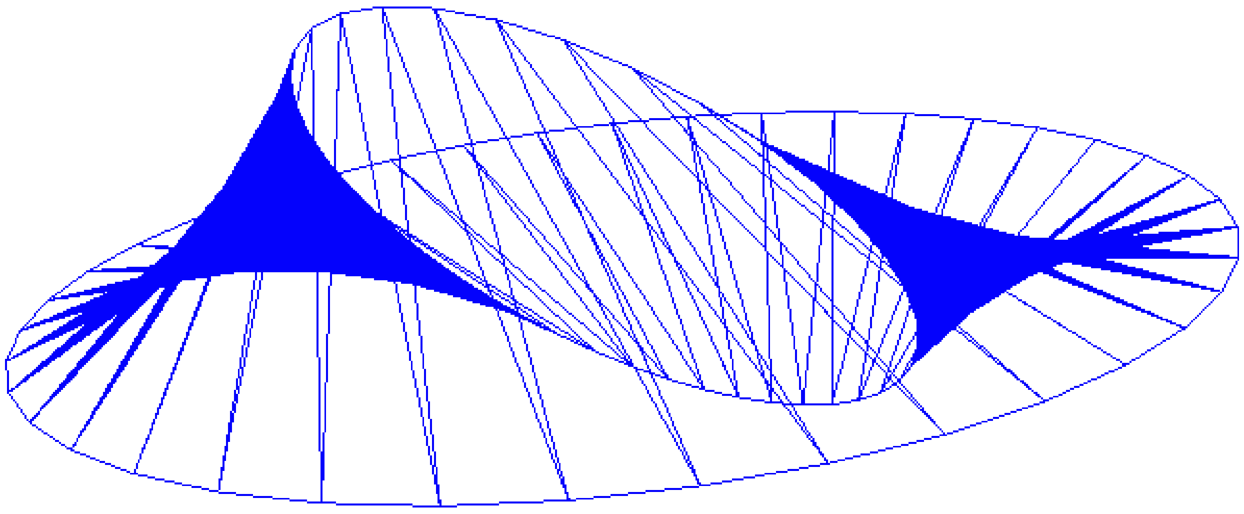
Getting Started with the Java 3D™ API

Chapter 2
Creating Geometry



Dennis J Bouvier

K Computing

tutorial v1.4 (Java 3D API v1.1.2)

Chapter 2: Creating Geometry

Creating Geometry 2-1

2.1 Virtual World Coordinate System 2-1

2.2 Visual Object Definition Basics 2-2
2.2.1 An Instance of Shape3D Defines a Visual Object 2-2
2.2.2 Node Components 2-4
2.2.3 Defining Visual Object Classes 2-5

2.3 Geometric Utility Classes 2-6
2.3.1 Box 2-7
2.3.2 Cone 2-8
2.3.3 Cylinder 2-9
2.3.4 Sphere 2-9
2.3.5 More About Geometric Primitives 2-10
2.3.6 ColorCube 2-10
2.3.7 Example: Creating a Simple Yo-Yo From Two Cones 2-11
Advanced Topic: Geometric Primitive 2-14

2.4 Mathematical Classes 2-15
2.4.1 Point Classes 2-18
2.4.2 Color Classes 2-19
2.4.3 Vector Classes 2-19
2.4.4 TexCoord Classes 2-20

2.5 Geometry Classes 2-20
2.5.1 GeometryArray Class 2-21
2.5.2 Subclasses of GeometryArray 2-25
2.5.3 Subclasses of GeometryStripArray 2-27
2.5.4 Subclasses of IndexedGeometryArray 2-31
2.5.5 Axis.java is an Example of IndexedGeometryArray 2-34

2.6 Appearance and Attributes 2-34
2.6.1 Appearance NodeComponent 2-36
2.6.2 Sharing NodeComponent Objects 2-36
2.6.3 Attribute Classes 2-37
2.6.4 Example: Back Face Culling 2-42

2.7 Self Test 2-44

List of Figures

Figure 2-1 Orientation of Axis in Virtual World 2-2
Figure 2-2 A Shape3D Object Defines a Visual Object in a Scene Graph. 2-3
Figure 2-3 Partial Java 3D API Class Hierarchy Showing Subclasses of NodeComponent. 2-5
Figure 2-4 Class Hierarchy for Utility Geometric Primitives: Box, Cone, Cylinder, and Sphere 2-7
Figure 2-5 Class Hierarchy of ColorCube Utility Geometric Class 2-11
Figure 2-6 Scene Graph for ConeYoyoApp. 2-12
Figure 2-7 Multiple Parent Exception While Attempting to Reuse a Cone Object 2-13
Figure 2-8 An Image Rendered by ConeYoyoApp.java 2-13
Figure 2-9 Mathematical Classes Package and Hierarchy 2-16
Figure 2-10 Geometry Class Hierarchy 2-21
Figure 2-11 Axis Class in AxisApp.java Creates this Scene Graph 2-25
Figure 2-12 Non-Indexed GeometryArray Subclasses 2-26
Figure 2-13 GeometryArray Subclasses 2-26
Figure 2-14 GeometryStripArray Subclasses 2-27
Figure 2-15 Three Views of the Yo-yo 2-28
Figure 2-16 Yo-yo with Colored Filled Polygons 2-31
Figure 2-17 Index and Data Arrays for a Cube 2-32
Figure 2-18 IndexedGeometryArray Subclasses 2-32
Figure 2-19 An Appearance Bundle 2-35
Figure 2-20 Appearance Bundle Created by Code Fragment 2-9. 2-36
Figure 2-21 Multiple Appearance Objects Sharing a Node Component 2-37
Figure 2-22 Twisted Strip with Back Face Culling 2-42
Figure 2-23 Twisted Strip without Back Face Culling 2-43
Figure 2-24 Determining the Front Face of Polygons and Strips 2-44

List of Tables

Table 2-1 Attribute Defaults 2-42

List of Code Fragments

Code Fragment 2-1 Skeleton Code for a VisualObject Class 2-6
Code Fragment 2-2 Class ConeYoyo From ConeYoyoApp.java Example Program 2-14
Code Fragment 2-3 Example ColorConstants Class 2-19
Code Fragment 2-4 GeometryArray Constructors 2-23
Code Fragment 2-5 Storing Data into a GeometryArray Object 2-24
Code Fragment 2-6 GeometryArray Objects Referenced by Shape3D Objects 2-25
Code Fragment 2-7 yoyoGeometry() Method Creates TriangleFanArray Object 2-29
Code Fragment 2-9 Using Appearance and ColoringAttributes NodeComponent Objects 2-35
Code Fragment 2-10 Disable Back Face Culling for the Twisted Strip 2-43

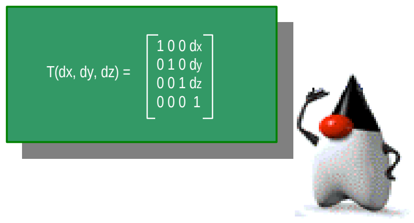
List of Reference Blocks

Shape3D Constructors 2-3
Shape3D Methods (partial list)....................2-3
Shape3D Capabilities....................2-4
Box Constructors (partial list) 2-8
Box, Cone, and Cylinder Methods 2-8
Cone Constructors (partial list)....................2-9
Cylinder Constructors (partial list) 2-9
Sphere Constructors (partial list)....................2-9
Sphere Methods 2-10
Primitive Methods (partial list)....................2-15
Tuple2f Constructors 2-17
Tuple2f Methods (partial list)....................2-17
Point3f Methods (partial list)....................2-18
Color* Classes....................2-19
Vector3f Methods (partial list) 2-20
GeometryArray Constructor....................2-22
GeometryArray Methods (partial list)....................2-23
GeometryArray Methods (partial list, continued) 2-24
GeometryArray Subclass Constructors....................2-26
GeometryStripArray Subclass Constructors....................2-27
Triangulator Class 2-28
Constructor Summary....................2-28
Method Summary....................2-28
IndexedGeometryArray and Subclasses Constructors....................2-33
IndexedGeometryStripArray and Subclasses Constructors 2-33
IndexedGeometryArray Methods (partial list) 2-34
Appearance Constructor....................2-36
Appearance Methods (excluding lighting and texturing) 2-36
PointAttributes Constructors 2-37
PointAttributes Methods....................2-38
LineAttributes Constructors 2-38
LineAttributes Methods....................2-38
PolygonAttributes Constructors....................2-39
PolygonAttributes Methods 2-39
ColoringAttributes Constructors....................2-39
ColoringAttributes Methods 2-40
TransparencyAttributes Constructors 2-40
TransparencyAttributes Methods....................2-40
RenderingAttributes Constructors....................2-41
RenderingAttributes Methods 2-41

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`

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 for visual objects

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.

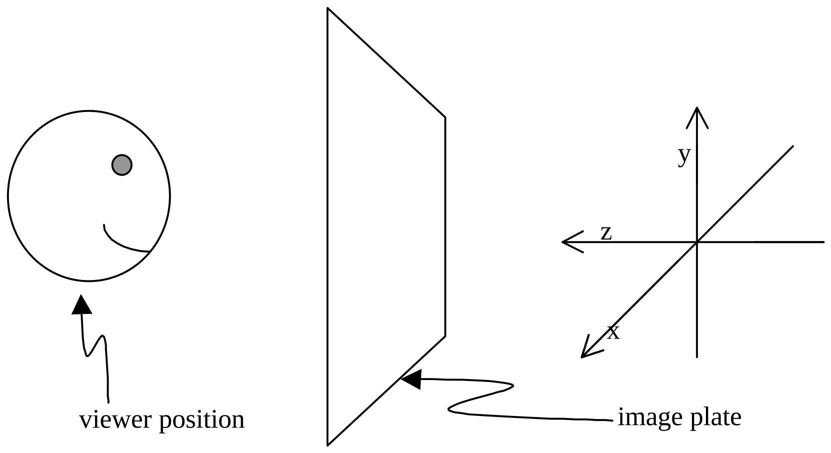


Figure 2-1 Orientation of Axis in Virtual World

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.

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[2].

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

[2] 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.

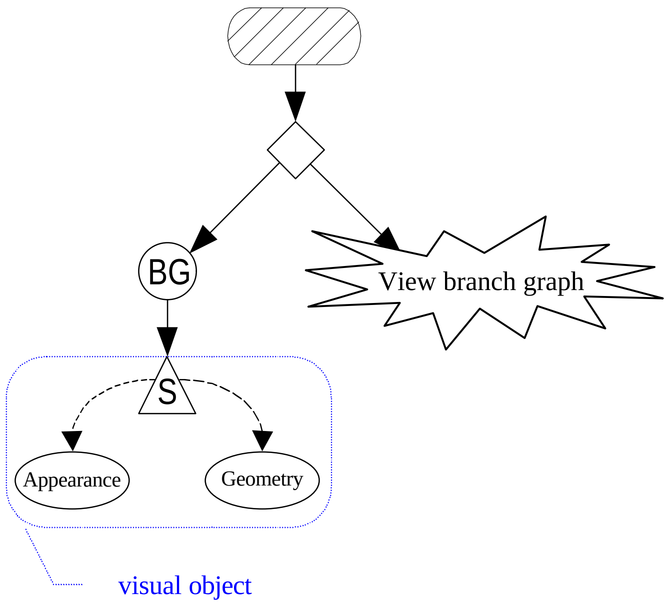


Figure 2-2 A Shape3D Object Defines a Visual Object in a Scene Graph.

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 without 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 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

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.

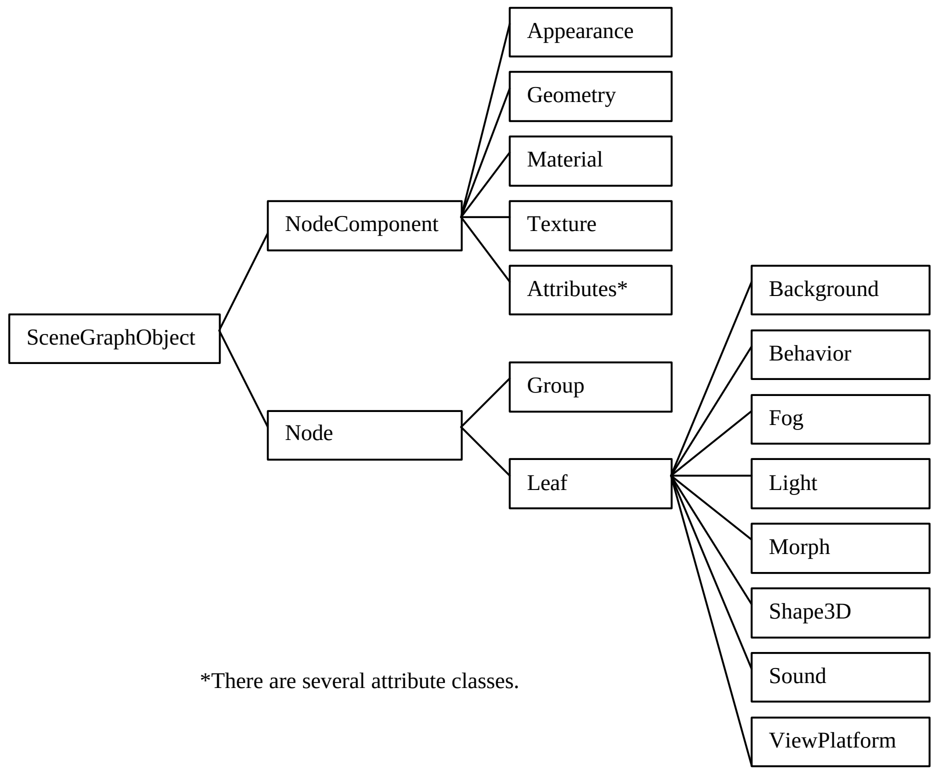


Figure 2-3 Partial Java 3D API Class Hierarchy Showing Subclasses of NodeComponent.

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.

```
public class VisualObject extends Shape3D{

    private Geometry voGeometry;
    private Appearance voAppearance;

    // create Shape3D with geometry and appearance
    // the geometry is created in method createGeometry
    // the appearance is created in method createAppearance
    public VisualObject() {

        voGeometry = createGeometry();
        voAppearance = createAppearance();
        this.setGeometry(voGeometry);
        this.setAppearance(voAppearance);
    }

```

```
        private Geometry createGeometry() {
            // code to create default geometry of visual object
        }

        private Appearance createAppearance () {
            // code to create default appearance of visual object
        }

    } // 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[3]. 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[4]. 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.

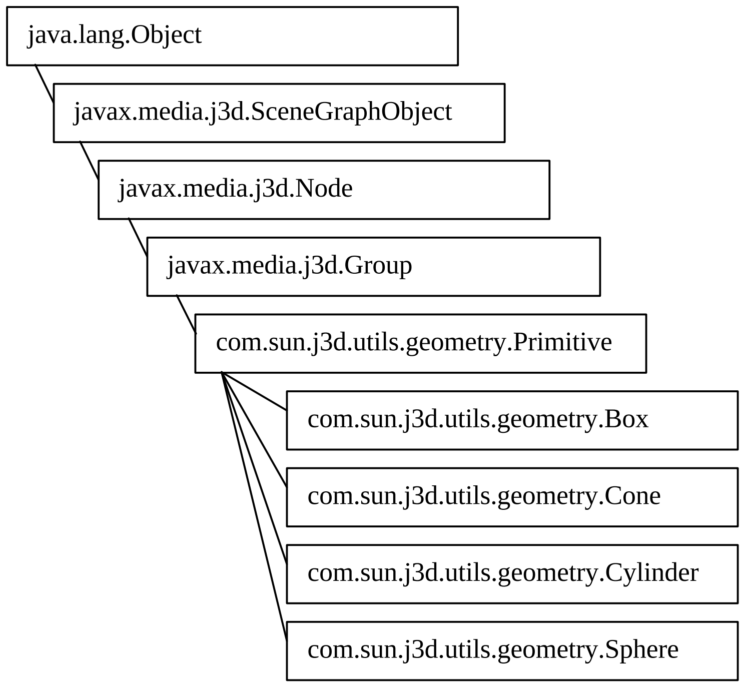

java.lang.Object
javax.media.j3d.SceneGraphObject
javax.media.j3d.Node
javax.media.j3d.Group
com.sun.j3d.utils.geometry.Primitive
com.sun.j3d.utils.geometry.Box
com.sun.j3d.utils.geometry.Cone
com.sun.j3d.utils.geometry.Cylinder
com.sun.j3d.utils.geometry.Sphere

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[5]. 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

---

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

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

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

length, width, and height can be specified at object creation time. Of course, TransformGroup along 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 extention 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.

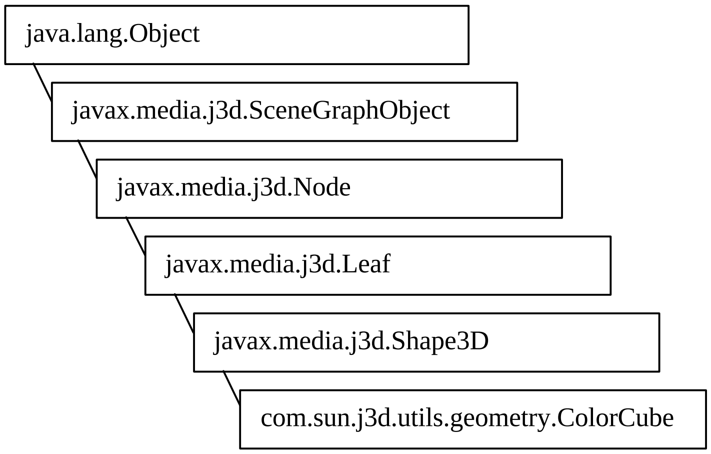


Figure 2-5 Class Hierarchy of ColorCube Utility Geometric Class

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.

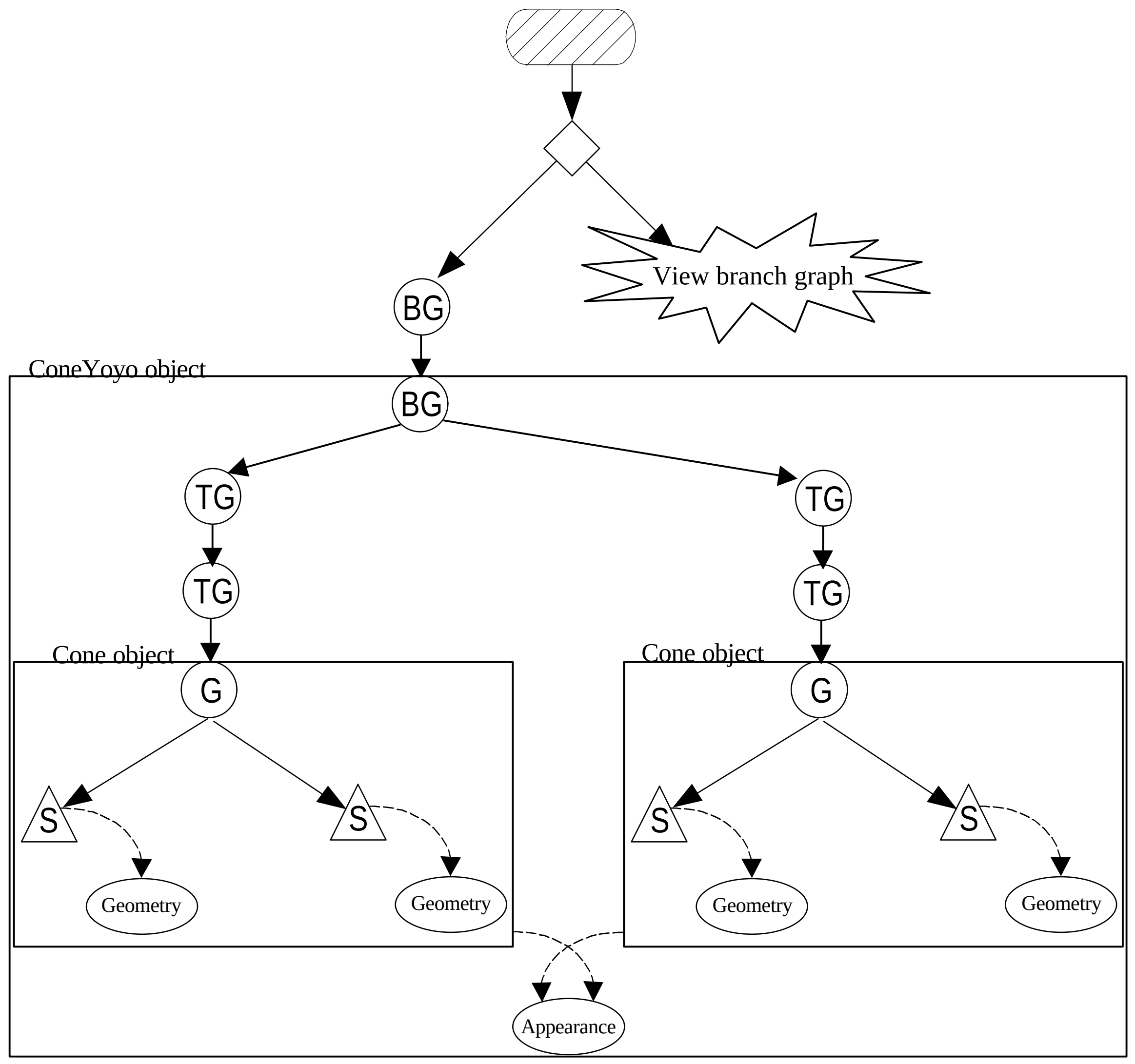


Figure 2-6 Scene Graph for ConeYoyoApp[6]

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.

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

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

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

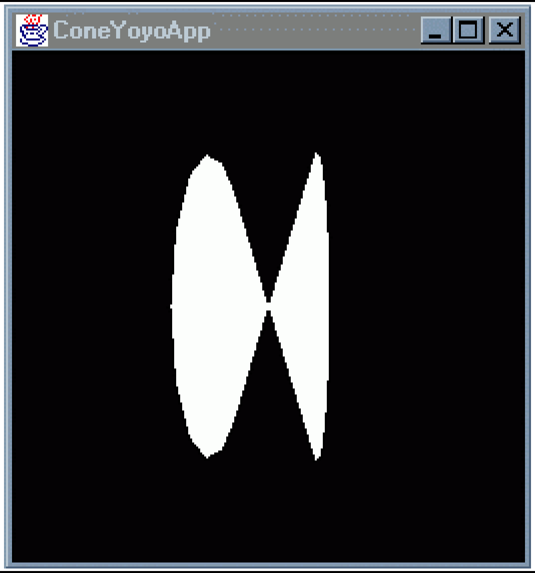


Figure 2-8 An Image Rendered by ConeYoyoApp.java

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.
```

```
        yoyoBG = new BranchGroup();
        Transform3D rotate = new Transform3D();
        Transform3D translate = new Transform3D();
        Appearance yoyoAppear = new Appearance();

        rotate.rotZ(Math.PI/2.0d);
        TransformGroup yoyoTGR1 = new TransformGroup(rotate);

        translate.set(new Vector3f(0.1f, 0.0f, 0.0f));
        TransformGroup yoyoTGT1 = new TransformGroup(translate);

        Cone cone1 = new Cone(0.6f, 0.2f);
        cone1.setAppearance(yoyoAppear);

        yoyoBG.addChild(yoyoTGT1);
        yoyoTGT1.addChild(yoyoTGR1);
        yoyoTGR1.addChild(cone1);

        translate.set(new Vector3f(-0.1f, 0.0f, 0.0f));
        TransformGroup yoyoTGT2 = new TransformGroup(translate);

        rotate.rotZ(-Math.PI/2.0d);
        TransformGroup yoyoTGR2 = new TransformGroup(rotate);

        Cone cone2 = new Cone(0.6f, 0.2f);
        cone2.setAppearance(yoyoAppear);

        yoyoBG.addChild(yoyoTGT2);
        yoyoTGT2.addChild(yoyoTGR2);
        yoyoTGR2.addChild(cone2);

        yoyoBG.compile();

   } // end of ConeYoyo constructor

   public BranchGroup getBG(){
       return yoyoBG;
   }

} // 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.

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).

```
      myCone.setPrimitiveFlags(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 setPrimitiveFlags(int fl)`
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.

`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[7].

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, Tuple4f, and Tuple4d. In each case the number indicates the number of elements in the tuple, and the letter indicates the data type of the elements. ‘f’ indicates single-precision floating point, ‘d’ indicates double-precision floating point, and ‘b’ is for bytes. 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.

[7] TexCoord* classes are not used in Java 3D API version 1.1. This will change in subsequent versions.

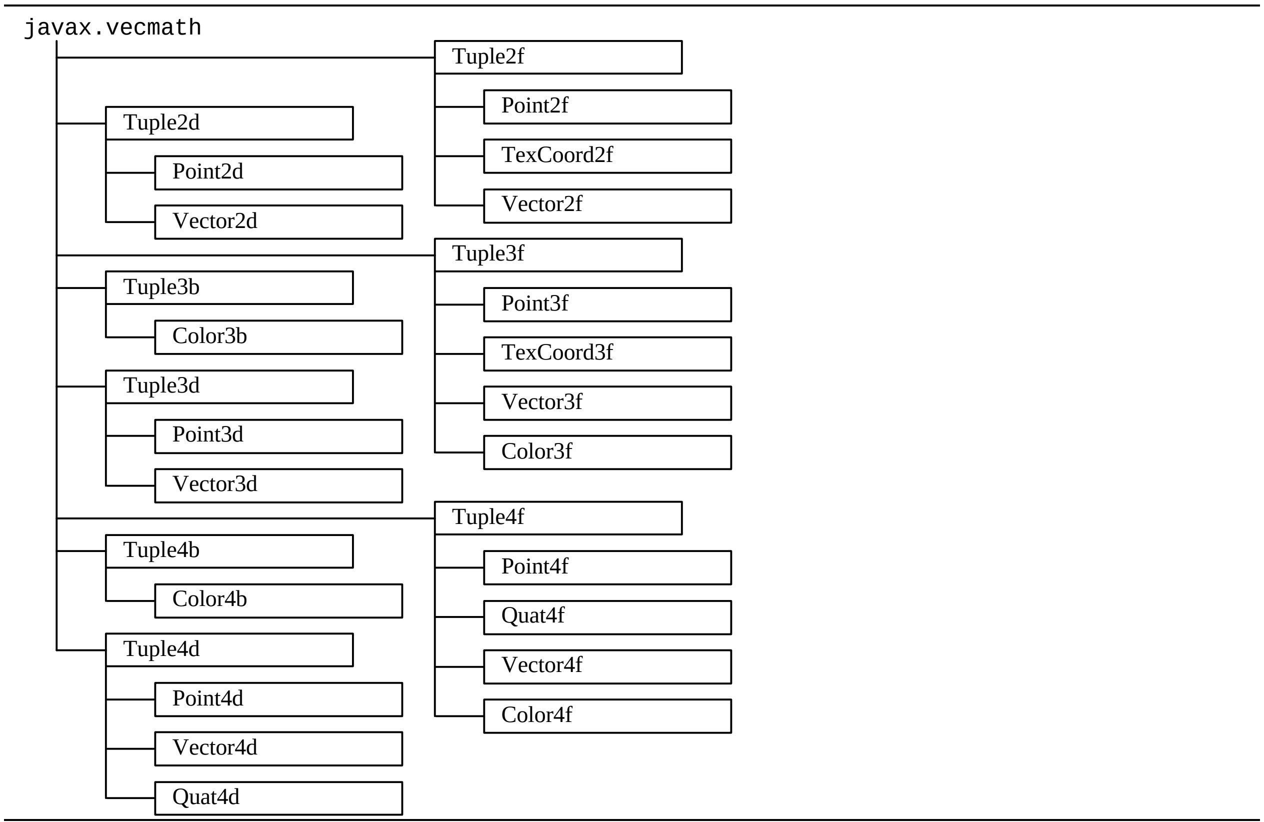


Figure 2-9 Mathematical Classes Package and Hierarchy

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)

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.

(The 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.

`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 range between 0 and 255, inclusive. For single-precision floating point data, color values range between 0.0 and 1.0, inclusive.

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 do not have additional methods, so they rely upon the methods they inherit from their Tuple* superclasses.

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);
```

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.

```
import javax.vecmath.*;

class ColorConstants{
   public static final Color3f red     = new Color3f(1.0f,0.0f,0.0f);
   public static final Color3f green   = new Color3f(0.0f,1.0f,0.0f);
   public static final Color3f blue    = new Color3f(0.0f,0.0f,1.0f);
   public static final Color3f yellow  = new Color3f(1.0f,1.0f,0.0f);
   public static final Color3f cyan    = new Color3f(0.0f,1.0f,1.0f);
   public static final Color3f magenta = new Color3f(1.0f,0.0f,1.0f);
   public static final Color3f white   = new Color3f(1.0f,1.0f,1.0f);
   public static final Color3f black   = new Color3f(0.0f,0.0f,0.0f);
}
```

Code Fragment 2-3 Example ColorConstants Class

Color* Classes

Package: `javax.vecmath`

The Color* classes are derived from Tuple* classes. Each instances of the Color* classes represents a single color in three components (RGB), or four components (RGBA). The Color* classes do not add any methods to those supplied by Tuple* classes.

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.

The constructors for TexCoord* classes are again similar to the Tuple* constructors. Like the Color* classes, the TexCoord* classes also 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

.

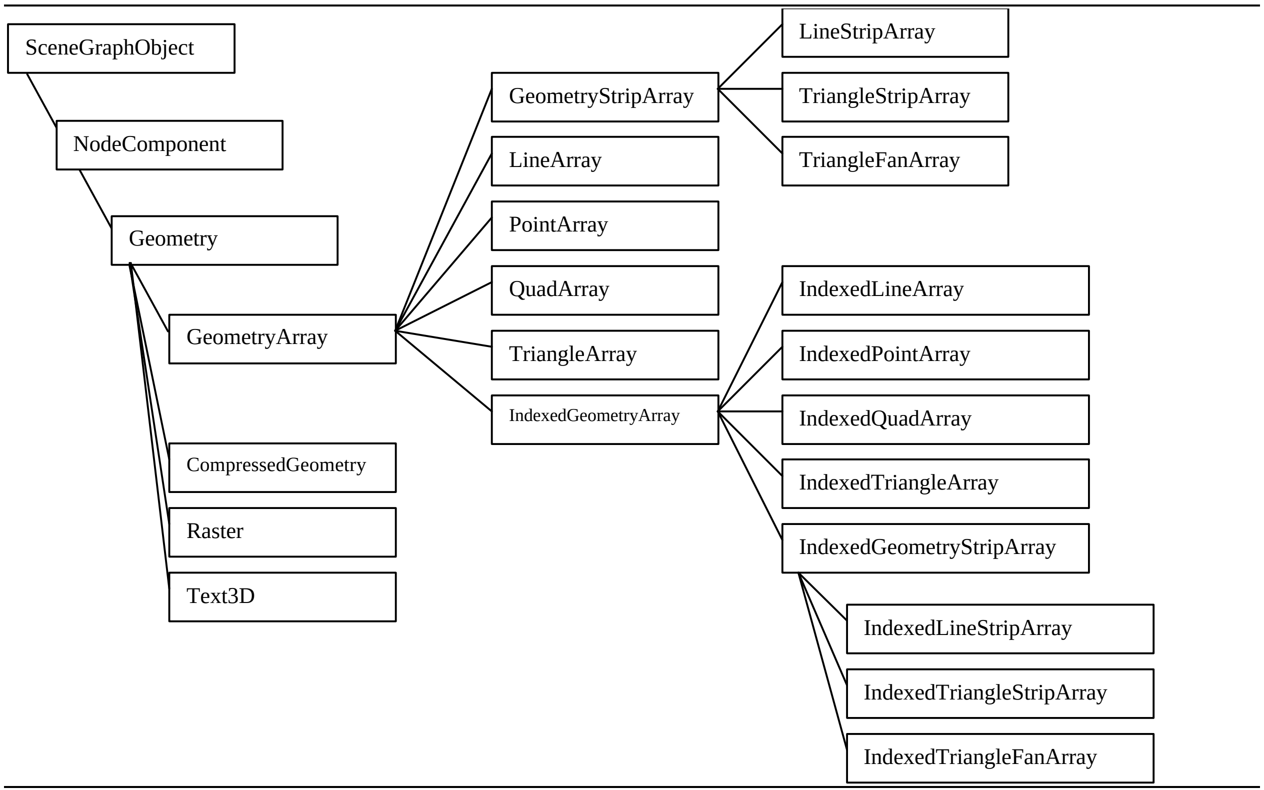


Figure 2-10 Geometry Class Hierarchy

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.

```
// construct object to represent the X axis
LineArray axisXLines= new LineArray (2, LineArray.COORDINATES);

// construct object to represent the Y axis
LineArray axisYLines = new LineArray(2, LineArray.COORDINATES
                                        | LineArray.COLOR_3);

// construct object to represent the Z axis
LineArray axisZLines = new LineArray(10, LineArray.COORDINATES
                                         | 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.

GeometryArray Methods (partial list, continued)

`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 index, float[] texCoord)`

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

`void setTextureCoordinates(int index, float[] texCoords)`

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

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.

```
axisXLines.setCoordinate(0, new Point3f(-1.0f, 0.0f, 0.0f));
axisXLines.setCoordinate(1, new Point3f( 1.0f, 0.0f, 0.0f));

Color3f red   = new Color3f(1.0f, 0.0f, 0.0f);
Color3f green = new Color3f(0.0f, 1.0f, 0.0f);
Color3f blue  = new Color3f(0.0f, 0.0f, 1.0f);
axisYLines.setCoordinate(0, new Point3f( 0.0f,-1.0f, 0.0f));
axisYLines.setCoordinate(1, new Point3f( 0.0f, 1.0f, 0.0f));
axisYLines.setColor(0, green);
axisYLines.setColor(1, blue);

axisZLines.setCoordinate(0, z1);
axisZLines.setCoordinate(1, z2);
axisZLines.setCoordinate(2, z2);
axisZLines.setCoordinate(3, new Point3f( 0.1f, 0.1f, 0.9f));
axisZLines.setCoordinate(4, z2);
axisZLines.setCoordinate(5, new Point3f(-0.1f, 0.1f, 0.9f));
axisZLines.setCoordinate(6, z2);
axisZLines.setCoordinate(7, new Point3f( 0.1f,-0.1f, 0.9f));
axisZLines.setCoordinate(8, z2);
axisZLines.setCoordinate(9, new Point3f(-0.1f,-0.1f, 0.9f));

Color3f colors[] = new Color3f[9];
colors[0] = new Color3f(0.0f, 1.0f, 1.0f);
for(int v = 0; v < 9; v++)
  colors[v] = red;
axisZLines.setColors(1, colors);
```

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.)

```
axisBG = new BranchGroup();

axisBG.addChild(new Shape3D(axisYLines));
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`.

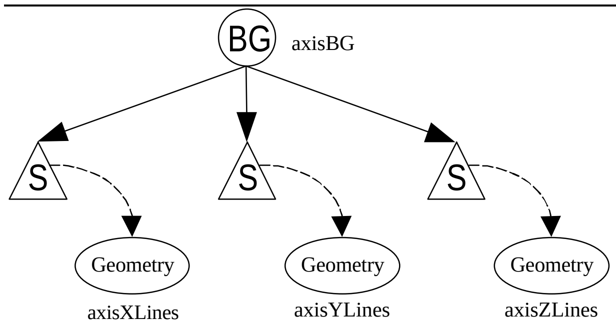


Figure 2-11 Axis Class in AxisApp.java Creates this Scene Graph

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.

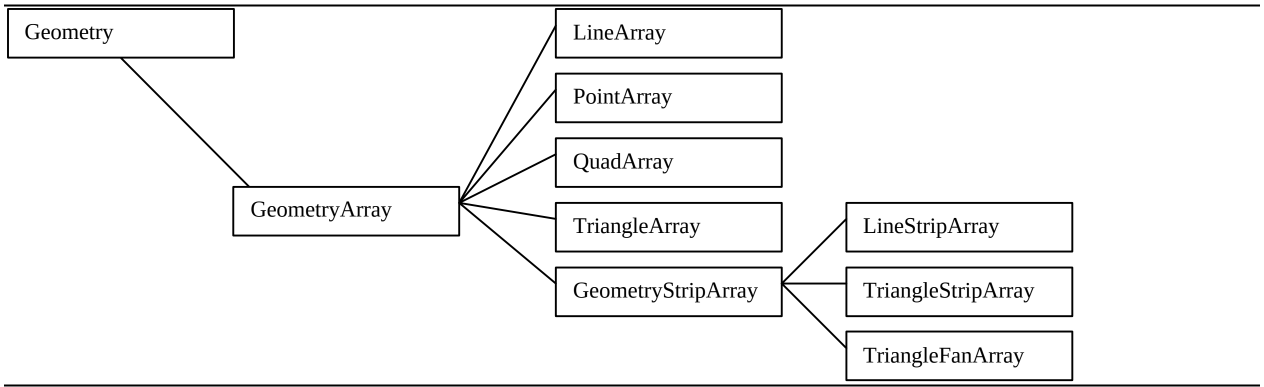


Figure 2-12 Non-Indexed GeometryArray Subclasses

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.

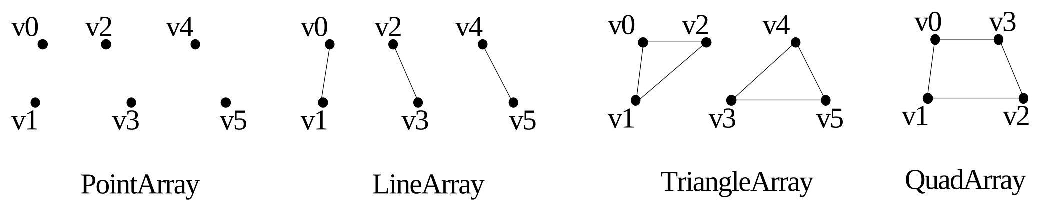


Figure 2-13 GeometryArray Subclasses

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.

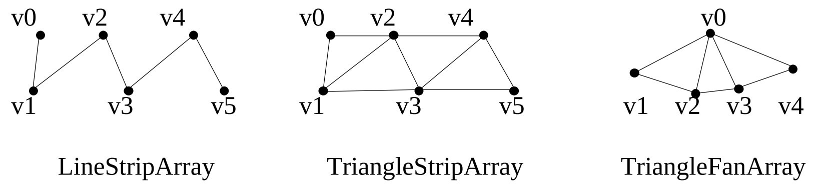


Figure 2-14 GeometryStripArray Subclasses

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[8].

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

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.

Constructor Summary

`Triangulator()`
Create a Triangulator object.

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.

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.

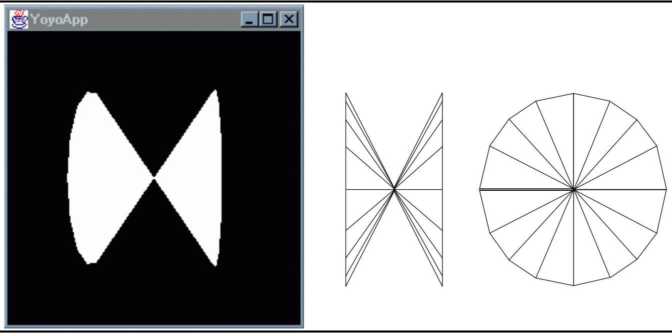


Figure 2-15 Three Views of the Yo-yo

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.

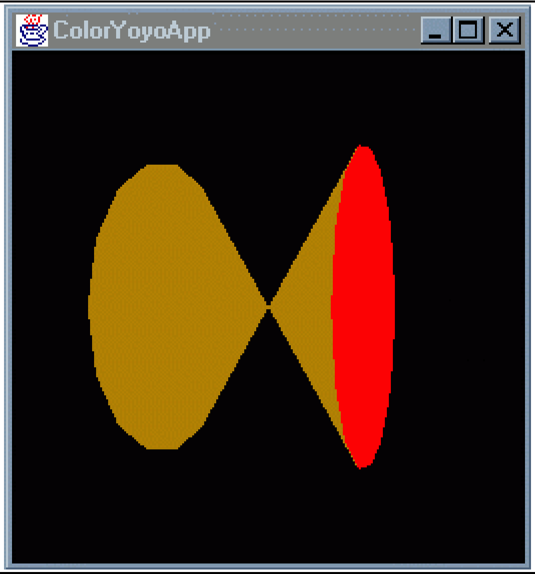


Figure 2-16 Yo-yo with Colored Filled Polygons

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.

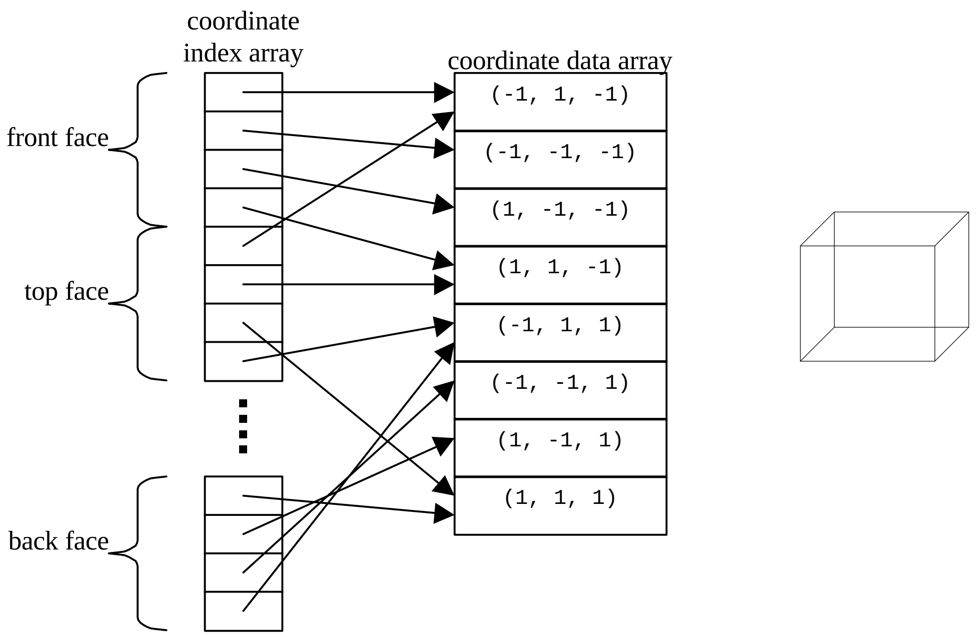


Figure 2-17 Index and Data Arrays for a Cube

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

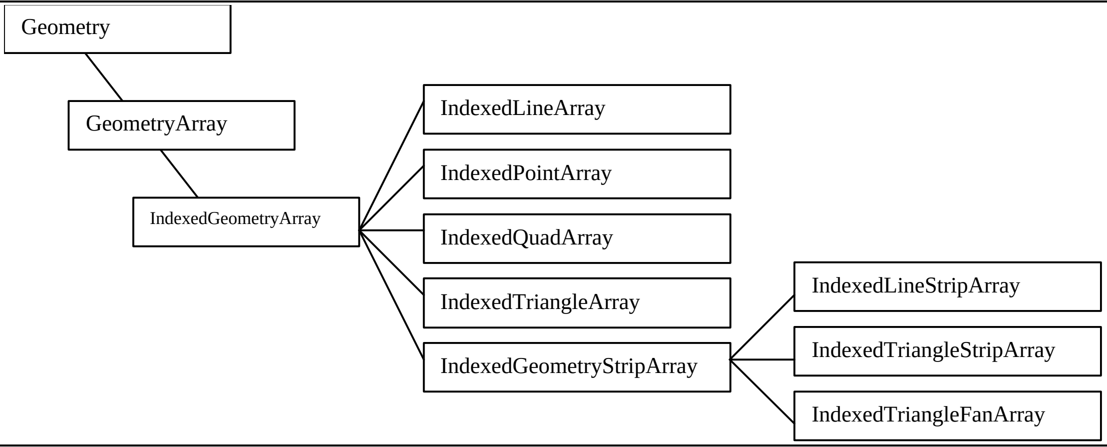


Figure 2-18 IndexedGeometryArray Subclasses

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 index, int texCoordIndex)`
Sets the texture coordinate index associated with the vertex at the specified index for this object.

`void setTextureCoordinateIndices (int index, int[] texCoordIndices)`
Sets the texture coordinate indices associated with the vertices starting at the specified index 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.

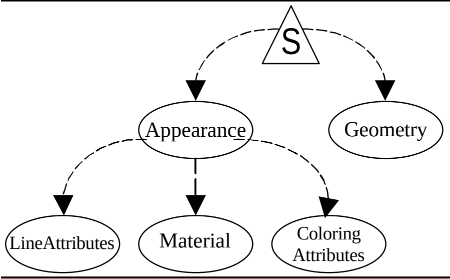


Figure 2-19 An Appearance Bundle

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 advanced topics, which are not discussed in this section.

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:

```
ColoringAttributes ca = new ColoringAttributes();
ca.setColor (1.0, 1.0, 0.0);
Appearance app = new Appearance();
app.setColoringAttributes(ca);
Shape3D s3d = new Shape3D();
s3d.setAppearance (app);
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.

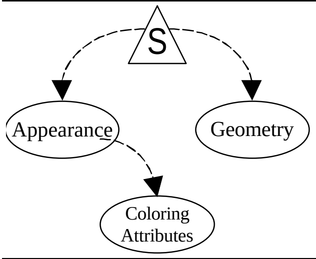


Figure 2-20 Appearance Bundle Created by Code Fragment 2-9.

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.

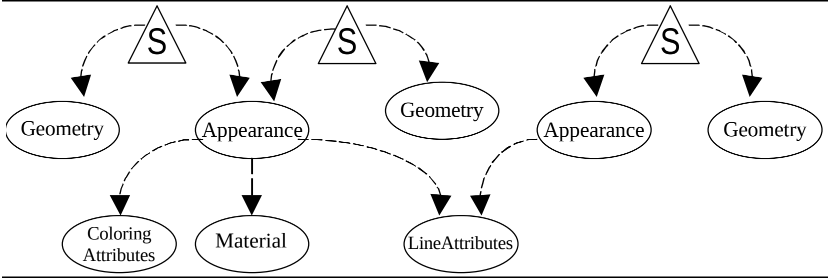


Figure 2-21 Multiple Appearance Objects Sharing a Node Component

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

In this section, six of the NodeComponent subclasses that can be referenced by Appearance are described (excluding the ones used for lighting and texturing).

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 point is larger than 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().

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, or PATTERN_DASH_DOT. Describes how the pixels of a line should be filled.

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

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.

By default, vertices rendered as both wire frame and filled polygons are not always rasterized with the same depth values, which can cause *stitching* when the wire frame should be fully visible. With `setPolygonOffset()`, the depth values of the filled polygons could be nudged toward the image plate, so that the wire frame would outline the filled object properly. `setBackFaceNormalFlip()` is useful to render a lit, filled polygon, where a both sides of the polygon are to be shaded. See Section 2.6.4 for an example program that shades both sides of polygons.

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 reverses how front and back facing polygons are determined.

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 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 opacity value (often known as alpha blending) for the primitive. setTransparencyMode() enables transparency and selects what kind of rasterization is used to produce transparency.

TransparencyAttributes Constructors

`TransparencyAttributes()`
Creates a component object with the transparency mode of FASTEST.

`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 Methods

`void setTransparency(float tVal)`
where tVal specifies an object’s opacity (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.

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.

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 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.

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

color	white (1, 1, 1)
texture environment mode	TEXENV_REPLACE
texture environment color	white (1, 1, 1)
depth test enable	true
shade model	SHADE_GOURAUD
polygon mode	POLYGON_FILL
transparency enable	false
transparency mode	FASTEST
cull face	CULL_BACK
point size	1.0
line width	1.0
point antialiasing enable	false
line antialiasing enable	false

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.

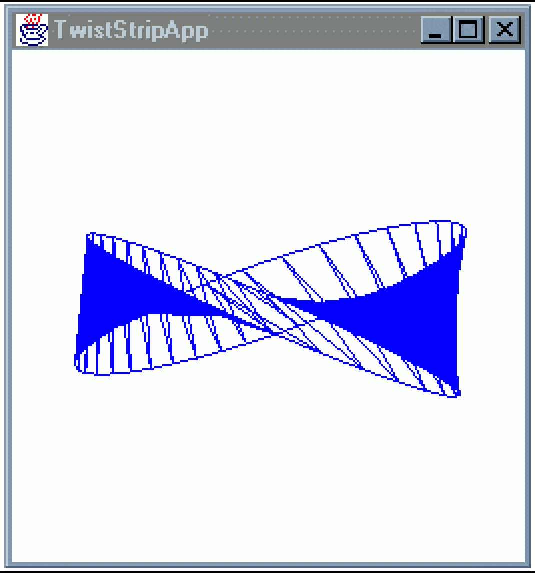


Figure 2-22 Twisted Strip with Back Face Culling

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.

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

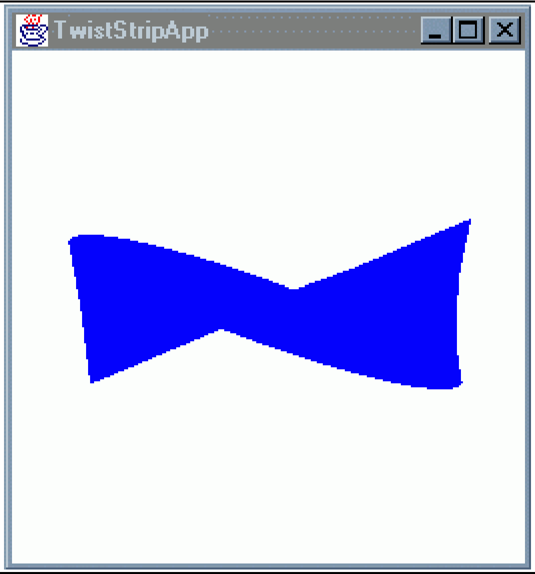


Figure 2-23 Twisted Strip without Back Face Culling

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.

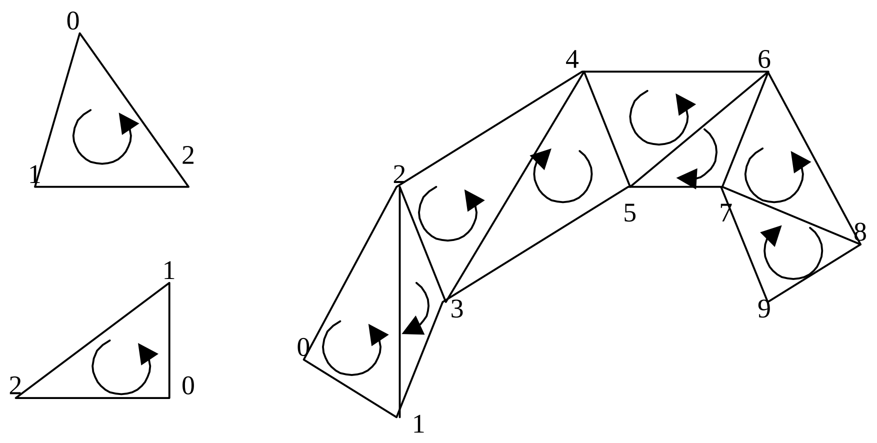


Figure 2-24 Determining the Front Face of Polygons and Strips

2.7 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?