The Java 3D™ API

Technical White Paper

Please
Recycle

Contents

Introduction 1

Internet technologies, and, in particular the Java™ language have brought about a fundamental change in the way that applications are designed and deployed.

Java's "write once, run anywhere" model has lessened the complexity and cost normally associated with producing software on multiple distinct hardware platforms. With Java, the browser paradigm has emerged as a compelling way to produce applications for the Internet and the corporate intranet.

As new classes of applications emerge for the Web environment, there is increasing pressure for full multimedia capabilities to become seamlessly integrated into the browser paradigm. Application developers are demanding a small number of high-level interfaces to work with and users prefer a seamless environment which is easy to operate and maintain.

In addition to software innovations, ASIC technology and high levels of integration have made accelerated 3D graphics- and multimedia-capable hardware affordable. High end graphics features, previously only available on specialized graphics workstations, are now widely available on low-cost platforms, making them available for new kinds of applications.

The Need for Integrated Web-based Multimedia

In the past, integration of multimedia technology like audio, video, and 3D graphics into Web-based applications has required the use of external applications (or "plug-ins") to handle data which the browser wasn't capable of interpreting or displaying. For example, VRML (Virtual Reality Modeling

Language) scene navigation generally requires that a separate VRML plug-in application be installed on the client system.

Though widely used, plug-ins are non-flexible in that they represent separately compiled, platform-specific code which must be individually installed on each client platform. From the client perspective, plug-ins are poorly integrated into the browser environment since they are a separate application and often operate in a separate window from the browser. Additionally, useful plug-ins may only be available for the more popular client platforms.

To address these issues, and in response to the demand for fully-integrated Web-based multimedia, JavaSoft and a cooperative industry contingent have extended Java with the Java Media family of APIs.

Java Media

Described in Table 1-1, Java Media provides seamless integration of a wide range of multimedia technologies into the Web environment.

Node Name	Description
Java 2D	Provides an abstract imaging model that extends the JDK 1.0.2 AWT package, including line art, images, color, transforms, and compositing
Java Media Framework	Specifies a unified architecture, messaging protocol, and programming interface for media players, media capture, and conferencing; Comprises three separate APIs (Java Media Player, Java Media Capture, and Java Media Conference
Java Collaboration	Allows for interactive, two-way, multi-party communications over a variety of networks
Java Telephony	Integrates telephones with computers and provides basic functionality for first-party and third-party call control
Java Speech	Provides Java based speed recognition and speech synthesis (text-to-speech)
Java Animation	Provides for motion and transformations of 2D objects while utilizing the Java Media Framework for synchronization, composition, and timing
Java 3D	Provides an abstract, interactive imaging model for behavior and control of 3D objects

Table 1-1 Java Media APIs

Java 3D

As a part of Java Media, the Java 3D API is an application programming interface used for writing stand-alone three-dimensional graphics applications or Web-based 3D applets. It gives developers high level constructs for creating and manipulating 3D geometry and tools for constructing the structures used in rendering that geometry. With Java 3D constructs, application developers can describe very large virtual worlds, which, in turn, are efficiently rendered by Java 3D.

The Java 3D specification is the result of a joint collaboration between Silicon Graphics, Inc., Intel Corporation, Apple Computer, Inc., and Sun Microsystems, Inc. All had advanced, retained mode APIs under active internal development, and were looking at developing a single, compatible, cross-platform API in Java.

The Java 3D API draws its ideas from the considerable expertise of the participating companies, from existing graphics APIs, and from new technologies. Java 3D's low-level graphics constructs synthesize the best ideas found in low-level APIs such as Direct3D™, OpenGL™, QuickDraw3D™, and XGL™. Similarly, Java 3D's higher-level constructs leverage the best ideas found in several modern scene graph-based systems. Java 3D also introduces some concepts not commonly considered part of the graphics environment, such as 3D spacial sound to provide a more immersive experience for the user.

Markets and Applications

Java 3D is designed as a high-level platform-independent 3D graphics programming API and is amenable to very high performance implementations across a range of platforms. Java 3D will scale gracefully as the rendering speed and capabilities of the underlying 3D hardware increase by orders of magnitude over time.

Java 3D is targeted at a wide spectrum of 3D application environments from small to very large virtual universes.

- *Browsers*

  Providing Java 3D support in Web browsers will obviate the need for separately compiled plug-ins and will enable full integration of 3D navigation completely within the browser environment.

- *Virtual Reality systems*

  Java 3D's scene-graph based model makes it ideal for Virtual Reality systems and other applications that wish to represent and navigate complex 3D worlds.

- *3D games*

  Though many programmers will continue to implement games in low-level assembly code, Java 3D includes many features that enable high performance 3D games to be written in a platform-independent fashion.

- *CAD systems*

  Both Web-based CAD applications and stand-alone MCAD systems represent a key opportunity for Java 3D.

- *3D logos, Web pages (graphic designers)*

  Tools that enable Java 3D output are anticipated for Web designers who wish to include interactive 3D representations, or dynamic 3D logos into Web page design.

- *VRML Implementations*

  Though Java 3D does not support VMRL directly, it is expected that applications will be developed which provide support for loading the various VRML formats into Java 3D representations.

- *Authoring Systems*

  Though Java 3D is not an authoring environment, and does not provide built-in authoring tools, it is quite likely that independent software suppliers will build authoring systems that generate Java 3D applications as output.

Java 3D Overview 2

Historically, 3D graphics programmers have needed to wring every last ounce of performance from their graphics hardware in order to obtain a high degree of visual realism. Developers have often had to leverage extensive knowledge of underlying hardware details in order to obtain maximum performance from a given graphics accelerator.

Even low-level cross-platform APIs, like OpenGL, have required a high degree of programming expertise in order to exact optimized performance from different hardware platforms. These realities have resulted in a difficult and expensive development process which has mandated platform-specific development efforts — leaving few resources to focus on application functionality.

A Platform Independent 3D API

Java 3D represents an evolution to a standard, high-level 3D API that yields a high degree of interactivity while preserving true platform independence.

Java 3D Design Goals

Java 3D was designed to satisfy the following goals:

- *High Performance*

  Many design decisions were made so that Java 3D implementations could deliver the highest level of performance to application users. In particular, when trade-offs were made, the alternative that benefited runtime execution was chosen.

- *Rich set of 3D features*

  Java 3D was designed to provide a rich set of features for creating interesting 3D worlds, tempered by the need to avoid non-essential or obscure features. Features that could be written in Java and layered on top of Java 3D were not included.

- *High-level, Object-oriented paradigm*

  Java 3D was designed to offer a high-level, object-oriented, programming paradigm that enables developers to rapidly deploy sophisticated applications and applets.

- *Wide Variety of File Formats*

  Support for run-time loaders was included to allow Java 3D to accommodate a wide variety of file formats such as vendor-specific CAD formats, interchange formats, VRML 1.0, and VRML 2.0.

High Performance

Java 3D's scene graph programming model (described later in this document) allows the Java 3D API to perform mundane tasks (traversal of the scene graph, management of state attribute changes, etc.), thereby simplifying the job for the application. Java 3D does this without sacrificing performance.

At first glance, it might appear that this high-level approach would create more work for the API. However, it actually has the opposite effect. Java 3D's higher level of abstraction not only changes the amount, but more importantly, the kind of work that the API must perform. Java 3D is freed from the constraints found in interfaces with a lower level of abstraction and allows introduction of optimizations not possible with these lower-level APIs.

Additionally, leaving the details of rendering to Java 3D allows it to tune the rendering to the underlying hardware. For example, relaxing the strict rendering order imposed by other APIs allows parallel traversal of the scene graph as well as parallel rendering. Knowing which parts of the scene graph cannot be modified at runtime allows Java 3D to flatten the tree, pre-transform geometry, or represent the geometry in a native hardware format without the need to keep the original data.

Layered Implementation

Besides optimizations at the API level, one of the more important factors that determines the performance of Java 3D is the time it takes to render the visible geometry.

To optimize rendering, Java 3D implementations are layered to take advantage of the native, low-level API that is available on a given system. In particular, Java 3D implementations that utilize OpenGL, Direct3D, and QuickDraw3D will be available. This means that Java 3D rendering will be accelerated across the same wide range of systems that are supported by these lower-level APIs.

Target Hardware Platforms

Java 3D is aimed at a wide range of 3D-capable hardware and software platforms, from low cost PC game cards and software renderers, through mid-range workstations, all the way up to very high-performance, specialized, 3D image generators.

It is expected that Java 3D implementations will provide useful rendering rates on most modern PCs, especially those with 3D graphics accelerator cards. On mid-range workstations, Java 3D is expected to provide applications with nearly full-speed hardware performance.

Finally, Java 3D was designed to scale as the underlying hardware platforms increase in speed over time. Tomorrow's 3D PC game accelerators will support more complex virtual worlds than the high-priced workstations of a few years ago. Java 3D is prepared to meet this increase in hardware performance.

2

Programming Paradigm

Java 3D provides several basic classes that are used to construct and manipulate a *scene graph*, and to control viewing and rendering. Details for the object hierarchies and classes which make up the scene graph and its component parts are provided in Chapter 3.

The Scene Graph Programming Model

Java 3D's scene graph-based programming model provides a simple and flexible mechanism for representing and rendering potentially complex 3D environments. The scene graph contains a complete description of the entire scene, or virtual universe. This includes the geometric data, the attribute information, and the viewing information needed to render the scene from a particular point of view.

The Java 3D API improves on previous graphics APIs by eliminating many of the bookkeeping and programming chores that those APIs impose. Java 3D allows the programmer to think about geometric objects rather than about triangles—about the scene and its composition rather than about how to write the rendering code for efficiently displaying the scene.

A Java 3D Application Scene Graph Example

Figure 2-1 illustrates a scene graph for a fairly simple application. For source code appropriate to this example, please see `HelloUniverse.java` in Appendix A.

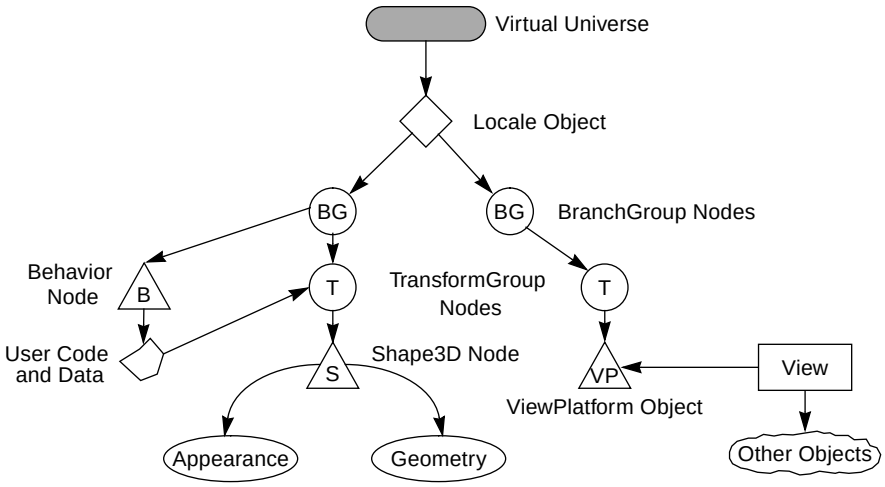

Figure 2-1 Java 3D application scene graph

The scene graph consists of superstructure components—a VirtualUniverse object and a Locale object—and scene graph branches, or subgraphs. Each subgraph is rooted by a BranchGroup node that is attached to the superstructure. For more information, see Chapter 3, “Scene Graph Overview.”

A VirtualUniverse object defines a named universe. Java 3D permits the creation of more than one universe, though the vast majority of applications will use just one. The VirtualUniverse object provides a grounding for scene graphs. All Java 3D scene graphs must connect to a Virtual Universe object to be displayed.

Below the VirtualUniverse object is a Locale object. The Locale object defines the origin, in high-resolution coordinates, of its attached subgraphs. A Virtual Universe may contain as many Locales as needed. In this example, a single Locale object is defined with its origin at (0.0, 0.0, 0.0).

The scene graph itself starts with the BranchGroup nodes. A BranchGroup serves as the root of a subgraph, or branch graph, of the scene graph. Only BranchGroup objects can attach to Locales.

In the example in Figure 2-1, there are two subgraphs and, thus, two BranchGroup nodes. Attached to the left BranchGroup are two subtrees. One subtree consists of a user-extended Behavior leaf node which contains Java code to manipulate the transform matrix associated with the object's geometry.

The other subtree in this BranchGroup consists of a TransformGroup node that specifies the position (relative to the Locale), the orientation, and the scale of the geometric object in the virtual universe. A single child, a Shape3D node, refers to two component objects: a Geometry object and an Appearance object. The Geometry object describes the geometric shape of a 3D object (a cube in our simple example). The Appearance object describes the appearance of the geometry (color, texture, material reflection characteristics, etc.).

The right BranchGroup has a single subtree that consists of a TransformGroup node and a ViewPlatform leaf node. The TransformGroup specifies the position (relative to the Locale), the orientation, and the scale of the ViewPlatform. This transformed ViewPlatform object defines the end user's view within the virtual universe.

Finally, the ViewPlatform is referenced by a View object that specifies all of the parameters needed to render the scene from the point of view of the ViewPlatform. Also referenced by the View object are other objects that contain information such as the drawing canvas that Java 3D renders into, the screen that contains the canvas, and information about the physical environment.

Java 3D View Model

Java 3D introduces a new view model that takes Java's vision of "write once, run anywhere" and generalizes it to include display devices and six-degree-of-freedom input peripherals such as headtrackers.

An application or applet written using the Java 3D view model can render images to a broad range of display devices including flat screen displays, portals/caves, and head-mounted displays all without modification to the code. The same application or applet, (once again without modification) can render stereoscopic views and can take advantage of the input from a head-tracker to control the rendered view.

Java 3D's view model achieves this versatility by cleanly separating the virtual and the physical world. This model distinguishes between how an application positions, orients and scales a ViewPlatform object (a viewpoint) within the virtual world and how Java 3D's renderer constructs the final view from that viewpoint's position and orientation.

The application controls the ViewPlatform's position and orientation while the renderer computes what view to render using this position and orientation, a description of the end-user's physical environment, and the user's position and orientation within the physical environment.

The Camera Based Model

Most low-level APIs utilize a Camera-based view model which emulates a camera in the virtual world. Developers must continuously reposition a simulated camera in such an API to emulate a human in the virtual world.

Camera-based view models give developers control over all rendering parameters. This control make sense when dealing with custom applications but is less useful when dealing with systems such as viewers or browsers that load and display whole worlds as a single unit, or that let their end-users view, navigate, display, and even interact with that world.

Making the application control all rendering parameters is problematic in systems where the physical environment dictates some of the view parameters: For example, a head-mounted display (HMD), where the optics of the head-mounted display directly determine the field-of-view that the application should use. Different HMDs have different optics, and hard-wiring such parameters into an application or allowing end-users to vary these parameters are poor solutions.

Additionally, view parameters change as a function of the user's head position with an HMD. The specification of a world and a predefined flight path through that world may not exactly specify an end-user's view should the user look to the side and change the view.

Java 3D's view model incorporates the appropriate abstractions to compensate *automatically* for such variability in end-user hardware environments.

Camera Based View Model Support

In order to assist with porting existing applications, Java 3D provides a compatibility mode which enables a camera-based view model. With a simple function-call, applications can enable compatibility mode for room-mounted, non head-tracked display environments.

It should be noted that use of these view compatibility functions will disable some of Java 3D's view model features and will limit portability.

2

Input

The `java.awt` package of classes already contains an abstraction for common desktop interaction peripherals like keyboards and mice. Java 3D uses that implementation for simple devices rather than creating a separate, incompatible I/O model. However, since Java 3D provides support for a variety of continuous input devices such as six-degree-of-freedom (6DOF) trackers and joysticks, a new input model was required.

For these time-critical devices, Java 3D introduces a new real-time class of I/O device which provides for very low latency. In real-time graphics systems, low latency can be more important than missing an event since an event more than a few 30th's of a second old may no longer be of any interest (particularly if it tracks a user's current head position).

Java 3D abstracts the idea of a tracking device or joystick in the form of a sensor. A sensor consists of a time-stamped sequence of input values and the state of any buttons or switches on the sensor at the time that Java 3D sampled the value.

Behavior, Animation, and Picking

Behavior nodes provide the means for animating objects, processing keyboard and mouse inputs, reacting to movement, and enabling and processing pick events. Behavior nodes contain Java code and state variables. The Java code in a Behavior node can interact with Java objects, change node values within a Java 3D scene graph, change internal state, and perform other computations.

Simple behaviors can add surprisingly interesting effects to a scene graph. For example, a rigid object can be animated by using a behavior node to repetitively modify the TransformGroup node that points to the object. Alternatively, a behavior node can track the current position of a mouse and modify portions of the scene graph to implement picking.

Behavior Object

A Behavior leaf node object contains a scheduling region and two methods: an initialization method called once when the behavior becomes "live" and a processStimulus method called whenever appropriate by the Java 3D behavior scheduler. The behavior object will also contain whatever state information the Behavior code requires.

The **scheduling region** defines a spatial volume that serves to enable the scheduling of Behavior nodes. A Behavior node is *active* (can receive stimuli) whenever a ViewPlatform's activation volume intersects a Behavior object's scheduling region. Only active behaviors can receive stimuli.

The **initialization method** allows a behavior object to initialize its internal state and specify its initial wakeup condition(s). Java 3D invokes a behavior's initialization code when the behavior's containing BranchGroup node is added to the virtual universe.

The **processStimulus method** receives and processes a behavior's ongoing messages. Java 3D invokes a behavior node's processStimulus-method when a ViewPlatform's activation volume intersects a behavior object's scheduling region and one of the behavior's wakeup criterion is satisfied. The processStimulus method performs its computations and actions (possibly including the registration of state change information that could cause Java 3D to wake other behavior objects), establishes its wakeup conditions, and exits.

Scheduling

As a virtual universe grows large, Java 3D must carefully husband its resources to ensure adequate performance. In a 10,000-object virtual universe with 400 or so behavior nodes, a naive implementation of Java 3D could easily end up consuming the majority of its compute cycles by executing the behaviors associated with the 400 behavior objects before it draws a frame. In such a situation, the frame rate could easily drop to unacceptable levels.

Java 3D mitigates the problem of a large number of behavior nodes in a high-population virtual universe through execution culling—choosing only to invoke those behaviors that have high relevance.

In a universe consisting of a large number of Behavior nodes controlling a similar number of objects, only a few of the controlled objects are typically visible at any one time. The sizeable fraction of the behavior nodes which are associated with non-visible objects, need never execute. The remaining, substantially smaller number of behavior objects associated with visible objects, must be executed.

Java 3D requires each behavior to have a scheduling region and to post a wakeup condition. Together a behavior's scheduling region and wakeup criterion provide Java 3D's behavior scheduler with sufficient domain knowledge to selectively prune behavior invocations and only invoke those behaviors that absolutely need to execute.

Execution Culling

Java 3D finds all scheduling regions associated with behavior nodes and constructs a scheduling-volume tree. It also creates an AND-OR tree containing all the behavior node wakeup criterion. These two data structures provide the domain knowledge Java 3D needs to prune unneeded behavior execution.

Java 3D must track a behavior's activation conditions only if a ViewPlatform object's activation volume intersects with that behavior object's scheduling region. If the ViewPlatform object's activation volume does not intersect with a behavior's scheduling region, Java 3D can safely ignore that behavior's wakeup criterion.

In essence, the Java 3D scheduler performs the following checks:

1. Find all Behavior objects with scheduling regions that intersect the ViewPlatform object's activation volume.
2. For each Behavior object that falls within the ViewPlatform's activation volume, check the Behavior's WakupCondition
3. Schedule that behavior object for execution if the WakupCondition is true

Java 3D's behavior scheduler executes those behavior objects that have been scheduled by calling the behavior's processStimulus method.

Predefined Behaviors — Interpolators

Java 3D provides a number of predefined behaviors. These predefined behaviors are known as Interpolators because they smoothly interpolate between a minimum and a maximum value.

The Java 3D API provides interpolators for a number of functions: for manipulating transformations within a TransformGroup, for modifying the values of a Switch node, and for modifying Material attributes such as color and transparency.

LOD Behaviors

The LOD (Level of Detail) behavior node is an abstract behavior class that operates on a list of Switch group nodes to select one of the children of the Switch node. Specializations of the LOD abstract behavior node implement various level-of-detail policies. For example, the Distance LOD behavior node

implements a distance-based LOD policy by selecting one of the Switch nodes' children based on distance from the viewer.

BillBoard Behaviors

The Billboard behavior node operates on a TransformGroup node to specify a transform that always aligns itself perpendicular to a specified world-coordinate axis or to a viewer's view vector. This transform is independent of transforms above the specified node in the scene graph.

Billboard nodes provide the most benefit for complex, roughly-symmetric objects. In a typical application, a quadrilateral that contains a texture of a tree might be held perpendicular to the user.

Picking

Behavior nodes also provide the means for building developer-specific picking semantics. An application developer can define custom picking semantics using Java 3D's behavior mechanism. For example, the developer might wish to define pick semantics that use a mouse to shoot a ray into the virtual universe from the current viewpoint, find the first object along that ray, and highlight that object when the user releases the mouse button.

Rendering Model, Rendering Modes, and Execution Path

Java 3D assumes a double-buffered, true-color, and Z-buffered rendering model. This was done primarily to avoid creating multiple APIs and incompatible programs to support indexed and true color, or Z-buffered and non Z-buffered environments.

Note that this set of minimal requirements is for the Java 3D rendering model, and does not necessarily describe hardware necessary to run a Java 3D application. The rendering model simply describes Java 3D's frame of reference and defines those properties that must be present in a low-level API in order to use it as a base for a Java 3D implementation. How a low-level API implements that functionality is arbitrary to Java 3D.

Rendering Modes

Java 3D includes three different rendering modes: immediate mode, retained mode, and compiled-retained mode. Each successive rendering mode allows Java 3D more freedom in optimizing an application's execution. Most Java 3D applications will want to take advantage of the convenience and performance benefits that retained and compiled-retained modes provide.

- *Immediate Mode*

  Immediate mode leaves very little room for optimization. Even so, Java 3D has raised the level of abstraction. An application must provide a Java 3D draw method with a complete set of points, lines, or triangles. Of course, the application can build these lists of points, lines, or triangles in any manner it chooses.

- *Retained Mode*

  Retained mode requires an application to construct a scene graph and specify which elements of that scene graph may change during rendering. The scene graph describes the objects in the virtual universe, the arrangement of those objects, and how the application animates those objects.

- *Compiled-Retained Mode*

  Compiled-retained mode, like retained mode, requires the application to construct a scene graph and specify which elements of the scene graph may change during rendering. Additionally, the application can compile some, or all, of the subgraphs that make up a complete scene graph.

  Java 3D compiles these graphs into an internal format. The compiled representation of the scene graph bears little resemblance to the original tree structure provided by the application but is functionally equivalent.

Java 3D Runtime Execution Path

Once a Java 3D program has been created, the following steps are taken by the program to create the scene graph elements and link them together. Java 3D will then render the scene graph and display it in a window on the screen:

1. Create the Canvas3D and View objects.
2. Establish the virtual universe and high-res Locale
3. Construct the scene graph elements.

4. Link them together to form the two subgraphs.
5. Optionally compile the subgraphs.
6. Insert the two subgraphs into the Locale.

The Java 3D renderer then starts running in an infinite loop. The renderer conceptually performs the following operations:

```
while(true) {
   Process input
   If (request to exit) break
   Perform Behaviors
   Traverse the scene graph and render visible objects
}
Cleanup and exit
```

An example Java 3D program is provided in Appendix A.

Sound Model

Java 3D uses Java's standard audio API in order to provide general sound and midi support. Because Java 3D is the only Java API that includes support for headtrackers, parameterization of a user's physical head, generalized transformations, and other such concepts, Java 3D adds additional support for fully spatialized audio.

Vector Math Library

Java 3D defines a number of additional objects that are used in the construction and manipulation of other Java 3D objects. These objects provide low-level storage and manipulation control for users. They provide methods for representing vertex components (e.g., color and position), volumes, vectors, and matrices.

The tuple and matrix math classes are not part of Java 3D per se, but they are needed by Java 3D and are defined here for convenience. Java 3D uses these classes internally and also makes them available for use by applications. These classes will be delivered in a separate `java.vecmath` package for use with other Java APIs. The vector and matrix math classes are described in detail in an Appendix of the "Java 3D API Specification".

Vector Objects

The Vector objects included in `java.vecmath` (listed in Table 2-1), store vectors of length two, three, and four. Java 3D vectors are used to store various kinds of information such as colors, normals, texture coordinates, vertices, etc.

The classes are further subdivided by storage class, whether the vector consists of single or double precision floating point numbers or bytes. Only the floating-point vector classes support math operations.

Class	Description
Tuple2f	Used to represent two-component coordinates in single-precision floating point format. This class is further divided into: Point2f: Represents x,y point coordinates TexCoord2f: Represents x,y texture coordinates Vector2f: Represents x,y vector coordinates
Tuple3b	Used to represent three-component color information; stored as three bytes. This class is further divided into: Color3b: Represents RBG color values
Tuple3d	Used to represent point and vector coordinates, in double-precision floating point format. This class is further divided into: Point3d: Represents x,y,z point coordinates Vector3d: Represents x,y,z vector coordinates
Tuple3f	Used to represent three-component colors, point coordinates, texture coordinates, and vectors, in single-precision floating point format. This class is further divided into: Color3f: Represents RGB color values Point3f: Represents x,y,z point coordinates TexCoord3f: Represents x,y,z texture coordinates Vector3f: Represents x,y,z vector coordinates
Tuple4b	Used to represent four-component color information stored as four bytes. This class is further divided into: Color4b: Represents RGBα color values
Tuple4d	Used to represent four-component color information, quaternions, and vectors, stored in double-precision floating point format. This class is further divided into: Point4d: Represents x,y,z,w point coordinates Quat4d: Represents x,y,z,w quaternion coordinates Vector4d: Represents x,y,z,w vector coordinates
Tuple4f	Used to represent four-component color information, quaternions, and vectors, in single-precision floating point format. This class is further divided into: Color4f: Represents RBGα color values Point4f: Represents x,y,z,w point coordinates Quat4f: Represents x,y,z,w quaternion coordinates Vector4f: Represents x,y,z,w vector coordinates

Class	Description
AxisAngle4d	Used to represent four-component axis-angle rotations consisting of double-precision floating point x, y, and z coordinates and a rotation angle in radians
AxisAngle4f	Used to represent four-component axis-angle rotations consisting of single-precision floating point x, y, and z coordinates and a rotation angle in radians
GVector	Used to represent a general, dynamically-resizable one-dimension vector class

Table 2-1 Vector objects

Matrix Objects

The matrix objects, listed inTable 2-2, define a complete 3×3 or 4×4 floating-point transform matrix. All vector subclasses operate using this matrix type.

Class	Description
Matrix3d	Used to represent a double-precision floating point 3 x 3 matrix
Matrix3f	Used to represent a single-precision floating point 3 x 3 matrix
Matrix4d	Used to represent a double-precision floating point 4 x 4 matrix
Matrix4f	Used to represent a single-precision floating point 4 x 4 matrix
GMatrix	A double-precision, general, dynamically-resizable N x M matrix class

Table 2-2 Matrix objects

Geometry Compression

Java 3D allows programmers to specify geometry using a binary geometry compression format. This compression format is used with APIs other than just Java 3D, and can be used both as a run-time in-memory format for describing geometry, as well as a storage and network format.

Using geometry compression allows geometry to be represented in an order of magnitude less space than most traditional 3D representations, with very little loss in object quality. Parameters exist to allow the user to trade off space versus quality by adjusting the degree of loss in the compression algorithm.

Like the vector mathematics library, geometry compression is provided with the Java 3D specification but is likely to become its own specification over time. The Java 3D geometry compression format is fully described in an appendix to the Java 3D API specification.

2

Scene Graphs and Java 3D Objects

3

This chapter provides an overview of the Java 3D scene graph along with its component objects. For a more detailed discussion of these objects, along with their methods, the “Java 3D API Specification” should be consulted.

Scene Graph Overview

As discussed earlier, a Java 3D scene graph consists of objects, called nodes, arranged in a tree structure. The user creates one or more scene subgraphs and attaches them to a virtual universe.

The individual connections between Java 3D nodes are always a directed relationship: parent to child. Java 3D differs from other scene graph-based systems in that scene graphs may not contain cycles. Thus, a Java 3D scene graph is a directed acyclic graph (DAG) as shown in Figure 3-1.

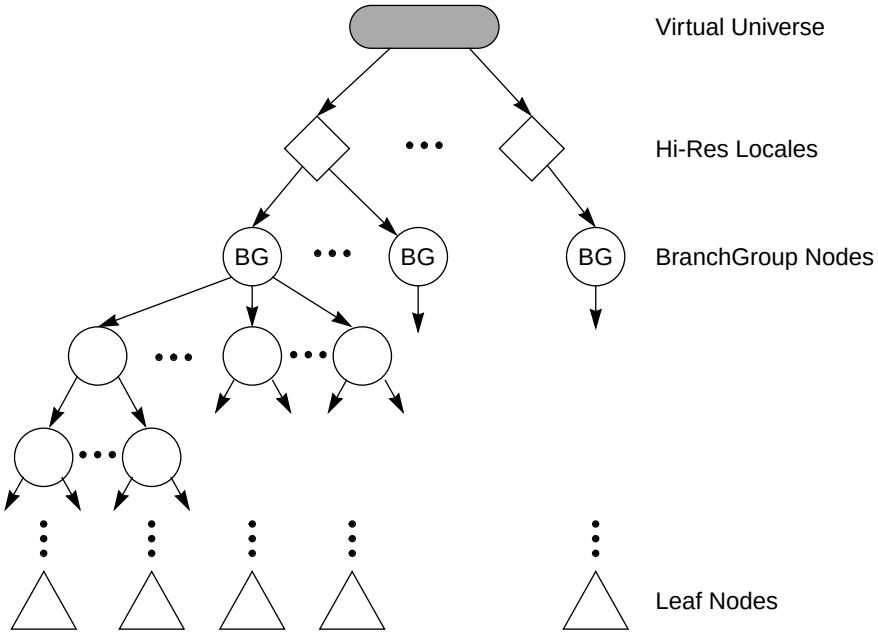

Figure 3-1 A Java 3D scene graph is a DAG (Directed Acyclic Graph)

Scene Graph Structure

A scene graph organizes and controls the rendering of its constituent objects. All Java 3D renderers draw a scene graph in a consistent manner that allows for concurrence. A Java 3D renderer can draw one object independently of other objects. Java 3D can allow such independence because its scene graphs have a particular form and cannot share state among branches of a tree.

Spatial Separation

The hierarchy of the scene-graph imposes a natural spatial grouping on the geometric objects found at the leaves of the graph. Internal nodes act to group their children together. Group nodes also define a spatial bound that contains all the geometry defined by its descendants. Spatial grouping allows for efficient implementation of operations such as proximity detection, collision detection, view frustum culling, and occlusion culling.

State Inheritance

A leaf node's state is defined by the nodes in a direct path between the scene graph's root and the leaf. Because a leaf's graphics context only relies on a linear path between the root and that node, the Java 3D renderer can decide to traverse the scene graph in whatever order it wishes. Except for spatially bounded attributes (i.e. lights and fog), the scene graph can be traversed from left to right and top to bottom, level order from right to left, or even in parallel.

Many older scene graph-based APIs (including PHIGS and SGI's Inventor™) feature less flexible rendering traversal. In these systems, if a node above or to the left of a given node changes the graphic state, the change affects the graphic state of all nodes below it or to its right.

The most common node object, along the path from the root to the leaf, that changes the graphics state is the TransformGroup object. The TransformGroup object can change the position, orientation, and/or scale of the objects below it.

Most graphics state is set by a Shape3D node through its constituent Appearance object which allows for parallel rendering. The Shape3D node also has a constituent geometry object that specifies its geometry, permitting different Shape3D objects to share common geometry without sharing material attributes (or vice-versa).

Rendering

The Java 3D renderer incorporates all graphics state changes made in a direct path from a scene-graph root to a leaf object in the drawing of that leaf object. Java 3D provides this semantic for both retained and compiled-retained modes.

Scene Graph Objects

A Java 3D scene graph consists of a collection of Java 3D node objects connected in a tree structure. These node objects reference other scene graph objects called *component objects*. All scene graph node and component objects are subclasses of a common SceneGraphObject class (see Figure 3-2). The

SceneGraphObject class is an abstract class that defines methods that are common among nodes and component objects.

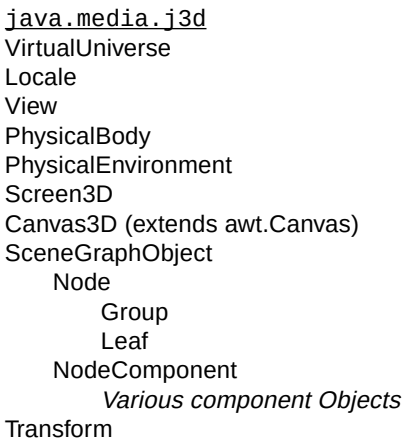

Figure 3-2 Java 3D object hierarchy

Scene graph objects are constructed by creating a new instance of the desired class and are accessed and manipulated using the object's set and get methods. Once a scene graph object is created and connected to other scene graph objects to form a subgraph, the entire subgraph can be attached to a virtual universe—via a high-resolution Locale object—making the object live. Prior to attaching a subgraph to a virtual universe, the entire subgraph can be compiled into an optimized, internal format.

An important characteristic of scene graph objects is that they can only be accessed or modified during the creation of a scene graph, except where explicitly allowed. Access to most set and get methods of objects that are part of a live or compiled scene graph is restricted. Such restrictions provide the scene-graph compiler with usage information it can use in optimally compiling or rendering a scene graph.

Each object has a set of *capability bits* that enable certain functionality when the object is live or compiled. By default, all capabilities are disabled. Only those set and get methods corresponding to capability flags that are explicitly enabled—prior to the object being compiled or made live—are legal. The methods for setting and getting capability flags are described in the "Java 3D API Specification."

Scene Graph Superstructure Objects

Java 3D defines two Scene graph superstructure objects, the VirtualUniverse and Locale, which are used to contain collections of subgraphs that comprise the scene graph.

VirtualUniverse Object

A VirtualUniverse object consists of a name and a list of Locale objects that contain a collection of scene graph nodes that exist in the named universe. Typically, an application will need only one VirtualUniverse, even for very large virtual databases. Operations on a VirtualUniverse include retrieving its name and enumerating the Locale objects contained within the universe.

Locale Object

The Locale object acts as a container for a collection of subgraphs of the scene graph that are rooted by a BranchGroup node. A Locale also defines a location within the virtual universe using high resolution coordinates (HiResCoord) to specify its position. This HiResCoord serves as the origin for all scene graph objects contained within the Locale.

A Locale has no parent in the scene graph, but is implicitly attached to a named virtual universe when it is constructed. A Locale may reference an arbitrary number of BranchGroup nodes, but has no explicit children.

The coordinates of all scene graph objects are relative to the HiResCoord of the Locale in which they are contained. Operations on a Locale include setting or getting the HiResCoord of the Locale, adding a subgraph, and removing a subgraph.

Node Objects

Java 3D refines the Node object class into two subclasses: Group and Leaf node objects. Group node objects group together one or more child nodes. A group node can point to zero or more children but can have only one parent (some group nodes have no parents).

Leaf node objects contain the actual definitions of shapes in the form of geometry objects, lights, fog, sounds, etc. The semantics of both group and leaf nodes are described later in this chapter.

Scene Graph Component Objects

Scene graph component objects include the actual geometry and appearance attributes used to render the geometry. These component objects are described later in this chapter.

Scene Graph Viewing Objects

Java 3D defines six scene graph viewing objects that are not part of the scene graph per se, but serve to define the viewing parameters and to connect with the physical world. Scene graph viewing objects are illustrated in Figure 3-3.

- *ViewPlatform*

  The ViewPlatform object is a Leaf node that locates a view within a scene graph. The ViewPlatform's parents specify its location, orientation, and scale within the Virtual Universe.

- *View Object*

  The View object specifies information needed to render the scene graph. The View object is the central Java 3D object for coordinating all aspects of viewing. All viewing parameters in Java 3D are either directly contained within the View object or within objects pointed to by a View object. Java 3D supports multiple simultaneously active View objects, each of which can render to one or more canvases.

- *Canvas3D Object*

  The Canvas3D object encapsulates all of the parameters associated with the window being rendered into. When a Canvas3D object is attached to a View object, the Java 3D traverser renders the specified view onto the canvas. Multiple Canvas3D objects can point to the same View object.

- *Screen3D Object*

  The Screen3D object encapsulates all of the parameters associated with the physical screen containing the canvas, such as the width and height of the screen in pixels, the physical dimensions of the screen, and whether the screen is head-mounted or is fixed in position.

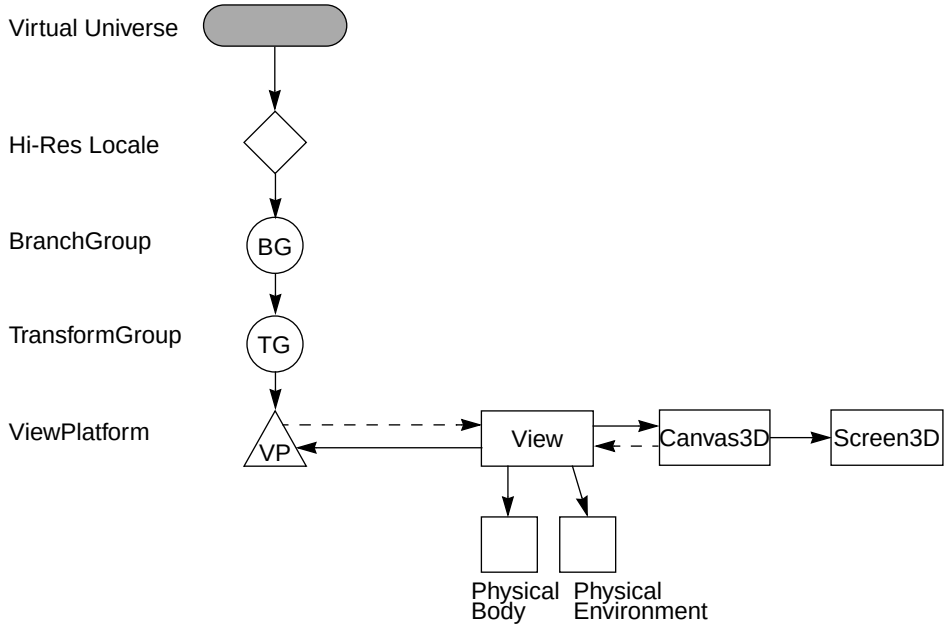

Figure 3-3 A portion of a scene graph containing a ViewPlatform object

- *Physical Body Object*

  The Physical Body Object contains calibration information describing the user's physical body.

- *Physical Environment Object*

  The Physical Environment Object contains calibration information describing the physical world, mainly information that describes the environment's six-degree-of-freedom tracking hardware, if present.

Group Node Objects

The Group node is an abstract class for the general purpose of grouping objects used in constructing a scene graph. The group node object hierarchy is listed in Figure 3-4.

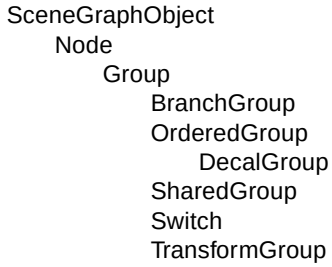

Figure 3-4 Group node hierarchy

Group Node

All group nodes can have a variable number of child node objects—including other group nodes as well as leaf nodes. Group nodes have exactly one parent.

The children of a group node have associated with them an index that allows operations to specify a particular child. However, unless one of the special ordered group nodes is used, the Java 3D renderer can choose to render a group node's children in whatever order it chooses (including rendering the children in parallel).

Operations on Group node objects include: adding, removing, and enumerating the children of the Group node. The subclasses of Group add additional semantics. Subclasses of the Group Node are listed in Table 3-1 below along with their descriptions.

Node Name	Description
BranchGroup Node	The root of a subgraph of a scene that may be compiled as a unit, attached to a virtual universe, or included as a child of a group node in another subgraph
OrderedGroup Node	Guarantees that Java 3D will render its children in their index order
DecalGroup Node	Specifies that its children generate coplanar objects
SharedGroup Node	Provides a mechanism for sharing the same subgraph in different parts of the tree via a Link node
Switch Node	Allows a Java 3D application to choose dynamically among a number of subgraphs
Transform Group Node	Specifies a single spatial transformation that can position, orient, and scale all of its children

Table 3-1 Group node objects

Leaf Node Objects

The leaf node is an abstract class for all scene graph nodes which have no children. Figure 3-5 depicts the Leaf node object hierarchy.

- SceneGraphObject
  - Node
    - Leaf
      - Background
      - Behavior
        - *Pre-defined behaviors*
      - BoundingLeaf
      - Clip
      - Fog
        - ExponentialFog
        - LinearFog
      - Light
        - AmbientLight
        - DirectionalLight
        - PointLight
          - SpotLight
      - Link
      - Morph
      - Shape3D
      - Sound
        - BackgroundSound
        - PointSound
          - ConeSound
      - Soundscape
      - ViewPlatform

Figure 3-5 Leaf node object hierarchy

Leaf Node

Leaf nodes specify lights, geometry and sounds and provide special linking and instancing capabilities for sharing scene graphs, as well as a view platform for positioning and orienting a view in the virtual world. Subclasses of the Leaf Node are listed in Table 3-2 along with their descriptions.

Node Name	Description
Background 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.
Behavior Node	Allows an application to manipulate a scene graph at runtime; behavior is an abstract class that defines properties common to all behavior objects in Java 3D
BoundingLeaf Node	Defines a bounding region object that can be referenced by other leaf nodes to define a region of influence, activation region, or scheduling region
Clip Node	Defines the far clipping plane used to clip objects in the virtual world; also specifies an application region in which this Clip node is active
Fog Node	Specifies the attributes that control fog, or depth cueing, in the scene
ExponentialFog Node	Extends the Fog leaf node by adding a fog density that is used as the exponent of the fog equation
LinearFog Node	Extends the Fog leaf node by adding a pair of distance values, in Z, at which fog should start obscuring the scene and should maximally obscure the scene
Light Node	Abstract class that defines the properties common to all light nodes. A Light node has associated with it a color, state (on/off), and a Bounds object.
AmbientLight	Defines an ambient light source
DirectLight Node	Defines an oriented light with an origin at infinity
PointLight Node	Defines a point light source located at some point in space and radiating light in all directions (also known as a *positional light*)
SpotLight Node	Defines a point light source located at some point in space and radiating in a specific direction
Link Node	Allows an application to reference a shared subgroup, rooted by a SharedGroup node, from within a branch of the scene graph
Morph Node	Permits an application to morph between multiple GeometryArrays (geometries)

Node Name	Description
Shape3D Node	Specifies all geometric objects and contains two components: a reference to the shape's geometry, and its appearance
Sound Node	Defines the properties common to all sound nodes; a scene graph can contain multiple sounds
BackgroundSound Node	Defines an unattenuated sound source that has no position or direction
PointSound Node	Defines a spatially-located sound whose waves radiate uniformly in all directions from some point in space
ConeSound Node	Defines a point sound source located at some location in space whose amplitude is constrained not only by maximum and zero amplitude spheres but by two concentric cone volumes directed down a vector radiating from the sound's location
Soundscape Node	Defines the attributes that characterize the listener's environment as it pertains to sound
ViewPlatform Node	Defines a viewing platform that is referenced by a View object

Table 3-2 Leaf node objects

Scene Graph Component Objects

The node objects described above do not fully specify their exact semantics by themselves. Some information is specified as an attribute and an associated floating-point or integer value.

In many cases, however, node objects utilize references to more complex entities called *component objects* in order to specify appearances or other properties. Node component objects encapsulate related state information in a single entity.

Within Java 3D two main groupings of objects are provided. One grouping provides *attribute* information, the other provides *geometry* information required to describe the geometry of a Shape3D Node.

Node Component Objects - Attribute

Attribute node component objects provide information relating to the appearance (materials, textures, images), or implement other properties (bounding boxes, etc.) to the nodes that reference them.

The attribute component object hierarchy is shown in Figure 3-6.

SceneGraphObject
- NodeComponent
  - Appearance
  - AuralAttributes
  - ColoringAttributes
  - LineAttributes
  - PointAttributes
  - PolygonAttributes
  - RenderingAttributes
  - TextureAttributes
  - TransparencyAttributes
  - Material
  - MediaContainer
  - TexCoordGeneration
  - Texture
    - Texture2D
    - Texture3D

PixelArray
- PixelArray2D
- PixelArray3D

DepthImage
- DepthImageFloat
- DepthImageInt
- DepthImageNative

Bounds
- BoundingBox
- BoundingPolytope
- BoundingSphere

Transform3D

Figure 3-6 Attribute component object hierarchy

The Attribute Component Objects are described in Table 3-3.

Node Name	Description
Appearance Component Object	Component object of a Shape3D node that defines all rendering state for that Shape3D node
Aural Attributes Component Object	Component object of a Soundscape node that defines environmental audio parameters that affect sound rendering
ColoringAttributes	Defines attributes that apply to color mapping
LineAttributes	Defines all rendering state that can be set as a component object of a Shape3D node (solid, dash, dot, dash-dot, width, pattern, antialiasing)
PointAttributes	Defines all rendering state that can be set as a component object of a Shape3D node (point size, antialiasing)
PolygonAttributes	Defines all rendering state that can be set as a component object of a Shape3D node (face culling, rasterization mode, offset)
RenderingAttributes	Defines all rendering state that can be set as a component object of a Shape3D node (alpha, depth buffering)
TextureAttributes	Defines attributes that apply to texture mapping
TransparencyAttributes	Defines all attributes affecting transparency of the object
Material Object	Component object of an Appearance object that defines the material properties used when lighting is enabled
MediaContainer	Component object of a Sound node that defines the sound data associated with a Sound node.
TexCoordGeneration	Component object of an Appearance object that defines the parameters used when texture coordinate generation is enabled
Texture Object	Component object of an Appearance object that defines the texture properties used when texture mapping is enabled
Texture2D Object	Extends the Texture class by adding a constructor and a mutator method for setting a 2D texture image
Texture3D Object	Extends the Texture class by adding a third texture coordinate and by adding a constructor and a mutator method for setting a 3D texture image
PixelArray Object	An abstract class that is used to define 2D and 3D PixelArray classes used for texture images and background images
PixelArray2D Object	Defines a 2D array of pixels, used for texture and background images
PixelArray3D Object	Defines a 3D array of pixels, used for texture and background images

Node Name	Description
DepthImage	An abstract class that defines a 2D array of depth (Z) values
DepthImageFloat	Extends the DepthImage Object to define a 2D array of depth (Z) values in floating point format
DepthImageInt	Extends the DepthImage object and defines a 2D array of depth (Z) values in integer format
DepthImageNative	Extends the DepthImage object and defines a 2D array of depth (Z) values stored in the most efficient format for a particular device
Bounds Object	Define three varieties of containing volumes (BoundingBox, Bounding Polytope, and BoundingSphere) which Java 3D uses to support various culling operations
BoundingBox Object	Axis-aligned bounding box volumes
BoundingPolytope Object	Defines a set of planes that prescribe a convex polygonal bounding region
BoundingSphere Object	Spherical bounding volume with two associated values, a center point and a radius of the sphere
Transform3D	Transformations are represented by matrix multiplication and include such operations as rotation, scaling, and translation. The Transform3D object is represented internally as a 4x4 floating point matrix

Table 3-3 Attribute component objects

Node Component Objects - Geometry

A Geometry object is an abstract class which specifies the geometry component information required by a Shape3D node. Geometry objects describe both the geometry and topology of the Shape3D nodes which reference them.

The geometry component object hierarchy is shown in Figure 3-7

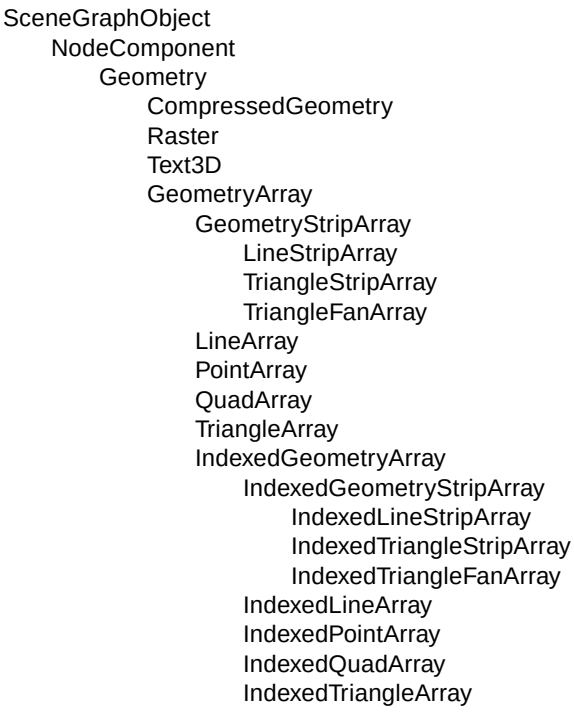

Figure 3-7 Geometry component object hierarchy

Geometry objects consist of four generic geometric types: CompressedGeometry, Raster, Text3D, and GeometryArray (of which there are many subclasses). Each of these geometric types define a visible object or set of objects. Non GeometryArray objects are listed and described in Table 3-4.

Node Name	Description
CompressedGeometry Object	Employs a special format for representing geometric information in one order of magnitude less space
Raster Geometry Object	Extends geometry to allow drawing a raster image that is attached to a 3D location in the virtual world.
Text3D Geometry Object	Consists of a set of 3D glyphs, created from an extruded font, that represents a text string

Table 3-4 Non GeometryArray geometry component objects

GeometryArray Objects

A GeometryArray object is an abstract class from which several classes are derived to specify a set of geometric primitives. A GeometryArray contains separate arrays of the following vertex components: coordinates, colors, normals and texture coordinates, and a bitmask indicating which of these components are present.

A single GeometryArray contains a predefined collection of per-vertex information and all of the vertices in a GeometryArray object have the same format and primitive type. Different GeometryArrays can contain different per-vertex information. One GeometryArray might contain only three-space coordinates while another might contain per-vertex coordinates, normals, colors, and texture coordinates. Yet a third GeometryArray might contain any subset of the previous examples.

The Geometry Component Objects which are derived from the GeometryArray object are listed and described in Table 3-5.

Node Name	Description
GeometryArray Objects	An abstract class from which several classes are derived to specify a set of geometric primitives
GeometryStripArray	An abstract class from which all strip primitives (line strip, triangle strip and triangle fan) are derived
LineStripArray	Draws an array of vertices as a set of connected line strips
TriangleStripArray	Draws an array of vertices as a set of connected triangle strips
TrangleFanArray	Draws an array of vertices as a set of connected triangle fans
LineArray	Draws the array of vertices as individual line segments using each pair of vertices to define a line segment to be drawn
PointArray	Draws the array of vertices as individual points
QuadArray	Draws the array of vertices as individual quadrilaterals using each group of four vertices to defines a quadrilateral to be drawn
TriangleArray	Draws the array of vertices as individual triangles using each group of three vertices to define a triangle to be drawn
IndexedGeometryArray Objects	An abstract class that extends GeometryArray to allow vertex data to be accessed via a level of indirection by adding corresponding arrays of coordinate indices, color indices, normal indices, and texture coordinate indices

Node Name	Description
IndexedGeometryStrip Array	An abstract class from which all indexed strip primitives (line strip, triangle strip and triangle fan) are derived; additionally specifies a number of strips and an array of per-strip vertex counts that indicates where the separate strips appear in the vertex array
IndexedLineStripArray	Draws an array of vertices as a set of connected line strips while an array of per-strip vertex counts specifies where the separate strips appear in the vertex array
IndexedTriangleStrip Array	Draws an array of vertices as a set of connected triangle strips while an array of per-strip vertex counts specifies where the separate strips appear in the vertex array
IndexedTrinagleFan Array	Draws an array of vertices as a set of connected triangle fans while an array of per-strip vertex counts specifies where the separate strips (fans) appear in the vertex array
IndexedLineArray	Draws the array of vertices as individual line segments with each pair of vertices defining a line segment to be drawn
IndexedPointArray	Draws the array of vertices as individual points
IndexedQuadArray	Draws the array of vertices as individual quadrilaterals with each group of four vertices defining a quadrilateral to be drawn
IndexedTriangleArray	Draws the array of vertices as individual triangles with each group of three vertices defining a triangle to be drawn

Table 3-5 GeometryArray geometry component objects

References

Sun Microsystems Computer Company posts product information in the form of data sheets, specifications, and white papers on its Internet World Wide Web page at: *http://www.sun.com*.

Look for this and other Sun technology publications:

The Java 3D API Specification Version 0.98, Sun Microsystems, Inc.

HelloUniverse: A Java 3D Sample Program

`HelloUniverse.java` is a simple code fragment that creates a cube, and a behavior object that rotates the cube at a constant rate of π/120.0 radians per frame.

HelloUniverse.java

```
public class HelloUniverse extends Applet {
   public BranchGroup createSceneGraph() {
       // Create the root of the subgraph
       BranchGroup objRoot = new BranchGroup();

       // Create the transform group node and initialize it to the
       // identity.  Enable the TRANSFORM_WRITE capability so that
       // our behavior code can modify it at runtime.  Add it to the
       // root of the subgraph.
       TransformGroup objTrans = new TransformGroup();
       objTrans.setCapability(
                         TransformGroup.ALLOW_TRANSFORM_WRITE);
       objRoot.addChild(objTrans);
       // Create a simple shape leaf node, add it to the scene graph.
       objTrans.addChild(new ColorCube().getShape());
```

```
        // Create a new Behavior object that will perform the desired
        // operation on the specified transform object and add it into
        // the scene graph.
        Transform3D yAxis = new Transform3D();
        Alpha rotationAlpha = new Alpha(
               -1, Alpha.INCREASING_ENABLE,
               0, 0,     4000, 0, 0,   0, 0, 0);
        RotationInterpolator rotator = new RotationInterpolator(
               rotationAlpha, objTrans, yAxis,
               0.0f, (float) Math.PI*2.0f);
        BoundingSphere bounds =
           new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
        rotator.setSchedulingBounds(bounds);
        objTrans.addChild(rotator);

        return objRoot;
    }

    public HelloUniverse() {
        setLayout(new BorderLayout());
        Canvas3D c = new Canvas3D();
        add("Center", c);
        // Create a simple scene and attach it to the virtual universe
        BranchGroup scene = createSceneGraph();
        UniverseBuilder u = new UniverseBuilder(c);
        u.addBranchGraph(scene);
    }
}

public class UniverseBuilder extends Object {
    // User-specified canvas
    Canvas3D canvas;

    // Scene graph elements that the user may want access to
    VirtualUniverse     universe;
    Locale              locale;
    TransformGroup      vpTrans;
    View                view;

    public UniverseBuilder(Canvas3D c) {
        this.canvas = c;

        // Establish a virtual universe, with a single hi-res Locale
        universe = new VirtualUniverse();
        locale = new Locale(universe);
```

```
        // Create a PhysicalBody and Physical Environment object
        PhysicalBody body = new PhysicalBody();
        PhysicalEnvironment environment =
                                        new PhysicalEnvironment();

        // Create a View and attach the Canvas3D and the physical
        // body and environment to the view.
        view = new View();
        view.addCanvas3D(c);
        view.setPhysicalBody(body);
        view.setPhysicalEnvironment(environment);

        // Create a branch group node for the view platform
        BranchGroup vpRoot = new BranchGroup();

        // Create a ViewPlatform object, and its associated
        // TransformGroup object, and attach it to the root of the
        // subgraph.  Attach the view to the view platform.
        Transform3D t = new Transform3D();
        t.set(new Vector3f(0.0f, 0.0f, 5.0f));
        ViewPlatform vp = new ViewPlatform();
        TransformGroup vpTrans = new TransformGroup(t);

        vpTrans.addChild(vp);
        vpRoot.addChild(vpTrans);

        view.attachViewPlatform(vp);
        view.setFieldOfView(40.0 * Math.PI / 180.0);
        view.setFrontClipDistance(1.0);
        view.setBackClipDistance(10.0);

        // Attach the branch graph to the universe, via the Locale.
        // The scene graph is now live!
        locale.addBranchGraph(vpRoot);
    }

    public void addBranchGraph(BranchGroup bg) {
        locale.addBranchGraph(bg);
    }
}
```

```
public class ColorCube extends Object {
   private static final float[] verts = {
   // front face
       1.0f, -1.0f,  1.0f,     1.0f,  1.0f,  1.0f,
      -1.0f,  1.0f,  1.0f,    -1.0f, -1.0f,  1.0f,
   // back face
      -1.0f, -1.0f, -1.0f,    -1.0f,  1.0f, -1.0f,
       1.0f,  1.0f, -1.0f,     1.0f, -1.0f, -1.0f,
   // right face
       1.0f, -1.0f, -1.0f,     1.0f,  1.0f, -1.0f,
       1.0f,  1.0f,  1.0f,     1.0f, -1.0f,  1.0f,
   // left face
      -1.0f, -1.0f,  1.0f,    -1.0f,  1.0f,  1.0f,
      -1.0f,  1.0f, -1.0f,    -1.0f, -1.0f, -1.0f,
   // top face
       1.0f,  1.0f,  1.0f,     1.0f,  1.0f, -1.0f,
      -1.0f,  1.0f, -1.0f,    -1.0f,  1.0f,  1.0f,
   // bottom face
      -1.0f, -1.0f,  1.0f,    -1.0f, -1.0f, -1.0f,
       1.0f, -1.0f, -1.0f,     1.0f, -1.0f,  1.0f,
   };
   private static final float[] colors = {
   // front face (red)
      1.0f, 0.0f, 0.0f,       1.0f, 0.0f, 0.0f,
      1.0f, 0.0f, 0.0f,       1.0f, 0.0f, 0.0f,
   // back face (green)
      0.0f, 1.0f, 0.0f,       0.0f, 1.0f, 0.0f,
      0.0f, 1.0f, 0.0f,       0.0f, 1.0f, 0.0f,
   // right face (blue)
      0.0f, 0.0f, 1.0f,       0.0f, 0.0f, 1.0f,
      0.0f, 0.0f, 1.0f,       0.0f, 0.0f, 1.0f,
   // left face (yellow)
      1.0f, 1.0f, 0.0f,       1.0f, 1.0f, 0.0f,
      1.0f, 1.0f, 0.0f,       1.0f, 1.0f, 0.0f,
   // top face (magenta)
      1.0f, 0.0f, 1.0f,       1.0f, 0.0f, 1.0f,
      1.0f, 0.0f, 1.0f,       1.0f, 0.0f, 1.0f,
   // bottom face (cyan)
      0.0f, 1.0f, 1.0f,       0.0f, 1.0f, 1.0f,
      0.0f, 1.0f, 1.0f,       0.0f, 1.0f, 1.0f,
   };

   private Shape3D shape;
```

```
    public ColorCube() {
        QuadArray cube = new QuadArray(24,
                         QuadArray.COORDINATES | QuadArray.COLOR_3);

        cube.setCoordinates(0, verts);
        cube.setColors(0, colors);

        shape = new Shape3D(cube, new Appearance());
    }

    public Shape3D getShape() {
        return shape;
    }
}
```

Sun Microsystems Computer Company
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300
FAX 415 969-9131
http://www.sun.com

Sales Offices

Argentina: +54-1-311-0700
Australia: +61-2-9844-5000
Austria: +43-1-60563-0
Belgium: +32-2-716-7911
Brazil: +55-11-524-8988
Canada: +905-477-6745
Chile: +56-2-638-6364
Colombia: +571-622-1717
Commonwealth of Independent States:
+7-095-956-5470
Czech/Slovak Republics:
+42-2-205-102-33
Denmark: +45-44-89-49-89
Estonia: +372-6-308-900
Finland: +358-0-525-561
France: +33-01-30-67-50-00
Germany: +49-89-46008-0
Greece: +30-1-680-6676
Hong Kong: +852-2802-4188
Hungary: +36-1-202-4415
Iceland: +354-563-3010
India: +91-80-559-9595
Ireland: +353-1-8055-666
Israel: +972-9-956-9250
Italy: +39-39-60551
Japan: +81-3-5717-5000
Korea: +822-3469-0114
Latin America/Caribbean:
+1-415-688-9464
Latvia: +371-755-11-33
Lithuania: +370-729-8468
Luxembourg: +352-491-1331
Malaysia: +603-264-9988
Mexico: +52-5-258-6100
Netherlands: +31-33-450-1234
New Zealand: +64-4-499-2344
Norway: +47-2218-5800
People's Republic of China:
Beijing: +86-10-6849-2828
Chengdu: +86-28-678-0121
Guangzhou: +86-20-8777-9913
Shanghai: +86-21-6247-4068
Poland: +48-22-658-4535
Portugal: +351-1-412-7710
Singapore: +65-224-3388
South Africa: +2711-805-4305
Spain: +34-1-596-9900
Sweden: +46-8-623-90-00
Switzerland: +41-1-825-7111
Taiwan: +886-2-514-0567
Thailand: +662-636-1555
Turkey: +90-212-236-3300
United Arab Emirates:
+971-4-366-333
United Kingdom: +44-1-276-20444
United States: +1-800-821-4643
Venezuela: +58-2-286-1044
Worldwide Headquarters:
+1-415-960-1300