VRML Testing: Making VRML Worlds Look the Same Everywhere

Mary Brady, Alden Dima, Len Gebase, Michael Kass, Carmelo Montanez-Rivera, and Lynne Rosenthal
National Institute of Standards and Technology

At the National Institute of Standards and Technology (NIST), information technology focuses on developing freely available diagnostic tools and tests for building robust, interoperable commercial solutions. Applying such tools early in the life cycle process helps industry determine whether its products conform to the specification and, ultimately, will interoperate with other products. In addition, the development and use of these metrology tools fosters thorough review of the specification, resolving ambiguities and errors.

During Siggraph 96, NIST staff met with interested members of the VRML community to discuss various approaches to testing the Virtual Reality Modeling Language (VRML) specification. The standard defines requirements for creating VRML worlds, tools that generate VRML worlds (authoring tools), and tools that interpret and properly render VRML worlds (browsers). Participants at this meeting agreed unanimously that VRML worlds, whether generated by hand or through an authoring tool, must be fully compliant with the standard. Furthermore, they must be viewable and reasonably similar in a variety of VRML browsers, regardless of the underlying hardware and software platforms.

Participants acknowledged that interoperability among VRML files, authoring tools, and browsers was best achieved through a combination of approaches. Due to the interactive, internetworked, 3D nature of VRML, the testing methodology had to address VRML's ability to represent and properly render or capture multimedia-based content, network accessible links to reusable content, and static or animated scenes. A complete VRML test system needed to address these requirements for VRML content, browsers, and authoring tools in a reliable and intuitive manner.

Consequently, NIST developed metrology tools to support testing VRML content, authoring tools, and browsers. VRML content and the associated authoring tools are tested using a locally developed reference parser, Viper. VRML browsers are tested using a test suite of conformant files, called the VRML Test Suite (VTS). The VTS tests the VRML built-in nodes, VRML extensible components, and base execution model. Finally, the true dynamic nature of VRML is tested using automatic test generation techniques built through extension of the Viper source code. In this article, we also address using the Web as a vehicle for delivering these metrology tools.

NIST tools address problems posed by testing 3D graphics. This article explains the test development strategy and design issues in developing and delivering these testing tools.

Testing methodologies

Depending on the type of specification under test, you can apply various methodologies in developing conformance tests. The major portion of the VRML specification is written as a metafile language description. Similar to programming language standards, the specification captures both the syntax and semantics of the language. This type of metafile language specification lends itself to testing in three areas, including the file itself (the VRML world), the authoring tool, and the browser.

Metafile testing begins with syntax testing. It's best accomplished through introducing various types and levels of errors. Instead of pursuing this type of test case generation, we decided to expend our efforts in building and releasing to the VRML community a public domain parser, Viper. In some respects, this reference parser follows up the parser released by Silicon Graphics (VRML 2.0 Parser, http://vrml.sgi.com/). Using this reference parser, content providers can check the validity of their VRML file, whether generated by hand or with an authoring tool. In addition, browser and authoring tool developers can use the reference parser as a starting point for their implementations.

Browser testing lends itself to a type of conformance testing known as falsification testing. Falsification test-

0272-1716/99/$10.00 © 1999 IEEE

```
java -mx16m Viper -s -o -f C:\CGA\
MULTICOLOR_INDEXEDFACESET_VERTICES.WRL

World
|
+--Viewpoint
|
+--Viewpoint
|
+--Viewpoint
|
+--Viewpoint
|
+--Viewpoint
|
+--Viewpoint
|
+--NavigationInfo
|
+--Shape
    |
    +--Appearance
    |   |
    |   +--Material
    |
    +--IndexedFaceSet
        |
        +--Coordinate
        |
        +--Color
```

1 Viper scene graph.

ing gives confidence that an implementation has the required capabilities, implemented correctly. The objective is to produce tests for as many of the standard's requirements as feasible and use these tests to find errors in the implementation. Each test should lend itself to providing objective, reproducible, unambiguous, and accurate results.

In VRML, the browser must be able to read syntactically correct files and render them as specified by the minimum conformance requirements of the VRML 97 standard. The test files check a browser's abilities to produce correct observable behaviors as permitted by the standard. If the observed behavior matches the foreseen test outcome (the expected behavior), the implementation "passes" the test. Failure to pass all the tests implies failure to conform. However, passing all the tests provides no guarantee that a browser really does conform, as it might well violate the standard in untestable areas.

Falsification testing effectively tests the VRML built-in nodes, extensibility mechanisms, base execution model, and scripting language interfaces. However, some VRML constructs, such as pointing device sensors, do not immediately generate graphical output. Rather, these constructs set an internal state that will apply later, when the user points to the geometry influenced by a specific pointing device. Thus, some of the VRML language constructs have effects that are either indirect or inaccessible to the test program. The interactive, dynamic aspects of the language are therefore tested using automatic test generation techniques and evaluated by inspecting the "state" of the VRML browser.

Viper

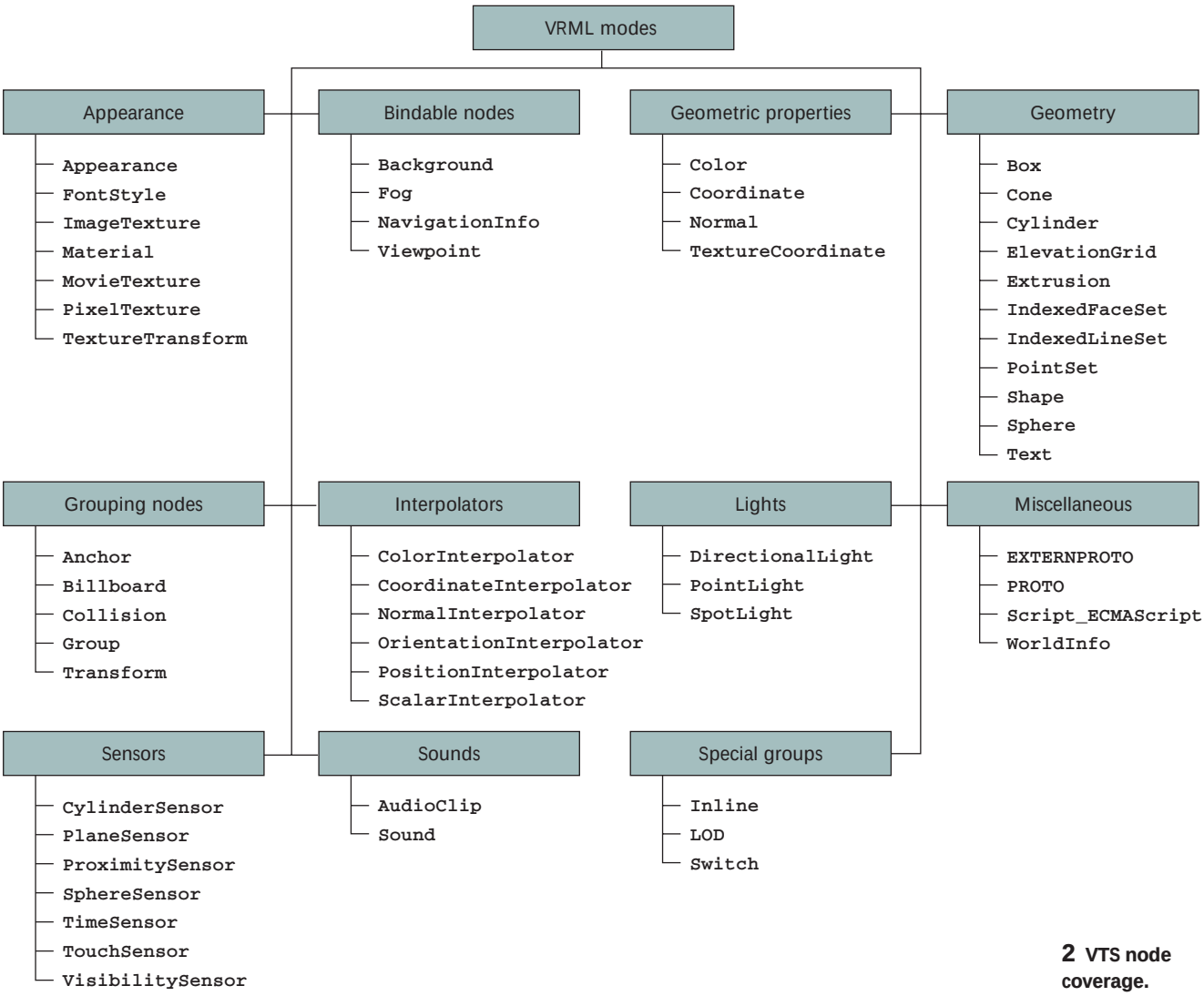
At NIST, we developed the Viper VRML parser using the object-oriented language Java[1] and the Java parser generator tool, JavaCC (Java Compiler Compiler, http://www.suntest.com/JavaCC/). This tool acts as a counterpart to the more traditional parser generator tool, yacc.[2] Like yacc, JavaCC accepts a BNF-like (Backus-Naur Form) grammar specification as input. Whereas yacc produces C code as output, JavaCC produces Java code as output. Java code fragments are embedded within the grammar rules. JavaCC also provides a convenient mechanism for passing objects to grammar rules and for allowing a rule to return an object. The JavaCC generated code is used to read VRML files, perform syntax analysis, and build a parse tree that captures the file contents. An optional feature permits checking the VRML file against the file limits specified in the VRML specification's minimum conformance requirements. This option will ensure that any VRML compliant browser can display the VRML file.

After Viper generates the parse tree, it constructs a scene graph, modeled as a directed acyclic graph, and performs a variety of semantic checks. During this phase, Viper casts all VRML constructs as valid VRML types and performs field constraint checks. **`DEF`**/**`USE`** name resolution and associated name scoping rules are applied. Hierarchy checks ensure proper placement of VRML nodes within the VRML file, and route semantics ensure that eventOuts are routed to eventIns and that the associated data types match.

Figure 1 illustrates example output from the Viper parser. This test world offers several **`Viewpoints`**, **`NavigationInfo`**, an **`IndexedFaceSet`** with **`Color`** and **`Coordinate`** properties, and a **`Material`** node. This output gives the user a quick synopsis of the file's structure.

At the same time that Viper was being developed, Trapezium Development Corporation developed a Java-based VRML syntax checker, Vorlon (http://www.trapezium.com/). In their current releases, each of these resources has similar base capabilities. In addition, Viper has options for generating a scene graph, and Vorlon has options for inspecting polygons and provides support for the H-Anim work. Viper is freely available, including source code; Vorlon, although freely available, has some licensing restrictions.

The Interoperability/Conformance Working Group considers both parsers viable options for determining whether a VRML world is VRML 97 compliant. It has provided links to each from its working group page, http://www.vrml.org/WorkingGroups/vrml-conf/. In addition, you can execute both over the Web through an interactive fill-out form. This form prompts you for various options, runs the indicated file through your parser of choice, and reports the results (http://autumn.ncsl.nist.gov/vrml-conf/online-parsers.cgi).



2 VTS node coverage.

The VTS method

In the VTS system, we design conformance tests for VRML browsers using concepts derived from falsification testing, realized using the same principles as in the mathematical axiomatic method.[3] We start by defining a set of axioms that succinctly describe the requirements of a conforming VRML implementation. Using this set of axioms, we develop a set of tests that will exercise each of these requirements. These tests, when loaded into a VRML browser, will generate a pass/fail condition. Failure to pass a test implies failure to conform.

Applying this technique to information processing standards poses the problem of translating a document written in English prose to a set of axioms. This translation process is accomplished through the creation of semantic requirements (SRs), which play the role of axioms. The SRs, in turn, serve to generate actual test cases (TCs); the TC results play the role of conclusions. Cuguni first discussed this approach,[4] which we discuss here as it applies to the VTS. In the following sections, we provide some insights from our experience in developing the VTS.

The VRML specification

The VRML 97 specification provides the definitive source for VRML behavior. Within this specification, sections exist for VRML concepts, node references, field and event references, and minimum support requirements. To properly capture browser behavior, the test designer must pull information from all of these sections. The organization of the VTS system follows the VRML specification. Individual VRML nodes, as defined in the VRML specification,[5] are grouped within a node category, as in Figure 2.

This organization provides a natural grouping, where each category deals primarily with the strongly related requirements of a topic. Each of the node categories links directly to a topic within the Concepts section of the VRML 97 specification, and each of the nodes is defined within the Node Reference section of the specification. As part of the test system, these natural links enabled us to provide Hypertext Markup Language (HTML) links from each of the node categories and all of the nodes to the appropriate section of the specification.

Semantic Requirements

Basic Functionality, applying colors to geometry faces and vertices:

1. <Color> nodes apply one or more colors to faces of an <IndexedFaceSet> and <ElevationGrid>, polylines of an <IndexedLineSet>, and points of a <PointSet>.

2. <Color nodes apply one or more colors to vertices of an <IndexedFaceSet>, <ElevationGrid>, and <IndexedLineSet>.

Color supercedes diffuseColor, texture supercedes diffuseColor and geometry color:

3. If a <color> field of a geometry contains a <Color> node, that <Color> node takes precedence over a previously defined <diffuseColor> field in the associated <Material> node for that geometry.

4. The <texture> field of a <Material> node takes precedence over the <color> field of an <IndexedFaceSet> and <ElevationGrid> node.

5. …

3 SRs from the `Color` node.

Test color interpolation: Apply multiple colors to vertices of all geometry types:

12. **Multiple colors applied to IndexedFaceSet vertices** — A test of a browser's ability to interpolate colored geometry vertices in the IndexedFaceSet node. Test should generate a cube with a red front face, a green rear face, and all other face colors interpolated between the red and green faces.

13.…

4 `Color` node TCs.

Semantic requirements

Each node contains a set of semantic requirements. These semantic requirements attempt to capture behavior known to be true for use as an axiom in our logic-based system. Well-designed SRs should be

- *Independent*. If one SR implies another, we keep only the stronger of the two, since the other is redundant.
- *Complete*. The SRs should require everything that the standard requires.
- *Consistent*. The SRs should not contradict each other.
- *Specific*. Each SR must have some testable consequence, perhaps in conjunction with other SRs. Broad generalities are avoided in favor of lower level concrete assertions.
- *Simple*. An SR should not be a long list of requirements; to a reasonable extent, each SR should state an atomic rule about conforming implementations.

Figure 3 shows an example of SRs for the `Color` node. It's important to describe not only the behavior of the node itself, but also any interactions with other nodes. In the case of the `Color` node, SRs are defined for applying colors to geometry faces and vertices, interactions with material `diffuseColor` and textures, color interpolation, applying colors to geometries, and minimum conformance requirements.

Even given these guidelines, no magic formula exists for specifying SRs for a given node. However, these SRs do provide a clear indication of the requirements for specifying conformance to the standard. The level of understanding evident from specifying SRs sharpens any interpretation questions that emerge. These cases serve as feedback to the standardization process, permitting correction of inconsistent or incomplete specifications in the standard.

Test cases

An individual test case (TC) is a testable conclusion derived from one or more SRs. Each TC should be written so as to explicitly state the behavior of a conforming implementation. In the VTS, these TCs are realized as actual test worlds. Loading the test world into an implementation will generate a pass/fail condition. The result of this process helps determine conformance to the specification.

Typically, a single SR does not lend itself to a single testable conclusion; rather, there exists a many-to-many relationship between SRs and TCs; that is, one TC may be derived from many SRs, and each SR may be used to specify several TCs.

Defining a complete chain of inference from a particular TC back to the standard involves indicating, as a comment within the test world itself, which SRs are currently under test. In turn, the user can determine by inspection that each SR is well grounded in the specification. This explicit mapping of TCs back to SRs exhibits the validity of the TC. As previously stated, passing all of the TCs does not prove that an implementation conforms to the specification. Failure in a particular TC does strictly imply failure to conform.

Figure 4 shows an example of test cases defined for the `Color` node. For the `Color` node we defined 21 test cases, which covered applying single and multiple colors to geometry faces, lines, points, and vertices, preference of geometry color over `Material diffuseColor` field, preference of `texture` field to geometry color field, and minimum conformance requirements.

This approach is employed in testing the VRML built-in nodes, the extensions, and sensors and scripts. In

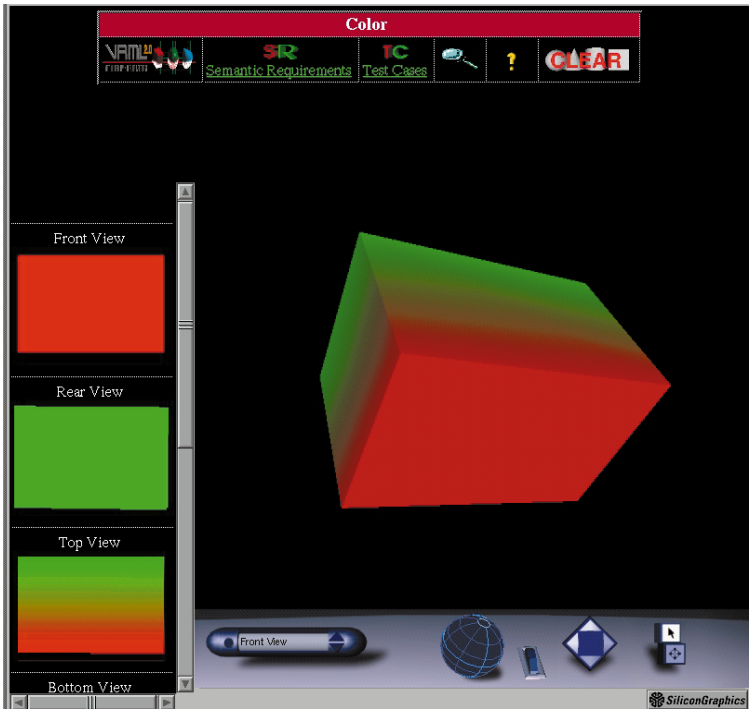
each case, we initially created tests to cover all functional conditions. Each field is exercised within a node, and where the specification indicates, all combinations with fields within other nodes are also exercised. We quickly determined that testing the various combinations was supremely important in ensuring that the interactions between nodes were indeed properly defined. For instance, we found that to obtain proper coverage of the lighting model, we needed to define tests that not only individually tested various components, but also tests that demonstrated combinations for unlit, gray-scale, rgb, and rgb+alpha properties applied to all geometries.

We also created tests for all boundary conditions for a given field and for a third category that we named "divergent" conditions. As we began to exercise each node in the specification, we quite often uncovered conditions where various implementations diverged. When this occurred, we explicitly created tests that pointed out the differences among implementations. In most circumstances, this divergence resulted from an ambiguity or unspecified case in the specification. These tests served to raise awareness of these issues within the community. Moreover, they often lead to clarifications in the specification.

The extensibility mechanisms within VRML, including **PROTO**s and **EXTERNPROTO**s, are tested similarly. In these cases, we gave extra attention to defining tests that would exercise the calling interface, namespace scoping, and nested conditions. To properly exercise the various **PROTO** and **EXTERNPROTO** constructs, we also had to presuppose that the VRML built-in nodes worked properly. For instance, to test that an interface definition supports all valid VRML field types, we had to generate a matching eventOut for each VRML field type.

We also test VRML Sensors and Scripts in this manner with a few additional constructs. In the case of Sensors, base functionality is exercised; true dynamic testing—setting off a series of events—is reserved for dynamic testing. VRML Scripts are tested by testing the language interface. Appropriate Java or ECMAScript code is generated along with VRML code to test this interface. Here again, we had to presuppose that the VRML built-in nodes worked properly and that the language interface remained to be tested.

Still the dynamic components of VRML have yet to be thoroughly exercised. Although we covered base functionality using the VTS tests, we still had to test the language's interactive, dynamic nature. We further discuss a novel approach to testing these components in "Testing dynamic VRML," below.



5 Running a `Color` node test in the VTS. Here, we apply multiple colors to `IndexedFaceSet` vertices.

Expected results

In past testing efforts, we have had a mutually agreed-upon reference implementation to definitively capture proper behavior. In the case of the VTS, the VRML community faced an immediate need for conformance test cases. In the interest of time, we agreed to use whatever browser worked best for a given test case to display and capture a graphical representation of the test results. Results were to be captured as JPEG (Joint Photographic Experts Group) images for tests involving static scenes and as MPEG (Moving Pictures Experts Group) movies for tests involving animated scenes. In each case we retained links to the original VRML scenes.

Figure 5 provides an example of running one of the **Color** node tests in the VTS. This test determines a browser's ability to interpolate colored geometry vertices in the **IndexedFaceSet** node. The banner contains links to the **Color** node reference within the VRML specification, the semantic requirements, an individualized list of test cases, the ability to view the VRML source, and online help. The clear button takes the user back to a list of nodes within this category and an itemized list of the **Color** node tests. Along the left-hand side, orthogonal views of the expected results for this test case appear. The remaining portion of the screen remains available for viewing. Here, it shows the result of loading this test case into the currently installed VRML browser. This layout provides the ability to view a side-by-side comparison of the test case and the expected results. In addition, other resources related to this test are viewable with the click of a button.

As we began experimenting with various browsers and experienced first hand the vast differences in individual test cases, we sometimes felt uncomfortable suggesting a particular browser for a given test case. These

Color and Shading Fidelity

Maureen C. Stone
StoneSoup Consulting

One of the greatest challenges in making VRML worlds "look the same" everywhere is color and shading fidelity. Even the same code running on identical systems can produce visually different results due to variations in the brightness, contrast, and color temperature control settings on the monitor. Providing the same appearance across different hardware and software systems proves even more difficult. This problem—common to all color imaging applications—is the prime motivation for color management systems and device independent color standards.[1] Color fidelity is also a prime concern in the broadcast television and video industry.[2]

In the case of 3D rendering systems such as VRML, an additional source of variation lies in the implementation of the lighting and shading model. An ideal system performs the lighting calculations for each pixel. To provide adequate performance for desktop systems, however, requires approximating this ideal. Geometry in the scene is decomposed into triangles. The lighting calculations are performed at the vertex of each triangle, then the color is interpolated across its face. Both the triangulation and interpolation algorithms strongly affect the final result. Typically, these algorithms are implemented by rendering packages such as Microsoft's Direct3D (www.microsoft.com/directx/) or the public domain OpenGL (www.opengl.org), not by the VRML browser.

The VRML 97 specification describes color simply as RGB triples. Without further information, this is inadequate to calibrate VRML colors to a device-independent standard. The Color Fidelity Working Group, a subgroup of the Conformance and Interoperability Working Group (www.vrml.org/WorkingGroups/vrml-conf/), has informally proposed a way to supply the missing information. This would complete the specification and enable evaluating VRML color fidelity for ideal systems. This would also enable its integration with color management systems and color imaging standards.

To calibrate a color space specified as RGB triples, we need to construct a mapping between numeric values in the space and perceived colors. The perceived colors are usually specified as CIE (www.cie.co.at/cie/) tristimulus values, an internationally recognized method for specifying color. If we know the color of the red, green, and blue primaries, plus the intensity transfer function (ITF) that maps between numeric values and perceived brightness for each primary, we can compute the tristimulus value for any color in the space. The RGB color space of a typical computer monitor, for example, has primaries defined by the color of the RGB phosphors, typically similar to those specified by ITU-R BT 709 (www.itu.ch/publications/index.html). The transfer function typically takes the form $I = V^{\gamma}$ and is often called a gamma function.

The VRML specification must address color in three places: input colors, color in the rendering equation, and displayed color. Input colors are specified directly as RGB triples or as textures imported in some standard color image format. The RGB color values in the rendering equation represent intensity values, which means the color space has a linear ITF. Displayed color results from mapping the rendered colors to the specific display available on the user's computer system. Typically this is a color monitor, although flat, liquid-crystal displays (LCDs) are becoming more common.

To complete the specification of color in VRML, we need to define the primaries and the ITF for the colors in the file, the primaries for the rendering space, and the mapping from the rendering space to the viewer's display. There is an obvious choice for the primaries—the ITU-R BT 709 specification used for television and many color image formats. The current proposal for the file format ITF is linear, to match the rendering space. After much debate, the current proposal for the rendering-to-display mapping is the nearly linear function used in television.

Implementing this specification is a nontrivial performance issue. The major change for input is to correctly implement imported image textures. Most image formats have an ITF that is the inverse of the expected display ITF. That is, if you assume your display system ITF is a gamma function, then the pixel values in the image represent linear intensity values raised to the $1/\gamma$ power. This provides optimum display performance for pure imaging applications, as no transformations need to be performed to display the original values. For VRML, however, the image colors must be linearized during rendering to correctly implement the standard.

The more significant issue with respect to performance, however, is the need to map the pixel colors on output. The color values output by the renderer are linear values, but most display systems have a nonlinear ITF, requiring at minimum a table lookup per primary per pixel color. The emergence of LCDs creates a more substantial problem. The ITF for these systems is nearly linear, but the primaries vary significantly among displays. To transform from one set of primaries to another requires a 3×3 matrix multiplication per pixel color.

Adding calibration information to the VRML 97 specification will create a precise definition of color and shading fidelity. It seems unrealistic to expect current implementations to conform to this ideal. However, the trend towards making graphics and color imaging ubiquitous, especially as driven by the World Wide Web, should push hardware and system support for device-independent color. Once this is in place, making worlds "look the same everywhere" will become possible.

References

1. E.J. Giorgianni and T.E. Madden, *Digital Color Management: Encoding Solutions*, Addison Wesley, Reading, Mass., 1998.
2. Charles A. Poynton, *A Technical Introduction to Digital Video*, John Wiley & Sons, New York, 1996.

Table 1. Color node results.

Test	Test Requirement	Cosmo Player 2.0, Win95	Cosmo Player 1.0.2, SGI Irix 6.3	Worldview 2.1b, Win95
1	Single-colored `IndexedFaceSet` faces	Yes	Yes	Yes
2	Single-colored `IndexedLineSet` polylines	Yes	Yes	Yes
3	Single-colored `PointSet` geometry	Yes	Yes	Yes
4	Single-colored `ElevationGrid` faces	Yes	Yes	Yes
…	…	…	…	…

differences could be attributed to the differences in the underlying rendering software or the particular hardware the browser was built upon. They also reflect the problem of specifying and maintaining color and shading fidelity (described more fully in the sidebar). In spite of the difficulties involved, we believed it necessary to provide the community with some indication of "correct" behavior. In an effort to get the VRML community to reach consensus, we captured results for each of these tests and fed them back to the browser companies. In addition, after the browser companies had an opportunity to review the results, we made them available from the Interoperability/Conformance Working Group page, underl.org/WorkingGroups/vrml-conf/.

In Table 1, each test requirement is a concise definition of the test and a hyperlink that will take the user directly into the VTS system, load the test in question, and provide a side-by-side comparison of expected results and actual results. Questions can be further investigated from this point. This reporting mechanism has provided browser developers and content developers alike with a concise overview of various browsers' capabilities.

The VTS system attempts to cover the entire specification, exercising all features and feasible combinations of features. Thus far, the three browsers that we have tested have successfully passed more than 90 percent of our tests. As we developed these tests and shared information with the browser developers, many bugs were uncovered and subsequently fixed. With each new release, the browsers come closer together and, as such, closer to full compliance with the VRML 97 specification. Still, a number of issues have yet to be resolved. In the 10 percent of tests that browsers failed, 27 nodes were represented. In some cases, only one browser failed to conform; in others, all browsers failed to conform.

The VTS Web

The World Wide Web seemed a natural choice as a medium for combining all of the previously described design features into an interactive VRML testing system. Since many users are already Web-literate, setting up a Web server to deliver testing capabilities and supporting documentation to clients proves an inexpensive and effective method. In effect, anyone with a Web client configured with a VRML 97 plug-in can test that browser's conformance to the specification. For those companies who develop stand-alone VRML browsers, the VTS is also available as a downloadable archive.

Another feature that makes the Web particularly useful as a delivery medium is the publication of the VRML 97 specification as an HTML document. Indexing the document for text searching, as well as linking directly to it as a reference, help create a powerful tool for mapping the specification to semantic requirements, as well as the actual test cases themselves.

In addition, the user-friendly nature of Web applications, the immediate feedback possible through e-mail, and access to the VRML community through already established mailing lists provided an excellent environment to share testing information.

The application itself consists of standard HTML pages, with PERL (Practical Extraction and Report Language) applications providing program logic. The VTS (http://www.nist.gov/vrml.html) is presented as a group of three HTML "frames" that together provide a single test environment for comparing a VRML 97 node's specification, requirements, actual test worlds, and expected test results in a single Web browser window.

Testing dynamic VRML

In this section, we describe an approach to automatic test-case generation based on expanding Viper by inverting the parse tree. Furthermore, emphasizing state behavior during dynamic testing augments test case evaluation. The implementation is treated as a finite-state machine, and we use state transitions to identify points in the implementation to evaluate.

Automatic test-case generation

The parser's utility in facilitating the development of a VRML testing tool stems from two factors. First, the parser already includes a number of classes that support manipulation of VRML objects. Second, the construction of the parse tree itself suggests a method for automatic test-case generation. Given a VRML test-case file, the parser reads the file, builds the parse tree, and checks for syntactic and semantic correctness. Reversing this process is the basis for automatic generation of test cases.

Inverting the work performed by the parser is the key concept used in turning the parser into a test-case generation tool. Dynamically constructing an internal representation of the parse tree produces a test case. Producing the parse tree requires modules for constructing and combining nodes. Scene graph construction entails enforcement of all node hierarchy rules and route semantics, and ensures the validity of all nodes. It therefore provides the foundation for ensuring that nodes are built correctly and the parse tree is con-

structed correctly. Augmenting these modules with a constraint enforcement module provides further control over the VRML test-case files generated. The constraint enforcement module defines and applies additional rules and conditions to the construction of the test-case files.

Generating dynamic test cases

VRML worlds that exhibit dynamic behavior typically include a number of `ROUTE` statements. `ROUTE`s connect nodes together and allow the exchange of events. In effect, `ROUTE`s provide a mechanism for wiring nodes together. Once the nodes have been wired, an event received by one node can result in subsequent events being received by other nodes. The receiving nodes can, in turn, transmit additional events that are received by still other nodes. In this way, a chain of events can be propagated that dynamically change the VRML world.

Automatic test-case generation helps reduce the difficulty of VRML conformance testing; automating test-case evaluation fully exploits the benefits of automatic test-case generation.

Producing VRML worlds that include `ROUTE` statements requires a scheme for generating routes. One scheme builds a matrix of potential node connections, thus viewing each of the standard VRML nodes as having both an input and an output port. Enumerating the input ports along one dimension and the output ports along the other forms a matrix. Each element in the matrix then indicates a potential connection between node types. Thus, the matrix will control which nodes can change and which cannot. A mechanism for setting the matrix elements' values will provide a means for determining which nodes are subject to change.

When a VRML world includes named nodes (named with the `DEF` construct), the matrix is checked to see if the node might potentially be used in a route construct. If yes, it's marked for later use. The creation of specific routes depends on the node's fields, which requires an additional mechanism for connecting the node's fields. We employ an approach similar to the one outlined for connecting nodes. When two nodes are identified to be connected, a matrix made from the one node's eventOut fields and the other's eventIn fields is constructed. Enforcement of route semantics permits enabling only points connecting fields of the same type.

Three options exist for determining which points from this set are enabled. One option enables the entire set; a second randomly enables points from the set; a third option uses the field grid to control fields subject to change. For example, only enabling connections to the rotation field of the transform node permits conducting tests that investigate behavior when rotation of geometric figures is allowed, but other changes are not.

Simulating browsers

VRML conformance testing is a difficult task, made still more difficult by dynamic testing. Automatic test-case generation helps reduce the difficulty of the task, but the potential difficulty in evaluating test cases mitigates the value of automatic test-case generation.

Automating the process of test-case evaluation fully exploits the benefits of automatic test-case generation. At this point, we have already tested a browser using the VTS system, and we know that the browser has attained a base level of conformity. For dynamic testing, we capture the browser's state through the VRML `Script` node. This information provides important data regarding the state of the browser, which can be used for evaluating behavior. We generate expected results for this behavior by building a browser simulation capability.

Extending the parser's existing capabilities lets us implement a browser simulation model. The NIST parser supports functionality typically provided by one of the browser's major functional components. To support simulation, we augment the parser with a set of objects to simulate VRML node behavior. We determine expected browser behavior by viewing the browser as a finite-state machine and noting that the VRML world, too, can be treated as a finite-state machine.

The state of a VRML world is determined by each of its component nodes. The world initially occupies some state, S_i. In general, when an event occurs, a state transition from state S_j to a new state, S_k (k may equal j), occurs. These states identify points where test case evaluation takes place. In particular, at each state transition the browser must maintain an accurate representation of the Scene Graph. The browser's Scene Graph is derived using the `Script` node. The simulation model tracks state transitions and maintains the correct Scene Graph. Since an incorrect Scene Graph indicates incorrect browser behavior, it provides a tool for detecting problems in the browser.

Unfortunately, the inverse isn't true: incorrect browser behavior does not imply an incorrect Scene Graph—problems may still exist in rendering the Scene Graph. It's nevertheless possible to automate dynamic testing, if we divide testing into two major components: state evaluation and rendering evaluation. Following this approach permits evaluating the browser's rendering of the Scene Graph during VTS testing, and its state and Scene Graph maintenance during dynamic testing.

Ideally, an implementation undergoes VTS testing to assess its conformance in properly rendering the Scene Graph. Once we determine that the implementation properly renders the Scene Graph, dynamic testing begins.

Conclusion

We used the NIST-developed metrology tools to thoroughly review and exercise the VRML specification, associated VRML worlds, and VRML browsers. We defined and executed a method for measuring the conformance of VRML worlds and browsers, increasing confidence that a given world will be rendered similarly across various hardware and software solutions. This increased level of interoperability among VRML products fosters a pleasant user experience and promotes VRML's ultimate success. ■

References

1. K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, Reading, Mass., 1996.
2. S.C. Johnson, *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, N.J., 1978.
3. P.J. Denning, J.B. Dennis, and J.E. Qualitz, *Machines, Languages, and Computation*, Prentice-Hall, Englewood Cliffs, N.J., 1978.
4. J.V. Cuguni, "Interactive Conformance Testing for PHIGS," *Eurographics 91*, F.H. Post and W. Berth, eds., Elsevier Science, New York, 1991.
5. International Organization for Standardization and International Electrotechnical Committee, *The Virtual Reality Modeling Language*, ISO/IEC IS 14772, 1998.

Mary Brady *is a computer scientist at the National Institute of Standards and Technology in Gaithersburg, Maryland. As project leader for the VRML Testing effort, she has directed the development of the VTS and Viper, and served as co-chair of the VRML Consortium's Interoperability/Conformance Working Group. She has a BS in computer science and mathematics from Mary Washington College, and an MS in computer science from George Washington University.*

Alden Dima *is a computer scientist at the National Institute of Standards and Technology in Gaithersburg, Maryland. He has a BEE from the Georgia Institute of Technology, an MS in electrical engineering from the University of Connecticut, and an MS in computer science from the George Washington University. Dima is a member of the IEEE and the IEEE Computer Society.*

Len Gebase *is a computer scientist at the National Institute of Standards and Technology in Gaithersburg, Maryland currently working in the area of software conformance testing. Presently, he leads the development of Viper, a VRML parser and testing tool. His previous work has included developing and testing data communications protocols. Gebase obtained his bachelor's degree from Rutgers University in mathematics, and he holds an MS in computer science from Johns Hopkins University.*

Michael Kass *is a computer scientist in the Software Diagnostics and Conformance Testing Division of the Information Technology Lab at the National Institute of Standards and Technology in Gaithersburg, Maryland. He currently develops tests for the NIST VRML Test Suite. He has a BS from the Ohio State University.*

Carmelo Montanez *is a computer scientist at the National Institute of Standards and Technology in Gaithersburg, Maryland. He received a BS in math and an AA in chemistry from the University of Puerto Rico. His main interest is in the conformace testing area. Currently Montanez is working with Viper, simulating browser behavior.*

Lynne Rosenthal *is the manager of the Conformance Testing Group at the National Institute of Standards and Technology's Information Technology Laboratory in Gaithersburg, Maryland. Responsible for developing software conformance test methods, she has developed conformance tests for standards including VRML, Computer Graphics Metafile (CGM), Programmer's Hierarchical Interactive Graphics System (PHIGS), and Initial Graphics Exchange Specification (IGES). She has been an active member of the International Standards Organization (ISO) and the American National Standards Institute (ANSI) Technical Committees on Computer Graphics, and is the ISO editor for CGM Amendment 1, Rules for Profiles.*

Readers may contact Brady at Information Technology Laboratory, National Institute of Standards and Technology, Gaithersberg, MD 20899, e-mail mbrady@nist.gov.