IBM CONFIDENTIAL

VRML 2.0 Proposal - IBM Japan

What is our main design goal ?

A virtual environment in our VRML 2.0 architecture is an environment in which users can interact in real time with a time-dependent and autonomous virtual world. The virtual world simulates animated motions, collisions, and constraints in response to users' actions. A unique feature of our VRML 2.0 architecture is that it provides a time-dependent description of the world by the following means:

- Callback invoked in scheduled time
- Scheduled Fields of Scene Graph nodes for motion

Our time-dependent world is scheduled but can be changed through users' interaction. Our VRML 2.0 handles this interaction by providing a model of events and callbacks.

In preparing our specification of VRML 2.0, we gave consideration to the following points:

- A totally new language is hard for programmers to learn.
- Procedures, which may take a lot of processing time, should be excluded from action representation (callback function).

Bad assumption

The design goal of our architecture is to permit the real-time visualization of an ever-changing virtual space. Our VRML 2.0 does this by scheduled motions, and does not contain any method for describing actions procedurally by using script languages (Java *, etc.), because such descriptions allow a user to write procedures that are too complex and time-consuming for real-time rendering. Since our VRML 2.0 presumes that another system generates scheduled motion data in advance or in parallel, VRML 2.0 browsers only need to visualize a virtual scene.

1. Overview
    - World description
    - Scene composability
2. Description
    - Behavior model
    - Enclosing volume and intersection checking
    - Scene graph connection
    - Audio and Ear nodes
3. VRML2.0 File Format
    - General language syntax
    - Coordinate system
    - Remote connection
4. File format of VRML 2.0 nodes

5. Example
6. Background

Contributors

- Akio Koide
- Ryo Yoshida
- Tatsuo Miyazawa
- Takaaki Murao
- Masaaki Taniguchi

IBM Japan, Ltd., Tokyo Research Laboratory
1623-14 Shimotsuruma, Yamato-shi, Kanagawa 242, Japan

---

yoshidar@trl.ibm.co.jp

IBM CONFIDENTIAL

World description

A virtual world is represented by a collection of "nodes," which form a directed graph called a "scene graph." A node is a container of data called "fields." Fields may be geometrical data, colors, or angles, depending on the type of node. This representation is inherited from VRML 1.0.

1. Description of a time-dependent world

Our VRML 2.0 architecture allows a user to describe his own time-dependent autonomous world. This description is supported by the following simple means:

- Scheduled Fields with motion engines
- Callback invoked in scheduled time
- Constrained fields of nodes

Here motion engines contain time-dependent data that schedule the behavior of the fields of nodes connected to them.

2. Scheduled Fields

Our proposed motion engines contain sequences of time-dependent field data and their control parameters. A user can assign time-dependent behaviors to any nodes by connecting motion engines to the fields of those nodes. Since a behavior generally involves cooperative motion of multiple nodes, a motion engine can be connected simultaneously to the fields of different nodes. The resulting behaviors of motion engines are predictable. Therefore we call this the "scheduled fields" approach. The main points of this approach are as follows:

- It is simple and easy for a user to estimate the motion resulting from a particular assignment. If he schedules the fields of the transformation nodes, the behavior is the same as in an animated cartoon.
- It is simple for a browser to control resources in order to maintain time dependence, since a user's callbacks occupy less CPU time. The browser can easily visualize a motion and and event-driven callback simultaneously, even in a single-task operating system.

Motion engines support looping and interpolation for scheduled fields, to allow flexible description of the time-dependence of fields.

If a field value is allowed to be discrete, interpolation cannot be applied to the scheduled fields. The browser uses one specified parameter during each specified time interval. If a field value is allowed to be continuous, interpolation can be applied to the scheduled fields. The browser interpolates the field values at requested points in time. A user can also specify field values as velocities or accelerations of

changes. In this case, the browser computes the field values by integrating them.

Our VRML 2.0 architecture is more powerful than any existing animation system as a result of two simple but powerful new ideas: introduction of time-dependent descriptions for all kinds of fields, and support for the use of velocities and accelerations. For example, if a user sets a velocity for a car, it runs at a constant speed without any loop specification. Acceleration can be used to model gravity. The color of a sky can be changed gradually by applying a scheduled field to a material node (one of the subclasses of property nodes).

3. Scheduled callbacks

A callback is the reaction of an object to a particular event. In a conventional model of event and callback, a user can attach just one callback (reaction) to an object (node) for one type of event type, and the browser will invoke the callback as soon as the event occurs. In our VRML 2.0, a user can specify the time of a callback after the occurrence of the event that prompts its invocation. Therefore a user can add many callbacks to the node for a single type of event, as long as their scheduled times are different.

4. Constrained fields

There are two types of constraint on motion in our virtual environment:

- Simple type of constraint:
  This is the range of fields in a node. It is applicable to continuous fields generally.
- Extended type of constraint:
  The constraint of a target's motion is provided by its surrounding objects. The range of fields is required for constrained motions. This type of constraint is only applicable to a transform node.

Our VRML 2.0 supports constraints of the first type, that is, simple constraints. Except in the case of scheduled fields, the browser checks constraints when fields are changed.

5. Manipulation of a world description

Here we briefly describe the operations that can be applied to a world description.

- Update: Change the fields of nodes.
- Insert: Add or copy nodes. A user stores a created or copied node as a single node of a scene graph or attaches as it to some part of a scene graph. The field values of a created node are set to default values.
- Delete: Delete or move nodes. If a node is deleted, the data it contains are lost forever. Moving a node involves detaching it from some place in a scene graph and attaching it at another place in the graph or storing it as a single node of the graph. The data it contains are reserved.

If the world is active, a user can perform the above-mentioned manipulations only in callbacks to events.

VRML 2.0 proposal top page

[Tokyo Research Laboratory home page | IBM Japan home page]

IBM CONFIDENTIAL

IBM CONFIDENTIAL

Scene composability

VRML 2.0 can construct a very large virtual world. To achieve this, it uses the following concepts about the creation of a scene graph:

- Scalability
- Composability
- Multiple users

On the assumption that VRML 2.0 is used in a networked environment, we remove the limits on the design of a scene graph. As regards scalability and composability, we assume that our virtual world can be composed of worlds created by different people. Each graph of a world has its own root node that can be designed independently, and VRML 2.0 will select, not switch between, one or more graphs in order to draw the worlds. The scene graphs are connected through a special instance to share data with each other.

Generally, there are two approaches to supporting multiple users: the client/server approach and the multicast approach. Since we emphasize the scalability of the system and the composability of the scene, we consider that VRML 2.0 should use the multicast approach, which has already proved to be suitable for large-scale interactive multi-user system.

In our VRML 2.0, the separator nodes (Separator, ColliderSeparator, and SharedSeparator) have a field named "owner" to control access authority and priority when multiple users manipulate the same object (node) simultaneously.

VRML 2.0 proposal top page

[Tokyo Research Laboratory home page | IBM Japan home page]

[IBM home page | Order | Search | Contact IBM | Help | (C) | (TM)]
Last modified: Fri Feb 2 17:00:00 JST 1996

IBM CONFIDENTIAL

Behavior model

A behavior is a time-dependent and autonomous motion. Here, we describe our behavior model and its supported functions in detail.

1. Motion engines and scheduled fields

A motion engine can contain the following data:

- Type of data specification: acceleration, velocity, or relative or absolute data for specifying field values
- A sequence of pairs of sampled times and (field) data, or
- The name of a file (or URL) containing a sequence of pairs
- Output of this engine
- Input time
- Time to be invoked first
- Time scale factor
- Whether or not there are loops, and the number of repetitions
- Whether or not there are interpolations
- Basis values for cases other than absolute field data
- Output switch (active or inactive)

Sampled times are measured from the instant at which the motion engine is invoked. There is no need to sample times at equal intervals or in relation to the rendering frame rate. A sequence of pairs must be finite. The loop repetition can be finite or infinite.

If the output of a motion engine is not interpolated, the field value is replaced by sampled field data at the instant that the corresponding sample time arrives. If the output is interpolated, a browser linearly interpolates the data at the requested time.

The scheduled data can be absolute or relative field values, or they can be the velocities or accelerations of the changing field values. When the data are relative field values or velocities, the browser computes field values by adding a basis field value, which is a value specified by a user or the initial value at the time the motion engine is invoked. When the values are accelerations, a basis field and a basis velocity are required; these are values specified by a user or the initial values at the time the motion engine is invoked.

2. Callback

VRML 2.0 behavior can occur either as a result of an event's being triggered or regardless of the event. The behavior is represented in the VRML 2.0 file by a combination of the behavior and the object (a

group of shape nodes). The condition for an object to start or change a particular behavior is closely related to the type of event that triggers the behavior, and other conditions. This condition is also represented in the VRML 2.0 file.

- Event Type:
  - PICK: Execute a method when the object is picked.
  - GRAB: Execute a method when the object is held.
  - DRAG: Execute a method when the grabbed object is moved.
  - RELEASE: Execute a method when the object is released.
  - COLLIDE: Execute a method when the object is collided with another object.
  - MOVE: Execute a method when the object is moved.
  - RENDER: Execute a method when the rendering is completed.
  - WORLDIN: Execute a method when a world is loaded.
  - WORLDOUT: Execute a method when a world is unloaded.
  - ENTER: Execute a method when a viewpoint enters an area.
  - LEAVE: Execute a method when a viewpoint leaves an area.
  - USER: Execute a method when a user event is received.
  - NONE:
- Current State:
  The current state of an object when an event occurs affects the object's reaction to the event. For example, if the object grabbed by a user is hit by another object, it may not react.
- Action Object and Action Node:
  Each object can cause a particular event. The reaction will occur at a specified target node.
- Delay:
  After an event occurs, the reaction is delayed for a while.
- Manipulation (method) and Parameters:
  - SET: Set or change a field value.
  - COPY: Copy a node in a scene graph.
  - ADD: Add a node to a scene graph.
  - MOVE: Move a node in a scene graph.
  - REMOVE: Remove a node from a scene graph.

The parameters field contains a value or a path in the scene graph. A user can specify the name of a file containing the data for this parameters field.

3. Reaction and visualization

The event and callback mechanism in this architecture is similar to that of Motif * on the X Window System * on a UNIX * workstation. During the loading of a VRML file, a browser translates the scene and behavior description in the file into its own internal representation. If the file contains an object declared by both the CLASS and the Callback, the browser registers the callback functions with the object. The number of callbacks is not limited to one function. After loading and interpreting the file, the browser completes its internal representation of the scene and the behavior.

A browser is required to provide the following functions:

- When an object (or a node) receives an event specified as the "EVENTTYPE" of a callback for obtaining a reaction, the browser may invoke the callback.

- The browser detects collisions between specified target objects and invokes a collision event.
- While the invoked callbacks are being processed and the field values of the nodes connected to the engines are being evaluated, the browser changes its internal scene description.
- The browser visualizes a scene in the virtual world.

In our VRML 2.0, a user writes a callback that is simple in comparison with the callback described in the Java *. This is because we attach importance to the processing time for changing the scene before rendering, although there is a trade-off in programmability between a flexible, complex callback and an easy-to-process, simple callback that takes less time to calculate.

4. Combining motion engines

Our VRML 2.0 offers two ways of combining motion engines into an integrated, single motion engine:

- Serial combination of motion engines.
  Motion engines are connected as a time series, so that they work one after another. In this case, each motion engine must have fields of the same type.
- Parallel combination of motion engines.
  Motion engines are connected so that they work simultaneously in cooperative motion.

The same motion engines can be shared between these combined motion engines.

5. Constrained motion

The variable range of field values may be constrained by a browser or by a user:

- System-Specified Constraints:
  These can be applied to some vendor-supplied nodes. Our VRML 2.0 sets a default variable range of field values. For example, the colors of material nodes should be between 0 and 1. If the value is less than 0, the browser sets it to 0; if it is greater than 1, it sets it to 1.
- User-Specified Constraints:
  A user can apply a constraint to the field values of nodes.

In our VRML 2.0, a user can specify the range of a scalar field by setting a lower bound and/or an upper bound. For a vector-type field, he can specify the range by setting the bound of each field or the bound of a quadratic form.

VRML 2.0 allows two types of constraint when field values lie outside the constrained range:

- Field values remain at the point where they cross the boundary of the range, until they return within the range.
- Field values can move the boundary of the range as follows, until they return within the range. The browser connects the current position of a field value and a user-specified position with a straight line. It then locates the effective position of the field value on the intersection of the line and the boundary of the range. This action is only applicable to vector-type fields.

countries licensed exclusively through X/Open Company Limited.

VRML 2.0 proposal top page

[Tokyo Research Laboratory home page | IBM Japan home page]

IBM CONFIDENTIAL

Enclosing volume and intersection checking

1. Enclosing and representative volumes

In VRML 2.0, we distinguish enclosing and representative volumes.

- Enclosing volume (bounding volume):
  - A enclosing volume is a convex body that encloses a group of shape nodes including its children.
  - The browser computes an enclosing volume and automatically updates it when its descendants are moved or transformed. If a user specifies a minimal enclosing volume, the browser tries to compute the convex hull (convex closure).
  - An enclosing volume can be used for culling, picking, and automatic detection of collisions.
- Representative volume:
  - A representative volume is a 3D geometry that approximates some shape nodes. For implementation reasons related to intersection checking, the supported geometrical shapes may be somewhat limited.
  - An application programmer or VRML 2.0 supplier creates a representative volume.
  - A representative volume can be used for picking and automatic detection of collisions. However, it is not used for culling.

2. Colliders and Collidees

Our VRML 2.0 allows collisions. In the architecture, users can intersect each "collider" with a set of "collidees." A collider is a shape that can collide with geometric objects in a scene graph. A collidee is a shape node or a group of shape nodes in the scene graph that colliders can collide with. It has a collision group ID. The collision group ID allows users to define a group of shape nodes as belonging to different classes of geometric objects for intersection. A collider can collide only with collidees that have the same collision group ID as one of the colliders. The default set of collidees does not include any of the shape nodes in the scene graph.

A user can construct a collidee as a group of nodes that begins with a separator node. The separator node can contain the bounding box (and sphere, etc.) related to its child nodes for collision detection. On the other hand, a collider is constructed as a group of nodes that begins with a special separator node for the collider.

3. Description of collision detection

(a) Collidee specification
A separator node takes on a new role, "collision." The following data can be set as field data of the separator node:

- Collidable Flag:
  A flag is used to determine whether collision detection is performed on the collidee's children (shape nodes).
- Collide Group ID:
  SFString is used to determine whether a particular shape node should be examined during a particular collision request.

(b) Collider specification
VRML 2.0 uses collider-separator nodes to define colliders and their attributes. A special separator node takes on a new role, "collision detection." The following data can be set as field data of the node:

- Collider Flag:
  A flag is used to determine whether a group of shape nodes are colliders or not.
- Collide Group ID:
  MFString is used to determine whether a particular shape should be examined during a particular collision request.
- Collider Mode:
  A mask is used to specify the behavior of the collision detection.
  - BBOX: Intersect with bounding volumes of primitives
  - PRIM: Intersect with primitives
  - POINT: Return intersection point
  - NORM: Return primitive normal at intersection point
- Collider Geometry Type:
  A bounding box, or a surrogate such as a set of line segments of the collider called "Collider Geometry," can intersect with the collidee or the bounding box of the collidee, although a collider has its own geometrical shape. If the collider geometry is not specified explicitly, the bounding sphere of the collider is used as a collider geometry. A single-value field is used to specify the type of the collider geometry.
  - BBOX: Use bounding box of the shape
  - BSPHERE: Use bounding sphere of the shape
  - SEGMENT: Use a set of line segments
- Collider Geometry:
  A set of line segments of the collider is called a "collider geometry." A single line segment or each of several line segments is defined by specifying the coordinates of its starting point and end point.

VRML 2.0 proposal top page

[Tokyo Research Laboratory home page | IBM Japan home page]

IBM CONFIDENTIAL

Scene graph connection

1. Shared node

Suppose that a scene contains several rooms: a large conference room, a show-room, a private room, and a machine room. Since a room has generally doors, a user may look or move into the next room through the doorway. Here, if two scene graphs for each room are alternated, the user cannot see the other room, in which he does not exist, and the changing states of that room cannot be processed. Furthermore, these scenes are required to be designed as one large scene graph from the beginning, with switch nodes.

In our VRML 2.0, it is not required that the conference room and the the private room should have the same scene designer. Furthermore, even if a world is composed of several scene graphs, it is not required that one person should design the world as a scene graph.

Two scene graphs can be connected at the special node, called the shared separator node. In the Open Inventor concept, if the same shape node (or the same group of shape nodes) is referred to in scene graph traversal, a renderer interprets it as two instances with the same content. In VRML 2.0, on the other hand, if the shared node exists in two independent scene graphs, VRML 2.0 interprets it as one instance jointly owned by two graphs.

The transformation between the root node and the shared node in a scene graph can be calculated, and a similar transformation can be calculated in another scene graph. When the two graphs are connected through the shared node, they are exactly like a single scene graph in which the new pseudo-root has the two original root nodes as its children. For multiple scene graphs to be identified, it is necessary that each root should be named uniquely, using DEF.

The shared separator node contains the following fields:

- File
- Root Name
- World ID
- Node Name
- Connect Point

To connect two scenes, a user can specify the next scene graph by providing a "File" (URL) and "Root Name." Each scene is distinguished by a "world ID." Two scenes are connected at the point ("Connect Point") where the node is located ("Node Name").

2. Running through the distributed worlds

In our VRML 2.0, a man can walk through a shopping mall that consists of various shops designed by

different owners. If a scene is saved as a file, and that scene contains a man, the output file contains graphs of both the scene and the man. If the man is in another scene, the output contains only a graph of the scene.

Our VRML 2.0 can handle this case by using a RootConnect node. In this case, the man is written as a scene graph, the scene is written as another scene graph, and the RootConnect node connects the root nodes of both the graphs. The fields are the parent root name, the child root name, and the positional relations between both roots ("translation" and "rotation").

VRML 2.0 proposal top page

[Tokyo Research Laboratory home page | IBM Japan home page]

IBM CONFIDENTIAL

Audio and Ear nodes

An audio node contains audio data such as sound and voice. The node is located at some position in the virtual world, and it starts and stops playing audio data when some event occurs. Ear nodes have face position and orientation. For the convenience of those building the world, ear nodes should be able to inherit face position and orientation.

VRML 2.0 proposal top page

[**Tokyo Research Laboratory home page** | **IBM Japan home page**]

[**IBM home page** | **Order** | **Search** | **Contact IBM** | **Help** | **(C)** | **(TM)**]
Last modified: Fri Feb 2 17:00:00 JST 1996

IBM CONFIDENTIAL

General language syntax

1. Header line

The first line of the file is reserved as a system header line. The header line must start with the character '#', followed by the system identifier, version, and character sets, and must end with a at newline character. The following is an example:

```
#VRML V2.0 Japanese:sjis
```

A character sets is specified as a pair of a language type and a character set, separated by a colon. Our VRML 2.0 does not allow international characters encoded with the UTF-8 of the ISO standard, because that encoding cannot correctly handle double-byte characters.

2. Character sets and tokens

The header line defines the character sets in use. Japanese characters (7-bit JIS, shifted-JIS, or EUC) can be used in comments and string field values if the header line defines this capability.

A new-line character has a special meaning as a terminator of the first line of the header and of comments. Otherwise, new line characters are treated as white-space characters, like blanks, tabs, and carriage returns.

White-space characters function as separators for decomposing an input string into a collection of tokens. Extra white-space characters are ignored wherever they appear outside string fields. The character '_' is treated as a normal alphabetic character.

In some contexts, the "equal" sign '=' and the period '.' have special meanings as connectors, as will be described later. Other special symbols function as separators for decomposing tokens, like white-space characters. They also have their own meanings, but generally they must be interpreted in the context.

The place at which separation occurs is just before a special symbol or a sequence of special symbols. For example,

```
DEF basketball$0
```

is equivalent to

```
DEF basketball $0
```

There are some paired special symbols:

- An opening double quotation mark '"' must be matched by a closing double quotation mark '"'. What lies between them is interpreted as a string field. If a double quotation mark is to be included in the string field, it should be preceded by a backslash '\'.
- An opening brace '{' must be matched by a closing brace '}'.
- An opening square bracket '[' must be matched by a closing bracket ']'. These pairs are used for multiple-value fields.
- An opening parenthesis '(' must be matched by a closing parenthesis ')'.

3. Comments

At any place except in the first line of a file, the number sign '#' begins a comment; all characters until the next new-line character or carriage return are ignored. The only exception to this is within double-quoted SFString and MFString fields, where the '#' character is part of the string.

4. Declarations and assignments

All variables must be declared before they can be used. A variable name is an identifier, and a declaration is a type (field-type) followed a variable name.

```
<field type>   <variable name> ;
```

The "equal" sign '=' is the basic assignment operator.

```
<variable name>  =  <value> ;
```

The scope of a variable begins at the end of its declarator, and persists to the end of the file. The following list contains all the assignment operators:

```
=, +=, -=, *=, /=, %=
```

one method of transmitting information across files is to use external variables. A declaration for an external variable looks the same as a variable declaration, but the declaration must be preceded by an explicit "extern" keyword. An external variable must be defined exactly once in another file.

5. Multiplication and addition operators

The multiplication operators *, /, and % group left-to-right. The addition operators + and - group left-to-right. The operands of *, +, - must be of the type SFFloat, SFLong, SFVec2f, SFVec3f, SFColor, SFTime or SFMatrix. The operands of / must be of the type SFFloat, SFLong, SFVec2f, SFVec3f, SFcolor, or SFTime. The operands of % must be of the type SFFloat, SFLong, or SFTime. The * operator denotes multiplication. The / operator yields the quotient, and the % operator the remainder, of the division of the first operand by the second. The result of the + operator is the sum, and that of the - operator is the difference, of the operands. These operators must be used outside of a node's block. The "if" expressions below and these operators are evaluated only once at the time a file is loaded, mainly for quality-of-service control.

6. "If" expressions

In a construction of the form

```
if (expression1)
    statement1
else if (expression2)
    statement2
  . . . . .
else
    statement3
```

If expression1 is nonzero (true), then statement1 is executed, and statement2 and statement3 are skipped; if expression1 and statement2 are zero, then statement1 and statement2 are skipped and statement3 is executed. Each statement can be a single statement or several in braces {}. The following table contains the relational, equality, and logical operators.

```
less than:                  <
greater than:               >
less than or equal to:      <=
greater than or equal to:   >=
identically equal to:       ==
not equal to:               !=
negation:                   !
logical and:                &&
logical or:                 ||
```

7. File structure

Our VRML 2.0 is formed by a collection of class definitions of the following type:

```
<class_name>  {
    a collection of pairs  <method_name>  <values>
    a collection of pairs  <field_name>   <values>
        and/or
    a collection of children
}
```

Here <class_name>, <method_name>, and <field_name> are a sequence of alphabetic characters and the character '_'. <class_name> must be the name of a node type; that is to say, only nodes can be defined. Methods and fields determine the number of tokens as their specified values. In VRML 1.0, there is no symbol for terminating statements, but in VRML 2.0, the semicolon is a statement terminator.

After the required header, a VRML 2.0 file can contain multiple VRML nodes. Those nodes may be group nodes, each containing any number of other nodes. Braces are also used to group declarations and statements together into a block, so that they are syntactically equivalent to a single statement. There is no semicolon after the right brace that ends a block.

8. Field

Fields are written as follows:

```
<field_name>      <value(s)> ;
```

If the field value has the default value, the field is not written out. Multiple-value fields are written as a series of values separated by commas, and enclosed in square brackets. Single-value fields do not contain any commas or brackets. Connections are written as follows:

```
<Object containing the field> . <Name of the field>
```

The engines must be connected to at least one of the fields by using this connection representation. A connection between an engine and a field is written as follows:

```
<field name>    =    <object name>    .    <field name> ;
```

An expression followed by a dot followed by a field name is a postfix expression. The first operand expression is a class name (instance name) or a field name, and the second operand must be a field name of the class (the instance), or a member of the field type.

9. Naming convention

A clear convention on case for field names and node names is adopted, to allow parsers to tell when they have finished with fields and are now parsing children. Field names start with a lowercase letter "fieldName", while node names start with an uppercase letter "NodeName" with the prefix "Vrl". The rest of the characters can be any printable ASCII characters except curly braces, square brackets, parentheses, single or double quotes, sharp signs, backslashes, equals, signs operators, dollar signs, colons, semicolons, periods, or ampersands.

10. Named instances

A user can name nodes as instances. Since VRML 2.0 uses many keywords as class, field, and method names, a user-specified name must start with the keyword DEF or USE. The keyword DEF defines a named instance of a node. The keyword USE is used when it is referred to.

VRML 2.0 has no definite concept of scope for the names of instances. It processes the keywords DEF and USE according to the following rules:

- A named instance using DEF or USE is effective in a single file. There is no mechanism for referring to named instances of other files.
- The keyword USE indicates that the most recently defined instance should be used again.
  - The scope of named instances is not limited by the braces { and }.
  - The same name can be used for different instances with the keywords DEF and USE.
  - A name goes out of scope when another DEF of the same name is encountered.
- There is no mechanism for defining and referring to the names of field instances. If a user wants to

assign particular values to fields and refer to them, he should create instance nodes and substitute their fields at the places where he uses them.

```
VrlSeparator {
    DEF MyTrans VrlTranslation { translation 7 0 0; }
}
VrlSeparator {
    VrlTranslation {
        translation = MyTrans.translation;
    }
    VrlSphere {}
}
```

Node names must not begin with a digit, and must not contain spaces, control characters, single or double quotes, backslashes, curly braces, square brackets, parentheses, colons, semicolons, dollar signs, the operator's characters, equals signs, or periods.

The dollar sign '$' has a special meaning related to the name of an instance. Following numeric characters, it is a special name for the system to distinguish between unnamed instances and same-named instances.

11. CLASS

Our VRML 2.0 uses a new keyword, "CLASS." In contrast to the DEF keyword, the CLASS keyword defines an object without creating an instance. It is possible to creates labels for instances created by the USE keyword. CLASS is used to define a object that may have behaviors (the event handler) by the "Callback" keyword. The callback keyword is followed by definitions of the event type, the name of the object that triggers the event, and the name of the node to which the behavior is applied. The rules related to the scope follow those of DEF.

```
CLASS MyObject VrlSeparator {
    VrlMaterial { }
    VrlCube {
        ...
    }
    VrlCallback {
        eventType    COLLISION;
          ....
    }
}
```

12. Including file

Like VRML 1.0 and Open Inventor, our VRML 2.0 can include other files. It has the same syntax:

```
VrlFile {
    name  <file name>
}
```

Here <file_name> is a string that specifies a file name. Therefore it starts and ends with a double quotation mark '"'. VRML 1.0 treats File as a node and imposes the restriction so that in the including file, a user cannot refer to named instances of the included files.

A user can use the method "Include," which simply inserts the contents of an included file into the place. The user can then refer to named instances of the included files at any place after they are inserted. This is a VRML 2.0 language facility provided by means of a preprocessor, which is conceptually a separate first step in interpretation. The syntax is as follows:

```
VrlInclude {
    name  <file name>
}
```

13. Field types

There are two general classes of fields: fields that contain a single value (where a value may be a single number, a vector, or even an image), and fields that contain multiple values. Single-valued fields all have names that begin with "SF", which multiple-valued fields have names that begin with "MF". Each field type defines the format for the values it writes.

Multiple-valued fields are written as a series of values separated by commas, all enclosed in square brackets. If the field has no values then only the square brackets ("[ ]") are written. The last may optionally be followed by a comma. If the field has exactly one value, the brackets may be omitted and just the value written.

Field Types are;

- SFBitMask
- SFBool
- SFColor/MFColor
- SFEnum
- SFFloat/MFFloat
- SFImage
- SFLong/MFLong
- SFMatrix
- SFRotation
- SFString/MFString
- SFVec2f/MFVec2f
- SFVec3f/MFVec3f
- SFVelocity/MFVelocity A field containing an arbitrary velocity. SFVelocitys are written as four floating-point values separated by white-spaces. The field consists of SFvec3f for three-dimensional vector and SFFloat for the amount of velocity.
- SFAcceleration/MFAcceleration A field containing an arbitrary acceleration. SFAccelerations are written as four floating-point values separated by white-spaces. The field consists of SFVec3f for the three-dimensional vector and SFFloat for the amount of acceleration.
- SFTime Field containing a single time value. Each time value is written to file as a double-precision floating-point number in ANSI C floating-point format.

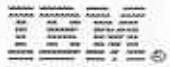
IBM CONFIDENTIAL

VRML 2.0 proposal top page

[Tokyo Research Laboratory home page | IBM Japan home page]

Last modified: Fri Feb 2 17:00:00 JST 1996

IBM CONFIDENTIAL



Coordinate system

VRML 2.0 and VRML 1.0 use a Cartesian, right-handed, three-dimensional coordinate system. By default, objects are projected onto a two-dimensional device by projecting them in the direction of the positive Z axis, with the positive X axis to the right and the positive Y axis up. A camera or modeling transformation may be used to alter this default projection. The standard units for specified lengths and distances and for angles are meters, and radians, respectively.

VRML 2.0 proposal top page

[Tokyo Research Laboratory home page | IBM Japan home page]

[IBM home page | Order | Search | Contact IBM | Help | (C) | (TM)]
Last modified: Fri Feb 2 17:00:00 JST 1996

IBM CONFIDENTIAL

Remote connection

The keyword "extern" also precedes a node name. An external (remote) node looks the same as an ordinary node except for the declarations of a URL and the name of a remote node. The remote node name must be named by DEF. As in visualizing the scene graph, a browser inquires the field values of the specified node on the remote machine. A file node actually loads the contents of a file, but the remote node can handle input and output without loading a file.

```
extern "file:/engine.wrl, MyRemoteEngine" MotionEngine {
    fields [ MFVelocity v, SFVec3f output ];
    output   0 0 0;
    active   ON;
}
```

VRML 2.0 proposal top page

[Tokyo Research Laboratory home page | IBM Japan home page]

[IBM home page | Order | Search | Contact IBM | Help | (C) | (TM)]
Last modified: Fri Feb 2 17:00:00 JST 1996

IBM CONFIDENTIAL

File format of VRML 2.0 nodes

All nodes have the prefix "Vrl".

> Collision (including modified Separator)
An object whose relations for collision you can specify.

```
CULLING ENUMS
     ON:   Always try to cull to the view volume
     OFF:  Never try to cull to the view volume
     AUTO: Implementation-defined culling behavior

COLLIDER/COLLIDABLE
    ON:  Include the object in the targets list for
         collision checking.
    OFF:

COLLIDEMODE
    BBOX:  Specify a bounding-box to perform collision checking.
    PRIM:  Specify a primitive to perform collision checking.
    POINT: Specify a point to perform collision checking.
    NORM:  Specify a primitive normal to perform collision checking.

COLLIDETYPE
    BBOX:    Specify a bounding-box to the collider geometry.
    BSPHERE: Specify a bounding-sphere to the collider geometry.
    SEGMENT: Specify a set of line segments to the collider geometry.

COLLISIONFLAG
    TRUE:   Has collided with a collidee.
    FALSE:  has not yet collided with any collidees.

FILE FORMAT/DEFAULTS
    ColliderSeparator {
        renderCulling           AUTO;     # SFEnum
        owner                   "";       # MFString
        collider                ON;       # SFEnum
        collideGroupID          "";       # MFString
        colliderMode            BBOX;     # SFEnum
        colliderType            SEGMENT;  # SFEnum
        colliderGeometry        [];       # MFVec3f
        collisionFlag           FALSE;    # SFBool
    }


FILE FORMAT/DEFAULTS
     Separator {
        renderCulling           AUTO;     # SFEnum
        owner                   "";       # MFString
        collidable              ON;       # SFEnum
        collideGroupID          "";       # SFString
```

```
}
```

> ConstrainedField

An object for which you can restrict the variable ranges of the fields.

```
MINACTIVE/MAXACTIVE
    ON:   Restrict the variable range.
    OFF:

PAUSE
    When the field value crosses the boundary of the range,
    ON:   The field value remains at the crossing point.
    OFF:  The field value moves along the boundary.

FILE FORMAT/DEFAULTS
    MinMaxConstrainedField {
        field    "";      # SFString
        input             # type declared by "field"
        output            # type declared by "field"
        minActive         # MFEnum
        min               # type declared by "field"
        maxActive         # MFEnum
        max               # type declared by "field"
        pause    OFF;     # SFEnum
    }

FILE FORMAT/DEFAULTS
    ConstrainedField {
        field    "";      # SFString
        input             # type declared by "field"
        output            # type declared by "field"
        center            # type declared by "field"
        boundary          # SFFloat
        matrix   1 0 0 0 # SFMatrix
                 0 1 0 0
                 0 0 1 0
                 0 0 0 1;
    }
```

> Choose

An object for which you can choose between several inputs to the output. It may be used for parallel combination of engines. All inputs must have the same type of field.

```
FILE FORMAT/DEFAULTS
    Choose {
        fields      "";  # SFString
        whichInput  0;   # SFLong
        output           # type declared by "field"
    }
```

> Callback

Callback within the CLASS object.

```
EVENTTYPE
    PICK:     Execute a method when the object is picked.
```

```
    GRAB:      Execute a method when the object is held.
    DRAG:      Execute a method when the grabbed object is moved.
    RELEASE:   Execute a method when the object is released.
    COLLISION: Execute a method when the object is collided.
    MOVE:      Execute a method when the object is moved.
    RENDER:    Execute a method when that event type is received.
    WORLDIN:   Execute a method when a world is loaded.
    WORLDOUT:  Execute a method when a world is unloaded.
    ENTER:     Execute a method when a viewpoint enters an area.
    LEAVE:     Execute a method when a viewpoint leaves an area.
    USER:      Execute a method when a user-event is received.
    NONE:

Manipulation
    SET:
    COPY:
    ADD:
    MOVE:
    DELETE:

FILE FORMAT/DEFAULTS
    Callback {
        eventType               PICK;    # SFEnum
        eventSubType            "";      # SFString
        currentState            NONE;    # SFEnum
        actionObject            "";      # SFString
        actionNode              "";      # SFString
        delay                   0;       # SFFloat
        manipulation            SET;     # SFEnum
        parameters
        file                    "";      # SFString
    }
```

> MotionEngine
Data generator.

```
LOOP and INTERPOLATION
    ON:   Always perform motion looping or interpolation.
    OFF:
ACTIVE
    TRUE: The engine is active.
    FALSE:

FILE FORMAT/DEFAULTS
    MotionEngine {
        fields [ (field-type) (field-name),
                 ......
                 (field-type) (output-name)  ];
        time              0;       # MFFloat
        now               0;       # SFFloat
        file              "";      # SFString
        baseTime          0.0;     # SFFloat
        timeScale         1.0;     # SFFloat
        loop              OFF;     # SFEnum
        repetition        1;       # SFLong
        interpolation     OFF;     # SFEnum
        basisValue
        active            TRUE;    # SFBool
    }
```

> Include

An object for which you can refer to named instances of the included files at any place after they have been inserted.

```
FILE FORMAT/DEFAULTS
    Include {
        name    "";     # SFString
    }
```

> SharedSeparator

Object you can connect two scene graphs. If the shared node exists in two independent scene graphs, the browser interprets the node as one instance jointly owned by two graphs.

```
RENDERCULLING
     ON:    Always try to cull to the view volume
     OFF:   Never try to cull to the view volume
     AUTO:  Implementation-defined culling behavior

FILE FORMAT/DEFAULTS
     SharedSeparator {
        renderCulling   AUTO;   # SFEnum
        owner           "";     # MFString
        file            "";     # SFString
        rootName        "";     # SFString
        nodeName        "";     # SFString
        worldId         "";     # SFString
        connectPoint    0 0 0;  # SFVec3f
     }
```

> RootConnect

An object for which you can establish relations between the two roots of the scene graph.

```
FILE FORMAT/DEFAULTS
    RootConnect {
        root            "";       # SFString
        tranlation      0 0 0;    # SFVec3f
        rotation        0 0 1 0;  # SFRotation
        childRoot       "";       # SFString
}
```

> Audio

(SoundEmitter's format proposed by G. Bell, et al. is basically acceptable.)

```
FILE FORMAT/DEFAULTS
    Speaker ( or SoundEmitter ) {
        type            AMBIENT;  # SFEnum
        description     "";       # SFString
        intensity       1;        # SFFloat
        maxRange        -1;       # SFFloat
        minRange        0;        # SFFloat
    }
```

```
}
```

> ConstrainedField

An object for which you can restrict the variable ranges of the fields.

```
MINACTIVE/MAXACTIVE
    ON:   Restrict the variable range.
    OFF:

PAUSE
    When the field value crosses the boundary of the range,
    ON:   The field value remains at the crossing point.
    OFF:  The field value moves along the boundary.

FILE FORMAT/DEFAULTS
    MinMaxConstrainedField {
        field    "";      # SFString
        input             # type declared by "field"
        output            # type declared by "field"
        minActive         # MFEnum
        min               # type declared by "field"
        maxActive         # MFEnum
        max               # type declared by "field"
        pause    OFF;     # SFEnum
    }

FILE FORMAT/DEFAULTS
    ConstrainedField {
        field    "";      # SFString
        input             # type declared by "field"
        output            # type declared by "field"
        center            # type declared by "field"
        boundary          # SFFloat
        matrix   1 0 0 0 # SFMatrix
                 0 1 0 0
                 0 0 1 0
                 0 0 0 1;
    }
```

> Choose

An object for which you can choose between several inputs to the output. It may be used for parallel combination of engines. All inputs must have the same type of field.

```
FILE FORMAT/DEFAULTS
    Choose {
        fields      ""; # SFString
        whichInput  0;  # SFLong
        output          # type declared by "field"
    }
```

> Callback

Callback within the CLASS object.

```
EVENTTYPE
    PICK:     Execute a method when the object is picked.
```

IBM CONFIDENTIAL

```
FILE FORMAT/DEFAULTS
    Microphone ( or Ear ) {
        interval        1;        # SFFloat
    }
```

> RegionSensor
(This node proposed by G. Bell, et al. is basically acceptable.)

```
FILE FORMAT/DEFAULTS
    RegionSensor {
        bboxCenter  0 0 0;        # SFVec3f
        bboxSize    [ 1, 1, 1 ]; # MFFloat
        sense       "";           # SFString
        isIn        OFF;          # SFEnum
    }
```

from Open Inventor *

File, ElapsedTime

> (modified) Calculator

```
FILE FORMAT/DEFAULTS
    Calculator {
        fields [ (field-type) (field-name),
                 ......
                 (field-type) (output-name)  ];
        expression      "";     # MFString
    }
```

The supported arithmetic, logical and conditional operators are +, -, *, /, %, <, >, <=, >=, ==, !=, &&, ||, !, sin, cos, tan, etc.

---

from VRML 1.0

AsciiText, Cone, Coordinate3, Cube, Cylinder, DirectionalLight, FontStyle, Group, IndexedFaceSet, IndexedLineSet, Info, LOD, Material, MaterialBinding, MatrixTransform, Normal, NormalBinding, OrthographicCamera, PerspectiveCamera, PointLight, PointSet, Rotation, Scale, ShapeHints, Sphere, SpotLight, Switch, Texture2, Texture2Transform, TextureCoordinate2, Transform, TransformSeparator, Translation, WWWAnchor, WWWInline

---

```
SFColor {
    SFFloat r,
    SFFloat g,
    SFFloat b;
}

SFMatrix {
    SFFloat a11, SFFloat a12, SFFloat a13, SFFloat a14,
    SFFloat a21, SFFloat a22, SFFloat a23, SFFloat a24,
    SFFloat a31, SFFloat a32, SFFloat a33, SFFloat a34,
    SFFloat a41, SFFloat a42, SFFloat a43, SFFloat a44;
```

```
}

SFRotation {
    SFFloat x,
    SFFloat y,
    SFFloat z,
    SFFloat radians;
}

SFVec2f {
    SFFloat x,
    SFFloat y;
}

SFVec3f {
    SFFloat x,
    SFFloat y,
    SFFloat z;
}

SFVelocity {
    SFFloat x,
    SFFloat y,
    SFFloat z,
    SFFloat magnitude;
}

SFAcceleration {
    SFFloat x,
    SFFloat y,
    SFFloat z,
    SFFloat magnitude;
}
```

* Open Inventor is a trademark of Silicon Graphics Inc.

VRML 2.0 proposal top page

[Tokyo Research Laboratory home page | IBM Japan home page]

IBM CONFIDENTIAL

Example

```
#VRML V2.0 Japanese:sjis

VrlInclude {
    name "foo.wrl";
}

#
# If MyObject hits a Door, ...
# or if MyObject is picked, ...
# Scene1 is connected to Scene2.
#

SFColor myColorComponents = 1.0 0.0 0.0;

CLASS ReactiveObj VrlColliderSeparator {
    collider            ON;
    collideGroupID     "Door";
    colliderMode        BBOX;
    colliderType        SEGMENT;
    colliderGeometry  [ 1.0  0.5  0.0, -1.0  0.5  0.0,
                        1.0 -0.5  0.0, -1.0 -0.5  0.0 ];
    VrlMaterial { emissiveColor 0.1 0.3 0.3; }
    VrlCube {
        ...
    }
    VrlCallback {
        eventType       COLLISION;
        eventSubType    "";
        currentState    NONE;
        actionObject   "Door";
        actionNode     "MyFlyEngine1";
        manipulation    SET;
        parameters      active FALSE;
        file            "";
    }
    VrlCallback {
        eventType       PICK;
        eventSubType    "";
        currentState    NONE;
        actionObject    "";
        actionNode     "MyEngineSwitcher";
        manipulation    SET;
        parameters      whichInput 1;
        file            "";
    }
    VrlCallback {
        eventType       PICK;
        eventSubType    "";
        currentState    NONE;
```

```
        actionObject    "";
        actionNode      "this";
        manipulation    SET;
        parameters      emissiveColor myColorComponents;
        file            "";
    }
};

DEF MyFlyEngine1 VrlMotionEngine {   # Scheduled field
    fields [ MFVec3f coordinate, SFVec3f output ];
    coordinate [ 0.0 0.0 0.0,
                 1.0 2.0 3.0,
                 2.0 4.0 6.0,
                 ....        ];
    output       0 0 0;
    time       [ 0.0,
                 1.0,
                 2.0,
                 ...  ];
    baseTime        0.0;
    timeScale       1.0;
    loop            OFF;
    interpolation   ON;
    active          TRUE;
}

DEF MyFlyEngine2 VrlMotionEngine {
    fields [ MFVec3f coordinate, SFVec3f output ];
    coordinate [ 0.0  0.0 -0.0,
                 1.0  2.0 -3.0,
                 2.0  4.0 -6.0,
                 ....        ];
    output       0 0 0;
    time       [ 0.0,
                 1.0,
                 2.0,
                 ...  ];
    baseTime        0.0;
    timeScale       1.0;
    loop            OFF;
    interpolation   ON;
    active          TRUE;
}

DEF Scene1 VrlSeparator {

    SFVec3f lightLoc = 0 4 0;
    SFFloat cameraHeight = 1;

    VrlSeparator {
        DEF Door VrlSeparator {
            collidable      ON;
            collideGroupID "MyObject";
            # draw door here
            ....
        }
        VrlSeparator {
            DEF MyEngineSwitcher VrlChoose {
                fields     "SFVec3f";
                whichInput  0;
                input0  =  MyFlyEngine1.output;
```

```
                input1  =  MyFlyEngine2.output;
            }
            VrlTranslation {
                translation = MyEngineSwitcher.output;
            }
            DEF MyObject USE ReactiveObj;
        }
        DEF DoorToNext1 VrlSharedSeparator {
            file          "";
            rootName     "Scene2";
            nodeName     "DoorToNext2";
            worldId      "world2";
            connectPoint 0 0 0;
            VrlSeparator {
                # draw door here
                ....
            }
        }
    }
    VrlSpotLight {
        location       = lightLoc;
        direction        0 -1 0;
        intensity        0.9;
        cutOffAngle      0.7;
    }

    cameraHeight = cameraHeight * 2;

    VrlOrthographicCamera {
        position         0 0 1;
        orientation      0 0 1  0;
        focalDistance    5;
        height         = cameraHeight;
    }
}

#
# RedCube moves on a Floor. If RedCube moves onto a DoorPad, ...
# Scene2 connects to Scene1 via NextDoor.
#

DEF MyMoveEngine VrlMotionEngine {
    fields [ MFVelocity velocity, SFVec3f coord ];
    velocity [ 1.0  0.0  0.0  3.0,
               1.0  0.0  1.0  4.0,
               0.8  0.2 -1.0  8.0,
                    ...           ];
    coord      0 0 0;
    time     [ 0.0,
               1.0,
               2.0,
               ...  ];
    file           "";
         # or "filename" containing the scheduled field data
    baseTime         0.0;
    timeScale        1.0;
    loop             ON;
    repetition       5;
    interpolation    ON;
    active           TRUE;
}
```

```
DEF Scene2 VrlSeparator {

    SFVec3f transxyz = 0 0 0;

    VrlSeparator {
        DEF Floor VrlSeparator {
            collidable      ON;
            collideGroupID "RedCube";
            # draw floor here
            ....
        }
        DEF RedCube VrlSeparator {
            transxyz = MyMoveEngine.coord;
            if (MyMoveEngine.coord.y < 0)
                transxyz.y = 0;
            VrlTranslation {
                translation = transxyz;
            }
            VrlTranslation { translation 0 0.5 0; }
            DEF MoveOnFloor VrlColliderSeparator {
                collider          ON;
                collideGroupID   "Floor";
                colliderMode      BBOX;
                colliderType      SEGMENT;
                colliderGeometry [ 1.0  1.0  0.0, -1.0  1.0  0.0,
                                   1.0  0.0  0.0, -1.0  0.0  0.0 ];
                VrlMaterial { emissiveColor 1.0 0.1 0.1; }
                VrlCube {
                    width   2.0;
                    height  1.0;
                    depth   2;
                }
            }
        }
        DEF DoorPad VrlRegionSensor {
            # define a volume above the doormat for sensing
            bboxCenter   0 0 0;
            bboxSize     [20, 20, 20];
            sense        "RedCube";
            VrlSeparator {
                # draw a picture of the doormat here
                .......
            }
        }
        VrlSeparator {
            DEF DoorSwing VrlSwitch {
                whichChild = DoorPad.isIn;
                             # connect value of isIn field
                Rotation { rotation 0 1 0  0; }
                Rotation { rotation 0 1 0  1.5; }
            }
            # draw door here
             ......
        }
        DEF DoorToNext2 VrlSharedSeparator {
            file          "";
            rootName     "Scene1";
            nodeName     "DoorToNext1";
            worldID      "world1";
            connectPoint 0 0 0;
            VrlSeparator {
```

```
                # draw door here
                ....
            }
        }
    }
    VrlSpotLight {
        location        -4 4 0;
        direction       1 -1 0;
        intensity       1.0;
        cutOffAngle     0.7;
    }
    VrlSeparator {
        VrlTranslation { translation 0 1 0; }
        VrlWWWInline {    # Reference another object
                name "file:/next.wrl";
        }
    }
}

#
# MyHand opens a Door.
#

CLASS ReactiveDoor VrlSeparator {
    DEF KnobCollision VrlSeparator {
        collidable      ON;
        collideGroupID "MyHand";
        # draw door here
         ......
    }
    VrlCallback {
        eventType       GRAB;
        eventSubType    "";
        currentState    "";
        actionObject    "" ;
        actionNode     "MySwitch";
        manipulation    SET;
        parameters      whichchild 1;
        file            "";
    }
};

DEF PushDoorEngine VrlMotionEngine {
    fields [ MFRotation rotation, SFRotation output ];
    rotation [ 0 1 0  0.0,
               0 1 0  0.6,
               0 1 0  1.2,
               0 1 0  0.6,
               0 1 0  0.0 ];
    output     0 0 0;
    time     [ 0.0,
               1.0,
               2.0,
               3.0,
               4.0 ];
    baseTime        0.0;
    timeScale       1.0;
    loop            ON;
    repetition      -1;
    interpolation   ON;
    active          TRUE;
```

```
}

DEF MyDoorRotation VrlMinMaxConstrainedField {
    field       "SFRotation";
    input     = PushDoorEngine.output;
    output      0  1  0   0;
    minActive   ON ON ON  ON;
    min         0  1  0   0;
    maxActive   ON ON ON  ON;
    max         0  1  0   1.6;
}

DEF Scene3 VrlSeparator {
    VrlSeparator {
        VrlTranslation { translation 0 2 0; }
        DEF MySwitch VrlSwitch {
            whichchild  0;
            VrlSeparator {
                ....
                DEF MyHand VrlColliderSeparator {
                    collider          ON;
                    collideGroupID  "RotDoor";
                    colliderMode      BBOX;
                    colliderType      SEGMENT;
                    colliderGeometry [ 0.0  0.0  -1.0,
                                       0.0  0.0   1.0 ];
                    VrlMaterial { emissiveColor 0.8 0.5 0.1; }
                    {
                        # draw hand
                        ....
                    }
                }
            }
            VrlSeparator {
                VrlSeparator {
                    VrlRotation {
                        rotation = MyDoorRotation.output;
                    }
                    VrlTranslation { translation = 1 0 0; }
                    DEF RotDoor USE ReactiveDoor;
                    VrlTranslation { translation = 1 0 0; }
                    USE MyHand;
                }
            }
        }
    }
    VrlSpotLight {
        location        4 4 0;
        direction       -1 -1 0;
        intensity       0.9;
        cutOffAngle     0.7;
    }
    VrlOrthographicCamera {
        position        0.2 0 1;
        orientation     0.2 0 1  0;
    }
}

#
# Example of parallel combining engines
#
```

```
DEF MyClock VrlElapsedTime {
    speed     1;
    on        TRUE;
    pause     FALSE;
    output    0;
}

DEF MyCombiEngine1 VrlMotionEngine {
    fields [ MFRotation rotation, SFRotation output ];
    rotation [ 0 1 0  0,
               0 1 0  1,
               0 1 0  2,
               0 1 0  3,
               0 1 0  2,
               0 1 0  1 ];
    output     0 0 0  0;
    time     [ 0.0,
               1.0,
               2.0,
               3.0,
               4.0,
               5.0,
               6.0 ];
    now      = MyClock.output;
    loop               ON;
    repetition         -1;
    interpolation      ON;
    active             TRUE;
}

DEF MyTimeScaling VrlCalculator {
    fields     [ SFFloat input0,
                 SFFloat output ];
    input0     = MyClock.output;
    expression  "output = input0 * 0.5";
}

DEF MyCombiEngine2 VrlMotionEngine {
    fields [ MFRotation rotation, SFRotation output ];
    rotation [ 0 1 0  0,
               0 1 0  1,
               0 1 0  2,
               0 1 0  3,
               0 1 0  2,
               0 1 0  1 ];
    output     0 0 0  0;
    time     [ 0.0,
               1.0,
               2.0,
               3.0,
               4.0,
               5.0,
               6.0 ];
    now      = MyTimeScaling.output;
    loop               ON;
    repetition         -1;
    interpolation      ON;
    active             TRUE;
}
```

IBM CONFIDENTIAL

VRML 2.0 proposal top page

[Tokyo Research Laboratory home page | IBM Japan home page]

Last modified: Fri Feb 2 17:00:00 JST 1996

IBM CONFIDENTIAL

Background

In 1994-95, we developed a prototype virtual environment system at the National Cancer Center in Japan, whose ultimate goal is to facilitate training, planning, and rehearsing of surgical procedures for removing remove brain tumors. A user can perform a simulated surgical procedure, such as cutting a patient's skin, on the visualized anatomy. This viewpoint-tracked, immersive VR system allows interactive operation. The configuration consists of graphics workstations, a head-mounted display, a data glove, and a tracked stylus with auditory feedback.

References

1. R. Ohbuchi, T. Miyazawa, et al., "Integrated Medical Image System for Cancer Research and Treatment," IBM Journal of Research and Development, to appear, 1996.
2. R. Ohbuchi, M. Aono, and T. Miyazawa, "Synthetic Environment System for Surgical Planning and Training," IBM Research Report, RT5109, 1995.
3. G. Bell, A. Parisi, M. Pesce, et al., "The Virtual Reality Modeling Language - Version 1.1 Draft," http://vag.vrml.org/vrml-1.1.html.
4. Microsoft Corp., "ActiveVRML White Paper," http://www.microsoft.com/intdev/inttech/avwhite.htm.
5. Mitra, Y. Honda, et al., "Moving worlds: behaviors for VRML," http://reality.sgi.com/employees/gavin/vrml/Behaviors.html.
6. G. Bell, A. Parisi, and M. Pesce, "The Virtual Reality Modeling Language - Version 1.0 Specification," http://vrml.wired.com/vrml.tech/vrml10-3.html
7. J. Wernecke, "The Inventor Mentor," Addison-Wesley, 1994.
8. Open Inventor * Architecture Group, "Open Inventor C++ Reference Manual," Addison-Wesley, 1994.

* Open Inventor is a trademark of Silicon Graphics Inc.

VRML 2.0 proposal top page

[Tokyo Research Laboratory home page | IBM Japan home page]

[IBM home page | Order | Search | Contact IBM | Help | (C) | (TM)]

Last modified: Fri Feb 2 17:00:00 JST 1996