OCT 95 ? > + Gavin Bell (gavin@sgi.com) UCD
Chee Yu
Rob Meyers — Silicon Graphics
Chris Marrin

Simple behaviors

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.

Needs to specify API
~~Gavin~~ mentioned "inputchanged" event.

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 parts of our design we have purposely made certain things difficult ~~to achieve~~ to encourage the creation of scalable VRML worlds.

Composability
: Related to scalability, we want to be able to compose VRML worlds or objects ~~together~~ 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 ~~from a~~ 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 eventually anticipate the needs of multi-user VRML in our design, considering the possibility that VRML browsers might 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 ~~our~~ 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.

("=USEfor.bw) CONNECTIONS — GOOD
PROTOTYPE — Good
INTERPOLATOR — Good
SENSORS BAD
LOGIC BAD

Overview

Monolithic Script.
Script run whenever inputs changed
~~Events~~
No Events
Hard wiring

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, possibly running asynchronously to the browser.

We are proposing a data flow model for describing how those changes propagate into the scene, with objects in the scene connected to input sources and to arbitrary pieces of logic that can operate on any number of inputs to produce any number of results.

This data flow model is combined with a new prototyping capability that allows the encapsulation and re-use of ~~scene graph~~s, behavior~~s~~ or both.
objects

Scene optimizations possible

does not include any mechanisms

Our proposal ~~provides no~~ way to dynamically change the structure of the scene graph or to dynamically change the structure of the connections in the data-flow graph. We believe that allowing such unlimited functionality ~~would~~ severely restricts the set of ~~optimizations a browser will be able to perform~~, would force browsers to maintain the scene graph structure internally, and would require a high level of expertise for VRML authors to create high-performance, scalable, composable worlds.

If changing the structure of the scene graph or they structure of the wiring between nodes in the scene graph is found to be necessary then that functionality can be easily added at a later time.

Specification

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**. Inputs and outputs are written in exactly 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.

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
```

believe it better to.. ~~force~~ require

We considered allowing fan-in, and defining automatic rules for combining multiple outputs into one value (for example, an automatic "choose smallest" rule when wiring the values '3', '7', and '10' into an integer field ~~might~~ would result in an input of '3'). However, we ~~feel that forcing~~ authors or authoring systems to define fan-in rules by replacing fan-in with logic nodes with multiple inputs and a single 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 ~~required~~ well-defined, deterministic, and useful.

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.

Sensors: input

Proximity Sensors

Proximity sensors are nodes that are triggered 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 output enter contains the last time that the camera entered this volume, while the output exit 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.

VolumeProximitySensor

The VolumeProximitySensor node is triggered whenever the camera enters and leaves the volume defined by the fields center and size.

```
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 is triggered whenever the camera enters and leaves the sphere defined by the fields center and radius.

```
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
```

Evaluation Model

Reasons: Determinism. Behaviors run same on all browsers

Rules:

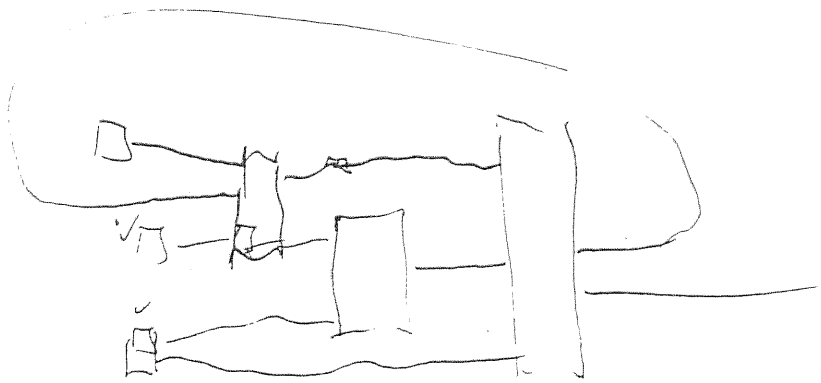
1. multiple changes to input sources may occur between evaluations
2. Logic evaluated at most once during an evaluation process
3. Logic evaluated only if some input changed
4. All inputs evaluated before Logic evaluated (and anything they depend on)
5. Loops are broken ~~only~~ during eval to avoid violating rule 2
6. Logic nodes with irrelevant outputs (defined by the browser) need not be executed.

Possible implementation:

? Are these rules overly restrictive?

Do we constrain implementations too much?

~~Do we want~~ Is some indeterminism OK?



```
        size            0 0 0 # SFVec3f
    }
```

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; ClickSensors 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.

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.

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. A script may either be ASCII or hex. The 'language' field specifies the language of the script.

When a Logic node receives input values, it executes the function perscribed by the script field, and then sends the results out on one or more outputs. A script is a pure function, i.e. it must not have side-effects on the scene graph - it only has access to the fields of its enclosing Logic node. A script cannot have static variables. Use the fields of the Logic node for static storage.

- *By disallowing side-effects, we are ensuring that the browser knows at all times when something has changed in the world and to correctly propagate these changes. Side-effects allow a script to make changes in the world without the knowledge of the browser. If side-effects are allowed then the script must notify the browser each time it cause a change to the world via a side-effect.*
- *By disallowing static variables in a script, we are requiring that any value that persists between invocations of the script be explicitly declared in the Logic node. This makes it easier for the browser to save and restore the state of a world. Otherwise, each script will have a support a serialization method that can be called by the browser.*

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 {
   fields [ input  SFBool  isOneOn,
            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
}
```

Interpolators : animation

RotationInterpolator

This node interpolates among a set of SFRotation values.

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

FloatInterpolator

This node interpolates among a set of SFFloat values.

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

ColorInterpolator

This node interpolates among a set of SFColor values.

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

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
     }
```

Prototyping

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

field is

```
PROTO twoColorChair [ input SFColor legColor IS leg.diffuseColor,
        input SFColor seatColor IS seat.diffuseColor ]
     Separator {
        Separator {
            DEF seat DynamicMaterial { 9 diffuseColor 1 0 0 }
            Cube { ... }
        }
        Separator {
            Transform { ... }
            DEF leg DynamicMaterial { }
            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: Can we pass in "plug-in" scene graphs using input SFNode ...? Should we extend USE to handle USE prototypename.inputname? If so, what happens if that input isn't specified; do we need a mechanism for specifying the default value of an SFNode input?

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

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.

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.

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:

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

A simple toggle-button prototype

```
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:
DEF TB ToggleButton { initialState FALSE }
```

~~Switch {~~

~~whichChild~~

Point Light {

on = USE TB.isOn

}

Fan-in prototype for combining translations

```
PROTO TranslationCombiner [ input SFVec3f t1 IS AVERAGE.t1,
                            input SFVec3f t2 IS AVERAGE.t2,
                            input SFVec3f t3 IS AVERAGE.t3,
                            input SFVec3f t4 IS AVERAGE.t4,
                            output SFVec3f result IS AVERAGE.RESULT ]

Logic {
    fields [ t1... t4  result ]
    t1  0 0 0
    t2  0 0 0
    t3  0 0 0
    t4  0 0 0
    result 0 0 0
    script "result= (t1+t2+t3+t4)/4;"
}
```

Note: it might be nice to be able to define an arbitrary number of inputs/outputs of a given type; e.g.

```
[ input SFVec3f t[] ]

t[] = 0 0 0;
script "Vec3f ~~result~~ sum
        ~~result~~ = Vec3f(0,0,0); for(int i=0; i< t.num(); i++)
                                   ~~result~~ sum += t[i];
        ~~result /= t.num()~~;
        result= sum/ t.num();"
```

However, I'm not convinced the extra implementation effort that this requires makes it worthwhile.

Allowing arbitrary types would also be nice, but that would require ~~yet m~~ even more syntax and more implementation.

These features could always be added to the language later

Gavin's Homework

– yes, I did all this while you were all out drinking beer. And yes, I do expect you to feel sorry for me, and to be swayed not only by my arguments but also by my demonstrated dedication to making VRML behaviors as useful as possible but also optimizable.

(and no, I'm not serious, that's a joke...)

Modifiable scene graphs:
SFNode as an output type

SFNode is a field type that contains a <u>pointer to</u> a node. (not part of VRML 1.0, <u>is</u> part of OpenInventor)

DEF/USE is used to share the same node between ~~two~~ different SFNode fields.

I propose: allow scripts to produce scene graphs by allowing them to have an SFNode output.

Missing piece: how do we insert an SFNode output from a script into the right place in the scene graph?

Proposal:

NodeReference, a new node:

NodeReference {

SFNode	input	nodeToUse	NULL
type		name	default value (NULL is special syntax for SFNode fields)

}

Functionally, NodeReference is a "do-nothing" node — it just behaves exactly like "nodeToUse". 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 that produces an SFNode output. For example, given two radii as input, you might write a script that generates a scene graph that is a Torus:

```
DEF GenerateTorus Logic {
    SFFloat input radius1 1
    SFFloat input radius2 .25
    SFNode output geometry NULL

    Script ~> script reads radius1 & radius2, creates
                Separator / Coordinate3 / IndexedFaceSet,
                and sets geometry to pointer to Separator
}
```

I'm being lazy here and using abbreviated syntax, giving type, fieldname & value all on one line...

The GenerateTorus Logic would then be used by a NodeReference:

```
NodeReference {
    nodeToUse = USE GenerateTorus.geometry
}
```

Even better, we could put the NodeReference and Logic in a prototype to define something that looks like a built-in Torus node:

this is the GenerateTorus Logic node

```
PROTO Torus [ input SFFloat radius1 IS GT.radius1,
              input SFFloat radius2 IS GT.radius2 ]

    NodeReference {
        nodeToUse = DEF GT Logic {
                ... the guts of GenerateTorus from
                        previous page ...
                }.geometry
    }
```

— That's it. To use after prototype is defined:

```
Torus {
    radius1  3.1415
    radius2  1.57
}
```

Implementation note:

Browsers that optimize scene graphs can implement NodeReference such that whenever it's input changes an optimized scene is created. When rendering, the optimized scene will be used instead of the unoptimized scene.

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

- hacker homework: figure out how reference counting for SFNode inputs or outputs and the NodeReference must work to get this to happen. Hint: remember that inputs and outputs are read-only and write-only... and you ~~may~~ can assume that NodeReference can figure out what it is connected to and play games with reference counts...

Generating nodes from a script I believe it is theoretically possible to do anything. For example, Avatars from a server might be done as:

```
Separator {
    NodeReference { #Avatars from server
        nodeToUse = FromServer {
            objectName "AllAvatars"
            Server "192.26.51.111:1080"
        }.geometry
    }
    ... rest of world ...
}
```

[whatever your protocol needs...

The scene returned from the server would look like:

```
Separator {
    #Avatar 1
    Separator {
        NodeReference {
            nodeToUse = FromServer {
                objectName "Avatar1Transform"
                Server "some gobbeldygook or URL or..."
            }.geometry
        }
        ... Avatar Geometry ...
    }
    .. etc, avatar 2 .... n
}
```

Note that the AllAvatars node reference can still be optimized, but that the Avatar transformations are <u>themselves</u> NodeReferences - which <u>cannot</u> be optimized, and must be maintained.

So, when an avatar's position changes, only the AvatarTransform is regenerated (remember, outputs are write-only). The AllAvatars NodeReference can be smart and won't need to regenerate/optimize the avatars scene graph until the server changes ~~something~~ the geometry of an avatar or adds a new avatar.

Side note: if those kinds of changes are common, we could add another level of NodeReference and have the server return:

```
Separator { # all avatars
    NodeReference { # first avatar
        nodeToUse = FromServer {
                        objectName "Avatar1Geometry"
                        server ....
                3. geometry
    3 ...etc
```

Side note 2:

The prototype for "FromServer" looks like:

```
PROTO FromServer [ field SFString objectName IS script.objNam
                   field SFString server IS script.server,
                   output SFNode geometry IS
                       script.geometry ]
```

Implementation:

```
DEF script Logic {
        field SFString objectName ""
        field SFString server ""
        Script
```

Script would have

```
init() {
    - get objectName + server field values, ERROR if haven't been set
    - connect to server
    - based on info from server, create geometry
    - set geometry output, de-reference geometry
    - if server may send stuff later, create thread
        (or register socket to listen on + callback w. browser
         or whatever) that generates new geometry
}
```

cont'd side note 2 :

- Thread or callback routine would do :

  When there's update from server :

  - generate geometry
  - ~~ask browser to set output~~
  - get some kind of browser synchronization lock
  - set output(s)
  - release lock

<u>BUT</u> what if you want your script to <u>modify</u> the scene graph?

Well...

No problem, you got it!

In the init() method at the bottom of page 8, I wrote:

- set geometry output, <u>de-reference</u> geometry

If we allow the script to maintain a reference to the geometry it created, then the thread or callback routine handling input from the server can do:

- get an edit-synchronization lock (actually, only needed if separate thread - if a callback registered w. browser don't need...)
- edit geometry created in init() method
  - I'm not going to try to specify the editing API. Might be just get named nodes and set their fields, or get prototyped nodes and set fields, or arbitrary Inventor-like operations on the scene...
- release lock (if asynchronous thread)

More implementation notes:

Remember, an SFNode field/input/output contains a <u>pointer to</u> a node. So when we do:

```
Node *node = new...
node->ref();
output = node; // set output to node
```

... in our scripts, <u>no copying</u> needs to take place. Both the script and the output will point to the same node (in our implementation of SFNode fields, anyway).

When this output is used in a NodeReference, NodeReference must notice that somebody else has a reference to the unoptimized scene - and must <u>not</u> free that scene from memory.

If it <u>does</u> optimize the scene, then both the optimized and non-optimized will be in memory, and

NodeReference must know to re-generate the optimized scene when the non-optimized changes (Inventor has node and field sensors that would be used ~~in~~ for this if using Inventor, for example).

So, sci-viz guys, you ~~will~~ may want to write your scenes like this:

```
#VRML V?.? utf8
Separator {
   NodeReference {
      nodeToUse = Logic {
         output SFNode geometry NULL
         Script
      }.geometry
   }
}
```

Script creates & maintains scene graph, performing arbitrary edits on it, perhaps based on input from server...

— COMPLETE CONTROL —

But probably not, you'll probably only want to control/edit parts of the scene so the browser can optimize the rest.

Advantages I see:

- Optimization possible
- Range of control, from little control / lots of possible optimizations to lots of control / restricted optimizations

Doing multi-user/avatars behind the browser's back:

- All we're missing is a way of sending reports on the current camera position to the server. The VolumeProximitySensor (see page 4 of my other proposal) is close to what we need; assume we add one more output:

  output curPosition 0 0 0 #SFVec3f

... that gives the viewer's position in the world. We'll wire that to some Logic that sends viewer-moved messages to the server:

when the viewer is inside the region being sensed

```
DEF VPS VolumeProximitySensor {
        center ... center of world...
        size ... size of world...
}
Logic {
    input Position = USE VPS.curPosition
    script → reads Position & sends to server
}
```

So the World is:

Separator {

PROTO FromServer ... see p. 8

Avatars ... see p. 6

VolumeProxSensor + Logic ... see p. 14

⋮

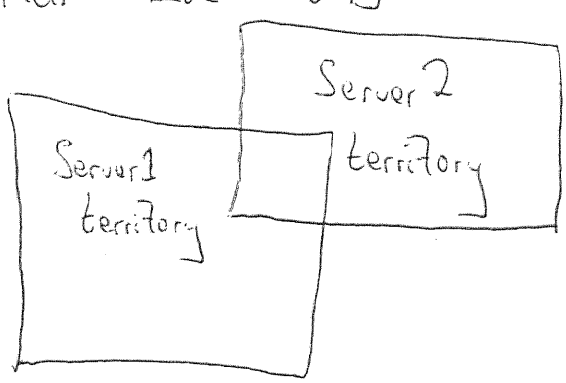
geometry

}

Oh, yeah, the best part is —

THIS IS SCALABLE!!

Just repeat the above scene graph to get different Avatar servers for different parts of the world. In 2D:



Overlap is probably desireable so you get avatars from both sub-worlds as you travel between worlds.

I'm too tired to figure out what happens if you combine all this with LOD.... but I bet it's cool.

And thank you all for sharing your thoughts ~ I never would have ~~gone~~ thought of this stuff if you hadn't pushed so hard...