Editors: Lawrence Rosenblum and Michael Macedonia

The Java 3D API and Virtual Reality

Henry A. Sowizral and Michael F. Deering

Sun Microsystems

Java 3D proves a natural choice for any Java programmer wanting to write an interactive 3D graphics program. A programmer constructs a scene graph containing graphic objects, lights, sounds, environmental effects objects, and behavior objects that handle interactions or modify other objects in the scene graph. The programmer then hands that scene graph to Java 3D for execution. Java 3D starts rendering objects and executing behaviors in the scene graph. Virtual reality applications go through an identical writing process. However, before a user can use such an application, Java 3D must additionally know about the user's physical characteristics (height, eye separation, and so forth) and physical environment (number of displays, their location, trackers, and so on). Not surprisingly, such information varies from installation to installation and from user to user. So Java 3D lets application developers separate their application's operation from the vagaries of the user's final display environment.

The Java 3D application programmer's interface (API) provides a very flexible platform for building a broad range of graphics applications. Developers have already used Java 3D to build applications in a variety of domains including mechanical CAD, molecular visualization, scientific visualization, animation previews, geographic information systems, business graphics, 3D logos, and educational offerings. Virtual reality applications have included immersive workbench applications, head-tracked shutter-glass-based desktop applications, and portals (a cave-like room with multiple back-projected walls).

What makes Java 3D good for VR?

Writing a VR program requires a substantial programming effort. A developer must either write code to handle the various input and display devices that the application might encounter or, alternatively, the developer will need to rely on a programming API designed to support VR applications. A typical VR application must track a user's head position and orientation (so that it can generate images of the virtual world as if the user's head were in that position and orientation). Additionally, the application might need to track other parts of the user's body, such as an arm, a hand, or a leg, so that it can make user interaction with that environment easier by including a virtual view of that body part. The application must use these tracker inputs in placing objects within a view and in specifying the appropriate position and orientation in generating 3D images.

An API that enables VR applications must support generating 3D graphics, handling input trackers, and continuously integrating that tracker information into the rendering loop. The Java 3D API includes specific features for automatically incorporating head tracker inputs into the image generation process and for accessing other tracker information to control other features. It does this through a new view model that separates the user's environment from the application code. The API also explicitly defines the values returned by six-degrees-of-freedom detectors as primitive Java 3D objects, called Sensors. Together, the new view and input models make it quite easy to turn interactive 3D graphics applications into VR-enabled applications.

A detailed description of the API appears in the Java 3D specification.

A Java 3D program

Creating a Java 3D application or applet involves constructing a virtual universe and inserting one or more scene graphs into that universe. A virtual universe consists of superstructure objects including one Universe object, one but possibly more Locale objects, and one or more scene graphs containing node objects arranged in tree structures. The scene graphs, called branch graphs because their root node must always be a `BranchGroup` node, contain the objects to render, the lights to apply, the behaviors to execute, the sounds to play, and so on. Branch graphs containing `content` nodes are called content branches. Those that contain a ViewPlatform object—the object that specifies the user's position and orientation—are called view branches.

Figure 1 shows a Java 3D universe with multiple branch graphs. Note that branch graphs describe what should be rendered, not the order in which to render objects. Neither node order nor node placement within a graph determines an order of execution or rendering. The path between a graph's root node and any leaf of that root node uniquely determines how to draw (or process) that path's leaf node. The almost total lack of implied order (the `Ordered` group node provides the only exception) means that Java 3D can execute behaviors and render objects in any order best suited to the system's underlying architecture, including parallel systems.

Branch graphs

Java 3D branch graphs are trees where each node in the branch graph has only one parent. Though this may

0272-1716/99/$10.00 © 1999 IEEE

seem restrictive, Java 3D does support sharing common scene graphs through an extra scene-graph mechanism. Leaf nodes called **`Link`** nodes can link to (reference) specially defined, shareable subgraphs.

The nodes in a branch graph fall into one of two categories: group or leaf nodes. Group nodes act as the glue that holds a branch graph together. Group nodes combine other nodes (group and leaf nodes) into one common unit, they position and orient those nodes, and, in a limited way, they control rendering order. Leaf nodes represent items in a branch graph such as objects, lights, and behaviors. Leaf nodes do not refer to nodes but only other Java 3D objects like NodeComponent objects. These objects hold Java-3D-specific information such as shape geometry, shape appearance, textures, media containers, and so on.

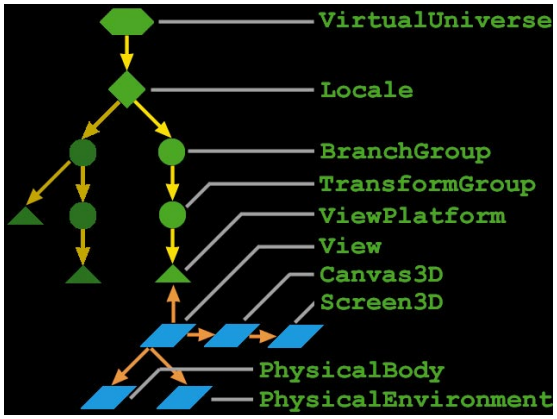
Group nodes include **`BranchGroup`**, **`TransformGroup`**, **`Switch`**, **`OrderedGroup`**, and **`SharedGroup`** nodes. The **`BranchGroup`** node serves as the root of a branch graph. The **`TransformGroup`** serves to position and orient its constituent subgraphs. The **`Switch`** node selects among one or more subgraphs for rendering. The **`OrderedGroup`** node ensures that its children are rendered in the specified order. Finally, a **`SharedGroup`** node, just like a **`BranchGroup`** node, serves as the root of a scene graph; however, they're called shared graphs. Shared graphs can never exist as part of a Java 3D scene graph directly; however, **`Link`** nodes can refer to them as if they were subroutines.

Leaf nodes do not have children. Instead they either contain information or refer to objects, called node component objects, that contain information Java 3D needs. Example leaf nodes include **`Shape3D`**, **`ViewPlatform`**, **`Sound`**, various types of light nodes, various predefined behaviors, user defined behaviors, and others. The first two leaf nodes, the **`Shape3D`** and **`ViewPlatform`** nodes, specify two important elements in a graphics system. The **`Shape3D`** node specifies a geometric object and its appearance, while the **`ViewPlatform`** node specifies the viewer's current location and orientation (and scale) within the virtual world. The other leaf nodes, though important for specifying various elements of a virtual world, have less importance in describing Java 3D's view model and the input device model.

The **`ViewPlatform`** node serves to locate and orient a viewer or a user within a virtual world. An application can manipulate a ViewPlatform just like any other object in a branch graph. The application can translate the ViewPlatform, rotate it, or even scale it. By manipulating the location and orientation of a ViewPlatform, the application can move the ViewPlatform (and thus any viewer associated with it) through the virtual world. The ViewPlatform acts very much like a cart in an amusement park ride. The cart takes you along a predetermined route through the world, but it doesn't restrict you from moving around or looking in different directions.

The Java 3D view model

The Java 3D view model consists of two intertwined components: the virtual world, as represented by the ViewPlatform object, and the physical world, as represented by the View object and its associated objects.



1 A Java 3D universe.

The View object and its associated objects describe the user's display environment in considerable detail. They specify how many displays occupy that environment, whether it consists of a desktop monitor, three back-projected walls, or two displays in a head-mount. They specify the availability (or not) of a head tracker and its location relative to the display(s). They specify the user's height and interpupilary distance. They also specify various policies that determine how to interpret various changes in the many parameters.

The view model imposes a clean separation between the virtual and the physical world. At the same time, it builds a bridge between the two worlds by defining a one-to-one-and-onto correspondence between the virtual space surrounding the ViewPlatform and the physical space as specified by the View objects. A point in one space has a corresponding point in the other space.

Why a new model?

A camera-based model provides applications with a variety of controls over the camera's parameters such as its location and orientation, its aspect ratio and field-of-view, and others. These controls work well in many settings. Unfortunately, they fail to provide sufficient flexibility in some cases, and in other cases they allow an inappropriate level of control. For example, in a portal (or cave-like) environment, where the user stands inside a multiwalled environment with multiple display surfaces, the camera model proves insufficient, since it can only provide control over one display surface at a time.

A different problem exists in an immersive environment, where users wear head-mounted displays. The optical elements in the head-mount determine the application's permissible field-of-view. Any deviation from that value will create a significant mismatch in perceived movement versus the movement shown in the display. Such a mismatch induces nausea rather quickly. This example demonstrates a case where the application should not control a particular parameter.

About the view model

The central concept in the view model is coexistence. The viewer exists in the physical world as well as in the virtual world. By allowing sufficient control over describing the user's physical environment, Java 3D can map movements of the ViewPlatform or the user's head onto

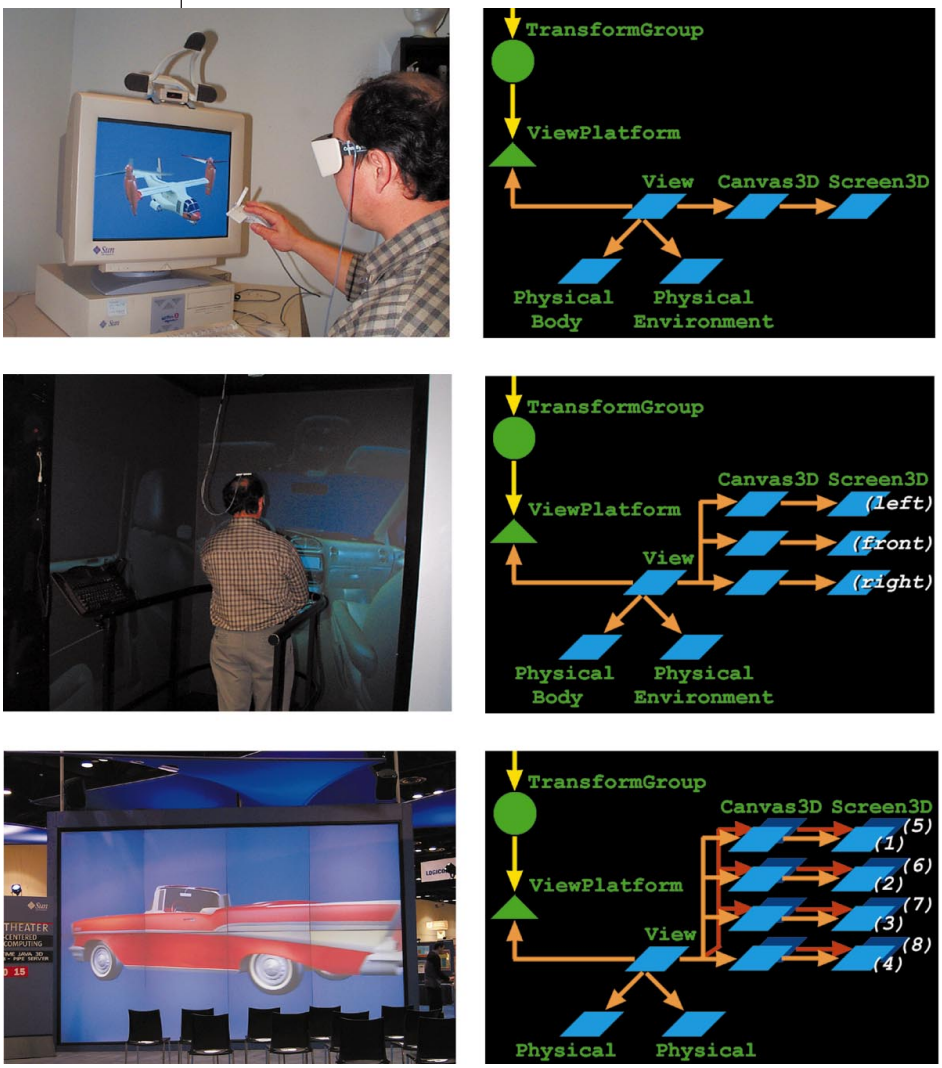
the corresponding, appropriate, view computations.

The View object and its associated objects describe the viewer's initial position relative to display screens in the display environment. To prevent confusion, we call the individual display screens image plates. The viewer's eyepoints and the environment's image plates have well-defined locations in the physical environment. Because of Java 3D's coexistence concept, the viewer's eyepoints and the image plates' position and orientation have well-defined locations in the virtual environment. Since we know the location of the eyepoints and the image plates in the virtual world, it's relatively straightforward to construct the appropriate view frustum for computing a projective 3D image on a per-image-plate basis. If head tracking is available, we can straightforwardly translate head movements into new eyepoints relative to the image plates. If head tracking is not available, the frustums only need to be constructed once per image plate.

Coexistence is essentially a mapping from the physical space containing the viewer and the image plates and the virtual space surrounding the ViewPlatform and back again. This mapping provides the means to translate movement or selection within one space directly into a similar movement or selection within the other space.

Think of the ViewPlatform as the cart in an amusement park ride. Wherever the ViewPlatform takes you (the viewer), you go. However, just as you can turn your head to look around while enjoying a ride, you can look around as the ViewPlatform navigates around the virtual environment. By tracking the viewer's head position and orientation, we can instantaneously convert that head movement into the corresponding movement within the ViewPlatform's coordinate system. This virtual head position and orientation give us sufficient information to automatically generate the corresponding view frustums. If a head tracker is not available, then the user experiences the application's navigation as if fixed rigidly in a seat without the ability to move head or limb.

If a developer doesn't know the exact structure of the user's environment, how can the developer write an application that can adapt to that environment? Java 3D provides a set of Universe utility classes that construct a virtual universe and all associated objects needed for a fully functioning view branch. The universe construction classes read the local machine's configuration properties and the user's configuration properties and, using that information, construct the appropriate view-branch objects reflecting those configuration values. The developer only needs to maneuver the location and orientation of the ViewPlatform object to get the desired effect.



2 Three different display environments and the associated view objects that define those environments. The top two images show a fish-tank VR system and its associated single-canvas single-screen view branch (the same view branch structure that a single screen nonheadtracked environment would use). The middle two images show a back projected three-walled immersive room and its associated three-canvas three-screen view branch. The bottom two images show a composite screen consisting of eight tessellated projectors providing an aggregate resolution of 4096 × 2560 and the associated eight-canvas eight-screen view branch.

How the view model works

The view model operates as a highly optimized constraint system. It has two grounding policies. One policy is used in environments where the user's eye can move relative to the display surface. This policy supports non-head-tracked, flat-screen display environments as well as all environments where the user can move relative to any display surfaces—environments such as immersive workbenches, multiwalled, back-projected rooms, or even desktop, head-tracked environments (see Figure 2). The second policy is used in environments where the user's eye does not move relative

to the display surface. This policy handles head-mounted displays, augmented-reality environments, and boom-mounted displays. The Java 3D API specification describes both constraint systems in detail.

The view model assumes that a Java 3D application will at times execute in an environment with a head tracker. In such environments, the view-model's constraint system automatically incorporates the head-tracker's latest position and orientation information when computing view frustums. The head tracker information comes via the input device interface.

Input devices in Java 3D

Supporting head tracking requires Java 3D to provide access to six-degrees-of-freedom (6DOF) tracker information. Different tracking devices work differently and require that the computer interact with them in different ways. A device driver that operates a Polhemus device will definitely not operate an Ascension device. A Magellan driver will not operate a Spaceball. As new devices become available, they too will require custom device drivers.

Rather than trying to support all possible 6DOF input devices, Java 3D defines an InputDevice interface that tracker vendors or developers can use to support a particular 6DOF device. The InputDevice interface requires the implementer of a device driver to define nine methods. The nine methods provide device-specific semantics for open, close, and read operations as well as a minimal number of state-setting and state-query methods.

A Java 3D environment can have any number of InputDevices. Moreover, InputDevices need not be real physical devices. They can be virtual devices. It's perfectly reasonable to write a software-input device that translates the manipulation of a virtual trackball via mouse movements into 6DOF values. Or, alternatively, to develop an InputDevice driver that reads time-stamped 6DOF values from a file or a network connection and then regenerates that data as if it arrived from a real tracker.

Each input device has associated with it a fixed number of Sensor objects. A Sensor object represents one source of 6DOF data associated with that device. Whenever an input device driver generates new data, it does so for all of its Sensor objects. A Sensor object contains a fixed number of SensorRead objects arranged as a circular buffer. This circular buffer contains the "*n*" most recent SensorRead values for that Sensor. Having multiple SensorRead values makes it possible to perform data averaging and prediction on a particular stream of input values.

SensorRead objects contain a time stamp, a 6DOF value, and an integer array of button values. The time stamp provides essential information for averaging or predicting 6DOF values. The button values let a device driver encode not only simple button-up and button-down states, but also a trigger, or a knob. Sensors can also be used to support fewer degree-of-freedom devices, such as conventional joysticks.

Using abstract sensors

Application developers do not use input devices directly. Java 3D abstracts away input devices by providing an array of available Sensors. This array of Sensor objects belongs under the PhysicalEnvironment object (one of the objects associated with the View object). The array consists of pointers to the objects associated with InputDevices.

This array of Sensors lets an application developer require some number of 6DOF inputs and then have users provide the device drivers available on their systems. Typically, a Universe construction utility enumerates InputDevices and the Sensors associated with each device, then assigns each Sensor it finds to a slot in the array of Sensors. Since application code never references devices, only Sensors, the Universe utility can rearrange sensors at will as long as the appropriate sensors are associated with the corresponding appropriate Sensor index used by the application.

The Universe construction utility reads a device and application configuration profile and, using that information, assigns a particular Sensor on a particular device to a particular Sensor index. In essence, this facility provides an input device/sensor patch panel.

Finally, as a last resort, if an application requires 6DOF inputs and the user does not have tracking hardware, the Universe construction utility can construct virtual sensors.

Summary

Java programmers can quickly and easily define graphics programs using Java 3D's scene graph classes. An expanded view model lets applications seamlessly operate in a variety of single- and multiple-display, non-head-tracked and head-tracked, display environments. This view model relies on the flexible InputDevice interface that Java 3D provides to remove most of the vagaries of hardware trackers. ■

Readers may contact Sowizral at Sun Microsystems, 901 San Antonio Rd., MS UMPK27-101, Palo Alto, CA 94303-4900, e-mail henry.sowizral@eng.sun.com.

Resources

The Java 3D specification includes details of the API:

H.A. Sowizral, K.C. Rushforth, and M.F. Deering, *The Java 3D API Specification*, Addison Wesley, Reading, Mass., 1998.

To explore sensors, see

H.A. Sowizral, "Virtual Sensors: Handling Real and Computationally Derived Information Consistently," *Proc. SPIE Stereoscopic Displays and Virtual Reality Systems II*, Vol. 2409, S. Fisher, J. Merritt, and M. Bolas, eds., SPIE, Bellingham, Wash., 1995, pp. 246-254.

K. Zikan et al., "Fusion of Absolute and Incremental Position and Orientation Sensors," *Proc. SPIE Telemanipulator and Telepresence Technologies*, Vol. 2351, H. Das, ed., SPIE, Bellingham, Wash., 1994, pp. 316-327.

For more on head tracking and orientation,

K. Zikan et al., "A Note on Dynamics of Human Head Motions and on Predictive Filtering of Head-Set Orientations," *Proc. SPIE Telemanipulator and Telepresence Technologies*, Vol. 2351, H. Das, ed., SPIE, Bellingham, Wash., 1994, pp. 328-336.

See the Sun Microsystems Web sites on Java 3D:

http://www.sun.com/desktop/java3d
http://java.sun.com/products/java-media/3D