

New York University Media Research Lab

Improv VRML API Specification

3:21 PM 5/1/97

Table of Contents

◆ Preface	
◇ Intended Audience	3
◇ Conventions	3
◆ Overview	
◇ The Improv System	4
◇ Why VRML and Java?	4
◇ What Can You Do With Improv?	5
◇ A Conceptual Sketch of Improv	5
◇ Sequences	6
◆ Getting Ready to Author	
◇ System Requirements	7
◇ How to Set Up the HTML File	8
◇ How to Set Up the VRML File	8
◆ Authoring In Improv	
◇ Structure of Improv	9
◇ How Channel Values Are Computed	11
◇ Authoring Example	15
◇ Reusing Nodes	23
◇ Controlling Improv From Other Nodes	23
◆ Specification of Nodes	
◇ Legend for Node Diagrams	24
◇ Improv node	25
◇ Actor node	26
◇ Channel nodes	28
◇ Layer node	31
◇ Sequence node	34
◇ Trigger node	37
◇ RandomTrigger node	38
◇ Hold node	39
◇ Repeat node	40
◇ State node	41
◇ Filter node	44
◇ Controller nodes	46
◇ Curve nodes	50
◇ Function nodes	53
◆ Examples	
◇ Simple Actor	56
◇ Simple Sequence	62
◆ Appendix	
◇ The Improv Server	63
◇ References	63

Preface

Intended Audience

This VRML API specification is written for a person familiar with authoring in VRML 2.0.

Conventions

The following type conventions are used:

Bold type in the body type style	names of nodes, fields, eventIns, etc.
Regular type in the monospace type style	VRML code, field values, HTML code
<i>italic type</i> in the body type style	important information
<i>italic type</i> in the monospace type style	a general term to be replaced with a specific file name or node type

The standard VRML naming conventions are used:

NODE : Actor	node names are capitalized
FIELD : timeScale	field names begin lowercase and all following words are capitalized

Overview

This document covers the VRML API for the VRML/Java implementation of the Improv Animation Engine. It assumes a working knowledge of VRML. Below you'll find a description of the system, a specification of the nodes used to apply the techniques of Improv to VRML models, and some examples.

The Improv System

The Improv system creates distributed 3D virtual environments in which human-directed avatars and computer-controlled agents interact with each other in real-time, through a combination of Procedural Animation and Behavioral Scripting techniques. The system is intended to operate over Local and Wide Area Networks using standard Internet protocols, enabling anyone with access to the World Wide Web to develop or participate in fully interactive, virtual experiences.

Improv provides tools to create actors that respond to users and to each other in real-time, with personalities and moods consistent with the author's goals and intentions.

Improv consists of two subsystems. The first subsystem is an Animation Engine that uses procedural techniques to enable authors to create layered, continuous, non-repetitive motions and smooth transitions between them. The second subsystem is a Behavior Engine that enables authors to create sophisticated rules governing how actors communicate, change, and make decisions. The combined system provides an integrated set of tools for authoring the "bodies" and "minds" of interactive actors.

The VRML API for the first subsystem, the Animation Engine as it is implemented using VRML and Java, is described here.

Why VRML and Java?

For Improv, VRML provides a stable, platform neutral rendering engine which also allows two way communication for user activity. Java provides a stable, platform independent animation computing engine. We have tried to exploit the established VRML and Java connection to its fullest.

VRML 2.0 Script nodes allow the developer to create nodes that are described by Java classes using the VRML 2.0 Java API. Scripts enable authors to create extensibility such as behaviors, communication, and as in this case, sophisticated animation.

Java's networking classes aid in creating applets that generate VRML code based on other data on the network. This results in a great potential for the automated generation of multi-user VRML worlds.

Additional features which make VRML and Java attractive include VRML's compact encoding for quick network transport and both VRML's and Java's support of an open, platform-independent specification. The VRML and Java combination results in executable content, optimized for low bandwidth environments.

What Can You Do With Improv?

VRML is significantly limited by its linear interpolation driven animations. Animations look stiff and mechanical. Using Improv, you can create sophisticated, lifelike animations controlled by complicated curves and functions. You can have several animations layered on top of each other and active simultaneously, and define responsive and flexible transitions and composite behaviors for your animations.

A Conceptual Sketch of Improv

Improv is designed as an animation compositing machine. Animations are defined using a structure of Actors, Channels, Layers, States, Filters, and Controllers. Don't worry about how to create these objects just yet, we're trying to give you an overview of how to approach the system.

At the top level is the Actor. The Actor holds everything used to control the Animation Engine. Generally, you will have one Actor per model.

Actors contain a list of Channels. Channels enable the animator to set up an arbitrary number of eventOuts for the Animation Engine's results. The results contained in the Channels are routed to the VRML model by way of author-defined routes. The author must define a Channel for each aspect of the model they want to animate.

Actors also contain a list of Layers. Layers hold the various animations and are used to control the compositing of those animations. In each Layer, there is generally one animation that is considered "active". Layers operate in parallel and are listed in order of increasing priority. This means the animation applied to the model by an earlier Layer (that Layer's active animation) may be overridden or supplemented by following Layers.

Animations can generally be viewed as either happening simultaneously or as replacing each other. For example, you can't really wave and have your hands on your hips at the same time. You can, however, have your hands on your hips, then wave, and then go back to hands on hips. In a sense, waving animation and hands on hips animation are mutually exclusive of each other; one replaces the other. However, you can be waving and walking at the same time. You can even continue walking while alternating from waving to hands on hips and back to waving. In other words, waving animation and walking animation are not mutually exclusive of each other.

Layers are used to define this replacing vs. coexisting relationship between animations. Animations which are mutually exclusive of each other are grouped into the *same* Layer. Animations which can coexist with each other should reside in *different* Layers. Also to allow Improv's compositing to work most effectively, Layers are generally defined from most general animations like body stances, to most specific animations like finger movements and facial expressions. This allows more specific animations to be used as modifiers over a general base animation.

Layers contain a list of States. States define the simple animations such as waving, walking, hands on hips, etc. Generally, one State is active within each Layer. Once you have a set of States defined, you organize your States into Layers based on the States' replacing/coexisting relationships.

Each State contains a list of Filters. Filters manipulate one Channel's value. For example, let's say you want to rotate your model's right arm. You would create a Filter to adjust the Channel for the rotation field of the right arm's transform node. If you also wanted to translate the model's right arm, you would need to create a second Filter. This second Filter would manipulate the Channel for the translation field of the right arm's transform node. Animations are created by defining sets of Filters that move the model's parts in a desired way (e.g. wave, hands on hips, salute, walk, nod, etc.). A Filter contains a Controller and a reference to the Channel to be modified.

Controllers are the work-horse of the Animation Engine; they generate values according to the Animation Engine. Inside a Controller are static, explicitly set values and/or curves or functions that generate values over time. Those values are then given to the Filters to be used in modifying the Channel's value.

The "Big Picture" looks like this: Actors hold a list of Channels and Layers. Channels are the outputs of the Animation Engine. A Layer holds a list of States. A State holds a list of Filters. A Filter contains a Controller and a reference to one of the Channel nodes. The Controller generates a value. The Filter takes that value and uses it to modify the Channel it references

Animating values using Improv can be seen from a "which, what, how, how much" view. The Channel says *which* value to change. The Controller says *what* the replacement value is. The Filter determines *how* to change the value using the new replacement value. And the weight of the Filter determines *how much* change to do. We'll talk more about weights when we talk about transitions between States.

Sequences

Authors can create sequences of animations as well. Sequences simply trigger States according to an author-defined timer. They can access States in any Layer and can specify the length of time the triggered State(s) should be active and what to do when the Sequence is over. Sequences allow you to trigger a list of States simultaneously, randomly trigger a State chosen from a list, and/or trigger a random length pause.

Sequences can also be layered. Sequences are listed in a Layer just like States. When a Sequence in a Layer becomes active, it generally starts to turn on other Sequences and States within the Actor.

For more information about the concepts introduced in this sketch, please see the descriptions of the specific nodes in the Specification of Nodes.

Getting Ready to Author

Now you need to set up your authoring environment: the HTML and VRML files. You will edit the VRML file and view the results in your browser by looking at the HTML file. First make sure you satisfy the System Requirements, then create the HTML page which will display your work.

System Requirements

This version uses the VRML External Authoring Interface and Netscape's LiveConnect to connect the Animation Engine applet and the VRML world. The preferred method involving Script nodes is not used simply because they are not fully implemented in current VRML Browsers.

This means you must embed your VRML file in an HTML file which also contains the ImprovEAI applet.

LiveConnect is a technology from Netscape which provides an interface between, applets, plugins, and JavaScript.

This version currently works on the SGI and Windows 95/NT platforms. You'll need Netscape 3.0 and SGI's CosmoPlayer version 1.0.2beta3 or better for the SGI and 1.0beta5 or better for Windows.

Netscape is available at <http://home.netscape.com>

CosmoPlayer is available at <http://vrml.sgi.com/cosmoplayer>

How to Setup the HTML File

To create the LiveConnect communication pipeline between the VRML world and the Improv applet, you need to include both in one HTML file. The essential elements of this HTML file are the Applet tag and the Embed tag.

```
<APPLET
  CODE="ImproveAI.class"
  ARCHIVE="improv.zip"
  MAYSCRIPT
  WIDTH = 1
  HEIGHT = 1
>
</APPLET>
```

The Applet tag should be written as shown above. NOTE : The improv.zip archive must be in the same directory as the HTML file.

In addition, an Improv Server may be specified inside the Applet tag to enable control of Improv from a Telnet connection. See the Appendix for more information on the Improv Server.

The embed tag SRC refers to the VRML file you are animating. The SRC field should be the name of your VRML .wrl file holding the model and Improv nodes.

```
<EMBED
  SRC="name of your .wrl file"
  WIDTH = 300
  HEIGHT = 200
>
```

How to Setup the VRML File

You will also need to include the Improv prototypes into your .wrl file. You can cut-and-paste the prototype definitions from the ImprovProtos.wrl file. The format looks like this:

```
PROTO nodename [
  fields
] {}
```

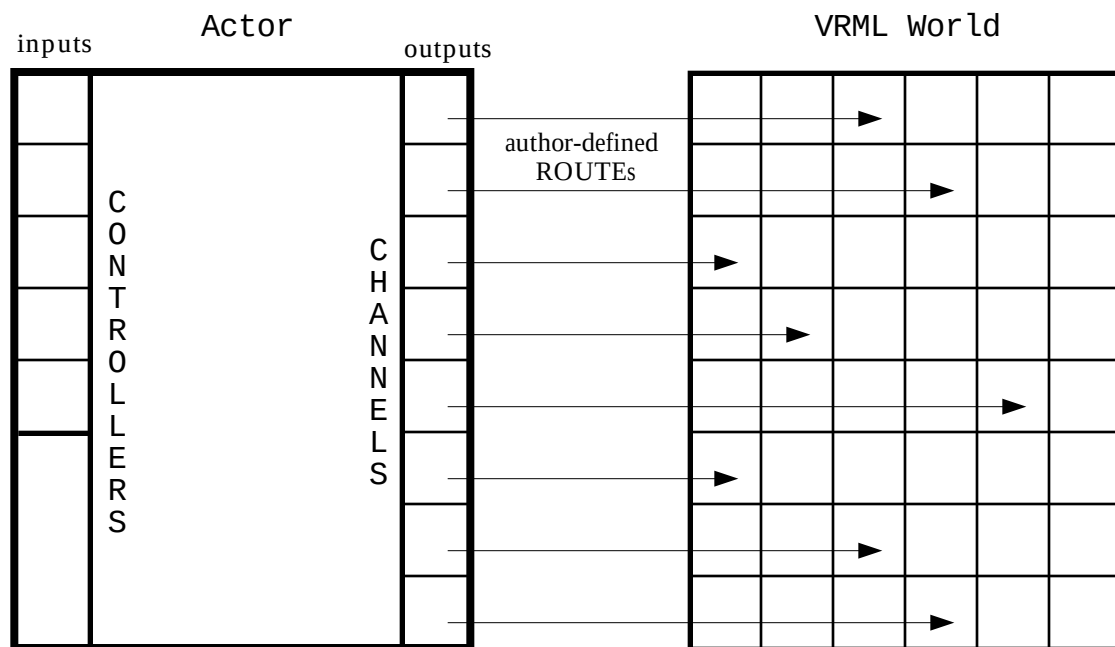
Due to a bug in current VRML browsers, these prototypes can not be defined as EXTERNPROTOs.

Authoring In Improv

Now let's examine the details of the Improv Animation Engine and how to use it to animate VRML models.

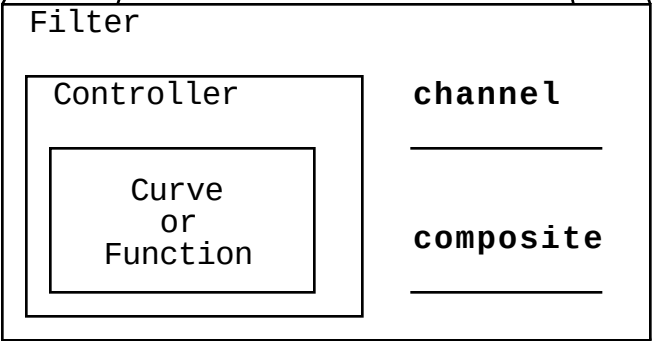
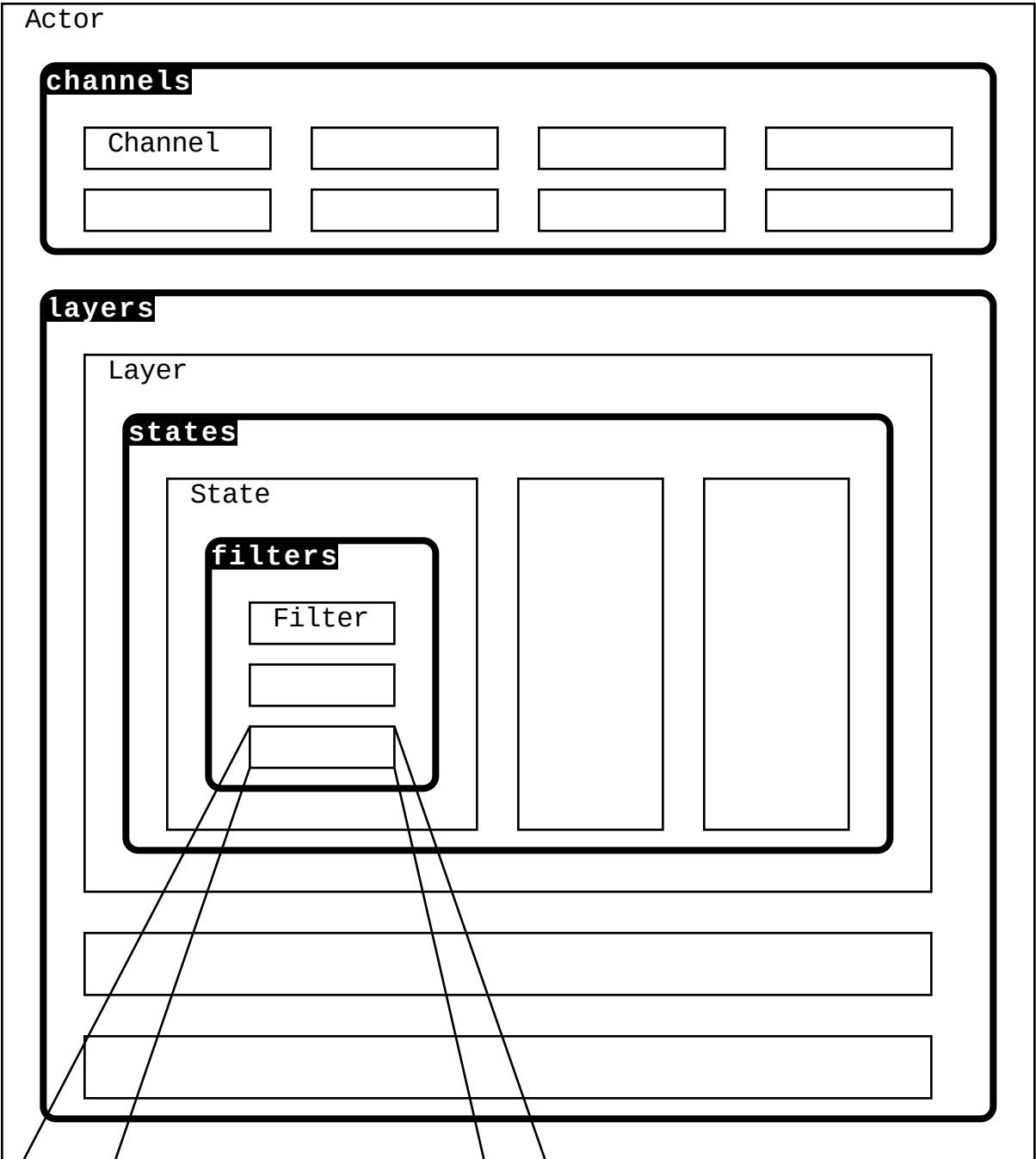
Structure of Improv

The inputs of Improv are the fields of the various Controller nodes. The outputs are the value fields of the various Channel nodes. Each frame, a value is generated for each Channel node according to the organization of the Filter, State, and Layer nodes. Authors can route their inputs into the Controller nodes or create Curve nodes to generate values for the Controller nodes. Then, to see the results, the author must route the value field of the Channel nodes into the to-be-animated fields of the VRML world nodes.



The Improv Actor node is structured as a nested grouping of nodes. (see the following diagram)

- The Improv node holds a list of Actor nodes.
- The Actor contains a list of Channels and Layers
- Channel nodes hold the values to be routed to the VRML model.
- Layer nodes are listed in order of increasing priority.
- State nodes are grouped into Layer nodes.
- Filter nodes are grouped into a State nodes.
- A Filter node takes the value from its Controller node and uses it to modify the value field of the Channel node it references.
- Controller node's values are either set explicitly or generated using Curves or Functions.



Legend

plain	node
bold	field
reverse	MFnode field

How Channel Values Are Computed

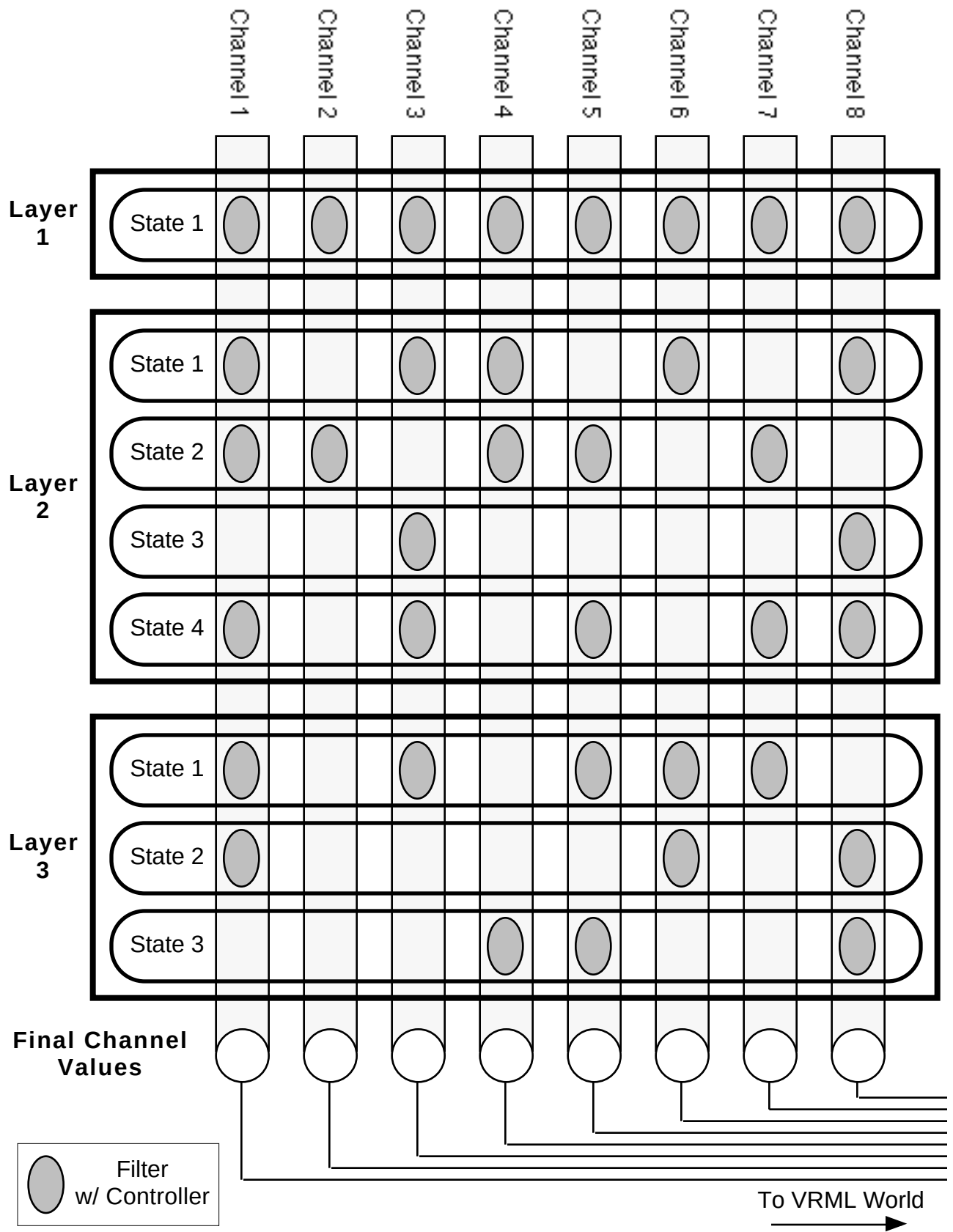
The diagram on the next page should help in visualizing the path of a Channel's value. Channel values basically "fall" down through all the Layers. If there is a Filter in the path of their fall the value is modified. If not, the value continues to fall through the Layers until it collects at the bottom. Once all the values are collected, they are routed to the VRML model.

In each Layer, there is a list of States. In each State, there are Filters assigned to a specific Channel. The Filters modify their assigned Channel's value. The Filter's Controller produces a value which will either influence or replace, depending on the Filter, the existing value of the Channel.

In modifying a Channel's value, the Filter may either completely replace the existing value or use the existing value to produce a new value for the Channel. If the Filter uses the existing value, it is called a *compositing* Filter. This means the *composite* field of the Filter is something other than the default value.

When Improv transitions from one State to another inside the same Layer, one State's weight or influence ramps up to 1 while the other State's weight ramps down to 0. During the transition, both States have weights greater than zero. As long as a State's weight is greater than zero, its Filters will compute new values for their Channels.

First, take a look at the diagram to understand the structure of the "computing Channel values" process and then read on to see a description of the process.



Generally, the first Layer node is an initialization Layer with one State node which sets all the Channel nodes to some default value. After this first Layer is computed, all the Channel nodes will have some “default” value. This default computed value for all the Channels is required for Improv to perform smooth transitions between States. We’ll talk more about this requirement in just a bit.

Then, the values for the next Layer node are computed. This is where compositing comes into play. For example, in the diagram above let’s say that State 2 is the active State of Layer 2. And that the Filter for Channel 2 has its composite field set to something other than the default. This makes this Filter node a compositing Filter. Instead of simply overriding the existing value, a compositing Filter node computes a new value based on the value generated by its Controller and the existing value for the Channel as set by the previous Layer. For example, you could create a composite Filter node that doubles, halves, or adds a constant to the existing value for the Channel. So, for a compositing Filter node, the Channel node’s new value would be:

$$NCV = ECV <CO> CtrV$$

NCV = Channel node’s new value according to the compositing Filter
 ECV = Channel node’s existing value from the previous Layer node’s computation
 <CO> = composite operator (+:ADD, -:SUB, *:MULT, /:DIV, MIN, MAX) in the compositing Filter node
 CtrV = the value provided by the Controller node in the compositing Filter node

So, in our example, the Filter for Channel 2 in State 2 in Layer 2 would take the existing value for Channel 2 according to Layer 1 and use it to generate a new value for Channel 2.

Once the Channel node values for the active State node are computed as described above, the Animation Engine attends to the running computation of the Channel. This is where the Animation Engine processes the step from Layer to Layer and any transitions between State nodes in the same Layer. When a transition occurs from one State node to another in the same Layer node, a smooth transition is generated by way of a weighted sum. As long as a State node’s weight is greater than zero, values for its Filter nodes are computed. During a transition, values for both State nodes are being computed, however the destination State node is considered the “active” State node. So let’s say we’re transitioning from State 1 to State 2 in Layer 3. As you can see, State 1 and State 2 have Filters in the following common channels: Channel 1 and Channel 6. Let’s also say that the transition time is one second.

During the transition, State 1, the *starting* State, has a weight (influence) which decreases from 1 to 0, while State 2, the *destination* State, has a weight which increases from 0 to 1. At any one time, the total weights of the all the Filters contributing to the Channel’s value inside one Layer should equal one. So for each step of the Channel value computation, there is a value according to the Filter and a weight assigned to that Filter.

The value for the Channel is

$$LCV = (SSCV * (1 - WDS)) + (DSCV * WDS)$$

LCV = the Channel Value according to this Layer
 SSCV = the Channel’s Value according to the starting State
 WDS = the Weight of the Destination State
 DSCV = the Channel’s Value according to the destination State

In this computation, the SSCV and DSCV values are computed according to the compositing formula listed earlier. It's important to note that all the compositing work has been done before the running totals are computed.

In the above example where the active States for each Layer are: Layer 1 – State 1, Layer 2 – State 2, and Layer 3 is transitioning between State 1 and State 2, what happens to the value for Channel 2? Nothing, it just passes through Layer 3 because neither of the States being computed has a Filter assigned to Channel 2. The value set by the Filter back in Layer 2 will be collected at the bottom.

Remember that the first Layer should set all the Channel nodes to some default value so Improv can perform smooth transitions?. To explain why, let's take a look at an example without this initialization Layer. Imagine that Layer 1 in the diagram doesn't exist. Let's say that in Layer 2 (now the first Layer) we're transitioning from State 1 to State 2. You can see that they are both animating Channels 1 and 4, but State 2 is also animating Channels 2, 5, and 7. State 1 has no Filters for these Channels, consequently there isn't an existing computed value for these Channels to transition from. This means that when the weighted sum calculation is performed, the SSCV value is *undefined*, the weighted sum calculation fails, and the new Channel value winds up becoming the value from the destination State (State 2) instantly. This results in the actor instantly "jumping" into place for those Channels. In the case of Channel 3, where the transition is toward an undefined value, the weighted sum calculation fails again and the Channel retains the value from the starting State until the end of the transition when it "jumps" to zero. By having a default Layer with one State that has a Filter for every Channel, you insure there will never be an undefined value for a Channel, and the weighted sum calculation will not fail.

For each frame, the above computations are done for the Filter nodes in all the States whose weights are greater than zero. When the Channel values according to that Layer are set, the Animation Engine moves on to the next Layer. When computing each Layer, the Animation Engine is only aware of the final results of the previous Layer's computation; it has no knowledge of how the values came to be. Once the Animation Engine has iterated through all the Layers and the final Channel values for the last Layer are computed and collected by the Channels, the VRML routes are followed and the scene is redrawn.

Authoring Example

Let's try making a basic Improv animation. Assuming we have a stick figure we'll animate her arms to raise over her head.

- Start with a VRML 2.0 model with a defined kinematic skeleton of named nodes. We'll assume we've got a stick figure with arm SFRotation nodes DEFed: L_UP_ARM and R_UP_ARM.
- Create an Improv node to connect your Actor node(s) to the Java Animation Engine. The Animation Engine looks for a node named IMPROV (by way of a DEF statement) to locate all of the Actors it should animate. The Improv node is the *only* connection between your VRML world and the Java Animation Engine.

```
DEF IMPROV Improv {  
    actors []  
}
```

- Create an Actor node to animate the model. The Actor node contains a list of Channel nodes which hold values to be animated by Improv. The Actor node also contains a list of Layer nodes which define the parallel state machine used to direct the animation of the Channel node values. If you want to route from the VRML world into the fields of the Actor node, you'll need to name it with a DEF statement.

```
DEF IMPROV Improv {  
    actors [  
        Actor {  
            name "stick"  
            channels []  
            layers []  
        }  
    ]  
}
```

- Next, define an initial set of Channel nodes of the *same* type as the to-be-animated fields in the VRML model. Channel nodes are treated as eventOuts and give the Animation Engine a place to put its results.

```

DEF IMPROV Improv {
  actors [
    Actor {
      name "stick"

      channels [
        DEF LUA_SFRotationChannel {
          name "l_up_arm"
        }
        DEF RUA_SFRotationChannel {
          name "r_up_arm"
        }
      ]

      layers []
    }
  ]
}

```

Here, we've created two Channel nodes to animate the left and right arm, respectively, of our stick VRML model.

- Create routes from the Channel nodes to the desired fields in the VRML model so the values generated by Improv for the Channel nodes are routed to the model.

```

ROUTE LUA.value_changed TO L_UP_ARM.set_rotation
ROUTE RUA.value_changed TO R_UP_ARM.set_rotation

```

Now, we have ROUTE statements set up to make a connection between the Channel nodes and the VRML model. Once the final value is computed, the Channel's value field will be placed into the rotation field of the specified model's node.

- Create a Layer node. Each Layer node contains a list of State nodes which are used to animate a subset of the Channel nodes. Layer nodes operate in parallel. When the active State node of two different Layer nodes animate a common set of Channel nodes, the values generated by the State node within the "higher" Layer node will override those of the "lower" Layer node (assuming a default composite field value for the Filters of the active State in the "higher" Layer.) Convention dictates you should first create a default Layer to initialize all your Channels to some default value. This allows for smooth transitions in your animations. You can find out more about the reasons behind this default Layer in the Actor node specification.

```

DEF IMPROV Improv {
  actors [
    Actor {
      name "stick"

      channels [
        DEF LUA SFRotationChannel {
          name "l_up_arm"
        }
        DEF RUA SFRotationChannel {
          name "r_up_arm"
        }
      ]

      layers [
        Layer {
          name "Default"
        }
        Layer {
          name "Arm_Movement"
        }
      ]
    }
  ]
}

```

Now, we've got two Layer nodes. The first one is the default Layer used to give all the Channels a default value, the second is to place our arm movement animations in.

- Create a State node within the Layer nodes. State nodes group Filter nodes which are intended to be active at the same time. The various State nodes in a Layer node are mutually exclusive, and at any given time, one of these will be considered active. Activating a State node causes a transition from the previously active State node's animation to the newly activated State node's animation. The transition can be defined by the author or automatically generated by the Animation Engine.

```

DEF IMPROV Improv {
  actors [
    Actor {
      name "stick"

      channels [
        DEF LUA SFRotationChannel {
          name "l_up_arm"
        }
        DEF RUA SFRotationChannel {
          name "r_up_arm"
        }
      ]

      layers [
        Layer {
          name "Default"
          states [
            State {
              name "Background"
              filters []
            }
          ]
        }
        Layer {
          name "Arm_Movement"
          states [
            State {
              name "Arms_Raised"
              filters []
            }
            State {
              name "Arms_Lowered"
              filters []
            }
          ]
        }
      ]
    }
  ]
}

```

Now, we've got two State nodes defining the framework for two animations. One for raising arms and one for lowering them. We also have one State node for our default Layer. This will set the initial values for of our Channels. These will be the first or background values for the Channels and will be the values the second Layer sees if it's performing any compositing Filter computations.

- Create a set of Filter nodes within those State nodes. Each State node contains a list of Filter nodes. Filter nodes modify a Channel node using the value provided by a Controller node. If no Controller is specified or the controller field is set to NULL, Improv generates a default Controller to match the type of the Channel.

```

DEF IMPROV Improv {
  actors [
    Actor {
      name "stick"

      channels [
        DEF LUA SFRotationChannel {
          name "l_up_arm"
        }
        DEF RUA SFRotationChannel {
          name "r_up_arm"
        }
      ]

      layers [
        Layer {
          name "Default"
          states [
            State {
              name "Background"
              filters [
                Filter { channel "l_up_arm" }
                Filter { channel "r_up_arm" }
              ]
            }
          ]
        }
        Layer {
          name "Arm_Movement"
          states [
            State {
              name "Arms_Raised"
              filters [
                Filter {
                  channel "l_up_arm"
                  controller NULL
                }
                Filter {
                  channel "r_up_arm"
                  controller NULL
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}

```

```
State{
  name "Arms_Lowered"
  filters [
    Filter {
      channel "l_up_arm"
      controller NULL
    }
    Filter {
      channel "r_up_arm"
      controller NULL
    }
  ]
}
```

Now, the Filter nodes are defined. And the Channel nodes for arm movement have been referenced. Notice how we haven't defined Channel nodes inside the Filter, but instead have just referenced them by name. This allows you to have several different State nodes animating the same Channel. Now, our default Layer will initialize all the Channels to a default value. How when there's no Controller to provide values? Remember that if there isn't a Controller to provide the Filter a value, Improv generates a Controller based on the type of the Channel. The Controller node that Improv generates has default values for its fields and those values are used by the Filters. You can see the default values for all the nodes in the Specification of Nodes section of this document. If you want values other than the defaults, you would just create Controllers for the Filters in the default Layer and set their values explicitly.

- Create Controller nodes for the Filters to produce the desired animation. The type of Controller node for each Filter node *must* match the type of the referenced Channel node. Controller nodes allow you to explicitly set values for the Channel node, or define Curve or Function nodes to provide the value for the Channel node. Curve and Function nodes allow you to set different values for the Channel node over time.

```

DEF IMPROV Improv {
  actors [
    Actor {
      name "stick"

      channels [
        DEF LUA SFRotationChannel {
          name "l_up_arm"
        }
        DEF RUA SFRotationChannel {
          name "r_up_arm"
        }
      ]

      layers [
        Layer {
          name "Default"
          states [
            State {
              name "Background"
              filters [
                Filter { channel "l_up_arm" }
                Filter { channel "r_up_arm" }
              ]
            }
          ]
        }
        Layer {
          name "Arm_Movement"
          states [
            State {
              name "Arms_Raised"
              filters [
                Filter {
                  channel "l_up_arm"
                  controller SFRotationController {
                    value 0 0 1 2
                  }
                }
                Filter {
                  channel "r_up_arm"
                  controller SFRotationController {
                    value 0 0 1 -2
                  }
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}

```

```

State {
  name "Arms_Lowered"
  filters [
    Filter {
      channel "l_up_arm"
      controller SFRotationController {
        value 0 0 1 0
      }
    }
    Filter {
      channel "r_up_arm"
      controller SFRotationController {
        value 0 0 1 0
      }
    }
  ]
}
]
}
]
}
]
}
]
}
]
}
}

```

Now, the Controller nodes are specified. When the Arms_Raised State is activated, the stick figure's arms will raise and stay there. When the Arms_Lowered State is activated, the stick figure's arms will lower and stay there. And because the Default Layer gives all the Channels an initial value, when the first State in the Arm_Movement Layer is activated, the stick figure will move her arms smoothly instead of snapping into position. Not very exciting, but, hopefully, you get the idea.

- If desired, define a Curve or Function node for the Controller node which will generate time-based values each frame to be given to the Filter node. There is an appropriate Curve and Function node for the type of field to be animated: float, 2D point, 3D point, or list of 3D points.

```

controller SFVec3fController
{
  value 0 0 1
  curve_x SFFloatCurve
  {
    type "SQUARE"
    mode "LOOP"
    key [ 0, .75, .76, 1, 1.1, 1.35, 1.36 ]
    keyValue [ 0, 40, 0, 38, 0, 42, 0 ]
  }
}

```

Here a SFFloatCurve node is defined which will provide values for the x value of the 3D point generated by this SFVec3fController node. A list of time points is specified in the key field and the corresponding values for the curve is specified in the keyValue field. This is just an example of a Curve inside a Controller node. See the Specification of Nodes section for more detail on Curve and Function nodes.

From here authors would generally go on to create additional Channel, Layer, State, Filter, Controller, Curve, and Function nodes as necessary.

Reusing Nodes

All of the reuse capabilities (DEF and USE) of VRML are available. For example, you can define a Curve node for one Controller node and then reuse it in a different Controller node. Or reuse an entire Controller node. However, remember that any change to a DEFed node will change all the other USEs of that node. To get true separate instances, you would need to define prototypes of the nodes.

Controlling Improv From Other Nodes

When used in a Filter, the value field of a Controller is used to explicitly set the value for the Filter's referenced Channel node. A ROUTE statement joining another node that produces those values and the implicit eventIn set_value field of a Controller node can be created to allow other nodes to control Improv animations. For example, if you had a custom node that computed SFRotation values based on a ProximitySensor node, you could route the values generated by that custom node into a SFRotationController node. And the Filter containing that SFRotationController node would adjust its referenced Channel's value according to the ProximitySensor node.

Specification of Nodes

This is the specification of the VRML API to the Java Improv Engine. The nodes are listed below with a diagram of the node's fields, a breakdown of the fields and their meanings, and the default syntax of the node showing:

- the node's fields
- the default values of each field
- the type of each field
- the type of data of each field value

NOTE: Important Bug Work Around : SFNodes defined as MFNodes

In order to work around a bug in the PC version of CosmoPlayer, SFNode fields need to be declared as MFNodes in the PROTO definitions. For the purposes of authoring, these fields should be treated as SFNodes. For clarity the nodes are defined and described as if the bug did not exist. Eventually, we expect this will be true.

Legend for Node Diagrams

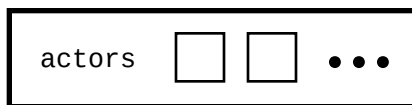
NOTE: For nodes which have both SF and MF versions (Channel, Controller, Curve, and Function), the diagram for the SF version is given.

Legend for Node Diagrams

Single Value	—		
List of Values	—	—	• • •
Single Node	□		
List of Nodes	□	□	• • •

Improv node

Improv



This is the root node of the Improv tree structure. It should be DEFed with the name IMPROV. It is used to provide a link between the Applet and the VRML world. The Applet looks for a node DEFed as IMPROV and will operate on all the Actors it finds listed in the actors field. The Improv node is the only link between the VRML world and the Animation Engine. If you do not have this node DEFed as IMPROV, the Animation Engine will not find your Actors and will not animate your model.

<u>Field</u>	<u>Description</u>
actors	The list of the Actor nodes to be animated.

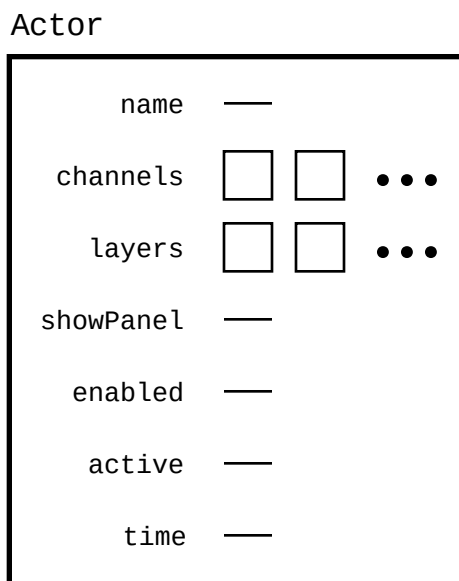
```
Improv {  
    actors
```

```
    []
```

```
    # exposedField MFNode
```

```
}
```

Actor node



The Actor is where all the Channels, Layers and animations for one character are specified. You can have as many Actor nodes as you'd like, but generally there is only one Actor per model.

There is a convention for the creation of Layers for an Actor node. The Layers are added together to produce final values. The first Layer has the lowest priority while the last Layer has the highest priority. So, if an earlier Layer's active State was animating the left arm to swing slowly, and a later Layer's active State was animating the left arm to salute vigorously, the vigorous left arm movement would override the earlier left arm animation. Convention also dictates that Sequence Layers should be listed before the animation Layers. See the Sequence node definition for more information on Sequence Layers.

Since you can not smoothly transition from nothing to an action, convention also dictates that you need a default Layer with one State node (the default State) that initializes all the Channels in this Actor. Once initialized, you can transition from this default "resting" State to any other State. If you do not have a defined "resting" State, the activated State will "jump" into the defined animation instead of smoothly ramping up from the resting position. In other words, every Channel must have a defined value to allow for smooth transitions from State to State.

The convention is for authors to create animation Layers starting with most general animations to most specific. So your first non-sequence Layer would be a background Layer with just one State which sets default values for all the Channels, thus allowing smooth transitions to any action. The next Layer would be for general animations for the entire body: i.e. stances or walking. The next would be for more specific animations: i.e. arm / leg movements. The next for head movements. Next for facial expressions. And so on.

<u>Field name</u>	<u>Description</u>
<code>name</code>	Used to distinguish between Improv Actors. This is used for the server identification so you can direct your messages to the right Improv Actor.
<code>channels</code>	A list of the Channel nodes this Actor node will control.
<code>layers</code>	A list of the Layer nodes in this Actor node.
<code>showPanel</code>	Used to control the display of the Actor State Panel TRUE : the Actor State Panel should be shown FALSE : the Actor State Panel should be hidden
<code>enabled</code>	Viewed as an eventOut. This delays starting the actor clock until after the Actor has been completely loaded. This should <i>not</i> be set by the user.
<code>active</code>	Viewed as an eventIn. The Actor will step through every Layer and activate every State with the name given. If one Layer has two States with the same name (which it shouldn't), the last one listed will be activated. Expects the State's name.
<code>time</code>	The clock for this Actor node. This is set by Improv.

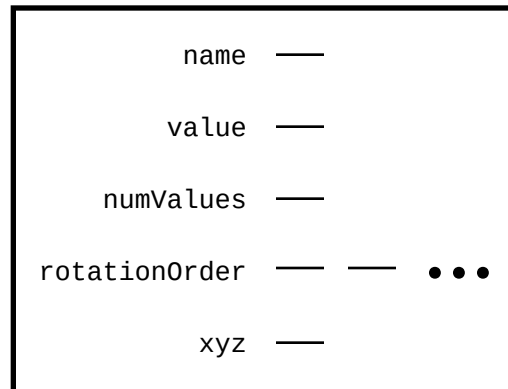
```

Actor {
  name          ""          # exposedField  SFString
  channels      []          # exposedField  MFNode
  layers        []          # exposedField  MFNode
  showPanel     TRUE        # exposedField  SFBool
  enabled       FALSE       # exposedField  SFBool
  active        ""          # exposedField  SFString
  time          # eventOut   SFTIME
}

```

Channel nodes

typeChannel



Channel nodes are a holding area for values from the Animation Engine; they are the eventOuts of Improv. The Channel's value field is set by the Filters nodes which reference the Channel. Once the Channel node's final value is computed, it is routed according to the author-defined route statement to whatever field of the scene the Channel node was created to animate. There is a type-specific Channel node for each type of field to be animated.

<u>Field</u>	<u>Description</u>
<code>name</code>	The name of the Channel node. This is the name the Filters use to reference the Channel.
<code>value</code>	The value of the Channel node. To the user this is seen as an eventOut since the user should never set this field directly.
<code>numValues</code>	The number of points listed in the value field of a MF node.
<code>rotationOrder</code>	Used to specify the order of rotations when using Euler angles. Currently, only the default setting allows you to mix Euler and Quaternion values in Controllers modifying this Channel. If you set this to any other order, you <i>must</i> use Euler values exclusively. The values are interpreted as x y z. So in the default setting, the rotation about x is done first, followed by y and finally z.
<code>xyz</code>	The Euler angles from a SFRotation Channel. These are the Euler equivalents of the Quaternion value output by an SFRotation Channel.

```

SFBoolChannel {
    name      ""
    value     FALSE
    # exposedField SFString
    # exposedField SFBool
}

SFStringChannel {
    name      ""
    value     ""
    # exposedField SFString
    # exposedField SFString
}

SFNodeChannel {
    name      ""
    value     []
    # exposedField SFString
    # exposedField SFNode
}

SFInt32Channel {
    name      ""
    value     0
    # exposedField SFString
    # exposedField SFInt32
}

SFFloatChannel {
    name      ""
    value     0
    # exposedField SFString
    # exposedField SFFloat
}

SFVec2fChannel {
    name      ""
    value     0 0
    # exposedField SFString
    # exposedField SFVec2f
}

SFVec3fChannel {
    name      ""
    value     0 0 0
    # exposedField SFString
    # exposedField SFVec3f
}

SFColorChannel {
    name      ""
    value     0 0 0
    # exposedField SFString
    # exposedField SFColor
}

SFRotationChannel {
    name      ""
    value     0 1 0 0
    rotationOrder [ 0 1 2 ]
    xyz       0 0 0
    # exposedField SFString
    # exposedField SFRotation
    # exposedField MFInt32
    # exposedField SFVec3f
}

MFStringChannel {
    name      ""
    value     []
    numValues 0
    # exposedField SFString
    # exposedField MFString
    # exposedField SFInt32
}

MFNodeChannel {

```

```

    name          ""          # exposedField SFString
    value         []          # exposedField MFNode
    numValues     0           # exposedField SFInt32
}

MFInt32Channel {
    name          ""          # exposedField SFString
    value         []          # exposedField MFInt32
    numValues     0           # exposedField SFInt32
}

MFFloatChannel {
    name          ""          # exposedField SFString
    value         []          # exposedField MFFloat
    numValues     0           # exposedField SFInt32
}

MFVec2fChannel {
    name          ""          # exposedField SFString
    value         []          # exposedField MFVec2f
    numValues     0           # exposedField SFInt32
}

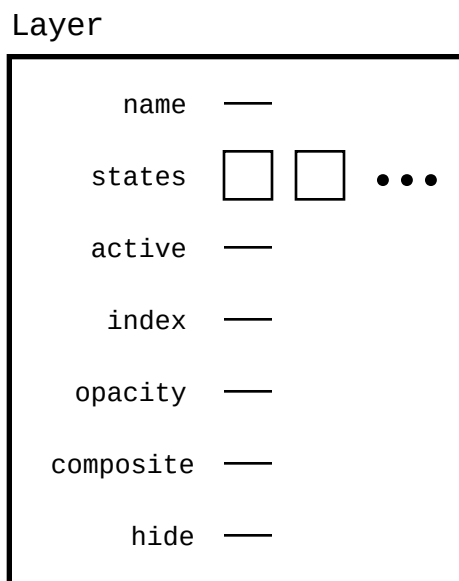
MFVec3fChannel {
    name          ""          # exposedField SFString
    value         []          # exposedField MFVec3f
    numValues     0           # exposedField SFInt32
}

MFColorChannel {
    name          ""          # exposedField SFString
    value         []          # exposedField MFColor
    numValues     0           # exposedField SFInt32
}

MFRotationChannel {
    name          ""          # exposedField SFString
    value         []          # exposedField MFRotation
    numValues     0           # exposedField SFInt32
}

```

Layer node



Layer nodes define the specific Layers that will be composited to create the final Channel values. In the Actor node, Layers are composited in a first-defined to last-defined order. So the last Layer node defined in an Actor node will, if controlling the same Channels as previously defined Layers, override or supplement the values generated by those Layers. Authors can control the compositing of Layers by making use of the `composite` field in the Filter, State, and Layer nodes. More information about Layer nodes and authoring conventions can be found in the Actor node definition. More information on compositing can be found in the Filter node definition and the section “How Channel Values Are Computed”.

Even though values for multiple States may be being computed, only one State from the `states` field is considered the active State at any one time. The active State is the State the Layer has reached or is moving toward. The first State node listed in the `states` field is considered the *default* State, as such it is active at startup and is the State returned to once the duration of an activated State has passed.

You can activate an animation by its name or position in the Layer, set the opacity for the layer, and specify a Layer wide composite behavior.

<u>Field</u> name	<u>Description</u>
states	The Layer's name. This is used in connecting to higher level behaviors and in the authoring widgets.
active	The list of State or Sequence nodes or that define the animations/sequences for this Layer node.
index	Viewed as an eventIn. Any State who's name matches this string will be activated. Used to activate a State from elsewhere in the VRML file. Expects the State's name. If the named State is not found inside the Layer, nothing changes.
opacity	Similar to the active field above, this will activate the State in the given numerical position. Used to activate a State from elsewhere in the VRML file. Expects the State's numerical location in the states field. If the position is out of bounds, nothing changes. The first State is in position 0.
composite	Controls how much the values of the previous Layer "bleeds through" into the values of this Layer. The default value of 1 means the Filters of this Layer will completely override the existing values from previous Layer. Values of less than one allow the Filters in the underlying Layer to "show through," influencing the values generated by this Layer by (1 - opacity of this Layer). So an opacity value of .75 would result in a value for the Channel that is based 75% on the current Layer and 25% on the lower Layer. With the opacity field you can create a Layer that adds a general effect to the animations. You control the Layer's opacity to apply as much effect as you desire.
composite	Sets the compositing behavior for all the State nodes in this Layer node. The compositing behavior is determines how the value provided by the Filter's Controller node is combined with the existing value of the referenced Channel node. The default is SET. SET: There is NO compositing. Any existing value for the Channel node is completely ignored and replaced by the value provided by the Controller node. ADD : set the value field of the referenced Channel node to the sum of the value provided by the Controller node and the existing value of the Channel node [ECV + CtrV]. SUB : set the value field to the difference of the Controller value and the existing value [ECV - CtrV]. MULT : set the value field to the product of the Controller value and the existing value [ECV * CtrV]. DIV : set the value field to the quotient of the Controller value and the existing value [ECV / CtrV]. MAX : set the value field to the larger of the two values [max(ECV,CtrV)]. MIN : set the value field to the smaller of the two values [min(ECV,CtrV)].

hide

Allows you to prevent this Layer from showing up in the Actor State Panel. The Layer is completely functional and its States are operating and can be called from Sequences, it just doesn't appear in the Actor State Panel. Used to save screen space.

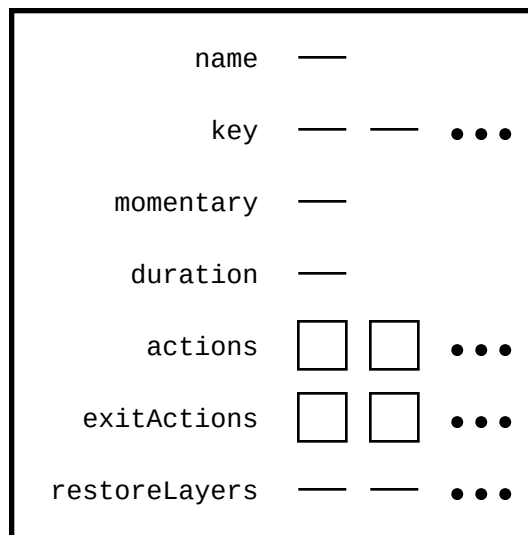
TRUE : This Layer will appear in the Actor State Panel

FALSE : This Layer will not appear in the Actor State Panel

```
Layer {
  name          ""          # exposedField  SFString
  states        []          # exposedField  MFNode
  active        ""          # exposedField  SFString
  index         0           # exposedField  SFInt32
  opacity       1           # exposedField  SFFloat
  composite     "SET"       # exposedField  SFString
  hide          "FALSE"    # exposedField  SFBool
}
```

Sequence node

Sequence



The Sequence node is used in the states field of a Layer node. It defines a sequence and time line of animations (States) for the Layer to perform. Sequences just activate States and move on; like a human selecting States from the applet's Actor State Panel. They do not manage transitions between the States they trigger. That is left to the Animation Engine. They do not deactivate the States they previously triggered when they trigger the next State(s). Therefore, if you trigger a State in one Layer, and then at a later time, trigger a State in a different Layer, the State triggered first will continue to be active unless its duration has passed or a different State in the same Layer has been triggered by some other process.

A Sequence node will *not* skip an action listed in the actions field. For example, let's say a Sequence is supposed to trigger an action at time 6, 10 and at time 12. The Sequence triggers its action at time 6. Let's say the action triggered at time 6 is a Hold of 7 seconds (see Hold node below). The sequence pauses for 7 seconds. When the pause is over, the Improv clock is at 13 seconds. The Sequence will trigger the action for time 10. Then, one frame later, it will trigger the action for time 12.

You can also cause an action to be triggered immediately after another action by setting its key to -1. In general, a Sequence checks the time and check's the key for the current action. If the key value is less than the time, the Sequence triggers the action. It then moves to the next action and check's its key value. A Sequence does *not* wait for the triggered State(s) to complete before moving on to the next action and performing the key value vs. time check. However, if the triggered action is a Hold, the Sequence pauses for the length of time specified before moving to the next action and performing the time check.

The exitActions field allows a Sequences to trigger actions when the Sequence exits. You can also tell a Layer to return to the State or Sequence that was active before the Sequence was activated by using the restoreLayers field.

Convention dictates that Sequences should be listed in a separate Sequence-only Layer. This Sequence Layer should have *no* States listed in its state field; just Sequences. The Sequence Layer's default (first) Sequence should be a Sequence that does nothing. The Sequence Layer should also be listed before any animation Layer (a Layer containing States).

A word about transitions: In a Layer when a transition occurs from State to State, all the Filters in the first State are ramped down (deactivated) while all the Filters in the second State are ramped up. When a transition occurs from a State to a Sequence, the Sequence is instantly activated; consequently, the State is instantly deactivated. This breaks the smooth transition between animations and is the reason behind the convention of not mixing Sequences and States in the same Layer. A transition from a Sequence to a Sequence causes an immediate transition between Sequences, but the States activated will still smoothly deactivate when some other State in their Layer is activated or their duration expires.

<u>Field</u> name	<u>Description</u>
key	The timing points for the actions listed in the actions field. The first key is for the first action, the second key for the second action and so on. A key value of -1 means the associated action should occur immediately after the triggering of the previous action. If the previous action is a Hold, the Sequence will pause for the length of the Hold before triggering the current action.
momentary	Determines how this sequence will behave when it finishes: TRUE : as long as the duration is not set to -1, this Sequence will run for the duration specified in the duration field and then will return to the <i>previously active</i> Sequence in this Layer. FALSE : the Sequence will run for the duration specified and then activate the <i>default</i> (first) Sequence in this Layer.
duration	The duration of time this Sequence will be active. This allows you to limit the amount of time the Sequence will be active. The default value, -1, means the Sequence should be active indefinitely. Can be used to put a time limit on a repeating Sequence (see Repeat node).
actions	The list of sequence actions (Trigger, RandomTrigger, Hold, and Repeat nodes) to be performed at the times specified in the key field.
exitActions	This takes a list of Triggers or RandomTriggers. These actions are executed immediately upon the exit of this Sequence. A Sequence exits because 1) the duration time has expired or 2) because another Sequence in the same Layer has been activated.
restoreLayers	This takes a list of Layer names. When the Sequence exits, these Layers return to the State they were in prior to this Sequence's activation. The actions specified in the exitActions field <i>override</i> the behavior of the restoreLayers field. So, if you have any exitActions in a Layer specified in the restoreLayers field, the exitActions will be activated immediately after the Layer is reset by the restoreLayers field.

```

Sequence {
  name          ""          # exposedField  SFString
  key           []          # exposedField  MFFloat
  momentary     FALSE       # exposedField  SFBool
  duration      -1          # exposedField  SFFloat
  actions       []          # exposedField  MFNode
  exitActions   []          # exposedField  MFNode
  restoreLayers []          # exposedField  MFString
}

```

Trigger node

Trigger



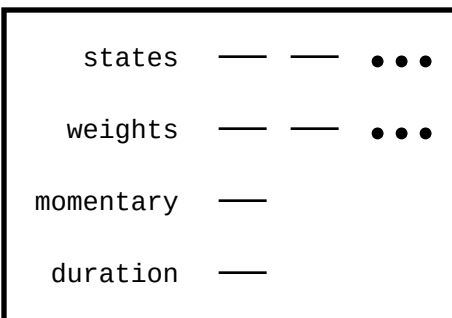
The Trigger node activates a list of States simultaneously. You can also specify how long the triggered States should remain active.

<u>Field</u>	<u>Description</u>
states	The list of States to trigger simultaneously.
momentary	Used to temporarily override a FALSE valued momentary field of the States or Sequences being triggered. States and Sequences whose momentary values are set to FALSE may have them temporarily set to TRUE for the time specified in the duration field. However, if the triggered State or Sequence has a momentary value of TRUE, it can not be set to FALSE using this field; it cannot be made temporarily un-momentary.
duration	The duration of time the States should be active. The default value -1 means the activated State should remain active until a different State in their Layer is activated. This value will override any duration specified at the State level.

Trigger {			
states	[]	# exposedField	MFString
momentary	FALSE	# exposedField	SFBool
duration	-1	# exposedField	SFFloat
}			

RandomTrigger node

RandomTrigger



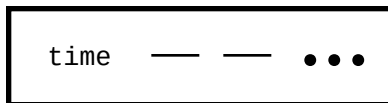
The `RandomTrigger` node is much like the `Trigger` node except that a random State is chosen from the states field. You can also specify how long the triggered State should remain active.

<u>Field</u>	<u>Description</u>
<code>states</code>	The list of States, one of which is chosen at random to be triggered.
<code>weights</code>	Allows you to bias the random selection. The <code>weights</code> field is used to specify a set of weights, one for each option in the <code>states</code> field, which are used to bias the random choice toward certain options and away from others. If no weights are specified, the choice is unbiased.
<code>momentary</code>	Used to temporarily override a FALSE valued <code>momentary</code> field of the State or Sequence being triggered. States and Sequences whose <code>momentary</code> values are set to FALSE may have them temporarily set to TRUE for the time specified in the <code>duration</code> field. However, if the triggered State or Sequence has a <code>momentary</code> value of TRUE, it can not be set to FALSE using this field; it cannot be made temporarily un-momentary.
<code>duration</code>	The duration of time the State should be active. The default value of -1 means the activated State should remain active until a different State in their Layer is activated. This value will override any duration specified at the State level.

```
RandomTrigger {
    states      [ ]      # exposedField  MFString
    weights     [ ]      # exposedField  MFFloat
    momentary   FALSE    # exposedField  SFBool
    duration    -1       # exposedField  SFFloat
}
```

Hold node

Hold



The Hold node creates a pause between the previous action in the Sequence and the next. The Sequence will hold for the number of seconds listed before it checks to see if it should trigger the next action. It's possible for the Sequence to “fall behind” the Improv clock due to a Hold. The sequence will just trigger all the “missed” actions until it gets back in sync with the Improv clock.

Field
time

Description

The length of time, in seconds, for the pause. If two values are given, a random value between the two values will be chosen for the length of the pause.

```
Hold {  
  time
```

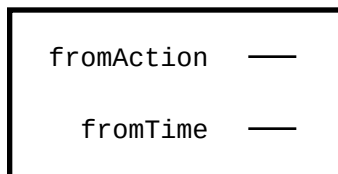
```
[]
```

```
# exposedField  MFFloat
```

```
}
```

Repeat node

Repeat



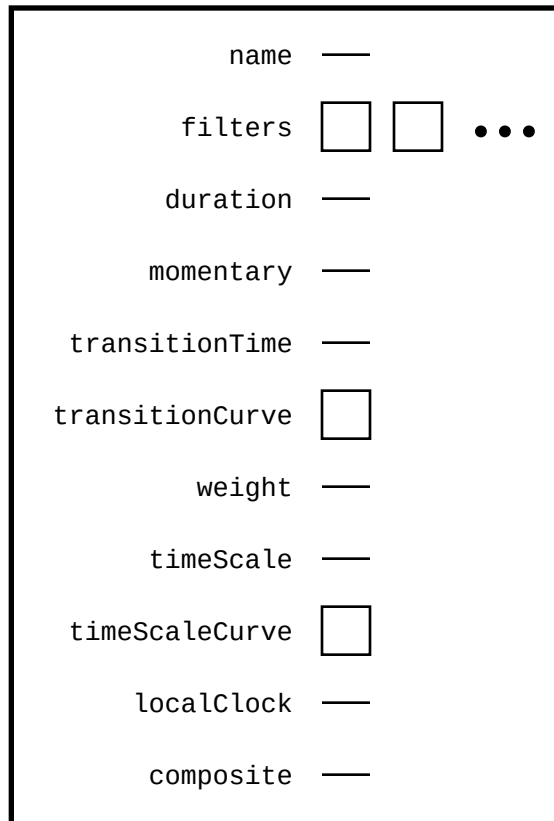
The Repeat node allows you to repeat a Sequence beginning at a specified action and time. The `fromAction` field specifies the position to begin execution. The first action in a Sequence is in position zero. The `fromTime` field will reset the local clock for the Sequence to the value specified. The default behavior is for the Sequence to repeat from the beginning and reset the local clock to zero, thus doing the exact Sequence over again.

<u>Field</u>	<u>Description</u>
<code>fromAction</code>	The Sequence will jump to the position specified given by this value. The first action in a Sequence is in position 0.
<code>fromTime</code>	This resets the local clock for the Sequence to the given value.

```
Repeat {
  fromAction 0 # exposedField SFInt32
  fromTime 0 # exposedField SFFloat
}
```

State node

State



State nodes are used to group the Filter nodes which create a specific animation into one unit. State nodes also control the transitions between different animations and the clock for the animations. You can also define a State wide composite behavior.

<u>Field</u> name	<u>Description</u>
filters	The list of Filter nodes that make up the animation.
duration	The number of seconds this State node's animation should be active before returning to the default State. (see Layer node). The default value is -1 which results in the State node being active indefinitely.
momentary	Determines how this State will behave when it finishes: TRUE : as long as the duration is not set to -1, this State will run for the duration specified in the duration field and then will return to the <i>previously active</i> State in this Layer. FALSE : the State will run for the duration specified and then activate the <i>default</i> (first) State in this Layer.
transitionTime	The number of seconds for this State node's animation to completely replace the previously active animation. Like a fade-in time. In terms of weight, this is the time it takes for this State's weight to go from 0 to 1. The default is one second.
transitionCurve	Expects a SFFloatCurve or SFFloatFunction which defines the transition curve for this State. The curve will output weight values for this State over a time duration defined by the curve. The curve's duration overrides any value in the transitionTime field. The weight is used in the Channel value computation. See How Channel Values Are Computed for more information. The default curve is an "S" curve whose duration is the value of the transitionTime field.
weight	The weight field is used to explicitly set the weight of this State node. By default, this is controlled by the automatic transition mechanism. When one State is activated, its influence rises from 0 to 1 as the weight of the previously active State drops from 1 to 0, over the transitionTime or according to the transitionCurve. When setting the weight explicitly, this transition mechanism is turned off, and the value of the weight field is used instead. This is intended to be an eventIn with values routed to it from some other source.
timeScale	Adjusts the cycle of the animation. A value of 2 would make the animation cycle twice as fast. When doing phased synchronized animations (two animations with the same keyValue fields) this field allows you to smoothly transition from one animation to the next. For example a walking animation might have a timeScale of 1 while a run animation has a timeScale of 2. When you transition from walking to running, the movements will synch up as the timeScale value is smoothly ramped up from 1 to 2.

timeScaleCurve Adjusts the cycle of the animation according to a SFFloatCurve or SFFloatFunction instead of using an explicit value. This will override the timeScale field if non NULL. For example: if a noise curve is used for a walk animation, the walker will complete a cycle in roughly the same amount of time, but with some variation to look realistic.

localClock Used in synchronizing animations.
 TRUE : this State node's clock is set to zero.
 FALSE : this State node's clock is set to the clock of the previously active animation.

composite Sets compositing behavior for all the Filter nodes listed in this State node. The compositing behavior is determines how the value provided by the Filter's Controller node is combined with the existing value of the referenced Channel node. The default is the empty string.
 "" : Use the composite setting from the enclosing Layer node.
 SET : set the value field of the referenced Channel node to the value provided by the Controller node.[CtrvV]
 ADD : set the value field of the referenced Channel node to the sum of the value provided by the Controller node and the existing value of the Channel node [ECV + CtrV].
 SUB : set the value field to the difference of the Controller value and the existing value [ECV - CtrV].
 MULT : set the value field to the product of the Controller value and the existing value [ECV * CtrV].
 DIV : set the value field to the quotient of the Controller value and the existing value [ECV / CtrV].
 MAX : set the value field to the larger of the two values [max(ECV,CtrV)].
 MIN : set the value field to the smaller of the two values [min(ECV,CtrV)].

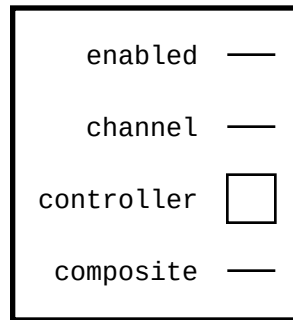
```

State {
  name          ""          # exposedField  SFString
  filters        []         # exposedField  MFNode
  duration       -1         # exposedField  SFFloat
  momentary      FALSE      # exposedField  SFBool
  transitionTime 1          # exposedField  SFFloat
  transitionCurve NULL      # exposedField  SFNode
  weight         0          # exposedField  SFFloat
  timeScale      1          # exposedField  SFFloat
  timeScaleCurve NULL      # exposedField  SFNode
  localClock     TRUE       # exposedField  SFBool
  composite      ""         # exposedField  SFString
}

```

Filter node

Filter

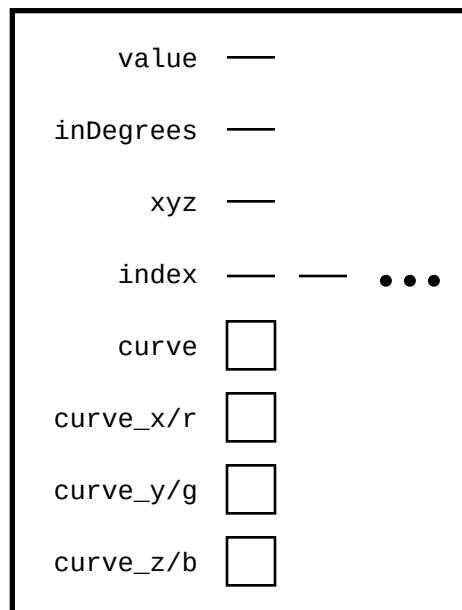


A Filter node represents some modification or animation of a specific Channel node. A Filter node takes the value produced by its Controller node and uses it to modify the value field of the referenced Channel. The types (Bool, SFVec3f, etc.) of the Controller and Channel *must* match.

<u>Field</u>	<u>Description</u>
enabled	This is a debugging tool allowing you to deactivate the Filter node without having to comment out the code: TRUE : the Filter node is active. FALSE : the Filter node is inactive.
channel	The referenced Channel node which will be modified using the value from the Controller node. This should be the value of the name field of the intended Channel node.
controller	The Controller node which will produce a value for this Filter.
composite	How the value provided by the Controller node for the referenced Channel node is composited with the existing value of the Channel node. The default is the empty string. "" : Use the composite setting from the enclosing State node. SET : set the value field of the referenced Channel node to the value provided by the Controller node.[CtrvV] ADD : set the value field of the referenced Channel node to the sum of the value provided by the Controller node and the existing value of the Channel node [ECV + CtrV]. SUB : set the value field to the difference of the Controller value and the existing value [ECV - CtrV]. MULT : set the value field to the product of the Controller value and the existing value [ECV * CtrV]. DIV : set the value field to the quotient of the Controller value and the existing value [ECV / CtrV]. MAX : set the value field to the larger of the two values [max(ECV,CtrV)]. MIN : set the value field to the smaller of the two values [min(ECV,CtrV)].
Filter {	
enabled	TRUE # exposedField SFBool
channel	"" # exposedField SFString
controller	NULL # exposedField SFNode
composite	"" # exposedField SFString
}	

Controller nodes

typeController



Controller nodes are the work horses of the Animation Engine. Controller nodes generate values which are used to modify the value field of the Channel node referenced inside their Filter node. Their values are either explicitly set, generated by Curves or Function nodes, or can be routed from other nodes in the scene.

<u>Field</u>	<u>Description</u>
value	This is the explicit value produced by this Controller.
inDegrees	NOT YET IMPLEMENTED - Sets the units of the xyz field in a SFRotationController node: TRUE : values for the xyz field are treated as degrees FALSE : values for the xyz value are treated as radians
xyz	These are the Euler values (roll, pitch, yaw) for the SFRotationController node. They are to be specified in units of <i>degrees</i> .
index	Indicates which points in the MFtypeChannel node are being animated by this MFtypeController node. For example, an index of [2, 6] means the generated values will be used to animate the second and sixth points in the MFtypeChannel node. Note the numValues field in MFtypeChannel nodes.
curve	This is the Curve or Function node that will output all the values for this Controller. This is NOT YET IMPLEMENTED for the SFRotationController node. If curve is NULL, the values in the value field of the Controller node are used.

curve_x, curve_y The dimension-specific curve fields. These take a SFFloatCurve and are used to animate one dimension of a multi-dimension value
 curve_z, curve_r field. In a SFRotationController node the curve_x, curve_y, and
 curve_g, curve_b curve_z fields map to the Euler angles in degrees of roll, pitch, and yaw respectively.

NOTE : priority If the value field above is set and a Curve node is also given for one of the dimension specific curves, the dimension specific curve field will override that dimension of the value field. For example, in a SFVec3fController node, the value field is set to 10 20 10. The curve_y field is also set to a defined Curve node. So, the value field in the Channel node is set to 10 <value from curve_y> 10. The fields are listed in increasing priority: the curve field will override the value field and the dimension specific curve fields (curve_x, curve_y, etc.) will override the curve field.

```
SFBoolController {
  value      FALSE      # exposedField  SFBool
  curve      NULL       # exposedField  SFNode
}

SFStringController {
  value      ""          # exposedField  SFString
}

SFNodeController {
  value      NULL        # exposedField  SFNode
}

SFInt32Controller {
  value      0           # exposedField  SFInt32
  curve      NULL       # exposedField  SFNode
}

SFFloatController {
  value      0           # exposedField  SFFloat
  curve      NULL       # exposedField  SFNode
}

SFVec2fController {
  value      0 0         # exposedField  SFVec2f
  curve      NULL       # exposedField  SFNode
  curve_x    NULL       # exposedField  SFNode
  curve_y    NULL       # exposedField  SFNode
}
```

```

SFVec3fController {
    value          0 0 0      # exposedField    SFVec3f
    curve          NULL      # exposedField    SFNode
    curve_x        NULL      # exposedField    SFNode
    curve_y        NULL      # exposedField    SFNode
    curve_z        NULL      # exposedField    SFNode
}

SFCOLORController {
    value          0 0 0      # exposedField    SFCOLOR
    curve          NULL      # exposedField    SFNode
    curve_r        NULL      # exposedField    SFNode
    curve_g        NULL      # exposedField    SFNode
    curve_b        NULL      # exposedField    SFNode
}

SFRotationController {
    value          0 1 0 0    # exposedField    SFRotation
    inDegrees      TRUE      # exposedField    SFBool
    xyz            0 0 0      # exposedField    SFVec3f
    curve          NULL      # exposedField    SFNode
    curve_x        NULL      # exposedField    SFNode
    curve_y        NULL      # exposedField    SFNode
    curve_z        NULL      # exposedField    SFNode
}

MFStringController {
    value          []         # exposedField    MFString
    index          []         # exposedField    MFInt32
}

MFNodeController {
    value          []         # exposedField    MFNode
    index          []         # exposedField    MFInt32
}

MFInt32Controller {
    value          []         # exposedField    MFInt32
    index          []         # exposedField    MFInt32
    curve          NULL      # exposedField    SFNode
}

MFFloatController {
    value          []         # exposedField    MFFloat
    index          []         # exposedField    MFInt32
    curve          NULL      # exposedField    SFNode
}

MFVec2fController {
    value          []         # exposedField    MFVec2f
    index          []         # exposedField    MFInt32
    curve          NULL      # exposedField    SFNode
}

MFVec3fController {

```

```

    value      []      # exposedField  MFVec3f
    index      []      # exposedField  MFInt32
    curve      NULL    # exposedField  SFNode
}

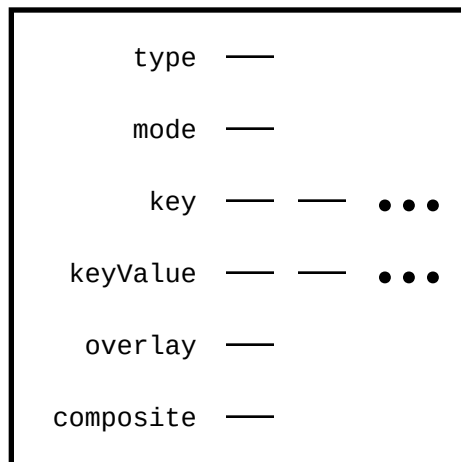
MFColorController {
    value      []      # exposedField  MFColor
    index      []      # exposedField  MFInt32
    curve      NULL    # exposedField  SFNode
}

MFRotationController {
    value      []      # exposedField  MFRotation
    index      []      # exposedField  MFInt32
    curve      NULL    # exposedField  SFNode
}

```

Curve nodes

typeCurve



When used in a Controller, Curve nodes can be used to animate Channel node values by way of a Filter. There is a specific curve type for the type of Channel to be animated. Curves work on a “key-frame” principle where authors explicitly define data points : pairs of time points and the values at those points. Then a curve is generated to fit those data points. At each frame, the Improv engine computes a value for each active Curve. For example in a `SFVec2fCurve` if the `key` field was `[0 .4 3]` and the `keyValue` field was `[1 3, 5 5, 9 1]` : at time 0 the curve would return values of 1 3, at time 3 the curve would return values of 9 1. At time .7 the curve would return an interpolation of points based on the `key` and `keyValue` fields and the type of the curve. Curves can also be used to manipulate State transitions, animation speeds and Function parameters.

<u>Field</u> type	<u>Description</u> The type of curve; how the values are computed: HERMITE : a smooth, spline curve fitting the keyValues. LINEAR : a straight line interpolation between the keyValues. SQUARE : a flat line curve between the keyValues.
mode	How the curve is traversed: CONSTANT : curve is traversed once and terminates on the final keyValue. LOOP : curve is traversed from start to finish, then jumps back to start and repeats. BOUNCE : curve is traversed from start to finish then backwards from finish to start then start to finish and so on.
key	A list of time values.
keyValue	A list of position values to be associated with the time values in the key field.
overlay	Another Curve or Function node to be combined with this current curve.
composite	How the current and overlay curve/function are combined: ADD : return the sum of the current curve value and the overlay curve value [CC + OC]. SUB : return the difference from subtracting the overlay curve value from the current curve value [CC - OC]. MULT : multiply the current curve value to the overlay curve value [CC * OC]. DIV : divide the current curve value by the overlay curve value [CC / OC]. MAX : return the larger of the two curve values [MAX(CC,OC)]. MIN : return the smaller of the two curve values [MIN(CC,OC)].

```

SFFloatCurve {
    type          "HERMITE" # exposedField SFString
    mode          "CONSTANT" # exposedField SFString
    key           0 # exposedField MFFloat
    keyValue      0 # exposedField MFFloat
    overlay       NULL # exposedField SFNode
    composite     "ADD" # exposedField SFString
}

SFVec2fCurve {
    type          "HERMITE" # exposedField SFString
    mode          "CONSTANT" # exposedField SFString
    key           0 # exposedField MFFloat
    keyValue      0 0 # exposedField MFVec2f
    overlay       NULL # exposedField SFNode
    composite     "ADD" # exposedField SFString
}

SFVec3fCurve {
    type          "HERMITE" # exposedField SFString
    mode          "CONSTANT" # exposedField SFString
    key           0 # exposedField MFFloat
    keyValue      0 0 0 # exposedField MFVec3f
    overlay       NULL # exposedField SFNode
    composite     "ADD" # exposedField SFString
}

MFFloatCurve {
    type          "HERMITE" # exposedField SFString
    mode          "CONSTANT" # exposedField SFString
    key           [] # exposedField MFFloat
    keyValue      [] # exposedField MFFloat
    overlay       NULL # exposedField SFNode
    composite     "ADD" # exposedField SFString
}

MFVec2fCurve {
    type          "HERMITE" # exposedField SFString
    mode          "CONSTANT" # exposedField SFString
    key           [] # exposedField MFFloat
    keyValue      [] # exposedField MFVec2f
    overlay       NULL # exposedField SFNode
    composite     "ADD" # exposedField SFString
}

MFVec3fCurve {
    type          "HERMITE" # exposedField SFString
    mode          "CONSTANT" # exposedField SFString
    key           [] # exposedField MFFloat
    keyValue      [] # exposedField MFVec3f
    overlay       NULL # exposedField SFNode
    composite     "ADD" # exposedField SFString
}

```

Function nodes

typeFunction

type	—
min	—
max	—
frq	—
curve_min	☐
curve_max	☐
curve_frq	☐
overlay	☐
composite	—

A Function node can be used anywhere a Curve node can be used. Function nodes allow you to define curves based on functions instead of data points.

Much like Controller nodes, you can specify the min, max, and frequency of the function using a curve instead of an explicit value. This allows the properties of the function to change in time. If any of the *curve_name* fields are non NULL, it will override the corresponding explicit field.

<u>Field</u>	<u>Description</u>
type	The type of function: NOISE : Perlin noise function. SIN : sine wave. COS : cosine wave.
min	The low bounds of the function.
max	The high bounds of the function.
frq	The frequency of the function.
curve_min	The curve that defines the minimum value.
curve_max	The curve that defines the maximum value.

curve_frq	The curve that defines the frequency value.
overlay	Another Curve or Function node to be combined with this current function.
composite	How the current and overlay function/curve are combined: ADD : return the sum of the current function value and the overlay function value [CF + OF]. SUB : return the difference from subtracting the overlay function value from the current function value [CF - OF]. MULT : multiply the current function value to the overlay function value [CF * OF]. DIV : divide the current function value by the overlay function value [CF / OF]. MAX : return the larger of the two function/curve values [MAX(CF,OF)]. MIN : return the smaller of the two function/curve values [MIN(CF,OF)].

```

SFFloatFunction {
  type          "NOISE"      # exposedField  SFString
  min           0            # exposedField  SFFloat
  max           0            # exposedField  SFFloat
  frq           0            # exposedField  SFFloat
  curve_min     NULL         # exposedField  SFNode
  curve_max     NULL         # exposedField  SFNode
  curve_frq     NULL         # exposedField  SFNode
  overlay       NULL         # exposedField  SFNode
  composite     "ADD"        # exposedField  SFString
}

```

```

SFVec2fFunction {
  type          "NOISE"      # exposedField  SFString
  min           0 0          # exposedField  SFVec2f
  max           0 0          # exposedField  SFVec2f
  frq           0 0          # exposedField  SFVec2f
  curve_min     NULL         # exposedField  SFNode
  curve_max     NULL         # exposedField  SFNode
  curve_frq     NULL         # exposedField  SFNode
  overlay       NULL         # exposedField  SFNode
  composite     "ADD"        # exposedField  SFString
}

```

```

SFVec3fFunction {
    type          "NOISE"      # exposedField SFString
    min           0 0 0        # exposedField SFVec3f
    max           0 0 0        # exposedField SFVec3f
    frq           0 0 0        # exposedField SFVec3f
    curve_min     NULL         # exposedField SFNode
    curve_max     NULL         # exposedField SFNode
    curve_frq     NULL         # exposedField SFNode
    overlay       NULL         # exposedField SFNode
    composite     "ADD"        # exposedField SFString
}

MFFloatFunction {
    type          "NOISE"      # exposedField SFString
    min           0            # exposedField MFFloat
    max           0            # exposedField MFFloat
    frq           0            # exposedField MFFloat
    curve_min     NULL         # exposedField SFNode
    curve_max     NULL         # exposedField SFNode
    curve_frq     NULL         # exposedField SFNode
    overlay       NULL         # exposedField SFNode
    composite     "ADD"        # exposedField SFString
}

MFVec2fFunction {
    type          "NOISE"      # exposedField SFString
    min           0 0          # exposedField MFVec2f
    max           0 0          # exposedField MFVec2f
    frq           0 0          # exposedField MFVec2f
    curve_min     NULL         # exposedField SFNode
    curve_max     NULL         # exposedField SFNode
    curve_frq     NULL         # exposedField SFNode
    overlay       NULL         # exposedField SFNode
    composite     "ADD"        # exposedField SFString
}

MFVec3fFunction {
    type          "NOISE"      # exposedField SFString
    min           0 0 0        # exposedField MFVec3f
    max           0 0 0        # exposedField MFVec3f
    frq           0 0 0        # exposedField MFVec3f
    curve_min     NULL         # exposedField SFNode
    curve_max     NULL         # exposedField SFNode
    curve_frq     NULL         # exposedField SFNode
    overlay       NULL         # exposedField SFNode
    composite     "ADD"        # exposedField SFString
}

```

Examples

Here you'll find two simple examples of authoring in Improv. First, you'll find a complete Actor node with routes to show what a basic animation node looks like. Then, you'll find an example of a sequence node.

Simple Actor

This example assumes you have a VRML model with a defined kinematic skeleton and body parts defined according to the ROUTE statements.

```
DEF IMPROV Improv {
  actors [
    Actor {
      name "Stick"
      channels [
        DEF ROOT_MOVEMENT SFVec3fChannel { name "root" }
        DEF LIDS_CONTROL SFRotationChannel { name "lids" }
        DEF PELVIS_CONTROL SFRotationChannel { name "pelvis" }
        DEF TORSO_CONTROL SFRotationChannel { name "torso" }
        DEF NECK_CONTROL SFRotationChannel { name "neck" }
        DEF HEAD_CONTROL SFRotationChannel { name "head" }
        DEF LUA_CONTROL SFRotationChannel { name "l_up_arm" }
        DEF RUA_CONTROL SFRotationChannel { name "r_up_arm" }
        DEF LUL_CONTROL SFRotationChannel { name "l_up_leg" }
        DEF RUL_CONTROL SFRotationChannel { name "r_up_leg" }
        DEF ROOT_ROTATION SFRotationChannel {
          name "root_rotation"
        }
      ] # end channels
    }
  ]
```

```

layers [
  Layer {
    name "Background"
    states [
      State {
        name "Background"
        filters [
          Filter { channel "root" }
          Filter { channel "root_rotation" }
          Filter { channel "lids" }
          Filter { channel "pelvis" }
          Filter { channel "torso" }
          Filter { channel "neck" }
          Filter { channel "head" }
          Filter { channel "l_up_leg" }
          Filter { channel "r_up_leg" }
          Filter { channel "l_up_arm" }
          Filter { channel "r_up_arm" }
          Filter { channel "l_up_leg" }
          Filter { channel "r_up_leg" }
        ] # end filters
      } # end State
    ] # end states
  } # end Layer

  Layer {
    name "Body Movement"
    states [
      State { name "No Movement" }
      State {
        name "Levitate"
        filters [
          Filter {
            channel "root"
            controller SFVec3fController { value 0 .4 0 }
          }
        ]
      }
      State {
        name "Turn Left"
        filters [
          Filter {
            channel "root_rotation"
            controller SFRotationController { xyz 0 30 0 }
          }
        ] # end filters
      }
    ]
  }
]

```

```

    State {
      name "Turn Right"
      filters [
        Filter {
          channel "root_rotation"
          controller SFRotationController {
            xyz 0 -30 0
          }
        }
      ] # end filters
    } # end State
  ] # end states
} # end Layer

Layer {
  name "Arm Movement"
  states [
    State {
      name "Arms Lowered"
      filters [
        Filter {
          channel "l-up-arm"
          controller SFRotationController { xyz 0 0 0 }
        }
        Filter {
          channel "r-up-arm"
          controller SFRotationController { xyz 0 0 0 }
        }
      ]
    }
    State {
      name "Wave"
      filters [
        Filter {
          channel "l-up-arm"
          controller SFRotationController {
            xyz -170 0 0
            curve_z Function {
              type "COS"
              min -25
              max 25
              frq 2
            }
          }
        }
      ] # end filters
    } # end State
  ]
}

```

```

State {
  name "Arms Raised"
  filters [
    Filter {
      channel "l-up-arm"
      controller SFRotationController {
        xyz -170 0 0
        curve_x Function {
          type "NOISE"
          min -165
          max -175
          frq 1
        }
      }
    }
    Filter {
      channel "r-up-arm"
      controller SFRotationController {
        xyz -170 0 0
        curve_x Function {
          type "NOISE"
          min -165
          max -175
          frq .8
        }
      }
    }
  ] # end filters
} # end State
] # end states
} # end Layer

```

```

Layer {
  name "Eye Movement"
  states [
    State {
      name "Blinking"
      filters [
        Filter {
          channel "lids"
          controller SFRotationController {
            value 0 0 1 0
            curve_x Curve {
              type "SQUARE"
              mode "LOOP"
              key [ 0, .75, .76, 1, 1.1, 1.35, 1.36 ]
              keyValue [ 0, 40, 0, 38, 0, 42, 0 ]
            }
            overlay Curve {
              type "SQUARE"
              mode "LOOP"
              key [ 0, 1.15, 1.16 ]
              keyValue [ 0, 40, 0 ]
            }
          }
          composite "MAX"
        } # end Curve
      ] # end controller
    } # end Filter
  ] # end filters
} # end State
] # end states
} # end Layer
] # end layers
} # end Actor
] # end actors
} # end Improv

```

ROUTES

```

ROUTE ROOT_MOVEMENT.value_changed T0 ROOT.set_translation
ROUTE ROOT_ROTATION.value_changed T0 ROOT.set_rotation
ROUTE LIDS_CONTROL.value_changed T0 L_LID.set_rotation
ROUTE LIDS_CONTROL.value_changed T0 R_LID.set_rotation
ROUTE PELVIS_CONTROL.value_changed T0 PELVIS.set_rotation
ROUTE TORSO_CONTROL.value_changed T0 TORSO.set_rotation
ROUTE NECK_CONTROL.value_changed T0 NECK.set_rotation
ROUTE HEAD_CONTROL.value_changed T0 HEAD.set_rotation
ROUTE LUA_CONTROL.value_changed T0 L_UP_ARM.set_rotation
ROUTE RUA_CONTROL.value_changed T0 R_UP_ARM.set_rotation
ROUTE LUA_CONTROL.value_changed T0 L_UP_LEG.set_rotation
ROUTE RUA_CONTROL.value_changed T0 R_UP_LEG.set_rotation

```

In this example, the Actor “Stick” is defined. First the Channels are defined to hold the values generated by the Animation Engine.

The first Layer is the default Layer with one State which creates a Filter for each Channel and sets it to the default value. This allows a smooth transition to any triggered State. Next a body movement Layer is defined followed by an arm movement and eye movement Layer.

The body movement Layer has four States. The no movement State is the default State since it is defined first. The levitate State raises the model while the turn left and turn right States rotate the model. Since levitate and turn right are in the same Layer, the Actor will “return to earth” when transitioning from “levitate” to “turn right”.

The arm movement Layer has four States as well. Here you can see the layering effects as you can have the Actor levitate and raise their arms by activating both States. A Sequence node let’s you create combinations of States. The noise functions in the arms raised State makes the arms twitch a little. The different frequency keeps them out of sync.

The eye movement Layer just has one State, blinking. Since each Layer has one active State and this layer has only one State, the Actor will always be blinking. The first curve has an overlay curve applied to avoid the eyes blinking in a recognizable pattern.

Notice how the Layers are listed in an order of increasing detail. This allows animations to be superimposed on top of each other with body part specific actions (blinking) adding detail to general body actions (levitate).

Also, notice that even though Channels and routes were created for the legs, there is no problem of not using them. Later, you could add a leg movement Layer to create complex cheer-leader positions for the Actor. Then, using a Sequence, make the Actor levitate while striking those poses to turn her into an action hero. Next, we’ll take a look at an example of a Sequence.

Simple Sequence

```
Layer {
  name "Sequence Tests"
  states [
    Sequence { name "Default" }
    Sequence {
      name "Move_Around"
      key [ 0 2 -1 -1 -1 15 ]
      actions [
        Trigger { states "Walk" }
        RandomTrigger {
          states [
            "Stand"
            "ShiftLeft"
            "ShiftRight"
          ]
        }
        Hold { time [ 2 10 ] }
        Trigger { states [ "Walk" ] }
        RandomTrigger {
          states [
            "TurnLeft-Slow"
            "TurnLeft-Medium"
            "TurnLeft-Fast"
            "TurnRight-Slow"
            "TurnRight-Medium"
            "TurnRight-Fast"
          ]
        }
        Trigger { states [ "Stand" "Default" ] }
      ]
    }
  ]
}
```

In this example the Actor begins walking, then at 2 seconds chooses to stand, shift left or shift right. Immediately following the start of this action, the Actor waits for 2 to 10 seconds at the end of which, she immediately begins to walk. Immediately following this, the Actor chooses from one of the 6 turning options. At 15 seconds the Actor stands and returns to the default behavior for this Layer (the “Default” sequence).

When a Trigger or RandomTrigger is activated, all the States or Sequences listed by a Trigger or chosen by a RandomTrigger are activated; in all Layers. So in the example above, at time 15, all States or Sequences named Stand and Default are activated. They remain active until another State or Sequence is called or the length of time in their duration field passes.

Appendix

The Improv Server

An Improv Server can be used to control the animations from an outside source by way of a Telnet connection. To run the Improv server, simply do the following:

Run the server on the machine serving the web page displaying the VRML world. You assign a port number for the server in the invocation. In this example, the port number is 9000:

```
java ImprovServer 9000
```

You'll need to inform the Improv applet that the server will be able to control the operation of the Animation Engine.

In your HTML file, set the Host and Port parameters of the applet accordingly

```
<PARAM NAME="Host" VALUE="HostAddress">  
<PARAM NAME="Port" VALUE="PortNumber">
```

HostAddress is the server's DNS address

PortNumber is replaced by the port number specified when invoking the Improv server.

To communicate with Improv, telnet to the server and send a string of the format:

```
@ ActorName StateName
```

ActorName is the name of the Actor to target

StateName is the name of the State to activate.

Improv will echo back the command to the telnet shell. The command will also be echoed in the server shell.

NOTE: The server must be running on the *same* machine that's serving the web page.

References

Please refer to the following documents for more information:

VRML:

VRML 2.0 Sourcebook, 2nd Edition : Ames, Nadeau, Moreland. John Wiley & Sons, 1996.

VRML 2.0 Specification : <http://vrml.sgi.com/moving-worlds/spec/index.html>

Java:

Java Language Specification 1.0 : http://www.javasoft.com/doc/language_specification.html

Improv:

Improv: A System for Scripting Interactive Actors in Virtual Worlds, SIGGRAPH 1996 Paper :
<http://www.mrl.nyu.edu/improv/sig-paper96/>