OCT 95

OCT

Simple behaviors for VRML

Gavin Bell, gavin@sgi.com
Chee Yu
Chris Marrin
Rob Meyers
Silicon Graphics

Our goal is to design a set of extensions to VRML that give the user a much richer experience than is possible in VRML 1.0. We believe that the ability to interact with intelligent objects with simple behaviors, the ability to create animated 3D objects, and the addition of sound into 3D worlds will provide the rich, interactive experience that will enable a completely new set of applications of VRML.

Goals

Our design was guided by the following constraints (in rough order of priority):

Performance
: We believe that speed is a key to a good interactive experience, and that it is important to design the system so that VRML browsers will be able to optimize the VRML world.

Scalability
: Our goal is to allow the creation of very large virtual worlds. Any feature which limits scalability is unacceptable, and in several areas of our design we have purposely made certain things difficult to encourage the creation of scalable VRML worlds.

Composability
: Related to scalability, we want to be able to compose VRML worlds or objects to create larger worlds. We assume that we will be able to compose worlds that are created by different people simply by creating a 'meta-world' that refers to the other worlds.

Authoring
: We assume that sophisticated VRML authoring tools will be created, and wish to make it possible to perform most of the tasks necessary to create an interesting, interactive VRML world using a graphical user interface. We believe that VRML will not be successful until artists and creative people that are not interested in programming are able to create compelling, interactive VRML content.

Power
: We must allow programmers to seamlessly extend VRML's functionality, by allowing them to create arbitrary scripts/applets/code that can then be easily re-used by the non-programmer.

Multi-user potential
: We expect VRML to evolve into a multi-user shared experience in the near future, allowing complete collaboration and communication in interactive 3D worlds. We have attempted to anticipate the needs of multi-user VRML in our design, considering the possibility that VRML browsers might eventually need to support synchronization of changes to the world, locking, persistent distributed worlds, event rollback and dead reckoning in the future.

We will not know if our goals and constraints have been met until we have implemented this system in both a browser and an authoring system. However, we have considerable experience with both browsers (WebSpace Navigator) and authoring systems (WebSpace Author) for VRML, and believe that this design will strike a good balance between speed for the VRML spectator, power for the VRML hacker and ease of use for the VRML artist.

Overview

The essence of simple behaviors is changes to the world over time. With a single spectator interacting with the world, we identify three sources of change:

1. User input from some device, translated into the 3D world by the browser.
2. Time.
3. Other input sources, accessed through some scripting/logic language, probably running asynchronously to the browser.

We are proposing a model for describing how changes are communicated between objects in the VRML scene. VRML's existing notion of a 'field' of an object is extended to allow 'inputs' and 'outputs'. Inputs and outputs can be thought of either as communication channels or messages that can be sent or received.

This change communication model is combined with a new prototyping capability that allows the encapsulation and re-use of objects, behavior, or both.

For some applications of VRML (such as database visualization, scientific visualization, and possibly shared, multi-user, distributed worlds) it is desireable to allow the creation and modification of VRML scene graphs. We address this need with a proposal that allows the VRML developer to either take complete control over changes to the scene, which may limit the opportunities a browser has for optimization, or to either limit their control to a portion of the scene or to limit their changes to value changes, without changing the structure of the scene or the communication paths between objects..

Specification

Adding events requires compiling VRML AND full access to the scene Graph & connection to ~~the~~ obj to add events to key in the ~~scene~~ logic code

Requires maintenance of ◇ scenegraph & logic as default value ◇ events handled by editing scene graph

No support for multi procedural applets so unclear how adding an event would work anyway

The following describe in detail what is proposed:

Connections

We propose that many of the current VRML nodes be redefined such that their fields are declared to be **inputs**. An input is merely a field which may be connected to an **output**. Connecting an input to an output establishes a communication pathway between two objects; the object containing the input will be told whenever the output changes, and may then change its own outputs, render itself differently, etc. Inputs and outputs are written using the same format as VRML fields.

An input of some node is connected to an output of some other node using the following syntax:

```
Node {
  input = Node{}.output
}
```

An input may have only one thing connected to it (**fan-in** is not allowed); the syntax chosen enforces this rule.

Note: We considered allowing fan-in, and defining automatic rules for combining multiple outputs into one value. Note that a "last-one-wins" rule does NOT work, because "last" is ill-defined; outputs may change at exactly the same time. We believe it better to require authors or authoring systems to handle situations in which fan-in is desireable by defining Logic nodes with multiple inputs that correctly combine the multiple input streams.

An output may be connected to several inputs (**fan-out** is allowed), simply by using VRML's existing multiple-instancing mechanism:

```
Node1 {
  input1 = DEF NAME Node{}.output
}
Node2 {
  input2 = USE NAME.output
}
```

DEF/USE may also be used to create **loops**:

```
DEF NAME Node {
  input = USE NAME.output
  output 10.5
}
```

Loops are well-defined, deterministic, and useful in this proposal.

If an input is not connected, it may be given a constant value. Inputs that are connected may not be given a value. Inputs that are not connected and are not given any value will have a default value.

Prototyping

Prototyping is a mechanism that allows the set of node types to be extended from within a VRML file.

PROTO

A prototype is defined using the **PROTO** keyword, as follows:

```
PROTO typename [ input fieldtypename name
                    IS nodename.fieldname nodename.fieldname ... ,
                 output fieldtypename name IS nodename.outputname,
                 ... ]
      node { ... }
```

A prototype is NOT a node; it merely defines a prototype (named '*typename*') that can be used later in the same file as if it were a built-in node. The implementation of the prototype is contained in the scene graph rooted by *node*. The implemenation may not be DEF'ed or USE'ed.

The input and output declarations export inputs and outputs inside the scene graph given by *node*. Specifying the type of each input or output in the prototype is intended to prevent errors when the implementation of prototypes are changed, and to provide consistency with external prototypes. Specifying a name for each input or output allows several inputs or outputs with the same name to be exported with unique names. Prototypes do not have fields, and fields inside the implementation may not be exported (only inputs and outputs).

The node names specified in the input and output declarations must be DEF'ed inside the prototype implementation. The first node DEF'ed in lexical (not traversal) order will be exported. It is an error (and results are undefined) if there is no node with the given name, or the first node found does not contain a field of the appropriate type with the given field name.

Prototype declarations have file scope, and prototype names must be unique in any given file.

A prototype is instantiated as if *typename* were a built-in node. A prototype instance may be DEF'ed or USE'ed. For example, a simple chair with variable colors for the leg and set might be prototyped as:

```
PROTO TwoColorChair [ input SFColor legColor IS leg.diffuseColor,
        input SFColor seatColor IS seat.diffuseColor ]
     Separator {
        Separator {
            DEF seat Material { diffuseColor .6 .6 .1 }
            Cube { ... }
        }
        Separator {
            Transform { ... }
            DEF leg Material { diffuseColor .8 .4 .7 }
            Cylinder { ... }
        }
    }
# Prototype is now defined.  Can be used like:
DEF redGreenChair TwoColorChair { legColor 1 0 0  seatColor 0 1 0 }

USE redGreenChair # Regular DEF/USE rules apply
```

Issue: definition of legal node type names is never specified. Should they be the same a legal node names?

Would the PROTO syntax be clearer if we added curly braces after the prototype name that surrounded the interface declaration and the implementation? E.G:

```
PROTO TwoColorChair { [ input ... IS ... ] Separator { ... } }
```

Note: PROTO sort of gives people their non-instantiating DEF: PROTO foo [] Cube { } is roughly equal to DEF foo Cube { }, except that foo is now a type name instead of an instance name (and you say foo { } to get another cube instead of USE foo). Smart implementations will automatically share the unchanging stuff in prototype implementations, so the end result will be the same.

NodeReference

What if we wanted a prototype that could be instantiated with arbitrary geometry? For example, we might want to define a prototype chair that allowed the geometry for the legs to be defined, with the default (perhaps) being a simple cylinder.

VRML 1.1 will include the SFNode field type-- a field that contains a pointer to a node. Using SFNode, it is easy to write the first part of the PROTO definition:

```
PROTO Chair [ input SFNode legGeometry
```

... but then we get stuck when we try to define the IS part of the prototype. We need some way of taking an SFNode input and inserting it into the scene. This can be accomplished with a new node, the NodeReference node:

```
NodeReference {
    nodeToUse NULL  # input SFNode (NULL is valid syntax for SFNode)
}
```

Functionally, NodeReference is a "do-nothing" node-- it just behaves exactly like whatever 'nodeToUse' points to (unless nodeToUse is NULL, of course, in which case NodeReference does nothing). For example, this would be a verbose way to add a Sphere to the scene:

```
NodeReference { nodeToUse Sphere { } }
```

NodeReference is only interesting if the nodeToUse input is connected to something. So, for example, our Chair with arbitrary leg geometry example can be filled out as:

```
PROTO Chair [ input SFNode legGeometry  IS  NR.nodeToUse ]
     Separator {
        Separator {
            Transform { ... }
            DEF NR NodeReference { nodeToUse Cylinder { } }
        }
        Separator {
            Transform { ... }
            USE NR
        }
        ... would re-use leg with a USE NR, would have
        geometry for seat/back/etc...
    }
```

Note that NodeReference allows a default value to be given for the input of a prototype, just like any other field type. Using the Chair prototype would look like:

```
Chair {
    legGeometry Separator { Coordinate3/IndexedFaceSet/etc }
}
```

It might also make sense to share the same geometry between several prototype instances; for example, you might do:

```
Chair {
    legGeometry DEF LEG Separator { Coordinate3/IndexedFaceSet/etc }
}
Chair {
    legGeometry USE LEG
}
```

Note that SFNode fields follow the regular DEF/USE rules, and that SFNode fields contain a **pointer to** a node; using DEF/USE an SFNode field may contain a pointer to a node that is also a child of some node in the scene graph, that is pointed to by some other SFNode field, etc.

The NodeReference node has nice, clean semantics, and allows a lot of flexibility and power for defining prototypes. It also has some nice implementation side effects:

Browsers that want to maintain a different internal representation for the scene graph can implement NodeReference so that nodeToUse is read and the different internal representation is generated. Optimizations might also be performed at the same time:

Browsers that optimize scene graphs can implement NodeReference such that whenever nodeToUse changes an optimized scene is created. When rendering, the optimized scene will be used instead of the unotimized scene.

A really smart browser will figure out that nobody is using the unoptimized scene and may free it from memory.

Something else to think about: should a prototype be allowed to expose the outputs or use the inputs of the contents of an SFNode that is passed in? E.g. could I define a prototype that took a node as input, assumed that the node had a "foo" output, and published the output of that node as an output of the prototype?

EXTERNPROTO

A second form of the prototype syntax allows prototypes to be defined in external files:

```
EXTERNPROTO typename [ input fieldtypename name,
                       output fieldtypename name, ... ]
            URL
```

In this case, the implementation of the prototype is found in the given URL. The file pointed to by that URL must contain ONLY a single prototype implementation (using **PROTO**). That prototype is then given the name *typename* in this file's scope (allowing possible naming clashes to be avoided). It is an error if the input/output declaration in the **EXTERNPROTO** is not a subset of the input/output declaration specified in *URL*.

Note: The rules about allowing exporting only from files that contain a single PROTO declaration are consistent with the WWWInline rules; until we have VRML-aware protocols that can send just one object or prototype declaration across the wire, I don't think we should encourage people to put multiple objects or prototype declarations in a single file.

We need to think about scalability when using nested EXTERNPROTO's. EXTERNPROTOS don't have bounding boxes specified like WWWInlines, and they might need them. I'm starting to think that we might need to add bboxCenter/Size fields to Separator instead of having them only on WWWInline; with animations possible, pre-specifying maximum-possible bounding boxes could save a lot of work recalculating bounding boxes as things move.

Logic: scripting

The Logic node encapsulates a pure executable function in VRML. It consists of a description of its fields, a script and the script language. The field description is written just after the opening curly-brace for the node and consists of the keyword 'fields' followed by a list of the access modifiers, types and names of fields used in square brackets and separated by commas. The 'script' field contains either the location of the script or the actual script itself. The 'language' field specifies the language of the script.

For example, a Logic node which is a latch that toggles its output value whenever both of its inputs are TRUE can be written like:

```
Logic {
```

```
            input  SFBool  isTwoOn,
            output SFBool  value,
            SFBool state]
    script "if (isOneOn && isTwoOn){
                state = !state;
                value = state;
            }"
    language "C"
    isOneOn FALSE isTwoOn FALSE value FALSE state FALSE
}
```

This allows the creation of:

1. Functions that read inputs, perform some calculation on them, and writes outputs. The functions will be executed whenever the inputs change. Examples: Logic nodes that are connected to time and perform keyframe animation, or that combine several values together to produce a composite output.
2. Scripts that spontaneously generate values, perhaps by communicating with a server of some kind. For example, a script might read stock prices from a server and make them available as outputs.
3. Scripts that communicate changes to their inputs to a server of some kind; for example, the movement of the viewer through the scene might be attached as an input to a script that then sends the positions to a multi-user motion server that then rebroadcasts all avatar's locations.
4. Scripts that maintain and edit part or all of the scene graph.

The Logic node is scripting-language neutral, and this document describes only the properties required of any scripting language, and describes in general terms what kind of functionality the scripting language / browser interface will need to support.

Authors are encouraged to keep Logic nodes simple. In particular, a browser will be better able to optimize the execution of behaviors if most Logic nodes are of types 1-3. Logic nodes that are pure functions-- whose outputs depend solely on the value of their inputs-- will be most efficient. Logic nodes that can arbitrarily edit parts of the scene graph will make it difficult for browsers to fully optimize worlds.

Logic nodes can maintain either state that the browser knows about ("public state", which is implemented simply as fields of the Logic node) or internal state that the browser is not aware of. Scalable implementations will load and unload Logic nodes as different parts of the virtual world are loaded and unloaded into and out of memory (we assume that virtual worlds will be larger than available disk or memory). To maintain the illusion of a continuous virtual world, when unloading part of the world browsers will need to save any state that has changed since that part of the world was loaded. For example, if the user walks into a room, turns the lights on, the walks out of the room, the browser must remember that the lights in that room are on even if the room is unloaded from memory (or disk). When the user later walks back into the room, the browser will read the room from disk (or over the network). If the on/off state of the lights is known to the browser (it is a field, input or output of a Logic node, an output of a sensor, etc) then all of this can happen automatically.

To make this saving/restoring process fast and small, authors should try to minimize the amount of public state. Any state that can be easily recreated should be stored as private state inside the Logic node. For example, a Logic node that output the last 100 trade prices for a specific stock on the stock market might have public state of:

```
Logic {
    tickerSymbol "SGI"
    stockServer "stock://www.sgi.com/stockfeed"
    # ... outputs an MFFloat that is last 100 prices
}
```

Given the stock server and stock symbol, the output can be recreated, so this is all the public state needed.

Evaluation Order

To guarantee that a behavior produces similar results on different VRML implementations, rules are defined for determining the evaluation order of Logic nodes.

Authors need a well-defined, consistent mental model of how behaviors interact. The challenge is to describe this model precisely enough so that authors can be confident that they are creating behaviors that will work across implementations, but also to allow enough freedom so that different implementations of behaviors are possible.

The mental model we have chosen consists of continuous changes over time and discrete events that occur at particular points in time. "Time" refers to an idealized notion of time corresponding to the everyday notion of time ("wall-clock" time in other systems).

Implementations will present the results of behaviors by sampling the behavior's output at a particular point in time and rendering the result. This sampling of continuous time is an implementation detail; the author has no control over it (implementations may choose to sample behaviors just before or during rendering, may choose to run a separate simulation thread that samples behaviors more frequently than the rendering, or might choose to temporarily suspend the sampling of behaviors to

maintain interactivity).

It is possible for authors to create behaviors that are dependent on how often behaviors are sampled. However, doing so will result in non-portable behaviors, and should be discouraged.

This section needs more work. We need to clearly define what authors can/can't assume when they write Logic nodes. For example, when one or more inputs of a Logic node changes, should an author be allowed to ask which inputs changed to cause the evaluation? Since, potentially, all inputs might change at the same time, is that a useful thing to know? It could potentially allow authors to optimize their scripts-- to know if something does/doesn't need to be recomputed. But they could just store and compare values to accomplish the same thing... Loop semantics also need to be clearly stated; is it possible to do that without severely restricting implementations?

Scripts with internal state cannot be optimized as much as scripts whose outputs depend solely on their inputs. The Browser/Script API should probably allow a script to specify "Please evaluate me _whenever_ this/these inputs change"; for example, if detecting double-clicks a script will care about every mouse click, but if starting an animation when the button goes up a script only cares about the last mouse click. We will realize significant savings if we can be lazy about evaluating scripts.

Asynchronous Logic

To support sources of input from outside the virtual world (Logic nodes that spontaneously produce output), it may be desireable to allow Logic nodes to run as asynchronous processes. In this case, the Logic nodes function very much like the built-in Sensor classes.

The evaluation model remains the same. The asynchronous process must be synchronized with the browser whenever it does any of the following:

- gets the value of an input
- gets the value of a field
- sets the value of a field
- sets the value of an output

The browser/script API will need at least the following methods:

start
: Some start or init method, to allow the script to startup an asynchronous process. The start method will need some way of returning information back to the browser about whether or not the browser should expect spontaneous changes and whether the script cares only about the final values of its inputs or whether it cares about intermediate changes (for example, if the implementation runs behaviors just before rendering, but rendering takes 1/10 of a second, if the input of a Logic node changes twice during that 1/10 of a second, does the Logic node care about both changes or only the final value?).

inputChanged
: When one or more inputs of a Logic node changes, some kind of notification/evaluation method must be called. The inputChanged method is synchronous, and is expected to perform some calculations, set outputs, and return. An asynchronous Logic node with inputs will probably have an inputChanged method that communicates with the asynchronous process, telling it that something has changed. If an asynchronous Logic node is not being used, then the inputChanged method will typically look like:

```
get state of inputs/fields
perform some calculation
set state of outputs/fields
```

end
: The browser will call the end method when the Logic node is destroyed. It will clean up anything allocated in the start method, including terminating any asynchronous processes started.

sync/unsync/set/get
: If running asynchronously, the evaluation loop will probably look something like:

```
do forever:
  wait for input from somewhere in the real world
  sync with browser
  get state of inputs/fields
  perform some calculation
  set state of outputs/fields, based on VRML and real-world inputs
  unsync with browser
```

: Compare this with the synchronous inputChanged method described above.

Exactly what all this looks like in the scripting language depends on which scripting language is being used. And exactly what this API looks like in the implementation of the browser and scripting language depends on which language the browser is implemented in, the calling conventions for the operating system, how asynchronous processes are implemented, etc.

Editing the scene

Combining the NodeReference and the Logic nodes results in scripts that can produce geometry or

edit parts of the scene graph. Combining this with the prototyping mechanism makes some pretty interesting things possible; for example, to define a Torus node you might:

```
PROTO Torus [ input SFFloat radius1 IS GenerateTorus.radius1,
              input SFFloat radius2 IS GenerateTorus.radius2 ]
  NodeReference {
    nodeToUse = DEF GenerateTorus Logic {
      fields [ input SFFloat radius1, input SFFloat radius2,
               output SFNode geometry ]
      radius1 1 radius2 1 geometry NULL  # Default values

      script "reads radius1/radius2, generates
              Separator/Coordinate3/IndexedFaceSet,
              writes Separator pointer to geometry output"
    }.geometry
  }
```

A Torus prototyped this was will look exactly as if it is a built-in node. New property nodes (based on existing properties) can also be defined in this way.

Because an SFNode field stores a pointer to a node, a Logic node may decide to maintain a pointer to a node it is outputting as part of either its public or internal state. This allows a Logic node to make arbitrary changes to one or more parts of the scene graph.

More implementation notes: for this to work, NodeReference will have to notice that scene graph that nodeToUse points to has been changed, and may need to re-optimize (or regenerate an internal rep) the next time it is rendered. Really smart optimizing browsers might keep track of how often a NodeReference is changing, and not bother doing the optimizations if it is changing too often-- authors should still try to minimize which parts of the scene they allow to be changed arbitrarily, since browsers with different internal representations will take a significant hit if the scene's that NodeReference nodes point to are constantly changing.

For example, a Logic node might receive scene graph changes from a server. The VRML file would look like:

```
Separator {
  NodeReference {
    nodeToUse = Logic { .... } . geometry
}
```

And the Logic node would have methods that did:

start
: Started up an asynchronous process to listen to the server (whose name/address might be a field or input of the Logic node), construct an initial scene graph based on information from the sever and set the geometry output to that scene. Also maintain a pointer to the scene so that the asynchronous process can:

asynchronous process
: Wait until the server sends information about changes to the scene, then:
Synchronize with the browser (making changes while it was in the middle of rendering would be bad)
Make changes to the scene
Tell the browser we're done making changes

end
: Exit asynchronous process cleanly.

Sensors: input

Proximity Sensors

Proximity sensors are nodes that detect when the camera enters and exits a defined region. A proximity sensor can be made inactive at any time by setting its enabled input to FALSE. An inactive sensor is reactivated by setting its enabled input to TRUE. The *enter* output contains the last time that the camera entered this volume, while the *exit* output contains the last time the camera exited this volume. While the camera remains within the region, the output isIn remains TRUE. There are two types of proximity sensors: VolumeProximitySensor and PointProximitySensor.

Issue: there are some other things we might like to know-- current position and orientation while the viewpoint is inside the region could be very useful. Providing position and orientation when the user is outside the region would kill scalability and composability, so those should NOT be provided (authors can create proximity sensors that enclose the entire world if they want to track the viewpoint wherever it is in the world). Position and orientation at the time the user entered/exited the world might also be useful, but I'm not convinced they're useful enough to add (and besides, you could write a Logic node with internal state that figured these out...).

VolumeProximitySensor

The VolumeProximitySensor node reports when the camera enters and leaves the volume defined by the fields center and size (an object-space axis-aligned box).

A VolumeProximitySensor that surrounds the entire world will have an enter time equal to the time that the world was entered, and can be used to start up animations or behaviors as soon as a world is loaded.

```
FILE FORMAT/DEFAULTS
     VolumeProximitySensor {

          input  enabled  TRUE  # SFBool
          output enter    0     # SFTime
          output exit     0     # SFTime
          output isIn     FALSE # SFBool
          center          0 0 0 # SFVec3f
          size            0 0 0 # SFVec3f
     }
```

PointProximitySensor

The PointProximitySensor reports when the camera enters and leaves the sphere defined by the fields center and radius.

```
FILE FORMAT/DEFAULTS
     PointProximitySensor {

          input  enabled  TRUE  # SFBool
          output enter    0     # SFTime
          output exit     0     # SFTime
          output isIn     FALSE # SFBool
          center          0 0 0 # SFVec3f
          radius          0     # SFFloat
     }
```

Pointing Device Sensors

A PointingDeviceSensor is a node which tracks the pointing device with respect to its child geometry. A PointingDeviceSensor can be made inactive at any time by setting its enabled input to FALSE. An inactive sensor is reactivated by setting its enabled input to TRUE. There are two types of PointingDeviceSensors; ClickSensor and DragSensors.

ClickSensor

The ClickSensor is triggered when the viewer points and clicks at its child geometry. The output enter contains the last time the cursor passed over any of the shape nodes contained underneath the ClickSensor, while the output exit contains the last time the cursor passed off of any of its contained shape nodes. While the cursor remains over any of the ClickSensor's children, the output isOver remains TRUE.

Issue: this is for locate-highlighting. Is that too much to ask, or should we assume fast picking from implementations?

If the user presses the button associated with the pointing device while the cursor is located over its children, the ClickSensor will grab all further motion events from the pointing device until the button is released. The output press contains the last time the button was pressed over the ClickSensor's children, while the output exit contains the last time the button was released after a grab by this ClickSensor. While the pointing device remains grabbed, the output isActive remains TRUE. Motion of the pointing device while it has been grabbed by a ClickSensor is referred to as a "drag".

As the user drags the cursor over the ClickSensor's child geometry, the point on that geometry which lies directly underneath the cursor is determined. When isOver and isActive are TRUE, three output values are updated with information about the point on the object beneath the cursor. The output hitPoint contains the 3D point on the surface of the underlying geometry, given in the ClickSensor's coordinate system. The output hitNormal contains the surface normal at the hitPoint. The output hitTexture contains the texture coordinate of that surface at the hitPoint, which can be used to index back into an image mapped onto the surface, to support the 3D equivalent of an image map.

```
FILE FORMAT/DEFAULTS
     ClickSensor {
          input  enabled     TRUE  # SFBool
          output enter       0     # SFTime
          output exit        0     # SFTime
          output isOver      FALSE # SFBool
          output press       0     # SFTime
          output release     0     # SFTime
          output isActive    FALSE # SFBool
          output hitPoint    0 0 0 # SFVec3f
          output hitNormal   0 0 0 # SFVec3f
          output hitTexture  0 0   # SFVec2f
     }
```

DragSensors

A DragSensor tracks pointing and clicking over its child geometry just like the ClickSensor; however, DragSensors track dragging in manner suitable for continuous controllers such as sliders, knobs, and levers. When the pointing device is pressed and dragged over the node's child geometry, the cursor's raw screen position is mapped onto idealized 3D geometry.

DragSensors extend the ClickSensor's interface; enabled, enter, exit, isOver, press, release and isActive are implemented identically. The outputs hitPoint, hitNormal, and hitTexture are only updated upon the initial click down on the DragSensors' child geometry. There are five types of DragSensors; LineSensor and PlaneSensor support translation-oriented interfaces, and DiscSensor, CylinderSensor and SphereSensor establish rotation-oriented interfaces.

LineSensor

The LineSensor maps dragging motion into a translation in one dimension, along the x axis of its local space. The output translation reflects the mapped drag position in local space.

```
FILE FORMAT/DEFAULTS
     LineSensor {
          input  enabled      TRUE  # SFBool
          output enter        0     # SFTime
          output exit         0     # SFTime
          output isOver       FALSE # SFBool
          output press        0     # SFTime
          output release      0     # SFTime
          output isActive     FALSE # SFBool
          output hitPoint     0 0 0 # SFVec3f
          output hitNormal    0 0 0 # SFVec3f
          output hitTexture   0 0   # SFVec2f
          minPosition         0     # SFFloat
          maxPosition         0     # SFFloat
          output trackPoint   0 0 0 # SFVec3f
          output translation  0 0 0 # SFVec3f
     }
```

minPosition and maxPosition may be set to clamp the output translation to a range of values as measured from the origin of the x axis. If minPosition is less than or equal to maxPosition, the output translation is not clamped. The output trackPoint always reflects the unclamped drag position along the x axis.

PlaneSensor

The PlaneSensor maps dragging motion into a translation in two dimensions, in the x-y plane of its local space. The output translation reflects the mapped drag position in local space.

```
FILE FORMAT/DEFAULTS
     PlaneSensor {
          input  enabled      TRUE  # SFBool
          output enter        0     # SFTime
          output exit         0     # SFTime
          output isOver       FALSE # SFBool
          output press        0     # SFTime
          output release      0     # SFTime
          output isActive     FALSE # SFBool
          output hitPoint     0 0 0 # SFVec3f
          output hitNormal    0 0 0 # SFVec3f
          output hitTexture   0 0   # SFVec2f
          minPosition         0 0   # SFVec2f
          maxPosition         0 0   # SFVec2f
          output trackPoint   0 0 0 # SFVec3f
          output translation  0 0 0 # SFVec3f
     }
```

minPosition and maxPosition may be set to clamp the output translation to a range of values as measured from the origin of the x-y plane. If the x or y component of minPosition is less than or equal to the corresponding component of maxPosition, the output translation is not clamped in that dimension. The output trackPoint always reflects the unclamped drag position in in the x-y plane.

DiscSensor

The DiscSensor maps dragging motion into a rotation around the z axis of its local space. The feel of the rotation is as if you were 'scratching' on a record turntable. The output rotation reflects the mapped drag position in local space.

```
FILE FORMAT/DEFAULTS
     DiscSensor {
          input  enabled      TRUE    # SFBool
          output enter        0       # SFTime
          output exit         0       # SFTime
          output isOver       FALSE   # SFBool
          output press        0       # SFTime
          output release      0       # SFTime
```

```
        output isActive     FALSE   # SFBool
        output hitPoint     0 0 0   # SFVec3f
        output hitNormal    0 0 0   # SFVec3f
        output hitTexture   0 0     # SFVec2f
        minAngle            0       # SFFloat
        maxAngle            0       # SFFloat
        output trackPoint   0 0 0   # SFVec3f
        output rotation     0 0 1 0 # SFRotation
    }
```

minAngle and maxAngle may be set to clamp the output rotation to a range of values as measured in radians about the z axis. If minAngle is less than or equal to maxAngle, the output rotation is not clamped. The output trackPoint always reflects the unclamped drag position in the x-y plane.

CylinderSensor

The CylinderSensor maps dragging motion into a rotation around the y axis of its local space. The feel of the rotation is as if you were turning rolling pin. The output rotation reflects the mapped drag position in local space.

```
FILE FORMAT/DEFAULTS
    CylinderSensor {
        input  enabled      TRUE    # SFBool
        output enter        0       # SFTime
        output exit         0       # SFTime
        output isOver       FALSE   # SFBool
        output press        0       # SFTime
        output release      0       # SFTime
        output isActive     FALSE   # SFBool
        output hitPoint     0 0 0   # SFVec3f
        output hitNormal    0 0 0   # SFVec3f
        output hitTexture   0 0     # SFVec2f
        minAngle            0 0     # SFVec2f
        maxAngle            0 0     # SFVec2f
        output onCylinder   FALSE   # SFBool
        output trackPoint   0 0 0   # SFVec3f
        output rotation     0 0 1 0 # SFRotation
    }
```

minAngle and maxAngle may be set to clamp the output rotation to a range of values as measured in radians about the y axis. If minAngle is less than or equal to maxAngle, the output rotation is not clamped.

Upon the initial click down on the DragSensors' child geometry, the hitPoint determines the radius of the cylinder used to map cursor input while dragging. The output trackPoint always reflects the unclamped drag position on the face of this cylinder, or in the plane perpendicular to the view vector if the cursor moves off of this cylinder. The output onCylinder is always TRUE at the initial click down, and is set to FALSE if the cursor is dragged off of the cylinder.

SphereSensor

The SphereSensor maps dragging motion into a free rotation about its center. The feel of the rotation is as if you were rolling a ball. The output rotation reflects the mapped drag position in local space.

```
FILE FORMAT/DEFAULTS
    SphereSensor {
        input  enabled      TRUE    # SFBool
        output enter        0       # SFTime
        output exit         0       # SFTime
        output isOver       FALSE   # SFBool
        output press        0       # SFTime
        output release      0       # SFTime
        output isActive     FALSE   # SFBool
        output hitPoint     0 0 0   # SFVec3f
        output hitNormal    0 0 0   # SFVec3f
        output hitTexture   0 0     # SFVec2f
        output onSphere     FALSE   # SFBool
        output trackPoint   0 0 0   # SFVec3f
        output rotation     0 0 1 0 # SFRotation
    }
```

The free rotation of the SphereSensor is always unclamped.

Upon the initial click down on the DragSensors' child geometry, the hitPoint determines the radius of the sphere used to map cursor input while dragging. The output trackPoint always reflects the unclamped drag position on the face of this sphere, or in the plane perpendicular to the view vector if the cursor moves off of this sphere. The output onSphere is always TRUE at the initial click down, and is set to FALSE if the cursor is dragged off of the sphere.

TimeSensor

The TimeSensor is a node which tracks the progress of real time. A TimeSensor remains inactive until its input startTime is set to the beginning of a desired time sequence. At the first simulation tick

where real time >= startTime, the TimeSensor will begin generating time outputs, which may be connected to the input of other nodes to drive continuous animation or simulated behaviors. The input cycleCount may be set to repeat a fixed number of intervals, each of duration cycleInterval; the output alpha varies between 0 and 1 over each interval. The input cycleMethod may be set to FORWARD, causing alpha to rise from 0 to 1, 0 to 1, over each interval, BACK which is equal to 1-FORWARD, or it may be set to SWING, causing alpha to alternate 0 to 1, 1 to 0, on each successive interval.

```
FILE FORMAT/DEFAULTS
     TimeSensor {
          input  startTime      0        # SFTime
          input  pauseTime      0        # SFTime
          input  cycleInterval  0        # SFTime
          input  cycleCount     1        # SFLong
          input  cycleMethod    FORWARD # SFEnum FORWARD | BACK | SWING
          output time           0        # SFTime
          output alpha          0        # SFloat
     }
```

The output pauseTime may be set to interrupt the progress of the TimeSensor; at the first simulation tick where real time >= pauseTime, outputs time and alpha will not be propagated. If pauseTime is reset to < startTime, outputs will resume on their original schedule, allowing it to be used like the hold on a stopwatch. If startTime is also reset to the current Time - pauseTime, outputs will resume where they were interrupted, allowing the TimeSensor to be used like the pause on a video deck.

If cycleCount is <= 0, the TimeSensor will continue to tick continuously, without a cycle interval; in this case, cycleInterval and cycleMethod are ignored, and the output alpha remains at 0. This use of the TimeSensor should be used with caution, since it incurs continuous overhead on the simulation.

Interpolators : animation

Interpolators are nodes that are useful for doing keyframed animation. Given a sufficiently powerful scripting language, all of these interpolators could be implemented using Logic nodes (browsers might choose to implement these as pre-defined prototypes of appropriately defined Logic nodes). We believe that keyframed animation will be common enough to justify the inclusion of these classes as built-in types:

Issue: do we want to do automatic SF/MF type conversion? Or define different interpolators?

RotationInterpolator

This node interpolates among a set of SFRotation values. The values field must contain exactly as many rotations as there are keyframe times in the keys field, or an error will be generated and results will be undefined.

```
FILE FORMAT/DEFAULTS
     RotationInterpolator {
          keys           [ ]        # MFFloat
          values         [ ]        # MFRotation
          input  alpha    0         # SFFloat
          output outValue  1 0 0 0  # SFRotation
     }
```

FloatInterpolator

This node interpolates among a set of SFFloat values. The values field must contain exactly as many numbers as there are keyframe times in the keys field, or an error will be generated and results will be undefined.

```
FILE FORMAT/DEFAULTS
     FloatInterpolator {
          keys           [ ] # MFFloat
          values         [ ] # MFFloat
          input  alpha    0  # SFFloat
          output outValue  0  # SFFloat
     }
```

ColorInterpolator

This node interpolates among a set of MFColor values, to produce an MFColor as output. The number of colors in the values field must be an integer multiple of the number of keyframe times in the keys field; that integer multiple defines how many colors will be produced in the outValue output. For example, if 7 keyframe times and 21 colors are given as input, each keyframe consists of 3 colors; the first keyframe will be colors 0,1,2, the second colors 3,4,5, etc. The values are interpolated in RGB space.

The description of outputting MF values belongs in the general interpolator section above, or maybe we should split up the interpolators into single-valued and multi-valued sections.

```
FILE FORMAT/DEFAULTS
     FloatInterpolator {
          keys            0 # MFFloat
          values          0 # MFColor
          input  alpha    0 # SFFloat
          output outValue 0 # MFColor
     }
```

PointInterpolator

This node interpolates among a set of SFVec3f values, doing linear interpolation. This would be appropriate for interpolating a translation of a transformation.

```
FILE FORMAT/DEFAULTS
     PointInterpolator {
          keys            0 # MFFloat
          values          0 # MFVec3f
          input  alpha    0 # SFFloat
          output outValue 0 # SFVec3f
     }
```

VectorInterpolator

This node interpolates among a set of SFVec3f values, suitable for transforming normal vectors..

```
FILE FORMAT/DEFAULTS
     VectorInterpolator {
          keys            0 # MFFloat
          values          0 # MFVec3f
          input  alpha    0 # SFFloat
          output outValue 0 # SFVec3f
     }
```

Need to add way to interpolate value of a Switch...

Examples

All of these examples are "pseudo-VRML" to save space (coordinates are not specified, etc). They also use a C or C++-like language for all Logic nodes.

A cube that changes color when picked

```
DEF CLICKSENSOR ClickSensor {
  DEF LOGIC Logic {
    fields [ input SFBool isBeingClicked, output SFColor color ]
    script "if (isBeingClicked) {
              color = Color(1,0,0);  // Red
            } else {
              color = Color(0,1,0);  // Green
            }"
    isBeingClicked = USE CLICKSENSOR.isActive;
  }
  Material {
    diffuseColor = USE LOGIC.color
  }
  Cube { }
}
```

The Logic node was defined as the first child of the ClickSensor. The colors it produces are hard-wired in the script, but could also have been provided as inputs:

```
  DEF LOGIC Logic {
    fields [ input SFColor color1, input SFColor color2,
             input SFBool isBeingClicked, output SFColor color ]
    script "if (isBeingClicked) color = color1; else color = color2;"
    isBeingClicked = USE CLICKSENSOR.isActive
    color1 1 0 0
    color2 0 1 0
  }
```

Also, the Logic node is defined to be a child of the ClickSensor for readibility; we could have made this example 20 bytes smaller by not bothering to DEF/USE the Logic node, but instead just putting it in the connection to diffuseColor:

```
    diffuseColor = Logic { fields[]/script/inputs/etc... } . color
```

Start a keyframe animation whenever a cube is picked

```
Separator {

  # This is the cube that will start the animation:
  DEF START ClickSensor {
    Cube { }
  }

  # And this is the object that will move:
  Separator {

    # For readability, I put the TimeSensor here:
    DEF TIMESENSOR TimeSensor {
      # Animate for 10 seconds starting whenever the button is
      # released from the cube:
      interval 10.0
      startTime = USE START.releaseTime
    }

    Transform {
      translation = PointInterpolator {
        keys [ .... ]
        values [ .... ]
        alpha = USE TIMESENSOR.alpha
      }.outValue
    }
    ... objects to be animated...
  }
}
```

This example just starts a 10-second keyframe animation that changes the translation of some objects when the user clicks and releases the mouse (or other pointing device) over the cube. To both animate and rotate the objects, we could just add a set of keyframes to the Transform's rotation field:

```
Transform {
     translation = PointInterpolator { ... }.outValue
     rotation = RotationInterpolator {
       keys [ ... ] values [ ... ]
       alpha = USE TIMESENSOR.alpha
     }.outValue
}
```

A Logic node could also be inserted between the Transform and the TimeSensor to (for example) start the animation 3 seconds after the cube is picked, or to only start an animation IF a button has been pressed or a puzzle has been solved, or to start a series of animations (e.g. animation #1 starts right away and lasts 14 seconds, animations #2 and #3 start 14 seconds from now and last 5 and 7 seconds, respectively, animation #4 starts when animation #2 ends (19 seconds from now), etc--arbitrary scheduling can be done using Logic nodes with multiple inputs/outputs).

A simple toggle-button prototype

This is a very simple implementation of a button that changes its output when clicked:

```
PROTO ToggleButton [ field initialState  IS  LOGIC.state,
                     output isOn  IS  LOGIC.output ]
  DEF CLICKSENSOR ClickSensor {
    DEF LOGIC Logic {
      fields [ input SFBool toggle, SFBool state, SFBool output ]
      toggle = USE CLICKSENSOR.isActive
      script "if (toggle) state = !state; output = state"
    }
    ... Geometry of button is here...
  }

... later, to use a toggle button:
ToggleButton { initialState FALSE }
```

This is overly simple; a real implementation would also have a Switch node that changed the button's appearance depending on its state. A good implementation would also use the isOver output of the ClickSensor to locate-highlight the button.

HSV color space Material node

This prototype demonstrates why considering properties first-class objects is powerful. An HSV equivalent of the Material node is defined:

```
PROTO HSVMaterial
  [ input MFColor ambientColor IS CONVERT.ambientIn,
    input MFColor diffuseColor IS CONVERT.diffuseIn,
```

```
    input MFColor specularColor IS CONVERT.specularIn,
    input MFColor emissiveColor IS CONVERT.emissiveIn,
    input MFFloat shininess IS M.shininess,
    input MFFloat transparency IS M.transparency ]
# Implementation: Material hooked up to Logic:
  DEF M Material {
    ambientColor = DEF CONVERT Logic {
       fields [
         input MFColor ambientIn,
         output MFColor ambientOut,
         ... etc...
       ]
       ambientIn .2 .2 .2   # Define default value
       ... etc ...
       script "
           Script's inputChanged method does:
           read ambientIn/diffuseIn/specularIn/emissiveIn,
           for each do:
             for all values:
               convert to HSV
               set corresponding output
           done."
     }.ambientOut
    diffuseColor = USE CONVERT.diffuseOut
    emissiveColor = USE CONVERT.emissiveOut
    specularColor = USE CONVERT.specularOut
  }
```

Note that attaching a ColorInterpolator (which interpolates in RGB space) to an HSVMaterial gives color interpolation in HSV space (the "RGB" values that are being interpolated will be translated into HSV values as they're interpolated by the HSVMaterial's logic).

Useful properties such as a camera defined by a look-at and a look-up direction, or a Transform whose rotation is defined by yaw/pitch/roll values could also be implemented this way.

Talking to a multi-user Avatar motion server

There are two main parts to this example: Reporting "our" position to the Avatar server, and getting Avatar geometry and the positions of all Avatars from the server. The overall structure of the world looks like:

```
#VRML V1.1 utf8
Separator {
    DEF PositionReporter VolumeProximitySensor {
       ...
    }
    Logic {
       input currentPosition = USE PositionReporter.currentPosition
       script ...
    }
    ...AvatarsGeometry...
    Separator { ... geometry of the world... }
}
```

So, first, the task of reporting "my" position to the server is accomplished by using a VolumeProximitySensor that is big enough to surround the entire world, with my currentPosition inside that world-sized region being reported to the server by using a Logic node. The Logic script just has an InputChanged method that reads the current position and sends it to the server (that's all assuming that your scripting language supports that kind of communication). Details are left as an exercise for the hacker.

The second part of the problem is displaying the avatars. To make this easier to read, lets define a "FromServer" prototype that lets us refer to a node that we get from the server:

```
PROTO FromServer [ field SFString objectName IS L.objectName,
                   field SFString serverAddr IS L.serverAddr ]
    NodeReference {
       nodeToUse = DEF L Logic {
          fields [ SFString objectName, SFString serverName,
                   output SFNode geometry ]
          objectName "" serverName ""
          script "....something..."
       }.geometry
    }
```

really expensive

The script would ask the server (given in serverName) for the geometry for some object (whose name is given in objectName). If that object might change over time, the script will start an asynchronous process that changes the geometry whenever it gets a change message from the server.

So, in our example, the "...AvatarsGeometry..." would actually simply be:

```
FromServer {
    objectName "AllAvatars"
```

```
    serverName "...some protocol/address..."
}
```

When an Avatar moves, the server might tell the script to just modify some transformation in the AllAvatars scene graph. That will work well for browsers that don't optimize the scene and that use the scene graph as their internal representation; it won't work very well for other browsers. To be most efficient for those other browsers, the AllAvatars scene graph could itself contain FromServer NodeReference objects, like this:

```
# AllAvatars scene created by script (from server info):
Separator {
    # Avatar 1
    Separator {
        FromServer {
            objectName "Avatar1Transform"
            serverName "...vrtp://www.sgi.com/Q_14_12_server"
        }
        ... geometry for avatar 1
    }
    # Avatar 2
    Separator {
        FromServer {
           objectName "Avatar2Transform"
           serverName "...."
        }
    }
    ... etc, for each avatar in this world
}
```

Now, the server will frequently generate changes to the Avatar#Transform objects, but will rarely generate changes to the top-level AllAvatars object (e.g. perhaps that changes only when somebody decides to change how they look, or when a new avatar enters/leaves the world). Using NodeReferences in this way to mark the parts of the scene that will change often will allow implementations to be smarter about their optimization or conversion to their internal formats.