VRML for Visualization

James Earl Johnson

ABSTRACT: *VRML, the Virtual Reality Modeling Language, is heading for a browser near you. VRML promises a write once, view anywhere capability for visualizing the results of engineering and scientific calculations. The same results may be visualized on a multitude of platforms, locally or over the web, using familiar web browsers. The choice of platform determines the performance of the visualization while the format and functionality remains unchanged. Simulation results can be packaged in a VRML format file to be loaded by a browser. Alternatively, a Java applet can read an existing data format and inject the data into a running VRML browser.*

1 Introduction

According to authors of The VRML97 ISO specification[1], "Technically speaking, VRML is neither virtual reality nor a modeling language." Rather, they say, "At its core, VRML is simply a 3D interchange format.", and that "VRML is a 3D analog to HTML. This means that VRML serves as a simple, multiplatform language for publishing 3D Web pages."[2]

There are a multitude of 3D file formats. Why should I choose VRML for scientific and engineering needs?

The first reason is cost. VRML97 is a high-level language, akin to a scripting language or HTML, so it is possible to write an application quickly. A VRML application need only describe what should displayed. The VRML browser attends to the details of how to display the model on its host platform.

The second reason for choosing VRML is the availability of VRML browsers. Currently, VRML browsers are available on several popular platforms including the Apple Macintosh, the PC, and SGI. Furthermore, the leading HTML browser vendors have pledged to distribute VRML browsers along with their respective web browsers. This means that a model published in VRML may be readily viewed by a large segment of the population.

The remaining question is whether VRML will perform adequately for visualization tasks. It must provide the features necessary for visualization. It must execute the visualization with acceptable speed. Later in this paper, I will examine three applications to test the feasibility of using VRML for visualization.

I will first identity features of VRML that enable visualization, then I will examine methods to get adequate performance.

2 Representing the Model in VRML

2.1 VRML Basics

The VRML97 specification has its roots in OpenInventor™, which is widely used in visualization. The basic building construct in VRML97 is called a node. There are 54 node types defined in VRML97. Each node type has field parameters that customize an instance of the node type. A node type may also have input fields that can accept events to alter its state during display. The node may have output fields which distribute events generated by the node. Instances of nodes are grouped in an acyclic graph to represent a model. This is commonly called a scene graph.

The VRML97 node types may be categorized into several groups by their function and their legal placement within a scene graph.

Geometry nodes provide the drawing primitives that represent all the visible elements of the model. The VRML97 authors deemed a few geometric solid shapes worthy of a node type: Sphere, Cylinder, Box, and Cone.

Other general shapes and surfaces must be composed of the geometry primitives of points, lines, and faces. These primitives are defined by the geometry nodes: PointSet, IndexedLineSet, and IndexedFaceSet. These nodes need to reference a set of coordinates points. VRML97 places the coordinate points in a separate Coordinate node. The separation of geometry from the coordinates allows for easy animation of the geometry by allowing new coordinates to be substituted over time. It also allows multiple geometry nodes to reference the same set of coordinates points.

These primitive geometry nodes may also reference a Color node that contains an array of colors to be applied to the geom-

Copyright © 1998. Cray Research, A Silicon Graphics Company. All rights reserved.

etry. This is important for visualization, since color is often used to indicate the value of a computed field at each coordinate point.

Unfortunately, VRML97 does not define an IndexedPointSet node. This implies that a PointSet cannot reference a subset of points in a Coordinate node. This makes it more difficult to switch between representing a portion of the model as points instead of either lines or faces.

VRML97 requires each instance of a geometry node in a scene graph to be contained within a Shape node. The purpose of the Shape node is to optionally associate an Appearance node with the geometry. The Appearance node may provide material color properties and texture for the geometry of the Shape node.

The grouping nodes allow us to build the viewing and transformation hierarchy of a scene graph. They reference a set of child node instances, and they provide the tree structure for the scene graph. The Transform node allows us to scale, translate, and orient the underlying geometry. The Switch node allows us to selectively choose one of a set of nodes to display. The Inline node allows us to include VRML from another file. The Group node simply allows us to reference a set of nodes under a single node instance.

Interpolator nodes provide key frame animation capabilities for a model. They use an input event value to interpolate an output value from the key frame values.

Sensor nodes generate output events based upon the state of the scene graph. The sensors that respond to the cursor position, such as TouchSensor, are often used to construct VRML control widgets. The TimeSensor node generates events at specified time intervals, and it is frequently used to drive an animation.

The Script node is the extension mechanism of VRML97. Any functionality not provided by the other VRML97 node types can be supplied in a Script node instance by the model creator. A Script node instance can have an arbitrary number of defined fields, including event input and output fields. The model creator supplies the Script node instance with programming code to handle events. Browser vendors have not been required to support any particular coding languages for the Script node, although Java and ECMAScript are now recommended by the VRML Consortium.

A VRML model is event driven. Node instances may generate output events that can be routed to the input fields of other node instances. In order to specify event routing information for events, nodes instances need to be assigned a name. This is accomplished with the DEF keyword. Event routing between the output field of one node to the input field of another node is specified with the ROUTE TO keywords. The following VRML code illustrates the routing syntax:

```
DEF BUTTON TouchSensor{}
DEF CLOCK TimeSensor{}
ROUTE BUTTON.touchtTime TO CLOCK.startTime
```

The DEF keyword also provides another capability. An instance of a node that has been named by the DEF syntax can be referenced later by the USE keyword.

One last VRML facility will prove useful for us. This is the node prototype capability specified with the PROTO keyword. This allows us to compose a new author-defined node type from existing nodes.

2.2 Examples

I have taken three types of simulation results as examples to determine the usefulness of VRML for visualization needs: molecular dynamics, finite element analysis, and computational fluid dynamics.

The easiest VRML-based visualization to produce was a ball-stick model for visualizing the results from a molecular dynamics code. To represent an atom I needed a sphere of a specified radius and color that I could position throughout the animation.

I used the PROTO capability to characterize an atom in VRML as follows:

```
Switch { choice DEF AtomSphere Sphere {} }PROTO Atom [
exposedField SFColor color
  exposedField SFVec3f scale
  exposedField SFVec3f position
]
{
  Transform {
    translation IS position
    scale IS scale
    children [
      Shape {
        geometry USE AtomSphere
        appearance Appearance {
          material Material {
            diffuseColor IS color
          }
        }
      }
    ]
  }
}
```

Similarly, a bond is two cylinders positioned and oriented between atoms. The color of each cylinder matches the closest atom.

```
Switch { choice DEF BondCylinder Cylinder {} }
PROTO Atom [
  exposedField SFColor color
  exposedField SFVec3f scale
  exposedField SFVec3f position
  exposedField SFRotation orientation
]
{
  Transform {
    translation IS position
```

```
      rotation IS orientation
      scale IS scale
      children [
       Transform { translation 0 1 0 children
        Shape {
         geometry USE BondCylinder
         appearance Appearance {
          material Material {
           diffuseColor IS color1
          }
         }
        }
       }
       Transform { translation 0 -1 0 children
        Shape {
         geometry USE BondCylinder
         appearance Appearance {
          material Material {
           diffuseColor IS color2
          }
         }
        }
       }
      ]
     }
    }
```

Each atom has a PositionInterpolator node that contains an array of calculated positions for that atom during the animation. The output of the interpolator is sent to the atom transform using the VRML event routing mechanism.

Bonds require an OrientationInterpolator node as well as the PositionInterpolator.

A simple animation may look like:

```
DEF A1 Atom {}
DEF A2 Atom {}
DEF B Bond{}
DEF P1 PositionInterpolator {
 key [ 0 1]
 keyValue [ 0 0 0  1 0 0]
}
DEF P2 PositionInterpolator {
 key [ 0 1]
 keyValue [ 1 0 0  1 1 0]
}
DEF PB PositionInterpolator {
 key [ 0 1]
 keyValue [ .5 0 0  1 .5 0]
}
DEF OB OrientationInterpolator {
 key [ 0 1]
 keyValue [ 0 0 1  1.57   0 0 1  0]
}
ROUTE P1.value_changed TO A1.set_position
ROUTE P2.value_changed TO A2.set_position
ROUTE PB.value_changed TO B.set_position
ROUTE OB.value_changed TO B.set_orientation
```

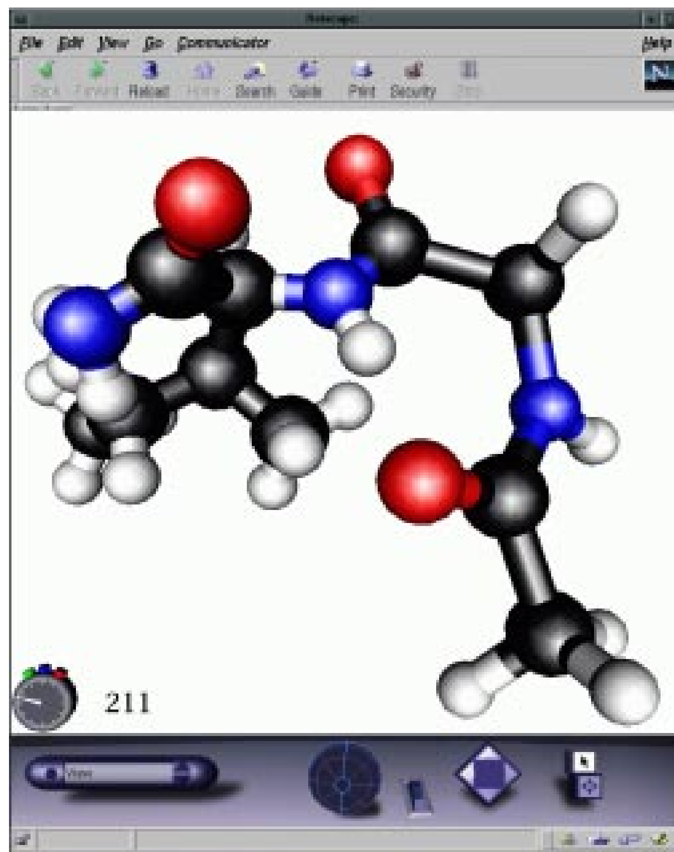

Figure 1: A Molecular Dynamics Simulation

The other two examples, computation fluid dynamics and finite element model, use meshes.

These often have several parts which one may want to selectively display. I created a PROTO for parts:

```
PROTO Part [
 exposedField MFVec3f translation 0 0 0
 exposedField MFRotation rotation 0 0 1 0
 exposedField MFVec3f scale 1 1 1
 exposedField SFInt32 mode 2
 exposedField SFNode coord NULL
 exposedField SFNode color NULL
 exposedField SFColor partColor 1 1 1
 exposedField SFFloat transparency 0
 field MFInt32 lineIndex []
 field MFInt32 faceIndex []
]
{
 Transform {
  translation IS translation
  rotation IS rotation
  scale IS scale
  children [
   Switch { whichChoice IS mode
    choice [
```

```
        Shape { geometry PointSet {
           coord IS coord  color IS color }
         appearance DEF App Appearance {
           material Material { emissiveColor IS partColor
             transparency IS transparency} }
        }
        Shape { geometry IndexedLineSet {
           coord IS coord  color IS color
           coordIndex IS lineIndex }
         appearance USE App
        }
        Shape { geometry IndexedFaceSet {
           coord IS coord  color IS color
           coordIndex IS faceIndex
           solid FALSE }
         appearance DEF App Appearance {
           material Material { diffuseColor IS partColor
             transparency IS transparency} }
        }
      ]
    }
   ]
  }
}
```

I use the switch to selectively hide the part, or to show it as coordinate points, a wireframe of line segments, or a surface of polygonal faces. By exposing the color and transparency fields of the Material node, I can adjust the overall appearance of parts. I can make parts semitransparent to expose internal geometries. I can differentiate parts by changing their overall color.

I like to place parts in Transform nodes so that I can refer to parts and slide them out of the way.

Given the PROTO for a Part, I can define an instance of a part:

```
DEF PART1 Part { mode 1 partColor 1 0 0
  coord DEF PART1COORD Coordinate {point [0 0 0  1 0 0  1
1 0]}
   color DEF PART1COLOR Color {color[1 0 0  0 1 0  0 0 1]}
   lineIndex [ 0 1 2 0 -1]
   faceIndex [ 0 1 2 -1]
}
```

2.3 Animating a Mesh Model

The coordinates of the mesh are sometime displaced during a simulation. Also, we may want to color the vertices to display information about some computed value at each coordinate point. There are several ways to animate this to show the simulation values over time.

2.3.1 Animating with Interpolators

The simplest method to displace coordinates is to use the CoordinateInterpolator node. One can simply place all the coordinate points of all the timesteps into the keyValue field of the interpolator. The CoordinateInterpolator node will generate an output event with all the coordinates for the timestep which can be routed to the Coordinate node.

I discovered that VRML97 does not define a corresponding interpolator node for pervertex color. There isn't an interpolator node for an array of scalar values either. I wrote my own versions of these nodes using PROTOs with Script nodes.

Interpolators have some drawbacks. Using Interpolators requires the browser to perform a fair amount of calculation and data copying for each frame displayed. This is even more noticeable when the calculation is being performed by scripting code as opposed to native code. Furthermore, a linear interpolation between computed timesteps may display misleading information if the values do not change in a linear fashion.

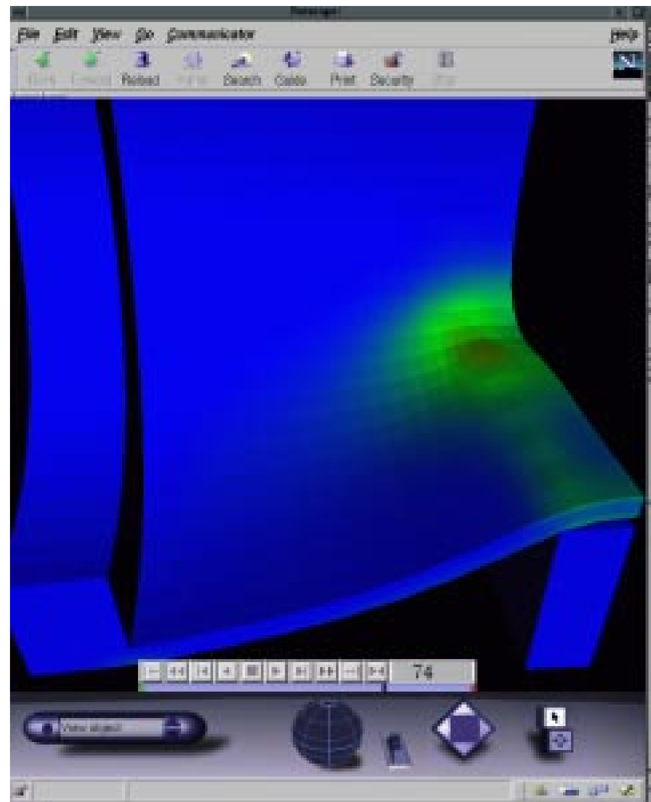

Figure 2: A finite element analysis of a pipe being creased.

2.3.2 Animating with a Switch node

The second method I have employed to show animation is to use a Switch node in which each choice node within the Switch encompasses a timestep. It can be more difficult to conserve space with this approach. We would need duplicate geometry nodes for each timestep so that each could contain the specific coordinate and color nodes for that timestep. This means the coordIndex field would need to be duplicated for each timestep when using either an IndexedLineSet or an IndexedFaceSet.As long as all the geometry fits in memory, this method displays at good frame rate. The browser simply traverses a different subtree for each frame.

I think about how much simpler this could have been if Switch nodes had been allowed anywhere within the node hierarchy, but unfortunately we can't have a Switch node select the Coordinate node of our geometry.

2.3.3 Animating with a Script node

The last method I'll describe for animating a model involves setting the coord and color fields for the geometry via a Script node. It is best to set the node fields of the geometry nodes rather than setting the point field of the Coordinate node, since the latter would likely involve copying all the coordinate points to a new array each time a new frame is displayed.

A simple instance of such an animation Script node follows:

```
Script {
directOutput TRUE
field SFNode partnode USE PART1
field MFNode coordnode [
  Coordinate {point [0 0 0  1 0 0  1 1 0]}
  Coordinate {point [0 0 0  1 0 0  .5 1 0]}
  Coordinate {point [0 0 0  1 0 0  0 1 0]}
]
field MFNode colornode [
  Color {point [1 0 0  0 1 0  0 0 1]}
  Color {point [1 0 0  0 1 0  1 0 1]}
  Color {point [1 0 0  0 1 0  1 1 0]}
]
eventIn SFFloat set_fraction
url "vrmlscript:
  function set_fraction(f) {
    i = Math.round((coordnode.length - 1) * f);
    if ( i < coordnode.length)
      partnode.set_coord = coordnode[i];
    if ( i < colornode.length)
      partnode.set_color = colornode[i];
  }
  "
}
```

3 Getting Performance from VRML

3.1 General Performance Tips

There are several performance aspects to consider when structuring a VRML based visualization: the time to load the model, the time required before the viewer can start viewing and interacting with the model, and the frame rate at which the model can be viewed.

The load time will be most sensitive to file size. File size may be the dominating consideration if the VRML is being loaded across a slow network.

The simplest action to reduce network transmission time is to compress the file. This may slightly increase the time for the client to load the model as it now must decompress the file first.

File size can be reduced by stripping out comments and superfluous white space. This will improve both transmission and parsing time, at the expense of making the file less readable to humans.

The PROTO construct can also greatly reduce the file size. The use of the Atom PROTO and the Bond PROTO in the molecular dynamics example replaced a significant amount of redundant VRML code.

Run time performance is affected by several factors. In general, we want to do minimize the amount of calculation required to generate a frame, as well as the amount geometry we need to render. This means limiting the use of interpolators and scripts.

Memory usage also effects run time performance. Frame rate will likely plummet if the entire model no longer fits memory. The DEF and USE keywords can be used to decrease memory usage. For example in the molecular dynamics model, I defined a Sphere node outside of the Atom PROTO and I referenced that single Sphere node within the Atom PROTO. This means that all instances of Atoms will share one Sphere. The Browser will only have to store one set of coordinates for a sphere. If I had simply included a Sphere node within the PROTO, the browser may well have created a new sphere, along with all its implied coordinates, for each atom.

3.2 Model Translation Processes

I have used two processes to produce VRML visualizations from simulation data. First, I have written translator applications that read simulation result files and write out corresponding VRML files. Second, I have written java applets that read result files and interact directly with a running VRML browser using the External Application Interface.

Writing out a VRML file reduces the dependencies on the java implementation and the interface methods of the HTML browser. However, then all controls for the visualization must be VRML widgets, for which there are no conventions yet. Otherwise, one may have Script nodes launch control windows. However, script node language support is as yet inconsistent, so one cannot rely on having either ECMAScript or Java available from a Script node.

Translating into a single VRML file can produce a huge ASCII file that takes a long time to parse and start executing. I placed the coordinates for each step inside an Inline node so that the mesh would appear right away and the frames would be read as requested.

A java Applet allows much more programmer control of the run time environment. It allows the data to be transmitted in a compact binary format. The applet can inject new frames into the browser as they are read, allowing the user to start visualizing the initial frames immediately.

3.3 The Evolution of the HyperTrace Viewer

The most involved VRML project I have undertaken, was to write a VRML-based viewer for HyperTrace^{TM} visualization. HyperTrace is an analysis program for computational fluid dynamics simulations. HyperTrace technology traces and visualizes thousands of particles in a computed CFD flow field.

My first step was to write a translation program that would read the binary HyperTrace file format and print out a VRML ASCII format file. That file could then be loaded into a web browser and viewed by the VRML plugin.

I represented the mesh as a semitransparent IndexedFaceSet. I put the points for the particle trace into a PointSet node. I put all of the point positions for every time step into a CoordinateInterpolator node. HyperTrace also has the option of coloring each particle to represent its velocity at each time step. Since VRML97 doesn't have an interpolator for arrays of colors, I was forced to write a Script node for that task.

The frame rate at this point was terrible. The script-based interpolator could not evaluate tens of thousands of particles for each frame in a reasonable time. Even using the CoordinateInterpolator alone was slow since it had to calculate all the coordinates points then copy them to the Coordinate node.

I adopted the animation method using a Switch mode. This required a PointSet containing a Coordinate node and a Color node for each time step, but replicating those nodes produced trivial amount of overhead. The animation now ran reasonably close to the speed of the visualization program written directly in OpenGL.

Finally, I sorted the particles into 32 different colors at each timestep. I created an Appearance node for each color. Each time step under the Switch node was now a Group node containing a Shape node with a PointSet and an Appearance for each color. This eliminated the need for the pervertex Color node, and nearly doubled performance for large particle traces. The VRML viewer was now displaying at the same frame rate as the HyperTrace program.

The translation approach still had a major drawback. The VRML file was huge, and it took a long time to transmit, load, and parse the file. In order to have the viewer display some initial frames more quickly, I changed the translator to write a separate VRML file for the particle positions for each frame. The main file then referenced those files from an Inline node for each time step. Now viewer displayed each frame as it was being loaded.

Using a translator program required a preprocessing step before the HyperTrace output could be visualized. My goal had been to provide a convenient, web-based tool to view results. I feared that making users translate a file first before they could load the resulting VRML file would discourage use. Furthermore, HyperTrace files are a compact, binary representation of the model that would be much quicker to transmit over a network than the large ASCII text VRML equivalent. This was the impetus for using a java applet to load the HyperTrace file directly and communicate with the VRML browser via the External Application Interface[3], EAI.

The java applet allowed me to place the model viewing controls within the java applet. This allowed the full VRML window to be used for viewing the model itself, rather than containing control widgets.

Most of the performance measures I had used in the translation process still applied when using the EAI. Only the Inline nodes were no longer needed.

I executed the applet's file reading code in a separate java thread. This allowed the main thread to respond immediately to user interaction. The read thread adds geometry to the VRML browser as it is read. Therefore, the viewer can start viewing geometry and frames with little delay.

I needed to retain references to some nodes in order to modify them during the simulation. I avoided retaining any references to large arrays used within nodes, since that would then require a copy of the array to be allocated for both the java applet and the VRML browser.

There were several performance considerations for the java code. With Just-In-Time compilers for java, the java computation code performed very well. But, the performance of io methods in java vary greatly. I initially used the ReadByte method, but this proved very slow. I needed to read larger quantities of data into a program buffer to get good performance.

The other major time consumer was memory allocation. The EAI method for setting coordinate points requires a two-dimensional array of exact length as an argument. Since I was sorting the particles by color for each time step, each array would almost invariably be a different length, preventing the reuse of the previously allocated array. Memory allocation was consuming up to 90 percent of execution time.

I was not able to convince browser vendors to add a method to the EAI specification that would also take a length argument for arrays. I resorted to work-around involving a Script node and a feature of ECMAScript. I now reuse a two-dimensional array that is large enough to hold any set of particle points in the model. I pass the array as an input event to a Script node, which then copies the full array into browser allocated memory. I then pass the length into the Script node, which then truncates the array to the correct length, and copies it to the Coordinate node. I created this Script node from within the java code as follows:

```
String ScriptString =
“Script {“ +
“directOutput TRUE “ +
“field SFNode coord NULL “ +
“field MFVec3f point [] “ +
“eventIn SFNode set_coord “ +
“eventIn MFVec3f set_point “ +
“eventIn SFInt32 set_length “ +
“url [ “ +
“  \”vrmlscript: “ +
  “ function set_coord(node) { coord = node;} “ +
  “ function set_point(points) { point = points;} “ +
  “ function set_length(len) { “ +
    “ point.length = len; “ +
    “ coord.set_point = point; “ +
  “ } “ +
“  \” “ +
“] “ +
“}”;
Script = browser.createVrmlFromString(ScriptString)[0];
```

```
Node PointSet = browser.createVrmlFromString("PointSet {}")[0];
((EventInSFNode) Shape.getEventIn("set_geometry")).setValue(PointSet);
Node Coordinate = browser.createVrmlFromString("Coordinate {}")[0];
((EventInSFNode)PointSet.getEventIn("set_coord")).setValue(Coordinate);
((EventInSFNode)Script.getEventIn("set_coord")).setValue(Coordinate);
((EventInMFVec3f)Script.getEventIn("set_point")).setValue(points);
((EventInSFInt32)Script.getEventIn("set_length")).setValue(idx);
```

The VRML HyperTrace viewer now performs well. It is uses VRML, java, and ECMAScript, that are widely supported, so that this same viewer can be used on almost computer platform.

4 Conclusion

The availability of VRML browsers make VRML a desirable format in which to disseminate simulation results. VRML can represent many of the common models used for visualization. With careful use of VRML, scripting and java, a VRML visualization can perform well.

Acknowledgements

The author wishes to thank Cal Kirchhof, Silicon Graphics, for his support of this project. Thanks also to the HyperTrace team from Silicon Graphics: L. Zullo, M. Liu, C. Andreasen, R. LaRoche, and S. Choudhary.

References

[1] ISO/IEC 14772-1, The Virtual Reality Modeling Language, 1997.

http://www.vrml.org/Specifications/VRML97

[2] Rikk Carey, Gavin Bell, The Annotated VRML 97 Reference Manual, 1997.

http://www.best.com/~rikk/Book/book.shtml

[3] C. Marrin, Proposal for a VRML2.0 Informative Annex, External Authoring Interface Reference, January 21, 1997.

[4]External Authoring Interface Working Group,,

http://www.vrml.org/WorkingGroups/vrml-eai/

[5]HyperTrace, http://wwwche.cray.com/hypertrace/

Trademarks

OpenInventor is a trademark of Silicon Graphics, Inc.

HyperTrace is a trademark of Silicon Graphics, Inc.

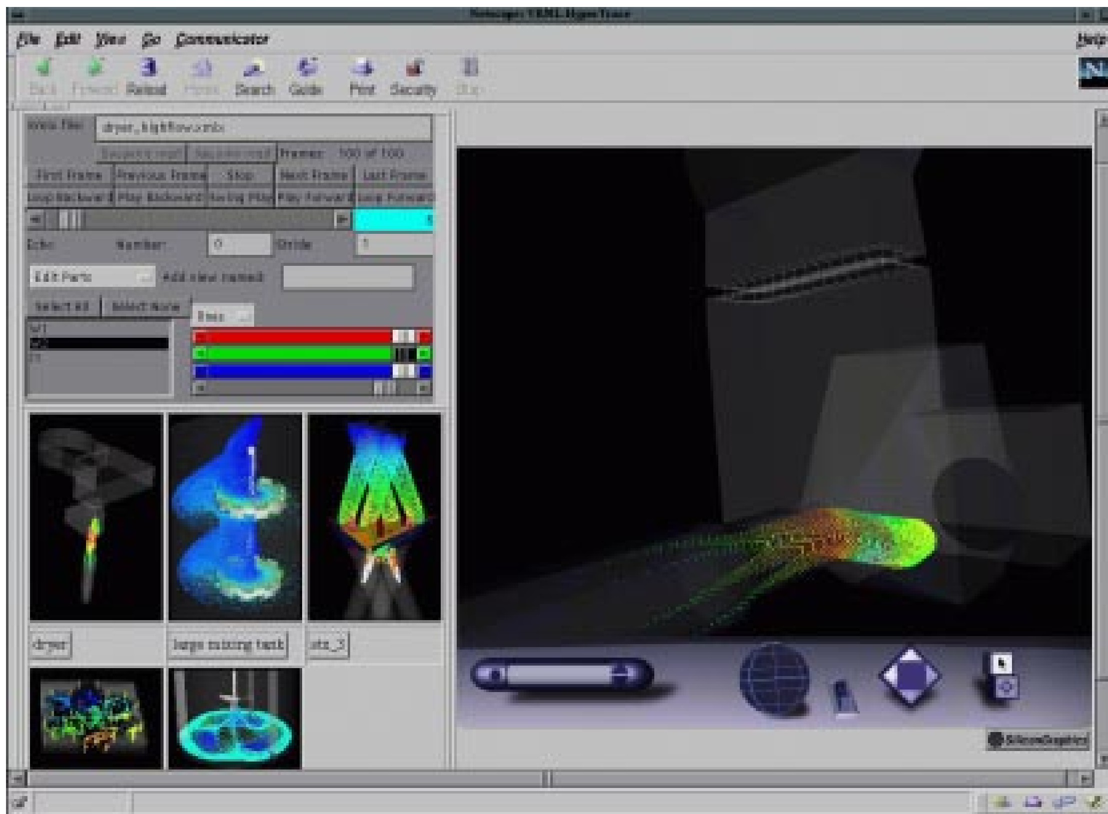

Figure 3: The VRML HyperTrace Viewer