texture coordinate generation must be enabled. Texture coordinate generation will take precedence for those texture units for which a texture coordinate set is specified and texture coordinate generation is enabled.

The following reference block shows yet another constructor of the GeometryArray class. This particular constructor has parameters for specifying the number of texture coordinate sets and a texture coordinate set mapping array.

GeometryArray constructor (partial list)

```
GeometryArray(int vertexCount, int vertexFormat,                    <new in 1.2>
              int texCoordSetCount, int[] texCoordSetMap)
```

Constructs an empty GeometryArray object with the specified number of vertices, vertex format, number of texture coordinate sets, and texture coordinate mapping array. Defaults are used for all other parameters.

vertexCount - the number of vertex elements in this GeometryArray
vertexFormat - a mask indicating which components are present in each vertex
texCoordSetCount - the number of texture coordinate sets in this GeometryArray object
texCoordSetMap - an array that maps texture coordinate sets to texture units

The following reference block lists texture related methods of the GeometryArray class. Additional methods of the GeometryArray class are found in the reference block on page 7-6, in various other chapters of the tutorial, and in the Java 3D API Specification.

GeometryArray texture related methods (partial list)

```
int getTexCoordSetCount()                                            <new in 1.2>
```

Retrieves the number of texture coordinate sets in this GeometryArray object.

```
int getTexCoordSetMapLength()                                        <new in 1.2>
```

Retrieves the length of the texture coordinate set mapping array of this GeometryArray object.

```
void getTexCoordSetMap(int[] texCoordSetMap)                         <new in 1.2>
```

Retrieves the texture coordinate set mapping array from this GeometryArray object.

texCoordSetMap - an array that will receive a copy of the texture coordinate set mapping array. The array must be large enough to hold all entries of the texture coordinate set mapping array.

7.9 TextureLoader and NewTextureLoader API

This section lists the reference blocks for the TextureLoader and NewTextureLoader classes. The texture loader is explained in some detail in Step 2a of the simple texture recipe as defined in Section 7.2.1 on page 7-4. The texture loader is used in the all the example programs of this chapter. Of particular interest are the examples on using the texture loader for the MIPmap applications (see Section 7.6).

The NewTextureLoader class extends the TextureLoader class providing an easier to use texture loader utility – one that does not require a awt.component image observer for each constructor.

7.9.1 TextureLoader API

The following reference block lists the field constants used in creating TextureLoader objects.

TextureLoader Field Summary

GENERATE_MIPMAP
MIPmaps are generated for all levels

BY_REFERENCE **<new in 1.2>**
the ImageComponent2D will access the image data by reference

Y_UP **<new in 1.2>**
the ImageComponent2D will have a y-orientation of y up, meaning the origin of the image is the lower left

The following reference block lists some constructors for the TextureLoader class. There are a number of constructors not listed in the constructors reference block which allow loading texture images from other sources. Consult the Java 3D API Specification for a complete list of constructors.

TextureLoader Constructor Summary (partial list)

extends: `java.lang.Object`
package: `com.sun.j3d.utils.image`

This class is used for loading a texture from an Image or BufferedImage. Methods are provided to retrieve the Texture object and the associated ImageComponent object or a scaled version of the ImageComponent object. Default format is RGBA.

Other legal formats are: RGBA, RGBA4, RGB5_A1, RGB, RGB4, RGB5, R3_G3_B2, LUM8_ALPHA8, LUM4_ALPHA4, LUMINANCE and ALPHA

TextureLoader(java.lang.String fname, java.awt.Component observer)
TextureLoader(java.lang.String fname, int flags, java.awt.Component observer)
Contructs a TextureLoader object using the specified file, option flags and default format RGBA

TextureLoader(java.net.URL url, java.awt.Component observer)
TextureLoader(java.net.URL url, int flags, java.awt.Component observer)
Contructs a TextureLoader object using the specified URL, option flags and default format RGBA

The following reference block lists the methods of the TextureLoader class.

TextureLoader Method Summary

ImageComponent2D getImage()
Returns the associated ImageComponent2D object

ImageComponent2D getScaledImage(float xScale, float yScale)
Returns the scaled ImageComponent2D object

ImageComponent2D getScaledImage(int width, int height)
Returns the scaled ImageComponent2D object

Texture getTexture()
Returns the associated Texture object

7.9.2 NewTextureLoaderAPI

The reason to use the NewTexureLoader is to avoid needing an image observer to construct a texture loader. The following reference block lists some constructors for the NewTextureLoader class.

NewTextureLoader has the same constructors as TextureLoader except none require an awt component to server as the image observer. `NewTextureLoader.html`, a javadoc file included in the examples jar, gives complete documentation for the NewTextureLoader class.

NewTextureLoader Constructor Summary (partial list)

extends: `com.sun.j3d.utils.image.TextureLoader`

This class is used for loading a texture from an file or URL. This class differs from com.sun.j3d.util.image.TextureLoader only in the absence of an image observer in the constructor and the method to set a single image observer for all subsequent uses. All TextureLoader class constructors requiring an image observer have a corresponding NewTextureLoader constructor without an image observer awt.component.

`NewTextureLoader(java.lang.String fname)`
`NewTextureLoader(java.lang.String fname, int flags)`
Contructs a TextureLoader object using the specified file, option flags and default format RGBA

`TextureLoader(java.net.URL url)`
`TextureLoader(java.net.URL url, int flags)`
Contructs a TextureLoader object using the specified URL, option flags and default format RGBA

The following reference block lists the two methods of the defined in the NewTextureLoader class. All other methods are defined by the TextureLoader class. To use a NewTextureLoader object an image observer must be set first. This is normally done when the Canvas3D object is created.

NewTextureLoader Method Summary (partial list)

`java.awt.component getImageObserver()`
Returns the awt.component object used as the image observer for NewTextureLoader objects.

`void setImageObserver(java.awt.component imageObserver)`
Sets a awt.component object as the object to use as an image observer in subsequent constructions of NewTextureLoader objects.

7.10 Chapter Summary

This chapter presents the Java 3D features for rendering objects with textures. Texturing is primarily a function of the Texture class and its two subclasses, Texture2D and Texture3D. Section 7.1 provides motivation for texturing and introduces some texturing terminology. Section 7.2 presents the basics of the Texture, Texture2D, and Texture3D classes. Section 7.2 also includes a simple texturing recipe and a simple texturing application. Section 7.3 demonstrates the use of Texture2D in some other applications. Section 7.4 presents the TextureAttributes class. TextureAttributes objects are used to customize certain aspects of texturing applications. Section 7.5 presents the TexCoordGeneration class. Objects of this class are used to automatically generate texture coordinates for visual objects. Section 7.6 explains multi level texturing. Section 7.7 presents the API for the Texture, Texture2D and Texture3D classes. Along with the reference blocks for these classes is a 'directory' of texturing features (see Table 7-3 on page 7-31).

Java 3D API version 1.2 introduces multitextures. Section 7.8 explains all about multitexture. Section 7.9 presents the API for the TextureLoader class. TextureLoader objects are used in the example programs throughout the chapter and briefly discussed in Sections 7.2 and 7.6. The chapter concludes with the traditional set of "self test" questions.

7.11 Self Test

1. What happens if a texture coordinate assignment is not made for a vertex in some geometry? Is there an exception? A warning? Does it render? If it renders, what is the result?
2. How can a single image of a texture (not repeated) be mapped onto a visual object and have the image surrounded by a single solid color? Assume the texture image is the typical non-solid-color image. How could the surrounding color be assigned or modified at runtime?
3. How can multiple textures be applied to a single visual object? How can you apply different textures to the opposite sides of the same visual object? How can you apply different textures to the lines and the surfaces of a polygon?
4. How would you animate a shadow that moves across a stationary visual object as the shadow moves?
5. How would you animate a shadow that moves across a visual object as the object passes through a stationary shadow?