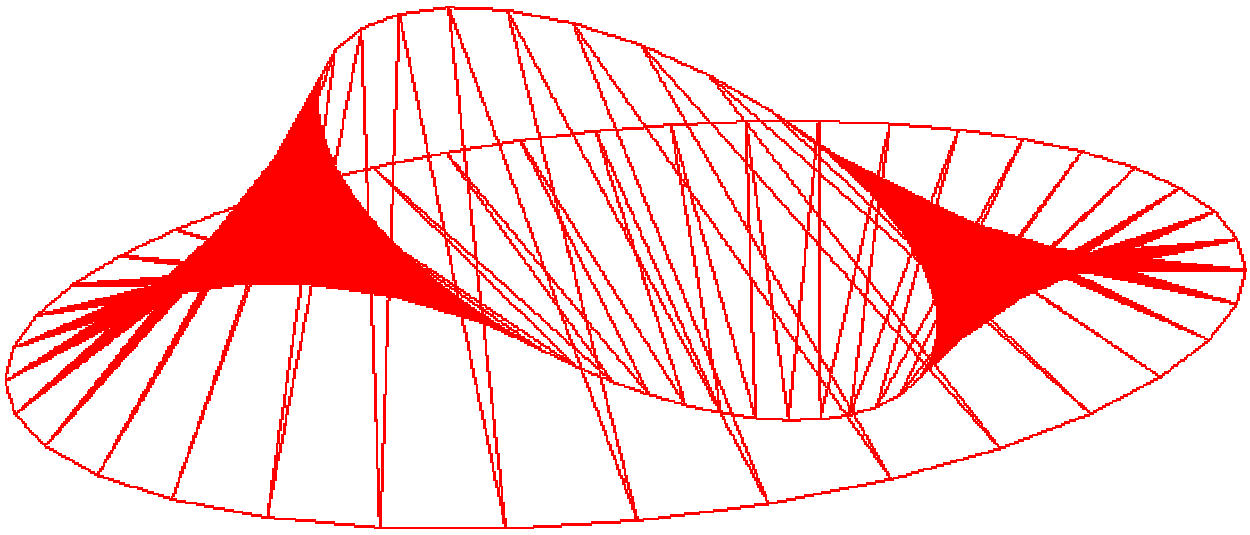
Getting Started with the Java 3D™ API

Chapter 2
Creating Geometry



Dennis J Bouvier

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

Chapter 2:

List of Figures

List of Tables

List of Code Fragments

List of Reference Blocks

Preface to Chapter 2

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`

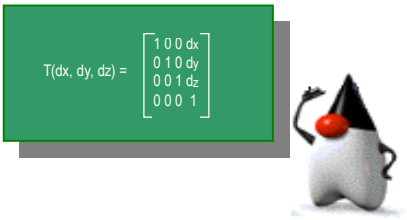
New for Java 3D API version 1.2

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 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.

In addition to the API 1.2 features added, Chapter 2 includes more of the Java 3D API. Some of the additional material is new to the tutorial (e.g., BoundingBox and BoundingPolytope), some of it has been moved to Chapter 2 from other chapters (BoundingLeaf, and the discussion of scope). The changes have been made in an effort to improve coverage of the API and the organization of the tutorial.

(intentional almost blank page)

CHAPTER 2
Creating Geometry



Chapter Objectives

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

- Use geometric primitive utility classes
- Write classes to define visual objects
- Specify geometry using core classes
- Specify appearance and alternative appearance attributes for visual objects
- Use geometry by reference
- Use clip and model clip nodes to change what is viewed

Chapter 1 explores the basic concepts of building a Java 3D virtual universe, concentrating on specifying transforms and simple behaviors. The HelloJava3D examples in Chapter 1 use the ColorCube class for the only visual object. With ColorCube, the programmer doesn't specify shape or color. The ColorCube class is easy to use but can not be used to create other visual objects.

There are three major ways to create new geometric content. One way uses the geometric utility classes for box, cone, cylinder, and sphere. Another way is for the programmer to specify the vertex coordinates for points, line segments, and/or polygonal surfaces. A third way is to use a geometry loader. This chapter demonstrates creating geometric content the first two ways.

The focus of this chapter is the creation of geometric content, that is, the shape of visual objects. A few topics related to geometry are also covered, including math classes and appearance. Before describing how to create geometric content, more information on the virtual universe coordinate system is presented in section 2.1.

2.1 Virtual World Coordinate System

As discussed in Chapter 1, an instance of VirtualUniverse class serves as the root of the scene graph in all Java 3D programs. The term **virtual universe** commonly refers to the three dimensional virtual space

Java 3D objects populate. Each Locale object in the virtual universe establishes a virtual world Cartesian coordinate system.

A Locale object serves as the reference point for visual objects in a virtual universe. With one Locale in a SimpleUniverse, there is one coordinate system in the virtual universe.

The coordinate system of the Java 3D virtual universe is right-handed. The x-axis is positive to the right, y-axis is positive up, and z-axis is positive toward the viewer, with all units in meters. Figure 2-1 shows the orientation with respect to the viewer in a SimpleUniverse.

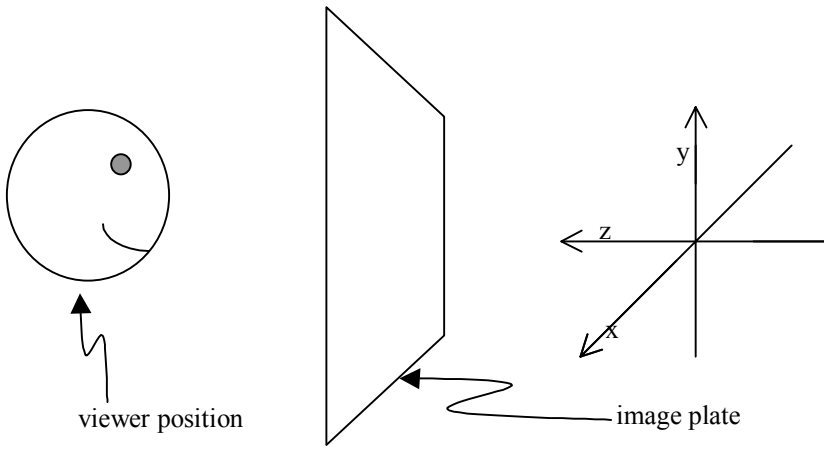


Figure 2-1 Orientation of Axis in Virtual World

2.2 Visual Object Definition Basics

Section 2.2.1 presents the Shape3D class. A general discussion of the NodeComponent class follows in section 2.2.2. After discussing geometry primitives defined in the utility package, the rest of the chapter covers Geometry and Appearance node components.

2.2.1 An Instance of Shape3D Defines a Visual Object

A Shape3D scene graph node defines a visual object[1]. Shape3D is one of the subclasses of Leaf class; therefore, Shape3D objects can only be leaves in the scene graph. The Shape3D object does not contain information about the shape or color of a visual object. This information is stored in the NodeComponent objects referred to by the Shape3D object. A Shape3D object can refer to one Geometry node component and one Appearance node component.[2]

[1] Shape3D objects define the most common visual objects of a virtual universe, but there are other ways.

[2] In Java3D API 1.2 Shape3D objects can refer to multiple Geometry node components. Section 2.8.1 explains this API feature.

In the HelloJava3D scene graphs in Chapter 1, the generic object symbol (rectangle) was used to represent the ColorCube object. The simple scene graph in Figure 2-2 shows a visual object represented as a Shape3D leaf (triangle) and two NodeComponents (ovals) instead of the generic rectangle[3].

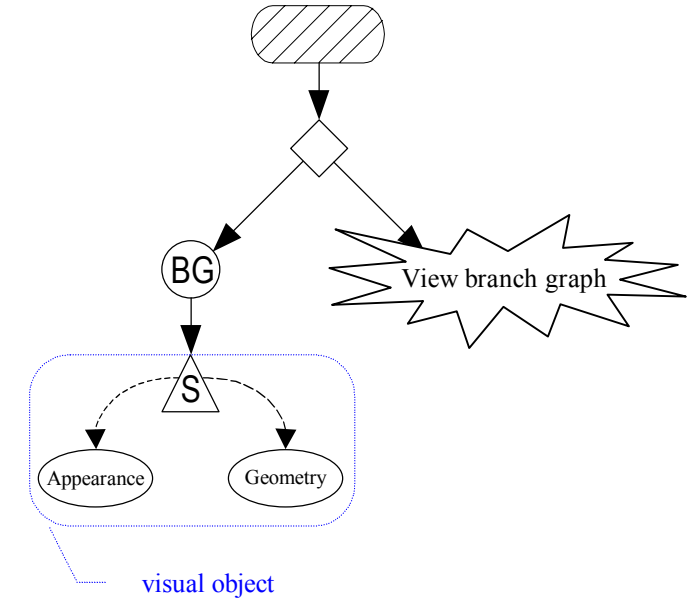


Figure 2-2 A Shape3D Object Defines a Visual Object in a Scene Graph.

A visual object can be defined using just a Shape3D object and a Geometry node component. Optionally, the Shape3D object refers to an Appearance node component as well. The constructors for Shape3D (presented in the next reference block) show that a Shape3D object can be created without node component references, with just a Geometry node component reference, or with references to both types of node components.

Shape3D Constructors

`Shape3D()`
Constructs and initializes a Shape3D object with null Geometry and Appearance node components.

`Shape3D(Geometry geometry)`
Constructs and initializes a Shape3D object with the specified geometry and a null appearance component.

`Shape3D(Geometry geometry, Appearance appearance)`
Constructs and initializes a Shape3D object with the specified geometry and appearance components.

As long as the Shape3D object is not live and not compiled, the node component references can be changed with the methods shown in the next reference block. These methods can be used on live or compiled Shape3D objects if the capabilities to do so are set first. Another reference block below lists

[3] This scene graph is not correct for a ColorCube object. ColorCube does not use an Appearance NodeComponent. This is an example of a typical visual object.

the Shape3D capabilities. Be sure to read the "Reading Reference Blocks" section. It applies to many future reference blocks.

Shape3D Methods (partial list)

A Shape3D object references Geometry and/or Appearance NodeComponent objects. Along with the set-methods shown here, there are complementary get-methods.

```
void setGeometry(Geometry geometry)
void setAppearance(Appearance appearance)
```

Reading Reference Blocks

The reference blocks in this tutorial do not list all of the constructors, methods, and capabilities for each Java 3D API class. For example, the Shape3D methods reference block (above) does not list all the methods of the Shape3D class. Two of the methods not listed are the "get-methods" that match the "set-methods" shown. That is, Shape3D has `getGeometry()` and `getAppearance()` methods. Each of these methods returns a reference to the appropriate NodeComponent.

Since many Java 3D API classes have many methods, not all are listed. The ones listed in the reference blocks in this tutorial are the ones that pertain to the tutorial topics. Also, many classes have get-methods that match set-methods. The get-methods are not listed in the reference blocks in this tutorial to reduce the length of the reference blocks.

The following reference block shows the capabilities of Shape3D objects. This reference block introduces a shorthand notation for listing capabilities. Each line in the reference block lists two capabilities instead of one. There is an ALLOW_GEOMETRY_READ and an ALLOW_GEOMETRY_WRITE capability in each Shape3D object. Quite often there are read and write pairs of capabilities. To reduce the size of the reference blocks, capability reference blocks list the matched read and write capability pairs together in the short hand notation.

Consult the API specification for the complete list of constructors, methods, and capabilities.

Shape3D Capabilities (partial list)

Shape3D objects inherit capabilities from SceneGraphObject, Node, and Leaf classes. They are not listed here. Refer to section 1.8.2 for more information on Capabilities.

```
ALLOW_GEOMETRY_READ | WRITE
ALLOW_APPEARANCE_READ | WRITE
ALLOW_COLLISION_BOUNDS_READ | WRITE
```

2.2.2 Node Components

NodeComponent objects contain the exact specification of the attributes of a visual object. Each of the several subclasses of NodeComponent defines certain visual attributes. Figure 2-3 shows part of the Java 3D API hierarchy containing the NodeComponent class and its descendants. Section 2.5 presents the Geometry NodeComponent. Section 2.6 presents the Appearance NodeComponent.

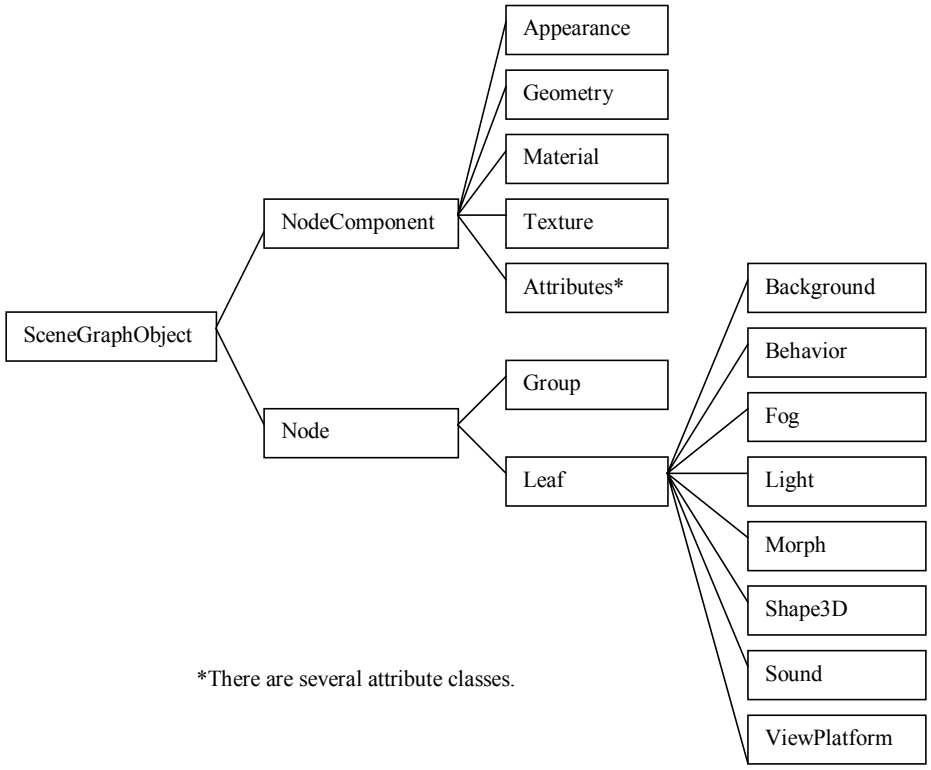


Figure 2-3 Partial Java 3D API Class Hierarchy Showing Subclasses of NodeComponent.

2.2.3 Defining Visual Object Classes

The same visual object will quite often appear many times in a single virtual universe. It makes sense to define a class to create the visual object instead of constructing each visual object from scratch. There are several ways to design a class to define a visual object.

Code Fragment 2-1 shows the skeleton code of VisualObject class as an example of one possible organization for a generic visual object class. The methods are empty in the code. The code of VisualObject does not appear in the examples distribution because is it not particularly useful as is.

```
1.     public class VisualObject extends Shape3D{
2.
3.         private Geometry voGeometry;
4.         private Appearance voAppearance;
5.
6.         // create Shape3D with geometry and appearance
7.         // the geometry is created in method createGeometry
8.         // the appearance is created in method createAppearance
9.         public VisualObject() {
10.
11.            voGeometry = createGeometry();
12.            voAppearance = createAppearance();
13.            this.setGeometry(voGeometry);
14.            this.setAppearance(voAppearance);
15.        }
16.
```

```
17.         private Geometry createGeometry() {
18.             // code to create default geometry of visual object
19.         }
20.
21.         private Appearance createAppearance () {
22.             // code to create default appearance of visual object
23.         }
24.
25.     } // end of class VisualObject
```

Code Fragment 2-1 Skeleton Code for a VisualObject Class

The organization of the VisualObject class in Code Fragment 2-1 is similar to the ColorCube utility class in that it extends a Shape3D object. The VisualObject class is a suggested starting point for defining custom content classes for use in scene graph construction. Each individual Java 3D programmer will almost certainly customize the VisualObject class for their own purposes. For a complete example of this class organization, read the source code for ColorCube class in the `com.sun.j3d.utils.geometry` package, which is available with the Java 3D API distribution.

Using Shape3D as a base for creating a visual object class makes it easy to use in a Java 3D program. The visual object class can be used as easily as the ColorCube class in the HelloJava3D examples from Chapter 1. The constructor can be called and the newly created object inserted as the child of some Group in one line of code. In the following example line of code, objRoot is an instance of Group. This code creates a VisualObject and adds it as a child of objRoot in the scene graph:

```
objRoot.addChild(new VisualObject());
```

The VisualObject constructor creates the VisualObject by creating a Shape3D object which references the NodeComponents created by the methods `createGeometry()` and `createAppearance()`. The method `createGeometry()` creates a Geometry NodeComponent to be used in the visual object. The method `createAppearance()` is responsible for creating the NodeComponent that defines the Appearance of the visual object.

Another possible organization for a visual object is to define a container class not derived from Java 3D API classes. In this design, the visual object class would contain a Group Node or a Shape3D as the root of the subgraph it defines. The class must define method(s) to return a reference to this root. This technique is a little more work, but may be easier to understand. Some program examples presented later in this chapter give examples of independent visual object class definitions.

A third possible organization for a visual object class is one similar to the classes Box, Cone, Cylinder, and Sphere defined in the `com.sun.j3d.utils.geometry` package. Each class extends Primitive, which extends Group. The design details of Primitive and its descendants are not discussed in this tutorial, but the source code for all of these classes is available with the Java 3D API distribution. From the source of Primitive class, and other utility classes, the reader can learn more about this class design approach.

2.3 Geometric Utility Classes

This section covers the utility classes for creating box, cone, cylinder, and sphere geometric primitives. The geometric primitives are the second easiest way to create content in a virtual universe. The easiest way is to use the ColorCube class.

The primitive classes provide the programmer with more flexibility than the ColorCube class provides. A ColorCube object defines the geometry and color in a Geometry node component. Consequently,

everything about a ColorCube is fixed, except its size[4]. The size of a ColorCube is only specified when the object is created.

A primitive object provides more flexibility by specifying shape without specifying color. In a geometric primitive utility class, the programmer cannot change the geometry, but can change the appearance[5]. The primitive classes give the programmer the flexibility to have multiple instances of the same geometric primitive where each can have a different appearance by having a reference to different Appearance NodeComponents.

The Box, Cone, Cylinder and Sphere utility classes are defined in the `com.sun.j3d.utils.geometry` package. Details of the Box, Cone, Cylinder, and Sphere classes are presented in Sections 2.3.1 through 2.3.4, respectively. The superclass of these primitives, Primitive, is discussed in Section 2.3.5. The portion of the `com.sun.j3d.utils.geometry` package hierarchy that contains the primitive classes is shown in Figure 2-4.

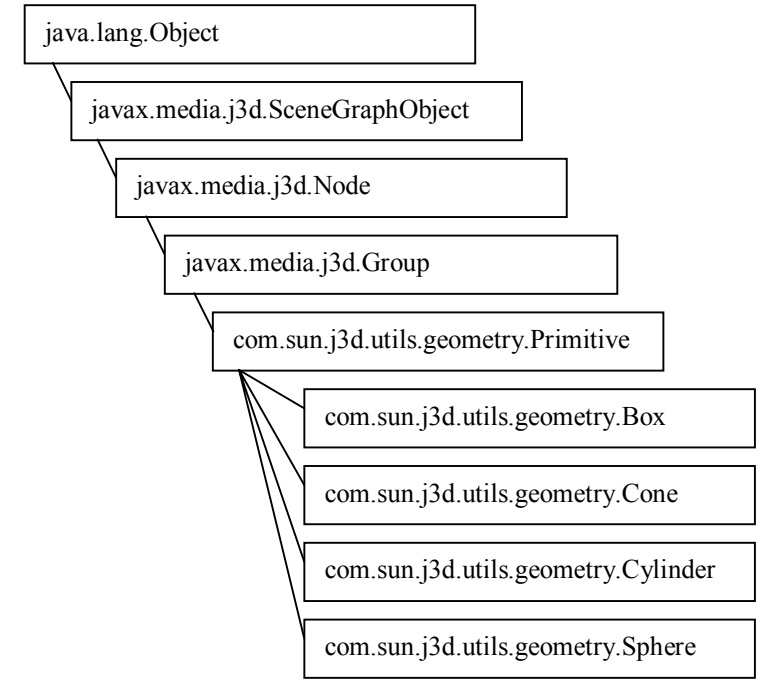


Figure 2-4 Class Hierarchy for Utility Geometric Primitives: Box, Cone, Cylinder, and Sphere

2.3.1 Box

The Box geometric primitive creates 3D box visual objects[6]. The defaults for length, width, and height are 2 meters, with the center at the origin, resulting in a cube with corners at (-1, -1, -1) and (1, 1, 1). The length, width, and height can be specified at object creation time. Of course, TransformGroup along

[4] The Geometry NodeComponent referenced by a ColorCube object can be changed, but then it wouldn't appear as a ColorCube.

[5] Just like with ColorCube, the Geometry NodeComponent referenced by a primitive object can be changed, but then it wouldn't appear as the primitive.

[6] Technically, a box is a six-sided polyhedron with rectangular faces.

the scene graph path to a Box can be used to change the location and/or orientation of instances of Box and other visual objects.

Box Constructors (partial list)

Package: `com.sun.j3d.utils.geometry`

Box extends Primitive, another class in the com.sun.j3d.utils.geometry package.

`Box()`
Constructs a default box of 2.0 meters in height, width, and depth, centered at the origin.

`Box(float xdim, float ydim, float zdim, Appearance appearance)`
Constructs a box of a given dimension and appearance, centered at the origin.

While the constructors differ by class, Box, Cone, and Cylinder classes share the same methods. The following reference block lists the methods for these classes.

Box, Cone, and Cylinder Methods

Package: `com.sun.j3d.utils.geometry`

These methods are defined in each of the Primitive classes: Box, Cone, and Cylinder. These primitives are composed of multiple Shape3D objects in a group.

`Shape3D getShape(int id)`
Gets one of the faces (Shape3D) from the primitive that contains the geometry and appearance. Box, Cone, and Cylinder objects are composed of more than one Shape3D object, each with its own Geometry node component. The value used for partid specifies which of the Geometry node components to get.

`void setAppearance(Appearance appearance)`
Sets appearance of the primitive (for all of the Shape3D objects).

2.3.2 Cone

The Cone class defines capped, cone shaped objects centered at the origin with the central axis aligned along the y-axis. The default for radius is 1.0 and 2.0 for height. The center of the cone is defined to be the center of its bounding box rather than its centroid.

Cone Constructors (partial list)

Package: `com.sun.j3d.utils.geometry`

Cone extends Primitive, another class in the com.sun.j3d.utils.geometry package.

`Cone()`
Constructs a default Cone of radius of 1.0 and height of 2.0.

`Cone(float radius, float height)`
Constructs a default Cone of a given radius and height.

2.3.3 Cylinder

Cylinder class creates a capped, cylindrical object centered at the origin with its central axis aligned along the y-axis. The default for radius is 1.0 and 2.0 for height.

Cylinder Constructors (partial list)

Package: `com.sun.j3d.utils.geometry`

Cylinder extends Primitive, another class in the com.sun.j3d.utils.geometry package.

`Cylinder()`
Constructs a default cylinder of radius of 1.0 and height of 2.0.

`Cylinder(float radius, float height)`
Constructs a cylinder of a given radius and height.

`Cylinder(float radius, float height, Appearance appearance)`
Constructs a cylinder of a given radius, height, and appearance.

2.3.4 Sphere

The Sphere class creates spherical visual objects centered at the origin. The default radius is 1.0.

Sphere Constructors (partial list)

Package: `com.sun.j3d.utils.geometry`

Sphere extends Primitive, another class in the com.sun.j3d.utils.geometry package.

`Sphere()`
Constructs a default Sphere of radius of 1.0.

`Sphere(float radius)`
Constructs a default Sphere of a given radius.

`Sphere(float radius, Appearance appearance)`
Constructs a Sphere of a given radius and a given appearance.

Sphere Methods

Package: `com.sun.j3d.utils.geometry`

As an extension of Primitive, a Sphere is a Group object that has a single Shape3D child object.

`Shape3D getShape()`
Gets the Shape3D that contains the geometry and appearance.

`Shape3D getShape(int id)`
This method is included for compatibility with the other Primitive classes: Box, Cone, and Cylinder. However, since a Sphere has only one Shape3D object, it can be called only with id = 1.

`void setAppearance(Appearance appearance)`
Sets appearance of the sphere.

2.3.5 More About Geometric Primitives

The geometry of a primitive utility class does not define color. Geometry that does not define color derives its color from its Appearance node component. Without a reference to an Appearance node component, the visual object will be white, the default appearance color. Color is first discussed in Section 2.4.2 and added to geometry in Section 2.5.1. Section 2.6 presents the details of Appearance node components.

The Primitive class defines default values common to Box, Cone, Cylinder, and Sphere. For example, Primitive defines the default value for the number of polygons used to represent surfaces. Section 2.3.8 presents some of the details of the Primitive class. Since the default values defined by Primitive are fine for most applications, Java 3D programs can be written without even using the Primitive class. For this reason, the section describing the Primitive class is considered an advanced topic (which can be skipped). You will recognize advanced sections when you get there by the Duke figure hanging from the double-line outline.

2.3.6 ColorCube

The ColorCube class is presented here to contrast with the geometric primitive classes of Box, Cone, Cylinder, and Sphere. The ColorCube class extends a different hierarchy than the graphic primitive classes. It is a subclass of Shape3D. This hierarchy for ColorCube is shown in Figure 2-5. Chapter 1 contains the reference blocks for ColorCube.

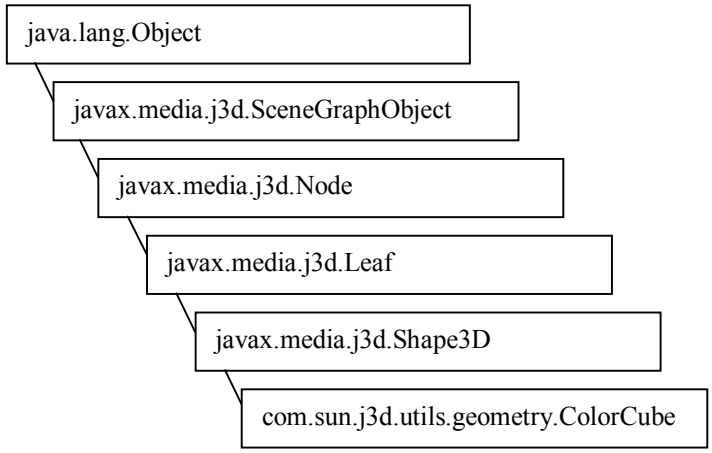


Figure 2-5 Class Hierarchy of ColorCube Utility Geometric Class

ColorCube is the only class distributed with the Java 3D API that allows a programmer to ignore the issues of colors and lights. For this reason, ColorCube class is useful for quickly assembling scene graphs for testing or prototyping.

2.3.7 Example: Creating a Simple Yo-Yo From Two Cones

This section presents a simple example that uses the Cone class: `ConeYoyoApp.java`. The goal of the program is to render a yo-yo. Two cones are used to form the yo-yo. Java 3D API behaviors could be used to make the yo-yo move up and down, but that is beyond the scope of this Chapter. The program spins the yo-yo so the geometry can be appreciated. The scene graph diagram in Figure 2-5 shows the designs for the ConeYoyo and ConeYoyoApp classes in the ConoYoyoApp example program.

The default position of a Cone object is with its bounding box centered at the origin. The default orientation is with the tip of the Cone object in the direction of the positive y-axis. The yo-yo is formed of two cones that are rotated about the z-axis and translated along the x-axis to bring the tips of the cones together at the origin. Other combinations of rotation and translation transformations could bring the tips of the Cone objects together.

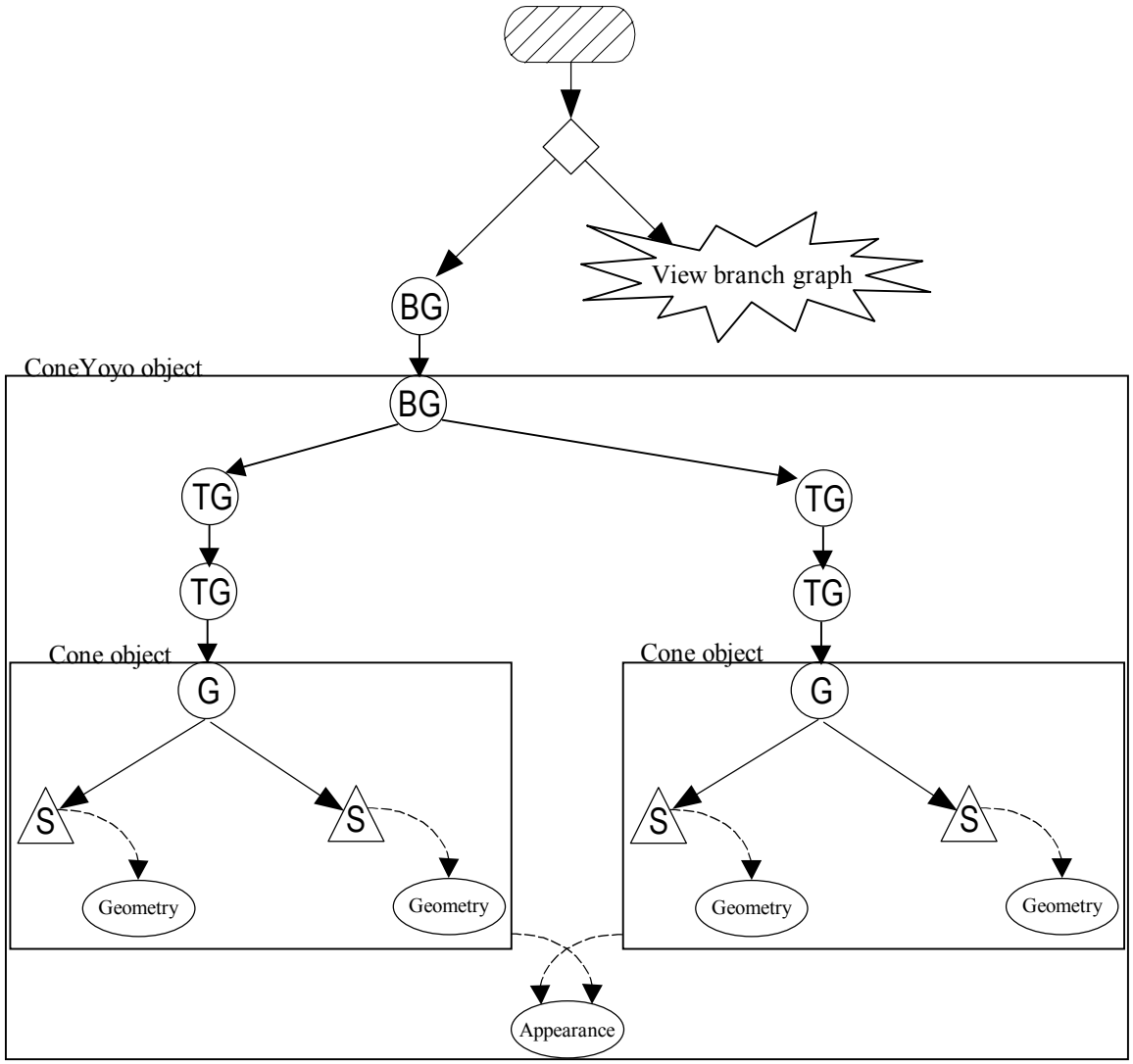


Figure 2-6 Scene Graph for ConeYoyoApp[7]

In the branch graph that begins with the BranchGroup object created by the ConeYoyo object, the scene graph path to each Cone object begins with the TransformGroup object that specifies the translation, followed by the TransformGroup that specifies the rotation, and terminates at the Cone object.

Several scene graphs may represent the same virtual world. Taking the scene graph of Figure 2-6 as an example, some obvious changes can be made. One change eliminates the BranchGroup object whose child is the ConeYoyo object and inserts the ConeYoyo object directly in the Locale. The BranchGroup is there to add future visual objects to the visual world. Another change combines the two TransformGroup objects inside the ConeYoyo object. The transformations are shown this way simply as an example.

[7] Actually, the Cone primitive is shared automatically as a feature of the Primitive class. This feature is discussed in Section 2.3.8.

Shape3D nodes of the Cone objects reference Geometry node components. These are internal to the Cone objects. The Shape3D objects of the Cone are children of a Group in the Cone. Since Cone objects descend from Group, the same Cone (or other Primitive object) can not be used more than once in a scene graph. Figure 2-7 shows an example error message produced when attempting to use the same Cone object in a single scene graph. This error does not exist in the example program distributed with this tutorial.

```
Exception in thread "main" javax.media.j3d.MultipleParentException:
Group.addChild: child already has a parent
       at javax.media.j3d.GroupRetained.addChild(GroupRetained.java:246)
       at javax.media.j3d.Group.addChild(Group.java:241)
       at ConeYoyoApp$ConeYoyo.<init>(ConeYoyoApp.java:89)
       at ConeYoyoApp.createSceneGraph(ConeYoyoApp.java:119)
       at ConeYoyoApp.<init>(ConeYoyoApp.java:159)
       at ConeYoyoApp.main(ConeYoyoApp.java:172)
```

Figure 2-7 Multiple Parent Exception While Attempting to Reuse a Cone Object

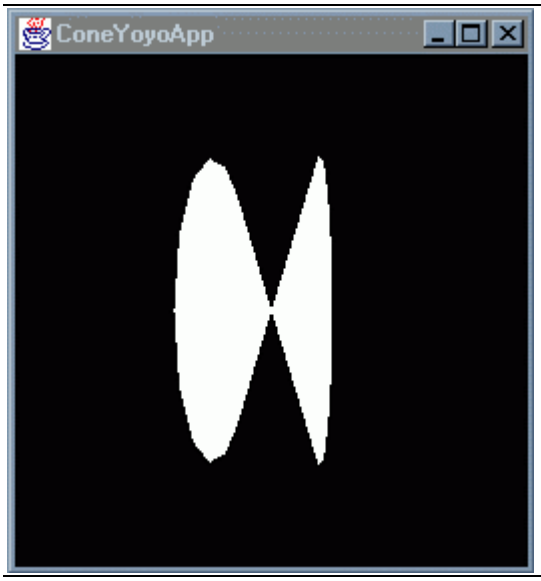


Figure 2-8 An Image Rendered by ConeYoyoApp.java

Figure 2-8 shows one of the possible images rendered by `ConeYoyoApp.java` as the ConeYoyo object spins. `ConeYoyoApp.java` is found in the `example/Geometry` subdirectory. The ConeYoyo class in the program is reproduced here in Code Fragment 2-2.

Lines 14 through 21 create the objects of one half of the yo-yo scene graph. Lines 23 through 25 create the relationships among these objects. The process is repeated for the other half of the yo-yo on lines 27 through 38.

Line 12 creates **`yoyoAppear`**, an Appearance node component with default values, to be used by the Cone objects. Lines 21 and 34 set the appearance for the two cones.

```
1.      public class ConeYoyo{
2.
3.          private BranchGroup yoyoBG;
4.
5.          // create Shape3D with geometry and appearance
```

```
6.          //
7.          public ConeYoyo() {
8.
9.              yoyoBG = new BranchGroup();
10.             Transform3D rotate = new Transform3D();
11.             Transform3D translate = new Transform3D();
12.             Appearance yoyoAppear = new Appearance();
13.
14.             rotate.rotZ(Math.PI/2.0d);
15.             TransformGroup yoyoTGR1 = new TransformGroup(rotate);
16.
17.             translate.set(new Vector3f(0.1f, 0.0f, 0.0f));
18.             TransformGroup yoyoTGT1 = new TransformGroup(translate);
19.
20.             Cone cone1 = new Cone(0.6f, 0.2f);
21.             cone1.setAppearance(yoyoAppear);
22.
23.             yoyoBG.addChild(yoyoTGT1);
24.             yoyoTGT1.addChild(yoyoTGR1);
25.             yoyoTGR1.addChild(cone1);
26.
27.             translate.set(new Vector3f(-0.1f, 0.0f, 0.0f));
28.             TransformGroup yoyoTGT2 = new TransformGroup(translate);
29.
30.             rotate.rotZ(-Math.PI/2.0d);
31.             TransformGroup yoyoTGR2 = new TransformGroup(rotate);
32.
33.             Cone cone2 = new Cone(0.6f, 0.2f);
34.             cone2.setAppearance(yoyoAppear);
35.
36.             yoyoBG.addChild(yoyoTGT2);
37.             yoyoTGT2.addChild(yoyoTGR2);
38.             yoyoTGR2.addChild(cone2);
39.
40.             yoyoBG.compile();
41.
42.         } // end of ConeYoyo constructor
43.
44.         public BranchGroup getBG(){
45.             return yoyoBG;
46.         }
47.
48.     } // end of class ConeYoyo
```

Code Fragment 2-2 Class ConeYoyo From ConeYoyoApp.java Example Program

2.3.8 Advanced Topic: Geometric Primitive

The class hierarchy of Figure 2-4 shows Primitive as the superclass of Box, Cone, Cylinder, and Sphere classes. It defines a number of fields and methods common to these classes, as well as default values for the fields.

Each of the subclasses of Primitive has a constructor that allows the setting of flags when the object is constructed. For example, the Sphere class has the constructor Sphere(int). The available flags are listed below.

The primitive flags are:

GEOMETRY_NOT_SHARED	Normals are generated along with the positions.
GENERATE_NORMALS_INWARD	Normals are flipped along the surface.
GENERATE_TEXTURE_COORDS	Texture coordinates are generated.
GEOMETRY_NOT_SHARED	The geometry created will not be shared by another node.

The Primitive class provides a way to share Geometry node components among instances of a primitive of the same size. By default, all primitives of the same size share one geometry node component. An example of a field defined in the Primitive class is the GEOMETRY_NOT_SHARED integer. This field specifies the geometry being created will not be shared by another. Set this flag to prevent the geometry from being shared among primitives of the same parameters (e.g., spheres with radius 1).

```
Cone myCone = new Cone(Primitive.GEOMETRY_NOT_SHARED);
```

Primitive Methods (partial list)

Package: `com.sun.j3d.utils.geometry`

Primitive extends Group and is the superclass for Box, Cone, Cylinder, and Sphere.

`public void setNumVertices(int num)`
Sets total number of vertices in this primitive.

`void setAppearance(int partid, Appearance appearance)`
Sets the appearance of a subpart given a partid. Box, Cone, and Cylinder objects are composed of more than one Shape3D object, each potentially with its own Appearance node component. The value used for partid specifies which of the Appearance node components to set.

`void setAppearance()`
Sets the main appearance of the primitive (all subparts) to a default white appearance.

Additional constructors for Box, Cone, Cylinder, and Sphere allow the specification of Primitive flags at object creation time. Consult the Java 3D API specification for more information.

2.4 Mathematical Classes

To create visual objects, the Geometry class and its subclasses are required. Many Geometry subclasses describe vertex-based primitives, such as points, lines, and filled polygons. The subclasses of Geometry will be discussed in Section 2.5, but before that discussion, several mathematical classes (Point*, Color*, Vector*, TexCoord*) used to specify vertex-related data need to be discussed[8].

Note the asterisk used above is a wildcard to represent variations of class names. For example, Tuple* refers to all Tuple classes: Tuple2f, Tuple2d, Tuple3b, Tuple3f, Tuple3d, Tuple4b, Tuple3i, Tuple4f, Tuple4d, and Tuple4i. In each case the number indicates the number of elements in the tuple, and the

[8] TexCoord* classes were not used in Java 3D API version 1.1.

letter indicates the data type of the elements. ‘f’ indicates single-precision floating point, ‘d’ indicates double-precision floating point, ‘b’ is for bytes, and 'i' is for signed integer. So Tuple3f is a class that manipulates three single-precision floating point values.

All these mathematical classes are in the `javax.vecmath.*` package. This package defines several Tuple* classes as generic abstract superclasses. Other more useful classes are derived from the various Tuple classes. The hierarchy for some of the package is shown in Figure 2-9.

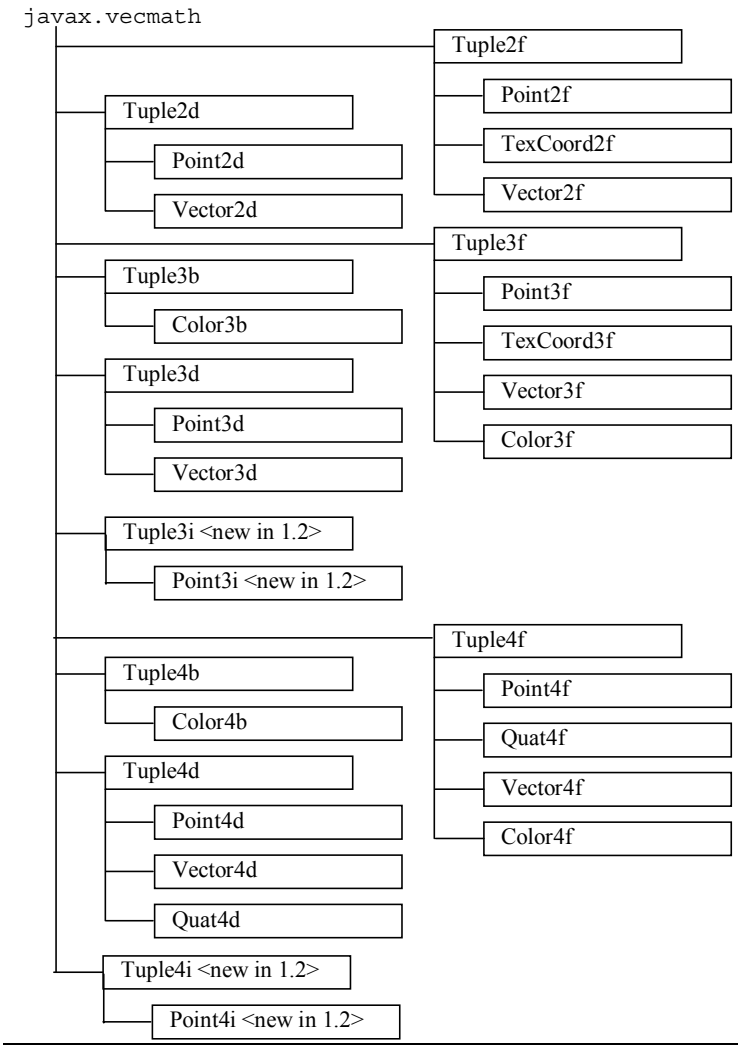


Figure 2-9 Mathematical Classes Package and Hierarchy

Each vertex of a visual object may specify up to four `javax.vecmath` objects, representing coordinates, colors, surface normals, and texture coordinates. The following classes are commonly used:

- Point* (for coordinates)
- Color* (for colors)
- Vector* (for surface normals)
- TexCoord* (for texture coordinates, see Chapter 7)

Note that coordinates (Point* objects) are necessary to position each vertex. The other data is optional, depending upon how the primitive is rendered. For instance, a color (a Color* object) may be defined at each vertex and the colors of the primitive are interpolated between the colors at the vertices. If lighting is enabled, surface normals (and therefore Vector* objects) are needed. If texture mapping is enabled, then texture coordinates may be needed.

(Quat* objects represent quaternions, which are only used for advanced 3D matrix transformations.)

Since all the useful classes inherit from the abstract Tuple* classes, it's important to be familiar with the Tuple constructors and methods, which are listed below.

Tuple2f Constructors

Package: `javax.vecmath`

Tuple* classes are not typically used directly in Java 3D programs but provide the base for Point*, Color*, Vector*, and TexCoord* classes. In particular, Tuple2f provides the base for Point2f, Color2f, and TexCoord2f. The constructors listed here are available to these subclasses. Tuple3f and Tuple4f have similar sets of constructors.

`Tuple2f()`
Constructs and initializes a Tuple object with the coordinates (0,0).

`Tuple2f(float x, float y)`
Constructs and initializes a Tuple object from the specified x, y coordinates.

`Tuple2f(float t[])`
Constructs and initializes a Tuple object from the specified array.

`Tuple2f(Tuple2f t)`
Constructs and initializes a Tuple object from the data in another Tuple object.

`Tuple2f(Tuple2d t)`
Constructs and initializes a Tuple object from the data in another Tuple object.

Tuple2f Methods (partial list)

Package: `javax.vecmath`

Tuple* classes are not typically used directly in Java 3D programs but provide the base for Point*, Color*, Vector*, and TexCoord* classes. In particular, Tuple2f provides the base for Point2f, Color2f, and TexCoord2f. The methods listed here are available to these subclasses. Tuple3f and Tuple4f have similar sets of methods. Additional methods exist for clamping values to a range and linear interpolation.

`void set(float x, float y)`

`void set(float t[])`
Sets the value of this tuple from the specified values.

`boolean equals(Tuple2f t1)`
Returns true if the data in the Tuple t1 are equal to the corresponding data in this tuple.

`final void add(Tuple2f t1)`
Sets the value of this tuple to the vector sum of itself and Tuple t1.

`void add(Tuple2f t1, Tuple2f t2)`
Sets the value of this tuple to the vector sum of tuples t1 and t2.

`void sub(Tuple2f t1, Tuple2f t2)`
Sets the value of this tuple to the vector difference of tuple t1 and t2 (this = t1 - t2).

`void sub(Tuple2f t1)`
Sets the value of this tuple to the vector difference of itself and tuple t1 (this = this - t1).

`void negate()`
Negates the value of this vector in place.

`void negate(Tuple2f t1)`
Sets the value of this tuple to the negation of tuple t1.

`void absolute()`
Sets each component of this tuple to its absolute value.

`void absolute(Tuple2f t)`
Sets each component of the tuple parameter to its absolute value, and places the modified values into this tuple.

There are subtle, but predictable, differences among Tuple* constructors and methods, due to number and data type. For example, Tuple3d differs from Tuple2f, because it has a constructor method:

```
Tuple3d(double x, double y, double z);
```

which expects three, not two, double-precision, not single-precision, floating point parameters.

Each of the Tuple* classes has public members. For Tuple2*, they are x and y. For Tuple3* the members are x, y, and z. For Tuple4* the members are x, y, z, and w.

2.4.1 Point Classes

Point* objects usually represent coordinates of a vertex, although they can also represent the position of a raster image, point light source, spatial location of a sound, or other positional data. The constructors for Point* classes are similar to the Tuple* constructors, except they return Point* objects. (Some constructors are passed parameters which are Point* objects, instead of Tuple* objects.)

Point3f Methods (partial list)

Package: `javax.vecmath`

The Point* classes are derived from Tuple* classes. Each instance of the Point* classes represents a single point in two-, three-, or four-space. In addition to the Tuple* methods, Point* classes have additional methods, some of which are listed here.

`float distance(Point3f p1)`
Returns the Euclidean distance between this point and point p1.

`float distanceSquared(Point3f p1)`
Returns the square of the Euclidean distance between this point and point p1.

`float distanceL1(Point3f p1)`
Returns the L_1 (Manhattan) distance between this point and point p1. The L_1 distance is equal to:
$abs(x_1 - x_2) + abs(y_1 - y_2) + abs(z_1 - z_2)$

Once again, there are subtle, predictable differences among Point* constructors and methods, due to number and data type. For example, for Point3d, the distance method returns a double-precision floating point value.

2.4.2 Color Classes

Color* objects represent a color, which can be for a vertex, material property, fog, or other visual object. Colors are specified either as Color3* or Color4*, and only for byte or single-precision floating point data types. Color3* objects specify a color as a combination of red, green, and blue (RGB) values. Color4* objects specify a transparency value, in addition to RGB. (By default, Color3* objects are opaque.) For byte-sized data types, color values (including alpha) range between 0 and 255, inclusive. For single-precision floating point data, color values (including alpha) range between 0.0 and 1.0, inclusive.

The alpha value of a Color4* object relates to its opacity. If the alpha value is 0 then the color is not opaque (it is fully transparent). If the alpha value is 1.0 (or 255) then the color is fully opaque.

Note that Java defines a byte as a signed integer in the range [-128, 127]. However, colors are more typically represented by values in the range [0, 255]. The Java 3D API recognizes this and treats byte color values as if the range were [0, 255]. [9]

It is sometimes convenient to create constants for colors that are used repetitiously in the creation of visual object. For example,

```
Color3f red = new Color3f(1.0f, 0.0f, 0.0f);
```

[9] In API v1.2 if the `toString()` method is used for a byte data format Color object, the values will be displayed in the range [-128, 127].

instantiates the Color3f object **red** that may be used multiple times. It may be helpful to create a class that contains a number of color constants. An example of such a class appears in Code Fragment 2-1.

```
1. import javax.vecmath.*;
2.
3. class ColorConstants{
4.    public static final Color3f red     = new Color3f(1.0f,0.0f,0.0f);
5.    public static final Color3f green   = new Color3f(0.0f,1.0f,0.0f);
6.    public static final Color3f blue    = new Color3f(0.0f,0.0f,1.0f);
7.    public static final Color3f yellow  = new Color3f(1.0f,1.0f,0.0f);
8.    public static final Color3f cyan    = new Color3f(0.0f,1.0f,1.0f);
9.    public static final Color3f magenta = new Color3f(1.0f,0.0f,1.0f);
10.   public static final Color3f white   = new Color3f(1.0f,1.0f,1.0f);
11.   public static final Color3f black   = new Color3f(0.0f,0.0f,0.0f);
12.}
```

Code Fragment 2-3 Example ColorConstants Class

Once again, the constructors for Color* classes are similar to the Tuple* constructors, except they return Color* objects. (Some constructors are passed parameters which are Color* objects.) The Color* classes define only one constructor and two methods in addition to those inherited from their Tuple* superclasses.

Color* Constructors (partial list)

Package: `javax.vecmath`

The Color* classes are derived from Tuple* classes. Each instances of the Color* classes represents a single color in three components: red, green, and blue (RGB), or four components: red, green, blue and alpha (RGBA). This alpha value represents transparency of the color where 0 is fully transparent and 1 is fully opaque. Several other constructors are listed in the Tuple3* and Tuple4*

`Color*(Color color)` **`<new in 1.2>`**

Create a Java 3D Color* object (Color3b, Color3f, Color4b, Color4f) using a AWT Color object. When constructing a Color4* object using a AWT Color object, the alpha component (opacity) is set to 1.0.

Color* Methods

Package: `javax.vecmath`

The Color* classes (Color3b, Color3f, Color4b, Color4f) are derived from Tuple* classes. Each instances of the Color* classes represents a single color in three components (RGB), or four components (RGBA). Several other methods are listed in the Tuple3* and Tuple4*

`void set(Color color)` **`<new in 1.2>`**
Set the color using a color from AWT Color object.

`Color get()` **`<new in 1.2>`**
Get the color as a color AWT Color object.

2.4.3 Vector Classes

Vector* objects often represent surface normals at vertices although they can also represent the direction of a light source or sound source. Again, the constructors for Vector* classes are similar to the Tuple* constructors. However, Vector* objects add many methods that are not found in the Tuple* classes.

Vector3f Methods (partial list)

Package: `javax.vecmath`

The Vector* classes are derived from Tuple* classes. Each instances of the Vector* classes represents a single vector in two-, three-, or four-space. In addition to the Tuple* methods, Vector* classes have additional methods, some of which are listed here.

`float length()`
Returns the length of this vector.

`float lengthSquared()`
Returns the squared length of this vector.

`void cross(Vector3f v1, Vector3f v2)`
Sets this vector to be the vector cross product of vectors v1 and v2.

`float dot(Vector3f v1)`
Computer and return the dot product of this vector and vector v1.

`void normalize()`
Normalizes this vector.

`void normalize(Vector3f v1)`
Sets the value of this vector to the normalization of vector v1.

`float angle(Vector3f v1)`
Returns the angle in radians between this vector and the vector parameter; the return value is constrained to the range [0,PI].

And yes, there are subtle, predictable differences among Vector* constructors and methods, due to number or data type.

2.4.4 TexCoord Classes

There are only two TexCoord* classes which can be used to represent a set of texture coordinates at a vertex: TexCoord2f and TexCoord3f. TexCoord2f maintains texture coordinates as an (s, t) coordinate pair; TexCoord3f as an (s, t, r) triple. Chapter 7 explains TexCoord* classes in context.

The constructors for TexCoord* classes are again similar to the Tuple* constructors. The TexCoord* classes do not have additional methods, so they rely upon the methods they inherit from their Tuple* superclasses.

2.5 Geometry Classes

In 3D computer graphics, everything from the simplest triangle to the most complicated jumbo jet model is modeled and rendered with vertex-based data. With Java 3D, each Shape3D object should call its method `setGeometry()` to reference one and only one Geometry object. To be more precise, Geometry is an abstract superclass, so the referenced object is an instance of a subclass of Geometry.

Subclasses of Geometry fall into three broad categories:

- Non-indexed vertex-based geometry (each time a visual object is rendered, its vertices may be used only once)
- Indexed vertex-based geometry (each time a visual object is rendered, its vertices may be reused)
- Other visual objects (the classes Raster, Text3D, and CompressedGeometry)

This section covers the first two aforementioned categories. The class hierarchy for Geometry classes and subclasses is shown in Figure 2-10 Geometry Class Hierarchy

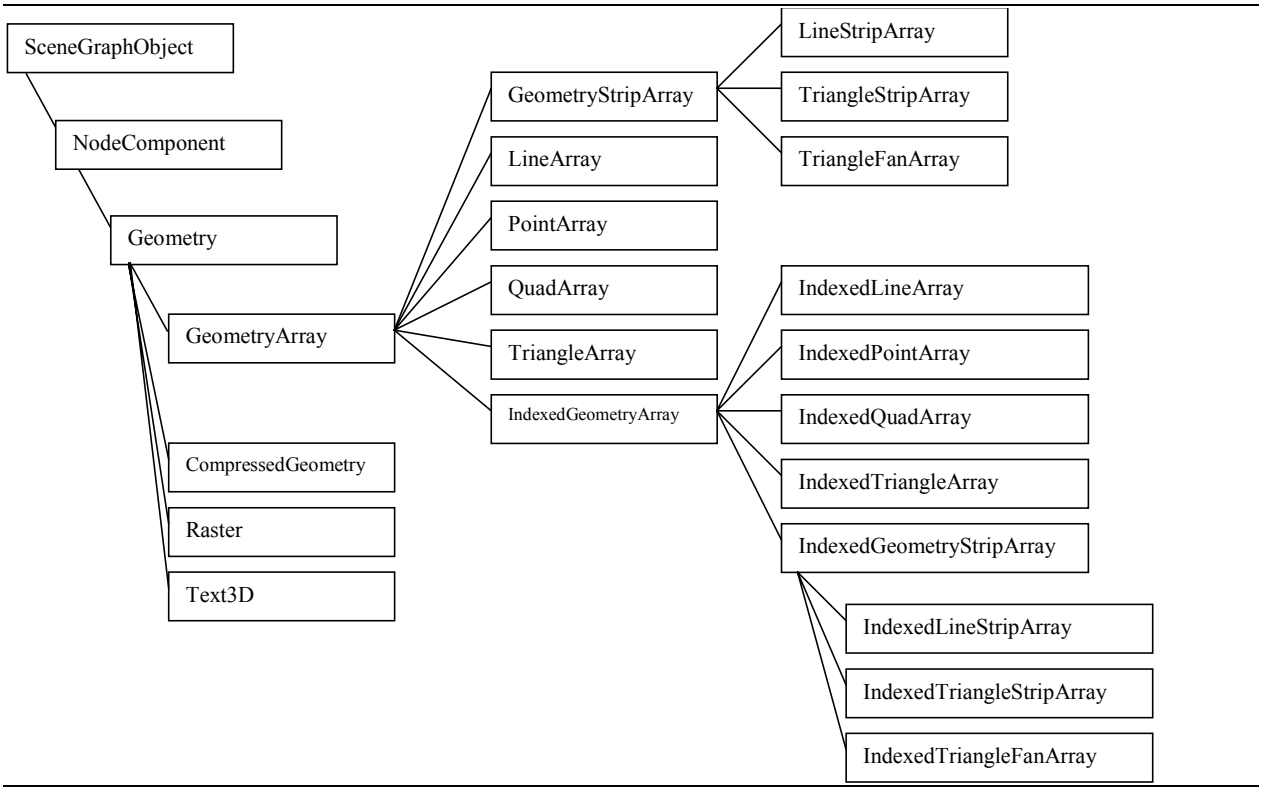


Figure 2-10 Geometry Class Hierarchy

2.5.1 GeometryArray Class

As you may deduce from the class names, the Geometry subclasses may be used to specify points, lines, and filled polygons (triangles and quadrilaterals). These vertex-based primitives are subclasses of the GeometryArray abstract class, which indicates that each has arrays that maintain data per vertex.

For example, if a GeometryArray object is used to specify one triangle, a three-element array is defined: one element for each vertex. Each element of this array maintains the coordinate location for its vertex (which can be defined with a Point* object or similar data). In addition to the coordinate location, three more arrays may be optionally defined to store color, surface normal, and texture coordinate data. These arrays, containing the coordinates, colors, surface normals, and texture coordinates, are the "data arrays."

There are three steps in the life of a GeometryArray object:

1. Construction of an empty object.
2. Filling the object with data.
3. Associating (referencing) the object from (one or more) Shape3D objects.

Step 1: Construction of an Empty GeometryArray Object

When a GeometryArray object is initially constructed, two things must be defined:

- the number of vertices (array elements) to be needed.
- the type of data (coordinate location, color, surface normal, and/or texture coordinate) to be stored at each vertex. This is called the **vertex format**.

There is only one GeometryArray constructor method:

GeometryArray Constructor

`GeometryArray(int vertexCount, int vertexFormat)`
Constructs an empty GeometryArray object with the specified number of vertices, and vertex format. One or more individual flags are bitwise "OR"ed together to describe the per-vertex data. The flag constants used for specifying the format are:

COORDINATES: Specifies this vertex array contains coordinates. This bit must be set.
NORMALS: Specifies this vertex array contains surface normals.
COLOR_3: Specifies this vertex array contains colors without transparency.
COLOR_4: Specifies this vertex array contains colors with transparency.
TEXTURE_COORDINATE_2: Specifies this vertex array contains 2D texture coordinates.
TEXTURE_COORDINATE_3: Specifies this vertex array contains 3D texture coordinates.

For each vertex format flags set, there is a corresponding array created internal to the GeometryArray object. Each of these arrays is vertexCount in size.

Let's see how this constructor works, but first recall that GeometryArray is an abstract class. Therefore, you actually call the constructor for one of GeometryArray's subclasses, for instance, LineArray. (A LineArray object describes a set of vertices, and each two vertices defines the endpoints of a line. The constructor and other methods of LineArray are very similar to its superclass GeometryArray. LineArray is explained in more detail in Section 2.5.2.)

Code Fragment 2-4 shows the Axis class from the program `examples/Geometry/AxisApp.java` which uses multiple LineArray objects to draw lines to represent the x, y, and z axes. The X axis object creates an object with two vertices (to draw one line between them), with only coordinate location data.

The Y axis object also has two vertices, but allows for RGB color, as well as coordinate location, at each vertex. Therefore, the Y axis line may be drawn with colors interpolated from one vertex to the other. Finally, the Z axis has ten vertices with coordinate and color data at each vertex. Five color-interpolated lines may be drawn, one line between each pair of vertices. Note the use of the bitwise "OR" operation for the vertex format of both the Y and Z axes.

```
1. // construct object to represent the X axis
2. LineArray axisXLines= new LineArray (2, LineArray.COORDINATES);
3.
4. // construct object to represent the Y axis
5. LineArray axisYLines = new LineArray(2, LineArray.COORDINATES
6.                                        | LineArray.COLOR_3);
7.
8. // construct object to represent the Z axis
9. LineArray axisZLines = new LineArray(10, LineArray.COORDINATES
10.                                         | LineArray.COLOR_3);
```

Code Fragment 2-4 GeometryArray Constructors

Be careful! The Axis class in `AxisApp.java` is different from the Axis class defined in `examples/geometry/Axis.java`, which uses only one LineArray object. Make sure you have the right one. The Axis class defined in `Axis.java` is intended for use in your programs, where AxisApp.java is the demonstration program for this tutorial. Also, the Axis class defined in Axis.java demonstrates creating a visual object class that extends Shape3D.

Step 2: Fill the GeometryArray Object with Data

After constructing the GeometryArray object, assign values to the arrays, corresponding to the assigned vertex format. This may be done per vertex, or by using an array to assign data to many vertices with one method call. The available methods are:

GeometryArray Methods (partial list)

GeometryArray is the superclass for PointArray, LineArray, TriangleArray, QuadArray, GeometryStripArray, and IndexedGeometryArray.

`void setCoordinate(int index, float coordinate[])`

`void setCoordinate(int index, double coordinate[])`

`void setCoordinate(int index, Point* coordinate)`
Sets the coordinate associated with the vertex at the specified index for this object.

`void setCoordinates(int index, float coordinates[])`

`void setCoordinates(int index, double coordinates[])`

`void setCoordinates(int index, Point* coordinates[])`
Sets the coordinates associated with the vertices starting at the specified index for this object.

`void setColor(int index, float color[])`

`void setColor(int index, byte color[])`

`void setColor(int index, Color* color)`
Sets the color associated with the vertex at the specified index for this object.

```
void setColors(int index, float colors[])
void setColors(int index, byte colors[])
void setColors(int index, Color* colors[])
```
Sets the colors associated with the vertices starting at the specified index for this object.

```
void setNormal(int index, float normal[])
void setNormal(int index, Vector* normal)
```
Sets the normal associated with the vertex at the specified index for this object.

```
void setNormals(int index, float normals[])
void setNormals(int index, Vector* normals[])
```
Sets the normals associated with the vertices starting at the specified index for this object.

```
void setTextureCoordinate(int texCoordSet, int index, float texCoord[])
void setTextureCoordinate(int texCoordSet, int index, TexCoord* coordinate)
```
Sets the texture coordinate associated with the vertex at the specified index for the specified texture coordinate set of this object.

GeometryArray Methods (partial list, continued)

```
void setTextureCoordinates(int texCoordSet, int index,            <new in 1.2>
                           float texCoords[])
void setTextureCoordinates(int texCoordSet, int index,            <new in 1.2>
                           TexCoord* texCoords[])
```
Sets the texture coordinates associated with the vertices starting at the specified index for the specified texture coordinate set of this object.

```
void setInitialVertexIndex(int initialVertexIndex)                <new in 1.2>
```
Sets the initial vertex index for this GeometryArray object. Allows some of the vertex data to be ignored in rendering, picking and other operations.

```
void setValidVertexCount(int validVertexCount)                    <new in 1.2>
```
Sets the valid vertex count for this GeometryArray object. Allows some of the vertex data to be ignored in rendering, picking and other operations.

Code Fragment 2-5 shows use of the GeometryArray methods to store coordinate and color values in the LineArray objects. The X axis object calls only the method setCoordinate() to store coordinate location data. The Y axis object calls both setColor() and setCoordinate() to load RGB color and coordinate location values. And the Z axis object calls setCoordinate() ten times for each individual vertex and setColors() once to load all ten vertices with one method call.

```
1. axisXLines.setCoordinate(0, new Point3f(-1.0f, 0.0f, 0.0f));
2. axisXLines.setCoordinate(1, new Point3f( 1.0f, 0.0f, 0.0f));
3.
4. Color3f red   = new Color3f(1.0f, 0.0f, 0.0f);
5. Color3f green = new Color3f(0.0f, 1.0f, 0.0f);
6. Color3f blue  = new Color3f(0.0f, 0.0f, 1.0f);
7. axisYLines.setCoordinate(0, new Point3f( 0.0f,-1.0f, 0.0f));
8. axisYLines.setCoordinate(1, new Point3f( 0.0f, 1.0f, 0.0f));
9. axisYLines.setColor(0, green);
10.axisYLines.setColor(1, blue);
11.
12.axisZLines.setCoordinate(0, z1);
13.axisZLines.setCoordinate(1, z2);
14.axisZLines.setCoordinate(2, z2);
15.axisZLines.setCoordinate(3, new Point3f( 0.1f, 0.1f, 0.9f));
16.axisZLines.setCoordinate(4, z2);
17.axisZLines.setCoordinate(5, new Point3f(-0.1f, 0.1f, 0.9f));
18.axisZLines.setCoordinate(6, z2);
19.axisZLines.setCoordinate(7, new Point3f( 0.1f,-0.1f, 0.9f));
20.axisZLines.setCoordinate(8, z2);
21.axisZLines.setCoordinate(9, new Point3f(-0.1f,-0.1f, 0.9f));
22.
23.Color3f colors[] = new Color3f[10];          // array of colors
24.colors[0] = new Color3f(0.0f, 1.0f, 1.0f);   // set the first color
25.for(int v = 1; v < 10; v++)                  // set the rest of the colors
26.   colors[v] = red;
27.axisZLines.setColors(0, colors);             // use array colors in geom.
```

Code Fragment 2-5 Storing Data into a GeometryArray Object

The default color for vertices of a GeometryArray object is white, unless either COLOR_3 or COLOR_4 is specified in the vertex format. When either COLOR_3 or COLOR_4 is specified, the default vertex color is black. When lines or filled polygons are rendered with different colors at the vertices, the color is smoothly shaded (interpolated) between vertices using Gouraud shading.

Step 3: Make Shape3D Objects Reference the GeometryArray Objects

Finally, Code Fragment 2-6 shows how the GeometryArray objects are referenced by newly created Shape3D objects. In turn, the Shape3D objects are added to a BranchGroup, which is added elsewhere to the overall scene graph. (Unlike GeometryArray objects, which are NodeComponents, Shape3D is a subclass of Node, so Shape3D objects may be added as children to a scene graph.)

```
1. axisBG = new BranchGroup();
2.
3. axisBG.addChild(new Shape3D(axisXLines));
4. axisBG.addChild(new Shape3D(axisYLines));
5. axisBG.addChild(new Shape3D(axisZLines));
```

Code Fragment 2-6 GeometryArray Objects Referenced by Shape3D Objects

Figure 2-11 shows the partial scene graph created by the Axis class in `AxisApp.java`.

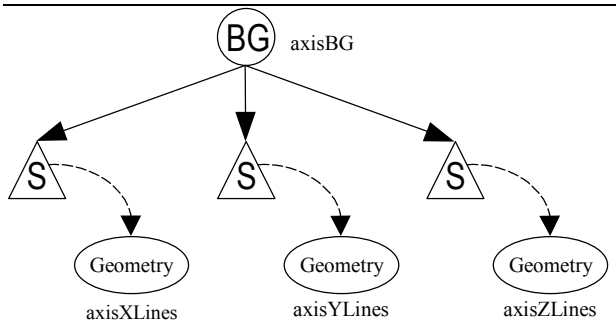


Figure 2-11 Axis Class in AxisApp.java Creates this Scene Graph

2.5.2 Subclasses of GeometryArray

As was discussed in the previous section, the GeometryArray class is an abstract superclass for more useful subclasses, such as LineArray. Figure 2-12 shows the class hierarchy for GeometryArray and some of its subclasses. The main distinction among these subclasses is how the Java 3D renderer decides to render their vertices.

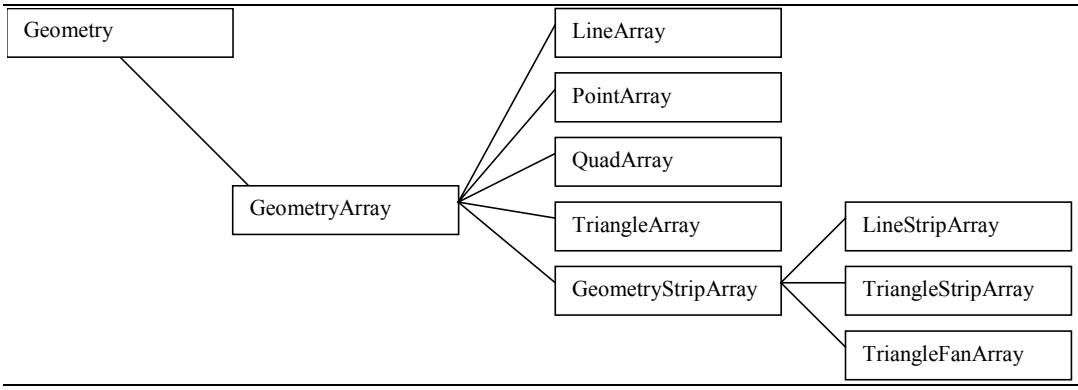


Figure 2-12 Non-Indexed GeometryArray Subclasses

Figure 2-13 shows examples of the four GeometryArray subclasses: PointArray, LineArray, TriangleArray, and QuadArray (the ones which are not also subclasses of GeometryStripArray). In this figure, the three leftmost sets of vertices show the same six vertex points rendering six points, three lines, or two triangles. The fourth image shows four vertices defining a quadrilateral. Note that none of the vertices are shared: each line or filled polygon is rendered independently of any other.

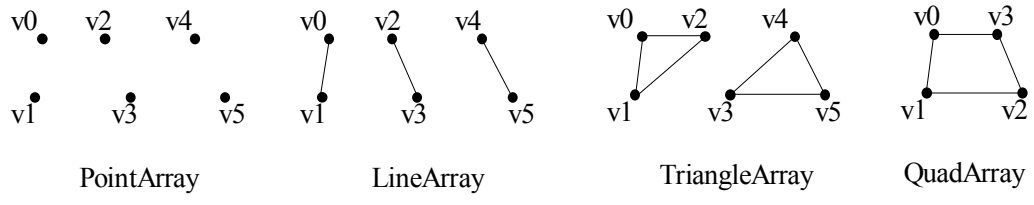


Figure 2-13 GeometryArray Subclasses

By default, the interiors of triangles and quadrilaterals are filled. In later sections, you will learn that attributes can influence how filled primitives can be rendered in different ways.

These four subclasses inherit their constructors and methods from GeometryArray. Their constructors are listed below. For their methods, go back to the listing entitled GeometryArray Methods.

GeometryArray Subclass Constructors

Constructs an empty object with the specified number of vertices and the vertex format. The format is one or more individual flags bitwise "OR"ed together to describe the per-vertex data. The format flags are the same as defined in the GeometryArray superclass.

```
PointArray(int vertexCount, int vertexFormat)
LineArray(int vertexCount, int vertexFormat)
TriangleArray(int vertexCount, int vertexFormat)
QuadArray(int vertexCount, int vertexFormat)
```

To see the use of these constructors and methods, go back to Code Fragment 2-4, Code Fragment 2-5, and Code Fragment 2-6, which use a LineArray object.

If you are rendering quadrilaterals, be careful that the vertices do not create concave, self-intersecting, or non-planar geometry. If they do, they may not be rendered properly.

2.5.3 Subclasses of GeometryStripArray

The previously described four subclasses of GeometryArray do not allow for any reuse of vertices. Some geometric configurations invite the reuse of vertices, so specialized classes may result in better rendering performance.

The GeometryStripArray is an abstract class from which strip primitives (for creating compound lines and surfaces) are derived. GeometryStripArray is the superclass of LineStripArray, TriangleStripArray, and TriangleFanArray. Figure 2-14 shows an instance of each type of strip and how vertices are reused. The LineStripArray renders connected lines. The TriangleStripArray results in triangles that share an edge, reusing the most recently rendered vertex. The TriangleFanArray reuses the very first vertex in its strip for each triangle.

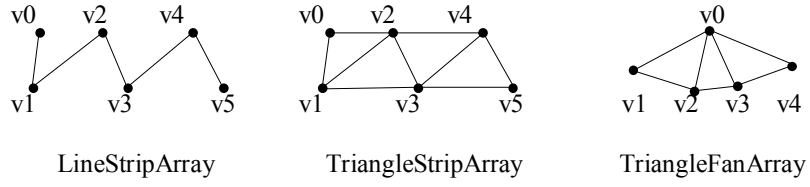


Figure 2-14 GeometryStripArray Subclasses

The GeometryStripArray has a different constructor than GeometryArray. The GeometryStripArray constructor has a third parameter, which is an array of vertex counts per strip, enabling a single object to maintain multiple strips. (GeometryStripArray also introduces a couple of querying methods, getNumStrips() and getStripVertexCounts(), which are infrequently used.)

GeometryStripArray Subclass Constructors

Constructs an empty object with the specified number of vertices, the vertex format, and an array of vertex counts per strip. The format is one or more individual flags bitwise "OR"ed together to describe the per-vertex data. The format flags are the same as defined in the GeometryArray superclass. Multiple strips are supported. The sum of the vertex counts for all strips (from the stripVertexCounts array) must equal the total count of all vertices (vtxCount).

```
LineStripArray(int vtxCount, int vertexFormat, int stripVertexCounts[])
TriangleStripArray(int vtxCount, int vertexFormat, int stripVertexCounts[])
TriangleFanArray(int vtxCount, int vertexFormat, int stripVertexCounts[])
```

Note that Java 3D does not support filled primitives with more than four sides. The programmer is responsible for using tessellators to break down more complex polygons into Java 3D objects, such as triangle strips or fans. The Triangulator utility class converts complex polygons into triangles[10].

Triangulator Class

Package: `com.sun.j3d.utils.geometry`

Used for converting non-triangular polygon geometry into triangles for rendering by Java 3D. Polygons can be concave, nonplanar, and can contain holes (see GeometryInfo.setContourCounts()). Nonplanar polygons are projected onto the nearest plane. NOTE: See the current class documentation for limitations. See Section 3.3 of this tutorial for more information.

Triangulator Constructor Summary

`Triangulator()`
Create a Triangulator object.

Triangulator Method Summary

`void triangulate(GeometryInfo ginfo)`
This routine converts the GeometryInfo object from primitive type POLYGON_ARRAY to primitive type TRIANGLE_ARRAY using polygon decomposition techniques.

Parameters:
ginfo - `com.sun.j3d.utils.geometry.GeometryInfo` to be triangulated.

Example of usage:

```
Triangulator tr = new Triangulator();
tr.triangulate(ginfo);                // ginfo contains the geometry
shape.setGeometry(ginfo.getGeometryArray());   // shape is a Shape3D
```

Yo-yo Code Demonstrates TriangleFanArray

The Yoyo object in the `YoyoApp.java` program shows how to use a TriangleFanArray object to model the geometry of a yo-yo. The TriangleFanArray contains four independent fans: two exterior faces (circular disks) and two internal faces (cones). Only one TriangleFanArray object is needed to represent the four fans.

[10] The Triangulator class and related classes are explained in more detail in Chapter 3.

Figure 2-15 shows three renderings of the TriangleFanArray. The first view shows its default rendering, as white, filled polygons. However, it's hard to see detail, especially the location of the vertices. To show the triangles better, the other two views show the TriangleFanArray with its vertices connected with lines. To render what would be filled polygons as lines, see the class PolygonAttributes in Section 2.6.

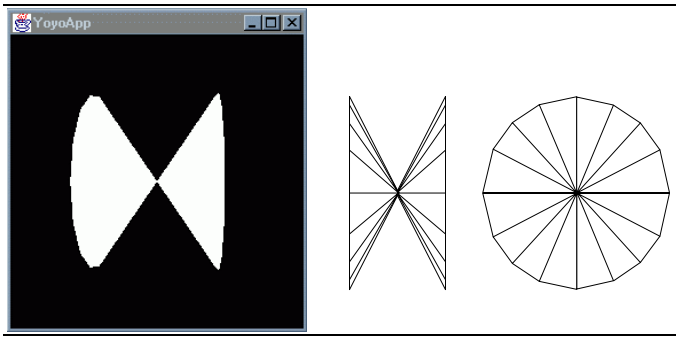


Figure 2-15 Three Views of the Yo-yo

In Code Fragment 2-7, the method `yoyoGeometry()` creates and returns the desired TriangleFanArray. Lines 15-18 calculates the central points for all four fans. Each fan has 18 vertices, which are calculated in lines 20-28. Lines 30-32 construct the empty TriangleFanArray object, and then line 34 is where the previously calculated coordinate data (from lines 15-28) is stored into the object.

```
1. private Geometry yoyoGeometry() {
2.
3.      TriangleFanArray tfa;
4.      int     N = 17;
5.      int     totalN = 4*(N+1);
6.      Point3f coords[] = new Point3f[totalN];
7.      int     stripCounts[] = {N+1, N+1, N+1, N+1};
8.      float   r = 0.6f;
9.      float   w = 0.4f;
10.     int     n;
11.     double  a;
12.     float   x, y;
13.
14.     // set the central points for four triangle fan strips
15.     coords[0*(N+1)] = new Point3f(0.0f, 0.0f, w);
16.     coords[1*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);
17.     coords[2*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);
18.     coords[3*(N+1)] = new Point3f(0.0f, 0.0f, -w);
19.
20.     for (a = 0,n = 0; n < N; a = 2.0*Math.PI/(N-1) * ++n){
21.          x = (float) (r * Math.cos(a));
22.          y = (float) (r * Math.sin(a));
23.
24.          coords[0*(N+1)+N-n] = new Point3f(x, y, w);
25.          coords[1*(N+1)+n+1] = new Point3f(x, y, w);
26.          coords[2*(N+1)+N-n] = new Point3f(x, y, -w);
27.          coords[3*(N+1)+n+1] = new Point3f(x, y, -w);
28.     }
29.
30.     tfa = new TriangleFanArray (totalN,
31.                                 TriangleFanArray.COORDINATES,
32.                                 stripCounts);
33.
34.     tfa.setCoordinates(0, coords);
```

```
35.
36.     return tfa;
37.} // end of method yoyoGeometry in class Yoyo
```

Code Fragment 2-7 yoyoGeometry() Method Creates TriangleFanArray Object

The all white yo-yo is just a starting point. Figure 2-16 shows a similar object, modified to include colors at each vertex. The modified `yoyoGeometry()` method, which includes colors in the TriangleFanArray object, is shown in Code Fragment 2-8. Lines 23 through 26, 36 through 39, and line 46 specify color values for each vertex.

More possibilities exist for specifying the appearance of a visual object through the use of lights, textures, and material properties of a visual object. These topics are not covered in this tutorial module. Lights and textures are the topics of tutorial module 2.

```
1. private Geometry yoyoGeometry() {
2.
3.      TriangleFanArray tfa;
4.      int     N = 17;
5.      int     totalN = 4*(N+1);
6.      Point3f coords[] = new Point3f[totalN];
7.      Color3f colors[] = new Color3f[totalN];
8.      Color3f red = new Color3f(1.0f, 0.0f, 0.0f);
9.      Color3f yellow = new Color3f(0.7f, 0.5f, 0.0f);
10.     int     stripCounts[] = {N+1, N+1, N+1, N+1};
11.     float   r = 0.6f;
12.     float   w = 0.4f;
13.     int     n;
14.     double  a;
15.     float   x, y;
16.
17.     // set the central points for four triangle fan strips
18.     coords[0*(N+1)] = new Point3f(0.0f, 0.0f, w);
19.     coords[1*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);
20.     coords[2*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);
21.     coords[3*(N+1)] = new Point3f(0.0f, 0.0f, -w);
22.
23.     colors[0*(N+1)] = red;
24.     colors[1*(N+1)] = yellow;
25.     colors[2*(N+1)] = yellow;
26.     colors[3*(N+1)] = red;
27.
28.     for(a = 0,n = 0; n < N; a = 2.0*Math.PI/(N-1) * ++n){
29.          x = (float) (r * Math.cos(a));
30.          y = (float) (r * Math.sin(a));
31.          coords[0*(N+1)+n+1] = new Point3f(x, y, w);
32.          coords[1*(N+1)+N-n] = new Point3f(x, y, w);
33.          coords[2*(N+1)+n+1] = new Point3f(x, y, -w);
34.          coords[3*(N+1)+N-n] = new Point3f(x, y, -w);
35.
36.          colors[0*(N+1)+N-n] = red;
37.          colors[1*(N+1)+n+1] = yellow;
38.          colors[2*(N+1)+N-n] = yellow;
39.          colors[3*(N+1)+n+1] = red;
40.     }
41.     tfa = new TriangleFanArray (totalN,
42.                     TriangleFanArray.COORDINATES|TriangleFanArray.COLOR_3,
43.                     stripCounts);
44.
```

```
45.     tfa.setCoordinates(0, coords);
46.     tfa.setColors(0,colors);
47.
48.     return tfa;
49.} // end of method yoyoGeometry in class Yoyo
```

Code Fragment 2-8 Modified yoyoGeometry() Method with Added Colors

The observant reader will notice the differences in lines 36 through 39. The code is written to make the front face of each triangle in the geometry the outside of the yo-yo. The discussion of front and back triangle faces, and why it makes a difference is in Section 2.6.4.

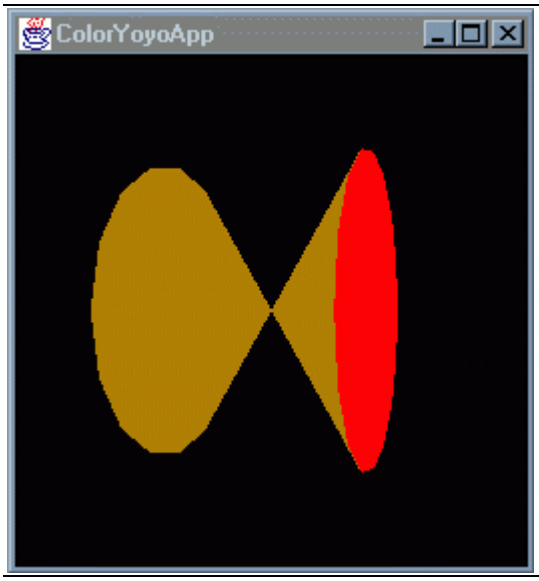


Figure 2-16 Yo-yo with Colored Filled Polygons

2.5.4 Subclasses of IndexedGeometryArray

The previously described subclasses of GeometryArray declare vertices wastefully. Only the GeometryStripArray subclasses have even limited reuse of vertices. Many geometric objects invite reuse of vertices. For example, to define a cube, each of its eight vertices is used by three different squares. In a worse case, a cube requires specifying 24 vertices, even though only eight unique vertices are needed (16 of the 24 are redundant).

IndexedGeometryArray objects provide an extra level of indirection, so redundant vertices may be avoided. Arrays of vertex-based information must still be provided, but the vertices may be stored in any order, and any vertex may be reused during rendering. We call these arrays, containing the coordinates, colors, surface normals, and texture coordinates, the "data arrays."

However, IndexedGeometryArray objects also need additional arrays ("index arrays") that contain indices into the "data arrays." There are up to four "index arrays": coordinate indices, color indices, surface normal indices, and texture coordinate indices, which corresponds to the "data arrays." The number of index arrays is always the same as the number of data arrays. The number of elements in each index array is the same and typically larger than the number of elements in each data array.

The "index arrays" may have multiple references to the same vertex in the "data arrays." The values in these "index arrays" determine the order in which the vertex data is accessed during rendering. Figure 2-17 shows the relationships between index and data coordinate arrays for a cube as an example.

It is worth mentioning that there is a price to pay for the reuse of vertices provided by indexed geometry – you pay for it in performance. The indexing of geometry at render time adds more work to the rendering process. If performance is an issue, use strips whenever possible and avoid indexed geometry. Indexed geometry is useful when speed is not critical and there is some memory to be gained through indexing, or when indexing provides programming convenience.

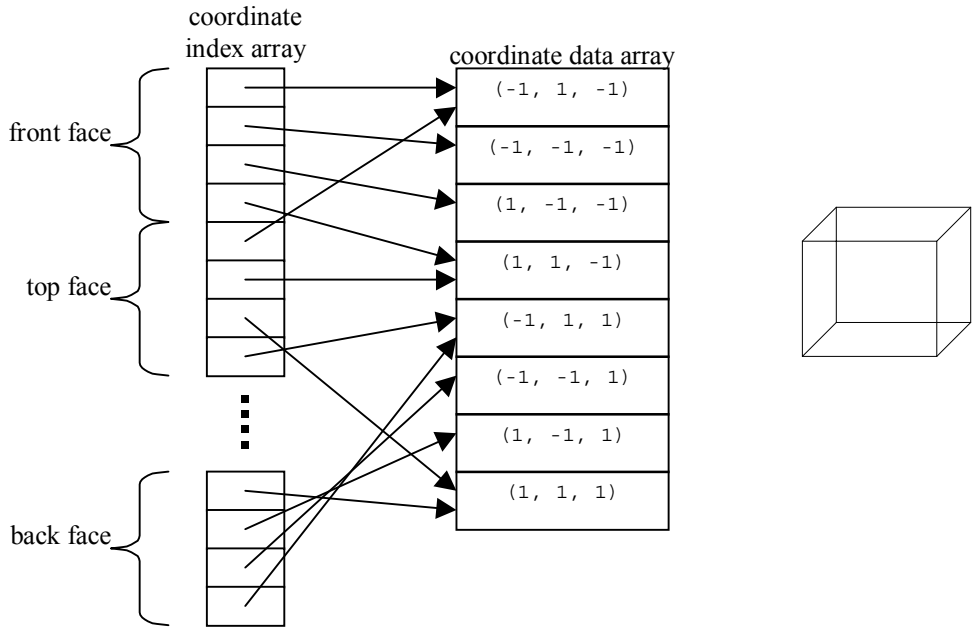


Figure 2-17 Index and Data Arrays for a Cube

Subclasses of IndexedGeometryArray parallel the subclasses of GeometryArray. The class hierarchy of IndexedGeometryArray is shown in Figure 2-18.

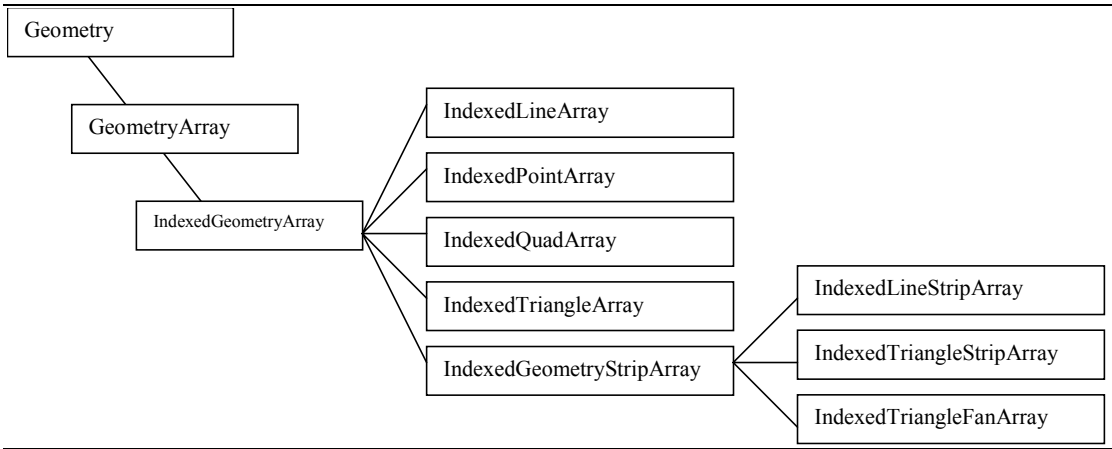


Figure 2-18 IndexedGeometryArray Subclasses

The constructors for IndexedGeometryArray, IndexedGeometryStripArray, and their subclasses are similar to constructors for GeometryArray and GeometryStripArray. The classes of indexed data have an additional parameter to define how many indices are used to describe the geometry (the number of elements in the index arrays).

IndexedGeometryArray and Subclasses Constructors

Constructs an empty object with the specified number of vertices, vertex format, and number of indices in this array.

```
IndexedGeometryArray(int vertexCount, int vertexFormat, int indexCount)
IndexedPointArray(int vertexCount, int vertexFormat, int indexCount)
IndexedLineArray(int vertexCount, int vertexFormat, int indexCount)
IndexedTriangleArray(int vertexCount, int vertexFormat, int indexCount)
IndexedQuadArray(int vertexCount, int vertexFormat, int indexCount)
```

IndexedGeometryStripArray and Subclasses Constructors

Constructs an empty object with the specified number of vertices, vertex format, number of indices in this array, and an array of vertex counts per strip.

```
IndexedGeometryStripArray(int vc, int vf, int ic, int stripVertexCounts[])
IndexedLineStripArray(int vc, int vf, int ic, int stripVertexCounts[])
IndexedTriangleStripArray(int vc, int vf, int ic, int stripVertexCounts[])
IndexedTriangleFanArray(int vc, int vf, int ic, int stripVertexCounts[])
```

IndexedGeometryArray, IndexedGeometryStripArray, and their subclasses inherit the methods from GeometryArray and GeometryStripArray to load the "data arrays." The classes of indexed data have added methods to load indices into the "index arrays."

IndexedGeometryArray Methods (partial list)

`void setCoordinateIndex(int index, int coordinateIndex)`
Sets the coordinate index associated with the vertex at the specified index for this object.

`void setCoordinateIndices(int index, int coordinateIndices[])`
Sets the coordinate indices associated with the vertices starting at the specified index for this object.

`void setColorIndex(int index, int colorIndex)`
Sets the color index associated with the vertex at the specified index for this object.

`void setColorIndices(int index, int colorIndices[])`
Sets the color indices associated with the vertices starting at the specified index for this object.

`void setNormalIndex(int index, int normalIndex)`
Sets the normal index associated with the vertex at the specified index for this object.

`void setNormalIndices(int index, int normalIndices[])`
Sets the normal indices associated with the vertices starting at the specified index for this object.

`void setTextureCoordinateIndex(int texCoordSet, int index, int texCoordIndex)` **`<new in 1.2>`**
Sets the texture coordinate index associated with the vertex at the specified index in the specified texture coordinate set for this object.

`void setTextureCoordinateIndices(int texCoordSet, int index, int texCoordIndices[])` **`<new in 1.2>`**
Sets the texture coordinate indices associated with the vertices starting at the specified index in the specified texture coordinate set for this object.

2.5.5 Axis.java is an Example of IndexedGeometryArray

The `examples/geometry` subdirectory contains the `Axis.java` source code. This file defines the Axis visual object useful for visualizing the axis and origin in a virtual universe. It also serves as an example of indexed geometry.

The Axis object defines 18 vertices and 30 indices to specify 15 lines. There are five lines per axis used to create a simple 3D arrow.

2.6 Appearance and Attributes

Shape3D objects may reference both a Geometry and an Appearance object. As was previously discussed in Section 2.5, the Geometry object specifies the per-vertex information of a visual object. The per-vertex information in a Geometry object can specify the color of visual objects. Data in a Geometry object are often insufficient to fully describe how an object looks. In most cases, an Appearance object is also needed.

An Appearance object does not contain the information for how the Shape3D object should look, but an Appearance object knows where to find appearance data. An Appearance object (already a subclass of NodeComponent) may reference several objects of other subclasses of the NodeComponent abstract

class. Therefore, information which describes the appearance of a geometric primitive is said to be stored within an "appearance bundle," such as the one shown in Figure 2-19.

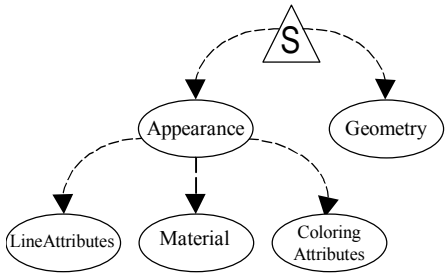


Figure 2-19 An Appearance Bundle

An Appearance object can refer to several different NodeComponent subclasses called appearance attribute objects, including:

- PointAttributes
- LineAttributes
- PolygonAttributes
- ColoringAttributes
- TransparencyAttributes
- RenderingAttributes
- Material
- TextureAttributes
- Texture
- TexCoordGeneration

The first six of the listed NodeComponent subclasses are explained in this section. Of the remaining four subclasses in the list, Material is used for lighting, and the last three are used for texture mapping. Lighting and texture mapping are discussed in Chapters 6 and 7, respectively.

An Appearance object with the attributes objects it refers to is called an appearance bundle. To reference any of these node components, an Appearance object has a method with an obvious name. For example, for an Appearance object to refer to a ColoringAttributes object, use the method `Appearance.setColoringAttributes()`. A simple code example looks like Code Fragment 2-9:

```
1. ColoringAttributes ca = new ColoringAttributes();
2. ca.setColor (1.0, 1.0, 0.0);
3. Appearance app = new Appearance();
4. app.setColoringAttributes(ca);
5. Shape3D s3d = new Shape3D();
6. s3d.setAppearance (app);
7. s3d.setGeometry (someGeomObject);
```

Code Fragment 2-9 Using Appearance and ColoringAttributes NodeComponent Objects

The scene graph that results from this code is shown in Figure 2-20.

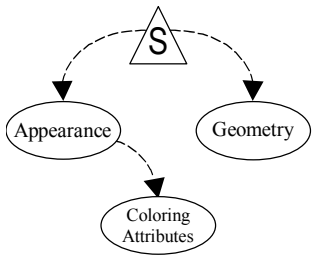


Figure 2-20 Appearance Bundle Created by Code Fragment 2-9.

2.6.1 Appearance NodeComponent

The next two reference blocks list the default constructor and other methods of the Appearance class.

Appearance Constructor

The default Appearance constructor creates an Appearance object with all component object references initialized to null. The default values, for components with null references, are generally predictable: points and lines are drawn with sizes and widths of 1 pixel and without antialiasing, the intrinsic color is white, transparency is disabled, and the depth buffer is enabled and is both read and write accessible.

```
Appearance()
```

An Appearance component usually references one or more attribute components, by calling the following methods. These attribute classes are described in Section 2.6.3.

Appearance Methods (excluding lighting and texturing)

Each method sets its corresponding NodeComponent object to be part of the current Appearance bundle.

```
void setPointAttributes(PointAttributes pointAttributes)
void setLineAttributes(LineAttributes lineAttributes)
void setPolygonAttributes(PolygonAttributes polygonAttributes)
void setColoringAttributes(ColoringAttributes coloringAttributes)
void setTransparencyAttributes(TransparencyAttributes transparencyAttributes)
void setRenderingAttributes(RenderingAttributes renderingAttributes)
```

2.6.2 Sharing NodeComponent Objects

It is legal and often desirable for several different objects to reference, and therefore share, the same NodeComponent objects. For example in Figure 2-21, two Shape3D objects reference the same Appearance component. Also, two different Appearance objects are sharing the same LineAttributes component.

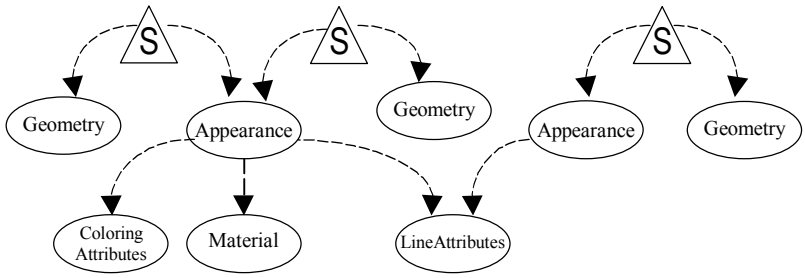


Figure 2-21 Multiple Appearance Objects Sharing a Node Component

Sharing the same NodeComponent can enhance performance. For instance, if several Appearance components share the same LineAttributes component, which enables antialiasing, the Java 3D rendering engine may decide to group the antialiased wire frame shapes together. That would minimize turning antialiasing on and off, which should be faster.

Note that it is illegal for a Node to have more than one parent. However, since NodeComponents are referenced, they aren't Node objects, so they really don't have any parents. Therefore, NodeComponent objects may be shared (referenced) by any number of other objects.

2.6.3 Attribute Classes

This section describes six of the Attribute NodeComponent subclasses that can be referenced by Appearance objects. Discussion of the TextureAttribute class is deferred to Chapter 7.

PointAttributes

PointAttributes objects manage how point primitives are rendered. By default, if a vertex is rendered as a point, it fills a single pixel. You can use setPointSize() to make a point larger. However, by default, a larger point still looks square, unless you also use setPointAntialiasingEnable(). Antialiasing points changes the colors of the pixels to make the point look "rounder" (or at least, less visibly square).

PointAttributes Constructors

`PointAttributes()`
Creates a component object that describes 1 pixel size points without antialiasing.

`PointAttributes(float pointSize, boolean state)`
Creates a component object that describes the pixel size for points and whether to enable antialiasing.

PointAttributes Methods

`void setPointSize(float pointSize)`
Describes pixel size for points.

`void setPointAntialiasingEnable(boolean state)`
Enables or disables point antialiasing. Visually interesting only if pointSize >= 1 pixel.

LineAttributes

LineAttributes objects change how line primitives are rendered in three ways. By default, a line is drawn solidly filled, one pixel wide, and without antialiasing (the smoothing effect). You can change these attributes by calling the methods setLinePattern(), setLineWidth(), and setLineAntialiasingEnable().

User-defined Line Patterns **`<new in 1.2>`**

In addition to the standard solid, dash, dot, and dot-dash choices, API v1.2 gives the application programmer the ability to define a line pattern. A line pattern is specified by a pattern mask and an optional scale factor.

A 16-bit value specifies the pattern mask. Each bit in the 16-bit value specifies whether a pixel in the pattern is on or off. A bit-value of 1 specifies that the corresponding pixel is drawn (on), while a value of 0 specifies that the corresponding pixel is not drawn (off). After all 16 bits in the pattern are used, the pattern is repeated. Pattern mask values are usually set using octal values and bit 0 in the pattern mask corresponds to the first pixel of the pattern.

For example, a mask of `0x00ff` (`0b0000000011111111`) defines a pattern of 8 pixels on followed by 8 pixels off. Remember, the least significant bit is used first in drawing; so the bit pattern is read right to left. Pattern mask value 0x0101 defines a repeating pattern of 1 pixel on and 7 pixels off.

The pattern is repeated as many times as necessary for individual line segments of a line strip primitive. The pattern restarts at the beginning of each new line strip. For line array primitives, the pattern is restarted at the beginning of each line.

The pattern can be expanded to as large as 240 pixels using the scale factor. The pattern is multiplied by the scale factor by repeating each bit in the pattern mask scale factor times. For example, a scale factor of 3 applied to a pattern mask of 0x001f would produce a repeating pattern of (3 x 5 =) 15 pixels on followed by (3 x 11 =) 33 pixels off. The valid range for the scale factor attribute is [1, 15]. Values outside this range are clamped.

LineAttributes Constructors

`LineAttributes()`
Creates a component object that describes 1 pixel wide, solidly filled lines without antialiasing.

`LineAttributes(float pointSize, int linePattern, boolean state)`
Creates a component object that describes the pixel size for lines, the pattern to use for drawing, and whether to enable antialiasing.

LineAttributes Methods

`void setLineWidth(float lineWidth)`
Describes pixel width for lines.

`void setLinePattern(int linePattern)`
where linePattern is one of the following constants: PATTERN_SOLID (the default), PATTERN_DASH, PATTERN_DOT, PATTERN_DASH_DOT, or PATTERN_USER_DEFINED.
Describes how the pixels of a line should be filled. More information on line patterns appears below.

`void setLineAntialiasingEnable(boolean state)`
Enables or disables line antialiasing.

`void setPatternMask(int mask)` **`<new in 1.2>`**
Sets the line pattern mask to the specified value.

`void setPatternScaleFactor(int scaleFactor)` **`<new in 1.2>`**
Sets the line pattern scale factor to the specified value.

Line Attributes Line Patterns

PATTERN_SOLID	solid lines (no pattern). This is the default.
PATTERN_DASH	dashed lines; ideally, a repeating pattern of 8 pixels on and 8 pixels off.
PATTERN_DOT	dotted lines; ideally, a repeating pattern of 1 pixel on and 7 pixels off.
PATTERN_DASH_DOT	dashed-dotted lines; ideally, a repeating pattern of 7 pixels on, 4 pixels off, 1 pixel on, and 4 pixels off.
PATTERN_USER_DEFINED	lines with a user-defined line pattern. See "User-defined Line Patterns," above.

PolygonAttributes

PolygonAttributes governs how polygon primitives are rendered in three ways: how the polygon is rasterized, if it is culled, and whether a special depth offset is applied. By default, a polygon is filled, but setPolygonMode() can change the polygon rasterization mode so that the polygon is drawn as wire frame (lines) or only as the points at the vertices. (In the latter two cases, the LineAttributes or PointAttributes also affect how the primitive is visualized.) The method setCullFace() may be used to reduce the number of polygons which are rendered. If setCullFace() is set to either to CULL_FRONT or CULL_BACK, on average, half the polygons are no longer rendered.

Due to efficiencies in the rendering system, vertices rendered as both wire frame and filled polygons are not always rasterized with the exact same depth values. As a result, the polygons and wire frame may render to slightly different depths, which can cause stitching. **Stitching** is the term used to describe when the wire frame of a polygon is randomly visible due to the variance in depths between the filled polygon and its wire frame. The term stitching refers to the visual effect – the wire frame appearing and disappearing around the edge of a polygon resembles a hand-sewn thread. With the `PolygonOffset`, and `PolygonOffsetFactor` attributes the depth values of the filled polygons could be nudged toward the image plate, so that the wire frame outlines the filled object properly.

`setBackFaceNormalFlip()` is useful to render a lit, filled polygon, where a both sides of the polygon are to be shaded. Chapter 6 introduces lighting; consequently, example programs that shade both sides of polygons appear in that chapter.

PolygonAttributes Constructors

`PolygonAttributes()`
Creates a component object with default filled polygons, no face culling, and no polygon offset.

`PolygonAttributes(int polygonMode, int cullFace, float polygonOffset)`
Creates a component object to render polygons as either points, lines, or filled polygons, with the specified face culling, and the specified polygon offset.

`PolygonAttributes(int polygonMode, int cullFace, float polygonOffset, boolean backFaceNormalFlip)`
Creates a component object similar to the previous constructor, but also allows specification of how front and back facing polygons are determined.

`PolygonAttributes(int polygonMode, int cullFace, float polygonOffset, boolean backFaceNormalFlip, float polygonOffsetFactor)` **`<new in 1.2>`**
Constructs PolygonAttributes object with specified values

PolygonAttributes Methods

`void setCullFace(int cullFace)`
where cullFace is one of the following: CULL_FRONT, CULL_BACK, or CULL_NONE. Cull (do not render) front facing polygons or back facing polygons, or don't cull any polygons at all.

`void setPolygonMode(int polygonMode)`
where polygonMode is one of the following: POLYGON_POINT, POLYGON_LINE, or POLYGON_FILL. Render polygons as either points, lines, or filled polygons (the default).

`void setPolygonOffset(float polygonOffset)`
where polygonOffset is the screen-space offset added to adjust the depth value of the polygon primitives.

`void setPolygonOffsetFactor(float polygonOffsetFactor)` **`<new in 1.2>`**
Sets the polygon offset factor to the specified value.

`void setBackFaceNormalFlip(boolean backFaceNormalFlip)`
where backFaceNormalFlip determines whether vertex normals of back facing polygons should be flipped (negated) prior to lighting. When this flag is set to true and back face culling is disabled, a polygon is rendered as if the polygon had two sides with opposing normals.

ColoringAttributes

ColoringAttributes controls how any primitive is colored. setColor() sets an intrinsic color, which in some situations specifies the color of the primitive. Also, setShadeModel() determines whether there is color interpolation across primitives (usually polygons and lines).

ColoringAttributes Constructors

`ColoringAttributes()`
Creates a component object using white for the intrinsic color and SHADE_GOURAUD as the default shading model.

`ColoringAttributes(Color3f color, int shadeModel)`

`ColoringAttributes(float red, float green, float blue, int shadeModel)`
where shadeModel is one of SHADE_GOURAUD, SHADE_FLAT, FASTEST, or NICEST. Both constructors create a component object using parameters to specify the intrinsic color and shading model. (In most cases, FASTEST is also SHADE_FLAT, and NICEST is also SHADE_GOURAUD.)

ColoringAttributes Methods

`void setColor(Color3f color)`

`void setColor(float red, float green, float blue)`
Both methods specify the intrinsic color.

`void setShadeModel(int shadeModel)`
where shadeModel is one of the following constants: SHADE_GOURAUD, SHADE_FLAT, FASTEST, or NICEST. Specifies the shading model for rendering primitives.

Since colors can also be defined at each vertex of a Geometry object, there may be a conflict with the intrinsic color defined by ColoringAttributes. In case of such a conflict, the colors defined in the Geometry object overrides the ColoringAttributes intrinsic color. Also, if lighting is enabled, the ColoringAttributes intrinsic color is ignored altogether.

TransparencyAttributes

TransparencyAttributes manages the transparency of any primitive. setTransparency() defines the transparency value (where 0.0 denotes fully opaque and 1.0 fully transparent) for the primitive. `setTransparencyMode()` enables transparency and selects the rasterization technique used to produce transparency. The transparency mode choices include SCREEN_DOOR, BLENDED, and NONE.

SCREEN_DOOR transparency is achieved by selecting some pixels to be fully transparent and others fully opaque. The percentage of pixels considered fully transparent is approximately equal to the value specified by the transparency parameter. BLENDED transparency is achieved by having all pixels be pariri ally transparent in the percentage specified by the transparency parameter. The blend equation is specified by the srcBlendFunction and dstBlendFunction attributes.

SCREEN_DOOR transparency is faster than BLENDED at the expense of image quality. However, SCREEN_DOOR transparency is of suitable quality in many applications.

TransparencyAttributes Constructors

`TransparencyAttributes()`
Creates a component object with the transparency mode of NONE.

`TransparencyAttributes(int tMode, float tVal)`
where tMode is one of BLENDED, SCREEN_DOOR, FASTEST, NICEST, or NONE, and tVal specifies the object's opacity (where 0.0 denotes fully opaque and 1.0, fully transparent). Creates a component object with the specified method for rendering transparency and the opacity value of the object's appearance.

`TransparencyAttributes(int tMode, float tVal, int srcBlendFunction, int dstBlendFunction)` **`<new in 1.2>`**
Construct TransparencyAttributes object with specified values.

TransparencyAttributes Methods

`void setTransparency(float tVal)`
where tVal specifies an object's transparency (where 0.0 denotes fully opaque and 1.0 fully transparent).

`void setTransparencyMode(int tMode)`
where tMode (one of BLENDED, SCREEN_DOOR, FASTEST, NICEST, or NONE) specifies if and how transparency is performed.

`void setDstBlendFunction(int blendFunction)` **`<new in 1.2>`**
Sets the destination blend function used in blended transparency and antialiasing operations.

`void setSrcBlendFunction(int blendFunction)` **`<new in 1.2>`**
Sets the source blend function used in blended transparency and antialiasing operations.

RenderingAttributes

RenderingAttributes controls two different per-pixel rendering operations: the depth buffer test and the alpha test. setDepthBufferEnable() and setDepthBufferWriteEnable() determine whether and how the depth buffer is used for hidden surface removal. setAlphaTestValue() and setAlphaTestFunction() determine whether and how the alpha test function is used.

Raster operation determine how rendered pixels affect the image plate. The default (disabled) will be used most of the time. Raster operation settings are primarily useful in rendering to the front buffer in immediate mode. For more information on raster operations and immediate mode refer to the API Specificaiton.

A color can be specified per vertex in a Geometry object. These vertex colors can be ignored for the Geometry objects associated with this RenderingAttributes object. If ignoreVertexColors is true, per-vertex colors defined in the Geometry are ignored. In this case, if lighting is enabled, the Material diffuse color will be used as the object color. If lighting is disabled, the ColoringAttributes color is used. The default value is false.

Visual objects are made invisible using the Visibility flag. When the Visibility flag is false, visual objects are not rendered, but they still exist and; therefore, can be picked or collided with. This flag is set with the setVisible method. By default, the visibility flag is true.

The **depth buffer** is the collection of depth values for rendered pixels. It is used to determine the visibility or occlusion of pixels as they are rendered. The depth buffer is used in differently when rendering opaque and transparent objects as transparent objects do not occlude opaque objects. Consequently, transparent objects do not normally update the depth buffer.

The Depth buffer can be enabled or disabled for this RenderingAttributes component object. Disabling the depth buffer ensures that an object is always visible, regardless of any occlusion that would have normally occurred. The setDepthBufferWriteEnable method enables or disables writing the depth buffer for this object. By default, the depth buffer is enabled and the depth buffer write is enabled. Refer to the API Specification for more information on the role of the depth buffer in rendering transparent and antialiased objects.

RenderingAttributes Constructors

```
RenderingAttributes()
```
Creates a component object which defines per-pixel rendering states with enabled depth buffer testing and disabled alpha testing.

```
RenderingAttributes(boolean depthBufferEnable, boolean depthBufferWriteEnable,
                    float alphaTestValue, int alphaTestFunction)
```
where depthBufferEnable turns on and off the depth buffer comparisons (depth testing), depthBufferWriteEnable turns on and off writing to the depth buffer, alphaTestValue is used for testing against incoming source alpha values, and alphaTestFunction is one of ALWAYS, NEVER, EQUAL, NOT_EQUAL, LESS, LESS_OR_EQUAL, GREATER, or GREATER_OR_EQUAL, which denotes what type of alpha test is active. Creates a component object which defines per-pixel rendering states for depth buffer comparisons and alpha testing.

```
RenderingAttributes(boolean depthBufferEnable, boolean depthBufferWriteEnable,
                    float alphaTestValue, int alphaTestFunction,
                    boolean visible,  boolean ignoreVertexColors,
                    boolean rasterOpEnable, int rasterOp)          <new in 1.2>
```
Constructs a RenderingAttributes object with specified values

RenderingAttributes Methods

`void setDepthBufferEnable(boolean state)`
turns on and off the depth buffer testing.

`void setDepthBufferWriteEnable(boolean state)`
turns on and off writing to the depth buffer.

`void setAlphaTestValue(float value)`
specifies the value to be used for testing against incoming source alpha values.

`void setAlphaTestFunction(int function)`
where function is one of ALWAYS, NEVER, EQUAL, NOT_EQUAL, LESS, LESS_OR_EQUAL, GREATER, or GREATER_OR_EQUAL, which denotes what type of alpha test is active. If function is ALWAYS (the default), then the alpha test is effectively disabled.

`void setDepthBufferEnable(boolean state)` **`<new in 1.2>`**
Enables or disables depth buffer mode for this RenderingAttributes component object.

`void setDepthBufferWriteEnable(boolean state)` **`<new in 1.2>`**
Enables or disables writing the depth buffer for this object.

`void setIgnoreVertexColors(boolean ignoreVertexColors)` **`<new in 1.2>`**
Sets a flag that indicates whether vertex colors are ignored for this RenderingAttributes object.

`void setRasterOp(int rasterOp)` **`<new in 1.2>`**
Sets the raster operation function for this RenderingAttributes component object.

`void setRasterOpEnable(boolean rasterOpEnable)` **`<new in 1.2>`**
Sets the rasterOp enable flag for this RenderingAttributes component object.

`void setVisible(boolean visible)` **`<new in 1.2>`**
Sets the visibility flag for this RenderingAttributes component object.

Appearance Attribute Defaults

The default Appearance constructor initializes an Appearance object with all attribute references set to null. Table 2-1 lists the default values for those attributes with null references.

Table 2-1 Attribute Defaults

Attributes Class	Parameter	Default Value
ColoringAttributes	color	white (1, 1, 1)
	shade model	SHADE_GOURAUD
LineAttributes	line width	1.0
	line pattern	PATTERN_SOLID
	line antialiasing enable	false
PointAttributes	point size	1.0
	point antialiasing enable	false
PolygonAttributes	cull face	CULL_BACK
	backFaceNormalFlip	false

	polygon mode	POLYGON_FILL
	polygonOffset	0.0
	polygonOffsetFactor	0.0
RenderingAttributes	depthBufferEnable	true
	depthBufferWriteEnable	true
	alphaTestFunction	ALWAYS
	alphaTestValue	0.0
	visible	true
	ignoreVertexColors	false
	rasterOpEnable	false
	rasterOp	ROP_COPY
TextureAttributes	textureMode	REPLACE
	textureBlendColor	black (0, 0, 0, 0)
	transform	identity
	perspectiveCorrectionMode	NICEST
	textureColorTable	null
TransparencyAttributes	transparencyMode	NONE
	transparencyValue	0.0
	srcBlendFunction	SRC_BLEND_ALPHA
	dstBlendFunction	BLEND_ONE_MINUS_ALPHA

2.6.4 Example: Back Face Culling

Polygons have two faces. For many visual objects, only one face of the polygons need be rendered. To reduce the computational power required to render the polygonal surfaces, the renderer can cull the unneeded faces. The culling behavior is defined on a per visual object basis in the PolygonAttribute component of Appearance. The front face of an object is the face for which the vertices are defined in counter-clockwise order.

`TwistStripApp.java` creates a 'twisted strip' visual object and rotates it about the y-axis. As the twisted strip rotates, parts of it seemed to disappear. The missing pieces are easily noticed Figure 2-22.

Actually, TwistStripApp defines two visual objects, each with the same geometry - that of a Twisted strip. One of the visual objects renders as a wireframe, the other as a solid surface. Since the two visual objects have the same location and orientation, the wireframe visual object is only visible when the surface is not visible.

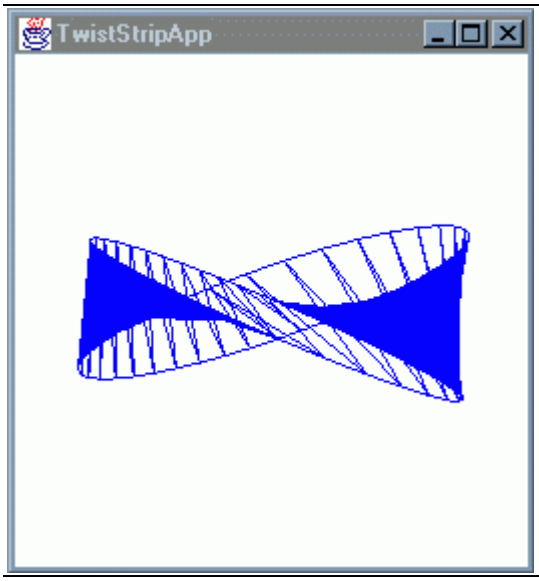


Figure 2-22 Twisted Strip with Back Face Culling

The reason for the missing polygons is the culling mode hasn't been specified, so it defaults to CULL_BACK. The triangles of the surface disappear when their back side (back face) face the image plate. This is a feature that allows the rendering system to ignore rendering triangle surfaces that are unnecessary, unwanted, or both.

However, sometimes back face culling is a problem, as in the TwistStripApp. The problem has a simple solution: turn off culling. To do this, create an Appearance component that references a PolygonAttributes component which disables culling, as shown in Code Fragment 2-10.

```
1. PolygonAttributes polyAppear = new PolygonAttributes();
2. polyAppear.setCullFace(PolygonAttributes.CULL_NONE);
3. Appearance twistAppear = new Appearance();
4. twistAppear.setPolygonAttributes(polyAppear);
5. //  several lines later, after the twistStrip TriangleStripArray has
6. //  been defined, create a Shape3D object with culling turned off
7. //  in the Appearance bundle, and add the Shape3D to the scene graph
8. twistBG.addChild(new Shape3D(twistStrip, twistAppear));
```

Code Fragment 2-10 Disable Back Face Culling for the Twisted Strip

In Figure 2-23, disabling back face culling clearly fills in the cracks. Now all polygons are rendered, no matter which direction they are facing.

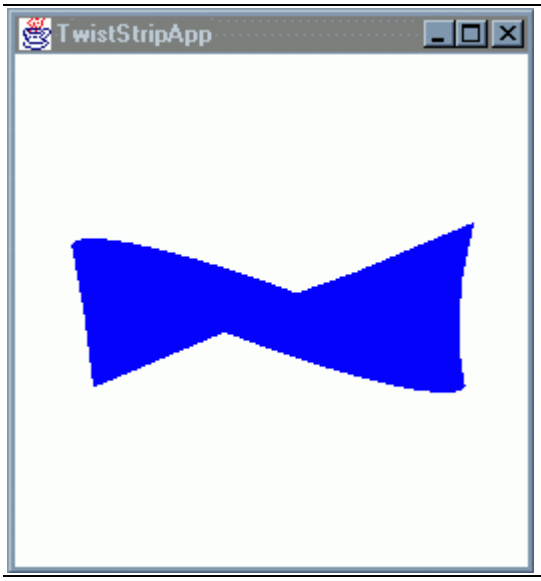


Figure 2-23 Twisted Strip without Back Face Culling

The front face of a polygon is the side for which the vertices are appear in counter-clock wise order. This is often referred to as the "right-hand rule" (see the glossary). The rule used to determine the front face of a geometric strip (i.e., triangle strip, quad strip) alternates for each element in the strip. Figure 2-24 shows examples of using the right-hand rule for front face determination.

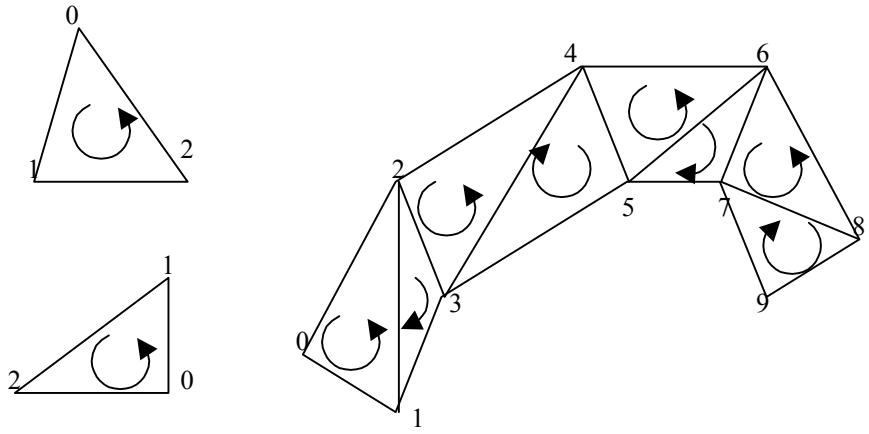


Figure 2-24 Determining the Front Face of Polygons and Strips

2.7 Bounds and Scope

As seen in the previous sections on Appearance and Attribute node components, it is often desirable to share node components. Node component objects are easily shared through multiple references from the scene graph. There are Nodes that could be shared as well. Light, sound, and fog nodes often are shared across some region of the virtual world. However, since nodes are not referenced by other scene graph objects, the technique of having multiple references to specify application of a node does not work for Node objects.

Probably the most easily understood example of sharing a node is in sharing a single light source. A light source may illuminate a number of visual objects (the rendering of those visual objects is affected by the properties of the light source). The definition of which visual objects are affected by a light source are made in two ways: bounds and scope.

Bounds defines a space, or region, for which a node is applied. Bounds are defined using Bounds class objects such as BoundingSphere (see Section 1.9), BoundingBox, or BoundingPolytope. To continue the lighting example, each light source defines a region of influence or **influencing bounds**. If any part of a visual object intersects a light source's influencing bounds, then the visual object can be affected by the light[11].

The concept of bounds is touched upon in Section 1.9 where it is explained that each behavior object has a **scheduling bounds**. A behavior is only active when the bounds of the behavior object intersects the view's activation volume.

Scope specifies a group of scene graph object by their position in the scene graph. Specifically, a scope is collection of Group objects (e.g., BranchGroup) for which the node applies. Certain nodes can specify which other scene graph objects they affect by specifying a scope.

While bounds and scope provides similar functionality, they function in different ways. A bounds specification is independent of an object's position in the scene graph. The specification of scope is independent of a visual object's position in the virtual universe. The use of bounds is mandatory for all nodes while a scope is optional. Specifying a bounds adds objects of Bounds class to a scene graph while a scope neither adds objects or is a class.

Bounds and scope are used for many applications. In addition to the lighting and behavior uses, bounds and scope are used for fog, sound, and, as presented later in this chapter, for AlternativeAppearance nodes.

2.7.1 Bounds Node Components

Bounds are used with lights, behaviors, backgrounds, and a variety of other applications in Java 3D. As a Node Component, a Bounds object is referenced by the object that needs to specify a bounds. Bounds allow the programmer to vary action, appearance, and/or sound over the virtual landscape. Bounds specification also allows the Java 3D rendering system to perform execution culling, thereby improving rendering performance[12].

Bounding regions are defined by one of a few classes. Bounds is the abstract base class for bounds classes. Bounds objects define a convex, closed volume used for various intersection and culling operations. Bounds is extended by three classes: BoundingBox, BoundingPolytope, and BoundingSphere. This section describes each of these classes.

When the selection of a bounds object is not dictated by the details of a particular bounds application, keep in mind there small performance differences in performing bounds intersection calculations. BoundingSpere is the easiest and BoundingPolytope is the hardest. If you have a large number of bounds in use, the small differences in performance add up and can make a difference in the overall performance

---

[11] The issues in determining how a visual object is rendered is more complex than in this simple explanation, especially when it comes to lighting. That is why there it is said that a visual object "can be" affected by the light. Chapter 6 is devoted to lighting.

[12] Bounds should be chosen as small as possible, while still achieving the desired effect, to reduce the computation required to render the Java 3D scene.

of an application. The bounding node classes are listed in least to most computational expense in this section.

One last, but important, general note about bounds: since smaller bounds often leads to less work, always use the smallest bounding region possible.

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

Bounds Method Summary (partial list)

Bounds is an abstract class; consequently, the listed methods are all abstract methods of Bounds and concrete methods of BoundingBox, BoundingPolytope, and BoundingSphere.

`Bounds closestIntersection(Bounds[] boundsObjects)`
Finds closest bounding object that intersects this bounding object.

`void combine(Bounds boundsObject)`
Combines this bounding object with a bounding object so that the resulting bounding object encloses the original bounding object and the given bounds object. There are other combine methods

`boolean equals(java.lang.Object bounds)`
Indicates whether the specified bounds object is equal to this Bounds object.

`boolean intersect(Bounds boundsObject)`
Test for intersection with another bounds object. There are other intersect methods.

`boolean isEmpty()`
Tests whether the bounds is empty.

`void set(Bounds boundsObject)`
Sets the value of this Bounds object.

`void transform(Transform3D trans)`
Transforms this bounding object by the given matrix.

BoundingSphere

Section 1.9.3 introduces the BoundingSphere class. BoundingSphere is the simplest definition of a bounding region. With a center point and a radius, a bounding sphere object defines spherical bounding region in the local coordinate system.

The default constructor creates a BoundingSphere centered at the local origin with a radius of 1. The constructor listed second creates a BoundingSphere using a specified center point and radius. The last two constructors create a BoundingSphere the encloses that specified Bounds or array of Bounds, respectively.

BoundingSphere Constructor Summary (partial list)

`BoundingSphere()`
Constructs and initializes a BoundingSphere with radius = 1 at 0 0 0.

`BoundingSphere(Point3d center, double radius)`
Constructs and initializes a BoundingSphere from a center and radius.

`BoundingSphere(Bounds boundsObject)`
Constructs a BoundingSphere from the specified bounds object.

`BoundingSphere(Bounds[] boundsObjects)`
Constructs a BoundingSphere from the specified array of bounds objects.

BoundingSphere Method Summary (partial list)

Consult the Bounds Method reference block (page 2-50), or the API Specification, for additional methods.

`void setCenter(Point3d center)`
Sets the position of this bounding sphere from a point.

`void setRadius(double r)`
Sets the radius of this bounding sphere from a double.

BoundingBox

A bounding box is a rectilinear space defined in the local coordinate system. A bounding box is defined by two points. One point is the combination of the minimum x, minimum y, and minimum z coordinates of the bounding region. The other point is the maximum for x, y, and z. The side of the bounding box region are parallel to the axis in the local coordinate system. This constraint on the BoundingBox makes bounds intersection calculations easier. Of course, you can rotate the local coordinate system to achieve a desired orientation of a BoundingBox object in some other coordinate system.

The following reference block lists the constructors for BoundingBox. The default constructor creates a BoundingBox that is a cube centered at the local origin with edge length of 2. The constructor listed second creates a BoundingBox using the specified minimum (lower/left/back) and maximum (upper/right/front) points. The last two constructors create a BoundingBox that encloses the specified Bounds or array of Bounds, respectively.

BoundingBox Constructor Summary

This class defines an axis aligned bounding box which is used for bounding regions.

`BoundingBox()`
Constructs and initializes a bounding box that is a 2 meter cube centered about the origin.

`BoundingBox(Point3d lower, Point3d upper)`
Constructs and initializes a BoundingBox given min and max in x, y, and z.

`BoundingBox(Bounds boundsObject)`
Constructs a BoundingBox from a bounding object.

`BoundingBox(Bounds[] bounds)`
Constructs a BoundingBox from an array of bounding objects.

BoundingBox Method Summary (partial list)

Consult the Bounds Method reference block (page 2-50), or the API Specification, for additional methods.

`void setLower(double xmin, double ymin, double zmin)`
Sets the lower corner of this bounding box.

`void setLower(Point3d p1)`
Sets the lower corner of this bounding box. There is a corresponding get method with the same interface.

`void setUpper(double xmax, double ymax, double zmax)`
Sets the upper corner of this bounding box.

`void setUpper(Point3d p1)`
Sets the upper corner of this bounding box. There is a corresponding get method with the same interface.

BoundingPolytope

A BoundingPolytope defines a polyhedral bounding region using the intersection of four or more half spaces. A half space is a region of space bounded on one side by an infinite plane. Consequently, the definition of a BoundingPolytope is made by listing the planes that create a closed convex region that is the polytope. See Appendix D for information regarding the specification of a plane by a Tuple4* object.

The default constructor creates a BoundingPolytope that is a cube centered at the local origin with edge length of 2. The constructor listed second creates a BoundingPolytope using a specified set of planes. The last two constructors create a BoundingPolytope that encloses the specified Bounds or array of Bounds, respectively. In all of the constructors except the second, the BoundingPolytope object is usually created with six planes forming a rectilinear bounding region. Once a BoundingPolytope is created, the number of its planes can not be changed.

BoundingPolytope Constructor Summary

`BoundingPolytope()`
Constructs a BoundingPolytope and initializes it to a set of 6 planes that defines a cube such that -1 <= x, y, z <= 1.

`BoundingPolytope(Vector4d[] planes)`
Constructs a BoundingPolytope using the specified planes.

`BoundingPolytope(Bounds boundsObject)`
Constructs a BoundingPolytope that circumscribes the specified bounds object.

`BoundingPolytope(Bounds[] boundsObjects)`
Constructs a BoundingPolytope that circumscribes the specified array of bounds objects.

BoundingPolytope Method Summary (partial list)

Consult the Bounds Method reference block (page 2-50), or the API Specification, for additional methods.

`int getNumPlanes()`
Returns the number of planes for this BoundingPolytope objects. There is no setNumPlanes method.

`void getPlanes(Vector4d[] planes)`
Returns the equations of the bounding planes for this bounding polytope.

`void setPlanes(Vector4d[] planes)`
Sets the bounding planes for this polytope.

2.7.2 BoundingLeaf Node

The typical bounds specification utilizes a Bounds object to specify a bounding region. In the resulting scene graph the Bounds object moves with the object that references it. This is fine for many applications; however, there may be situations in which it is desirable to have the bounding region move independently of the object using the bounds.

For example, if a world includes a stationary light source that illuminates moving objects, the bounds for the light must include the moving object. One way to handle this would be to make the bounds large enough to include all of the places the object moves. This is not the best answer in most cases. A better solution is to use a BoundingLeaf. Placed in the scene graph with the visual object, the BoundingLeaf object moves with the visual object and independently of the light source. Figure 2-25 shows a scene graph in which a light uses a BoundingLeaf node.

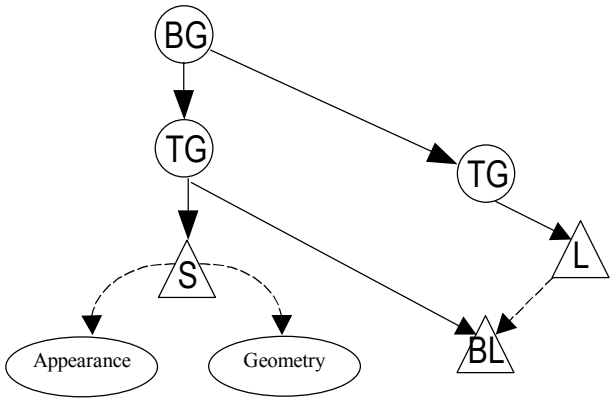


Figure 2-25 BoundingLeaf Moves with a Visual Object and Independently of a Light Source

While the applications for BoundingLeaf include the ApplicationBounds of Background objects (Section 3.5), the SchedulingBounds of Behaviors (Section 4.2), and the InfluencingBounds of Lights (Section 6.6), it would not make sense to repeat this information in all three places[13].

One interesting application of a BoundingLeaf object places a BoundingLeaf in the viewPlatform. This BoundingLeaf can be used for an "always on" scheduling bounds for a Behavior, or "always applies" application bounds for Background or Fog.

Code Fragment 2-11 presents an example of adding a BoundingLeaf as a child of the PlatformGeometry to provide an "always applies" bounds for a Background[14]. In this code, the standard `createSceneGraph()` method is modified to takes a single parameter, that of the SimpleUniverse object[15]. This is necessary for creating a PlatformGeometry object.

Lines 2, 3, and 4, create the BoundingLeaf object, create the PlatformGeometry object, and make the BoundingLeaf object a child of the PlatformGeometry, in that order. If there were to be more to the PlatformGeometry, it would be added at this point. The PlatformGeometry object is then added to the view branch graph in line 6.

The BoundingLeaf object is set as the application bounds for the background object on line 11. This same BoundingLeaf can be used for other purposes in this program. For example, it can also be used for behaviors. Note that using the BoundingLeaf in this program as the InfluencingBoundingLeaf of a light would not make the light influence all objects in the virtual world.

[13] BoundingLeaf nodes are also useful for light, clip, fog, sound, and soundscape objects, some of which are covered in this tutorial.

[14] PlatformGeometry moves with the viewer, so it is the appropriate parent for geometry associated with a viewer. For example, if the viewer is to be riding in a vehicle, it would be appropriate to make the geometry that represents the instrumentation of the vehicle a child of PlatformGeometry.

[15] The createSceneGraph() method is only standard in that it appears in many of the examples of this tutorial.

```
1.    void createSceneGraph (SimpleUniverse su) {
2.        BoundingLeaf boundingLeaf = new BoundingLeaf();
3.        PlatformGeometry platformGeom = new PlatformGeometry();
4.        platformGeom.addChild(boundingLeaf);
5.        platformGeom.compile();
6.        simpleUniv.getViewingPlatform().setPlatformGeometry(platformGeom);
7.
8.        BranchGroup contentRoot = new BranchGroup();
9.
10.       Background backg = new Background(1.0f, 1.0f, 1.0f);
11.       backg.setApplicationBoundingLeaf(boundingLeaf);
12.       contentRoot.addChild(backg);
```

Code Fragment 2-11 Adding a BoundingLeaf to the View Platform for an "Always-On" Bounds

BoundingLeaf Class

The parameterless constructor for BoundingLeaf creates a bounds of a unit sphere. The other constructor allows the specification of the bounds for the BoundingLeaf object. The following reference block lists the two BoundingLeaf constructors.

BoundingLeaf Constructor Summary

The BoundingLeaf node defines a bounding region object that can be referenced by other nodes to define a region of influence, an activation region, or a scheduling region.

`BoundingLeaf()`
Constructs a BoundingLeaf node with a unit sphere object.

`BoundingLeaf(Bounds region)`
Constructs a BoundingLeaf node with the specified bounding region.

The following reference block lists the two methods of BoundingLeaf. The `get*` method is listed since its parameters are different than that of the corresponding `set*` method.

BoundingLeaf Method Summary

`Bounds getRegion()`
Retrieves this BoundingLeaf's bounding region

`void setRegion(Bounds region)`
Sets this BoundingLeaf node's bounding region.

2.7.3 Scope

As mentioned in the introduction to this section, specifying a **scope** is a way to limit the application or influence of a node. Scope is specified as a collection of Group nodes. The node that specifies the scope can influence the object of the subgraphs rooted at the Group objects of the specified scope subject to bounds limitations. When a scope is not specified for a node the node has **universal scope.** Universal scope means the object applies throughout its virtual universe – limited by its bounds.

Some nodes allow the specification of both a bounds and a scope. But, specifying scope is not an alternative to specifying a bounds. For that node to have an effect on an object, the object must be in

both the bounds and the scope of the node. An object without a specified bounds will affect no other object.

Scope is used to limit the application of a node when using the bounds alone is difficult or impossible. Recall that a bounds specifies when something applies based on virtual world geometry. It is conceivable that using just a bounds to discriminate between objects that are near each other could be difficult. In fact, there could be times when using a bounds to discriminate could be impossible. If two visual objects touch or overlap and one object should be included in the bounds and the other excluded. This situation is not uncommon as seen in the following specific example.

A scene consists of a lamp and some visual objects on a table. The lamp has a shade, so not all of the objects should be lit by the lamp. Just using a bounding volume it is very difficult to specify the proper lighting application since lit and unlit objects are near to each other.

Specifying scope limitations for the light allows easier control of the lighting in this scene. The scene graph of Figure 2-26 is one possible solution. That scene graph separates the lit and unlit visual objects into two branches of the scene graph. Specifying the group with all the lit visual objects as the scope of the light solves the problem. Note that the light also requires a bounds that includes the object to be illuminated. Usually it is unnecessary to organize a scene graph around desired scopes. Chapter 6 provides further discussion of using scope for lighting applications.

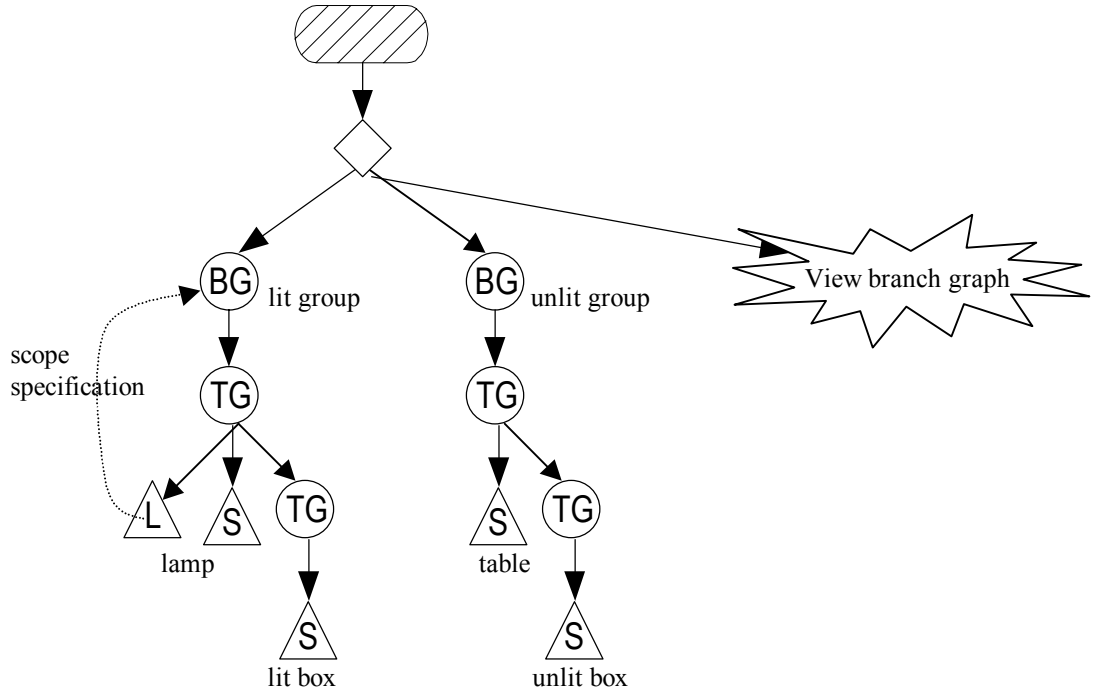


Figure 2-26 Scene Graph Showing the Scope Limited Lighting

2.8 Advanced Geometry

Java 3D API version 1.2 two make two significant enhancements to the classes used in specifying geometry. The first allows multiple geometries to be node components of a single Shape3D node. This feature can help in achieving better rendering performance.

The other new feature allows geometry data to be referenced by a Geometry object instead of being copied. Copying the data was the only mechanism in before API v1.2. Copying the data obviously takes runtime and memory when the copy of the geometry is created. Using geometry by reference eliminates the overhead of copying and may improve performance in some applications. However, the issue of performance for referenced geometry is not quite that straightforward.

Each of these new API features affect classes introduced earlier in this chapter and are addressed in the sections that follow.

2.8.1 Multiple Geometries in a Single Shape3D Node <new in 1.2>

Chapter 1 introduces the Shape3D class. Section 2.2.1 presents Shape3D as the definition of a visual object and shows the basic use of Shape3D object. API version 1.2 added a number of methods to the Shape3D class to allow Shape3D object to reference multiple Geometry NodeComponents[16]. This new feature may yield better performance due to the reduction of the number of objects in a scene graph.

In `AxisApp.java`, presented in section 2.5.1, uses three LineArray objects to draw lines to represent the x, y, and z axes. In `AxisApp.java` three Shape3D objects refer to the three geometry objects. By contrast, in `AxisApp2.java` a single Shape3D object refers to the same three geometry objects. `AxisApp2.java` has three fewer objects than `AxisApp.java`. A single Shape3D object references multiple Geometry objects in the program `examples/Geometry/AxisApp2.java`. Figure 2-27 shows the partial scene graph created by the Axis class in the `AxisApp2.java` program. Compare this scene graph with the one of Figure 2-11 (page 2-27).

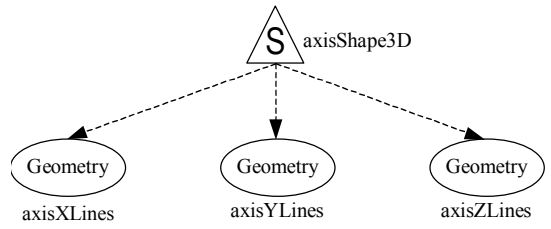


Figure 2-27 Axis Class in AxisApp.java Creates this Scene Graph

Code Fragment 2-12 shows some of the code for making an Axis object in the `AxisApp2.java`.

```
1. Shape3D axisShape3D = new Shape3D();❶
2.
3. axis.removeGeometry(0); ❷
4. axisShape3D.addGeometry(axisXLines); ❸
5. axisShape3D.addGeometry(axisYLines); ❸
6. axisShape3D.addGeometry(axisZLines); ❸
```

Code Fragment 2-12 Some Code from examples/Geometry/AxisApp2.java

Code Fragment 2-12 includes an unexpected line of code labeled as ❷. This line of code is recommended because a Shape3D object is created with its default constructor, it refers to null Geometry and null Appearance objects. This poses no problem for appearances. However, if geometry references are added to such a Shape3D object without removing the null geometry entry, then that null entry remains.

[16] The Geometry NodeComponents of one Shape3D object must be of the same *equivalence class* as later explained.

Therefore, in this example, after creating the object without a reference to geometry ❶, the null entry in the Shape3D object's list of geometry references is deleted ❷ before adding the desired geometric references ❸.

While Shape3D objects can now refer to a list of Geometry objects, the list can not be arbitrary. A single Shape3D object can only refer to geometry objects of the same geometry equivalence class. Each equivalence class includes only one 'type of primitive'. For example, the first equivalence class, point geometry, includes the two point geometry classes: PointArray and IndexedPointArray. The second equivalence class, line geometry includes the four line geometry equivalence classes LineArray, LineStripArray, and the two indexed versions of these classes. The reference block below lists the other geometry equivalence classes.

Shape3D Geometry Equivalence Classes

The Shape3D leaf node contains a list of one or more Geometry component objects and a single Appearance component object. The list of geometry objects must all be of the same equivalence class, that is, the same basic type of primitive. The equivalence classes are as follows:

Point GeometryArray: [Indexed]PointArray
Line GeometryArray: [Indexed]{LineArray, LineStripArray}
Polygon GeometryArray: [Indexed]{TriangleArray, TriangleStripArray, TriangleFanArray, QuadArray}
CompressedGeometry
Raster
Text3D

With the restriction of a Shape3D list to a single geometry equivalence class it may seem that this improvement to Shape3D is not an improvement at all. Again, the three LineArray objects defining the geometry in either of the AxisApp programs could be replaced by a single LineArray object and a single Shape3D object; so, this example is a bit contrived. However, there are cases in which this new feature is valuable.

One example is found in the polygon geometry equivalence class. This equivalence class includes geometry classes that have very different uses. The TriangleFanArray can make circles, cones, spirals, and other surfaces with curved edges, while the other geometry classes in that equivalence class are generally not used to make curved edge geometry. To be more specific, one Shape3D object is used to reference both a TriangleFanArray object and a QuadArray object in the `examples/Geometry/MultiGeomApp.java` program that makes the cart geometry shown in Figure 2-28.

Figure 2-28 Mixing TriangleFan and Quad in a Single Shape3D object in `MultiGeomApp.java`

Other equivalence classes offer real advantages as well. Before API v1.2 there was no way to have multiple Text3D or Raster objects referenced by a single Shape3D object. With API v1.2 a visual object can change its geometry by adding or removing geometry at runtime. Of course, runtime changes of geometry references require setting the proper capabilities for the Shape3D object.

The following reference block lists methods of the Shape3D class for adding and removing geometry. Section 2.2.1 lists other methods for the class.

Shape3D Method Summary (partial list, see Section 2.2.1)

`void addGeometry(Geometry geometry)` **`<new in 1.2>`**
Appends the specified geometry component to this Shape3D node's list of geometry components.

`java.util.Enumeration getAllGeometries()` **`<new in 1.2>`**
Returns an enumeration of this Shape3D node's list of geometry components.

`Geometry getGeometry(int index)` **`<new in 1.2>`**
Retrieves the geometry component at the specified index from this Shape3D node's list of geometry components.

`void insertGeometry(Geometry geometry, int index)` **`<new in 1.2>`**
Inserts the specified geometry component into this Shape3D node's list of geometry components at the specified index.

`int numGeometries()` **`<new in 1.2>`**
Returns the number of geometry components in this Shape3D node's list of geometry components.

`void removeGeometry(int index)` **`<new in 1.2>`**
Removes the geometry component at the specified index from this Shape3D node's list of geometry components.

`void setGeometry(Geometry geometry, int index)` **`<new in 1.2>`**
Replaces the geometry component at the specified index in this Shape3D node's list of geometry components with the specified geometry component.

2.8.2 GeometryArray for 'BY_REFERENCE' `<new in 1.2>`

In constructing geometry, the steps are to create the geometry object, then assign values to define the for the geometry data (i.e., coordinates, colors, normals, etc.). Prior to API v1.2, the internal operation of setting the geometry data for a Geometry object was to make a copy of the data internal to the Geometry object. API v1.2 provides an alternative to the default copying operation called geometry BY REFERENCE. To appreciate what BY_REFERENCE means, lets first look at the alternative.

When a 'copy' GeometryArray object is created, private arrays to hold all the geometry data are created 'internal' to the GeometryArray object. Then, as geometry data are assigned to the GeometryArray object, the geometry data are copied to the private geometry arrays of the GeometryArray object. Typically, any user data geometry arrays are garbage collected.

When a BY_REFERENCE GeometryArray object is created, the user creates geometry arrays and populates those arrays with geometry data. There are no geometry arrays internal to the Geometry object no copy of the geometry data made. Instead the BY_REFERENCE GeometryArray object simply stores a reference to the user array(s) of geometry data.

In comparing the two approaches, it is easy to see that copying has the disadvantages of requiring more memory (although, this is often temporary) and taking time to make the copy. However, this superficial analysis does not consider the whole runtime picture. There is a significant advantage a GeometryArray object has in keeping the geometry data private. If the geometry data is private, a GeometryArray object takes advantage of the fact that the data will not change and is, therefore, capable of more optimizations when it is compiled or live typically yielding measurable rendering performance advantages. The

increase is optimization is due to the fact that changes to the private data are not allowed, even if the user changes the data in the original arrays.

Since rendering performance is likely to be better with copied geometric data, in API version 1.2.1 the correct approach is to avoid BY_REFERENCE geometry unless there is a reason to do otherwise. The performance difference between BY_REFERENCE and copied data will be addressed in a later version of the Java 3D API.

So when should BY_REFERENCE geometry be used? There are two good reasons for using BY_REFERENCE GeometryArray:

1. if the geometry data is dynamic
2. if the geometry data requires too much memory

The most common reason for choosing to use the BY_REFERENCE functionality of a GeometryArray object is if the geometry will change. Geometry might be changed at runtime for any number of application specific reasons. In having BY_REFERENCE geometry data, the geometry may be changed even when the GeometryArray object is live or compiled (subject to capability settings). One way to change the geometry is to change the data in the referenced geometry arrays. Another way is to adjust the portion of the user arrays used by the Geometry object. Section 2.8.3 explains more of the details on changing the geometry referenced by a live or compiled GeometryArray object.

BY_REFERENCE Performance Hints and Warnings

If using BY_REFERENCE geometry, using the set*RefFloat methods is preferable to the alternatives. For example, use `setCoordRefFloat()` instead of `setCoordRef3f()`. Using an float array allows the data to be accessed directly, rather than through an array of object references. Arrays of Objects in Java are always stored by reference rather than by value, whereas arrays of native data types, such as float, are stored by value. Also, the interleaved format is likely to be more efficient than the non-interleaved format.

Keep in mind when using BY_REFERENCE geometry the methods to set the geometry are different that when it is not BY_REFERENCE. Also, the application programmer is responsible for assuring that the geometry data is valid. Specifically, all colors used in the geometry array object must be in the range [0.0,1.0]. Color values outside this range will cause undefined results. All surface normals used in the geometry array object must be unit length vectors. That is their geometric length must be 1.0. Normals that are not unit length vectors will cause undefined results.

GeometryArray BY_REFERENCE Example

Using BY_REFERENCE geometry with a GeometryArray for static geometry is not more difficult than the copied Geometry. Code Fragment 2-13, from the program `TwistByRefApp.java`, recreates the twisted strip geometry used in a previous tutorial example program, `TwistStripApp`, except this program uses referenced user geometry arrays. In this example the coordinate and color data is created in the user arrays `coord` and `color`, respectively.

The `TwistByRefApp.java` code to create the geometry shown in Code Fragment 2-13 differs little from the comparable code of `TwistStripApp`. The differences highlight the steps of creating BY_REFERENCE geometry. ❶ Declare user arrays to hold the geometric data. ❷ Construct the desired GeometryArray object with the BY_REFERENCE vertex format option. In this case the GeometryArray object is a TriangleStripArray. ❸ Fill the arrays with geometric data. ❹ Reference the user data with the GeometryArray object. The order of the steps can be changed as long as the use arrays are declared before they are referenced or filled and the GeometryArray object is constructed before the arrays are referenced.

```
1.      final Color3b red  = new Color3b(Color.red);
2.
3.      int N = 80;                     // number of vertices
4.      int stripCounts[] = {N};        // 1 strip of N verts
5.      float[] coord = new float[N*3];     ❶
6.      Color3b[] color = new Color3b[N*3]; ❶
7.
8.      // create triangle strip for twist
9.      TriangleStripArray twistStrip = new TriangleStripArray(N,
10.                                     TriangleStripArray.COORDINATES ❷
11.                                     | TriangleStripArray.BY_REFERENCE
12.                                     | TriangleStripArray.COLOR_3,
13.                                     stripCounts
14.                                     );
15.
16.     double a;
17.     int v;
18.     for(v = 0, a=0.0; v < N; v+=2, a=v*2.0*Math.PI/(N-2)){
19.         coord[v*3+0] = (float)(0.7*Math.sin(a)+0.2*Math.cos(a));
20.         coord[v*3+1] = (float)(0.3*Math.sin(a));
21.         coord[v*3+2] = (float)(0.7*Math.cos(a)+0.2*Math.cos(a));
22.         coord[v*3+3] = (float)(0.7*Math.sin(a)-0.2*Math.cos(a));     ❸
23.         coord[v*3+4] = (float)(-0.3*Math.sin(a));
24.         coord[v*3+5] = (float)(0.7*Math.cos(a)-0.2*Math.cos(a));
25.         color[v]   = red;
26.         color[v+1] = red;
27.     }
28.
29.     twistStrip.setCoordRefFloat(coord); ❹
30.     twistStrip.setColorRef3b(color);    ❹
```

Code Fragment 2-13 Creating Geometry BY_REFERENCE for the TwistStrip

GeometryArray

As explained in Section 2.5, GeometryArray is the base class for nearly all geometry classes (see Figure 2-10 Geometry Class Hierarchy on page 2-22). Consequently, the GeometryArray class defines many of the methods for managing by reference data.

To use by reference geometry, set the BY_REFERENCE bit in the vertexFormat field of the constructor for the GeometryArray object. Then instead of using the various set* methods for coordinates, normals, colors, and texture coordinates use the new set reference methods (e.g., setCoordRefFloat, setColorRefFloat, etc.). These methods are enumerated in reference blocks that follow.

GeometryArray Methods for Referencing Geometry Data (partial list)

This collection of methods set the references to geometry data in user arrays. In each case, the method will throw an exception if the data mode for this geometry array object is not BY_REFERENCE. In some cases the data mode must also be INTERLEAVED, in other cases the data mode must not be INTERLEAVED.

`void setColorRef*(Color*[] colors)` **`<new in 1.2>`**
Sets the color array reference to the specified Color* (Color3b, Color3f, Color4b, or Color4f) array.

`void setColorRefByte(byte[] colors)` **`<new in 1.2>`**
Sets the byte color array reference to the specified array.

`void setColorRefFloat(float[] colors)` **`<new in 1.2>`**
Sets the float color array reference to the specified array.

`void setCoordRef*(Point*[] coords)` **`<new in 1.2>`**
Sets the coordinate array reference to the specified Point* (Point3d, Point3f) array.

`void setCoordRefDouble(double[] coords)` **`<new in 1.2>`**
Sets the double coordinate array reference to the specified array.

`void setCoordRefFloat(float[] coords)` **`<new in 1.2>`**
Sets the float coordinate array reference to the specified array.

`void setInterleavedVertices(float[] vertexData)` **`<new in 1.2>`**
Sets the interleaved vertices array reference to the specified array.

`void setNormalRef3f(Vector3f[] normals)` **`<new in 1.2>`**
Sets the Vector3f normal array reference to the specified array.

`void setNormalRefFloat(float[] normals)` **`<new in 1.2>`**
Sets the float normal array reference to the specified array.

`void setTexCoordRef*(int texCoordSet, TexCoord*[] texCoords)` **`<new in 1.2>`**
Sets the texture coordinate array reference for the specified texture coordinate set to the specified TexCoord* (TexCoord2f, TexCoord3f) array.

`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.

A simple way to change the geometry used BY_REFERENCE is to change the number of values used in an array. The following reference block lists methods allowing changes to the way referenced user geometry arrays are used. These methods can only be used when a GeometryArray object is in BY_REFERENCE mode. These method are only allowed on a live or compiled GeometryArray object if the ALLOW_REF_DATA_WRITE capability is set.

GeometryArray Methods for Setting Initial Location of Referenced Geometry Data (partial list)

This collection of methods set the initial location used in referenced geometry data in user arrays. In each case, the method will throw an exception if the data mode for this geometry array object is not BY_REFERENCE.

`void setInitialColorIndex(int initialColorIndex)` **`<new in 1.2>`**
Sets the initial color index for this GeometryArray object. This method is easily confused with the setInitialVetrexIndex method which is used when the geometry is not BY_REFERENCE.

`void setInitialCoordIndex(int initialCoordIndex)` **`<new in 1.2>`**
Sets the initial coordinate index for this GeometryArray object.

`void setInitialNormalIndex(int initialNormalIndex)` **`<new in 1.2>`**
Sets the initial normal index for this GeometryArray object.

`void setInitialTexCoordIndex(int texCoordSet,` **`<new in 1.2>`**
`int initialTexCoordIndex)`
Sets the initial texture coordinate index for the specified texture coordinate set for this GeometryArray object.

The following reference block lists the Capabilities of GeometryArray

GeometryArray Capabilites (partial list)

`BY_REFERENCE` **`<new in 1.2>`**
Specifies the position, color, normal, and texture coordinate data for this object are accessed by reference

`INTERLEAVED` **`<new in 1.2>`**
Specifies that the position, color, normal, and texture coordinate data for this GeometryArray are accessed via a single interleaved, floating-point array reference. This is only used in by-reference geometry mode.

`ALLOW_COUNT_READ | WRITE` **`<new in 1.2>`**
allow write access to the count or initial index information for this object

`ALLOW_REF_DATA_READ` **`<new in 1.2>`**
allow read access to the geometry data reference information; only used in by-reference geometry mode

`ALLOW_REF_DATA_WRITE` **`<new in 1.2>`**
allow write access to the geometry data reference information for this object. It also enables writing the referenced data itself, via the GeometryUpdater interface. This is only used in by-reference geometry mode.

IndexedGeometryArray

While it is possible to have user geometry (e.g., vertex, normal, color, texture, etc.) data referenced; it is not possible to reference index information. For this reason there are no new methods for IndexedGeometryArray. Also for the same reason, IndexedGeometryArray objects are good candidates for runtime manipulated by a GeometryUpdater when the vertices are dynamic, but the indices are static.

2.8.3 GeometryUpdater Interface <new in 1.2>

Geometry data in any array that is referenced by a live or compiled GeometryArray object may only be modified via the updateData method of the GeometryArray (subject to the ALLOW_REF_DATA_WRITE capability bit). Applications must exercise care not to violate this rule. If any referenced geometry data is modified outside of the updateData method, the results are undefined. The changes are made via another object which implements the GeometryUpdater interface. The GeometryUpdater object is assigned to a GeometryArray object using the updateData method shown in the following reference block.

updateData method of GeometryArray

`public void updateData(GeometryUpdater updater)` **`<new in 1.2>`**

Updates geometry array data that is accessed by reference. This method calls the updateData method of the specified GeometryUpdater object to synchronize updates to vertex data that is referenced by this GeometryArray object. Applications that wish to modify such data must perform all updates via this method.

GeometryUpdater objects are used to automatically change geometry of by reference GeometryArray data. GeometryUpdater objects are used in interactive (chapter 4) and/or animated (chapter 5) applications; consequently, this interface is not covered in this chapter (but will appear in a future version of Chapter 5).

2.8.4 AlternateAppearance <new in 1.2>

As seen in section 2.6.1 each Shape3D object can have a reference to an Appearance node component. That appearance bundle (i.e., the Appearance node component along with its referenced Attributes objects) defines visual properties of the geometry of the Shape3D object (see Figure 2-20 on page 2-37). Section 2.6.2 explains how it is possible to share a single appearance bundle among Shape3D objects. An AlternateAppearance gives the application developer an alternative method for sharing a single appearance bundle among Shape3D objects. The AlternateAppearance is more than an alternative to sharing an AppearanceBundle – it provides an easy way to temporarily change appearances.

An AlternateAppearance's application is controlled in three ways. First, a visual objects (i.e., Shape3D and Morph Nodes) must lie in the influencing bounding region of the AlternateAppearance. Second, a AlternateAppearance may have a scope limiting its application. Third, only those visual objects allowing their Appearance to be overridden are subject to the AlternateAppearance.

Using an AlternativeAppearance Node

To use an AlternativeAppearance, first create the appearance bundle to be used as the alternative and reference this appearance bundle by the AlternativeAppearance object. Then set an influencing bounds through a Bounds or BoundingLeaf object. Create a scope for the AlternateAppearance as desired. Don't forget to add the AlternateAppearance to the scenegraph. Set the appearanceOverrideEnable flag on the desired Shape3D objects.

Shape3D setAppearanceOverrideEnable() Method

Other Shape3D methods are discussed and listed in Sections 2.2.1 and 2.8.1.

`void setAppearanceOverrideEnable(boolean flag)` **`<new in 1.2>`**
Sets a flag that indicates whether this node's appearance can be overridden by an AlternativeAppearance node. This method can be called for a live or compiled object if the appropriate Capability is set.

Shape3D Capabilities (partial list, see Section 2.2.1)

`ALLOW_APPEARANCE_OVERRIDE_READ | WRITE` **`<new in 1.2>`**
allow read (write) access to the appearanceOverrideEnable flag

AlternativeAppearance Class **`<new in 1.2>`**

The following reference block lists the constructors for the AlternateAppearance class.

AlternateAppearance Constructor Summary

`AlternateAppearance()` **`<new in 1.2>`**
Constructs an AlternateAppearance node with default parameters: `appearance`, `scope`, `influencingBounds`, and `influencingBoundingLeaf` = `null`

`AlternateAppearance(Appearance appearance)` **`<new in 1.2>`**
Constructs an AlternateAppearance node with the specified appearance.

The following reference block lists the methods for the AlternateAppearance class.

AlternateAppearance Method Summary (partial list)

`void addScope(Group scope)` **`<new in 1.2>`**
Appends the specified Group node to this AlternateAppearance node's list of scopes.

`java.util.Enumeration getAllScopes()` **`<new in 1.2>`**
Returns an enumeration of this AlternateAppearance node's list of scopes.

`void insertScope(Group scope, int index)` **`<new in 1.2>`**
Inserts the specified Group node into this AlternateAppearance node's list of scopes at the specified index.

`int numScopes()` **`<new in 1.2>`**
Returns the number of nodes in this AlternateAppearance node's list of scopes.

`void removeScope(int index)` **`<new in 1.2>`**
Removes the node at the specified index from this AlternateAppearance node's list of scopes.

`void setAppearance(Appearance appearance)` **`<new in 1.2>`**
Sets the appearance of this AlternateAppearance node.

`void setInfluencingBoundingLeaf(BoundingLeaf region)` **`<new in 1.2>`**
Sets the AlternateAppearance's influencing region to the specified bounding leaf.

`void setInfluencingBounds(Bounds region)` **`<new in 1.2>`**
Sets the AlternateAppearance's influencing region to the specified bounds.

`void setScope(Group scope, int index)` **`<new in 1.2>`**
Replaces the node at the specified index in this AlternateAppearance node's list of scopes with the specified Group node.

The following reference block lists the capabilities for the AlternateAppearance class.

AlternativeAppearance Capabilities

**`ALLOW_APPEARANCE_READ	WRITE`**	allow read (write) access to its appearance information
**`ALLOW_INFLUENCING_BOUNDS_READ	WRITE`**	allow read (write) access to its influencing bounds and bounding leaf information
**`ALLOW_SCOPE_READ	WRITE`**	allow read (write) access to its scope information at runtime

2.9 Clipping

The subject of clipping is not truly one of creating geometry, but one that affects how geometry is viewed. For example, when geometry extends beyond the viewing boundaries it is clipped. That is, some of the geometry is not viewed.

Clipping of geometry is a complex technical topic. This section covers the basic discussion of clipping parameters defined by the View object in the view sub-graph and the clipping parameters defined by Clip and ModelClip objects in the content sub-graph. Basically this is a discussion of clipping from a

application programmer's point of view. For more technical information consult the Java 3D API Specification.

2.9.1 View Defines a Frustum

In Chapter 1 introduces the virtual coordinate system and explains the position of the eye relative to it. In your experimentation (and you should be experimenting with programs) you may have encountered situations in which a visual object does not render. The reason a visual object does not render could be any one of a long list of possibilities; among these is that the visual object is outside the viewing frustum.

The viewing **frustum** is the volume in which visual objects, and portions of visual objects, are displayed. The frustum is a combination of what volume can be viewed (i.e., based on viewing position, viewing direction, and field of view) and what is practical and efficient to render (i.e., a reasonable distance from the viewer). Figure 2-29 illustrates the specification of a frustum by these parameters.

Note that Figure 2-29 shows the front clipping plane behind the image plate. This is not illegal, but isn't typical (or recommended). However, it does make for a neater illustration of the frustum.

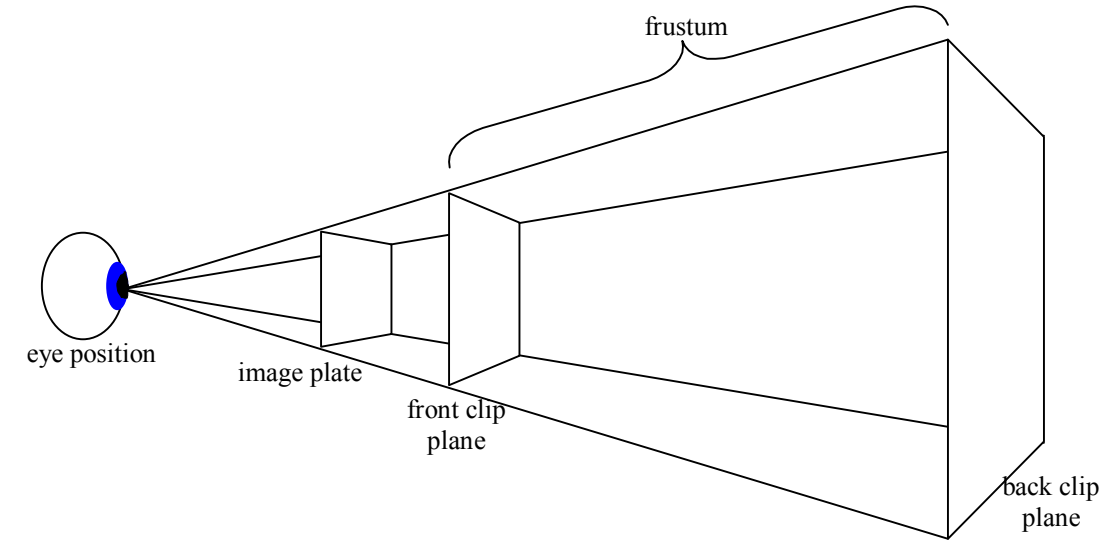


Figure 2-29 Viewing Frustum is Defined by Field of View and Front and Back Clip Distances

The frustum is defined relative to the eye position by the field of view and the front and back clipping distances. These values have reasonable defaults. Of course the application programmer can adjust the frustum by adjusting the field of view and/or the clip distances.

The only technical issue to consider in setting the parameters that define the frustum is the ratio of clip distances. Due to the implementation limitations of the various low level graphics systems that the Java 3D API employs (e.g., OpenGL), the application programmer should not use front and back clip distances that differ by a factor of more than 3000. That is, use front and back clip distances such that:

```
(back clip distance) / (front clip distance) < 3000.
```

In practice, ratios between 100 and 1000 work much better.

The frustum parameters are defined universally in the View object; however, the back clip distance can be changed locally[17]. Universal adjustments are make by adjusting the appropriate parameters in the View object. Section 2.9.2 shows how to use the Clip node to make localized adjustments to the back clip distance. The following reference block lists some View class methods for making frustum adjustments.

View Methods for Adjusting the Frustum (partial list)

The View object in the view branch graph of a Java 3D scene graph defines many viewing parameters. There are numerous methods for managing the parameters of the View. Only three such methods are listed below. See the Java 3D API Specification for more information.

`void setBackClipDistance(double distance)`
Sets the view model's back clip distance. This distance must be greater than the front clip distance.

`void setFrontClipDistance(double distance)`
Sets the view model's front clip distance. This distance must be **greater** than 0.0 and less than the back clip distance.

`void setFieldOfView(double fieldOfView)`
Sets the view model's horizontal field of view in radians.

2.9.2 Clip Node

As mentioned above, the back clip distance of the frustum can be adjusted locally. The Clip node is the object which accomplishes this. Each Clip node defines an application region specified as either a Bounds object or a BoundingLeaf object. When the application region of a Clip node intersects the activation volume of the view, the back clip distance of the Clip object is used in rendering. The front clip distance defined by the View object is unaffected.

While any number of aesthetic reasons could motivate the use of a Clip object; keep in mind that its use could improve rendering performance, especially in scenes where there is much complex geometry.

A Clip object is easy to use. However, it is also easy to forget to define the application region (i.e., bounds or bounding leaf) and to add the Clip object to the scene graph. These common mistakes are especially treacherous for the Clip object as the correct application of a Clip object may not change the rendered image.

The following reference block lists constructors for the Clip class.

Clip Constructor Summary

`Clip()`
Constructs a Clip node with default parameters: `backDistance` = 100, `applicationBounds` and `applicationBoundingLeaf` = `null`

`Clip(double backDistance)`
Constructs a Clip node with the specified back clip distance.

[17] "Universal(ly)" in this paragraph means 'for all visual objects as viewed by a particular view branch graph'. The same universe with a different view branch graph may have a different frustum.

The following reference block lists methods for the Clip class.

Clip Method Summary

`void setApplicationBoundingLeaf(BoundingLeaf region)`
Set the Clip's application region to the specified bounding leaf.

`void setApplicationBounds(Bounds region)`
Set the Clip's application region to the specified bounds.

`void setBackDistance(double backDistance)`
Sets the back clip distance to the specified value.

The following reference block lists capabilities for the Clip class.

Clip Field Summary

**`ALLOW_APPLICATION_BOUNDS_READ	WRITE`**	allow read (write) access to its application bounds and bounding leaf at runtime.
**`ALLOW_BACK_DISTANCE_READ	WRITE`**	allow read (write) access to its back distance at runtime.

2.9.3 ModelClip **`<new in 1.2>`**

ModelClip nodes can clip visual objects in arbitrary ways. A ModelClip object defines a set of six clipping planes of arbitrary orientations in the local coordinate system. The clipping planes define half-spaces for which vertices are culled from rendering. The definition of the half-spaces is similar to those of the BoundingPolytope and is discussed in Appendix D. However, the clipping planes of a ModelClip node do not have to form a convex closed volume as in the BoundingPolytope, in fact any number of the six clipping planes may be used without the others.

The intersection of the set of half-spaces corresponding to the enabled planes in this ModelClip node defines a region in which points are clipped. Only vertices not clipped are candidates for rendering. Those vertices not clipped may not be rendered for many possible reasons including occlusion and clipping of the view frustum.

The influence of a ModelClip node is controlled through the prescribed region of influence and scope. A visual object must be in the region of influence for the ModelClip node to be applied. In addition, if a scope is defined, a visual object must be in the scope of the ModelClip object.

2.9.4 ModelClip Example

Code Fragment 2-14 is taken from the `ModelClipApp.java` example program. This fragement shows the definition of the ModelClip object used in that application to clip the top off of a twisted strip object. In this application, only one of the six ModelClip clipping planes are used.

A ModelClip node is not hard to use. The steps include: ❶ Setting the influencing bounds for the ModelClip object, ❷ enabling and disabling clipping planes as desired, ❸ setting the orientation and position of the planes to use, and ❹ adding the ModelClip object to the scene graph. Although it is not done in this example, a ModelClip node may specify a scope.

```
1.          // create a ModelClip object and set the clip planes
2.          ModelClip modelClip = new ModelClip();
3.          modelClip.setInfluencingBounds(bounds); ❶
4.          boolean enables[] = {true, false, false, false, false, false};❷
5.          modelClip.setEnables(enables); ❷
6.          modelClip.setPlane(0, new Vector4d(0, 1, 0, -0.1)); ❸
7.          contentRoot.addChild(modelClip); ❹
```

Code Fragment 2-14 Specification of Clipping Planes in a ModelClip Node.

Figure 2-30 shows an image rendered by ModelClipApp. As you can see, the top of the twisted strip geometry has been clipped by the ModelClip object.

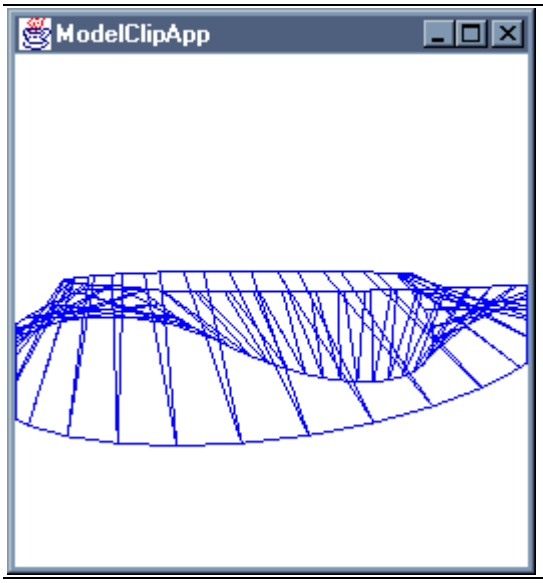


Figure 2-30 Clipped Twisted Strip

2.9.5 ModelClip Node

<new in 1.2>

Keep in mind that there are always six planes in a ModelClip object. Consequently, the Vector4d array used in the constructor should be of size 6.

ModelClip Constructor Summary

`ModelClip()` **<new in 1.2>**
Constructs a ModelClip node with default parameters.

`ModelClip(Vector4d[] planes)` **<new in 1.2>**
Constructs a ModelClip node using the specified planes.

`ModelClip(Vector4d[] planes, boolean[] enables)` **<new in 1.2>**
Constructs a ModelClip node using the specified planes and enable flags.

Keep in mind that there are always six planes in a ModelClip object. Consequently, the Vector4d array used in the setPlanes() method and the boolean array in the setEnables() method should both be of size 6.

ModelClip Method Summary (partial list)

`void addScope(Group scope)` **`<new in 1.2>`**
Appends the specified Group node to this ModelClip node's list of scopes.

`void insertScope(Group scope, int index)` **`<new in 1.2>`**
Inserts the specified Group node into this ModelClip node's list of scopes at the specified index.

`int numScopes()` **`<new in 1.2>`**
Returns the number of nodes in this ModelClip node's list of scopes.

`void removeScope(int index)` **`<new in 1.2>`**
Removes the node at the specified index from this ModelClip node's list of scopes.

`void setEnable(int planeNum, boolean enable)` **`<new in 1.2>`**
Sets the specified enable flag of this ModelClip node.

`void setEnables(boolean[] enables)` **`<new in 1.2>`**
Sets the per-plane enable flags of this ModelClip node to the specified values.

`void setInfluencingBoundingLeaf(BoundingLeaf region)` **`<new in 1.2>`**
Set the ModelClip node's influencing region to the specified bounding leaf.

`void setInfluencingBounds(Bounds region)` **`<new in 1.2>`**
Set the ModelClip node's influencing region to the specified bounds.

`void setPlane(int planeNum, Vector4d plane)` **`<new in 1.2>`**
Sets the specified clipping plane of this ModelClip node.

`void setPlanes(Vector4d[] planes)` **`<new in 1.2>`**
Sets the clipping planes of this ModelClip node to the specified planes.

`void setScope(Group scope, int index)` **`<new in 1.2>`**
Replaces the node at the specified index in this ModelClip node's list of scopes with the specified Group node.

The following reference block lists the capabilities for the ModelClip class.

ModelClip Field Summary

**`ALLOW_ENABLE_READ	WRITE`**	allow read (write) access to its enable flags at runtime
**`ALLOW_INFLUENCING_BOUNDS_READ	WRITE`**	allow read (write) access to its influencing bounds and bounding leaf at runtime
**`ALLOW_PLANE_READ	WRITE`**	allow read (write) access to its planes at runtime
**`ALLOW_SCOPE_READ	WRITE`**	allow read (write) access to its scope information at runtime

2.10 Chapter Summary

In this chapter are many topics on the creation of visual content. The chapter begins explaining the basics of visual objects and the use of Shape3D objects. Section 2.3 presents utility classes for basic geometric primitives such as sphere, box, and cone. Later an example application is built using those cone primitives. The chapter continues with an outline of the mathematical classes for point, color, normal, and other geometric data types. Then, after introducing the geometry primitives in section 2.5, which allow the creation of polygonal geometry, examples of using the math classes in creating geometry is given. Section 2.6 moves the chapter to the discussion of appearances and how the same geometry can be rendered with a variety of visual attributes applied.

Section 2.7 discusses Bounds and Scope as a prelude to more advanced geometry topics. Section 2.8 presents advanced geometry including AlternativeAppearance, multiple Geometry objects in a Shape3D node, and Geometry by BY_REFERENCE. Section 2.9 presents clipping topics including clipping as defined by the View, Clip Nodes, and ModelClip Nodes.

This is Section 2.10. Section 2.11 has review questions intended to stimulate thought and understanding of the presented material. Answers to some of the questions are found in the appendicies.

2.11 Self Test

On the next couple of pages are a few exercises designed to test and enhance your understanding of the material presented in this chapter. The solutions to some of these exercises are given in Appendix C.

1. Try your hand at creating a new yo-yo using two cylinders instead of two cones. Using `ConeYoyoApp.java` as a starting point, what changes are needed?
2. A two-cylinder yo-yo can be created with two quad-strip objects and four triangle-fan objects. Another way is to reuse one quad-strip and one triangle fan. What objects would form this yo-yo visual object? The same approach can be used to create the cone yo-yo. What object would form this yo-yo visual object?
3. The default culling mode is used in YoyoLineApp.java and YoyoPointApp.java. Change either, or both, of these programs to cull nothing, then compile and run the modified program. What difference do you see?
4. What happens in the ModelClipApp.java program if the ModelClip object rotates and the visual object (the Twist) remains still?
5. Can you make a hole in the some geometry using the ModelClip object?