



hersto: Specification draft for the Follower nodes

This document gives a first draft of a specification of the *Follower* nodes that we are discussing to put into the X3D spec (ISO/IEC 19775:2004).

Introduction

Overview

The group of *Follower* nodes allows to create transitions of parameters at runtime (dynamically) by receiving a destination value upon which they create an animation that transitions their output value from its current value towards the newly set destination value.

In case a transition triggered by reception of a previous destination value is not yet finished while the new destination is received, both, the new and old transition is merged, so that a smooth animation is created, where the previous movement degrades and gradually becomes a movement towards the new destination which is then eventually being reached.

Follower nodes accomplish this by implementing **f**inite **i**mpulse **r**esponse (FIR) filters and **i**nfinite **i**mpulse **r**esponse (IIR) filters according to the subject of system theory. Due to this distinction the group of *Follower* nodes is divided into the group of *Chaser* nodes (FIR) and the group of *Damper* nodes (IIR).

Followers are a bit like *Sensor* nodes because they usually send events at times where they don't receive events, however their behaviour is completely determined by the events they receive from the scene graph itself at earlier times.

Follower nodes are not affected by their position in the transformation hierarchy nor are they affected by the state of containing *Switch* nodes, *LOD* nodes and other nodes that affect the visibility of their children.

Concepts

Node Hierarchy

In this node hierarchy chart abstract nodes are formatted in italicic.

```

X3DFollowerNode
|
+---- X3DChaserNode
|      +---- PositionChaser
|      +---- OrientationChaser
|      +---- PositionChaser2D
|      +---- ScalarChaser
|
+---- X3DDamperNode

```

```

+--- PositionDamper
+--- OrientationDamper
+--- ColorDamper
+--- PositionDamper2D
+--- CoordinateDamper
+--- TexCoordDamper

```

Abstract types

X3DFollowerNode

```

X3DFollowerNode : X3DChildNode {
  [S|M]F<type> [in] set_destination
  [S|M]F<type> [out] value_changed

  [S|M]F<type> [] initialDestination
  [S|M]F<type> [] initialValue
  [S|M]F<type> [in] set_value

  SFBool [out] isActive

  SFNode [in,out] metadata NULL [X3DMetadataObject]
}

```

The abstract node `X3DFollowerNode` forms the basis for all nodes specified in this clause. The data type place holder `[S|M]F<type>` evaluates to the same data type for all fields of a specialization of the abstract node class `X3DFollowerNode`.

An `X3DFollowerNode` maintains an internal state that consists of a *current value* and a *destination value*. Both values are of the same data type the term `[S|M]F<type>` evaluates into for a given specialization. It is the '*data type of the node*'. Alternative terms for the state variables are *input* for *destination value* and *output* for *current value*.

Whenever the *current value* differs from the *destination value*, the current value gradually changes until it reaches the *destination value* producing a smooth transition. It generally grows towards the *destination value* but can also temporarily grow into another direction if the requirement of smoothness dictates this. This may happen at a short time period after a new *destination value* has been received if the previous transition is still in progress.

The `value_changed` outputOnly field exposes the *current value* of the internal state. Whenever the *current value* changes, `value_changed` sends out updates to the scene.

The `set_destination` field sends a new *destination value* to the `X3DFollowerNode`. As a result the `value_changed` field begins sending values in most cases.

The initializeOnly fields `initialDestination` and `initialValue` initialize the internal state of the `X3DFollowerNode`. The *current value* receives the value of `initialValue` and the *destination value* receives the value of `initialDestination`. If both fields have the same values, the `X3DFollowerNode` sends that value through the `value_changed` field in a single event upon initialization. If both fields have different values, the `X3DFollowerNode` creates an animation from the value of `initialValue` towards the value of `initialDestination`. The shape of that transition is the same as if the *current value* internal state had always been at the value of `initialValue` and the node had just received the *destination value*.

With the `set_value` inputOnly field one can immediately force the *current value* towards a certain value. When the `X3DFollowerNode` receives a value on `set_value`, any current transition is stopped and the *current value* assumes that value. The `value_changed` field outputs that value and then moves towards the value currently set for the *destination value*. This animation has the same shape as if the *current value* had already been at the newly

received value for a long time and the node had just received an event on `set_destination` carrying the value of the currently set *destination value*.

One can achieve various results by sending certain values to `set_value`, `set_destination` or both at the same time:

- `set_destination` and `set_value` receive different values:
Then a transition is created that goes from the value of `set_value` towards the value of `set_destination`. The transition is independent of the previous history of the node. With most parameter settings the transition starts with zero speed and then accelerates towards the destination.
- `set_destination` and `set_value` receive the same value:
Then `output_changed` assumes that value immediately and stays there. No Transition is created.
- `set_value` receives the value `value_changed` currently has:
Then `value_changed` stops moving immediately and begins a new transition towards the currently set *destination value*. With most parameter settings the result is that `value_changed` stops moving and then accelerates towards the *destination value*, which it was targeting to already before.
- `set_value` receives the value currently set as destination:
Then the `output_changed` jumps to the *destination value* immediately.
- `set_destination` and `set_value` both receive the current value of `value_changed`:
Then the transition produced comes to an immediate halt at its current value.

The `isActive` `outputOnly` field indicates the beginning and end of a transition. It sends *TRUE* before `set_value` begins animating and it sends *FALSE* after `set_value` has reached the *destination* or has been stopped by another means.

X3DChaserNode

Fields inherited from the `X3DFollowerNode` are formatted in grey color for clarity.

```
X3DChaserNode: X3DFollowerNode {
  [S|M]F<type>  [in]    set_destination
  [S|M]F<type>  [out]   value_changed

  [S|M]F<type>  []      initialDestination
  [S|M]F<type>  []      initialValue
  [S|M]F<type>  [in]    set_value

  SFBool        [out]   isActive

  SFNode        [in,out] metadata    NULL [X3DMetadataObject]

  SFTime        []      duration    [0..8]
```

An ideal `X3DChaserNode` calculates the output on `value_changed` as a finite impulse response (FIR) based on the events received on `set_destination` in the following way:

Each time an event is received on `set_destination` a transition A_n from the previously received destination to the new destination is created according to the following equation. The data types of all variables are floating point numbers or

integers in case of indices, except for d_n , d_{n-1} , $A_n(t)$ and $O(t)$. These variables have the data type of the node, which is `SFVec3F`, `SFColor`, and so on.

$$A_n(t) = \begin{cases} d_n - d_{n-1} & \forall t \geq T_n \\ (d_n - d_{n-1})R\left(\frac{t - T_n}{D}\right) & \forall T_n < t < T_n + D \\ 0 & \forall t \geq T_n + D \end{cases} \quad (\text{I})$$

Here T_n is the point in time where the event has been received, D is the value of the `duration` field, d_n is the new destination value received with the event, d_{n-1} is the value that was the destination before the event and $R(x)$ is the core function of the filter:

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

All the transitions created for every event on `set_destination` are added together to form the output on `value_changed`.

$$O(t) = d_{k-1} + \sum_{n=k}^l A_n(t) \quad (\text{III})$$

l (ell) is the number of events received so far on `set_destination`. If k is set to 0, d_{-1} is the value of the `initialValue` field and d_0 is the value of `initialDestination`. This way the initial transition determined by these two fields is produced.

Theoretically the start index k could be always set to 0 (zero) meaning that all `set_destination` events since initialization need to be stored. However, k can be increased without changing the result $O(t)$ as long as the time stamp T_{k-1} is more than D seconds before the current time stamp. This is due to the facts a) that after a period of D seconds (the `duration` field) the transitions $A_n(t)$ are constantly $d_n - d_{n-1}$ and b) that d_{k-1} is the sum of all differences $d_n - d_{n-1}$ so far.

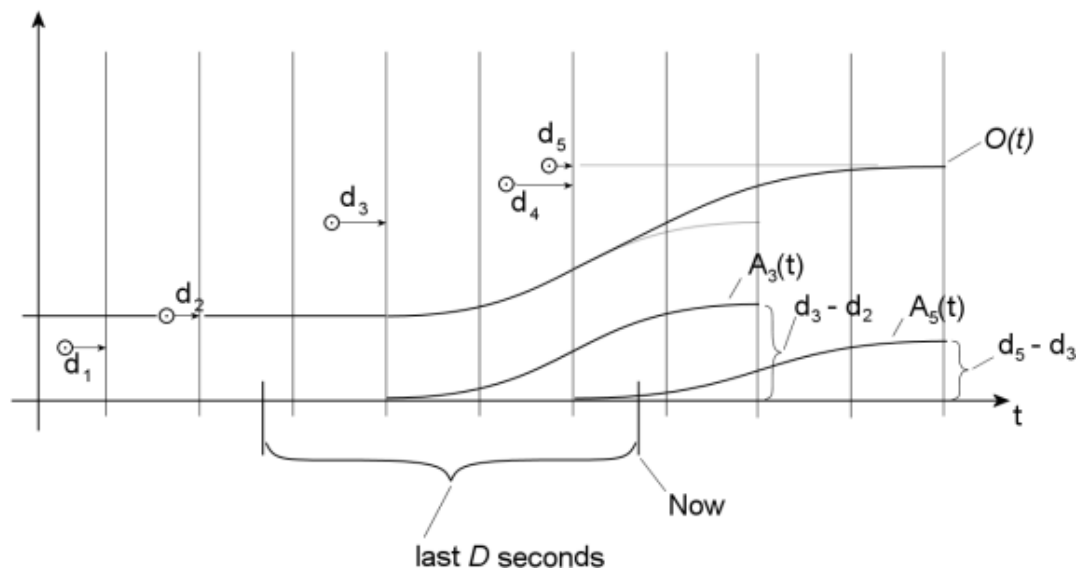
This way the ideal `X3DChaserNode` implementation has to remember the values and time stamps of all `set_destination` events received in the last period of `duration` seconds, plus the value received latest before that period. For calculating the current value of `value_changed` it uses that latest received value as a starting point (d_{k-1}) and adds to it all transitions $A_n(t)$ generated by the stored events.

The proposed, more optimal implementation could divide the tim-line into equidistant time-slots and store only the latest `set_destination` event received for each time-slot. This way a fixed length array could be used for describing the input during the period of the last `duration` seconds. This however could create little jumps in the animation created at `value_changed` because a `set_destination` event may cause the beginning of a transition being produced and may then be replaced by a later event received in the same time-slot. To avoid this, events should be associated with the end of the time-slot rather than with the time-stamp they are received at.

This of course causes the output reach the value received at `set_destination` up to the length of a time-slot later than is dictated by the `duration` field. To compensate, an implementation should subtract the length of a time-slot from `duration` and use the result for D . It might be bad finishing a bit too late, but

finishing too early should never be a problem.

It is proposed that the implementation uses (about) 10 time-slots per duration duration.



The above diagram illustrates how an implementation calculates the output at an arbitrary point in time. It uses only 4 time-slots per duration D . The period goes from the current time-stamp *Now* back by D seconds, not necessarily matching the grid of the time slots. The events d_1 and d_2 have happened before this period and are therefore summarized by the value of d_2 . The event d_3 however falls into the period of D seconds. It is moved towards the end of the time-slot it falls into and generates the transition $A_3(t)$ with the amplitude $d_3 - d_2$. The event d_4 gets ignored because it is followed by d_5 in the same time-slot. Therefore only d_5 generates a transition, which is $A_5(t)$. The amplitude of $A_5(t)$ is $d_5 - d_3$ because d_4 got ignored. The output $O(t)$ is thus calculated by adding:

$$O(t) = d_2 + A_3(t) + A_5(t) \quad (IV)$$

When the current time-stamp has advanced until after the end of curve $A_3(t)$, which is when the time-slot containing event d_3 is no longer part of the last D seconds, the start value for the addition d_2 is replaced with d_3 and the curve $A_3(t)$ is removed from the addition, so that $O(t) = d_3 + A_5(t)$.

The above diagram uses 4 time slots per duration D . With the above recommendations of making D one time-slot shorter than the `duration` field specifies, this means that a time-slot is a 5th of what is specified by `duration`.

X3DDamperNode

An *X3DDamperNode* abstract node uses the following signature:

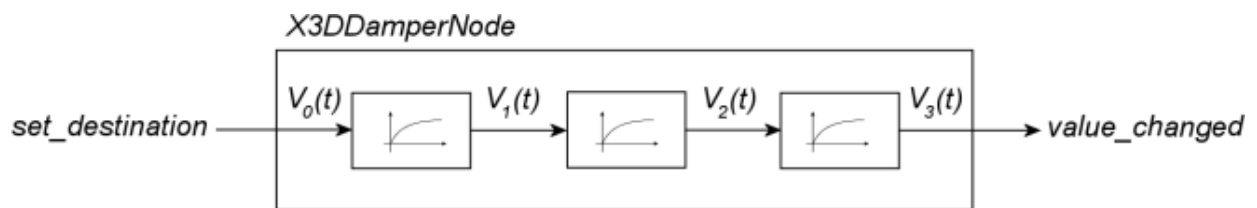
```
X3DDamperNode: X3DFollowerNode {
  [S|M]F<type> [in] set_destination
  [S|M]F<type> [out] value_changed

  [S|M]F<type> [] initialDestination
  [S|M]F<type> [] initialValue
  [S|M]F<type> [in] set_value
```

SFBool	[out]	isActive	
SFNode	[in,out]	metadata	NULL [X3DMetadataObject]
SFTime	[in,out]	tau	[0..+8]
SFInt32	[]	order	[0..5]
SFFloat	[in,out]	tolerance	-1 or [0..8]

The `X3DDamperNode` creates an IIR response which approaches the destination value only asymptotically but very quickly according to the shape of the e-function.

An `X3DDamperNode` node is parametrized by the `tau`, `order` and `tolerance` fields. Internally it consists of a set of linear first-order filters which each process the output of the previous filter. The input of the first filter is fed by the values received on `set_destination` and the output of the last filter goes to the `value_changed` field.



The calculations of the output for the current time-stamp T_n for each filter are based on the output of that filter from the previous time-stamp T_{n-1} and the current input using the following equation:

$$o_n = d_n + (o_{n-1} - d_n) e^{\frac{-\Delta T}{\tau}} \quad (V)$$

$$o_n = V_k(T_n)$$

$$o_{n-1} = V_k(T_{n-1})$$

$$d_n = V_{k-1}(T_n)$$

$$\Delta T = T_n - T_{n-1}$$

The field `order` specifies the number of such internal filters. Specifying 0 (zero) for `order` means that no filter is used. In this case the events received on `set_destination` are forwarded directly to `output_changed`. The larger the value for `order` the smoother the output on `value_changed` will be, but the more delay will be introduced. Since values larger than 5 don't introduce any more smoothing the range for `order` is limited to a maximum of 5.

The field `tau` specifies the time-constant of the internal filters, and thus the speed the output of an `X3DDamperNode` responds to the input. Its value is assigned to the variable t in the above equation. A value of 0 for `tau` means immediate response and the events received on `set_destination` are forwarded directly to `output_changed`. The field `tau` specifies how long it takes the output of an internal filter to reach the value of its input by 63 % ($1 - 1/e$). The remainder after that period is reduced by 63 % during another period of `tau` seconds and so on, provided that the input of the filter does not change. This behavior can be exposed if `order` is set to 1 (one).

Since the output of an `X3DDamperNode` approaches the input value only asymptotically, there must be a means to determine when the destination value

can be assumed to be reached and the node can stop emitting values and set `isActive` to `FALSE`. This is governed by the `tolerance` field. If `tolerance` is left at its default value -1 then it's up to the browser implementation to find a good way for detecting the end of a transition. Browsers that do not have an elaborate algorithm can just use .001 as the tolerance value instead. If a value larger than 0 (zero) is specified for `tolerance` then the browser must calculate the difference between output and input for each internal filter being used and stop the animation only when all filters fall below that limit or are equal to it. If 0 (zero) is specified for `tolerance` then a transition should be stopped only if input and output match exactly for all internal filters. This can for example happen if `set_value` receives an event.

Implementations should do the test for end of transition before they calculate the new output value and then either assign the *destination value* to the *output value*, if the difference falls below the tolerance limit, or calculate an updated output value.

Node reference

PositionChaser

```
PositionChaser: X3DChaserNode {
  SFVec3f      [in]      set_destination
  SFVec3f      [out]     value_changed

  SFVec3f      []        initialDestination
  SFVec3f      []        initialValue
  SFVec3f      [in]     set_value

  SFBool       [out]     isActive

  SFTime       []        duration      [0..+8]

  SFNode       [in,out]  metadata      NULLL [X3DMetadataObject]
```

The `PositionChaser` creates transitions for 3D Vectors.

OrientationChaser

```
OrientationChaser: X3DChaserNode {
  SFRotation   [in]     set_destination
  SFRotation   [out]    value_changed

  SFRotation   []       initialDestination
  SFRotation   []       initialValue
  SFRotation   [in]    set_value

  SFBool       [out]    isActive

  SFTime       []       duration      [0..+8]

  SFNode       [in,out] metadata      NULLL [X3DMetadataObject]
```

The `OrientationChaser` creates transitions for Orientations.

In order to implement the `OrientationChaser` node equations (I), (II) and (III) can be combined to equation (IV):

$$O(t) = d_{k-1} + \sum_{n=k}^i (d_n - d_{n-1})R(\dots) \quad (\text{VI})$$

This leads to the following loop denoted in pseudo code:

```
var Result=  $d_{k-1}$ ;
for(var n from k to l) {
  var Delta=  $d_n - d_{n-1}$ ;
  Result+= Delta *  $R(\dots)$ ;
}
 $O(t)$ = Result;
```

Since d_{k-1} , d_n , d_{n-1} and thus `Result` contain rotation values (`SFRotation`), the above code must be converted to use operations available for rotations. This can be achieved using the `slerp` operation. For the following let `slerp(A, B, t)` be a function that calculates the linear spherical interpolation from A to B by the amount t . Let also `Core(.)` be a function that calculates $R(\dots)$ and let `Buffer` be an array so that `Buffer[i]` evaluates to d_i . Then the above loop can be implemented as:

```
var Result= Buffer[k-1];
for(var n from k to l) {
  var Delta= Buffer[n-1].inverse().multiply(Buffer[n]);
  Result= slerp(Result, Result.multiply(Delta), Core(\dots));
}
 $O(t)$ = Result;
```

PositionChaser2D

```
PositionChaser2D: X3DChaserNode {
  SFVec2f      [in]      set_destination
  SFVec2f      [out]     value_changed

  SFVec2f      []        initialDestination
  SFVec2f      []        initialValue
  SFVec2f      [in]      set_value

  SFBool       [out]     isActive

  SFTime       []        duration      [0..+8]

  SFNode       [in,out]  metadata      NULL [X3DMetadataObject]
```

The `PositionChaser` creates transitions for 2D Vectors.

ScalarChaser

```
ScalarChaser: X3DChaserNode {
  SFFloat      [in]      set_destination
  SFFloat      [out]     value_changed

  SFFloat      []        initialDestination
  SFFloat      []        initialValue
  SFFloat      [in]      set_value

  SFBool       [out]     isActive

  SFTime       []        duration      [0..+8]

  SFNode       [in,out]  metadata      NULL [X3DMetadataObject]
') ?>
```

The `ScalarChaser` node creates transitions for scalar values.

PositionDamper

```

PositionDamper: X3DDamperNode {
  SFVec3f      [in]    set_destination
  SFVec3f      [out]   value_changed

  SFVec3f      []      initialDestination
  SFVec3f      []      initialValue
  SFVec3f      [in]    set_value

  SFBool       [out]   isActive

  SFNode       [in,out] metadata      NULL [X3DMetadataObject]

  SFTime       [in,out] tau           [0..+8]
  SFInt32      []      order          [0..5]
  SFFloat      [in,out] tolerance     -1 or [0..8]

```

The `PositionDamper` node creates transitions for 3D Vectors. If it receives a position value on

OrientationDamper

```

OrientationDamper: X3DDamperNode {
  SFRotation   [in]    set_destination
  SFRotation   [out]   value_changed

  SFRotation   []      initialDestination
  SFRotation   []      initialValue
  SFRotation   [in]    set_value

  SFBool       [out]   isActive

  SFNode       [in,out] metadata      NULL [X3DMetadataObject]

  SFTime       [in,out] tau           [0..+8]
  SFInt32      []      order          [0..5]
  SFFloat      [in,out] tolerance     -1 or [0..8]

```

The `OrientationDamper` node creates transitions for 3D Vectors.

TBD: describe how to implement it, similar as with the `OrientationChaser`.

ColorDamper

```

ColorDamper: X3DDamperNode {
  SFColor      [in]    set_destination
  SFColor      [out]   value_changed

  SFColor      []      initialDestination
  SFColor      []      initialValue
  SFColor      [in]    set_value

  SFBool       [out]   isActive

  SFNode       [in,out] metadata      NULL [X3DMetadataObject]

  SFTime       [in,out] tau           [0..+8]
  SFInt32      []      order          [0..5]
  SFFloat      [in,out] tolerance     -1 or [0..8]

```

The `ColorDamper` node creates transitions for color values. Calculations shall take place in HSV space.

PositionDamper2D

```

PositionDamper2D: X3DDamperNode {

```

```

SFVec2f      [in]    set_destination
SFVec2f      [out]   value_changed

SFVec2f      []      initialDestination
SFVec2f      []      initialValue
SFVec2f      [in]    set_value

SFBool       [out]   isActive

SFNode       [in,out] metadata      NULL [X3DMetadataObject]

SFTime       [in,out]   tau          [0..+8]
SFInt32      []         order        [0..5]
SFFloat      [in,out]   tolerance    -1 or [0..8]

```

The `ColorDamper` node creates transitions for 2D Vectors.

CoordinateDamper

```

CoordinateDamper: X3DDamperNode {
  MFVec3f      [in]    set_destination
  MFVec3f      [out]   value_changed

  MFVec3f      []      initialDestination
  MFVec3f      []      initialValue
  MFVec3f      [in]    set_value

  SFBool       [out]   isActive

  SFNode       [in,out] metadata      NULL [X3DMetadataObject]

  SFTime       [in,out]   tau          [0..+8]
  SFInt32      []         order        [0..5]
  SFFloat      [in,out]   tolerance    -1 or [0..8]

```

The `CoordinateDamper` node creates transitions for arrays of 3D vectors. The values which are sent to the `set_destination` or `set_value` field or which are possible assigned to the `initialDestination` or `initialValue` fields must all have the same number of elements. Otherwise the behaviour is not defined.

The 'data type' of the `CoordinateDamper` is an array of 3D vectors, which means that the *current destination* and *current value* are also arrays of 3D vectors. Each element in the *current value* array is animated independently until it reaches the value of the element at the same place in the *current destination* array. The `value_changed` field stops creating events when all elements have reached their destination and the `isActive` field sends *FALSE* at that moment.

TexCoordDamper

```

TexCoordDamper: X3DDamperNode {
  MFVec2f      [in]    set_destination
  MFVec2f      [out]   value_changed

  MFVec2f      []      initialDestination
  MFVec2f      []      initialValue
  MFVec2f      [in]    set_value

  SFBool       [out]   isActive

  SFNode       [in,out] metadata      NULL [X3DMetadataObject]

  SFTime       [in,out]   tau          [0..+8]
  SFInt32      []         order        [0..5]
  SFFloat      [in,out]   tolerance    -1 or [0..8]

```

The `TexCoordDamper` node creates transitions for arrays of 2D vectors. The values which are sent to the `set_destination` or `set_value` field or which are possible assigned to the `initialDestination` or `initialValue` fields must all have the same number of elements. Otherwise the behaviour is not defined.

The 'data type of the `TexCoordDamper`' is an array of 2D vectors, which means that the *current destination* and *current value* are also arrays of 2D vectors. Each element in the *current value* array is animated independently until it reaches the value of the element at the same place in the *current destination* array. The `value_changed` field stops creating events when all elements have reached their destination and the `isActive` field sends *FALSE* at that moment.

[To the paper HomePage.](#)

—.-.—