Lassie Dialogue

Lassie Shepherd now includes several advanced dialogue features to allow formatted subtitles, dynamic subtitles, and Lassie Player dialogue managed through a timeline animation. These capabilities build upon the Lassie Player's basic dialogue system to offer additional dynamic content control. Major advanced dialogue features include:

- HTML formatting for dialogue subtitles.
- Dynamic parsing of all dialogue texts, allowing custom content to be inserted into game dialogue at runtime.
- A cutscene framework for MovieClip timelines. This framework allows a timeline animation to send formatted dialogue into the Lassie Player at specific points as the timeline plays. The provided dialogue is then played using the Lassie Player's native dialogue system (which automatically handles language selection and voice-sound playback).

Dialogue HTML

The Lassie Player dialogue subtitle text supports basic HTML rendering. The HTML capabilities are, however, limited to what is naively supported by Flash. For complete documentation on Flash's HTML capabilities, seek Adobe's HTML Flash text documentation. In brief overview, the following HTML tags are supported by Flash:

- **<b>** : Bold tag. "Bold text <b>here</b>".
- **<i>** : Italic tag. "Italic text <i>here</i>".
- **<u>** : Underline tag. "Underlined text <i>here</i>".
- **<a>** : Anchor tag. "Go to my <a href="http://www.yoursite.com">website</a>".
- **<font>** : Font tag for setting basic font attributes. "Here is <font size="50">BIG</font> text!".

Note that HTML font styling (including bold and italics) may be dependent on the availability of an embedded font. This means that you'll probably specifically need to embed bold and italic versions of custom fonts into your UI SWF movie in order for bold and italic texts to render. In the event that Flash attempts to render styled text when the corresponding font is unavailable, the text will not display. There are many online resources that focus on Flash font management and HTML text support.

Dialogue Text Parsing

To support advanced content delivery, Lassie Player parses all subtitle text using the Logic API before displaying it. That means that you may parse custom values set within the game cache into a subtitle at runtime. This runtime parsing will allow static game texts to be customized with dynamic values. Let's build a basic example.

First, we'll create a "score" variable within the game cache.

```
<cache field="score" value="0"/>
```

Next, let's write a subtitle text that parses that score variable into it. You could place this text within any dialogue field within the Shepherd Editor:

```
"I have a score of %*[score]*%!"
```

In the above dialogue text, we've defined a static string that get a game value parsed into it. In the above example, our logic field is "%*[score]*%". When the Logic API parses text, it will parse any text that is enclosed within a "%*" section, like so: "%* --logic text here-- *%". When the above subtitle plays, it will produce the following result:

```
"I have a score of 0!"
```

Now, let's say that the user performs an action which increments their score. We can increment the score variable like so:

```
<cache field="score" math="+25"/>
```

In the above script, we've increased the player's score by twenty-five, Now if they access the dynamic subtitle field, the subtitle text will get the new score value parsed into it, like so:

```
"I have a score of 25!"
```

The Lassie Cutscene Framework

The Lassie cutscene framework integrates timeline animation with the Lassie Player's native dialogue system. This allows standard timeline animation sequences to utilize the Lassie Player's dialogue and language management capabilities to deliver voice sounds and dialogue subtitles inline with a timeline sequence. Also, the Lassie cutscene framework is easy to implement. It builds upon the process of creating Lassie media libraries; along with some single-line ActionScript calls placed along a timeline. If you've followed along with the Lassie media library tutorial and are successfully building Lassie media libraries, then you've got all the skills needed to use the Lassie cutscene framework.

The Lassie Player comes with a library of external script resources for a developer to build upon. These external resources allow external Flash media to integrate with the core Lassie Player. One key component is the Lassie Cutscene Engine, which allows a developer to build a timeline-based animations that send dialogue into the core Lassie Player for dynamic playback. This system provides the best of both worlds: Flash animation and managed content delivery.

Overview

The behavior of the Lassie cutscene framework is pretty basic. The system starts by loading an external XML document that defines all dialogue text that will be used within the cutscene. Once the XML data loads, the timeline animation will automatically start playing. Then, along the timeline you can insert timeline actions which call for specific dialogue items to play. When a dialogue item is called, the timeline is automatically stopped and holds on the calling frame of the timeline. The timeline remains stopped until notification is received from the Lassie Player that the dialogue has finished playing, at which time the timeline's playback will automatically resume. And that's it!

Now, the obvious question is – why is the timeline stopped while dialogue is being played? The rationale revolves around speaking animations. Chances are that you're going to have a sprite with an animated mouth on the screen while dialogue is playing. However, you have no way to predict the duration of the dialogue sequence, so you can't define a length for the mouth animation. Instead, the cutscene's root timeline is stopped while dialogue is playing so that you can place a looping mouth animation on the single dialogue delivery frame. The mouth animation will then loop for as long as it takes the dialogue to play.

Setting up XML

To set up a Lassie cutscene, you'll first need to create an XML data structure for it. There is no WYSIWYG within the Shepherd Editor for defining cutscene XML files, so you'll need to manage the XML by hand. Fear not –– XML is easy. It's the same as HTML except that tags have custom labels. To set up a new XML document, open any text editor, create a new plain text document, and save it as "myfile.xml" (feel free to customize the file name), then place this new XML document into your Shepherd project's "xml/" folder. Next, paste in the following XML into your blank document as a template to work from:

```
<?xml version="1.0" encoding="UTF-8"?>
<cutscene>
    <cue id="single">
        <dia puppet="" frame="" sound="" color="#FF0000"
plot="300,400">
            <en><![CDATA[English subtitle here.]]></en>
        </dia>
    </cue>
    <cue id="list">
        <dia puppet="" frame="" sound="" color="#FF0000"
plot="300,400">
            <en><![CDATA[English subtitle here.]]></en>
        </dia>

        <dia puppet="" frame="" sound="" color="#FF0000"
plot="300,400">
            <en><![CDATA[English subtitle here.]]></en>
        </dia>
    </cue>
</cutscene>
```

Looking at the above XML template, you'll first see the XML header tag and then a <cutscene> tag enclosure. This <cutscene> node is the root namespace of the document. All of your custom dialogue configuration data should be placed between the opening and closing <cutscene> tags.

Within the root <cutscene> node is a list of one or more <cue> nodes. A <cue> represents a single batch of dialogue that gets sent to the Lassie Player. Cues are called by ID, therefore all <cue> nodes MUST be given a unique "id" attribute. A cutscene may contain as many or as few <cue> nodes as necessary. To add additional <cue> nodes, just copy and paste a <cue> structure into the document at the bottom of the <cue> node list. The order of <cue> nodes within the list does not matter as long as all cues are given a unique "id" value. Note: <cue> ID values should be limited to upper case and lower case letters, numbers, and underscores.

Finally, within each <cue> node is a list of one or more <dia> nodes. A <dia> node structure defines text and parameters for a single line of dialogue to be played by the Lassie Player. When multiple <dia> nodes are listed within a single <cue> node, those lines will be played in sequence by the Lassie Player. To add additional <dia> nodes, just copy and paste a <dia> structure into a dialogue list. A dialogue node uses the following structure:

```
<dia puppet="" frame="" sound="" color="#FF0000" plot="300,400">
   <en><![CDATA[English subtitle here.]]></en>
   <de><![CDATA[Deutscher Untertitel hier.]]></de>
</dia>
```

In the above XML snippit, parameters are as follows:

- **puppet**: This attribute references a puppet layer Id within the main room layout to use as the dialogue target. Citing a puppet reference assumes that the puppet exists within the current game room in addition to the cutscene MovieClip. Citing a puppet reference works the same as normal Lassie dialogue playback: the puppet's rectangular bounds will be used to place the subtitle, and the puppet's speech color will be applied to the subtitle display. If you do not wish to target a specific room puppet with the dialogue line, just leave this value blank ("").
- **frame**: Use this attribute in conjunction with the "puppet" attribute. This specifies what speaking frame the puppet will animate with. If this value is blank but a puppet Id is specified, the target puppet will be animated with the standard "speak" frame.
- **sound**: Specifies a sound file to place in conjunction with this dialogue line, specified as "lib/filename.swf:SoundClass".
- **color**: This parameter is optional. Specifies a hexadecimal value (hex color) to set as the subtitle text color. The hex value must start with a hash character (#), such as "#FF0000". If a puppet target is specified for the dialogue line, then the puppet's assigned subtitle color will override this value.
- **plot**: This parameter is optional. Specifies X and Y coordinates to plot the subtitle at, formatted as "x,y"; or "200,100". If a puppet target is specified for the dialogue line, then the puppet's screen position will override this value. Also, a custom position may be provided while calling a cue's playback. If a cue is called with a custom position, those coordinates will also override this value.
- **language texts**: The <dia> node contains language-specific texts for the subtitle display. All texts MUST be wrapped in a language key node, even if you're only developing for a single language. By default, the Lassie Player is keyed to <en> (English), however your game's default language key may be customized on the Settings panel of the Shepherd Editor. When writing out subtitle texts, it is suggested that you wrap all text within a CDATA block. CDATA is an unparsed XML block, allowing it to include characters which are otherwise illegal within XML markup. So, CDATA is the easiest way to guarantee that your XML structure remains valid, regardless of the text content that you write. A CDATA block is written as: <![CDATA[--content here--]]>.

Defining "LPCutscene" Class Linkage

To utilize the Lassie cutscene framework, a cutscene MovieClip must extend the "LPCutscene" class, which is a provided resource within Shepherd's script library. Assigning the "LPCutscene" class extension is done in almost the exact same manner as setting class definitions for a standard Lassie media library (for information on that process, see the "Lassie Media Library" tutorial). Let's assume that you've read the media library tutorial and are familiar with this process, in which case, this process is easy:

Go into the Flash library panel, right-click on you main cutscene MovieClip symbol, and select "Linkage..." from the contextual menu that comes up. The symbol's linkage properties will display in a new window. A symbol has two main linkage properties: "class" and "base class". Normally while building Lassie media libraries, you just give a symbol a class name and leave its base class untouched. Well here's the exception. Give you main cutscene MovieClip a base class of "com.lassie.external.LPCutscene". This will enable your cutscene MovieClip with all functionality of the Lassie cutscene framework. Once you've given your cutscene clip the "LPCutscene" base class, all other media management goes strictly by the standard Lassie media library workflow. You'll still need to define a unique class name to the symbol, and then include that main class name in the list of class exports for the library.

Also note that the "com.lassie.external.LPCutscene" base class definition assumes that the FLA that you're working in is located in the same directory as the Lassie "com" folder (Lassie's script library). If the cutscene's FLA does not have access to Lassie's script library ("com"), then the script linkage will fail.

Adding Timeline Actions

Once you've set up an XML file for your cutscene and assigned the cutscene's MovieClip the "LPCutscene" class linkage, then you're almost done! All that's left is to load the XML and scatter in cue calls along your cutscene's timeline. To get started, create a new layer within your cutscene timeline and label it "actions". Arrange this "actions" layer as the top-most layer of the timeline, and make sure to always keep it at the top (because Flash renders a timeline from bottom-to-top by default, keeping the "actions" layer at the top of the timeline makes sure that actions are always called after all timeline media has rendered).

Next, select the blank keyframe on the first frame of the "actions" layer, then open your actions window and add the following line of ActionScript:

```
load("xml/your_xml_file.xml");
```

The above line of ActionScript will stop the cutscene's timeline an load the specified cutscene XML file. The cutscene's timeline will wait on the loading frame until the XML has been loaded, at which time the cutscene timeline will start playing automatically. Note that you may load your cutscene XML file from any location relative to the game player, however it is suggested that you keep your XML organized within your project's main "xml" folder.

Now all that's left is to add cue calls along your timeline. Whenever you reach a point along the cutscene's timeline that you want to stop and play dialogue, just create a blank keyframe on the "actions" layer and add the following line of ActionScript:

```
playCue("cueId");

// OR:

playCue("cueId", {x:100, y:100});

// OR:
```

```
playCue("cueId", movieClipRef);
```

The above line of ActionScript will seek a <cue> node with the specified ID, then send it to the Lassie Player for playback. No action will be taken if a <cue> node with the specified ID is not found. Once the Lassie Player receives the XML cue, the cutscene's timeline will be stopped and wait for all dialogue lines to finish playing within the Lassie Player, then the cutscene's timeline will automatically resume.

You'll notice that the "playCue()" command above call has three permutations involving it's second parameter. That second parameter defines the subtitle's location on screen. There are numerous ways to control a subtitle's screen position. Here are the options:

1. **Do not include a custom location** *(first line in the above code example)*. When no custom position is defined in the cue call, then the subtitle will be placed using the cue's <plot="x,y"> XML attributes (defined within the source XML), or at the XML puppet's stage coordinates (assuming a valid Lassie puppet ID is provided in the XML).
2. **Include a custom point reference** *(second line in the above code example)*. You may define custom stage coordinates by passing an object with "x" and "y" properties as the second argument of the "playCue()" command. This may be a generic Flash [Object object], or a Flash [Point object].
3. **Include a reference to a MovieClip instance on stage** *(third line in the above code example)*. If you'd like to place a subtitle relative to a MovieClip within your cutscene animation, then pass a reference to the target MovieClip as the second argument of the "playCue()" command. To reference a MovieClip on stage, give the clip a stage instance name. When the "playCue()" command is given a MovieClip reference, it will center the subtitle display over the top of the MovieClip's bounding rectangle.

Finally, we just need to handle the cutscene's conclusion. Chances are that you'll want your cutscene to automatically feed into another game room after playing. To do this, just add a blank keyframe onto your "actions" layer at the last frame of the timeline, and add the following line of ActionScript:

```
exitToRoom("roomId:positionId");
```

The above line of ActionScript will stop the cutscene's timeline an transition the game to the specified room target. The room target reference works just like the <game> XML command, where a target room is specified as the target room Id and target start position Id separated by a colon.

Also, one final note about cutscene timeline actions... cutscene timeline actions will only work within a MovieClip that DIRECTLY extends the "LPMovieClip" class. Nested MovieClips placed within a "LPMovieClip" instance will NOT have access to the Lassie Player cutscene features, nor will calling cutscene actions on a parent clip have an impact on the child. Can you nest multiple LPMovieClip instances inside one another? Sure; however you may find that your cutscene playback becomes difficult to manage as you add layers of dynamic integration. Whenever possible, it is suggested that you try to limit a cutscene to one main timeline that controls the main sequential progression of the animation.