Accordion Panel V3

ABOUT

The Accordion Panel V3 is a professional accordion panel component, based on VAccordion Panel Pro V2 (vertical) and HAccordion Panel Pro V2 (horizontal).

The Accordion Panel V3 supports both, vertical and horizontal orientation, using a simple XML input file (or variable). It also has customizable tweening, fixed and dynamic content sizes, scale and cropping for panel contents, easy access to panel contents and of course built-in preloaders.

Compatibility

ActionScript: **2.0, 3.0**
Flash IDE: **Flash 8, Flash CS3**
FlashPlayer: **FlashPlayer 8, FlashPlayer 9**

What's new in Accordion Panel V3?

- built from scratches based on the Jumpeye V3 Standards
- skin customization with styles
- full visual customization (both clips and fonts)
- full tween transition customization
- true event handling implemented
- component can be configured using XML
- component can load any type of visual content, both external and internal
- easy content references
- easy XML user- defined attribute access
- fixed and dynamic size panels
- scale and crop modes for panel contents
- built-in preloaders

The story of the Accordion Panel V3

For more than 2 years, Jumpeye has sold Accordions to hundreds of clients, accordions that were used on various websites from any basic flash website to the Microsoft Windows kind of website (windows.com in 2006), where a similar accordion was used to load the whole presentation for Windows software. From all Jumpeye UI components, the Accordion Panel stays out to be the Jumpeye flash component most used for flash designed and developed for the wealthiest companies around the globe; some of them are listed in Fortune 500.

With clients all over the world, Jumpeye Accordion Panel V3 was a must-do, a flash component that speeds up the work in flash, a component that looks perfect, can be easily skinned, performs customized tweenings, Jumpeye Accordion Panel is simply a best-seller that needed to get to the next level.

STYLES

Jumpeye's V3 Components allows skinning with visual elements, Skinning is done via styles (styles are declared and used in the XML of a component, but they may link to movieclip and fonts that need to be found inside a flash file library).

The styles need to be declared inside the **<styles>** tag, and should be used for items in the **<data>** tag of the XML file that feeds the component.

STYLE DECLARATION

All styles should be declared inside the XML that feeds the component under the node **<styles>,** in separate nodes using the name that will be used in the data node when using styles. E.g. ***<myStyleName>*** or ***<style1>***

A style node may contain some nodes that will be used for configuring all states, while it also have a **<states>** node, containing all/or some of the states allowed for the skin.

The ***<style>*** node may contain the following child nodes

<folderCollapsedIconId>
The value of the node should be the linkage id of the movieclip used for collapsed icon

<folderExpandedIconId>
The value of the node should be the linkage id of the movieclip used for expanded icon

<headerSize>
The value of the node should be a number that will be used for sizing the height of a header/title bar of an item

<contentSize>
The value of the node should be a number that will be used for sizing the height of a content of an item

<iconPaddingLeft>
The value of the node should be a number that will be used for left-padding the folder icon and the title text inside an header/title bar of an item.

<spacing>
The value will be used for spacing between icon and text, and where spacing is needed

<textFormat>
The node has exactly the same structure of a TextFormat Class but in XML format, all its childNodes will be added to the header/title item TextField as TextFormat properties.
E.g.

```
<textFormat>
    <font>Tahoma</font>
    <size>11</size>
    <color>0xFF0000</color>
    <bold>false</bold>
    <italic>false</italic>
    <underline>false</underline>
```

</textFormat>

<states>
The node may will contain a child node for each possible state, however, you can omit states, you can simply define only the mandatory **<up>** state and it will be used for all others in base of the state substitution order.

The possible states are the following:

<up>
<over>
<down>
<selected>
<disabled>
<expanded_up>
<expanded_over>
<expanded_down>
<expanded_selected>

Every state may contain the following child nodes:

<textFormat>
This OPTIONAL node has exactly the same structure of a TextFormat Class but in XML format, all its childNodes will be added to the header/title item TextField as TextFormat properties. It overrides the general textFormat for a speciffic state.

<mcLinkageId>
The linkage id of the movieclip that will be loaded as a specific state of the header/title item. This movieclip have to be found inside the flash library.

State substitution order
The **up** state has no substitute, all other states may have proximal substitute, or be substituted by the **up** state, as following:

- over = up
- down = over = up
- selected = up
- disabled = up
- expanded_up = up
- expanded_over = over = up
- expanded_down = expanded_over = down = over = up
- expanded_selected =selected = up

STYLE USAGE

The styles can be used within the **<data>** child node as follows. Each item may have its own **style**, or it can use/inherit the main **style**.

A theme has the visual items and styles defined; themes are packages containing the following:

1. .fla file containing the visual assets of the component
2. .xml file containing the styles and applied to a standard data
3. .as files (rarely, where classes are used to perform a movieclip special effect for component's states)

Inside the **<data>** node, each item node may contain the following style attributes:
style - the style of one particular item or the main style. E.g. style="style1"

A style can be applied to as many items you want. When an item has no style set, it will look for a style of the main data node:

ACCORDION PANEL DATA

In order to build the desired accordion panel, the data structure needs to be defined in the XML file, using the **<data>** tag.

The **<data>** tag will contain the **panels** as items (using the **<item>** tag) following this structure:

```
<data>
        <item />
        <item />
        <item />
</data>
```

The **<data>** tag may have the **style** attribute.

The **<item>** tag may have the following attributes:

GENERAL ATTRIBUTES:
title - the text that will be shown in the menu item
style - the style of the item (optional)
enabled - if the item is enabled(true) or disabled(false) (optional)
expanded - if the item is expanded or not (default:false)
contentPath - either an external file (jpg,swf) or an internal clip
contentSize - the size of the loaded content (if specified in item (primary) or in style (secondary) , will be fixed)
contentEvents – enables or disables the execution of events from the loaded content
headerSize - the size of the header/title bar of every item

URL METHOD ATTRIBUTES
url - a url address to be followed when user clicks on the menu (optional)
target - the url target (optional)

FUNCTION ATTRIBUTES
function - the function with brackets and params, its path by dot notation like in eg.: myFunc(param1,param2)

USER ATTRIBUTES
any other attributes can be inserted here. You will have access to them from the item itself with dot notation

PROPERTIES

Property	Type Component Inspector	Description
xmlPath	String Yes	Optional parameter. Sets the path to a XML file that can configure the Accordion Panel V3. Parameters that are set from the <properties> node of the XML (xmlPath) will override parameters set from script or Component Inspector. The <data> node will contain the panel data, entering this node is essential for the panels to display. You don't need to use xmlPath param, you can set the XML using the setXML method. The <styles> node will contain the styles for panel items, their content sizing, movieclip states and how their text is rendered. Usage: componentInstance.xmlPath = "XMLPanelConfig.xml"
orientation	String Yes	Sets the orientation of the accordion panel. Values:[horizontal,vertical] Default:vertical Usage: componentInstance.orientation = "horizontal"
closeFolderOnClick	Boolean Yes	If true, when a item is expanded and it gets clicked, the item will collapse, if false, the item will not collapse back until a different item is clicked and expands. Default: false Usage: componentInstance.closeFolderOnClick = true
embedFonts	Boolean Yes	If true, embed fonts will be used instead of device fonts. Default: false Usage: componentInstance.embedFonts = true
expandMode	String Yes	The expand mode of the menu. To expand one item at a time, choose "one", to expand all (more than one) items at a time choose "all". Vaues:[one,all] Default: one Usage: componentInstance.expandMode = "all"
navigationMode	String Yes	The mode of navigation with the mouse. Values:[press, release, rollover, none] Default: press Usage: componentInstance.navigationMode = "rollover"
hasSelectedState	Boolean Yes	This variable is used to show the selected item (last clicked), Default:true Usage: componentInstance.hasSelectedState = "false"

tweening	Boolean Yes	If false, the switching between the items will be done without sliding. Default: true Usage: componentInstance.tweening = false
tweeningDuration	Number Yes	Sets the duration of the slide tweening in frames. Default: 20 Usage: componentInstance.tweeningDuration = 10
tweenType	String Yes	Sets the tweening type. Values:[Regular, Strong, Bounce, Back, Elastic, None] Default: Strong Usage: componentInstance.tweenType = "Back"
easeType	String Yes	Sets the ease type for the sliding transition tween. Values:[easeOut, easeIn, easeInOut, none] Default: easeOut Usage: componentInstance.easeType ="easeOut"
loadExpandedItemsOnly	Boolean Yes	If true, the component loads only visible items in the accordion panel component, when an item appears to be visible it loads. This is very useful when loading many items inside a single component, while not all items are expanded(visible). Default: false Usage: componentInstance.loadExpandedItemsOnly = true
itemScaleMode	String Yes	The scaleMode property of a content Loader The mode of the scaling applied to the content in rapport with the sizes of the loader. Values:[scale, resize, crop, scaleCrop, none] Default: resize componentInstance.itemScaleMode = "scale"
itemHAlign	String Yes	The halign property of a content Loader The type of horizontal alignment. Values:[left, center, right] Default: center Usage: componentInstance.itemHAlign="center"
itemVAlign	String Yes	The valign property of a content Loader The type of vertical alignment. Values:[top, middle, bottom] Default: middle Usage: componentInstance.itemVAlign ="middle"
itemBuitInPreloader	Boolean Yes	The builtInPreloader property of a content Loader Sets the type of the bulit-in preloader/progress display. Values:[bar, line, circle, none] Default: none Usage: componentInstance.itemBuiltInPreloader="bar"
itemPreloaderColor	Number Yes	The preloaderColor property of a content Loader Sets the color of the built-in preloader and percentage. Default: 0xFFFFFF Usage: componentInstance.itemPreloaderColor=0xFF0000
itemShowPercentage	Boolean Yes	The showPercentage property of a content Loader Sets the visible property of the bulit-in percentage display. Default: false Usage: componentInstance.itemShowPercentage=true
width	Number No	Used to resize the component's width (header's length). Used when component's orientation is vertical.

		Usage: componentInstance.width=200
height	Number No	Used to resize the component's height (header's length). Used when component's orientation is horizontal. Usage: componentInstance.height=200

Note. Additional properties on child items:

enabled - sets the enable state of an item in runtime;

content - a reference to the child item's loaded content;

AS2 EVENTS

Event	Description

onLoadXML	Event is triggered after the XML is loaded, this happens before loading the content.
	Usage:
	myEventListener = new Object();
	myEventListener.onLoadXML = function(args){
	trace("onLoadXML "+ args.success);
	}
	componentInstance.addEventListener("onLoadXML",myEventListener);

onLoad	Event is triggered after the content is loaded.
	Usage:
	myEventListener = new Object();
	myEventListener.onLoad = function(args){
	trace("onLoad "+ args.success);
	}
	componentInstance.addEventListener("onLoad",myEventListener);

onDrawComplete	Event is triggered after the content has been visually drawn.
	Usage:
	myEventListener = new Object();
	myEventListener.onDrawComplete = function(args){
	}
	componentInstance.addEventListener("onDrawComplete",myEventListener);

onProgress	Event triggered while the contents are loading.
	Usage:
	myEventListener = new Object();
	myEventListener.onProgress = function(args){
	trace("onProgress "+args.target + args.loadedItems + args.totalItems);
	}
	myLoaderInstance.addEventListener("onProgress",myEventListener);

onRelease	Event is triggered when "onRelease" event is invoked on one item.
	Usage:
	myEventListener = new Object();
	myEventListener.onRelease= function(args){
	trace("onRelease event on item: " + args.item);
	}
	componentInstance.addEventListener("onRelease",myEventListener);

onPress	Event is triggered when "onPress" event is invoked on one item.
	Usage:
	myEventListener = new Object();
	myEventListener.onPress= function(args){
	trace("onPress event on item: " + args.item);
	}

	componentInstance.addEventListener("onPress",myEventListener);

onRollOver	Event is triggered when "onRollOver" event is invoked on one item.
	Usage:
	myEventListener = new Object();
	myEventListener.onRollOver = function(args){
	trace("onRollOver event on item: " + args.item);
	}
	componentInstance.addEventListener("onRollOver",myEventListener);

onRollOut	Event is triggered when "onRollOut" event is invoked on one item.
	Usage:
	myEventListener = new Object();
	myEventListener.onRollOut = function(args){
	trace("onRollOut event on item: " + args.item);
	}
	componentInstance.addEventListener("onRollOut",myEventListener);

onReleaseOutside	Event is triggered when "onReleaseOutside" event is invoked on one item.
	Usage:
	myEventListener = new Object();
	myEventListener.onReleaseOutside = function(args){
	trace("onReleaseOutside event on item: " + args.item);
	}
	componentInstance.addEventListener("onReleaseOutside",myEventListener);

onDragOver	Event is triggered when "onDragOver" event is invoked on one item.
	Usage:
	myEventListener = new Object();
	myEventListener.onDragOver= function(args){
	trace("onDragOver event on item: " + args.item);
	}
	componentInstance.addEventListener("onDragOver",myEventListener);

onDragOut	Event is triggered when "onDragOut" event is invoked on one item.
	Usage:
	myEventListener = new Object();
	myEventListener.onDragOut = function(args){
	trace("onDragOut event on item: " + args.item);
	}
	componentInstance.addEventListener("onDragOut",myEventListener);

onChange	Event is triggered when "onChange" event is invoked on one item.
	Usage:
	myEventListener = new Object();
	myEventListener.onChange = function(args){
	trace("onChange event on item: " + args.item);
	}
	componentInstance.addEventListener("onChange",myEventListener);

onClick	Event is triggered when the tweening is over.
	Usage:
	myEventListener = new Object();
	myEventListener.onClick = function(args){
	}
	componentInstance.addEventListener("onClick",myEventListener);

onExpand	Event is triggered when a folder item is expanded.
	Usage:
	myEventListener = new Object();
	myEventListener.onExpand = function(args){
	}
	componentInstance.addEventListener("onExpand",myEventListener);

onCollapse	Event is triggered when a folder item is collapsed.
	Usage:
	myEventListener = new Object();
	myEventListener.onCollapse = function(args){

```
}
componentInstance.addEventListener("onCollapse",myEventListener);
```

AS3 EVENTS

Event	Description

XML_LOAD	Event is triggered after the XML is loaded, this happens before loading the content.
	Usage:
	import com.jumpeye.Events.AccordionPanelEvents;
	componentInstance.addEventListener(AccordionPanelEvents.XML_LOAD,xmlLoadHdl);
	function xmlLoadHdl(evt: AccordionPanelEvents):void{
	}

LOAD	Event is triggered after the content is loaded.
	Usage:
	import com.jumpeye.Events. AccordionPanelEvents;
	componentInstance.addEventListener(AccordionPanelEvents. LOAD,contLoadHdl);
	function contLoadHdl (evt: AccordionPanelEvents):void{
	trace("LOAD "+ evt.success);
	}

DRAW	Event is triggered after the content has been visually drawn.
	Usage:
	import com.jumpeye.Events. AccordionPanelEvents;
	componentInstance.addEventListener(AccordionPanelEvents.DRAW,drawHdl);
	function drawHdl (evt: AccordionPanelEvents):void{
	}

PROGRESS	Event triggered while the contents are loading.
	Usage:
	import com.jumpeye.Events. AccordionPanelEvents;
	componentInstance.addEventListener(AccordionPanelEvents. PROGRESS,progressHdl);
	function progressHdl (evt: AccordionPanelEvents):void{
	trace("onProgress "+ evt.target + evt.loadedItems + evt.totalItems);
	}

ITEM_RELEASE	Event is triggered when "onRelease" event is invoked on one item.
	Usage:
	import com.jumpeye.Events. AccordionPanelEvents;
	componentInstance.addEventListener(AccordionPanelEvents.ITEM_RELEASE,releaseHdl);
	function releaseHdl (evt: AccordionPanelEvents):void{
	trace("onRelease event on item: " + evt.item);
	}

ITEM_PRESS	Event is triggered when "onPress" event is invoked on one item.
	Usage:
	import com.jumpeye.Events. AccordionPanelEvents;
	componentInstance.addEventListener(AccordionPanelEvents.ITEM_PRESS,clickHdl);
	function clickHdl (evt: AccordionPanelEvents):void{
	trace("onPress event on item: " + evt.item);
	}

ITEM_ROLL_OVER	Event is triggered when "rollOver" event is invoked on one item.
	Usage:
	import com.jumpeye.Events. AccordionPanelEvents;
	componentInstance.addEventListener(AccordionPanelEvents.ITEM_ROLL_OVER,rollOverHdl);
	function rollOverHdl (evt: AccordionPanelEvents):void{
	trace("onRollOver event on item: " + evt.item);

	}

ITEM_ROLL_OUT	Event is triggered when "rollOut" event is invoked on one item.
	Usage:
	import com.jumpeye.Events. AccordionPanelEvents;
	componentInstance.addEventListener(AccordionPanelEvents.ITEM_ROLL_OUT,rollOutHdl);
	function rollOutHdl (evt: AccordionPanelEvents):void{
	trace("onRollOut event on item: " + evt.item);
	}

CHANGE	Event is triggered when "onChange" event is invoked on one item.
	Usage:
	import com.jumpeye.Events. AccordionPanelEvents;
	componentInstance.addEventListener(AccordionPanelEvents.CHANGE,changeHdl);
	function changeHdl (evt: AccordionPanelEvents):void{
	trace("onChange event on item: " + evt.item);
	}

EXPAND	Event is triggered when a folder item is expanded.
	Usage:
	import com.jumpeye.Events. AccordionPanelEvents;
	componentInstance.addEventListener(AccordionPanelEvents.EXPAND,expandHdl);
	function expandHdl (evt: AccordionPanelEvents):void{
	trace("expand event on item: " + evt.item);
	}

COLLAPSE	Event is triggered when a folder item is collapsed.
	Usage:
	import com.jumpeye.Events. AccordionPanelEvents;
	componentInstance.addEventListener(AccordionPanelEvents.COLLAPSE,collapseHdl);
	function collapseHdl (evt: AccordionPanelEvents):void{
	trace("collapse event on item: " + evt.item);
	}

METHODS

Method	Description
expandAll	Expands all items of an accordion menu. Usage: componentInstance.expandAll()
collapseAll	Collapses all items of an accordion menu. Usage: componentInstance.collapseAll()
expand	Expands a folder item identified by index Usage: componentInstance.expand(1)
collapse	Collapses a folder item identified by index Usage: componentInstance.collapse(3)
click	Generates a click over an item identified by index Usage: componentInstance.click(2)
get	Returns a reference to an item identified by index Usage: componentInstance.get(2):MovieClip
expandByProperty	Expands an item identified by a property and its value.

	Usage: componentInstance.expandByProperty("myAttribute", "myValue")
collapseByProperty	Collapses an item identified by a property and its value. Usage: componentInstance.collapseByProperty("myAttribute", "myValue")
clickByProperty	Click an item identified by a property and its value. Usage: componentInstance.clickByProperty("myAttribute", "myValue")
getByProperty	Returns a reference to an item identified by a property an its value. Usage: componentInstance.getByProperty("myAttribute", "myValue"):MovieClip
setXML	Sets an XML for the component. You can pass a XML V3 standard to the component using this method. Usage: componentInstance.setXML(myxml:XML)
getXML	Returns the XML that fed the component. Usage: componentInstance.getXML():XML
setSize	Sets the size of the component in real time Usage: componentInstance.setSize(100,200)
item.expand	This method expands an item but it is applied directly on it, instead of applying it on the componentInstance. Usage: item.expand()
item.collapse	This method collapses an item but it is applied directly on it, instead of applying it on the componentInstance. Usage: item.collapse()
item.click	This method generates a click on an item but it is applied directly on it, instead of applying it on the componentInstance. Usage: item.click()
item.load	This method loads content in the child item. Usage: item.load(contentPath)
item.setSize	Sets the size of a single item. If the AP's orientation=="vertical" the dimension will represent "height" and if AP's orientation=="horizontal" the dimension will represent "width" Usage: item.setSize(dimension)