Linear Filters – Animating Objects in a Flexible and Pleasing Way

Herbert Stocker
Bitmanagement Software GmbH
herbert.stocker@bitmanagement.de

Abstract

In this work we propose an alternative animation approach to the traditional key frame based interpolation model. By way of illustration we propose a set of nodes that apply these principles to the X3D standard. In contrast to predefined key frame animations our way of defining animations allows an application to dynamically respond to the current situation and calculate an animation on the fly, while the content author can work with an extremely simple mental model for the animations. It is also our opinion that the way these nodes calculate an animation creates smooth and thus pleasing transitions. In addition, our node set can be used to approximate the effects of inertia, without the requirement and overhead of a heavy physics engine being present. With only a little of this inertia effect applied, objects (e.g. a slider thumb) can subjectively appear to have more physical substance.

Keywords

animation control, dynamics, linear filters, key frame animation, X3D, VRML, inertia, physical simulation, interaction, virtual reality

CCS Classification

I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism--Animation,Virtual Reality
I.6.8 [Simulation and Modeling]: Types of Simulation--Animation, Continuous, Discrete event
I.3.6 [Computer Graphics]: Methodology and Techniques--Interaction techniques

1. Introduction

Animations made with the standard means provided by X3D, the so-called Interpolator nodes which provide for key frame based animations, often lack the flexibility of responding to arbitrary user input or to conditions emerging at runtime. Also, if they are used in a low-bandwidth scenario they provide only jerky movements with a limited amount of realism. This paper proposes a set of nodes that overcome these limitations without requiring the author to use programming skills.

The proposed nodes can be seen as **linear filters** according to signal theory. They receive a time dependent value as their input, process it and output another time dependent value. The output value is calculated based on the current input value as well as on the past of the input value. Input can be received at discrete points in time or as a continuous series of values. We call these nodes 'Follower' nodes.

The Follower nodes serve to be useful in three scenarios:

1. **Creating animations.**
   A Follower node receives a new destination value, e.g. a position, at a single instance in time and creates a smooth transition from the current position to that destination position taking the current speed of movement into account.
2. **Smoothing existing animations.**
   A Follower node receives the output values of an Interpolator node, post-processes them and outputs a smoother version of the values changing over time. Such a transition may be more lifelike to end-users.
3. **Approximating inertia.**
   Usually when software processes user input in order to move an object, a viewpoint etc, then the data from the input device is mapped directly to the moving object. This makes the objects appear weightless and the created animation seems "robotic" or unnatural. Follower nodes overcome this by simulating a small amount of inertia.

Please see section *5.* Usage in X3D Content / Examples for a detailed description.

2. Motivation

The X3D standard has evolved from a file format for static 3D content (VRML 1), to a complex standard with many animation features and programmability options for creating interactive 3D applications. When it comes to creating animations it supports both novice users, who have no programming skills with nodes that allow them to create simple animations, as well as advanced programmers who can manipulate the scene graph dynamically.

However, the nodes that allow the creation of animations without using scripts – the Interpolator nodes in conjunction with the `TimeSensor` node – are limited to predefined animations. This means that the starting point, the end point and the path of an animation must be known at the design time of an X3D-based application. The standard nodes don't calculate an animation path from information that emerges at runtime, e.g. when fed with a position based on a user click.

Moreover, interactive applications have to respond to user input which is usually non-predictable. The simple case of a door that opens when the user clicks it and should close when they click it again, illustrates the dilemma for content authors with limtted or no skills in writing programmatic code. The dilemma arises when the author wants to specify the behavior that should occur when the user clicks the door while it is in transition. With Interpolator nodes the author can define the animation of an opening door and of a closing door, but not an animation from an arbitrary point in-between to either the opened or closed state.

Authors usually solve this by defining the animations between both states and disabling the ability to click the door while the door is in transition. Such solutions contribute to user frustration

because they make an application non-responsive at times. Other authors don't make provisions at all for this case and clicking the door while in transition creates a fan-out conflict for the two defined animations and the actual behavior of the door is left to the browser implementation. This leads to either a sharp jump in the door's motion at, or shortly after, the user click, which is never physically correct or aesthetically pleasing. Please see the example "door-classical.wrl" in [4] for an illustration of the issue. The example "door-desired.wrl" demonstrates how the issue should be solved.

Another simple, straight-forward scenario that shouldn't require the ability to program animations is the example of a simple interior design application. In such an application the user clicks a position on a wall and the selected piece of furniture moves to that location. Programming is required to calculate the right orientation of the piece, but should not be necessary to calculate the time-based transition of the object from the current position and orientation to the new one. This scenario is illustrated in "room-direct.wrl" and the desired behavior is shown in "room-desired.wrl" in [4].

From a usability point of view, transition animations help the user understand what's going on. In the interior design application, for example when the previous position is out of view the transition helps the user to understand that the piece did not simply materialize from nowhere, but instead came from somewhere offscreen in the direction implied by its motion.

In both scenarios, a node is desirable to which an application can send a destination position and which then calculates a transition from the current position to the desired one. In order to create a smooth transition it should also factor in the current speed of movement. Besides animations of positions, animations of orientations, colors, 2D coordinates, mesh coordinates, etc are also desirable, so that users only need to specify a new state and a transition is generated automatically.

This paper proposes a set of nodes that accomplish these tasks. Besides creating new transitions dynamically, these nodes also facilitate the smoothing of existing animations, which produces more natural, realistic results. For example, they can imbue user interface elements with more physical substance by projecting a feeling of inertia, or resistance to changes.

The problems solved in this paper have been addressed in the context of character, facial and physics animation et cetera ([6], [7], [8]), however, literature about this abstract scenario of simply and realistically transitioning from one value to another, without the onerous requirement of programming skills, is hard to find.

3. Linear Filters

The functionality of the animation nodes proposed in this paper can be implemented using the concept of linear filters. Linear filters shall be briefly explained in this section. Further information can be found at [1] and [2].

A filter in system theory is a device which accepts a time dependent value as its input, and outputs another time dependent value as its output. The value of the output at any instance in time depends on both the value of the input at that time and the values of the input in the past. Such a time dependent value is also called 'a signal'. A signal can be a scalar value (real number) as well as a multidimensional value (vector). If the discreteness of computer simulations is neglected, the output of Interpolator nodes can be seen as a continuous signal. A signal in the form of a two-dimensional vector is illustrated by the following equation:

$$v(t) = \begin{pmatrix} 10 * \cos(t * 3) \\ 10 * \sin(t * 3) \end{pmatrix}$$

Let *i(t)* be the signal input port of the filter, *o(t)* the signal output port and *T* the operator of the filter, then a filter can be described as:

$$o(t) = T[i(t)]$$

A linear filter is one which adheres to two useful conditions. Authors of X3D content need not understand these conditions in order to use Follower nodes, but these conditions explain how the Follower nodes work.

The first condition is that with a linear filter the output is independent of the time the input is supplied – if you supply the same input signal at a later time, it will respond with the same output as before.

$$T[i(t+\Delta)] = (T[i])(t+\Delta)$$

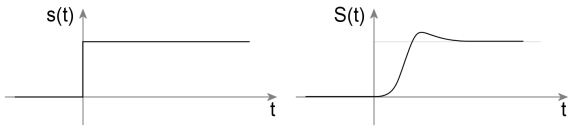
The other condition is that superimposing multiple input signals results in the same superimposition of the responses for each input signal – if you scale the input, the output will be scaled the same way, and if you send the sum of two signals to the input, the output will be the sum of the responses to the individual signals.

$$T[ai_1 + bi_2](t) = aT[i_1(t)] + bT[i_2(t)]$$

These two conditions lead to the fact that a linear filter can be described by the response signal to just one input signal. One such commonly used response function is the step response. The step response *S(t)* is the output of a filter that has received a step function *s(t)* as its input.

$$s(t) = \begin{cases} 0 & \forall\, t < 0 \\ 1 & \forall\, t \geq 0 \end{cases}$$

$$S(t) = T[s(t)]$$



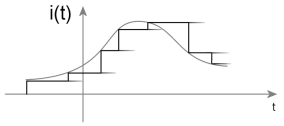
Since the Follower nodes proposed here should dynamically respond to input signals they receive at runtime, we need to limit our consideration to the so-called 'causal' filters. These filters produce output only after they receive an input signal or at the time they receive it. For the step response this means that the step response is *0* for *t<0*.

Another limitation to the set of filters that is applicable for Follower nodes is that the output value of the step response should move towards the value 1 for large t. If it would target another value, the output of a Follower node would never reach the value received as a destination value.

Any continuous signal can be approximated by the sum of a series of shifted and scaled step functions. The shifts for each step function need not necessarily be an integer multiple of some *Δt* value, they just need to be small enough. This allows for describing a signal with its value for each simulation tick in an X3D browser that usually does not have a constant delay between simulation ticks.

Because of this ability to approximate a signal by step functions, and due to the shift invariance and superimposition constraints for

linear filters, the output of a linear filter can be approximately described as the same combination of shifted and scaled step responses. The following diagram illustrates a smooth input function approximated by a series of step responses.



This means that if an input signal has been described by the approximation

$$i(t) = \sum_{n=-\infty}^{+\infty} a_n s(t - T_n)$$

then the output of the filter can be described as:

$$o(t) = \sum_{n=-\infty}^{+\infty} a_n S(t - T_n)$$

Linear filters can be put into two categories each of which has its drawbacks and advantages for authoring animations. These categories are **f**inite **i**mpulse **r**esponse (FIR) and **i**nfinite **i**mpulse **r**esponse (IIR) filter. Their names derive from the impulse response function, which is not used in this paper in favor of the step response. Basically, the impulse response is the first derivative of the step response.

3.1. Finite Impulse Response Filters

Finite impulse response (FIR) filters have an impulse response that goes to zero after a finite amount of time. For the step response this means that it reaches a static value after that amount of time and the output of a FIR filter does not change thereafter. Since only a value of 1 is applicable as the static value for creating Follower nodes, the step response of a FIR filter used here can be described with the following equation. Values other than 1 would create outputs that do not match the value of the inputs after a transition.

$$S(t) = \begin{cases} 0 & \forall\ t < 0 \\ f(t) & \forall\ 0 \le t < D \\ 1 & \forall\ t \ge D \end{cases}$$

Here D is the duration of the filter and f is a function that describes the transition from 0 to 1.

If the input to a FIR filter ceases at a certain point in time, the output also stops changing after a period of length D following that point in time.

3.2. Infinite Impulse Response Filters

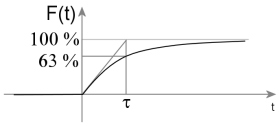
Infinite impulse response (IIR) filters have an impulse response that generally never goes to zero once they have received a non-zero input. They usually consist of a feed-back loop. However, they may approach the zero value asymptotically. For the step response of an IIR filter this means that their step response asymptotically approaches a certain output value. Useful for Follower nodes are those IIR filters whose step responses approach the value 1. Due to the causality constraint in real-world applications, IIR filters as used in this paper can be described with the following equations:

$$S(t) = 0 \quad \forall\ t < 0$$
$$S(t) \to 1 \quad \forall\ t \to \infty$$

3.3. Linear First-Order Filters

The type of IIR filters used here are linear first-order filters. They have an exponential step response described by the following equation:

$$S(t) = \begin{cases} 0 & \forall\ t \le 0 \\ 1 - e^{-\frac{t}{\tau}} & \forall\ t > 0 \end{cases}$$



Here τ is the time constant of the filter. It determines how fast the filter responds, i.e. how long it takes the output to reach the level of the input. Since the *e*-function never reaches zero for negative arguments, the output of a linear first-order filter reaches the input only approximately. However, it approaches it very fast due to the properties of the *e*-function. The parameter τ dictates the time required to reduce the output to 63 % of the difference from the input (*1 - 1/e*). After a few such periods the output can practically be said to have reached the value of the input.

The fact that the *e*-function is self-similar for scaling makes implementing linear first-order filters very simple and light-weight.

4. Proposed Nodes

The idea of a Follower node is that the application sends the Follower node a new destination value when the need for an object to change to a certain position, color or other type of value has emerged, and then the Follower node calculates a smooth transition to that value – the output of a Follower node seems to 'follow' its input.

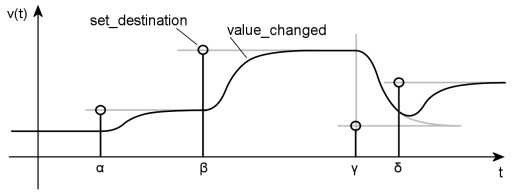
This idea leads to the below declaration of an `X3DFollowerNode`, which is an abstract node that has various specializations for different data types and for two different animation methods.

A Follower node has an input field called `set_destination` and an output field `value_changed`, thus forming a filter. If a Follower hasn't received input for a long time, its output has relaxed at a certain value – the value of the last received input – and no longer sends output anymore. The follower is inactive.

When such an inactive Follower receives a value on `set_destination` it begins sending events on `value_changed`. These events begin with the current value of the `value_changed` field and move gradually towards the value received until they reach that value, at which point `value_changed` stops sending events. In this way the Follower creates a smooth transition from the current to the desired value when it receives a new destination.

If a transition is currently in progress when a Follower node receives another destination value, it calculates a new animation that goes to the newly received destination starting at the value `value_changed` has at the time the Follower receives the new destination. It takes the current speed of movement (first derivative of the signal on `value_changed`) into account, and the transition created continues with this speed of movement for the first moments. The combined transition is smooth in that it

neither jumps nor suddenly changes the direction of movement.

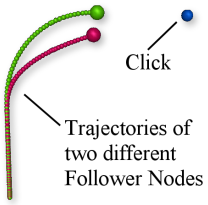


The above diagram illustrates this behavior: The circles mark the values that the Follower has received at certain instances in time and the curve shows the animation performed by the Follower on behalf of the destinations received. Before destination α has been received the Follower has relaxed at a certain value. Upon reception of α it creates a transition from this relaxation value to the new destination, whereupon it relaxes again. The transition features ease-in and ease-out. When the next destination β is received, another transition is created towards this new destination. Similarly for destination γ.

Destination δ follows γ very quickly, so that the animation triggered through γ has not yet been finished when δ is received. In this case the resulting animation after reception of δ smoothly diverges from the current animation until it becomes a transition towards the destination δ.

Follower nodes can also cope with continuous input on `set_destination`. If a Follower node receives a value on set_destination every simulation tick, and that value changes only minimallly on each event (like the transitions generated by Interpolator or Sensor nodes), then this creates another curve on `value_changed`. This curve follows the rules described in conjunction with event γ and δ above. The difference is that the distance between adjacent input events is much shorter and the Follower always has to calculate a new transition. The result is that the output curve is a smoother and slightly delayed version of the input curve.

Example "test_PosFollower.wrl" in [4] illustrates the behavior of Follower nodes. Click in the gray area to generate a single input value or click-drag the mouse to generate continuous input, then watch the two spheres following your input. They visualize the output of two implementation approaches of a Follower node.



This paper proposes two categories of Follower nodes, each of which services different needs and has unique advances and drawbacks. The first category takes only a limitted amount of time to reach the new destination. We call these nodes *Chaser nodes* because they reach their destination very quickly.

The other category is that of the *Damper nodes*. Damper nodes follow a dynamic equation that can be found in some physical systems (heat distribution, shock absorber - spring combinations, etc). Damper nodes approach their destination very smoothly. They actually don't reach it completely but approach it very quickly in an asymptotic way. In comparison to the Chaser nodes their implementation is more straight-forward and more light-weight.

In the abovementioned example the red sphere visualizes the output of an `X3DDamperNode`, and the green sphere visualizes the output of an `X3DChaserNode`.

4.1. Inheritance Structure

We propose an abstract node `X3DFollowerNode`, from which the `X3DChaserNode` and `X3DDamperNode` are derived nodes each having their specialization for the various data types. A few of the specializations are mentioned as examples, whilst others are conceivable.

```
X3DFollowerNode
+---   X3DChaserNode
|      +---   PositionChaser
|      +---   OrientationChaser
|      +---   PositionChaser2D
|      +---   ScalarChaser
|      +---   PlacementChaser
+---   X3DDamperNode
       +---   PositionDamper
       +---   OrientationDamper
       +---   ColorDamper
       +---   PositionDamper2D
       +---   CoordinateDamper
       +---   TexCoordDamper
       +---   PlacementDamper
```

The `PlacementChaser` and `PlacementDamper` are here for convenience in the cases where objects or viewpoints are moved. The `PlacementChaser` combines a `PositionChaser` and an `OrientationChaser`, and the `PlacementDamper` combines a `PositionDamper` and an `OrientationDamper`.

4.2. Definition of Chaser Nodes

`X3DChaserNode` nodes have a finite duration for their transition from the current value to the destination value. It can be specified as a field on the node. In the case in which the `X3DChaserNode` receives only a single destination value, the transition produced has the shape of a cosine function so that the resulting transition features ease-in and ease-out.

The following equation applies for an `X3DCahserNode` that has relaxed at the output value v_0 and receives the destination v_1 at time T_0. Its response duration has been set to D.

$$V(t) = \begin{cases} v_0 & \forall\ t \le T_0 \\ v_0 + (v_1 - v_0) R\left(\frac{t - T_0}{D}\right) & \forall\ T_0 < t < T_0 + D \\ v_1 & \forall\ t \ge T_0 + D \end{cases}$$

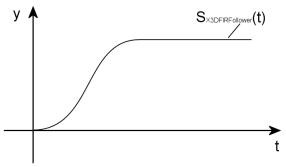
$$with \quad R(x) = \frac{1 - \cos \pi x}{2}$$

If an `X3DChaserNode` receives multiple destination values during a period of duration D, then each event received causes a similar response and all of them are added together (superimposed) in order to form the output. In that case the value v_0 is the value received before the current value received. This superimposition is in accordance with the ability to approximate signals by step functions described in 3. Linear Filters.

In terms of a mathematical description, an X3DChaserNode describes a linear filter with the step response

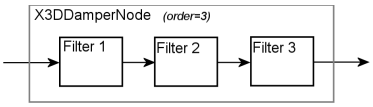
$$S_{X3DFIRFollower}(t) = \begin{cases} 0 & \forall\ t < 0 \\ \frac{1}{2}(1-\cos\frac{\pi t}{D}) & \forall\ 0 \le t < D \\ 1 & \forall\ t \ge D \end{cases}$$

$$with \quad D = duration\ of\ the\ transition$$



4.3. Definition of Damper Nodes

The X3DDamperNode uses an *e*-function for its step response. Its output asymptotically approaches the destination value received. X3DDamperNodes have an order parameter, which specifies how many linear first-order filters are chained together. In such a chain the n^{th} filter receives the output of its previous filter for n > 1 and the 1^{st} filter processes the input of the X3DDamperNode. Chaining filters together increases the smoothness of the output.



For the case in which the X3DDamperNode with *order=1* receives only a single destination value, the transition produced has the shape of a horizontally mirrored *e*-function.

The following equation applies for an X3DDamperNode with order=1 that has relaxed at the output value v_0 and receives the destination v_1 at time T_0. The horizontal stretch factor τ is a parameter of the node as well as the number of filters in the chain.

$$V(t) = \begin{cases} v_0 & \forall\ t \le T_0 \\ v_0 + (v_1 - v_0)E(t - T_0) & \forall\ t > T_0 \end{cases}$$

$$with \quad E(x) = 1 - e^{-\frac{x}{\tau}}$$

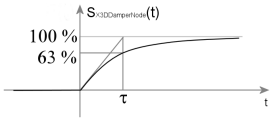
Since the response *V(t)* never reaches the value v_1 exactly but approaches it very quickly, there is a tolerance value *eps*, which specifies at what proximity of *V(t)* to v_1 the animation can be stopped.

If an X3DamperNode receives multiple destination values when the previous transitions have not yet finished, then each event received causes a similar response and all of them are added together (superimposed) in order to form the output. In that case the value v_0 is the value received before the current value received. This superimposition is in accordance with the ability to approximate signals by step functions described in 3. Linear Filters.

The step response of an X3DDamperNode with *order=1* is described with:

$$S_{X3DDamperNode}(t) = \begin{cases} 0 & \forall\ t \le 0 \\ 1 - e^{-\frac{t}{\tau}} & \forall\ t > 0 \end{cases}$$

with: τ = time constant of the filter.



4.4. Interface Descriptions

Here an abbreviated notation is used, where a derived node does not repeat the fields of its base type and the always present metadata field is omitted.

The abstract base types for all Follower nodes can be described as follows:

```
X3DFollowerNode: X3DChildNode {
  SFBool    [out]    isActive
}

X3DChaserNode: X3DFollowerNode {
  SFTime    []       duration
}

X3DDamperNode: X3DFollowerNode {
  SFFloat   [in,out]   tau     (0, ∞)
  SFInt32   []         order   (0, 5)

  SFFloat   [in,out]   eps     (0, ∞)
}
```

To describe each specialization, "<Type>" is used as a placeholder for the data type being animated, and "Xxxxx" is used as a placeholder for the descriptive component of the node name:

```
XxxxxChaser: X3DChaserNode {
  <Type>        [in]          set_destination
  <Type>        [out]             value_changed

  <Type>        []        initial_destination
  <Type>        []        initial_value
  <Type>        [in]          set_value
}

XxxxxDamper: X3DDamperNode {
  <Type>        [in]          set_destination
  <Type>        [out]             value_changed

  <Type>        []        initial_destination
  <Type>        []        initial_value
  <Type>        [in]          set_value
}
```

As can be seen, the interface for all Follower nodes is the same and differs only in the data type animated. The parameters defining the transition to be performed depends on whether a finite response is required or an exponential approach is sufficient.

The following assignments to the placeholders are possible:

Xxxxx	<Type>	Node name
Position	SFVec3f	PositionChaser, PositionDamper
Orientation	SFRotation	OrientationChaser, OrientationDamper
Color	SFColor	ColorChaser, ColorDamper
Scalar	SFFloat	ScalarChaser, ScalarDamper
Position'2D	SFVec2f	PositionDamper2D
Coordinate	MFVec3f	CoordinateDamper
TexCoord	MFVec2f	TexCoordDamper
Placement	SFVec3f, SFRotation	PlacementChaser, PlacementDamper

The PlacementChaser and PlacementDamper do not fit the above scheme well because there is no data type for holding the combination of a position and an orientation. MFFloat with 7 entries could be used, but this would complicate routes. The PlacementChaser and PlacementDamper are declared as follows:

```
PlacementChaser: X3DChaserNode {
  or
PlacementDamper:   X3DDamperNode {

  SFVec3f       [in]          set_destinationPos
  SFRotation    [in]          set_destinationOri
  SFVec3f       [out]             valuePos_changed
  SFRotation    [out]             valueOri_changed

  SFVec3f       []        initial_destinationPos
  SFRotation    []        initial_destinationOri
  SFVec3f       []        initial_valuePos
  SFRotation    []        initial_val ueOri
  SFVec3f       [in]          set_valuePos
  SFRotation    [in]          set_valueOri
}
```

Each field has been mirrored, it exists once for the position and once for the orientation.

4.5. Field Semantics

Follower nodes serve as filters in the context of signal theory. Therefore the fields of a Follower node serve to:

- provide input to the Follower;
- receive output from the Follower; and
- control the parameters of the filtering process.

All Follower nodes follow the same scheme of operation. They differ only in the data type on which they operate and whether it is a Chaser with a finite response time or a Damper with an exponential approach. Therefore the fields are, by way of example, described only for the PositionChaser and PositionDamper nodes. For the description of another Damper node, replace the data type SFVec3f with the respective type.

4.5.1.Fields Common for Followers and Dampers

SFVec3f [in] set_destination:

This sends input to the Follower. When a value is received, the Follower begins continuously sending values to value_changed. The values gradually grow from the current value of value_changed until they reach the value received on set_destination.

For a Chaser the length of this period is defined by the duration field. For a Damper it depends on the tau, order and eps parameters as well as on the current state and destination value of the Damper.

The set_destination field can be used either by sending it a single value at certain instances in time, each of which triggers a complete animation towards that value, or it can be used by sending it a gradually changing value on each simulation tick, so that the Follower serves as a post-processor, e.g. for the output of an Interpolator or Sensor node. If used with a single value at certain points in time, receiving a new destination value while the last transition is still in progress does not cause any irregularity. Instead, the Follower calculates a new transition based on the current and, in most cases, factoring in the speed of movement towards the new destination.

How the Damper decides when the destination value has been reached is specified by the value of the eps field. The Damper may not send an value_changed event at all if no animation is required to reach that value, i.e. if value_changed already has the value received and, in the case that order is larger than 0, if the Damper is already at rest.

In the special cases where order or tau is 0 for Dampers, or duration is 0 for a FIR Follower, the Follower simply forwards the value received to value_changed, without any processing.

SFVec3f [out] value_changed:
Emits the animation calculated by the Follower node. It usually sends values at times when there is not necessarily a stimulus to one of the input fields. The isActive field indicates whether value_changed is currently outputting values or not.

SFVec3f [] initial_destination,
SFVec3f [] initial_value:
These two fields allow initialization of the Follower to a certain state. If both fields are set to the same value, value_changed sends this value once on scene startup and then stops sending values. Alternatively, if the fields are set to different values, the Damper performs an animation from initial_value towards initial_destination on scene startup. The shape of that animation is that of the step response. This is to say it is the same as if it had received the value of initial_value a long time ago, and had received the value of initial_destination just at the moment of scene startup.

SFVec3f [in] set_value:
This allows an application to directly set the output of the Damper. When set_value receives a value, value_changed sends this value and thereafter animates from this to the destination. The shape of this animation is the same as that of the step response of the Follower, i.e. it is the same as if it had received the value sent to set_value on set_destination a long time ago, and had just received the current destination.

One can use this to force the output of the Follower node to jump immediately to a certain value. If the same value is sent to both set_value and set_destination, the Damper outputs that value and stays there. If that happens to be the value last output on value_changed, the effect is that the animation is immediately brought to a halt without a jump.

An application can set up an animation from one value to another by sending the 'from' value to set_value and the 'to' value to set_destination.

SFBool [out] isActive:
This indicates when an animation is being performed. It changes to `TRUE` when a new destination value is received via `set_destination` or when `set_value` forces `value_changed` away from the destination, and it flips to `FALSE` when `value_changed` has reached the destination value. For a Chaser not the flip to `FALSE` happens `duration` seconds after the last value has been received at `set_destination`. For a Damper this happens when the difference between `value_changed` and the destination falls below `eps`.

4.5.2.Fields specific to FIR Followers

SFTime [] duration (0,∞):
Specifies how long it takes the output value to reach the destination value.

4.5.3.Fields specific to Dampers

SFFloat [in,out] tau (0,∞):
Specifies how long it takes `value_changed` to reach the destination value. Strictly speaking a Damper never reaches its destination but nonetheless the destination is approached very quickly. After the period of time specified in the `tau` field, the distance of `value_changed` to the destination value has been reduced by 63 %. After the next such period of time the distance has again been reduced by 63 %, and so on. The value 63 % is derived from the euler number $e = 2.71828...$ by the equation $.63 = 1 - 1/e$.

SFInt32 [] order (0,5):
Specifies the number of linear first-order filters chained together internally. The larger the value of order is, the smoother the generated animation becomes, but the greater the overall delay between input and output is.

With values for `order` greater than 5, no improvement in smoothness is subjectively achieved, only the delay increases. Because of this, and the fact that other means can probably be implemented more efficiently to create a more accurate delay, only values up to 5 need to be supported for `order`.

In the case that `order` is 0, the Damper node just forwards every value received on `set_destination` to `value_changed` without any processing.

If `order` is 1, the whole Damper node is a linear first-order filter and the `value_changed` immediately changes the direction and/or speed of movement when a new destination value is received for `set_destination` at a single instance in time. For objects being moved by a `PositionDamper` and/or `OrientationDamper`, this effect can be quite noticeable to the user and can be desired.

If `order` is larger than 1, and a new destination value is received via `set_destination` while an animation is currently being performed, the values sent from `value_changed` continue to move with the current direction and speed. The Damper gradually changes these properties so that they eventually move towards the new destination.

SFFloat [in,out] eps (0,∞):
Theoretically a Damper never reaches its destination due to the properties of the e-function ($f(t)=e^{-t/\tau}$). The field `eps` allows authors to specify a threshold value, which the Damper uses to determine when it can assume to have reached the destination value. When the difference between output value and destination value becomes smaller than `eps`, the Damper sends the destination value via the `value_changed` field and stops the animation. This creates a little jump in the output, but this is always smaller than `eps`. The test is done individually for each internal filter, and the animation is only stopped if all of them are at rest.

5. Usage in X3D Content / Examples

Follower nodes can be used for creating new animations as well as for post processing existing animations or user input.

5.1. Creating Animations

As outlined in the sections above, Follower nodes can be used for easy creation of animations at runtime. When an object should move to a new place, change its color or should change shape (get new mesh coordinates), the application sends this value to the proper Follower node and the Follower node calculates a transition from the current value to the one given. No precautions have to be made for the case that the previous transition is still in progress. Instead of repeating the goal of the Follower nodes specified above, a few scenarios of application should be given here. The example files can be found at [4]. All the examples facilitate the following route structure:

User interface elements like buttons, drop-down boxes, menus, sliders, are still often done from scratch in context with X3D. These elements often change state, e.g. on mouse-over, button activation, slider movement, or fading in menus. With the Follower nodes authors need not be concerned about animating all these. They just insert a Follower node between the Script that calculates the color values, transparencies or positions and the geometry nodes of the UI element. Here Damper nodes can be prefered to Chaser nodes for their more light-weight implementation. The examples in [4] incorporate this concept.

The scenario of an interior design application where users can place objects in a room has already been outlined in section 2. Motivation. It's an example that demonstrates that with a Follower node consistent behavior can be achieved for **arbitrary input** and **if new destinations are received before the current transition is finished**. A `Script` node receives the mouse clicks onto the walls, calculates the proper object position and orientation and sends that information through a `PlacementFollower` to the `Transform` node containing the object. See "room-desired.wrl".

The door example shall also be mentioned here. It demonstrates the behavior when input is received while the current transition is still in progress. See "door_desired.wrl".

With **network communication** position or other updates come in with a quite low frequency. Yet smooth transitions are desired. This can be achieved by feeding the values received from the network through a Follower node. Example "MultiUser.wrl" demonstrates this in form of a simple multi-user world. Avatar positions are received about every .7 seconds and the `PlacementFollower` used has `duration` set to 1.

Example "test_OriFollower.wrl" demonstrates the use of Followers for rotations. On the top of the screen a few orientations can be predefined and then sent to the main object. The left side decides whether a Chaser or a Damper should be used. By rotating the below object on the right side a continuous signal can be sent to the Follower.

5.1.1. Initialization

Damper nodes can be set to perform animations directly after initialization. To do this, one must assign the starting point of the animation to `initial_value` and the destination to `initial_destination`.

By default, a Damper node initializes with zero as the state of the output and animates from there the first time it receives an input via `set_destination`. If this is not desired, the `initial_value` field can be used to initialize with another value. In that case, `initial_destination` should be assigned the same value, or an animation towards the zero state is performed after initialization.

5.1.2. Direct Control

When a Damper node receives a value via `set_value`, `value_changed` immediately goes to that value and starts a new animation towards the current destination. This allows for:

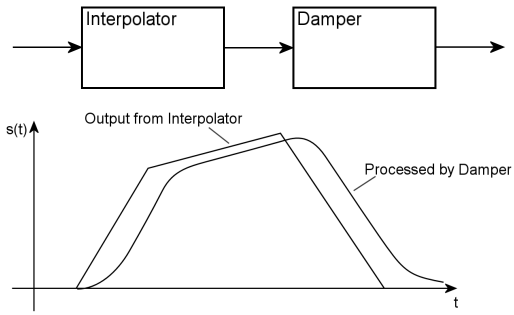
- stopping an animation if the value sent is the current value of `value_changed` and this value is also sent to `set_destination`;
- creating a jump prior to moving towards the destination if `set_destination` does not receive a value; or
- setting up new animations if both `set_destination` and `set_value` receive different values at the same time.

The last case is demonstrated in the user interface of the "test_OriFollower.wrl" example. Most buttons there exhibit a little flash when they are activated. All the author had to do for achieving this effect is to use a `ColorDamper`, set `initial_destination` and `initial_value` to the color of a non-highlighted button and send the color of a highlighted button to `set_value` each time a flash is to occur.

5.2. Smoothing Existing Animations

If fed with continuous input signals, i.e. the `set_destination` field receives a value almost every simulation tick, then the Follower nodes perform a low-pass filtering of the input signal. Fast changes in the input signal or edges are leveled out, and the resulting transitions become smooth, curved trajectories.

In combination with Interpolator nodes, this can be used to model smooth animations with a minimal number of key frame values.



By varying `tau` and `order` or `duration`, respectively the curve can be biased towards either accuracy or smoothness. Example "coaster-classical.wrl" and "coaster-damper.wrl" illustrate this behavior.

Another example of smoothing animations is given in "3rdPersonView-c.wrl" and "3rdPersonView-f.wrl". A 3rd person view is created through a Script node calculating an offset position and orientation to an avatar position and orientation. If fed directly to a Viewpoint node, the avatar appears stuck to the screen, like if being put into a HUD. If a `PlacementChaser` is inserted between the Script and Viewpoint, the avatar seams to move lifelike over the terrain.

5.3. Approximating Inertia

Although not creating a physically correct simulation, Dampers can be used to create the sensation of innertia. The delay introduced through Damper nodes and the effect of smoothening out the input signal create the sensation of inertia when they process user input that is used for creating motion.

User interfaces like sliders benefit from Dampers with small values for `tau`, say $\leq .3$. The controlled objects appear to become solid objects as opposed to hollow massless things. Such applications use the following route structure:

The example "test_Sliders.wrl" demonstrates this. It contains a few sliders that manipulate a box. By playing a bit with the sliders one can verify that a little smoothening with a Damper node does not impair the usability of the sliders. Clicking the box disables the Damper node. The object then appears massless/'computerish'.

With greater values for *tau*, say $\geq .7$ one can create some kind of inertia effect. "poor-mans-inertia.wrl" shows this effect: It is hard to accelerate, and once moving, it is hard to stop. Some work on the orientation should be done in this example in order to make it usable.

6. Implementation

Most of the nodes proposed here have been implemented as ExternProtos using EcmaScript. These nodes can be found at [4]. With the exception of the `PlacementDamper` all Damper nodes have been implemented completely and have been used in real-world applications. For the Chasers only the `PositionChaser`, `OrientationChaser` and `PlacementChaser` have been implemented for the sake of prove-of-concept. Trivial things like `set_output` or `initial_destination` have been omitted for now.

6.1. Chasers

In their implementations Chasers have to model the concept of a step response. This means that an array constituting the input signal history – the events received on `set_destination` – must be maintained. Since the shape of the transition is finite for Chasers, the input array needs to cover only a limited period of time, namely that extending from the current point in time back into the past by `duration` seconds. Due to the smoothness of the response function it is not necessary to remember each event received with its associated time stamp. It is sufficient to divide the history period into, say 20, equidistant slices, each of which is summarized by the latest received input value during that slice. This limits the number of evaluations of the step response function at each simulation tick to a certain value.

The Chaser nodes have a function `Tick(.)` which is evaluated each simulation tick. Due to the fact that the input to a linear filter

can be approximated by a series of step functions and the output is the sum of the step responses to each of those input steps, the `Tick(.)` function evaluates the current output value by evaluating the step response for each value in the history buffer and summing the results.

When the `Tick(.)` function is entered, the first thing it does is update the history buffer. As X3D browsers don't necessarily provide a constant frame rate, the function is not called on a regular basis, so it evaluates whether a new slice of time has been entered, and if so it shifts the contents of the history buffer towards the past so that it covers the right period of time. The newly freed slot in the history buffer is set to the latest value of `set_destination`. This code is outsourced into the function `UpdateBuffer(.)` and just contains time-stamp and buffer works, so that we don't include it here. See [4] for details.

```
function Tick(now) // will be called once for
{                  // every simulation tick.

  var Frac= UpdateBuffer(Now);
```

The value `Frac` we get from `UpdateBuffer(.)` is the amount of time we are ahead of the time-stamp of the latest value in the history buffer divided by the length of a time slice. Thus `Frac` is in the range $0 \leq$ `Frac` < 1.

The history buffer is designed to contain the latest received input value at index *0* and the values at index *n* have been recorded *n * Δt* seconds before that point in time, where *Δt* is the length of a time slice in seconds. Thus, the time an entry in the array has been recorded is $T_n = (n +$ `Frac`$) * \Delta t$ seconds ago.

`Tick(.)` simply calculates the difference between each entry in the history buffer and the previous entry, evaluates the step response function at the time of T_n, multiplies both and sums them up. For the oldest value in the buffer it uses the value itself because all input steps received before don't contribute to the shape of the response function anymore and their responses can be assumed to have reached that value already. This algorithm conforms to the equation:

$$O = \sum_{n=0}^{N-1} (B_n - B_{n-1}) R(n + Frac)$$

with

- B_n: Entry in the history buffer at index n.
- N: Number of entries in the history buffer.
- R: Response function as defined in section 4.2

```
  var Output= previousValue; // the value just
                         //shifted off the buffer.
  var DeltaIn= Buffer[Buffer.length - 1]
               .subtract(previousValue);
  var DeltaOut= DeltaIn.multiply(StepResponse(
        (Buffer.length - 1 + Frac) * cStepTime));
  Output= Output.add(DeltaOut);
  for(var C= Buffer.length - 2; C>=0; C-- )
  {
    var DeltaIn= Buffer[C].subtract(Buffer[C+1]);
    var DeltaOut= DeltaIn.multiply(StepResponse(
                         (C + Frac) * cStepTime));
    Output= Output.add(DeltaOut);
  }
  value_changed= Output;
}
```

6.2. OrientationChaser and SFRotation

The iteration in a Chaser is basically calculating the following equation:

$$O_n = O_{n-1} + (B_n - B_{n-1}) R(n + Frac)$$

where O_n is the sum after iteration *n*. With the Term substitutions

$$\alpha = R(n + Frac), \quad A = B_n - B_{n-1}$$

the equation can be written as:

$$O_n = O_{n-1} + A\alpha$$

This can be interpreted as going from one point O_{n-1} towards A by the relative amount of α, even if α is a bit out of the range 0..1. The `slerp(.)` method available on SFRotation objects does just that.

$$O_n = O_{n-1}.slerp(A, \alpha)$$

A can be calculated using the operator replacements:

$$X + Y \mapsto X.multiply(Y), \quad X - Y \mapsto Y.inverse().multiply(X)$$

Therefore Chaser nodes can also be implemented for orientation values.

6.3.Dampers

6.3.1. Core Formula

The core formula of a Damper node is the step response of a linear first-order filter:

$$F(t) = \begin{cases} 0 & \forall\, \mathbf{t < 0} \\ 1 - e^{-t/\tau} & \forall\, \mathbf{t \geq 0} \end{cases}$$

Usually in digital signal processing this step response is evaluated via the following equation, where the value of the output for the current simulation step is calculated from the output value at the last simulation step and the current input.

$$o_n = (1 - \alpha) o_{n-1} + \alpha i_n$$

Here o_n is the output value at simulation step n, i_n is the input value at simulation step n and α is a parameter that depends on τ and the time between simulation steps.

This equation requires the time between two simulation steps to be constant. X3D players are commonly best-effort systems, which try to run as fast as possible. No constant delay between two simulation ticks can be assumed for them. Therefore we have used the impulse response directly.

If a linear first-order filter receives a new input value every simulation tick or less often, then the real signal can be seen as approximated by the sum of a series of scaled step functions which have their temporal origin shifted towards the point in time of the simulation tick they belong to. The scale factor is the difference between the current signal value and the previous one.

$$i(t) \approx \sum_{n=-\infty}^{+\infty} (i(t_n) - i(t_{n-1}))\ s(t - t_n)$$

Due to the superimposition principle of linear filters (see section 3. Linear Filters) the output can be calculated as a similar sum of step responses.

$$o(t) \approx \sum_{n=-\infty}^{+\infty} (i(t_n) - i(t_{n-1}))\ S(t - t_n)$$

This leads to the formula below, which is used for calculating the output value for the current simulation tick from the value of the previous tick and the last received input value.

$$o(t_n) = i(t_n) + (o(t_{n-1}) - i(t_n))\, e^{\frac{-\Delta t}{\tau}}$$

Here $\Delta\tau$ is the time between the current and last simulation tick.

Since the Damper nodes contain one or more such filters, depending on the value of the order field, this leads to the following code in the EcmaScript implementation of the Damper. The code snippet is taken from the `PositionDamper`.

```
function Tick(now) // will be called once for
{                  // every simulation tick.

  var delta= now - lastTick;
  var alpha= Math.exp(-delta / tau);

  value1= order > 0 && tau
         ? input .add(value1.subtract(input)
                      .multiply(alpha))
         : input;

  value2= order > 1 && tau
         ? value1.add(value2.subtract(value1)
                       .multiply(alpha))
         : value1;

...

  value5= order > 4 && tau
         ? value4.add(value5.subtract(value4)
                      .multiply(alpha))
         : value4;
```

The remainder of the `Tick(.)` function contains the code described in the following section 6.3.2. Endign the Animation, a statement that outputs the newly calculated value and some house-keeping.

```
  <end detection>

  value_changed= value5;

  lastTick= now;
}
```

6.3.2. Ending the Animation

Due to the *e*-function in the step response of a first-order filter the output will never reach the destination value exactly. For practical reasons the animation calculations should stop after the destination has nearly been reached. The following code calculates the input-output distance for each internal filter and stops the animation if all are below the limit specified by the `eps` field.

```
var dist= value1.subtract(input).length();
if(order > 1)
{
    var dist2=
           value2.subtract(value1).length();
    if(dist2 > dist)  dist= dist2;
}
if(order > 2)
{
    var dist3=
           value3.subtract(value2).length();
    if( dist3 > dist) dist= dist3;
}

...


if(dist < eps)
{
    value1= value2= value3= value4=
             value5= input;
    value_changed= input;
    StopTimer();
return;
}
```

This code snippet is to be inserted in the `Tick(.)` function described in 6.3.1 Core Formula at the place marked with *<end detection>*.

6.3.3. Performance Issues

The `Tick(.)` function, which is called once for each simulation tick, is a straight block of consecutive statements. There is no loop and no recursion, which would cause a significant amount of CPU utilization if executed.

6.4. OrientationDamper and SFRotation

Similar to `OrientationChasers`, with the substitutions

$$\alpha = e^{\frac{-\Delta t}{\tau}}, \quad A = i(t_n), \quad B = o(t_{n-1})$$

the core term

$$o(t_n) = i(t_n) + (o(t_{n-1}) - i(t_n))\, e^{\frac{-\Delta t}{\tau}}$$

for a linear first-order filter can be written as:

$$o(t_n) = A + \alpha\,(B - A)$$

Since α is in the range 0..1, this can be interpreted as going from one point A towards B by the relative amount α.

The `slerp(.)` function available in EcmaScript in X3D players does exactly this for `SFRotation`s:

$$o(t_n) = A.slerp(B, \alpha)$$

Therefore the `OrientationDamper` can be implemented using the `slerp(.)` function.

7. Comparing Chasers and Dampers

Chaser nodes and Damper nodes each address the same kind of problems, namly dynamically creating animations. Therefore their benefits and drawbacks should be contrasted.

Chaser nodes feature a finite transition time, after which they come at rest completely. There is a clear point in time at which a transition has ended, and follow-on actions can be triggered.

Their drawback is that their implementation is more complex than with Damper nodes, however, with nowadays computers this will not be a problem unless a huge number of instances are used in a scene.

Damper nodes, due to their dynamic equation being closer to physical systems, create slightly more natural looking transitions.

Their drawback is that there is no clear end of an animation, however, this contributes to aestetics. This makes it difficult to trigger follow-on actions. Tweaking the `eps` parameter causes either a little jump noticable at the end of an animation, or the animation easing out for too long with no visual effect.

However, due to its straight-forward implementation a Damper node is very light-weight and user interface design, where many of its instances could be used, is a good application for Damper nodes.

8. Conclusion

In this work we outline the need for a flexible scheme to create animations based on data available only at runtime of a 3D application. We develop a set of nodes, which we call *Follower* nodes. It allows content creators to author transitions by just indicating which new value a certain parameter should assume and how much time the transition can take to perform. The animations generated are smooth and stable, even for orientations. Due to the easy application of the Follower nodes user interfaces can be made richer and conventional key frame based animations can be smoothed. Proof of concept is given through an implementation using EcmaScript and a rich set of examples. The implementation has been described in principle.

We plan to implement the missing nodes of the proposed node set. We believe that Follower nodes are a general means of creating animations and could be useful as part of the X3D specification. Therefore we plan to implement them as native nodes in our X3D player family BS Contact. We also want to investigate second-order filter, as they exhibit the behavior of physical systems like spring-mass-damper combinations. For the Chaser nodes we proposed to use a cosine based step response. A parameter could be added to switch to other step responses.

References

[1] LATHI, B. P. 2001. *Linear Systems and Signals*

[2] HEEGER, Pr. D. 2000. *Signals, Linear Systems, and Convolution*

[3] OPPENHEIM, WILLSKY, A. S., and YOUNG, I. T. 1983 *Signlas and Systems*. Prentice-Hall, Englewood Cliffs, New Jersey

[4] Followers home page, (Feb. 2006). http://www.hersto.com/redir.php?Followers

[5] Bitmanagement, BS Contact VRML/X3D home page, (Dec 2005). http://www.bitmanagement.de/products/bs_contact_vrml.en.html

[6] *Dynamic Motion Synthesis*, (Feb. 2006). http://www.naturalmotion.com/files/dms_wp2005.pdf

[7] STEWARD, J.A., and CREMER C.F. *Beyond keyframing: an algorithmic approach to animation (Proceedings of the conference on Graphics interface 1992)* p. 273-281

[8] HODGINS, J. K., SWEENEY, P. K., and LAWRENCE, D. G. *Generating natural-looking motion for computer animation. (Proceedings of the conference on Graphics interface 1992)* p. 265-272