Getting Started with the Java 3D™ API

Chapter 3
Easier Content Creation
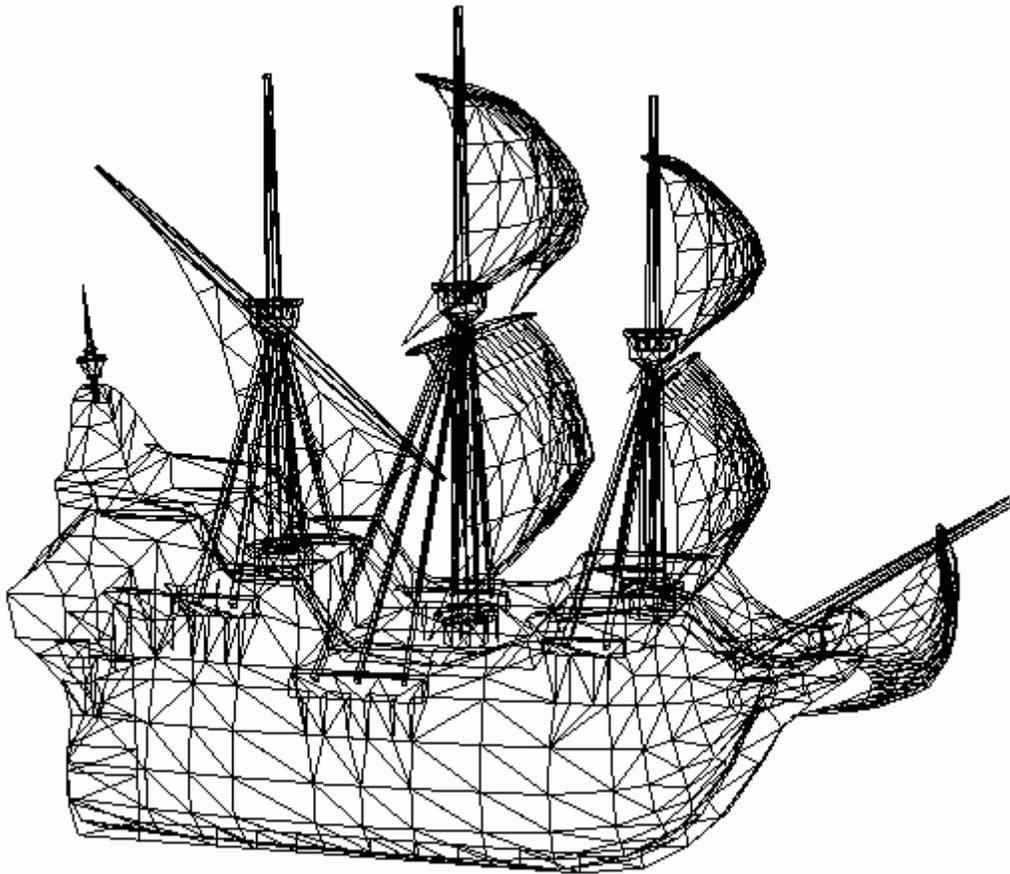
Dennis J Bouvier

K Computing

Getting Started with Java 3D

The Java 3D Tutorial

Table of Contents

List of Figures

List of Tables

List of Code Fragments

List of Reference Blocks

Preface to Chapter 3

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 are presented in the Module 0 document available at:
`http://java.sun.com/products/javamedia/3d/collateral`

Cover Image

The cover image is of a galleon as modeled in an `obj` file and rendered by Java 3D. The Java 3D program that produced the rendering, `ObjLoadApp.java,` is a modification of the `ObjLoad.java` example program discussed in Section 3.2. The program source code for `ObjLoad.java` and the galleon model file are available with the JDK 1.2 distribution.

CHAPTER 3
Easier Content Creation

$$T(dx, dy, dz) = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
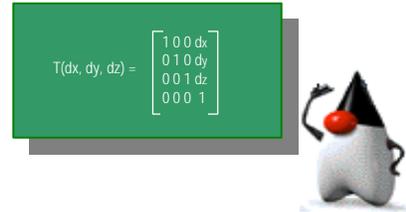
Chapter Objectives

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

- Use loader classes to load geometry from files into Java 3D worlds

- Use GeometryInfo to specify geometry as arbitrary polygons

- Use Text2D to add text to Java 3D worlds

- Use Text3D to add geometric text to Java 3D worlds

- Specify colors, images, or geometry for background

- Use BoundingLeaf for specifying bounds for backgrounds, behaviors, and lights

- Use the UserData field of SceneGraphObject class for a variety of applications

As the third chapter of the "Getting Started" Module, Chapter Three presents easier ways of creating visual content. Chapters one and two present the basic ways of creating virtual worlds, which includes creating visual objects from geometry classes. It only takes a little programming to learn that creating complex visual content one triangle at a time is tedious. Fortunately, there are a variety of easier ways to produce visual content. This chapter surveys a number of content creation methods and content issues beyond creating simple geometry.

3.1   What is in this Chapter

If you want to create a large or complex visual object, a great deal of code is required to just specify coordinates and normals. If you are concerned about performance, you spend more time, and code, to specify the geometry in as few triangle strips as possible. Geometry coding, fraught with details, can de a big sink on your development time. Fortunately, there are ways to create visual objects that require less code, resulting in fewer mistakes, and quiet often taking much less time.

Section 3.2 presents content loader classes, or "Loaders" as they are commonly referred to. Loaders, one alternative to hand coded geometry, create Java 3D visual objects from files created with 3D modeling

software. Loaders exist today for Alias `obj` files, VRML files, Lightwave files, Autocad `dfx` files, and a variety of other 3D file formats. New loaders are in development as well. The most important feature is the ability to write custom loaders for Java 3D.

Section 3.3 presents the GeometryInfo utility class, another alternative to hand coded geometry. GeometryInfo, along with the Triangulator, Stripifier, and NormalGeneration classes allows you to specify visual object geometry as arbitrary polygons. These classes convert the polygons to triangles, make strips of the triangles, and compute normals for the triangles at runtime, potentially saving you much coding time.

The next three sections present specific content creation techniques. Sections 3.4 and 3.5 present the Text2D utility and Text3D classes, respectively. These two classes represent two easy ways to add text to the contents of your virtual world. Section 3.6 presents the Background class. The Background class allows you to specify a color, image or geometry as the background for a virtual world.

The next two sections don't have as much to do with content. However, these are important topics. Section 3.7 presents the BoundingLeaf class. BoundingLeaf objects are useful in conjunction with backgrounds, behaviors, lights, fog, and other Java 3D classes that require a bounds specification. Section 3.8 discusses the use of the UserData field of SceneGraphObject class.

Of course, the Chapter concludes with a summary and Self-Test exercises for the adventurous.

3.2   Loaders

A loader class reads 3D scene files (not Java 3D files) and creates Java 3D representations of the file's contents that can be selectively added to a Java 3D world and augmented by other Java 3D code. The utility `com.sun.j3d.loaders` package provides the means for loading content from files created in other applications into Java 3D applications. Loader classes implement the loader interface defined in the utility `com.sun.j3d.loaders` package.

Since there are a variety of file formats for the purpose of representing 3D scenes (e.g., `.obj`, `.vrml`, etc.) and there will always be more file formats, the actual code to load a file is not part of Java 3D or of the loaders package; only the interface for the loading mechanism is included. With the interface definition, the Java 3D user can develop file loader classes with the same interface as other loader classes.

3.2.1   Simple Example of Using a Loader

Without a class that actually reads a file, it is not possible to load content from a file. With a loader class it is easy. Figure 3-1 presents the recipe for using a loader.

0.   find a loader (if one is not available, write one: see section 3.2.4)
1.   import the loader class for your file format (see Section 3.2.2 to find a loader)
2.   import other necessary classes
3.   declare a Scene variable (don't use the constructor)
4.   create a loader object
5.   load the file in a try block, assigning the result to the Scene variable
6.   insert the Scene into the scene graph

Figure 3-1 Recipe for Using a Loader

A loader example based on the ObjectFile class is distributed with JDK 1.2. It is found in `jdk1.2/demo/java3d/ObjLoad.` Code Fragment 3-1 presents a excerpt from the code from this demo.

The ObjectFile class is distributed in the `com.sun.j3d.loaders` package as an example file loader. Other loaders are available (some are listed in Table 3-1).

Class ObjectFile

Package: `com.sun.j3d.loaders`
Implements: `Loader`

The ObjectFile class implements the Loader interface for the Wavefront .obj file format, a standard 3D object file format created for use with Wavefront's Advanced Visualizer ™. Object Files are text based files supporting both polygonal and free-form geometry (curves and surfaces). The Java 3D .obj file loader supports a subset of the file format, but it is complete enough to load almost all commonly available Object Files. Free-form geometry is not supported.

Code Fragment 3-1 is annotated with numbers corresponding to the loader usage recipe given in Figure 3-1.

```
1.      import com.sun.j3d.loaders.objectfile.ObjectFile;        ❶
2.      import com.sun.j3d.loaders.ParsingErrorException;        ❷
3.      import com.sun.j3d.loaders.IncorrectFormatException;     ❷
4.      import com.sun.j3d.loaders.Scene;                        ❷
5.      import java.applet.Applet;
6.      import javax.media.j3d.*;
7.      import javax.vecmath.*;
8.      import java.io.*;                                        ❷
9.
10.     public class ObjLoad extends Applet {
11.
12.         private String filename = null;
13.
14.         public BranchGroup createSceneGraph() {
15.            // Create the root of the branch graph
16.            BranchGroup objRoot = new BranchGroup();
17.
18.     ❸     ObjectFile f = new ObjectFile();
19.     ❹     Scene s = null;
20.     ❺     try {
21.                s = f.load(filename);
22.            }
23.            catch (FileNotFoundException e) {
24.               System.err.println(e);
25.               System.exit(1);
26.            }
27.            catch (ParsingErrorException e) {
28.               System.err.println(e);
29.               System.exit(1);
30.            }
31.            catch (IncorrectFormatException e) {
32.               System.err.println(e);
33.               System.exit(1);
34.            }
35.
36.     ❻     objRoot.addChild(s.getSceneGroup());
37.         }
```

Code Fragment 3-1 An Excerpt from `jdk1.2/demo/java3d/ObjLoad/ObjLoad.java`.

This program goes on to add behaviors (the default spin, or the mouse interaction - covered in Chapter 4) and lights (Chapter 6) to provide a shaded rendering of the object model. Of course, you can do many

other things with the model in a Java 3D program such as add animations, add other geometry, change the color of the model, and so on.

A Lightwave loader example is available with the JDK 1.2 distribution and is found at `jdk1.2/demos/java3d/lightwave/Viewer.java`. This loader will load the lights and animations specified in a Lightwave `lws` file.

3.2.2   Publicly Available Loaders

A variety of loader classes exist for Java 3D. Table 3-1 lists file formats for which loaders are publicly available. At the time of this writing, at least one loader class is available for each of the file formats listed in Table 3-1.

Table 3-1 Publicly Available Java 3D Loaders

File Format	Description
3DS	3D-Studio
COB	Caligari trueSpace
DEM	Digital Elevation Map
DXF	AutoCAD Drawing Interchange File
IOB	Imagine
LWS	Lightwave Scene Format
NFF	WorldToolKit NFF format
OBJ	Wavefront
PDB	Protein Data Bank
PLAY	PLAY
SLD	Solid Works (prt and asm files)
VRT	Superscape VRT
VTK	Visual Toolkit
WRL	Virtual Reality Modeling Language

For a current list of loader classes, check the web. For that matter, loader classes can be downloaded from the web. Loaders can be found by following links from the Java 3D home page, or check the references section of this tutorial for a web address.

3.2.3   Loader Package Interfaces and Base Classes

The number and variety of loaders exist because the Java 3D designers made it easy to write a loader[1]. Loader classes are implementations of the Loader Interface which lowers the level of difficulty in writing a loader. More importantly, the interfaces make the various loader classes consistent in their interface.

As in the example, a program loading a 3D file actually uses both a loader and a scene object. The loader reads, parses, and creates the Java 3D representation of the file's contents. The scene object stores the scene graph created by the loader. It is possible to load scenes from more than one file (of the same format) using the same loader object creating multiple scene objects. Files of different formats can be combined in one Java 3D program using the appropriate loader classes.

The following reference block lists the interfaces in the `com.sun.j3d.loaders` package. A loader implements the loader interface and uses a class that implements the scene interface.

---

[1] Having the loader interface and base class actually makes it easy to write a loader for a simple file format. It also makes it possible to write a loader for a complex file format.

com.sun.j3d.loaders Interface Summary

Loader	The Loader interface is used to specify the location and elements of a file format to load.
Scene	The Scene interface is a set of methods used to extract Java 3D scene graph information from a file loader utility.

In addition to the interfaces, the `com.sun.j3d.loaders` package provides base implementations of the interfaces.

com.sun.j3d.loaders Class Summary

LoaderBase	This class implements the Loader interface and adds constructors. This class is extended by the authors of specific loader classes.
SceneBase	This class implements the Scene interface and extends it to incorporate methods used by loaders. This class is also used by programs that use loader classes.

The methods defined in the loader interface are used by programmers using loader classes.

Interface Loader Method Summary

Package: `com.sun.j3d.loaders`

The Loader interface is used to specify the location and elements of a file format to load. The interface is used to give loaders of various file formats a common public interface. Ideally the Scene interface will be implemented to give the user a consistent interface to extract the data.

`Scene load(java.io.Reader reader)`
This method loads the Reader and returns the Scene object containing the scene.

`Scene load(java.lang.String fileName)`
This method loads the named file and returns the Scene object containing the scene.

`Scene load(java.net.URL url)`
This method loads the named file and returns the Scene object containing the scene.

`void setBasePath(java.lang.String pathName)`
This method sets the base path name for data files associated with the file passed into the load(String) method.

`void setBaseUrl(java.net.URL url)`
This method sets the base URL name for data files associated with the file passed into the load(URL) method.

`void setFlags(int flags)`
This method sets the load flags for the file.

LOAD_ALL	This flag enables the loading of all objects into the scene.
LOAD_BACKGROUND_NODES	This flag enables the loading of background objects into the scene.
LOAD_BEHAVIOR_NODES	This flag enables the loading of behaviors into the scene.
LOAD_FOG_NODES	This flag enables the loading of fog objects into the scene.
LOAD_LIGHT_NODES	This flag enables the loading of light objects into the scene.
LOAD_SOUND_NODES	This flag enables the loading of sound objects into the scene.
LOAD_VIEW_GROUPS	This flag enables the loading of view (camera) objects into the scene.

LoaderBase class provides an implementation for each of the three `load()` methods of Interface Loader. LoaderBase also implements two constructors. Note the three loader methods return a Scene object.

LoaderBase Constructor Summary

Package: `com.sun.j3d.loaders`
Implements: `Loader`

This class implements the Loader interface. The author of a file loader would extend this class. The user of a file loader would use these methods.

LoaderBase()
Constructs a Loader with default values for all variables.

LoaderBase(int flags)
Constructs a Loader with the specified flags word.

In addition to the constructors listed in the reference block above, the methods listed in the following reference block are used by programmers using any loader class.

SceneBase Method Summary (partial list: loader users' methods)

In a departure from the normal formatting of a reference block, this reference block lists the methods.

```
Background[] getBackgroundNodes()
Behavior[] getBehaviorNodes()
java.lang.String getDescription()
Fog[] getFogNodes()
float[] getHorizontalFOVs()
Light[] getLightNodes()
java.util.Hashtable getNamedObjects()
BranchGroup getSceneGroup()
Sound[] getSoundNodes()
TransformGroup[] getViewGroups()
```

3.2.4   Writing a Loader

As I mentioned above, one of the most important features of loaders is that you can write your own - which means that all other Java 3D users can too! Space and time constraints do not permit me to go into details on writing a loader here. However, the point of this section is to outline the process. If you have no intention of writing a loader, at least not now, you can skip to the next section.

In writing a loader, the author of the new loader class extends the LoaderBase class defined in the `com.sun.j3d.loaders` package. The new loader class uses the SceneBase class of the same package.

There should be little need for future loaders to subclass SceneBase, or to implement Scene directly, as the functionality of a SceneBase is fairly straightforward. SceneBase class is responsible for both the storage and retrieval of data created by a loader while reading a file. The storage methods (used only by Loader authors) are all of the `add*` routines. The retrieval methods (used primarily by Loader users) are all of the `get*` routines.

Writing a file loader can be quite complex depending on the complexity of the file format. The hardest part is to parse the file. Of course, you have to start with the documentation for the file format you want to write a loader class for. Once the format is understood, begin by reading the loader and scene base classes. The new loader class will extend the loader base class and use the scene base class.

In extending the loader base class, most of the work will be writing methods that recognize the various types of content that can be represented in the file format. Each of these methods then create the

corresponding Java 3D scene graph component and store it in the scene object. The SceneBase constructor is listed in the following reference block. Other relevant reference blocks appear in the previous section.

SceneBase Constructor Summary

Package: `com.sun.j3d.loaders`

Implements: `Scene`

This class implements the Scene interface and extends it to incorporate utilities that could be used by loaders. This class is responsible for both the storage and retrieval of data from the Scene

SceneBase()
Create a SceneBase object - there should be no reason to use this constructor except in the implementation of a new loader class.

3.3   GeometryInfo

If you don't have access to geometric model files, or geometric modeling software, you have to create your geometry by hand. As mentioned in the chapter introduction, hand coding geometry often requires much time and is an error prone activity. As you know, when you specify geometry through the core classes, you are limited to triangles and quads. Using the GeometryInfo utiltity class can ease the time and tedium of geometry creation. Instead of specifying each triangle, you can specify arbitrary polygons, which can be concave, non-planer polygons - even with holes[2]. The GeometryInfo object, and other utility classes, convert the geometry into a triangular geometry that Java 3D can render.

For example, if you wanted to create a car in Java 3D, instead of specifying triangles, you can specify the profile of the car as a polygon in a GeometryInfo object. Then, using a Triangulator object, the polygon can be subdivided into triangles. The left image of Figure 3-2 shows a profile of a car as polygon. The right image is the polygon subdivided into triangles[3].
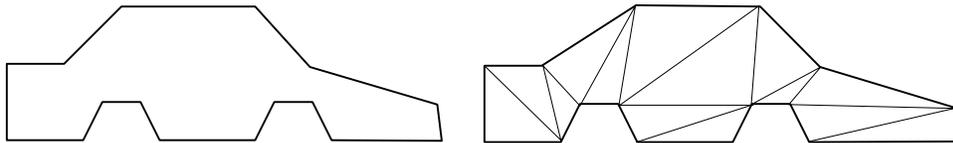
Figure 3-2 A GeometryInfo Polygon and One Possible Triangulation

If you are interested in performance, and who isn't, use a Stripifier object to convert the triangles to triangle strips. If you want to *shade* the visual object, use the NormalGenerator to calculate normals for the geometry[4].

---

[2] While you can specify non-planar polygons in GeometryInfo, and a Triangulator object will create a surface from it; non-planar contours do not specify a unique surface. In other words, if you specify a non-planar contour, you may not get the surface you want from Triangulator.

[3] Note that the figure does not necessarily represent the quality of the triangulation produced by the Triangulator class.

[4] If you are unfamiliar with the term *shade* as used in the context of computer graphics, check the glossary and read the introductory sections of Chapter 6.

An example program, GeomInfoApp.java, using the GeometryInfo, Triangulator, Stripifier, and NormalGeneration classes to create a car is included in the examples/easyContent directory. Figure 3-3 shows two renderings produced by GeomInfoApp.java.   In both renderings the blue outlines shows the contours specified in the GeometryInfo object. The red triangles (filled and shaded on the left, outline on the right) were computed by the GeometryInfo object with Triangulation, NormalGeneration, and Stripification done automatically.
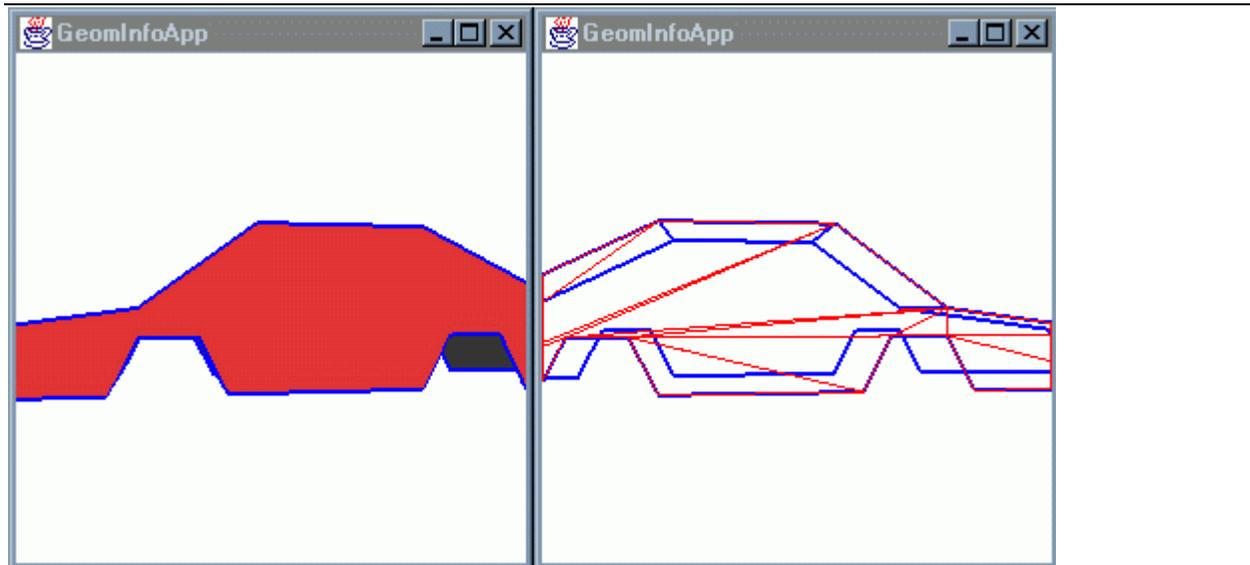
Figure 3-3 Two Renderings of a Car (facing opposite directions) Created Using GeometryInfo

A single planar polygon, similar to the one in shown in Figure 3-2, specifies the profile of the car (each side) in the GeomInfoApp example. Quadrilaterals specify the hood, roof, trunk lid, and other surfaces of the car.

3.3.1   Simple GeometryInfo Example

Using a GeometryInfo object is as easy as using core GeomertryArray classes, if not easier. In creating a GeomertyInfo object, simply specify the type of geometry you are going to need. The choices are POLYGON_ARRAY, QUAD_ARRAY, TRIANGLE_ARRAY, TRIANGLE_FAN_ARRAY, and TRIANGLE_STRIP_ARRAY.   Then set the coordinates and strip counts for the geometry. You don't have to tell the GeometryInfo object how many coordinates are in the data; it will be automatically calculated.

Code Fragment 3-2 shows an example GeometryInfo application. Lines 1 through 3 of Code Fragment 3-2 show creating a GeometryInfo object and the initial geometry specification.

After having created the GeometryInfo object, the other classes may be used. If you want to use the NormalGenerator, for example, first create a NormalGenerator object, then pass the GeometryInfo object to it. Lines 8 and 9 of Code Fragment 3-2 do just that.

```
1.         GeometryInfo gi = new GeometryInfo(GeometryInfo.POLYGON_ARRAY);
2.         gi.setCoordinates(coordinateData);
3.         gi.setStripCounts(stripCounts);
4.
5.         Triangulator tr = new Triangulator();
6.         tr.triangulate(gi);
7.
8.         NormalGenerator ng = new NormalGenerator();
9.         ng.generateNormals(gi);
10.
11.        Stripifier st = new Stripifier();
12.        st.stripify(gi);
13.
14.        Shape3D part = new Shape3D();
15.        part.setAppearance(appearance);
16.        part.setGeometry(gi.getGeometryArray());
```

Code Fragment 3-2 Using GeometryInfo, Triangulator, NormalGenerator, and Stripifier Utilities.

3.3.2   Classes for GeometryInfo

The GeometryInfo and related classes are members of the `com.sun.j3d.util.geometry` package and are subclasses of Object. Figure 3-4 shows the hierarchy for these classes.

```
java.lang.Object
    com.sun.j3d.utils.geometry.GeometryInfo
    com.sun.j3d.utils.geometry.NormalGenerator
    com.sun.j3d.utils.geometry.Stripifier
    com.sun.j3d.utils.geometry.Triangulator
```
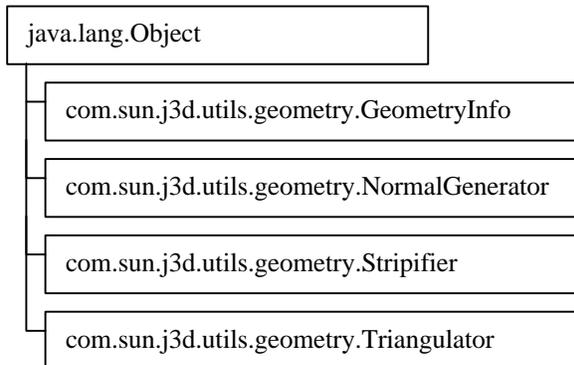
Figure 3-4 Class Hierarchy for the GeometryInfo Utility Class, and Related Classes

The GeometryInfo class has only one constructor and in this constructor you specify the kind of geometry to be specified by the coordinate data. The following reference block gives more detail.

GeometryInfo Constructor Summary

Package: `com.sun.j3d.utils.geometry`
Extends: `java.lang.Object`

The GeometryInfo object is where you put your geometry if you want to use the Java 3D utility libraries. Once you have your data in the GeometryInfo object, you can send it to any (or all) of several utilities to have operations performed on it, such as generating normals or turning it into long strips for more efficient rendering ("stripifying"). Geometry is loaded just as it is in the Java 3D GeometryArray object, but there are fewer options for getting data into the object. GeometryInfo itself contains some simple utilities, such as calculating indices for non-indexed data ("indexifying") and getting rid of unused data in your indexed geometry information ("compacting").

`GeometryInfo(int primitive)`
Construct a GeometryInfo object, where primitive is one of
`POLYGON_ARRAY`	possibly multi-contour, possibly non-planar polygons
`QUAD_ARRAY`	each set of four vertices forms an independent quad
`TRIANGLE_ARRAY`	each set of three vertices form an independent triangle
`TRIANGLE_FAN_ARRAY`	the stripCounts array indicates how many vertices to use for each triangle fan.
`TRIANGLE_STRIP_ARRAY`	that the stripCounts array indicates how many vertices to use for each triangle strip.

The GeometryInfo class has many methods. Most methods are for setting (or getting) coordinate, color, index, normal, or texture coordinate data. Most applications will only use a few of available methods. However, it is convienient to be able to specify geometry to any level of detail and have the rest computed.

GeometryInfo Method Summary (partial list)

void recomputeIndices()
Redo indexes to guarantee connection information.

void reverse()
Reverse the order of all lists.

void setColorIndices(int[] colorIndices)
Sets the array of indices into the Color array.

void setColors(Color3f[] colors)
Sets the colors array.

void setColors(Color4f[] colors)
Sets the colors array. There are other setColors methods.

void setContourCounts(int[] contourCounts)
Sets the list of contour counts.

void setCoordinateIndices(int[] coordinateIndices)
Sets the array of indices into the Coordinate array.

void setCoordinates(Point3f[] coordinates)
Sets the coordinates array.

void setCoordinates(Point3d[] coordinates)
Sets the coordinates array. There are other setCoordinates methods.

void setNormalIndices(int[] normalIndices)
Sets the array of indices into the Normal array.

void setNormals(Vector3f[] normals)
Sets the normals array.

void setNormals(float[] normals)
Sets the normals array.

void setStripCounts(int[] stripCounts)
Sets the array of strip counts.

void setTextureCoordinateIndices(int[] texCoordIndices)
Sets the array of indices into the TextureCoordinate array.

void setTextureCoordinates(Point2f[] texCoords)
Sets the TextureCoordinates array. There are other setTextureCoordinates methods.

Each of the GeometryInfo 'helper' classes are used in a way similar to the GeometryInfo class. The following reference blocks show the constructors and methods for Triangulator, Stripifier, and NormalGenerator, in that order. This is the order in which they would be used for a POLYGON_ARRAY.

The Triangulator utility is only used with POLYGON_ARRAY geometry. GeometryInfo objects with other primitive geometry would only use Stripifier and NormalGenerator, as appropriate.

The default constructor of the Triangulator class simply creates a Triangulation object. See the reference block for more information.

Triangulator Constructor Summary

Package: `com.sun.j3d.utils.geometry`
Extends: `java.lang.Object`

Triangulator is a utility for turning arbitrary polygons into triangles so they can be rendered by Java 3D. Polygons can be concave, nonplanar, and can contain holes (see GeometryInfo).

`Triangulator()`
Creates a new instance of the Triangulator.

The only method of the Triangulator class is to triangulate a polygon array GeometryInfo object.

Triangulator Method Summary

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

The only constructor of the Stripifier class creates a stripification object[5].

Stripifier Constructor Summary

Package: `com.sun.j3d.utils.geometry`
Extends: `java.lang.Object`

The Stripifier utility will change the primitive of the GeometryInfo object to Triangle Strips. The strips are made by analyzing the triangles in the original data and connecting them together.

For best results Normal Generation should be performed on the GeometryInfo object before Stripification.

`Stripifier()`
Creates the Stripifier object.

The only method of the Stripifier class is to stripify the geometry of a GeometryInfo class.

Stripifier Method Summary

`void stripify(GeometryInfo gi)`
Turn the geometry contained in the GeometryInfo object into an array of Triangle Strips.

The NormalGenerator class has two constructors. The first constructs a NormalGenerator with a default value for the crease angle. The second constructor allows the specification of a crease angle with the construction of the NormalGenerator object. See the reference block below.

---

[5] Each paragraph introduces a new word into the English language.

NormalGenerator Constructor Summary

Package: `com.sun.j3d.utils.geometry`
Extends: `java.lang.Object`

The NormalGenerator utility will calculate and fill in the normals of a GeometryInfo object. The calculated normals are estimated based on an analysis of the indexed coordinate information. If your data isn't indexed, index lists will be created.

If two (or more) triangles in the model share the same coordinate index then the normal generator will attempt to generate one normal for the vertex, resulting in a "smooth" looking surface. If two coordinates don't have the same index then they will have two separate normals, even if they have the same position. This will result in a "crease" in your object. If you suspect that your data isn't properly indexed, call GeometryInfo.recomputeIndexes().

Of course, sometimes your model has a crease in it. If two triangles' normals differ by more than creaseAngle, then the vertex will get two separate normals, creating a discontinuous crease in the model. This is perfect for the edge of a table or the corner of a cube, for instance.

NormalGenerator()
Construct a NormalGenerator with the default crease angle (0.76794 radians, or 44°).

NormalGenerator(double radians)
Construct a NormalGenerator with a specified crease angle in radians.

The methods for the NormalGenerator class include ones for setting and getting the crease angle, and computing normals for the geometry of a GeometryInfo object. See the NormalGenerator Constructor Summary reference block for a discussion of crease angle.

NormalGenerator Method Summary

void generateNormals(GeometryInfo geom)
Generate normals for the GeometryInfo object.

double getCreaseAngle()
Returns the current value of the crease angle, in radians.

void setCreaseAngle(double radians)
Set the crease angle in radians.

3.4   Text2D

There are two ways to add text to a Java 3D scene. One way uses the Text2D class and the other way uses the Text3D class. Obviously, one significant difference is that Text2D objects are two dimensional and Text3D objects are three dimensional. Another significant difference is how these objects are created.

Text2D objects are rectangular polygons with the text applied as a texture (texturing is the subject of Chapter 7). Text3D objects are 3D geometric objects created as an extrusion of the text.   See Section 3.5 for more information on the Text3D class and related classes.

As a subclass of Shape3D, instances of Text2D can be children of group objects. To place a Text2D object in a Java 3D scene, simply create the Text2D object and add it to the scene graph. Figure 3-5 presents this simple recipe.

1.   Create a Text2D object
2.   Add it to the scene graph

Figure 3-5 Recipe for Text2D

Text2D objects are implemented using a polygon and a texture. The polygon is transparent so that only the texture is visible. The texture is of the text string set in the named typeface with the specified font parameters[6]. The typefaces available on your system will vary. Typically, Courier, Helvetica, and TimesRoman typefaces will be available, among others. Any font available in the AWT is available to you for Text2D (and Text3D) applications. Using a Text2D object is straightforward as demonstrated in the next section.

3.4.1   Simple Text2D Example

Code Fragment 3-3 shows an example of adding a Text2D object to a scene. The Text2D object is created on lines 21 through 23. In this constructor, the text string, color, typeface, size, and font style are specified. The Text2D object is added to the scene graph on line 24. Note the import statement for Font (line 5) used for the font style constants.

```
1.      import java.applet.Applet;
2.      import java.awt.BorderLayout;
3.      import java.awt.Frame;
4.      import java.awt.event.*;
5.      import java.awt.Font;
6.      import com.sun.j3d.utils.applet.MainFrame;
7.      import com.sun.j3d.utils.geometry.Text2D;
8.      import com.sun.j3d.utils.universe.*;
9.      import javax.media.j3d.*;
10.     import javax.vecmath.*;
11.
12.     //   Text2DApp renders a single Text2D object.
13.
14.     public class Text2DApp extends Applet {
15.
16.        public BranchGroup createSceneGraph() {
17.           // Create the root of the branch graph
18.           BranchGroup objRoot = new BranchGroup();
19.
20.           // Create a Text2D leaf node, add it to the scene graph.
21.           Text2D text2D = new Text2D("2D text is a textured polygon",
22.                                      new Color3f(0.9f, 1.0f, 1.0f),
23.                                      "Helvetica", 18, Font.ITALIC));
24.           objRoot.addChild(text2D);
```

Code Fragment 3-3 A Text2D Object Created (excerpt from Text2DApp.java)

`Text2DApp.java` is a complete program that includes the above Code Fragment. In this example, the Text2D object rotates about the origin in the scene. As the application runs you can see, by default, the textured polygon is invisible when viewed from behind.

---

[6] A font is a specific typeface set at a size with a set of font style attributes. Refer to the glossary for definitions of font and typeface.
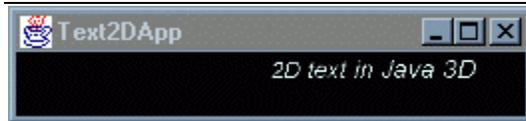
Figure 3-6 Image from Text2DApp.java

Some attributes of a Text2D object can be changed by modifying the referenced appearance bundle and/or Geometry NodeComponent. Code Fragment 3-4 shows the code to make **text2d,** the Text2D object created in Code Fragment 3-3, two-sided through modification of its appearance bundle.

```
25.      Appearance textAppear = text2d.getAppearance();
26.
27.      // The following 4 lines of code make the Text2D object 2-sided.
28.      PolygonAttributes polyAttrib = new PolygonAttributes();
29.      polyAttrib.setCullFace(PolygonAttributes.CULL_NONE);
30.      polyAttrib.setBackFaceNormalFlip(true);
31.      textAppear.setPolygonAttributes(polyAttrib);
```

Code Fragment 3-4 Making a Two-sided Text2D Object.

The texture created by one Text2D object can be applied to other visual objects as well. Since the application of textures to visual objects is the subject of Chapter 7, the details on this are left until then.

3.4.2   Classes Used in Creating Text2D Objects

The only class needed is the Text2D class. As you can see from Figure 3-7, Text2D is a utility class which extends Shape3D.
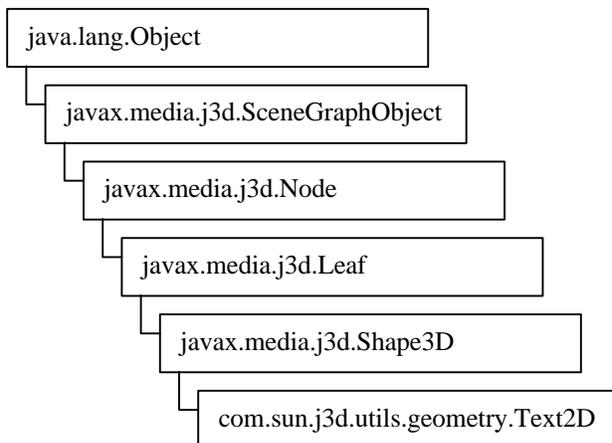

Figure 3-7 The Class Hierarchy for Text2D

In the Text2D constructor, the text string, typeface, font size, and font style are specified.

Text2D Constructor Summary

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

This class creates a texture-mapped rectangle which displays the text string sent in by the user, given the appearance parameters also supplied by the user. The size of the rectangle (and its texture map) is determined by the font parameters passed in to the constructor. The resulting Shape3D object is a transparent (except for the text) rectangle located at (0, 0, 0) and extending up the positive y-axis and out the positive x-axis.

```
Text2D(java.lang.String text, Color3f color, java.lang.String fontName, int
fontSize, int fontStyle)
```
Constructor.

With the Text2D constructor, there is one method. This method sets a scale factor to create Text2D objects larger or smaller than the specified point size. This method is not useful in version 1.1.x of the API, since it is only utilized when the text is specified. In version 1.2 a `setText()` method will be introduced making the `setRectangleScaleFactor()` useful.

Text2D Method Summary

```
void setRectangleScaleFactor(float newScaleFactor)
```
Sets the scale factor used in converting the image width/height to width/height values in 3D.

3.5 Text3D

Another way to add text to a Java 3D virtual world is to create a Text3D object for the text. Where Text2D creates text with a texture, Text3D creates text using geometry. The textual geometry of a Text3D object is an extrusion of the font.

Creating a Text3D object is a little more involved than creating a Text2D object. The first step is to create a Font3D object of the desired typeface, size, and font style. Then a Text3D object for a particular string is made using the Font3D object. Since the Text3D class is a subclass of Geometry, the Text3D object is a NodeComponent that is referenced by one or more Shape3D object(s). Figure 3-8 summarizes the process of adding Text3D objects to a scene graph.

1. Create a Font3D object from an AWT Font
2. Create Text3D for a string using the Font3D object, optionally specifying a reference point
3. Reference the object from a Shape3D object added to the scene graph

Figure 3-8 Recipe for Creating a Text3D Object

3.5.1 Simple Text3D Example

Code Fragment 3-5 shows the basic construction of a Text3D object. The Font3D object is created on lines 19 and 20. The typeface used here is "Helvetica". Just like with Text2D, any typeface available in the AWT can be used for Font3D and therefore Text3D objects. This Font3D constructor (lines 19 and 20 of Code Fragment 3-5) also sets the font size to 10 points and uses the default extrusion.

The statement on lines 21 and 22 create a Text3D object using the newly created Font3D object for the string "3DText" while specifying a reference point for the object. The last two statements create a Shape3D object for the Text3D object and add it to the scene graph. Note the import statement for Font (line 5) is necessary since a Font object is used in the Font3D creation.

```
1.      import java.applet.Applet;
2.      import java.awt.BorderLayout;
3.      import java.awt.Frame;
4.      import java.awt.event.*;
5.      import java.awt.Font;
6.      import com.sun.j3d.utils.applet.MainFrame;
7.      import com.sun.j3d.utils.universe.*;
8.      import javax.media.j3d.*;
9.      import javax.vecmath.*;
10.
11.     //   Text3DApp renders a single Text3D object.
12.
13.     public class Text3DApp extends Applet {
14.
15.       public BranchGroup createSceneGraph() {
16.         // Create the root of the branch graph
17.         BranchGroup objRoot = new BranchGroup();
18.
19.         Font3D font3d = new Font3D(new Font("Helvetica", Font.PLAIN, 10),
20.                         new FontExtrusion());
21.         Text3D textGeom = new Text3D(font3d, new String("3DText"),
22.                             new Point3f(-2.0f, 0.0f, 0.0f));
23.         Shape3D textShape = new Shape3D(textGeom);
24.         objRoot.addChild(textShape);
```

Code Fragment 3-5 Creating a Text3D Visual Object

Figure 3-9 shows a Text3D object illuminated to illustrate the extrusion of the type. In the figure, the extrusion is shown in gray while the type is shown in black. To recreate this figure in Java 3D, a Material object and a DirectionalLight is necessary. Since Chapter 6 covers these topics, they are not discussed here. You can't set the color of the individual vertices in the Text3D object since you don't have access to the geometry of the Text3D object.
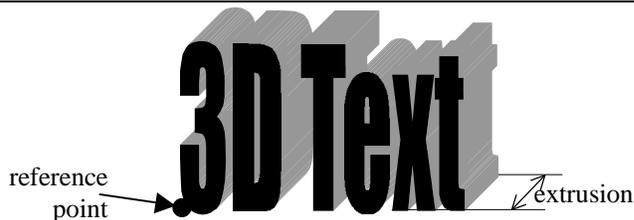
Figure 3-9 The Default Reference Point and Extrusion for a 3DText Object

The text of a Text3D object can be oriented in a variety of ways. The orientation is specified as a path direction. The choices are right, left, up, and down. See Table 3-2 and the Text3D reference blocks (beginning on page 3-19) for more information.

Each Text3D object has a reference point. The reference point for a Text3D object is the origin of the object. The reference point for each object is defined by the combination of the path and alignment of the text. Table 3-2 shows the effects of path and alignment specification on the orientation of the text and placement of the reference point.

The placement of the reference point can be defined explicitly overriding the path and alignment positioning. See the Text3D reference blocks (beginning on page 3-19) for more information.

Table 3-2 The Orientation of Text and Position of the Reference Point for Combinations of Text3D Alignment and Path

	ALIGN_FIRST (default)	ALIGN_CENTER	ALIGN_LAST
PATH_RIGHT (default)	●Text3D	Text3D	Text3D●
PATH_LEFT	D3txeT●	D3txeT	●D3txeT
PATH_DOWN	●T e x t	T e ●x t	T e x ●t
PATH_UP	t x e ●T	t x ●e T	●t x e T

Text3D objects have normals. The addition of an appearance bundle that includes a Material object to a Shape3D object referencing Text3D geometry will enable lighting for the Text3D object.

3.5.2   Classes Used in Creating Text3D Objects

This section presents reference material for the three classes used in creating Text3D objects: Text3D, Font3D, and FontExtrusion, in that order. Figure 3-10 shows the class hierarchy for Text3D class.
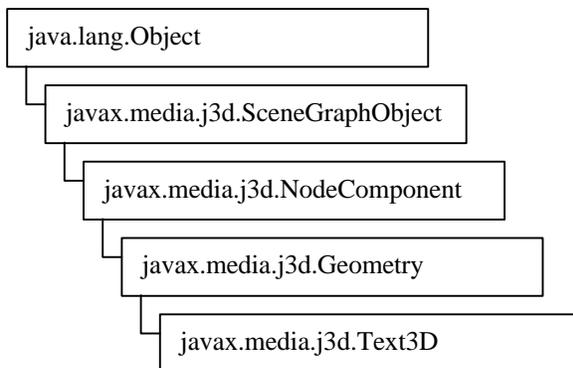
Figure 3-10 Class Hierarchy for Text3D

The Text3D class defines a number of constructors. Each constructor allows you to specify none, some, or all of the attributes of a Text3D object. The following reference block lists the constructors, along with the default values, for Text3D.

Text3D Constructor Summary

A Text3D object is a text string that has been converted to 3D geometry. The Font3D object determines the appearance of the Text3D NodeComponent object. Each Text3D object has a position - a reference point placing the Text3D object. The 3D text can be placed around this position using different alignments and paths.

Text3D()
Creates an empty Text3D object. The default values used for this, and other constructors as appropriate, are:

```
font 3D              null
string               null
position             (0,0,0)
alignment            ALIGN_FIRST
path                 PATH_RIGHT
character spacing  0.0
```

Text3D(Font3D font3D)
Creates a Text3D object with the given Font3D object.

Text3D(Font3D font3D, String string)
Creates a Text3D object given a Font3D object and a string.

Text3D(Font3D font3D, String string, Point3f position)
Creates a Text3D object given a Font3D object and a string.   The position point defines a reference point for the Text3D object. Its position is defined relative to the lower left front corner of the geometry.

Text3D(Font3D font3D, String string, Point3f position,
int alignment, int path)
Creates a Text3D object given a Font3D object and a string.

    ALIGN_CENTER alignment: the center of the string is placed on the position point.
    ALIGN_FIRST   alignment: the first character of the string is placed on the position point.
    ALIGN_LAST   alignment: the last character of the string is placed on the position point.
    PATH_DOWN   path: succeeding glyphs are placed below the current glyph.
    PATH_LEFT   path: succeeding glyphs are placed to the left of the current glyph.
    PATH_RIGHT   path: succeeding glyphs are placed to the right of the current glyph.
    PATH_UP   path: succeeding glyphs are placed above the current glyph.
See Table 3-2 for examples.

The Text3D class also defines a number of methods. Each allows you to modify (set) the attributes of the Text3D object. This class also defines corresponding `get*` methods. The following reference block lists the `set*` methods for the Text3D class.

Text3D Method Summary

`void setAlignment(int alignment)`
Sets the text alignment policy for this Text3D NodeComponent object.

`void setCharacterSpacing(float characterSpacing)`
Sets the character spacing to be used when constructing the Text3D string.

`void setFont3D(Font3D font3d)`
Sets the Font3D object used by this Text3D NodeComponent object.

`void setPath(int path)`
Sets the node's path direction.

`void setPosition(Point3f position)`
Sets the node's reference point to the supplied parameter.

`void setString(java.lang.String string)`
Copies the character string from the supplied parameter into the Text3D node.

The following reference block lists the Capabilities of the Text3D class.

Text3D Capabilities Summary

**`ALLOW_ALIGNMENT_READ	WRITE`**	allow reading (writing) the text alignment value.
`ALLOW_BOUNDING_BOX_READ`	allow reading the text string bounding box value
**`ALLOW_CHARACTER_SPACING_READ	WRITE`**	allow reading (writing) the text character spacing value.
**`ALLOW_FONT3D_READ	WRITE`**	allow reading (writing) Font3D component information.
**`ALLOW_PATH_READ	WRITE`**	allow reading (writing) the text path value.
**`ALLOW_POSITION_READ	WRITE`**	allow reading (writing) the text position value.
**`ALLOW_STRING_READ	WRITE`**	allow reading (writing) the String object.

Each Text3D object is created from a Font3D object. A single Font3D object can be used to create an unlimited number of Text3D objects[7]. A Font3D object holds the extrusion geometry for each glyph in the typeface. A Text3D object copies the geometry to form the specified string. Font3D objects can be garbage collected without affecting Text3D objects created from them.

The following reference block shows the constructor for the Font3D class.

---

[7] Of course, memory constraints will limit the actual number of Text3D objects to some number significantly less than infinity.

Font3D Constructor Summary

Extends: `java.lang.Object`

A 3D Font consists of a Java 2D font and an extrusion path. The extrusion path describes how the edge of a glyph varies in the Z axis. The Font3D object is used to store extruded 2D glyphs. These 3D glyphs can then be used to construct Text3D NodeComponent objects. Custom 3D fonts as well as methods to store 3D fonts to disk will be addressed in a future release.

See Also: java.awt.Font, FontExtrusion, Text3D

Font3D(java.awt.Font font, FontExtrusion extrudePath)
Creates a Font3D object from the specified Font object.

The following reference block lists the methods of the Font3D class. Normally, `get*` methods are not listed in reference blocks in this tutorial. However, since the Font3D class has no `set*` methods, the Font3D get* methods are listed here. The effect of a `set*` method would be essentially the same as invoking a constructor, so to keep the class smaller, no `set*` methods are defined.

Font3D Method Summary

void getBoundingBox(int glyphCode, BoundingBox bounds)
Returns the 3D bounding box of the specified glyph code.

java.awt.Font getFont()
Returns the Java 2D Font used to create this Font3D object.

void getFontExtrusion(FontExtrusion extrudePath)
Copies the FontExtrusion object used to create this Font3D object into the specified parameter.

The Font class is used in creating a Font3D object. The following reference block lists one constructor for Font. Other constructors and many methods are not listed in this tutorial. See the Java 3D API Specification for details.

Font Constructor Summary (partial list)

Package: `java.awt`

An AWT class that creates an internal representation of fonts. Font extends java.lang.Object.

public Font(String name, int style, int size)
Creates a new Font from the specified name, style and point size.
    Parameters:
      name - the typeface name. This can be a logical name or a typeface name. A logical name must be one of: Dialog, DialogInput, Monospaced, Serif, SansSerif, or Symbol.

      style - the style constant for the Font. The style argument is an integer bitmask that may be PLAIN, or a bitwise union of BOLD and/or ITALIC (for example, `Font.ITALIC` or `Font.BOLD|Font.ITALIC`). Any other bits set in the style parameter are ignored. If the style argument does not conform to one of the expected integer bitmasks then the style is set to PLAIN.

      size - the point size of the Font

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

FontExtrusion Constructor Summary

Extends: `java.lang.Object`

The FontExtrusion object is used to describe the extrusion path for a Font3D object. The extrusion path is used in conjunction with a Font2D object. The extrusion path defines the edge contour of 3D text. This contour is perpendicular to the face of the text. The extrusion has its origin at the edge of the glyph with 1.0 being the height of the tallest glyph. Contour must be monotonic in x. User is responsible for data sanity and must make sure that extrusionShape does not cause intersection of adjacent glyphs or within single glyph. The output is undefined for extrusions that cause intersections.

FontExtrusion()
Constructs a FontExtrusion object with default parameters.

FontExtrusion(java.awt.Shape extrusionShape)
Constructs a FontExtrusion object with the specified shape.

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

FontExtrusion Method Summary

java.awt.Shape getExtrusionShape()
Gets the FontExtrusion's shape parameter.

void setExtrusionShape(java.awt.Shape extrusionShape)
Sets the FontExtrusion's shape parameter.

3.6   Background

By default, the background of a Java 3D virtual universe is solid black. However, you can specify other backgrounds for your virtual worlds. The Java 3D API provides an easy way to specify a solid color, an image, geometry, or a combination of these, for a background.

When specifying an image for the background, it overrides a background color specification, if any. When geometry is specified, it is drawn on top of the background color or image.

The only tricky part is in the specification of a geometric background. All background geometry is specified as points on a unit sphere. Whether your geometry is a PointArray, which could represent stars light years away, or a TriangleArray, which could represent mountains in the distance, all coordinates are specified at a distance of one unit. The background geometry is projected to infinity when rendered.

Background objects have Application bounds which allows different backgrounds to be specified for different regions of the virtual world. A Background node is active when its application region intersects the ViewPlatform's activation volume.

If multiple Background nodes are active, the Background node that is "closest" to the eye will be used. If no Background nodes are active, then the window is cleared to black. However, the definition of "closest" is not specified. For closest, the background with the innermost application bounds that encloses the ViewPlatform is chosen.

It is unlikely that your application will need lit background geometry - in reality the human visual system can't perceive visual detail at great distances. However, a background geometry can be shaded. The background geometry subgraph may not contain Lights, but Lights defined in the scene graph can influence background geometry.

To create a background, follow the simple recipe given in Figure 3-11. Example backgrounds are presented in the next section.

---

1.   Create Background object specifying a color or an image
2.   Add geometry (optional)
3.   Provide an Application Boundary or BoundingLeaf
4.   Add the Background object to the scene graph

Figure 3-11 Recipe for Backgrounds

3.6.1   Background Examples

As explained in the previous section, a background can have either a color or an image. Geometry can appear in the background with either the color or image. This section provides an example of a solid white background. A second example shows adding geometry to a background.

Colored Background Example

As shown in Figure 3-11, the recipe for creating a solid color background is straightforward. The lines of code in Code Fragment 3-6 correspond to the recipe steps. The besides customizing the color, the only other possible adjustment to this code would be to define a more appropriate application bounds for the background (or use a BoundingLeaf).

```
1.     Background backg = new Background(1.0f, 1.0f, 1.0f);
2.     //
3.     backg.setApplicationBounds(BoundingSphere());
4.     contentRoot.addChild(backg);
```

Code Fragment 3-6 Adding a Colored Background

Geometry Background Example

Once again, the lines of code in Code Fragment 3-7 correspond to the background recipe steps shown in Figure 3-11. In this code fragment, the `createBackGraph()` method is invoked to create the background geometry. This method returns a BranchGroup object. For a more complete example, see `BackgroundApp.java` in the `examples/easyContent` directory.

```
1.     Background backg = new Background();      //black background
2.     backg.setGeometry(createBackGraph());     // add BranchGroup of background
3.     backg.setApplicationBounds(new BoundingSphere(new Point3d(), 100.0));
4.     objRoot.addChild(backg);
```

Code Fragment 3-7 Adding a Geometric Background

BackgroundApp.java

To appreciate a Background, you need to experience it. `BackgroundApp.java,` a program included in the examples/easyContent directory, is a complete working application with a geometric background. This application allows you to move in the Java 3D virtual world. While moving, you can see the relative movement between the local geometry and the background geometry.

BackgroundApp uses the KeyNavigatorBehavior class provided in the utility library for viewer motion. Interaction (as implemented through behaviors) is the subject of Chapter 4, so the details of this programming is delayed until then.

KeyNavigatorBehavior responds to the arrow keys, PgUp, and PgDn keys for motion. The Alt key also plays a role (more details in Chapter 4). When you run BackgroundApp, be sure to rotate to find the "constellation", as well as travel far into the distance.
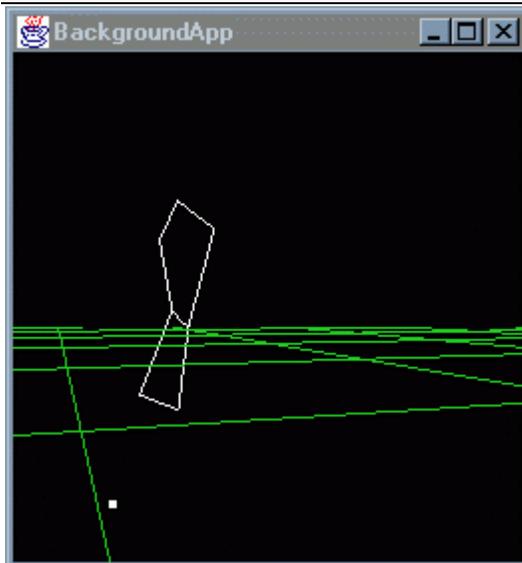
Figure 3-12 Viewing the "Constellation" in the Background of BackgroundApp.java

3.6.2   Background Class

Figure 3-13 shows the class hierarchy for Background class. As an extension of Leaf class, an instance of Background class can be a child of a Group object.
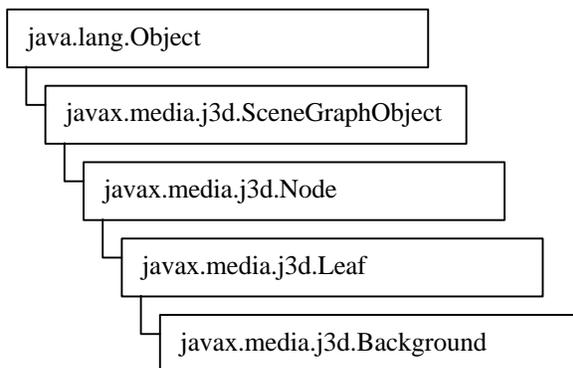
Figure 3-13 The Class Hierarchy for Background

Background has a variety of constructors. Background constructors with parameters allow the specification of a color or an image for a background. The following reference block gives more detail. Background Geometry can only be specified through the appropriate method.

Background Constructor Summary

The Background leaf node defines either a solid background color or a background image that is used to fill the window at the beginning of each new frame. It optionally allows background geometry to be referenced. Background geometry must be pre-tessellated onto a unit sphere and is drawn at infinity. It also specifies an application region in which this background is active.

Background()
Constructs a Background node with a default color (black).

Background(Color3f color)
Constructs a Background node with the specified color.

Background(float r, float g, float b)
Constructs a Background node with the specified color.

Background(ImageComponent2D image)
Constructs a Background node with the specified image.

Any attribute of a Background can be set through a method. The following reference block lists the methods of the Background class.

Background Method Summary

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

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

void setColor(Color3f color)
Sets the background color to the specified color.

void setColor(float r, float g, float b)
Sets the background color to the specified color.

void setGeometry(BranchGroup branch)
Sets the background geometry to the specified BranchGroup node.

void setImage(ImageComponent2D image)
Sets the background image to the specified image.

The following reference block lists the capability bits of the Background class.

Background Capabilities Summary
ALLOW_APPLICATION_BOUNDS_READ | WRITE   allow read (write) access to its application bounds
ALLOW_COLOR_READ | WRITE   allow read (write) access to its color
ALLOW_GEOMETRY_READ | WRITE   allow read (write) access to its background geometry
ALLOW_IMAGE_READ | WRITE   allow read (write) access to its image

3.7   BoundingLeaf

Bounds are used with lights, behaviors, backgrounds, and a variety of other applications in Java 3D. Bounds allow the programmer to vary action, appearance, and/or sound over the virtual landscape. Bounds specification also allows the Java 3D rendering system to perform execution culling, thereby improving rendering performance[8].

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

For example, if a world includes a stationary light source that illuminates moving objects, the bounds for the light must include the moving object. One way to handle this would be to make the bounds large enough to include all of the places the object moves. This is not the best answer in most cases. A better solution is to use a BoundingLeaf. Placed in the scene graph with the visual object, the BoundingLeaf object moves with the visual object and independently of the light source. Figure 3-14 shows a scene graph in which a light uses a BoundingLeaf node.
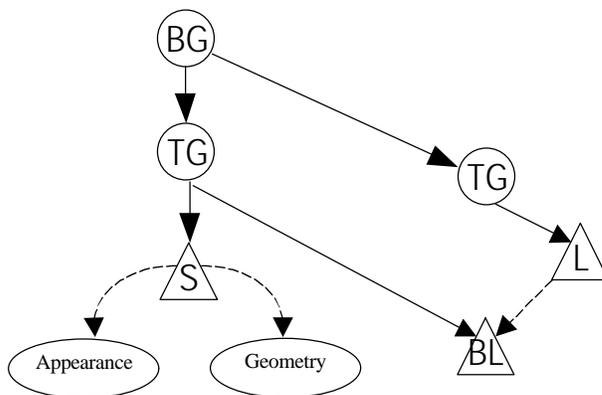
Figure 3-14 BoundlingLeaf Moves with a Visual Object and Independently of a Light Source

While the applications for BoundingLeaf include the ApplicationBounds of Background objects (Section 3.5), the SchedulingBounds of Behaviors (Section 4.2), and the InfluencingBounds of Lights (Section 6.6), it would not make sense to repeat this information in all three places[9].

---

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

[9] BoundingLeaf nodes are also useful for fog, clip, sound, and soundscape objects, which are not covered in this tutorial.

One interesting application of a BoundingLeaf object places a BoundingLeaf in the viewPlatform. This BoundingLeaf can be used for an "always on" scheduling bounds for a Behavior, or "always applies" application bounds for Background or Fog. Code Fragment 3-8 presents an example "always applies" BoundingLeaf application used with a Background. Another example application of a BoundingLeaf is presented in Section 6.6.

Code Fragment 3-8 presents an example of adding a BoundingLeaf as a child of the PlatformGeometry to provide an "always applies" bounds for a Background[10]. In this code, the standard `createSceneGraph()` method is modified to takes a single parameter, that of the SimpleUniverse object[11]. This is necessary for creating a PlatformGeometry object.

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

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

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

Code Fragment 3-8 Adding a BoundingLeaf to the View Platform for an "Always-On" Bounds

3.7.1   BoundingLeaf Class

The BoundingLeaf class extends Leaf class. Figure 3-15 presents the complete class hierarchy for BoundingLeaf.

---

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

[11] The createSceneGraph() method is only standard in that is keeps appearing in the examples of this tutorial.

```
┌─────────────────────────────┐
│ java.lang.Object            │
└─────────────────────────────┘
   ┌────────────────────────────────────────┐
   │ javax.media.j3d.SceneGraphObject        │
   └────────────────────────────────────────┘
      ┌──────────────────────────────────┐
      │ javax.media.j3d.Node              │
      └──────────────────────────────────┘
         ┌──────────────────────────────────┐
         │ javax.media.j3d.Leaf              │
         └──────────────────────────────────┘
            ┌──────────────────────────────────┐
            │ javax.media.j3d.BoundingLeaf      │
            └──────────────────────────────────┘
```
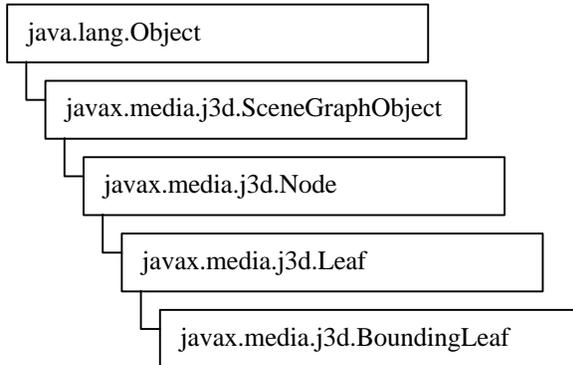
Figure 3-15 Java 3D API Class Hierarchy for BoundingLeaf

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

BoundingLeaf Constructor Summary
The BoundingLeaf node defines a bounding region object that can be referenced by other nodes to define a region of influence, an activation region, or a scheduling region. **BoundingLeaf()** Constructs a BoundingLeaf node with a unit sphere object. **BoundingLeaf(Bounds region)** Constructs a BoundingLeaf node with the specified bounding region.

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

BoundingLeaf Method Summary
Bounds getRegion() Retrieves this BoundingLeaf's bounding region **void setRegion(Bounds region)** Sets this BoundingLeaf node's bounding region.

3.8 User Data

Any SceneGraphObject can reference any object as user data[12]. First, you should realize that nearly every Java 3D API core class is a descendant of SceneGraphObject. The list of descendants of SceneGraphObject includes Appearance, Background, Behavior, BranchGroup, Geometry, Lights, Shape3D, and TransformGroup.

---

[12] This is not limited to Java 3D API classes, but any class derived from java.lang.Object.

The applications for this, the UserData field, are limited only by your imagination. For example, an application may have a number of pickable objects. Each of these objects could have some text information stored in the user data object. When a user picks an object, the user data text can be displayed.

Another application could store some calculated value for a scene graph object such as its position in virtual world coordinates. Yet another application could store some behavior specific information that could control a behavior applied to a variety of objects.

SceneGraphObject Methods (Partial List - User Data Methods)

SceneGraphObject is a common superclass for all scene graph component objects. This includes Node, Geometry, Appearance, etc.

```
java.lang.Object getUserData()
```
Retrieves the userData field from this scene graph object.

```
void setUserData(java.lang.Object userData)
```
Sets the userData field associated with this scene graph object.

3.9   Chapter Summary

This chapter presents the Java 3D features for easier content creation. Loader utilities and GeometryInfo classes are the primary easy content creation techniques. These topics are covered in sections 3.2 and 3.3, respectively. Text is added to the Java 3D world using Text2D and Text3D classes in sections 3.4 and 3.5. Background is covered in detail in section 3.6 and BoundingLeaf is covered in section 3.7. Section 3.8 presents the UserData field of the SceneGraphObject class. You are reading section 3.9.

3.10 Self Test

1. Using the wire frame view in the `GeomInfoApp.java` program, you can see the effect of the triangulation. Using the example program as a starting point, change the specification of the polygons to use three polygons (one for each side, and one for the roof, hood, trunk lid and other surfaces). How does the Triangulator do with this surface?

2. The code to make the Text2D object visible from both sides is included in `Text2DApp.java.` You can uncomment the code, recompile and run it. Other experiments with this program include using the texture from the Text2D object on other visual objects. For example, try adding a geometric primitive and apply the texture to that object. Of course, you may want to wait until you read Chapter 7 for this exercise.

3. Using `Text3DApp.java` as a starting point, experiment with the various alignment and path settings. Other experiments include changing the appearance of the Text3D object.

4. Playing with the `BackgroundApp.java` example program, if you move far enough away from the origin of the virtual world the background disappears. Why does this happen? If you add another Background object to the BackgroundApp, what will the effect be?