The Virtual Reality Modeling Language and Java

Don Brutzman
Code UW/Br, Naval Postgraduate School
Monterey California 93943-5000 USA
brutzman@nps.navy.mil

Communications of the ACM, vol. 41 no. 6, June 1998, pp. 57-64.
http://www.web3D.org/WorkingGroups/vrtp/docs/vrmljava.pdf

Abstract. The Virtual Reality Modeling Language (VRML) and Java provide a standardized, portable and platform-independent way to render dynamic, interactive 3D scenes across the Internet. Integrating two powerful and portable software languages provides interactive 3D graphics plus complete programming capabilities plus network access. Intended for programmers and scene authors, this paper provides a VRML overview, synopsizes the open development history of the specification, provides a condensed summary of VRML 3D graphics nodes and scene graph topology, describes how Java interacts with VRML through detailed examples, and examines a variety of VRML/Java future developments.

Overview. The Web is being extended to three spatial dimensions thanks to VRML, a dynamic 3D scene description language that can include embedded behaviors and camera animation. A rich set of graphics primitives provides a common-denominator file format which can be used to describe a wide variety of 3D scenes and objects. The VRML specification is now an International Standards Organization (ISO) specification (VRML 97).

Why VRML and Java together? Over twenty million VRML browsers have shipped with Web browsers, making interactive 3D graphics suddenly available for any desktop. Java adds complete programming capabilities plus network access, making VRML fully functional and portable. This is a powerful new combination, especially as ongoing research shows that VRML plus Java provide extensive support for building large-scale virtual environments (LSVEs). This paper provides historical background, a detailed overview of VRML 3D graphics, example VRML-Java test programs, and a look ahead at future work.

Development history. "Birds of a feather" sessions organized by Mark Pesce and Tony Parisi at the 1994 World Wide Web and SIGGRAPH conferences led to the formation of an open working group to examine potential technologies for a "virtual reality markup language," later changing markup to modeling. The initial goal of this effort was selection of a general 3D scene-description file format that was suitable for modification by addition of hyperlink semantics similar to the HyperText Markup Language (HTML). Open nomination and discussion of seven possible candidates led to selection of an extended subset of Silicon Graphics Inc. (SGI) *OpenInventor* file format [Wernicke 94] [Carey, Strauss 92]. Extensive online proposals and discussions punctuated by reported results became the hallmark of the VRML development process. Key contributions of the initial VRML 1.0 standard were selection of a core set of object-oriented graphics constructs augmented by hypermedia links, all suitable for geometry rendering by Web browsers on personal computers or workstations.

Language extensions for VRML. Although spirited debate and rough consensus successfully delivered an initial specification, there were two major limitations in VRML 1.0: lack of support for dynamic scene animation, and no traditional programming language constructs. Difficult issues regarding real-time animation in VRML 1.0 included entity behaviors, user-entity interaction and entity coordination. VRML 2.0 development tackled these issues directly, using event-driven ROUTEs to connect 3D nodes and fields to behavior-driven sensors and timing. "Language wars" were avoided by allowing other programming languages to communicate with the VRML scene via a Script node. Initial languages chosen are JavaScript for light-weight in-line calculations, and Java for full-fledged programming and network access. If Java or JavaScript are supported in a VRML browser, they must conform to the formal interface specified in (VRML 97) Annexes B and C, respectively. Major browsers now support both. Ongoing development of VRML continues via open working groups supported by the nonprofit VRML Consortium (VRMLC 97).

Behaviors. The term "behaviors" refers to making changes in the structure or appearance of a 3D scene. Thus a broad definition of a VRML behavior might be "any change in the nodes or fields of a VRML scene graph." VRML 97 provides local key-frame animation mechanisms and remote scripting language hooks (i.e. an applications

programming interface or API) to any scene graph component. Dynamic scene changes can be stimulated by scripted actions, message passing, user commands or behavior protocols, implemented using either via Java calls or complete VRML scene replacement. General approaches for VRML behaviors and scene animation are possible that provide simplicity, security, scalability, generality and open extensions. Using Java is the most powerful way for 3D scene authors to explore the many possibilities provided by VRML.

Getting started. Numerous useful resources for obtaining browser software, subscribing to the *www-vrml* mailing list, tutorials etc. are available via the VRML Repository (SDSC 97). Excellent books for learning VRML include (Ames 97) (Hartman 97). The VRML 97 specification is online, and an annotated version of the specification by principal VRML 97 architects Rikk Carey and Gavin Bell is available in book form and online (Carey 97). Resources on Java network programming include (Harold 97) and (Hughes 97). Intermediate and advanced textbooks for combined VRML and Java programming are (Lea 97) and (Roehl 97), respectively.

3D graphics nodes. For most programmers, there are many new language concepts and terms in VRML. An overview of this admittedly large language is necessary before describing how Java works in combination with it. This section describes the 3D-specific VRML nodes.

Big picture. Since VRML is a general 3D scene description language that can be used as an interchange file format, there are a large number of 3D graphics nodes available. These nodes are organized in a hierarchical structure called a *scene graph*, i.e. a directed acyclic graph. The primary interaction model for 3D VRML browsers is point and click, meaning that content can have embedded links just like the 2D HyperText Markup Language (HTML). VRML 3D browsers are typically installed as plugins within 2D browsers (such as Netscape and Internet Explorer). VRML is optimized for general 3D rendering and minimal network loading, taking advantage of extensive VRML browser capabilities. For examples, geometric primitives such as IndexedFaceSet allow authoring tools and datasets to create highly complex objects. Textures and MovieTextures can wrap 2D images over and around arbitrary geometry. Sound nodes embed spatialized audio together with the associated shapes. Lighting and camera control provide complete control of presentation and rendering, including animation of camera position to create flyby explorations or dramatic visual transitions. Finally the Script node interface to Java allows modification or generation of any VRML content.

Shape. The Shape node is a container node which collects a pair of components called geometry and appearance. Nodes that describe the objects being drawn are typically used in the geometry and appearance fields of the Shape node.

Geometry. Box, Cone, Cylinder and Sphere are nodes for simple regular polyhedra (sometimes called *primitives*) which provide basic building blocks for easy object construction. The Text node simplifies specification of planar or extruded polygonal text. ElevationGrid is a table of height (y) values corresponding to x-z horizontal spacing indices. The Extrusion node stretches, rotates and scales a cross-section along a spine into a wide variety of possible shapes. IndexedFaceSet, IndexedLineSet and PointSet can create 3D geometry of arbitrary complexity. Since Extrusion, IndexedFaceSet, IndexedLineSet and PointSet are specified by sets of coordinate points, color values and normal vectors can be specified point by point for these nodes.

Appearance. The appearance of geometry is primarily controlled by specifying color values or texture images. The Material node permits specification of diffuse, emissive and specular color components (which roughly correspond to reflective, glowing and shininess colors). Material nodes can also specify transparency. Colors are specified as red-green-blue (RGB) triples ranging from 0 (black) to 1 (full intensity). Transparency value a similarly ranges from 0 (opaque) to 1 (completely transparent). As an alternative or supplement to material values, three types of texture nodes are provided. ImageTexture is the most common: a 2D image is wrapped around (or over) the corresponding geometry. MovieTexture allows use of time-dependent textures (e.g. MPEG movie files) as the image source. Finally TextureTransform specifies a 2D transformation in texture coordinates for advanced texture-mapping techniques, i.e. applying specific repetitions or orientations of a texture to corresponding geometry features.

Scene topology: grouping and child nodes. VRML syntax and node typing also helps enforce a strict hierarchical structure of parent-child relationships, so that browsers can perform efficient rendering and computational optimizations. Grouping nodes are used to describe relationships between Shapes and other child nodes. Moreover, the semantics of a scene graph carefully constrain the ways that nodes can be organized together. Child nodes come under grouping nodes to comply with the scene graph hierarchy inherent in any author's VRML scene. In addition to Shapes, child nodes describe lighting, sound, viewing, action sensors and animation interpolators. This section

synopsizes the full scope of VRML, the language; readers familiar with 3D graphics concepts may prefer skipping ahead to the Java section.

Grouping. Fundamental to any VRML scene is the notion that graphics nodes can only be grouped in ways that make sense. Grouping is used to describe spatial and logical relationships between nodes, and as an intentional side result also enable efficient rendering by 3D browsers. The Group node is the simplest of the grouping nodes: it merely collects child nodes, with no implied ordering or relationship other than equivalent status in the scene graph. The Transform node similarly groups child nodes, but first applies rotation, scaling and translation to place child nodes in the proper coordinate frame of reference. The Billboard node keeps its child nodes aligned to always face the viewing camera, either directly or about an arbitrary rotation axis. The Collision node lets an author specify a bounding box which serves as a proxy to simplify collision detection calculations for grouped child nodes. The Switch node renders only one (or none) of its child nodes, and is useful for collecting alternate renderings of an object which might be triggered by external behaviors. The LOD (level of detail) node also renders only one of multiple child nodes, but child selection is triggered automatically based either on viewer-to-object distance or on frame rate. Thus LOD enables the browser to efficiently select high-resolution or low-detail alternative renderings on-the-fly in order to support interactive rendering.

Grouping and the Web. Since the Web capabilities of VRML are analogous to HTML, two types of grouping nodes enable Web connectivity in 3D scenes. The Inline node allows importing additional 3D data from another VRML world into the current VRML world. In contrast, the Anchor node creates a link between its child nodes and an associated Uniform Resource Locator (URL) web address. When the child geometry of an Anchor node is clicked by the user's mouse, the current VRML scene is entirely replaced by the VRML scene specified in the Anchor URL. Multiple strings can be used to specify any URL, permitting browsers to preferentially load local copies before searching for remote scenes or backup locations. Typically browsers highlight "hot links" in a 3D scene by modifying the mouse cursor when it is placed over Anchor-enabled shapes.

Lighting and sound. Virtual lights in a 3D scene are used to determine illumination values when rendering. Lights cannot be "seen" in the world directly, rather they are used to calculate visibility, shininess and reflection in accordance with a carefully specified mathematical lighting model. The DirectionalLight node illuminates using parallel rays, the PointLight node models rays radiating omnidirectionally from a point, and the SpotLight node similarly provides radial rays constrained within a conical angle. Lights include color and intensity values which are multiplied against material values to calculate produce proper shading. The Sound node places an audio clip at a certain location, and with nonrendered ellipsoids surrounding it determine minimum and maximum threshold distances. Sound can repeat, be rendered spatially relative to user location, and be triggered on/off by events. Embedding various lights and sounds at different locations within a scene can produce dramatic results.

Viewing. Most 3D nodes describe location, size, shape and appearance of a model. The Viewpoint node specifies position, orientation and field of view for the virtual camera that is used to "view" (i.e. calculate) the 3D scene and render the screen image. Most objects and scenes contain a number of named viewpoints to encourage easy user navigation. The NavigationInfo node extends the camera concept to include the notion of an Avatar bounding box, used to determine camera-to-object collision and height of eye when "gravity" (i.e. terrain following) is enabled. NavigationInfo also toggles a "headlight" in the field of view for default illumination, and can switch the browser among a variety of user-navigation metaphors (FLY, EXAMINE, WALK etc.). The Fog node specifies color and intensity of obscuring fog, which is calculated relative to distance from viewer. The Background node allows specifying sky and ground color profiles, ranging smoothly from zenith to horizon to nadir. Background also permits specifying six images for texture box, which always sits beyond other rendered objects. These various viewing nodes provide rich functionality for animating viewpoint, assisting user navigation, and providing environmental effects.

Action sensors. Sensors detect change in the scene due to passage of time (TimeSensor), user intervention or other activity such as viewer proximity (VisibilitySensor). Sensors produce time-stamped events whose values can be routed as inputs to other nodes in the scene. User intervention is often as simple as direct mouse interaction with a shape via a TouchSensor, or interaction with a constraining bounding geometry specified by PlaneSensor, ProximitySensor or SphereSensor. Consistently typed input and output events are connected to correspondingly typed fields in the scene graph via ROUTEs.

Animation interpolators. Key-frame animation typically consists of simple time-varying values applied to the fields of the appropriate node. Smooth in-between animation can be interpolated linearly as long as key values are at sufficient resolution. Linear interpolators are provided for Color, Coordinate, Normal, Orientation, Position and scalar fields. Linear interpolation is sufficient for many demanding applications, including most humanoid animation. As

with Sensors, Interpolator inputs and outputs are connected with other nodes via ROUTEs. For performance reasons, use of these standard VRML sensors and interpolators is usually preferable to writing a custom script, since they can efficiently perform most authors' intended tasks. Scripts are the mechanism where authors can extend the action and animation capabilities of VRML.

Prototypes. Prototypes (PROTO and EXTERNPROTO) allow creation of new VRML node types by authoring combinations of nodes and fields from other preexisting node types. In this sense, a PROTO definition is somewhat analogous to a macro definition. In order to avoid completely copying a PROTO for each file where it is used, the EXTERNPROTO definition specifies remote URL where the original PROTO is defined, along with the interface to permit local type-checking by browsers during scene loading. The EXTERNPROTO mechanism thus allows construction of PROTO libraries for easy access and reuse.

Graphics example. No doubt dedicated readers are fully convinced by this point that VRML contains a great deal of functionality! To regain clarity, a canonical "Hello world" example in Figures 2a and 2b displays basic VRML syntax. This scene is available at www.stl.nps.navy.mil/~brutzman/vrml/examples/course/hello_world.wrl

```
#VRML V2.0 utf8
Group {
  children [
   Viewpoint {
      description "initial view"
      position    6 -1 0
      orientation 0  1 0 1.57
   }
   Shape {
      geometry Sphere   { radius 1 }
      appearance  Appearance  {
        texture ImageTexture {
         url "earth-topo.png"
   }  } }
   Transform {
      translation 0 -2 1.25
      rotation    0  1 0   1.57
      children [
         Shape {
            geometry Text {
             string [" Hello" "world!"]
                  }
            appearance  Appearance {
              material Material {
               diffuseColor 0.1 0.5 1
         } } }
      ]
   }
  ]
}
```

Figure 2a. VRML source *hello_world.wrl*

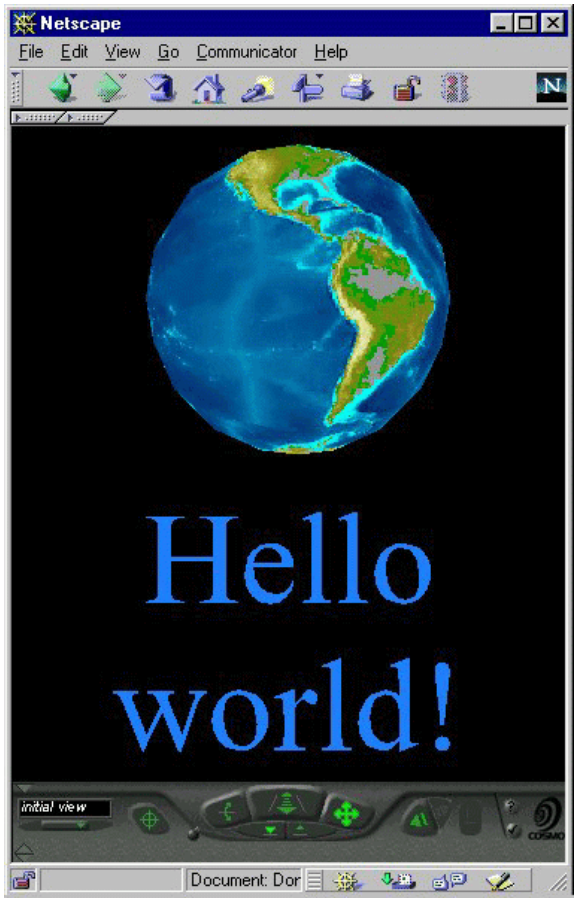

Figure 2b. VRML scene *hello_world.wrl*

VRML and Java: scripts, events, naming and ROUTEs. Interfaces between VRML and Java are effected through Script nodes, an event engine, DEF/USE naming conventions, and ROUTEs connecting various nodes and fields in the VRML scene. VRML provides the 3D scene graph, Script nodes encapsulate Java functionality, and ROUTEs provide the wiring that connects computation to rendering.

Scripts. Script nodes appear in the VRML file, encapsulating the Java code and providing naming conventions for interconnecting Java variables with field values in the scene. (Similar scripting conventions are specified for JavaScript). Interfaced Java classes import the `vrml.*` class libraries in order to provide type

conversion (for both nodes and simple data types) between Java and VRML. Java classes used by Script nodes must extend the `vrml.node.Script` class in order to interface properly with the VRML browser. The basic interface and a good description of Script nodes is excerpted from the (VRML 97) specification in Figure 3.

```
Script {
  exposedField MFString url           []
  field        SFBool   directOutput  FALSE
  field        SFBool   mustEvaluate  FALSE
  # And any number of:
  eventIn      eventType eventName
  field        fieldType fieldName initialValue
  eventOut     eventType eventName
}
```

Script node is used to program behavior in a scene. Script nodes typically

a. signify a change or user action;
b. receive events from other nodes;
c. contain a program module that performs some computation;
d. effect change somewhere else in the scene by sending events.

Figure 3. Script node specification (VRML 97).

Events. Events allow VRML scenes to be dynamic. Events are merely time-stamped values passed to and from different parts of a VRML world. EventIns accept events, and EventOuts send events (when triggered by some predefined behavior). Events must strictly match the simple type (such as integer, float, color) or node type (such as a Material node) being passed from input to output. Script parameters are designated as eventIn, eventOut or exposedField, which respectively correspond to in, out or in/out parameter semantics. Private fields are simply designated as field rather than exposedField.

DEF/USE naming conventions. Node naming and light-weight multiple instancing of nodes is possible through the DEF (define) and USE mechanisms. DEF is used to associate names with nodes. USE permits duplicate instances of nodes to be efficiently referenced without complete reinstantiation, significantly boosting performance. Node names created via DEF are also used for routing events to and from fields. Thus Script nodes (and other 3D nodes which interact with the script) all must be named using DEF. The scope of all DEF'ed names is kept local to the file (or PROTO) where the name is defined.

ROUTEs. ROUTE statements define connections between named nodes and fields, allowing events to pass from source to target. ROUTE statements usually appear at the end of a file since all nodes must be DEF'ed (named) prior to referencing. Typically ROUTEs are used for all events passed into (or out of) Script nodes. Use of ROUTEs is not always required, however, since nodes in the VRML scene can be passed by reference as fields to the encapsulated Script code. This second approach permits direct manipulation of VRML by Java without using events or ROUTEs.

Example: event-based control. A scene demonstrating VRML-Java connectivity using Scripts, events, node naming and ROUTEs is examined in Figure 4. This VRML scene and corresponding Java source code are available at *www.stl.nps.navy.mil/~brutzman/vrml/examples/course/ScriptNodeEventOutControl.wrl* and ***ScriptNodeEventOutControl.java***

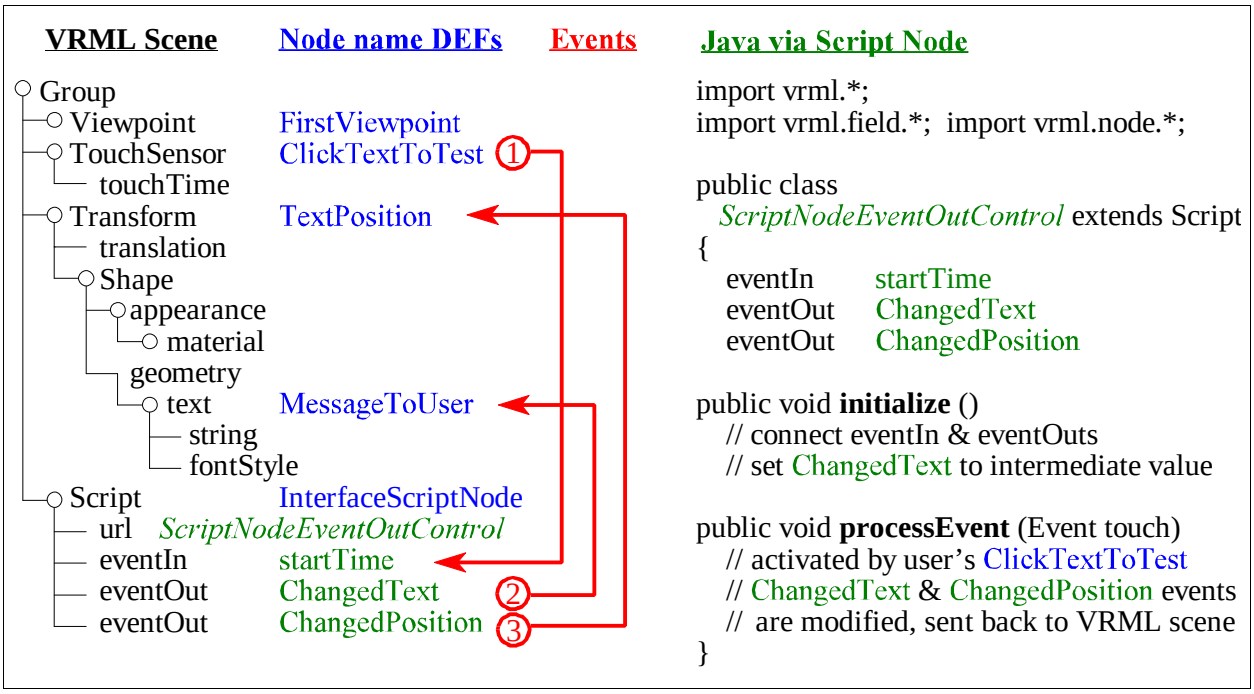

Figure 4. Script node interface between VRML and Java. This example tests event-based VRML-Java functionality. Note shared events startTime, ChangedText and ChangedPosition. The following sequence of events occurs:

(0) Initialize method on Java side establishes links, sets trace text in 3D scene to intermediate value.

(1) User clicks trace text in 3D scene with mouse, activating the TouchSensor built-in eventOut touchTime, which is ROUTEd to trigger the Script node EventIn startTime, which in turn invokes the processEvent() method in the corresponding Java class. Changed values for text and position are calculated by the Java class and then returned to the Script node as eventOut values.

(2) ChangedText eventOut sent to MessageToUser text node, sets trace text in 3D scene to final message value.

(3) ChangedPosition eventOut sent to TextPosition Transform node, moving trace text to bottom of scene.

Example: field-based control. An alternative to event passing via ROUTEs is to pass references to VRML nodes as values for fields in the Script node. In effect, Java gains direct control of VRML nodes and fields, rather than sending or receiving event messages to set/get values. Upon initialization, the Java class instantiates the node reference as a local variable. During subsequent invocations the Java class can read or modify any referenced field in the scene graph directly, without using ROUTEs. A second example follows which demonstrates the exact same functionality as the preceding example, but uses field-based control instead of events and ROUTEs. Figure 5 shows how nodes in the VRML scene are first defined, then passed as parameters to the Java class. The field-based example is available via *www.stl.nps.navy.mil/~brutzman/vrml/examples/course/ScriptNodeFieldControl.wrl* and *ScriptNodeFieldControl.java*

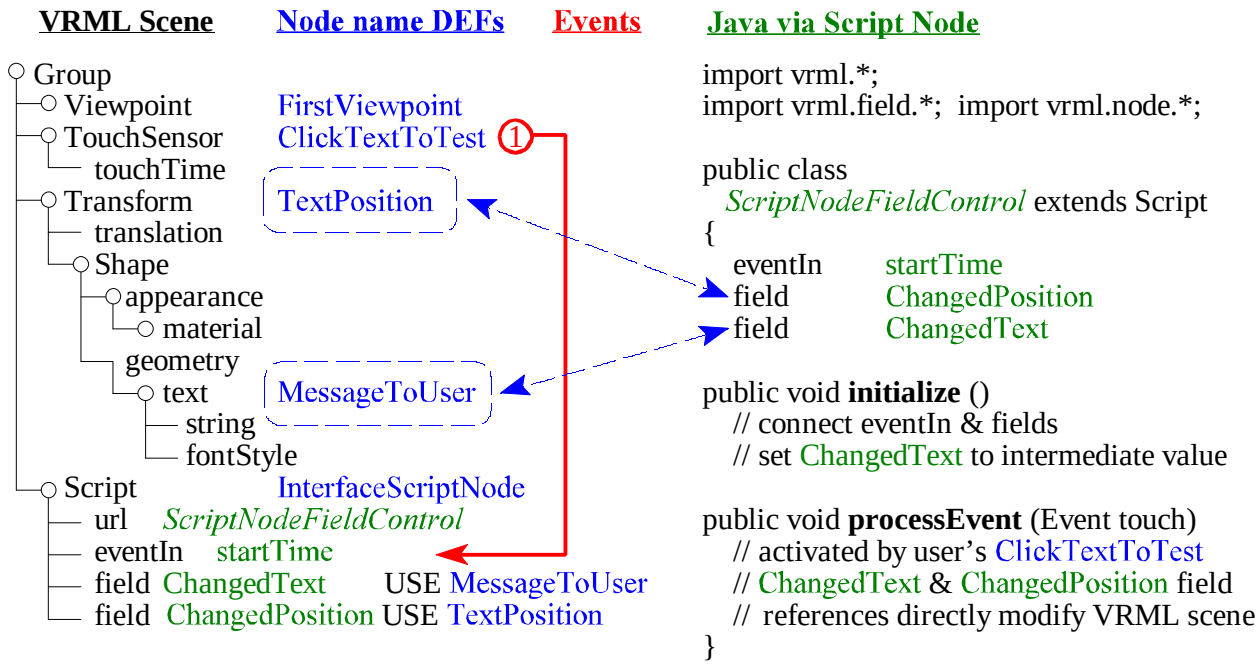

Figure 5. Field interface between VRML and Java. This example tests field-based VRML-Java functionality. Note shared event startTime, and shared fields ChangedText and ChangedPosition. Operation of this example is similar to Figure 4, except that the Java class directly manipulates VRML nodes via fields instead of sending events.

Script interface performance hints. Two authoring hints are provided as fields in the Script node for potential browser optimization: mustEvaluate and directOutput. In the first example (event-based control), mustEvaluate is set to FALSE as an author hint allowing the browser to postpone event passing until convenient, in order to optimize rendering. Similarly, the author hint directOutput is set to FALSE since the script only passes events and doesn't modify VRML nodes directly. In the second example (field-based control), the opposite values are used. Since values in the scene graph might be modified by the script directly (i.e. without notifying the browser via ROUTE activity), the hint field mustEvaluate is set to TRUE and the browser can't delay event passing as a performance optimization. Similarly, directOutput is set to TRUE to indicate that the script can modify VRML nodes directly via field control. If a scene uses both event and field control, the safest approach is to keep both values set to TRUE to maximize browser responsiveness to script actions.

Browser interface. Java via the Script node is provided a variety of methods to interact with the host Web browser. getName and getVersion provide browser identification information. getWorldURL provides a string containing the original URL for the top-level VRML scene. setDescription resets the top-level page heading. getCurrentSpeed and getCurrentFrameRate show user navigation and window redraw speeds. An author's Java program can also create and insert VRML source code (including nodes, PROTOs and additional ROUTEs) at run time. Java modifies VRML in the scene using the replaceWorld, createVrmlFromString, createVrmlFromURL, addRoute, deleteRoute, and loadURL methods. Valuable examples demonstrating these techniques appear throughout the public-domain JVerge class libraries, which provide a complete Java API mirroring all VRML nodes. JVerge accomplishes scene graph changes by sending modifications through the browser interface (Couch 97) (Roehl 97).

Future language interfaces. Java via VRML's Script node is well specified and multiple compliant browsers exist. Other interfaces are also on the horizon which can further extend Java-VRML functionality. Details follow.

External Authoring Interface (EAI). Rather than provide Java connectivity from "inside" the VRML scene via the Script node, the EAI defines a Java or Javascript interface for external applets which communicate from an "external" HTML web browser (Marrin 97). EAI applets can pass messages to and from VRML scenes embedded in an HTML page. Much of the browser interface is similar but somewhat different semantics and syntax are necessary

for event passing and flow of control. The primary benefit of the EAI is the ability for direct communications between the encapsulating HTML browser and the embedded VRML browser. The EAI will likely be proposed as an official extension to VRML 97. A next-generation Java-VRML working group is examining additional capabilities and possibly unifying the EAI with the Script node classes, for consideration in future versions of VRML.

Java3D. Sun has recently released the Java3D class library for 3D graphics programming (Deering 97). Java3D is an API, providing a programming interface for 3D that is analogous to the Abstract Window Toolkit (AWT) for 2D graphics. Java3D programs are saved as Java byte codes, not as a modeling format. Although Java3D is expected to include a loader capable of importing and exporting VRML geometry, it is not yet clear whether the Java3D event model will be able to similarly import and export VRML events. Public availability of the Java3D classes adds a number of new tools for Java programmers interested in VRML scene authoring. A VRML-Java3D working group is exploring usage conventions for interoperability, and also prototyping possible specification changes. Further unification of these already-complementary approaches holds tremendous promise.

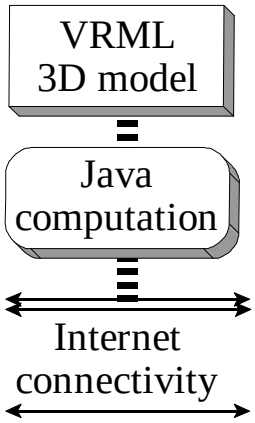

Figure 6. 3D, computation and network access provide a general entity model.

Example Java-VRML research. Programming Java in combination with VRML provides great expressive power. Simply illustrated in Figure 6, this combination of capabilities appears sufficient to provide a general entity model. VRML provides the 3D rendering and dynamic interaction capabilities, while Java provides general computation capabilities and network access. A variety of example projects follow.

Multi-user server worlds. A number of research laboratories and commercial companies are producing networked games and shared worlds that utilize centralized servers to share data among multiple participants. This alternative approach can reliably scale to several hundred participants. An extensive example that builds such a world appears in (Roehl 97).

Networking. In order to scale to many simultaneous users, peer-to-peer interactions are necessary in addition to client-server query-response. As one example, the IEEE 1278.1 Distributed Interactive Simulation (DIS) protocol is a well-specified way to pass entity behavior such as position, orientation, collision, fire/detonate and other message types. Use of multicast networking enables scalable many-to-many communications, avoiding server bottlenecks. DIS is particularly effective at communicating physics information at interactive rates in real time. The dis-java-vrml working group is implementing a public-domain DIS library in Java for use in multiple-entity VRML worlds, available via www.stl.nps.navy.mil/dis-java-vrml

Physics of motion. Displaying realistic entity motion requires properly modeling the underlying physics which govern entity equations of motion. Much work has been done to in the graphics and robotics communities to produce kinematic (velocities only) and dynamic (forces, accelerations) models for many different types of entities. Such models typically range from three to six spatial degrees of freedom (x, y, z, roll, pitch and yaw). The NPS *Phoenix* autonomous underwater vehicle (AUV) hydrodynamics model provides a perhaps-worst-case example how computationally demanding physical responses can be calculated in real time (10 Hz or better on a Pentium processor). NPS AUV software is available at www.stl.nps.navy.mil/~auv. Eventually we expect that interface conventions will emerge and physics libraries for most entity types will be widely available. A particularly appealing feature of such an approach is that computational load is distributed evenly, with each entity responsible for its own physical response calculations. Conventions for kinematic human body animation are already under development by the h-anim working group at ece.uwaterloo.ca/~h-anim/.

Sound. Computational requirements for spatial audio rendering are also demanding. VRML provides a simple sound node which localizes sound clips with their corresponding geometry. Widespread work in streaming audio is beginning to provide relatively low-bandwidth protocols with adequate sound quality and scalability. Further work in spatialized audio is beginning to show that advanced techniques for aural rendering are becoming possible on desktop machines. Further information is available via the Sound in Interactive Environments (SIE) mailing list at *www.cs.nps.navy.mil/people/phd/storms*.

vrtp. Finally, as the demanding bandwidth and latency requirements of virtual environments begin to be exercised by VRML and Java, some client-server design assumptions of the HyperText Transfer Protocol (http) may no longer be valid. Users won't be satisfied with network mechanisms that break down after a few hundred players. A spectrum of functionality is needed on each desktop which includes client, server, peer-to-peer and network

monitoring. Our research group is building a Virtual Reality Transfer Protocol (vrtp) to better take advantage of available transport-layer functionality for VRML and overcome bottlenecks in http. Experimentation and quantitative evaluation are essential to develop the next-generation code needed for diverse inter-entity virtual environment communications.

Next steps. A great deal of implementation work is now in progress. The best news: VRML and Java are powerful software languages for 3D modeling, general computation and network access. They are well matched, well specified, openly available and portable to most platforms on the Internet. VRML scenes in combination with Java can serve as the building blocks of cyberspace. Building large-scale internetworked worlds now appears possible. Using VRML and Java, practical experience and continued success will move the field of virtual reality past speculative fiction and isolated islands of research onto desktops anywhere, creating the next-generation Web.

References

Ames, Andrea L., Nadeau, David R. and Moreland, John L., *VRML 2.0 Sourcebook*, second edition, John Wiley & Sons, New York, 1997. Information available via *https://www.wiley.com/compbooks/vrml2sbk/cover/cover.htm*

Carey, Rikk and Bell, Gavin, *Annotated VRML 2.0 Reference Manual*, Addison-Wesley, Reading Massachusetts, 1997. Available via *www.best.com/~rikk/Book/book.shtml*

Couch, Justin, *VermelGen*, software distribution, Virtual Light Company, December 1997. Available via *www.vlc.com.au/JVerge*

Deering, Michael and Sowizral, Henry, *Java3D Specification*, Version 1.0, Sun Microsystems Corporation, Palo Alto, California, August 1 1997. Available via *java.sun.com/products/java-media/3D/*

Harold, Elliotte Rusty, *Java Network Programming*, O'Reilly and Associates, Sebastopol California, 1997. Available via *www.ora.com/catalog/javanetwk* with software at *ftp://ftp.ora.com/published/oreilly/java/java.netprog*

Hartman, Jed and Wernecke, Josie, *VRML 2.0 Handbook*, Addison-Wesley, Reading Massachusetts, 1996.

Hughes, Merlin, Conrad, Shoffner, Michael and Winslow, Maria, *Java Network Programming*, Manning Publications, Greenwich England, 1997. Available via *www.browsebooks.com/Hughes* with software available on CD-ROM (due to cryptographic software export restrictions).

Lea, Rodger, Matsuda, Kouichi and Miyashita, Ken, *Java for 3D and VRML Worlds*, New Riders Publishing, Indianapolis Indiana, 1996.

Marrin, Chris, *External Authoring Interface (EAI) Proposal*, Silicon Graphics Inc., Mountain View California, 1997. Available via *www.vrml.org*

Roehl, Bernie, Couch, Justin, Reed-Ballreich, Cindy, Rohaly, Tim and Brown, Geoff, *Late Night VRML 2.0 with Java*, Ziff-Davis Press, MacMillan Publishing, Emeryville California, 1997. Information available via *ece.uwaterloo.ca/~broehl/vrml/lnvj*

San Diego Supercomputing Center (SDSC), *VRML Repository*, 1997, available via *www.sdsc.edu/vrml*

VRML Consortium (VRMLC), working groups and other information, 1997, available via *www.vrml.org*

VRML 97, International Specification ISO/IEC IS 14772-1, December 1997, available via *www.vrml.org*

About the author. Don Brutzman (brutzman@nps.navy.mil) is an assistant professor at the Naval Postgraduate School in Monterey California. He serves as technology vice president for the VRML Consortium.