Introduction

Shepherd introduces a dramatically new system of programming to Lassie games. Where previous versions of Lassie were programmed with psudo-ECMA style script commands, Shepherd is programmed entirely using XML snippits. For those unfamiliar with XML, fear not. Composing XML is exactly like writing HTML markup, except that XML tags use custom names and attributes. A basic Shepherd script command looks like this:

```
<game room="beach"/>
```

XML also supports nested structures, which is leveraged by some Shepherd commands for layered script controls. A basic nested structure used in a Shepherd script would look like this:

```
<keyboard addKeyScript="i">
    <inventory display="toggle"/>
</keyboard>
```

Shepherd's scripts are composed in the Global Script panel and Room Script panel, as well as in each interaction manager throughout the Shepherd editor. When composing scripts, just line up multiple XML snippits in sequence, such as:

```
<cursor visible="0"/>
<layerClip target="_avatar" frame="reach" waitForComplete="1"/>
<inventory add="keys"/>
<cursor visible="1"/>
```

Additionally, Lassie Shepherd has been expanded with support for conditional logic operations. Conditional logic allows game behavior to be intelligently controlled based on current conditions within the engine. Using Shepherd's "logic" command, you can test to see if specific game conditions currently apply and take action (or not) in response. A basic Shepherd logic construct looks like this:

```
<logic eval="[_currentRoom] EQ 'beach'">
    <!-- add beach-specific behaviors here -->
</logic>
```

Working to Your Skill Level

As you get started with the Lassie XML scripting language, you'll likely find it helpful to skim the command overviews within this document to get a feel for what commands are available for what sorts of behaviors. Lassie Shepherd's XML scripting API is designed to be user friendly, non-intimidating, and generally easy to use.

For beginners, the majority of Lassie XML commands can be comfortably used with little-to-no programming experience. Beginners will want to spend most of their time getting familiar with the core Lassie commands including <game>, <dialogue>, <inventory>, and the <layer> commands. If you do find yourself confused by a command while learning the ropes of the system, just file it away and move along past it... chances are that it goes beyond what you need to know to build a basic game. However, as your proficiency with the system grows, revisit the commands that you didn't understand at first and you may find interesting new possibilities presented!

Once you feel generally comfortable using the system, an intermediate user will be ready to start experimenting with basic game logic through commands like <logic> and <cache>, or creating dynamic animations using the <puppetTween> command.

Finally, advanced users can begin to harness the full potential of the Lassie Player by exploring the powerhouse commands such as <method>, <process>, <logicParse>, or the external ActionScript API. These advanced features will allow you to create a truly unique gaming experience that pushes the Lassie Player beyond its basic functionality, allowing you to create a one-of-a-kind game.

Commenting Scripts

As you get into scripting an application, the best possible favor you can do for yourself as the developer is to heavily comment your script. This holds true for all programming done in any computer language.

What is a comment? It is a note written into your code by you, the developer, for you, the developer. Comments are non-parsed lines of script, which means that they are completely ignored by a computer program. A comment's entire purpose is to provide narrative and documentation as how a script works and why it works the way it does. While this information may seem obvious while you're actively writing a script, try coming back a week, a month, or a year later and try to remember why you scripted something the way you did. When it comes to remembering code, you'll find that very little is actually retained in your long-term memory!

So, write comments for yourself! While they add a small amount of file weight, the extra file size is minimal and is in NO WAY an excuse for not writing yourself frequent, detailed comments that will make your development job easier. Also, Lassie will likely have compile-time comment removal in the future, which means that comments will automatically be omitted from published XML files, so will not add any file weight to final data (this feature is planned but not currently supported).

The other good news is that Lassie's XML comments are formatted exactly like standard HTML comments that you've probably been writing for years. A comment node is formatted as:

```
<!-- Note to self: write lots of comments! -->
```

Comments are noted using opening and closing tokens (sets of characters). A comment is opened using "<!--" and then closed again using "-->". You may place any text with any formatting inside the bounds of the comment; you can even place unused XML command nodes within a comment! Here's an example of a commented script:

```
<!-- disable cursor, then play an animation -->
<cursor gameMouse="0" visible="0"/>

<!-- play reach up animation, add the item to inventory, then reach
back down -->
<layerClip target="_avatar" frame="reachUp"/>
<inventory add="apply"/>
<layerClip target="_avatar" frame="reachDown"/>

<!-- re-enable game cursor after sequence -->
<cursor gameMouse="1" visible="1"/>
```

Finally, keep in mind that you can "comment out" lines of code that you may want to only disable temporarily, rather than entirely removing. Just turn the opening and closing

carrots, like so:

```
<!--cursor visible="0"/-->
```

XML Commands

This section contains a listing of all XML scripts currently built into the Lassie Player API.

Preface: Attribute Order

One technical consideration to keep in mind while developing is that each command does have an order in which its specific attributes are processed. For example, consider the following:

```
<cursor gameMouse="0" visible="0"/>
```

The above command completely disables cursor interactivity within the Lassie Player, and then hides the cursor. However, those two actions were performed in a sequence while the script ran. The order in which individual attributes of a command are processed is NOT reflected by the order in which the attributes are listed in the node (due to technical considerations for some of the more complicated scripts). Also, the ordering of command attributes within this documentation does not reflect processing order either. In short, processing order is not a documented feature, given that it is subject to change in the future to accomodate the integration of other feature.

In the case of the cursor command example above, it really doesn't matter which of those attributes are processed first. The first action is not contingent on the second. However, in the case of less linear processes – like inventory management – the order in which you add and remove items may be extremely important. If there is ever a case where attribute process order is of great importance, then avoid pairing up attrbutes on a single command node. Instead, just break the attributes onto two seperate command nodes, placed in the proper processing order. For example:

```
<!-- Linear process: this could result in unpredictable results -->
<inventory remove="soccerball" add="baseball"/>

<!-- Split process: we know this will process in our intended order
-->
<inventory remove="soccerball"/>
<inventory add="baseball"/>
```

cache

Player version: 1.0

The cache command manages values within the game cache. The game's cache acts as storage for custom variables that you define within your game. You can use cached variables to track activity, status, and progress within your game based on values that you define. A game cache is created for each saved game, so the player's progress can be tracked between game sessions. Key capabilities of the cache command include:

- Ability to write game variables with a unique name an value into the cache.
- Ability to modify existing variables using math and/or modifiers.
- Ability to manage lists of values as a single variable field.

Basic example:

```
<cache field="doorOpened" value="1"/>
```

Required Parameters

field

type: string
values: "fieldName"

Specifies the variable field name targeted by this command. A "field name" is the name of a variable written into the cache. This field name acts as a reference to the specific field value.

Example:

```
<cache field="doorOpened" value="1"/>
```

Optional Parameters

value

type: string
values: "fieldValue"

A value to be written into the specified field. This value has no restrictions concerning its

content or length. However, developer discretion is encouraged. In other words, it is not advised that you use the cache to store massive entries of text or other data. Also, if you plan on performaing mathematical operations on a cache field, make sure the field is always set to a valid number.

To clear a value from the cache, you can set the field's value to empty ("").

Example:

```
<cache field="doorOpened" value="1"/>
```

---

math

type: Expression string
*values: ["+1" / "-1" / "*1" / "/1"]*

Performs a mathematical operation on the specified field. In order for this operation to be successful, the specified field must contain a valid number. The math parameter must then specify a valid mathematical operator followed by a numeric value. Valid operators include:

- + : Addition.
- - : Subtraction.
- * : Multiplication.
- / : Division.

Example:

```
<cache field="score" math="+10"/>
<cache field="score" math="-5"/>
<cache field="score" math="*3"/>
<cache field="score" math="/2"/>
```

---

append

type: string
value: "_nextValue"

Appends a value onto the end of the existing field value. If the current field value is "base" and "_suffix" is appended, the new field value will be "base_suffix".

Example:

```
<cache field="myList" append="_suffix"/>
```

---

add

type: string
value: "value1,value2"

Adds a new value or values into a comma-seperated list. If the current field value is "a" and "b" is added, the new field value will be "a,b". Comma-seperated lists offer some advanced capabilities when evaluated as part of logic operations. Note that you cannot add values that contain commas into a comma-seperated list unless you intend for the value to be broken into seperate list items.

Example:

```
<cache field="tasksComplete" add="doorOpened"/>
<cache field="tasksComplete" add="doorOpened,goldFound"/>
```

---

remove

type: string
value: "value1,value2"

Removes a value or values from a comma seperated list. If a field's current value is "a,b,c" and "b" is removed, then the new field value will be "a,c".

Example:

```
<cache field="equiptment" remove="helmet"/>
<cache field="equiptment" remove="helmet,shield"/>
```

---

sort

type: string
value: "value1,value2"

Adds a new value or values into a comma-seperated list, then sorts the list of values alphabetically. You may declare the sort value as blank ("") to sort an existing list field without adding a value to it.

Example:

```
<cache field="tasksComplete" sort="doorOpened"/>
<cache field="tasksComplete" sort="doorOpened,goldFound"/>
<cache field="tasksComplete" sort=""/>
```

context

Availability: Lassie Player v1.0

The <context> command is used to set the summary of the game's contextual display (which is the text that summarizes objects and actions currently being inspected). In addition, the <context> command has several visual styling properties which can be used to customize the appearance of the context summary.

Basic example:

```
<context verb="use" noun="beans"/>
```

Required Parameters

One or more of the optional parameter must be specified.

Optional Parameters

action

type: string
value: "clear" or "highlight"

Specifies an action to perform on the context display. This value may be set to "clear" or "highlight". Specifying "clear" will clear all configuration and text from the context display. Specifying "highlight" will quickly flash the context display with the highlight color (use this feature to imply when an action has been executed).

Example:

```
<context action="clear"/>
<context action="highlight"/>
```

---

format

type: string
value: "lower", "upper" or "" (none).

Specifies a text formatting to apply to the contextual display. The text may be specified to

display as all lowercase or all uppercase. By default, this value is blank which allows all text to display with its basic case configuration.

Example:

```
<context format="lower"/>
<context format="upper"/>
<context format=""/>
```

---

highlightColor

type: hexadecimal
value: "0xFFFFFF"

Specifies the color to highlight the contextual display with. The contextual text is highlighted whenever an action is performed, which visually implies that the action was "executed". The highlight color is specified as a hexadecimal value with a leading "0x"; so for example, the value for bright red would be specified as "0xFF0000".

Example:

```
<context highlightColor="0x00FF00"/>
```

---

normalColor

type: hexadecimal
value: "0xFFFFFF"

Specifies the normal color of the context display text. The highlight color is specified as a hexadecimal value with a leading "0x"; so for example, the value for bright red would be specified as "0xFF0000".

Example:

```
<context normalColor="0xFFFFFF"/>
```

---

noun

type: string
value: "noun"

Specifies the noun component to display as the contextual summary's target.

Example:

```
<context noun="coconut"/>
<context verb="break" noun="coconut"/>
```

---

verb

type: string
value: "verb"

Specifies the verb component to display as the contextual summary's action.

Example:

```
<context verb="break"/>
<context verb="break" noun="coconut"/>
```

cursor

Availability: Lassie Player v1.0

The <cursor> command controls the visual and interactive status of the game cursor. Features include:

- Enabling or disabling cursor response within the game.
- Showing or hiding the cursor display.
- Setting and clearing an item cursor tooltip.
- Setting the cursor graphics package.

Basic example:

```
<cursor visible="1" gameMouse="0"/>
```

Required Parameters

One or more of the optional parameter must be specified.

Optional Parameters

displaySet

type: string
value: "setName"

Specifies a base prefix to use while setting the cursor frame. By default, this value is a blank string (""). This parameter allows you to create multiple sets of cursor graphics and switch between them during game play. For example, default cursor behavior utilizes frames called "on" and "off" for the cursor's activity display. If you were to create cursor frames labled as "custom_on" and "custom_off", and then set the cursor displaySet to "custom_", these custom cursor graphics would be used during game play.

Example:

```
<cursor displaySet="custom_"/>
```

gameMouse

type: boolean
values: "1" (true) or "0" (false)

Specifies whether game-level mouse response is enabled. This setting will enabled or disable mouse interactivity of all elements on screen. When set to false (0), the player will not be able to interact with any room objects or game interface elements (action selector, inventory, dialogue tree menu). This setting will NOT hide the cursor display.

Example:

```
<cursor gameMouse="0"/>
```

---

roomMouse

type: boolean
values: "1" (true) or "0" (false)

Specifies whether room-level mouse response is enabled. This setting will enabled or disable mouse interactivity with all elements within the current room display. When set to false (0), the player will not be able to interact with any room objects, although game interface elements (inventory, dialogue tree menu) will still be interactive. This setting will NOT hide the cursor display.

Example:

```
<cursor roomMouse="1"/>
```

---

item

type: string
values: "itemId" (sets the item cursor) or "" (clears the item cursor)

Loads an inventory item into the cursor tooltip, or clears an item from the cursor tooltip. When loading an inventory item, the item property must reference a valid inventory item ID. To clear an existing item tooltip, specify an empty value ("").

Example:

```
<cursor item="coconut"/>
<cursor item=""/>
```

itemFrame

availability: Lassie Player v1.1
type: string
value: "frameLabel"

Sets the display frame of the the cursor's item tooltip. Setting this value will only have an effect if there is an item tooltip currently active. The value of "itemFrame" must reference a valid frame label within the item image MovieClip timeline. This setting may be called by itself, or in conjunction with an "item" property that will first set an item cursor prior to setting the display frame.

Example:

```
<cursor itemFrame="broken"/>
<cursor item="coconut" itemFrame="broken"/>
```

visible

type: boolean
values: "1" (true) or "0" (false)

Shows and hides the cursor. When false (0), the cursor display will not be visible. Note that even when the cursor display is hidden, there is still an active tooltip under the player's control. The player will still be able to interact with objects on screen despite the fact that the cursor graphic is hidden. Therefore, it is recommended that you disable the *gameMouse* when you hide the cursor.

Example:

```
<cursor visible="0"/>
```

curtain

Availability: Lassie Player v1.0

The curtain command controls the overlay used to fade the game display in and out.
Features:
Fades the screen blackout overlay in and out.
Fades the screen blackout to a specified opacity.

Basic example:

```
<curtain blackout="1"/>
```

Required Parameters

One of the optional parameter must be specified.

Optional Parameters

blackout

type: boolean
values: "1" (true) or "0" (false)

Specifies the screen overlay to fade entirely in or out. A value of true (1) will fade the screen to a complete blackout. A value of false (0) will fade the blackout to reveal the game display.

Example:

```
<curtain blackout="1"/>
```

---

to

type: number
values: ["0" (min) to "1" (max)]

Specifies an opacity percentage (represented by a decimal between 0 and 1) to fade the screen overlay to. The maximum value of 1 will completely blackout the display. The minimum value of 0 will completely reveal the game display. A mid-range value would fade

the screen to a partial blackout; for example: a value of 0.5 would fade the screen overlay to 50% opacity.

Note: the to parameter cannot be used in conjunction with the blackout parameter. If both parameters are defined within a single command node, the to parameter will be ignored.

Example:

```
<curtain to="0.5"/>
```

---

waitForComplete

type: boolean
values: "1" (true) or "0" (false)

Specifies if additional command processing should be suspended until the curtain's fade tween has completed. If TRUE, no further commands within the process queue will be called until the tween has completed. Otherwise, additional commands within the queue will continue processing while the curtain fades. If this parameter is not specified, then the default value is TRUE.

Example:

```
<curtain blackout="1"/>
```

data

Availability: Lassie Player v1.2

The <data> command deals with optimization and memory management during a game session. A game "session" refers to a saved game's playback. When a player starts a new game or loads a saved game, a new session is started. When a player saves a game, their session data is saved.

Now, it's important to understand that session data grows over time. While the Lassie engine is optimized to save as little data as possible about all world states, still – as the player explores deeper into a game world, they will accrue more and more game-state data that must be saved as part of their session data. While there is no defined limit on the maximum amount of data that Flash can save to a file, you may encounter some practical limitations on some systems. That said, it's always best to minimize the amount of data that you're saving as part of a session. However, don't feel that you need to be stingy. While you MIGHT encounter a practical data cap at some point, it will not be discovered quickly.

Basic example:

```
<data deleteRooms="beach,docks"/>
```

Optional Parameters

deleteRooms

type: list
value: "room1,room2"

Specifies a list of room IDs to unload and purge from the game session.

Examples:

```
<data deleteRooms="beach,docks"/>
```

---

deleteVars

type: list
value: "var1,var2"

Specifies a list of cache variables to purge from the game session data.

Examples:

```
<data deleteVars="num_gold,quest_status"/>
```

---

deleteVarPrefix

type: string
value: "prefix"

Specifies a prefix name to purge from cache variables in game session data. The <session> command will then look through all variables saved into the current game session and purge all fields starting with the specified prefix. For example, specifying a prefix of "act1_" would purge variables called "act1_status", "act1_score", and "act1_complete".

Examples:

```
<data deleteVarPrefix="act1_"/>
```

---

unloadRooms

type: list
value: "room1,room2"

Specifies a list of room Ids to unload from Lassie's data model. Once a room has been unloaded, the room's source XML will need to be reloaded the next time the player accesses the room. By default, the Lassie Player retains room data in memory because it has low overhead and allows a room to reload quickly

This command is very similar to "purgeRooms", except that "unloadRooms" will NOT purge room-state information from game session data. The method is useful for freeing up memory when you won't be revisiting a room anytime soon, however you want to retain data about the state in which you left the room.

Examples:

```
<data unloadRooms="beach,docks"/>
```

debug

Availability: Lassie Player v1.0

The debug command interfaces with the Lassie Player's debug console. The debug console shows player performance statistics and can output custom messages from within your XML script. This is extremely useful for debugging a script, given that you can insert debugging messages into a script and watch the order in which they play out. Note that in order to see debugging messages, you must specifically enable the debug console. Debug commands will be ignored while the debug console is inactive. It is suggested that you include only one "debug-enable" command in the start up script of your game; that way you can always make sure that the debug window is disabled when you release your game.

Basic example:

```
<debug echo="Hello World"/>
```

Required Parameters

One of the optional parameter must be specified.

Optional Parameters

enabled

type: boolean
values: "1" (true) or "0" (false)

Enables / diables the Lassie Player's debugger window. By default, the debugger is disabled. You must specifically enable the debugger before any debug commands will have an effect.

Example:

```
<debug enabled="1"/>
<debug enabled="0"/>
```

---

display

type: boolean
values: "show", "hide" or "toggle"

This parameter shows and hides the debugger window. Note that the debug window MUST be specifically enabled before it can be displayed.

Example:

```
<debug enabled="1" display="show"/>
```

---

stats

type: boolean
values: "1" (true) or "0" (false)

Specifies if the debug window should include performance statistics. The statistics display is enabled by default, so set "stats" to false to reduce the debugger window to just a message area.

Example:

```
<debug enabled="1" stats="1"/>
```

---

echo

type: string
value: "message"

Opens the debugging window and writes a message into it. This is useful for tracking the flow of your scripts by outputting messages at specific points in the command playback. Echoed messages will only display if the debugger has been enabled.

Example:

```
<debug enabled="1" echo="Hello World"/>
<debug echo="Hello World"/>
```

dialogue

Availability: Lassie Player v1.0

In most cases within the Shepherd editor, XML scripts are written in conjunction with a list of dialogue lines. This combination of dialogue and script is referred to as an "action". If the dialogue command is called from a script without an associated dialogue queue (ex: called from a global script), then it will have no result.

The <dialogue> command controls how much dialogue is released from an action's dialogue list, and at what point within the script the dialogue plays. By including multiple <dialogue> commands within a script, you can release dialogue lines in small batches that flow around other actions and animations that are triggered throughout the action's script.

The <dialogue> command is also used to disable default dialogue that is queued by the Lassie Player. By default, all action-based triggers (dialogue and script pairings) will be populated with a line of default dialogue if no custom dialogue is defined. In the event that you do not want default dialogue to play with an action, you'll specifically need to include a <dialogue> command which tells the engine to play "none".

As a broad overview, the dialogue command can manage dialogue in the following ways:

- Play **all** lines currently in the action's dialogue list.
- Play **no dialogue** – from the action's dialogue list or from default dialogue.
- Play a **single random** line of dialogue from the action's dialogue list.
- Play a **list** of specific lines from the action's dialogue list.
- Release a **batch** of the first X lines of dialogue from the action's remaining dialogue list.

When the dialogue command runs, it first builds itself a local queue of dialogue to play using dialogue from the parent action. Depending on the action the dialogue command takes, it may "pull" dialogue from its parent action, or "copy" the dialogue. When dialogue is pulled, it is removed from the parent action's queue and will not be available in subsequent <dialogue> commands called from the script. When dialogue is copied, it remains available within the parent command's dialogue list. Take special note of whether individual behaviors of the dialogue command pull or copy dialogue.

Basic examples:

```
<dialogue play="*"/>
<dialogue play="none"/>
<dialogue play="random"/>
<dialogue playList="1,3,5"/>
<dialogue playBatch="2"/>
```

Required Parameters

One of the optional parameter must be specified.

Optional Parameters

play

type: string
values: "" (all), "none" (no dialogue), "random" (single random dialogue line)*

The play attribute controls basic dialogue release methods, called using one of three key words:

- **"*"** (all) : Pulls and plays all lines from the action's dialogue list. Because all dialogue is pulled from the parent action, subsequent <dialogue> commands during the script will be ignored given that there is no dialogue left to play.
- **"none"** : Aborts the dialogue command without playing any dialogue, including default dialogue. This command is used to specifically instruct the engine NOT to play any dialogue during the script sequence. By default, the Lassie Player will populate all actions with default dialogue if they have no custom dialogue defined.
- **"random"** : Pulls and plays a single random line of dialogue from the action's dialogue list. Calling sequential random dialogues will never allow the same line to play twice, given that each line of dialogue becomes unavailable after being pulled from the parent action's dialogue list.

Examples:

```
<dialogue play="*"/>
<dialogue play="none"/>
<dialogue play="random"/>
```

---

playList

type: string
values: "0,1,3" (list of indices)

Specifies a comma-seperated list of dialogue lines to copy and play, referenced by their numeric index within the action's dialogue list. Dialogue lines are only copied from the parent action's dialogue list, so will remain available for replay in all subsequent <dialogue> calls. This allows all lines to retain a constant numeric index within the unchanging dialogue list. Also note that dialogue line indices are zero-indexed, meaning that the first line is "0", the second line is "1", etc.

Examples:

```
<!-- Plays the first, third, and forth lines -->
<dialogue playList="0,2,3"/>

<!-- Plays the second line -->
<dialogue playList="1"/>
```

---

playBatch

type: number
value: "X" (number of lines to release from the front of the dialogue list)

Specifies a number of dialogue lines to pull and play from the front of the action's dialogue list. Use this method to release batches of dialogue in sequence, where the first X lines of dialogue remaining in the action's dialogue list will be pulled and played.

Examples:

```
<!-- dialogue list has 4 lines -->
<dialogue playBatch="2"/>
<!-- First two lines are played, now the dialogue list has only 2
lines left -->
<dialogue playBatch="1"/>
<!-- First line is played, now the dialogue list has only 1 line
left -->
<dialogue play="*"/>
<!-- All remaining dialogue is played. -->
```

flash

Availability: Lassie Player v1.0

The flash command currently has only one purpose: it is used to dispatch custom event types off from the Flash stage.

Basic example:

```
<flash stageEvent="eventTypeName"/>
```

Required Parameters

stageEvent

type: string
value: "eventTypeName"

Use this parameter to dispatch a custom event type within the Flash environment. The event will be broadcast through the Flash Stage, which is the top-level display of the Flash player. All Flash media has the ability to reference "stage" once it exists within an active display list. This feature allows you to subscribe external Flash media as listeners of the stage, then broadcast messages to them through the stage.

Tech note: when subscribing Flash media as listeners of the stage, be sure to use a weak listener reference or else remove the listener after the media element has been deactivated. Otherwise, a registered stage listener will not be eligible for Flash garbage collection and will become an overhead memory resource that will hinder Flash Player.

Examples:

```
<flash stageEvent="eventTypeName"/>
```

game

Availability: Lassie Player v1.0

The game command configures the setup of the current game session. This includes everything from the current game actor to the master language setting. This command's most common use is to set the current game room, which will clear the existing room layout and introduce the new scene.

Basic example:

```
<game room="beach:path" actor="patrick"/>
```

Required Parameters

One or more option parameters are required.

Optional Parameters

room

type: string
value: "roomId:positionId"

Developers will likely find that the room parameter is their primary use for the `game` command. The room setting will clear any existing room layout, and then introduce the new specified room. A room is specified with two components, formatted as "roomId:positionId". The "roomId" specifies the room layout to set as the new game room, while the "positionId" specifies a grid node position in that layout at which to position the avatar.

When setting the game room, a sequence of events takes place: first the current room layout's exit script is called, then the game room is changed, then the new room's enter script is called. It is important to note that fade transitions are not automatically called while switching rooms. To have smooth fade-to-black transitions between rooms, you must add a curtain command into each room layout's enter and exit scripts (these are configured by default within the Shepherd editor). This manual call for transitions gives a developer the ability to sequence other intro/outro room actions around the curtain's fade in and out.

Examples:

```
<game room="beach"/>
<game room="beach:join_dunes"/>
```

actor

type: string
value: "roomId:positionId"

Developers will likely find that the room parameter is their primary use for the game command.

Examples:

```
<game actor="patrick"/>
```

inventory

Availability: Lassie Player v1.0

The `inventory` command manages the contents of inventory collections and the display of the game's inventory panel. Use this command to add, remove, and replace individual items or groups of items from inventory collections, or to hide and reveal the game's inventory panel.

Basic example:

```
<inventory target="main" add="coconut"/>
```

Required Parameters

One or more of the optional parameter must be specified.

Optional Parameters

target

type: string
values: "collectionId" (targets a specific collection), or "_current" (targets currently active collection)

Specifies a target inventory collection ID to access. A valid inventory collection Id must be provided, or specify "_current" to target the currently active game inventory. If the "target" setting is not defined within an <inventory> command, then "_current" will be assumed as the command's target.

Examples:

```
<inventory target="main" add="coconut"/>
<inventory target="_current" add="coconut"/>

// This is also valid... it will assume "_current" as the target setting.
<inventory add="coconut"/>
```

add

type: comma-separated list
value: "item1,item2,item3"

Specifies an item ID or a list of item IDs to add to the target inventory. All new items will be added onto the end of the existing inventory contents. To add a single item, specify a single inventory item ID. To add multiple items, specify a list of item IDs seperated by commas (ex: "item1,item2,item3"). Note: comma seperated lists should NOT include spaces between items.

Examples:

```
<inventory target="_current" add="itemId"/>
<inventory target="_current" add="item1,item2,item3"/>
```

[add] contentLimit

availability: Lassie Player v1.1
type: number

Optional parameter which can be used to customize the behavior of the "add" command. A "contentLimit" specifies the maximum number of instances of an item that may be placed into the inventory at the same time. All items within the "add" list will have their instances counted within the inventory, and then will only be added to the inventory if their current instance count is below the content limit. Note that a content limit will NOT remove items from the inventory if an item's instance count is already over the limit; a content limit will only prevent the addition of new instances of the item.

Examples:

```
// do not allow more than 2 "coconut" instances into the inventory
at the same time:
<inventory target="_current" add="coconut" contentLimit="2"/>
```

[add] historyLimit

availability: Lassie Player v1.1
type: number

Optional parameter which can be used to customize the behavior of the "add" command. A "contentLimit" specifies the maximum number of instances of an item that may be placed into the inventory at the same time. All items within the "add" list will have their instances counted within the inventory, and then will only be added to the inventory if their current

instance count is below the content limit. Note that a content limit will NOT remove items from the inventory if an item's instance count is already over the limit; a content limit will only prevent the addition of new instances of the item.

Examples:

```
// do not allow more than 2 "coconut" instances to EVER be added into the inventory:
<inventory target="_current" add="coconut" historyLimit="2"/>

// Only allow 1 "coconut" instance into the inventory at a time, and limit total allowed "coconuts" to 2:
<inventory target="_current" add="coconut" contentLimit="1" historyLimit="2"/>
```

---

remove

type: comma-separated list
value: "item1,item2,item3" or "" (all)*

Specifies an item ID or a list of item IDs to remove from the target inventory, or an asterisk ("*") may be specified to remove all items within the inventory. If multiple instance of a removal item exist within the inventory, only the first instance of that item will be removed. To remove multiple items, specify a list of item IDs separated by commas (ex: "item1,item2,item3"). Note: comma separated lists should NOT include spaces between items.

Examples:

```
<inventory target="_current" remove="itemId"/>
<inventory target="_current" remove="item1,item2,item3"/>
<inventory target="_current" remove="*"/>
```

---

purge

type: comma-separated list
value: "item1,item2,item3"

Specifies an item ID or a list of item IDs to completely purge from the target inventory. All instances of a purged item will be removed from the target inventory. To purge multiple item IDs, specify a list of item IDs separated by commas (ex: "item1,item2,item3"). Note: comma separated lists should NOT include spaces between items.

Examples:

```
<inventory target="_current" purge="itemId"/>
<inventory target="_current" purge="item1,item2,item3"/>
```

---

replace

type: comma-separated list
value: "oldItem:newItem1,newItem2,newItem3"

Replaces the first instance of a specified item ID with another item or a list of item IDs. The target item will be removed from the inventory, and then the replacement item(s) will be inserted at the removal index. To replace a single item with multiple items, specify a list of item IDs separated by commas as the replacement (ex: "item1,item2,item3"). Note: comma separated lists should NOT include spaces between items.

Examples:

```
<inventory target="_current" replace="oldItem:newItem"/>
<inventory target="_current"
replace="oldItem:newItem1,newItem2,newItem3"/>
```

---

mouseAction

type: string
values: "over" / "out"

Applies a mouse action to the inventory panel display. This parameter accepts one of two values, and requires child attributes be present to assign as a script:

- **over** : Called upon mouse over.
- **out** : Called upon mouse out.

Examples:

```
<inventory mouseAction="out">
    <inventory display="hide"/>
</inventory>
```

---

display

type: string
values: "show" / "hide" / "toggle"

Hides and reveals the inventory interface display. This parameter accepts one of three values:

- **show** : Reveals the inventory interface display.
- **hide** : Hides the inventory interface display.
- **toggle** : Toggles the visibility of the inventory: display is hidden if visible, or revealed if invisible.

Examples:

```
<inventory display="show"/>
<inventory display="toggle"/>
```

---

dynamicDisplay

availability: Lassie Player v1.1
type: boolean
values: "1" (true) / "0" (false)

Enables or disables dynamic inventory visibility. This setting configures the inventory to behave as a show-and-hide game overlay panel. When "dynamicDisplay" is enabled, the inventory panel will track mouse events and automatically hide itself when the inventory panel is rolled out from or clicked off of. The inventory will then require a manual prompt to display again, which is generally achieved by assigning an <inventory display="show"/> command to a key stroke. By default, "dynamicDisplay" is set to FALSE, meaning that the inventory will display as a static panel. Note that this setting may be configured within the game's "system.xml" settings.

Examples:

```
<inventory dynamicDisplay="1"/>
<inventory dynamicDisplay="0"/>
```

---

scrollTo

availability: Lassie Player v1.1
type: string or number
values: "X" (page index) / "itemId" / "_blank"

Specifies a target value to scroll the inventory page to. A "page" is one single view of items within a scrolling inventory display. Scroll target definitions include:

- **"X" (any number):** Any valid integer may be specified as the page number to scroll to. Note that inventory pagination is zero-indexed, meaning that the first page of items is "0", the second page is "1", etc. Specifying a page index greater than the total number of page views will automatically limit the value to the total number of pages.
- **"itemId":** An item ID to locate an scroll to display within the inventory. When multiple instances of the target item are present within the inventory, then the inventory will be scrolled to display the first instance. If the target item is not present within the inventory, then the scroll request will be ignored.
- **"_blank":** This keyword scrolls the inventory to the first blank item slot available at the end of the inventory. A new blank page is displayed if necessary. Use this keyword prior to inserting a new item into the inventory to make sure that the player can see the new addition.

keyboard

Availability: Lassie Player v1.0

The `keyboard` command assigns Lassie XML scripts to keyboard keys. An assigned XML script is called upon press of its corresponding key. Also, this command can be used to enable and disable the responsiveness of individual keyboard keys without modifying the XML scripts assigned to them.

When specifying keyboard targets, a literal key value is specified that the Lassie Player will automatically convert to a key code. All key targets should be lowercase. Valid key targets include the following:

- **"a-z"**: a letter key (specify only one at a time; ex: "a", "b", or "z").
- **"0-9"**: any standard number key (Does not include number pad. Specify only one at a time; ex: "0", "1", or "9").
- **"f1-f10"**: a function key (specify only one at a time; ex: "f1", "f2", or "f10").
- **"tab"**: the TAB key.
- **"space"**: the SPACE bar.
- **"esc"**: the ESCAPE key.
- **"up", "down", "left", "right"**: the arrow keys.

Note that some restrictions do apply to keyboard usage due to Flash Player configuration and security restrictions. The escape key ("esc") is reserved by Flash Player as the manual trigger for exiting fullscreen mode. Therefore, it is advisable to NOT hook events to the ESC key given that the key already has a purpose. Also, Flash Player restricts keyboard input while in fullscreen mode. Upon entering fullscreen mode, Flash Player will only respond to "tab", "space", and the arrow keys.

Basic example:

```
<keyboard addKeyScript="i">
    <inventory display="toggle"/>
</keyboard>
```

Required Parameters

One or more of the optional parameter must be specified.

Optional Parameters

addKeyScript

type: string
value: "keyValue" ("e", "x", etc)

Adds an XML script to the specified keyboard key. The assigned XML script will be called each time the specified key is pressed. The value of the keyboard key is defined as a lower case letter matching the keyboard value. The script that is attached to the key is a list of child XML nodes contained within the main `<keyboard>` command node. Note that assigning a key script will replace any existing script attached to the specified key.

Examples:

```
<keyboard addKeyScript="i">
    <inventory display="toggle"/>
</keyboard>
```

---

removeKeyScript

type: string
value: "keyValue" ("e", "x", etc)

Removes any existing key script that is currently assigned to the specified keyboard key. The value of the keyboard key is defined as a lower case letter matching the keyboard value. Calling this method on a key without an assigned key script will have no result.

Examples:

```
<keyboard removeKeyScript="i"/>
```

---

enableKey

type: string
value: "keyValue" ("e", "x", etc) or "" (all)*

Enables keyboard response from the specified keyboard key, defined as a lower case letter matching the key value. In addition, "*" is also a valid entity used for enabling the master keyboard response switch.

Examples:

```
<keyboard enableKey="i"/>
```

---

disableKey

type: string
value: "keyValue" ("e", "x", etc) or "" (all)*

Disables keyboard response from the specified keyboard key, defined as a lower case letter matching the key value. Disabled keys will not call their associated XML scripts when pressed. Use the entity "*" to disable all keyboard response.

Examples:

```
<keyboard disableKey="i"/>
```

layer

Availability: Lassie Player v1.0

The layer command sets properties of a room layer. The target layer may be any layer type (background, plane, or puppet), and may be located within any room of the game. Properties set using the `layer` command will be permanent changes within game configuration. If the target layer is present within the currently active room layout, then the layer's graphical appearance will be updated to reflect the new layer settings. To modify graphical properties of layers within the active room layout ***without*** permanently changing the layer's configuration, use the `layerSprite` command.

Basic example:

```
<layer target="_current:coconut" visible="0"/>
```

Required Parameters

target

type: string
value: "roomId:layerId" or "_current:layerId"

Specifies a room layer to target with the `layer` command. A layer is targeted using the ID of the room containing the layer, and then the ID of the target layer within that room. These two targeting elements are formatted as "roomId:layerId". Use "_current" as the room ID to target a layer within the currently active room layout. If both targeting elements are not specified, then the script will default to using "_current" as the room target.

Examples:

```
<layer target="_current:coconut" visible="0"/>
<layer target="beach:coconut" visible="0"/>
```

Optional Parameters

omit

type: boolean
values: "1" (true) or "0" (false)

Flags the layer to be omitted from the room layout in the future when the room is rendered.

When a layer is omitted, it will be completely excluded from the room layout (not just made invisible). This optimizes game performance: rather than just hiding elements that will not longer be needed in the room, you can completely omit them from being displayed in the future. Omitting a layer supersedes a layer's "render" setting. If a layer is set to render but has been omitted, then the "render" setting will be ignored.

Note that omitting a layer will not affect the game display until the next time the room loads. If the target layer is present within the current room, then enabling its "omit" setting will NOT immediately remove the layer from the display. If you'd like to immediately hide an element and specify that it should not render in the future, disable the layer's visibility and enable the layer's omit setting in a single layer command.

Example:

```
<layer target="_current:coconut" visible="0" omit="1"/>
```

---

render

type: boolean
values: "1" (true) or "0" (false)

Specifies if the layer should be rendered into the room layout in the future when the room loads. When a layer's "render" setting is set to false ("0"), it will be completely excluded from the room layout (not just made invisible). This optimizes game performance: rather than just hiding a disabled element, you can tell it not to render into the room until further notice. Note that rendering is superseded by a layer's "omit" setting. Omitting a layer will permanently disable a layer's render setting.

Also note that disabling a layer's "render" setting will not affect the game display until the next time the room loads. If the target layer is present within the current room, then disabling it's render setting will NOT immediately remove the layer from the display. If you'd like to immediately hide an element and specify that it should not render in the future, disable the layer's visibility and disable the layer's render setting in a single layer command.

Example:

```
<layer target="_current:coconut" visible="0" render="0"/>
```

---

state

type: string
value: "stateId"

Specifies a layer-state Id to set as the layer's active display state.

Example:

```
<layer target="_current:door" state="open"/>
```

---

update

type: boolean
value: "0" (false)

Specifies if other setting applied during the current <layer> command should immediately update the display of the target layer. By default this value is TRUE, meaning that all settings applied to the layer (with the exception of "omit" and "render") will immediately update the layer's appearance – if it's on screen. Given that this value is TRUE by default, you only need to declare the "update" setting if you do NOT want changes to the layer to be immediately reflected in its display.

Example:

```
<layer target="_current:coconut" visible="0" update="0"/>
```

---

visible

type: boolean
values: "1" (true) or "0" (false)

Toggles the layer's visibility. When FALSE, the layer will still be present on screen but will be made invisible. When permanently disabling a layer from a room layout, toggle the layer's "omit" setting to complete exclude the layer from the room in the future.

Example:

```
<layer target="_current:coconut" visible="0"/>
```

layerBlend

Availability: Lassie Player v1.2

The `layerBlend` command animates the graphics of layers which are currently present on screen. These layer animations are simple property tweens of a layer's alpha (opacity), position, scale, or rotation. You can use this command to fade layers in and out (alpha), pop layers in and out (scale), or slide layers on and off screen (position). The `layerBlend` command may ONLY target layers rendered into the active game room (layers with their "visible" property turned off are still valid targets). Appearance modifications made by the layerBlend command will NOT be cached in the game model, so changes in appearance will be reset upon reloading the parent room layout. Therefore, a layerBlend is best used in conjunction with the layer or layerState command to register a permanent change in appearance of the layer.

Basic example:

```
<layerBlend target="detail_panel" from="alpha:0,y:-50"
to="alpha:1,y:100" waitForComplete="1"/>
```

Required Parameters

target

type: string
value: "layerId"

Specifies a layer ID to target. The specified layer target MUST be present within the current game room.

Examples:

```
<layerBlend target="detail_panel" from="alpha:0" to="alpha:1"/>
```

---

to

type: property list
value: "property1:value1,property2:value2"

Specifies a list of properties and values to tween the layer appearance to. The assigned values will be the end point of the layer animation. Valid properties and their value scales

are as follows:

- **alpha**:[0 (transparent) to 1 (opaque)]
- **scaleX**:[0 (0% width) to 1 (100% width)]
- **scaleY**:[0 (0% height) to 1 (100% height)]
- **x**:[any X coordinate]
- **y**:[any Y coordinate]
- **rotation**:[0 to 360]

Examples:

```
<layerBlend target="detail_panel" to="alpha:1"/>
```

Optional Parameters

ease

type: string
value: "type:method"

Specifies an easing operation to animation the tween with. Easing is specified with two components, a type and a method, formatted a "type:method". There are numerous types of easing, each with a unique visual personality. An ease type like "strong" will create a very bold move, while another like "elastic" will create playful movement. The best way to get familiar with easing is just to experiment with and preview the different types. All types of easing have three basic methods: ease-in, ease-out, and ease in-out. Valid values for the "type" and "method" components are defined in the following lists.

Ease type values:

- "back"
- "bounce"
- "circ"
- "cubic"
- "elastic"
- "expo"
- "linear"
- "quad"
- "quart"
- "quint"
- "sine"
- "strong"

Ease method values:

- "easeIn"
- "easeOut"
- "easeInOut"

Examples:

```
<layerBlend target="detail_panel" to="alpha:1"
ease="strong:easeInOut"/>
```

---

from

type: property list
value: "property1:value1,property2:value2"

Specifies a list of properties and values to tween the layer appearance from. The assigned values will be the start point of the layer animation, and will be applied to the layer's appearance immediately upon starting the animation. If the "from" property is not specified, then the animation will start from the layer's current display values. Valid properties and their value scales are as follows:

- **alpha**:[0 (transparent) to 1 (opaque)]
- **scaleX**:[0 (0% width) to 1 (100% width)]
- **scaleY**:[0 (0% height) to 1 (100% height)]
- **x**:[any X coordinate]
- **y**:[any Y coordinate]
- **rotation**:[0 to 360]

Examples:

```
<layerSprite target="coconut" from="alpha:0" to="alpha:1"/>
```

---

seconds

type: number
value: "X" (number of seconds)

Specifies the number of seconds over which to perform the layer animation. By default, this value is 2.

Examples:

```
<layerBlend target="detail_panel" to="alpha:1,scaleX:1,scaleY:1"
seconds="3"/>
```

---

waitForComplete

type: boolean
value: "1" (true) or "0" (false)

Specifies if additional actions within the script should wait until this animation completes before proceeding. By default, this value is TRUE.

Examples:

```
<layerBlend target="detail_panel" to="alpha:1" seconds="3"
waitForComplete="1"/>
```

layerSprite

Availability: Lassie Player v1.0

The `layerSprite` command manipulates the graphics of layers which are currently present on screen in the active room layout. This command is similar in nature to the `layer` command, but has some unique purposes. Unlike the `layer` command, `layerSprite` ***cannot*** target layers in any room except the currently active game room. This command is designed to control graphics currently on screen without having a lasting effect on the room layout. As such, graphical properties set using layerSprite will ***not*** be retained after the room is unloaded. Upon exiting and returning to a room, the room's layout will load using all source `layer` properties. To permanently change layer properties, use the `layer` command.

In addition, the layerSprite command also has the ability to hook into the current room display and monitor animation playback. This is an important feature when scripting sequences that rely on commands firing in sequence around animations.

Finally, the layerSprite command supports some run-time graphics features which cannot be permanently assigned to a layer. Use this command to achieve advanced layering control in tandem with dynamically tweening layers around the room.

Basic example:

```
<layerSprite target="chicken" animFrame="cluck"
waitForComplete="1"/>
```

Required Parameters

target

type: string
value: "layerId"

Specifies a layer ID to target. Given that the <layerSprite> command only deals with layers currently on screen, the specified layer target MUST be present within the current game room.

Examples:

```
<layerSprite target="coconut" visible="0"/>
```

Optional Parameters

animFrame

type: string
value: "frameLabel"

Sets the frame label of the target layer's MovieClip image. This is specifically designed to set the layer to a frame with an animation placed on it. Use the "turnView" parameter in conjunction with "animFrame" to access a specific directional view of the animation. Also, use the "waitForComplete" parameter in conjunction with "animFrame" to suspend additional command processing until an ANIMATION_COMPLETE even is recieved from the layer animation. For more information on waiting for animations to finish, see to "Calling Animation Complete" section of this document.

Examples:

```
<layerSprite target="chicken" animFrame="cluck"/>
<layerSprite target="chicken" animFrame="cluck" turnView="3"/>
<layerSprite target="chicken" animFrame="cluck"
waitForComplete="1"/>
```

[animFrame] turnView

type: number
values: 1-8

Use this setting in conjunction with the "animFrame" parameter to specify a custom turn view (1-8) for the animation.

Examples:

```
<layerSprite target="chicken" animFrame="cluck" turnView="5"/>
```

[animFrame] waitForComplete

type: boolean
values: "1" (true) or "0" (false)

This property is used in conjunction with the "animFrame" property. When "waitForComplete" is enabled, the action processing is suspended after the layer's animation frame has been set. Action processing is not resumed until an ANIMATION_COMPLETE event is recieved from the layer's Flash animation, indicating that the timeline animation has

finished. By default, this value is false – meaning that the Lassie Player will not wait for an animation to complete unless specifically told to do so.

Examples:

```
<layerSprite target="chicken" animFrame="cluck"
waitForComplete="1"/>
<layerSprite target="chicken" animFrame="cluck"
waitForComplete="0"/>
```

---

floatBehind

type: string
value: "layerId"

This parameter forcibly stacks the target layer behind another layer within the room. The term "float" refers to the index of layers within the layering stack; in other words, the stack that causes layers to display above or below one another. The basic float setting (which can be enabled within the Shepherd Editor) applies a simple Y-position filter across layers, causing a floating layer with lesser Y-position (higher on screen) to stack below a floating layer with greater Y-position (lower on screen). However, this basic setting does not address complex layering scenario where a layer needs to stack both above and below another layer, regardless of Y-position.

For example, consider the situation of making the avatar walk over a hill. The hill layer's registration point is placed at the crest of the hill. The avatar is registered at the position of its feet. The avatar walks up the hill while infront of the hill layer, passes the crest of the hill, then moves back down behind the layer. In this situation, the avatar must be both in front of and behind the hill while its registration point remains consistently below the hill's registration. Therefore, a simple layer float does not work in this situation.

So, this is where the "floatBehind" setting comes into play. floatBehind defines another layer ID within the room that the target layer should permanently draw behind. In our above situation, we'd apply a floatBehind script to the grid node at the crest of the hill. When the avatar reaches the crest of the hill, we'd assign it [floatBehind="hill"], which would forcibly layer the avatar behind the layer sprite, regardless of their relative Y-positions.

Examples:

```
<layerSprite target="_avatar" floatBehind="hill"/>
```

---

frameOffset

type: number
values: "1" or "-1"

Defines a numeric offset for all frames set within the target layer's movieClip image. For example, a layer with a frameOffset of "1" will always go to frame "X+1" when its timeline frame is set. This feature can be used to define multiple graphics sets for an object while still treating the object as being in a single state.

Examples:

```
<layerSprite target="coconut" frameOffset="1"/>
<layerSprite target="coconut" frameOffset="-2"/>
```

---

imageMethod

availability: Lassie Player v1.1
type: expression
value: "methodName:param1,param2,param3"

Calls a method of the target layer's MovieClip image. This allows the developer to define custom ActionScript functions within their Flash media, then call those functions from Lassie. Custom methods called with "imageMethod" must be defined as members of the image MovieClip's root class, or else be defined within the MovieClip's root timeline actions (it is advisable to add methods directly to the MovieClip's class to guarantee availability. Methods added to the class must be public).

A method call is written with two components: "methodName:params", where "methodName" is the name of the function to call, and "params" is a comma-separated list of parameters to pass into the function. Note that the parameters list will NOT be parsed into data types, therefor you should declare all method arguments as string and then parse other values (numbers, booleans) from the raw input. When composing your method call, you are responsible for matching your function call to your custom method signature. Invalid method call errors will be caught and ignored by the Lassie Player (in effect: a failed method call will produce no result).

Example:

```
// within the image MovieClip's class:

public function triggerButton($num:String):void {
    var $index:int = parseInt($num);
    // take action with the provided button index...
}

// within Lassie:

<layerSprite target="control_panel" imageMethod="triggerButton:2"/>
```

---

invertTurn

type: boolean
values: "1" (true) or "0" (false)

Enables front-to-back turn view inversion on puppet layers. When a puppet's turn views are inverted, then front-facing animations will be replaced by back-facing animations and vice-versa. This is a handy feature to use with the "floatBehind" property, given that it addresses the same scenario that "floatBehind" is designed to address. Take the above "floatBehind" example that looks at an avatar walking over a hill. The avatar will walk over the hill with it's back toward the camera as it moves upwards across the stage. However, once the avatar crests the hill begins descending down the backside, the avatar will now be moving with a downward trend, which will normally animate the avatar with a front-facing view. However, we still want to see the avatar's back since it is moving away from camera, so we would invert its turn view at the crest of the hill.

Example:

```
<layerSprite target="_avatar" invertTurn="1"/>
<layerSprite target="_avatar" invertTurn="0"/>
```

---

register

availability: Lassie Player v1.1
type: boolean
values: "1" (true) or "0" (false)

Registers a layer's current display configuration to its data cache. When display configuration is registered to the cache, that configuration will be saved for the layer and persist between room and state changes. By default, this value is FALSE, meaning that any display manipulation performed by the <layerSprite> command will NOT be saved within the layer's source data.

Generally it is advisable to leave <layerSprite> unregistered (FALSE) and just use the <layer> and <layerState> commands when you need to set a persistent display value. That said, the real benefit of the "register" property is its ability to register the layer's current dynamically-applied configuration. In all other cases, a layer's configuration is only registered when a Lassie XML command assigns a new value. Using "register" however, the layer's currently configured display values are set directly to the cache, regardless of how the display was set to its current state. This is useful when working with commands like <layerTween>, which will tween a layer around the screen without retaining the layer's display coordinates.

When a layer is registered, the following four layer properties are committed to its cache:

- the layer's X-position
- the layer's Y-position
- the layer's current turn view

- the layer image's current display frame

Example:

```
<layerSprite target="chicken" register="1"/>
```

---

to

type: expression
values: "point:100,100" or "position:nodeId"

Moves a layer sprite to a new position within the room layout. The transition between the start and end points is immediate, meaning that the layer will NOT tween (animate) between the two positions. To animate a layer between two positions, use the layerTween command. The target distination is defined with two components, formatted as either "point:x,y" of "position:nodeId".

Examples:

```
<layerSprite target="chicken" to="point:100,100"/>
<layerSprite target="chicken" to="position:left_exit"/>
```

---

visible

availability: Lassie Player v1.1
type: boolean
values: "1" (true) or "0" (false)

Toggles the target layer's visibility on and off. This works the same as the <layer visible> command, however visibility changes set with the <layerSprite> command will not be retained after exiting the current game room. Use this command when you want to temporarily disable a layer's display, then have it restored upon reloading the room.

Examples:

```
<layerSprite target="coconut" visible="0"/>
```

---

waitForBreak

type: boolean
values: "1" (true) or "0" (false)

This property instructs the <layerSprite> command to hold off its processing until an "ANIMATION_BREAK" event is received from the layer sprite. This feature is most commonly used to transition smoothly between layer animations. For example, a layer may use a looping animation within its resting state. While creating a secondary animation for the layer, you have the secondary animation start with a single from within the main animation's loop. To smoothly transition between the two animations, you'd need to wait for the break frame to occur within the loop, then move to the secondary animation. This property is generally most useful in tandem with the "animFrame" property, however it may also be used on its own to set other <layerState> properties that do not involve animation.

Examples:

```
<layerSprite target="chicken" waitForBreak="1" animFrame="cluck"/>
<layerSprite target="chicken" waitForBreak="1" visible="0"/>
```

layerState

Availability: Lassie Player v1.0

The layer state command sets display properties assigned to a single state of a room layer. While this is similar in nature to the general <layer> command, <layerState> drills deeper into a layer's configuration by manipulating specific layer states. When working with the layerState command, all changes made are registered with the layer's data model, so will be retained within saved game data and will persist between room sessions.

Note that using the <layerState> command with the avatar layer is slightly unique. Since the avatar layer is controlled through global configuration, the Shepherd Editor does not allow a developer to define custom states for the avatar. However, the avatar layer does have one single state behind the scenes that can be manipulated using the <layerState> command. This single avatar state is called "main". When targeting the avatar layer with a <layerState> command, just specify "main" as its state reference.

Also note, it is NOT possible to dynamically reference a layer's current state. This is imposed by Lassie Shepherd's loading model where rooms are loaded individually; therefore layer configurations are not available until the room loads. By always requiring a defined state ID, Shepherd can create place-holder fields for data set prior to the layer loading.

Basic example:

```
<layerState target="roomId:layerId:stateId" x="100"/>
```

Optional Parameters

x

type: number

The X-position of the layer within the corrdinates of its parent room layout.

---

y

type: number

The Y-position of the layer within the corrdinates of its parent room layout.

---

mapX

type: number

The X-position within the room that the avatar layer is moved to when the parent layer is interacted with.

---

mapY

type: number

The Y-position within the room that the avatar layer is moved to when the parent layer is interacted with.

---

imageEnabled

type: boolean
values: "1" (true) / "0" (false)

Specifies if the layer's image display is enabled. When disabled, no image asset display for the layer, even if one is defined.

---

imageX

type: number

The X-position of the layer's image asset within the local coordinates of the layer.

---

imageY

type: number

The Y-position of the layer's image asset within the local coordinates of the layer.

---

imageScaleX

type: number

The X-scale of the layer's image display, expressed as a percentage value between 0 and 1. This percentage may utilize an extended range, meaning that "2" could also be specified for 200%.

---

imageScaleY

type: number

The Y-scale of the layer's image display, expressed as a percentage value between 0 and 1. This percentage may utilize an extended range, meaning that "2" could also be specified for 200%.

---

frameLabel

type: string = ["frameLabel"]

Specifies the image MovieClip's frame label display.

---

blendMode

type: string = ["flash.display.BlendMode"]

Specifies the parent layer's blend mode as one of the Flash-native BlendMode constants. For a list of valid blend modes in the current Flash Player version, see Flash documentation on the *flash.display.BlendMode* class.

---

alpha

type: Number = ["0.5"]

Specifies the alpha (transparency) of the parent layer, expressed as a percentage value between 0 and 1.

---

cacheAsBitmap

type: boolean
values: "1" (true) / "0" (false)

Specifies if the layer should utilize runtime bitmap caching. Bitmap caching will generally increase graphics performance so long as the target layer is NOT animated.

---

hitEnabled

type: boolean
values: "1" (true) / "0" (false)

Specifies if the layer's custom hit area sprite is enabled. When enabled, the layer's hit area

sprite will define the layer's positive mouse-hit bounds. When disabled, the layer's graphical display will determine its positive hit area.

---

hitX

type: Number = ["0"]

Specifies the X-position of the layer's hit area within its parent layer's coordinate space.

---

hitY

type: Number = ["0"]

Specifies the Y-position of the layer's hit area within its parent layer's coordinate space.

---

hitWidth

type: Number = ["0"]

Specifies the width of the hit area shape.

---

hitHeight

type: Number = ["0"]

Specifies the height of the hit area shape.

---

hitShape

type: String = ["rect" / "ellipse"]

Specifies the shape of the layer hit area. A layer's hit area my be drawn as a rectangle or ellipse; either shape will be scaled to match the hit area width and height.

---

rotation

type: Number = [0 - 360]

Specifies the rotation, in degrees, of the layer display. Rotation is applied in a clockwise fashion.

---

floatEnabled

type: boolean
values: "1" (true) / "0" (false)

Specifies if the layer should implement custom depth tracking within the layer stack. See the "Room Layout Editor: Floats" article within the Lassie Shepherd Manual for a summary of how floating depth layers are sorted and stacked.

---

mouseEnabled

type: boolean
values: "1" (true) / "0" (false)

Specifies if the layer should track mouse events – including rollovers and clicks. This setting works differently depending on layer types as follows:

- **puppet layer:** When mouse events are enabled, a puppet will track Lassie-native rollover and click behaviors. When disabled, the puppet will be completely unresponsive to mouse behaviors, but may still have its interactions called using the <puppet callAction> command.
- **plan layer:** When mouse events are enabled, child media within the plane layer will be given access to Flash mouse events. Enable this setting on a plane layer when adding custom Flash media into a room layout (ex: adding a mini-game MovieClip). When mouse events are disabled, a plane will block all mouse event propagation from its children.
- **background layer:** No effect; mouse events are static on the background layer.
- **avatar layer:** No effect; mouse events are permanently disabled on the avatar layer.

---

hoverCursor

type: String = ["frameLabel"]

Specifies a cursor frame label to display when the layer is rolled over. This setting will only have an effect with puppet layers.

---

subtitleColor

type: Number = ["0xFF0000"]

Specifies a puppet's dialogue subtitle color. This setting will only have an effect with puppet layers.

---

tweenRate

type: Number = ["7"]

Specifies a puppet's rate of movement (pixels moved per frame) while being dynamically animated around a room layout. The greater the number, the faster the puppet will be moved. Comfortable movement speeds generally range between 5 to 10 pixels per frame.

---

clickAction

type: Number = ["0"]

Specifies a puppet action (referenced by index) to trigger when a puppet layer is clicked. This property will only have an effect if the targeted layer is a puppet.

---

turnView

type: number
value: directional view (number 1-8)

Specifies a directional turn view to switch the layer to. For more information about directional turn views, see the "Creating Multi-Directional Puppet Animation Views" article within the Lassie Shepherd Manual.

---

scaleFilter

type: String = ["" / "grid" / "filterId"]

Specifies the scale filter applied to a puppet layer. This property will only have an effect if the targeted layer is a puppet. The specified filter ID must be a valid filter that exists within the puppet's parent room layout. To disable this filter setting, set the value as empty (""). To hook the puppet into the active grid-based matrix, assign a value of "grid".

---

colorFilter

type: String = ["" / "grid" / "filterId"]

Specifies the color filter applied to a puppet layer. This property will only have an effect if the targeted layer is a puppet. The specified filter ID must be a valid filter that exists within the puppet's parent room layout. To disable this filter setting, set the value as empty (""). To hook the puppet into the active grid-based matrix, assign a value of "grid".

---

speedFilter

type: String = ["" / "grid" / "filterId"]

Specifies the speed filter applied to a puppet layer. This property will only have an effect if the targeted layer is a puppet. The specified filter ID must be a valid filter that exists within the puppet's parent room layout. To disable this filter setting, set the value as empty (""). To hook the puppet into the active grid-based matrix, assign a value of "grid".

---

blurFilter

type: String = ["" / "grid" / "filterId"]

Specifies the blur filter applied to a puppet layer. This property will only have an effect if the targeted layer is a puppet. The specified filter ID must be a valid filter that exists within the puppet's parent room layout. To disable this filter setting, set the value as empty (""). To hook the puppet into the active grid-based matrix, assign a value of "grid".

---

parallaxAxis

type: String = ["" / "x" / "y" / "xy"]

Specifies the parallax axis of a plane layer. This property will only have an effect if the targeted layer is a plane. To disable parallax scrolling, set the "parallaxAxis" property to empty ("").

---

update

type: boolean
values: "1" (true) / "0" (false)

Specifies if the layer's display should be updated after applying new settings. This value is TRUE by default, meaning that a layer's display will be automatically updated with new settings unless the "update" property is *specifically* set to FALSE ("0"). If the layer is not present on stage when the <layerState> command is run, then display updates will have no effect.

layerTween

Availability: Lassie Player v1.0

The `layerTween` command can be used to perform a basic location shift of a layer or group of layers around a room. This command does NOT provide smart motion that follows the walkable grid or sets directional movement animations (those features require using the `puppetTween` command).

Basic example:

```
<layerTween target="_avatar,elevator" by="x,-300"/>
```

Required Parameters

targets

type: string
value: "layer1,layer2"

Specifies a comma-seperated list of target layer ID's that will be animated by this tween. Targeting multiple layers will move the targeted layers as a group.

Examples:

```
<layerTween targets="_avatar" to="50,100" seconds="2"/>
<layerTween targets="_avatar,elevator" to="50,100" seconds="2"/>
```

Optional Parameters

to

type: coordinates
value: "x,y"

Specifies a set of coordinates to tween the group of layers to, formatted as "x,y". When plotting a target position, a bounding box is drawn around all target layers within the tween group. The specified to coordinates will be target position that the group bounding's upper-left corner is tweened to.

You may leave one of the coordinates specified as a literal value of "x" or "y" to leave the group bounding's position unchanged along that axis. For example, "0,y" would shift the

layer group to the extreme left while leaving the group's Y position unchanged. Likewise, "x,0" would shift the layer group to the extreme top, while leaving the group's X position unchanged. Coordinates are NOT validated by the room size, so layers may be tweened out of bounds of the room's scrollable area.

Examples:

```
<layerTween targets="_avatar,elevator" to="50,100" seconds="2"/>
<layerTween targets="_avatar,elevator" to="50,y" seconds="2"/>
<layerTween targets="_avatar,elevator" to="x,100" seconds="2"/>
```

---

by

type: coordinates
value: "x,y"

Specifies a set of coordinates, formatted as "x,y", to offset the current viewport by. For example, scrolling by "-50,100" would scroll the viewport 50 pixels left and 100 pixels down from the current viewport position. You may leave one of the coordinates specified as a literal value of "x" or "y" to leave scrolling unchanged along that asix. For example, "-50,y" would scroll the room 50 pixels to the left while leaving the viewport's Y position unchanged. Likewise, "x,-50" would scroll the room 50 pixels up while leaving the viewport's X position unchanged. All offset coordinates will be validated and limit scrolling to within the bounds of the room's background layer.

Examples:

```
<layerTween target="_avatar,elevator" by="-50,100" seconds="2"/>
<layerTween target="_avatar,elevator" by="-50,y" seconds="2"/>
<layerTween target="_avatar,elevator" by="x,-50" seconds="2"/>
```

[to / by] seconds

type: number
value: "X"

Specifies the number of seconds over which to animate the group of layers. By default, this value is 2 seconds.

Examples:

```
<layerTween target="_avatar,elevator" to="0,0" seconds="2.5"/>
```

[to / by] ease

type: string
value: "type:method"

Specifies an easing operation to animation the tween with. Easing is specified with two components, a type and a method, formatted a "type:method". There are numerous types of easing, each with a unique visual personality. An ease type like "strong" will create a very bold move, while another like "elastic" will create playful movement. The best way to get familiar with easing is just to experiment with and preview the different types. All types of easing have three basic methods: ease-in, ease-out, and ease in-out. Valid values for the "type" and "method" components are defined in the following lists.

Ease type values:

- "back"
- "bounce"
- "circ"
- "cubic"
- "elastic"
- "expo"
- "linear"
- "quad"
- "quart"
- "quint"
- "sine"
- "strong"

Ease method values:

- "easeIn"
- "easeOut"
- "easeInOut"

Example:

```
<layerTween target="_avatar,elevator" to="0,0" seconds="2"
ease="bounce:easeOut"/>
```

[to / by] waitForComplete

type: boolean
values: "1" (true) or "0" (false)

Specifies whether the process queue should wait for the tween animation to complete before processing subsequent commands. The default is true, meaning that the queue will wait for the tween sequence to complete unless explicitly told not to.

Examples:

```
<layerTween targets="_avatar,elevator" to="0,0" seconds="2"
waitForComplete="1"/>
<layerTween targets="_avatar,elevator" to="0,0" seconds="2"
waitForComplete="0"/>
```

library

Availability: Lassie Player v1.0

The library command manages the loading and unloading of external media libraries. You will not need to use this command unless you wish to perform specialized asset management. By default, the Lassie Player automatically configures library loads for the game upon startup and for each room layout brought in. You can use this command to load specific media at specific times, or to unload media that will no longer be needed within the game to free up memory.

Basic example:

```
<library load="lib/myLib.swf"/>
```

Required Parameters

One of the optional parameter must be specified.

Optional Parameters

load

type: string
value: "file_path.swf" (library to load)

Loads the specified library SWF into the Lassie Player media library.

Examples:

```
<library load="lib/myLib.swf"/>
```

---

unload

type: string
value: "file_path.swf" (library to unload)

Unloads the specified library SWF from the Lassie Player media library. This will free up the library's memory upon the next cycle of the Flash Player garbage collector. Calling unload

on a library that was never loaded will have no effect.

Examples:

```
<library unload="lib/myLib.swf"/>
```

---

unloadRoom

type: string
value: "roomId" (room layout to unload), or "_current"

Unloads all assets associated with the specified room from the Lassie Player's media library. Specify "_current" to unload the currently active room. This script is most commonly used within a room's exit script so that room assets are purged as the player exits the scene. This will free up memory upon the next cycle of the Flash Player garbage collector. Calling unload on a room that was never loaded (or has already been unloaded) will have no effect.

Examples:

```
<library unload="lib/myLib.swf"/>
```

logic

Availability: Lassie Player v1.0

The logic command is used to communicate with the Lassie Player's Logic API, which provides an advanced level of engine control beyond the capacity of regular XML commands. The Logic API will recieve and parse logic expressions which can be used to customize the behavior of a game. For complete details, see the Logic API documentation.

The basic use of the logic command is to ask a question of the game engine, and then to take action if the answer is "yes". For example: is the current game room called "beach"? If so, then we want to take special action. See basic example below where we output a custom message into the debugger window if we find that the current room identifier is "beach".

Basic example:

```
<logic eval="[_currentRoom] EQ 'beach'">
    <debug echo="We are currently in the 'beach' room!">
</logic>
```

The above is a fast and easy way to create limited custom game behavior. However, it likely that you'll run into a scenario that requires a full logic tree. A logic tree asks several questions in sequence, continuing through the logic tree until one question proves true. A logic tree may also have a catch-all clause that will be called if all specific questions evaluated as false. A full logic tree is composed as:

```
<logic>
    <if eval="[_currentRoom] EQ beach">
        <!-- do custom actions here -->
    </if>
    <elseif eval="[_currentRoom] EQ terrace">
        <!-- do custom actions here -->
    </elseif>
    <else>
        <!-- do custom actions here -->
    </else>
</logic>
```

In the above example, the logic tree first evaluates the <if> clause to test if the current room is "`beach`", if so, the first set of custom actions are released, and the tree stops processing. If the first test fails, then the tree moves on to evaluate all <elseif> clauses in sequence until one clause proves true. There is no limit to the number of <elseif> clauses that you include in a tree. Also, you may completely omit <elseif> clauses if they are not

needed. If all <elseif> clauses evaluated as false, then the tree defaults to performing all custom actions provided in the <else> node. Again, the <else> node is optional. For the most part, you may free-form a logic tree however you want it, so long as the order of nodes is <if>, <elseif>, <else>.

Required Parameters

One or more of the optional parameter must be specified.

Optional Parameters

eval

type: string
value: "[logic expression]"

The eval parameter provides an expression for the Logic API to evaluate as being either TRUE or FALSE. That outcome is returned to the logic XML command, at which time all nested commands within the logic node will be run if the expression evaluated as TRUE.

Examples:

```
<logic eval="[_currentRoom] EQ 'beach'">
    <debug echo="We are currently in the 'beach' room!">
</logic>
```

---

parse

type: string
value: "[parse expression]"

The parse parameter provides an expression to be parsed by the Logic API. When an expression is parsed, it is simply run through the Logic API without returning any value. Any methods specified within the parse expression will be called by the Logic API. Upon completion of parsing, no additional action is taken by the logic XML command.

Examples:

```
<logic parse="{echo:'hello world'}">
```

menu

Availability: Lassie Player v1.0

The menu command opens and closes the game menu panel. The game menu includes controls such as loading, saving, and adjusting settings.

Basic example:

```
<menu display="show"/>
```

Required Parameters

display

type: string
values: "show", or "hide", or "toggle"

Hides and reveals the menu interface display. This parameter accepts one of three values:

- **show** : Reveals the inventory interface display.
- **hide** : Hides the inventory interface display.
- **toggle** : Toggles the visibility of the inventory: display is hidden if visible, or revealed if invisible.

Examples:

```
<menu display="show"/>
```

method

Availability: Lassie Player v1.0

The <method> command manages the Lassie Player's methodology table. In simplest terms, the Lassie Player's methodology table is a staging environment for XML scripts that will need to be called a some point in the future. The methodology table is an excellent way to synchronize game behavior across multiple room layouts. XML scripts defined within the methodology table are saved with game data, so any custom behavior configured through the methodology table will persist between game sessions. Also, methods can be called with custom parameters so that a single XML method can perform customized tasks.

Basic Use

In its basic use, the Lassie Player's methodology table can be used to store XML scripts between game rooms. For example, let's say that we have two rooms called "A" and "B". The player performs an action in room A which will lead to a custom sequence playing upon entering room B; however, this custom sequence should only play once following the trigger action that took place in room A.

This scenario *could* be done using cache values and conditional logic; however, the setup is tedious and adds a lot of unnecessary code complexity. Using the Lassie Player's methodology table offers a cleaner and more flexible alternative here. To implement this scenario using methodology, let's first specify a method to call every time that we enter room B. We'll label this methodology as "roomB_intro", and call it within room B's enterRoom script:

Room-B enterRoom script:

```
<method call="roomB_intro"/>
```

Once the above <method> call is in place within room B's enterRoom script, the method will now attempt to be called each time the player enters room B. There is no penalty for attempting to call a method that does not exist, so this method will only produce a result when and if an XML script has been defined for the method. Otherwise, the call will be ignored.

Now, let's move on to the trigger action in room A where we'll enable the room B intro. Within the room A trigger action, we'll add the following:

Room-A action trigger script:

```
<method define="roomB_intro">
  <!-- perform custom roomB intro command here! -->
```

```
  <method clear="roomB_intro"/>
</method>
```

In the above script, we define a method called "roomB_intro" within the Lassie Player methodology table. The method defines a list of XML commands which make up the room-B intro. This sequence of XML commands is NOT run at the time the method is defined; it is just stored within the methodology table. However, upon entering room B the method will be called thanks to our enterRoom method call!

Now, we only want the method to run once upon entering room B after having performed the trigger action in room A. This is actually very simple to do: note the last line of the <method> definition's XML script. The method definition includes a <method> command which clears itself from the methodology table. This will clear the method from the methodology table after being running once; so future entrances to room B will again have no result.

This system is also reusable. Let's say that in the future, a trigger action within room C will produce yet another custom intro sequence within room B. In that case, you already have the mechanism in place to handle this new intro: just define "roomB_intro" with another XML script over in room C, and it too will be called upon entering room B using the same method call that is already in place.

Ultimately, this many-to-one pattern of scripting (many triggers setting scripts to a single outlet) can also be used to control the traffic of your script flow. You can overwrite methodologies by defining a new method using an existing name. That way, multiple triggers can all write scripts to a single method; then the caller will only respond to the most recent trigger that defined the method.

Dynamic Parameter Parsing

The <method> command can also be used to dynamically parametrize scripts. That means custom parameters defined within the method call may be parsed into the method's XML text.

Let's say that you need an easy way to enabled and disable game controls. Whenever the player enters a scripted sequence, you need to disable the game mouse and keyboard input so that those controls cannot be used to interrupt the sequence flow. You could do this by manually writing a <cursor> and <keyboard> XML command at the start and end of each game sequence to disable then re-enable the controls, however that is cumbersome, and also doesn't allow for any future expansion of those controls. Let's say in the future you add a new UI layer to your game which also needs to be disabled during game sequences. You would manually need to go through your whole Shepherd project and update script with this new UI control toggle... that's a lot of work.

The easy alternative here is to set up a methodology that would handle control toggling. A single XML methodology could be written to both enable and disable mouse and keyboard controls. Here's the XML methodology:

```
<method define="controls" enabled="1" cursor="1">
    <cursor gameMouse="@enabled" visible="@cursor"/>
    <logic>
        <if eval="@enabled EQ 1">
            <keyboard enableKey="*"/>
        </if>
        <else>
            <keyboard disableKey="*"/>
        </else>
    </logic>
</method>
```

The above methodology script can be used to both enable and disable controls. The methodology's behavior is controlled by custom parameters that are parsed into the methodology script. Let's break down what's going on in the above example:

First, a method is being added to the Lassie Player's methodology table using the "add" attribute. A method is added using a reference name, in this case we're calling the method "controls". In the future, we can call and/or remove this method be referencing the name "controls".

Also included within the <method> node are a series of custom parameter attributes. In this example, we're specifying "enabled" and "cursor" as custom parameters of this method. When we call the "controls" method, we'll be able to include those parameters within the method call to have them parsed into the script.

Finally, the script that will run when this method is called is written within the <method> node. Note the "@" symbols included within the XML script. The "@" denotes a field to be parses into the script text. In the case of "@enabled", that field will be filled in when the method runs by the "enabled" parameter of the method call. So, now let's call this method in two different ways, and what the resulting script will be:

First Call to Disable Controls:

```
<method call="controls" enabled="0" cursor="0">
```

This call produces the following script:

```
<script>
    <cursor gameMouse="0" visible="0"/>
    <logic>
        <if eval="0 EQ 1">
            <keyboard enableKey="*"/>
        </if>
        <else>
            <keyboard disableKey="*"/>
        </else>
```

```
    </logic>
</script>
```

When the resulting script runs, the script will disable mouse clicks, hide the cursor, and disable all keyboard input.

Second Call to Enable Controls:

```
<method call="controls" enabled="1">
```

This call produces the following script:

```
<script>
    <cursor gameMouse="1" visible="1"/>
    <logic>
        <if eval="1 EQ 1">
            <keyboard enableKey="*"/>
        </if>
        <else>
            <keyboard disableKey="*"/>
        </else>
    </logic>
</script>
```

When the resulting script runs, the script will enable mouse clicks, show the cursor, and enable all keyboard input.

Following through the above examples, we see that the custom parameters of the method call were parsed into the script which modified the script's behavior. This works exactly like calling an XML script using the <script> command, which will also parse XML parameters in the script. However, using the methodology table has one crucial difference, as seen in our second call example while enabling controls: the methodology table retains default values for parameters.

Look again at the second method call... there is no "cursor" parameter provided. Why, then, was a value of "1" parsed in for the "cursor" parameter when the method was called? It is because a default value was specified for the "cursor" parameter while adding it to the methodology table. If no custom parameter value is provided, then a methodology command will fill in fields with all default values that were provided when the method was added to the methodology table.

Required Parameters

One and only one of the optional parameters is required.

Optional Parameters

You may add custom parameters as method arguments while defining and calling a method.

call

Calls a method by name. If there is no method with the specified name defined within the methodology table, the method call is ignored.

Basic example:

```
<method call="mymethod"/>
```

---

clear

Clears the specified method. If no method exists with the specified name, then the clear request will be ignored.

Basic example:

```
<method clear="mymethod"/>
```

---

define

Defines a method within the methodology table. Once a method has been defined, it may be called using the <method call=""> command.

Basic example:

```
<method define="mymethod">
    <!-- DO STUFF -->
</method>
```

params

Availability: Lassie Player v1.2

A <params> node is used to control the default engine behavior associated with puppet actions. This node currently has two specific uses:

- To disable the engine's default "walk-to" behavior when interacting with a puppet.
- To exempt an puppet from the engine's default "quick-exit" (double-click to immediately exit a room) behavior.

This node may ONLY be used in puppet action scripts, and should generally be included as the first element in the script. A <params> node is not technically an XML command, therefor calling a <params> node as part of a non-puppet script will have no result. In practice, a puppet action script will be searched for a <params> node before being called. If a <params> node is found, it is read, removed from the script, and the script processes as normal. While reading the params node, the Lassie engine will adjust default engine behaviors associated with puppet interactions. The <params> node currently supports two properties:

walkTo = ["auto" / "none"]
quickExit = ["1" / "0"]

process

Availability: Lassie Player v1.0

The <process> command forks a list of XML commands into a new process thread.

In simplest terms, a "process" is a queue of actions being run by the Lassie Player. When an XML script is sent to the game controller, the individual commands are dropped into a queue where each command is run in sequence. This process queue runs until all commands have finished, or until the process is aborted. Single processes are pretty easy to follow given that they are linear. Let's have a look at an example. Here's a script:

```
<debug enabled="1"/>
<wait seconds="1"/>
<debug echo="1-second wait complete!"/>
<wait seconds="2"/>
<debug echo="2-second wait complete!"/>
```

The above is a simple XML script that sends a sequence of messages to the Lassie Player's debugger window. The actions performed are as follows:

- enable the debugger window.
- wait 1 second.
- send a message to the debugger window.
- wait 2 seconds.
- send another message to the debugger window.

All told, that script plays out in sequence over the course of three seconds. To diagram that flow would look like this:

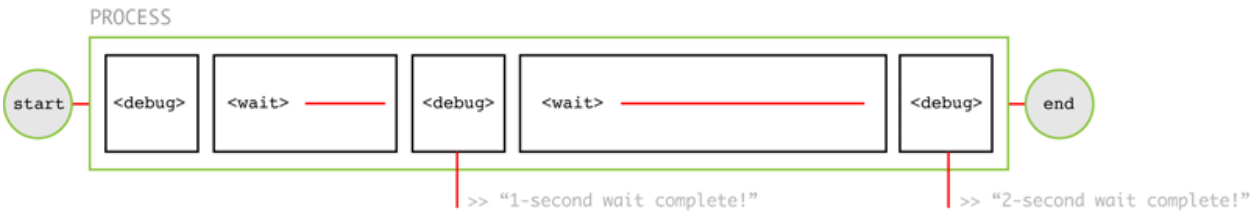

fig. 1: a single-threaded process.

In figure 1, the green box represents the process queue. The queue starts at the left, and steps through the sequence of commands in the queue until it completes on the right. So, this process is a linear pipeline.

However, not all Shepherd processes are linear. Lassie Shepherd includes the Lassie engine's first multi-threaded implementation, meaning that multiple process queues can

operate in tandem to perform asynchronous tasks. What does that mean? It means that multiple action sequences can be run simultaneously.

When getting into multi-threaded operations, you should first understand the concept of "trunk" and "branch" processes. A "trunk" is the engine's primary process thread that handles game actions. "Branch" processes are secondary threads operating in the background. The engine's trunk process controls major game flow like the character walking to a room object before interacting with it. The trunk process may be aborted or redefined when the player performs a new action that interrupts the current trunk. Branch processes will keep operating regardless of the trunk. So, let's take the previous example and split the <wait> commands into separate process threads to make the first example multi-threaded.

A multi-threaded process looks like this:

```
<debug enabled="1"/>
<process>
  <wait seconds="1"/>
  <debug echo="1-second wait complete!"/>
</process>
<process>
  <wait seconds="2"/>
  <debug echo="2-second wait complete!"/>
</process>
<debug echo="trunk complete!"/>
```

The above script expands upon the first process example. It is still just sending messages to the Lassie Player debugger window. However, the above script implements multiple processes which leads to the following behavior:

Trunk:

- enable the debugger window.
- create a new process (we'll call it branch #1)
- create a new process (we'll call it branch #2)
- send a message to the debugger window saying "trunk complete".

Branch #1

- wait 1 seconds.
- send a message to the debugger window.

Branch #2 (started immediately after branch #1)

- wait 2 seconds.
- send a message to the debugger window.

All told, that script plays out in sequence across three different processes over the course of two seconds (not three). To diagram the flow of that script would look like this:

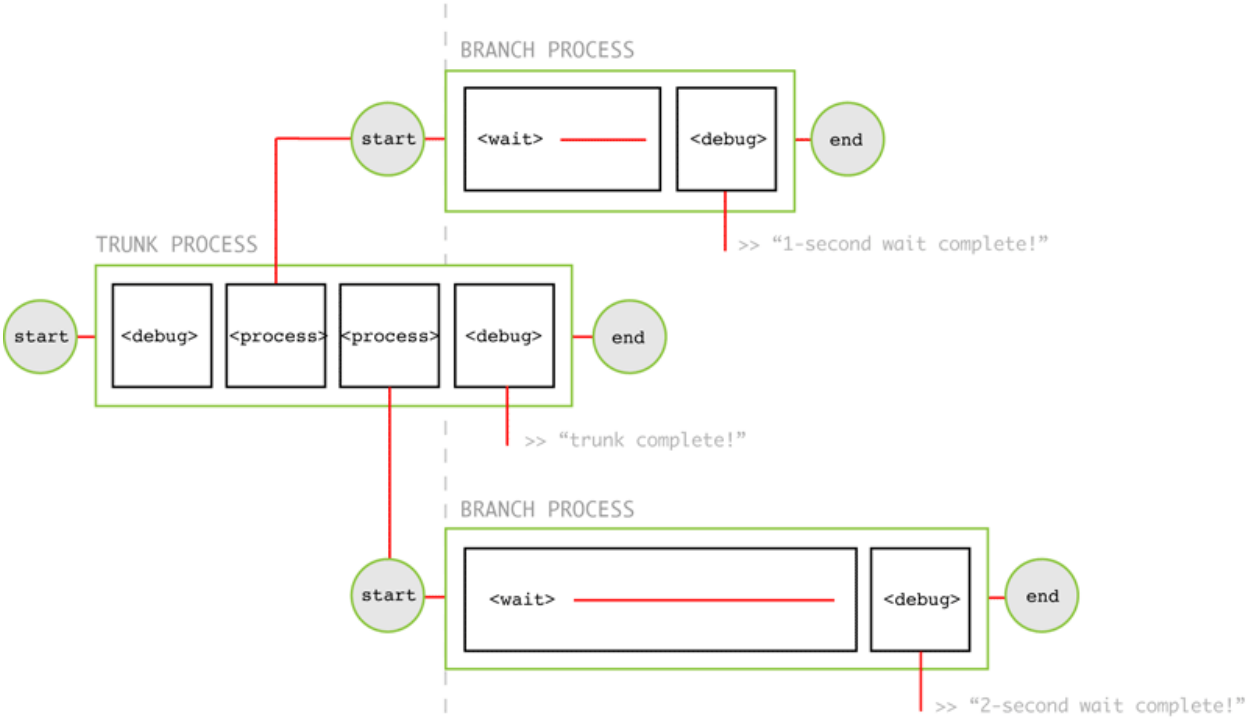

fig. 2: a multi-threaded process.

In figure 2, two new processes are spawned immediately after one another. Each <process> node's nested commands are passed off to branch processes rather than being performed in sequence along the trunk process. Therefore, the two processes effectively begin running their respective queues simultaneously. Now the two <wait> commands start running at the same time (rather than in sequence). Ultimately, all processes here are completed within two seconds of initiation rather than the three seconds it took to processes them in sequence. This is because the two branch processes ran in tandem to one another rather than running in sequence.

Normally, all branch processes are left alone and allowed to complete in their own time. The exception to this is when changing between game rooms using the <game> command. Since most processes are room-specific, the Lassie Player automatically aborts and destroys all existing branch processes while changing room layouts. All branches are eliminated after calling the current room's exitRoom script, leaving only the trunk process running to pull the engine through into the next room layout. To allow a branch process to remain active between multiple room layouts, you must specifically orphan the process upon creation (using the "orphan" parameter). Orphaned processes will be ignored by the Lassie Player, so will remain in existence until they complete themselves.

In closing – the <process> command is very powerful and extremely useful in a situation that requires it. However, it can also be a tricky command to use given that you, the developer, are taking over a decision-making role from the core Lassie Player application that may result in unpredictable behavior. When branching processes, be sure to thoroughly test every contingency surrounding a branch to make sure that you haven't introduced unwanted behavior. Generally speaking, if a situation does not absolutely require adding a branch process, then it's probably safer not to use one. However, that's not to say that you should avoid experimenting with processes! Indeed, you can introduce some extremely

unique gameplay scenarios using process threads that would not be possible otherwise.

Required Parameters

There are no required parameters for this command.

Optional Parameters

orphan

type: boolean
values: "0" (false) / "1" (true)

Specifies if the newly branched process should be orphaned by the Lassie Player. Orphaned processes are ignored by the Lassie Player and are allowed to exist until they complete themselves; otherwise, non-orphaned processes will be aborted upon changing rooms with the <game> command. By default this value is FALSE ("0"), meaning that processes will NOT be orphaned unless specifically instructed.

Example:

```
<process orphan="1">
   <!-- do branch actions -->
</process>
```

puppet

Availability: Lassie Player v1.0

The puppet command is used to call puppet-specific behaviors within the active room layout. This command can be used to mimic the game's verb selector and automatically call interactions assigned to a puppet layer. Also, the <puppet> command can be used to manually set the visual state (rest, move, speak) of a puppet. Note that the <puppet> command may only operate on puppet-class layers (configured within the Shepherd Editor), and can only target puppets within the game's active room layout.

Basic example:

```
<puppet target="door" callAction="1"/>
```

Required Parameters

target

type: string
value: "puppetId" (ID of puppet to target)

Specifies the ID of the puppet layer to target with this command. This ID must reference a valid puppet-class layer within the game's active room layout. In the event that a valid puppet target cannot be resolved, the <puppet> command will be aborted.

Examples:

```
<puppet target="door" callAction="1"/>
```

Optional Parameters

callAction

type: number
value: "X" (interaction index)

Specifies an action to call from the target puppet's interaction list. This value must be a valid number that falls within range of the puppet's defined action list (action indices are displayed within the puppet actions panel of the Shepherd Editor). Note that a puppet's actions list is zero-indexed; meaning that the first action is 0, the second action is 1, etc.

Examples:

```
<puppet target="door" callAction="0"/>
```

---

callItemAction

availability: Lassie Player v1.2
type: string
value: "itemId"

Specifies an item action to call from the target puppet's item interaction list. This value must be a valid item action defined in the puppet's current state.

Examples:

```
<puppet target="door" callItemAction="key"/>
```

---

visualState

type: string
values: "rest" / "move" / "speak"

Specifies a Lassie-native visual state to apply to the puppet. The Lassie Player has three native visual states that it manages for puppets, these are "rest" (a puppet's resting/ standing state), "move" (a puppet's moving/walking state), and "speak" (a puppet's speaking/talking state). Generally speaking, visual states are really only managed for puppets which behave as NPC's (non-player characters). This command may be useful for overriding the Lassie Player and taking manual control of visual state display management. Use in conjunction with the "turnView" <puppet> parameter.

Examples:

```
<puppet target="storekeeper" visualState="stand"/>
```

puppetTween

Availability: Lassie Player v1.0

The <puppetTween> command dynamically animates a puppet layer (the word "Tween" refers to "in between" frames generated between a start and end state). A puppet layer can by dynamically moved around a room following the walkable grid, or it can be rotated through its directional views to "turn" toward a specific target. This command is utilized by the Lassie Player to move the avatar around a room's walkable area grid during standard gameplay.

Basic example:

```
<puppetTween target="_avatar" type="moveThenTurn" to="layer:door"/>
```

Required Parameters

target

type: string
value: "puppetId" (ID of puppet layer to tween)

Specifies the ID of a puppet layer to tween. The target puppet MUST be present within the currently active game room, even if its position falls outside the framing of the current scoll.

Example:

```
<puppetTween target="_avatar" type="moveThenTurn" to="layer:door"/>
```

---

to

type: string
values: "point:0,0" (move to coords), or "position:id" (move to labeled position), or "layer:id" (move to layer's mapping point)

Specifies a destination to move the puppet to or turn the puppet toward. The "to" parameter consists of a type/value value pair, formatted as "targetType:targetValue". There are three possible type/value pairs:

- **point:x,y** - Specifies a set of room cordinates to tween the puppet to. The "x,y" value must be a set of numeric X and Y coordinates within the room layout's bounds. When the tween type is set to "move" with a point target, the puppet will

move to the target point. When the tween type is set to "turn" with a point target, the puppet will rotate to face the target point. A "moveThenTurn" tween will behave exactly like a "move" tween.

- **position:id** - Specifies a named position point within the room to tween the puppet to. The "id" value must reference a valid position ID within the current room layout. When the tween type is set to "move" with a position target, the puppet will move to the target position. When the tween type is set to "turn" with a position target, the puppet will rotate to face the target position. A "moveThenTurn" tween will behave exactly like a "move" tween.
- **layer:id** - Specifies a larget layer to tween the puppet to. The "id" value must reference a valid layer ID within the current room layout. When moving a puppet to a layer, the puppet will move to the layer's map point (a blue dot within the Shepherd Editor). When turning a puppet toward a layer, the puppet will rotate to face the layer's registration point (a white dot with crosshairs within the Shepherd Editor). A "moveThenTurn" tween will move the puppet to the layer's map point, then turn the puppet to face the layer's registration point.
- **view:1** - Specifies a directional turn view (number 1-8) to animate the puppet into. This target reference may ONLY be used when the tween type is set to "turn". If the tween type is set to "move" or "moveThenTurn", then you must use the optional "toView" parameter to specify a custom directional view for the puppet to turn toward after completing the motion sequence.

Example:

```
<puppetTween target="_avatar" type="move" to="point:500,350"/>
<puppetTween target="_avatar" type="move" to="position:window"/>
<puppetTween target="_avatar" type="moveThenTurn"
to="layer:doorway"/>
<puppetTween target="_avatar" type="turn" to="view:4"/>
```

---

type

type: string
values: "move" (move only), or "turn" (turn only), or "moveThenTurn" (move then turn to face the target)

Specifies the type of tween to perform on the puppet. There are three possible values:

- **move** : specifies that the puppet should move from its current position to the target.
- **turn** : specifies that the puppet should rotate through its turn views until it faces the target.
- **moveThenTurn** : specifies that the puppet should move to the target destination, and then rotate through its turn views to face the target.

Example:

```
<puppetTween target="_avatar" type="move" to="layer:door"/>
<puppetTween target="_avatar" type="turn" to="layer:door"/>
<puppetTween target="_avatar" type="moveThenTurn" to="layer:door"/>
```

Optional Parameters

confineToWalkarea

type: boolean
values: "1" (true), or "0" (false)

Specifies if a moving puppet following the walkable grid should have its target destination confined to within the safe grid area. When "confineToWalkarea" is enabled, the target "to" destination will be trapped using the grid's boxes and trapping nodes. Ultimately, a confined motion sequence will always end on a point that falls within a walkable box or on a trapping grid node. By default, this value is FALSE ("0"); which will allow the motion sequence to end at any point within the layout.

Example:

```
<puppetTween target="_avatar" type="move" to="layer:door"
followGrid="1" confineToWalkarea="1"/>
```

---

followGrid

type: boolean
values: "1" (true), or "0" (false)

Specifies if a moving puppet should follow the room's walkable grid. By default, this value is TRUE ("1"), even if the parameter is not specified. You must specifically disable "followGrid" if you do not want the puppet to follow grid behavior. When grid behavior is disabled, the puppet will animate in a straight line to the target point. This parameter only has an effect with a "move" or "moveThenTurn" type <puppetTween>. Grid behavior will not have an effect on animated puppet turns.

Example:

```
<puppetTween target="_avatar" type="move" to="layer:door"
followGrid="0"/>
```

---

toView

type: numver
values: "1-8" (directional view keys)

Specifies a directional view that the puppet should turn face at the end of a "moveThenTurn" tween sequence. When a "moveThenTurn" tween sequence targets a layer reference as the goal point, then the tweening puppet will automatically turn to face the layer's registration point at the end of the tween. However, this behavior does not apply to a point or position goal reference. When tweening to a point or position, the puppet will be left facing its final motion direction unless "toView" specifies a custom view to turn the puppet to.

Example:

```
<puppetTween target="_avatar" type="moveThenTurn"
to="point:500,650" toView="6"/>
```

---

waitForComplete

type: boolean
values: "1" (true), or "0" (false)

Specifies if the parent processes thread should be paused until after the puppet tween finishes. By default, this value is TRUE ("1"), even if the parameter is not specified. You must specifically disable "waitForComplete" if you want action processing to continue after calling a puppet tween.

Example:

```
<puppetTween target="_avatar" type="move" to="layer:door"
waitForComplete="0"/>
```

room

Availability: Lassie Player v1.0

The <room> dynamically configures settings of a game room.

Required Parameters

target

type: string
value: "roomId" / "_current"

Specifies the ID of the room to target, or "_current" to target the currently active game room.

Examples:

```
<room target="beach" grid="shoreline"/>
```

Optional Parameters

grid

type: string
value: "gridId"

Specifies a grid ID to activate as the room's walkable area grid. The specified ID must correspond to a grid that exists within the target room layout.

Unofficially – this property may also be set to a comma-separated list of grid ID's that exist within the target room layout. When multiple grids are activated, an operation is performed to join the grids into a united whole, where nodes of the two grids are joined to one another through overlapping box areas. This is extremely useful for expanding grid areas to include additional paths. However, this feature is listed as "unofficial" because it has not been fully tested. While the system has been implemented, the grid-joining algorithm may still be buggy. If you do test this feature, please report upon the outcome on the Lassie forums (http://lassie.10.forumer.com).

Examples:

```
<room target="beach" grid="shoreline"/>

<!-- unofficial feature... -->
<room target="beach" grid="shoreline,trail"/>
```

---

soundtrack1

type: string
value: "lib/libraryName.swf:TargetSoundClass"

Specifies...

Examples:

```
<room target="beach" soundtrack1="lib/beach.swf:BeachMusic"/>
```

---

soundtrack2

type: string
value: "lib/libraryName.swf:TargetSoundClass"

Specifies....

Examples:

```
<room target="beach" soundtrack2="lib/beach.swf:BeachFX"/>
```

scriptCall

Availability: Lassie Player v1.0

The <scriptCall> command accesses and runs a script from the global or local room script list. A script is accessed by ID, then its commands are stacked into the front of the active process queue. All of the script's commands will be run in sequence, then normal processing of the original queue that called the script command will resume.

Like the <method> command, a script may also be called with dynamically parsed parameters. *[Need contextual documentation. For temporary documentation, see "dynamically parsed parameters" section within the <method> command. The <scriptCall> command's parametrization works exactly the same as the <method> command, except that default parameter values may not be defined for <scriptCall>]*

Basic example:

```
<scriptCall room="openDoorSequence"/>
```

Required Parameters

ONE *of the following parameters must be specified.*

---

global

type: string
value: "globalScriptId"

Specifies the Id of a script within the global script list to retrieve and run. This ID must reference a valid script that was created within the Shepherd Editor's "Global Script" panel.

Examples:

```
<scriptCall global="gameSequence"/>
```

---

room

type: string
value: "roomScriptId"

Specifies the Id of a script within the active game room's script list to retrieve and run. This ID must reference a valid script that was created within the Shepherd Editor's "Room Script" panel. Additionally, this Id must reference a script defined within the currently active game

room (room layout overlays do not count).

Examples:

```
<scriptCall room="openDoorSequence"/>
```

scriptParse

Availability: Lassie Player v1.0

The scriptParse command parses fields into a text-representation of an XML command block, then sends the processed XML off for immediate processing. This method can be used to parse real-time game values into an XML command; which allows the command to be parametrized with current game values.

The scriptParse command takes no parameters. It only wraps a text representation of raw XML which is sent to the logic API for processing. The contents of the scriptParse command should be placed into a CDATA block, which excludes the contents from formal XML parsing. A CDATA block is an XML-native structure, and is written like so:

```
<scriptParse><![CDATA[ Unformatted XML content goes here!
]]></scriptParse>
```

Place Lassie XML commands to be parsed into the CDATA block. Because they are stored within CDATA, it does not matter if the XML commands are malformed. You can use the logic API to parse any formatting needed into the command. To identify fields that should be parsed and replaced by the logic API, use Lassie's logic field open and close symbols:

```
%* logic API expression here *%
```

In full context, the scriptParse method looks like this:

```
<scriptParse>
    <![CDATA[<layerTween targets="_avatar"
to="%*{math:100+100+100}*%,0" seconds="2" ease="elastic:easeOut"
waitForComplete="1"/>]]>
</scriptParse>
```

In the above example, the logic API is being used to parse a custom X value into a layerTween command. All parsing variables and methods of the logic API are available while parsing the text within logic field delimiters.

scroll

Availability: Lassie Player v1.0

The scroll command shifts the current game room's viewport with an automatic scroll to specific coordinates.

Basic example:

```
<scroll to="50,100" seconds="2" ease="elastic:easeInOut"/>
```

Required Parameters

ONE *of the following two parameter must be specified.*

---

to

type: coordinates
value: "x,y"

Specifies a set of target upper-left coordinates to scroll the room to, formatted as "x,y". You may leave one of the coordinates specified as a literal value of "x" or "y" to leave scrolling unchanged along that asix. For example, "0,y" would scroll the room to the extreme left while leaving the viewport's Y position unchanged. Likewise, "x,0" would scroll the room to the extreme top, while leaving the viewport's X position unchanged. All target coordinates will be validated and limit scrolling to within the bounds of the room's background layer.

Examples:

```
<scroll to="50,100" seconds="2"/>
<scroll to="50,y" seconds="2"/>
<scroll to="x,100" seconds="2"/>
```

---

by

type: coordinates
value: "x,y"

Specifies a set of coordinates, formatted as "x,y", to offset the current viewport by. For example, scrolling by "-50,100" would scroll the viewport 50 pixels left and 100 pixels down from the current viewport position. You may leave one of the coordinates specified as a

literal value of "x" or "y" to leave scrolling unchanged along that asix. For example, "-50,y" would scroll the room 50 pixels to the left while leaving the viewport's Y position unchanged. Likewise, "x,-50" would scroll the room 50 pixels up while leaving the viewport's X position unchanged. All offset coordinates will be validated and limit scrolling to within the bounds of the room's background layer.

Examples:

```
<scroll by="-50,100" seconds="2"/>
<scroll by="-50,y" seconds="2"/>
<scroll by="x,-50" seconds="2"/>
```

Optional Parameters

seconds

type: number
value: "X"

Specifies the number of seconds over which to animate the scroll. By default, this value is 2 seconds.

Examples:

```
<scroll to="0,0" seconds="2.5"/>
```

---

ease

type: string
value: "type:method"

Specifies an easing operation to animation the scroll with. Easing is specified with two components, a type and a method, formatted a "type:method". There are numerous types of easing, each with a unique visual personality. An ease type like "strong" will create a very bold move, while another like "elastic" will create playful movement. The best way to get familiar with easing is just to experiment with and preview the different types. All types of easing have three basic methods: ease-in, ease-out, and ease in-out. Valid values for the "type" and "method" components are as follows:

Ease type values

- "back"
- "bounce"

- "circ"
- "cubic"
- "elastic"
- "expo"
- "linear"
- "quad"
- "quart"
- "quint"
- "sine"
- "strong"

Ease method values

- "easeIn"
- "easeOut"
- "easeInOut"

Example:

```
<scroll to="0,0" seconds="2" ease="bounce:easeOut"/>
```

---

waitForComplete

type: boolean
values: "1" (true) or "0" (false)

Specifies whether the process queue should wait for the scroll animation to complete before processing subsequent commands. The default is TRUE, meaning that the queue will wait for the scroll sequence to complete unless explicitly told not to.

Examples:

```
<scroll to="0,0" seconds="2" waitForComplete="1"/>
<scroll to="0,0" seconds="2" waitForComplete="0"/>
```

skip

Availability: Lassie Player v2.0
The skip command was overhauled for Lassie Player v.2.

The <skip> command configures the system SKIP key to skip over a game sequence. The outcome of sequence skipping is "automatic". The Lassie Player runs through the current queue of actions contained within the <skip> node and immediately process all scripts. This is done by assigning a shortened version of a sequence script to trigger when the SKIP key is pressed. The Lassie Player system SKIP key is defined within the "system.xml" configuration file. In addition, the skip control automatically enables the on-screen "skip" UI button (included within the UI document).

```
<skip>
    <!--do stuff-->
</skip>
```

Optional Parameters

action

values = ["call" / "reset"]

The "action" attribute specifies an action to perform within the <skip> framework. Actions are:

- **"call"**: Calls the currently configured skip script.
- **"reset"**: Resets any configured skip action by clearing the skip action and disabling the SKIP key.

sound

Availability: Lassie Player v1.0

The sound command plays an embedded library sound. This command's intended use is for triggering sound effects during game play. This command is NOT designed to manage background music within a room (there is a soundtrack feature built specifically into room layouts for that). Use this command within a script to trigger a short, action-specific sound during game play.

Basic example:

```
<sound effect="lib/room01.swf:MySoundClass" waitForComplete="1"/>
```

Required Parameters

One or more optional parameters are required.

Optional Parameters

soundtrack1

type: media address
value: "file_path.swf:SoundClass"

Sends a sound asset to the first soundtrack channel for continuous looping playback. The first soundtrack channel is generally intended for an ambient music track. Note that changing a soundtrack channel with the `sound` command will not modify the sound configuration of the current room, so the room layout will continue to load in the future with it's default sound configuration. To permanently change the soundtrack of a room, use the `room` command (which currently not available).

Example:

```
<sound soundtrack1="lib/room01.swf:MyFxTrack"/>
```

---

soundtrack2

type: media address
value: "file_path.swf:SoundClass"

Sends a sound asset to the second soundtrack channel for continuous looping playback. The second soundtrack channel is generally intended for an ambient sound effects track. Note that changing a soundtrack channel with the `sound` commandwill not modify the sound configuration of the current room, so the room layout will continue to load in the future with it's default sound configuration. To permanently change the soundtrack of a room, use the `room` command (which currently not available).

Example:

```
<sound soundtrack2="lib/room01.swf:MyFxTrack"/>
```

---

effect

type: media address
value: "file_path.swf:SoundClass"

Sends a sound asset to the sound system for immediate, one-time only playback. The media address is formatted as "file_path:SoundClass"; where "file_path" is the full path of a loaded SWF library, and "SoundClass" is the name of a registered sound class within that library. Use this command to generate sound effects.

Example:

```
<sound effect="lib/room01.swf:MySoundClass" waitForComplete="1"/>
```

---

waitForComplete

type: boolean
values: "1" (true) or "0" (false)

This parameter is used in conjunction with the ***effect*** parameter. This parameter specifies if the process queue should suspend scripts until the designated effect sound finishes playing. If true (1), no additional queued scripts will be run until the sound effect finishes playing. If false (0), scripts will continue running immediately after starting the sound effect.

Examples:

```
<sound media="lib/room01.swf:MySoundClass" waitForComplete="1"/>
```

---

playback

type: string
values: "play" / "stop" / "toggle" / "stopAll"

Alters the status of global sound playback within the Lassie Player. You can use this parameter to globally play and stop all sounds configured within the Lassie Player's sound system. Note that this command will NOT change the status of sounds playing within individual MovieClips on stage. It is therefor suggested that you avoid embedding sounds into animations whevener possible so that sound volume and playback can remain under control of the Lassie Player at all times. This parameter accepts three values:

- **play**: Plays all sounds that are managed by the Lassie Player.
- **stop**: Stops all sounds that are managed by the Lassie Player.
- **toggle**: Toggles the playback status of sound that are managed by the Lassie Player. If sounds are currently stopped, then playback is resumed, and vice-versa.
- **stopAll**: Stops all sounds within the entire Flash runtime, including Lassie Player sounds and Flash media timeline sounds. This is done by triggering `SoundMixer.stopAll()` within ActionScript 3 (this command was "stopAllSounds()" in AS2). *Note: "stopAll" was added in Lassie Player v1.1.*

Example:

```
<sound playback="stop"/>
```

tree

Availability: Lassie Player v1.0

The tree command manages dialogue trees. Dialogue trees are configured on a room-by-room basis within the Shepherd editor, then may be loaded during game play. In addition, individual topics within a dialogue tree may be enabled or disabled at any time during the game runtime.

Basic examples:

```
<tree load="npcConversation"/>
<tree enableTopic="room:tree:topic:1"/>
```

Required Parameters

One or more of the optional parameter must be specified.

Optional Parameters

load

type: string
value: "treeId"

Loads a conversation tree into the Lassie Player's tree menu interface.

Example:

```
<tree load="npcTalk"/>
```

---

enableTopic

type: string
value: "roomId:treeId:topicId:0/1"

Specifies the enabled status of a topic within a conversation tree. This field is written with four data elements, separated by colons. The fields are:

[roomId] : [treeId] : [topicId] : [enabled status (1/0)]

Example:

```
<tree enableTopic="_current:dogTalk:someTopic:1"/>
<tree enableTopic="_current:dogTalk:someTopic:0"/>
```

---

omitTopic

type: string
value: "roomId:treeId:topicId:0/1"

Specifies the omission status of a topic within a conversation tree. When a topic has been omitted, it will not display within a dialogue tree menu even if its "enabled" status is set as TRUE. Omission is useful for permanently disabling a topic once it's associated game objective is completed, and never worrying that the topic may be re-enabled through another means. This field is written with four data elements, separated by colons. By default, all topic omissions are set to FALSE. You must explicitly tell a topic line to be omitted by setting its "omitTopic" status to true. The fields are:

[roomId] : [treeId] : [topicId] : [enabled status (1/0)]

Example:

```
<tree omitTopic="_current:dogTalk:someTopic:1"/>
<tree omitTopic="beach:surfer:someTopic:1"/>
```

ui

Availability: Lassie Player v1.0

The <ui> command loads a room layout into the player as a static game overlay. The room layout is rendered normally, with the exception that the room's background layer is omitted. The loaded room layout exists permanently on screen over the top of the interactive game room until the layout is specifically unloaded. This feature can be used to load custom UI graphics into your game display.

Required Parameters

ONE *(and only one) of the following parameter must be specified:*

---

load

type: string
value: "roomId"

Loads a room layout into the game UI as a global overlay. Once a room layout has been loaded into the UI, that layout can be targeted with transition properties and then hidden/ revealed like other Lassie-native UI layers. A loaded layout can thereafter be referenced with the <ui> command using its room Id.

While a room layout that is loaded into the UI will render layout, the room will not behave like a full fledged game room. The background layer of a UI layout will not be rendered, and the layout will not have any script support. Basically, a UI layout is not intended to behave like a second game room, it is simply designed as a mechanism to allow global layout to be added into the engine UI.

The <ui> command will always wait for the targeted layout to load before proceeding with additional scripts, so it is suggested that you incorporate a "load" command into a game transition where load latency would be expected. Once a new layout has been loaded, that UI layer become the target for the remainder of the <ui> command, at which time the new layer can have its transition properties set and can be show or hidden. By default, all newly loaded UI layout will be displayed.

Example:

```
<ui load="beach"/>
```

---

target

type: string
value: "uiLayerId"

Specifies a UI layer to target with this command. Lassie-native UI layers may be targeted, or custom layouts loaded using the "load" command. This target reference will indicate the UILayer to perform all optional command behaviors upon.

Example:

```
<ui target="_inventory"/>
<ui target="beach"/>
```

Optional Parameters

display

type: string
values: "show" / "hide" / "toggle"

Sets the display status the the UI layer. A layer can be show, hidden, or toggled. When toggled, the layer will always change to the opposite display state from what it current is in.

Example:

```
<ui target="room_overlay" display="show"/>
```

[display] tween

type: boolean
values: "1" (true) / "0" (false)

Use in conjunction with the "display" command. This value specifies whether the UI layer should be animated as it shows/hides, or if it should just cut to the new display state. This value is TRUE by default, meaning that a transition will animate unless specifically instructed not to. All <ui> display transitions will tween using the durations specified by "showSeconds" and "hideSeconds".

Example:

```
<ui target="room_overlay" display="show" tween="0"/>
```

hideSeconds

type: number
value: whole number or decimal

Specifies the duration (in seconds) of the target UI layer's hide animation. Setting this value to 0 will entirely eliminate the hide animation.

Example:

```
<ui load="room_overlay" hideSeconds="0.5"/>
```

showSeconds

type: number
value: whole number or decimal

Specifies the duration (in seconds) of the target UI layer's reveal animation. Setting this value to 0 will entirely eliminate the reveal animation.

Example:

```
<ui target="room_overlay" showSeconds="0.5" display="show"/>
```

unload

type: string
value: "uiLayerId"

Unloads a room layout from the UI that was added using the "load" command. An unloaded room will immediately be removed from display and destroyed. Once a room has been unloaded from the display, it can no longer be targeted with the <ui> command until re-loaded.

Example:

```
<ui unload="room_overlay"/>
```

wait

Availability: Lassie Player v1.0

The <wait> command creates a delay during script processing. If you'd like a time delay between two command calls, just add a <wait> command between them. A wait duration may be timed in seconds, or may wait an undefined period of time until a stage event occurs.

Basic example:

```
<wait seconds="1"/>
<wait stageEvent="gameSequenceComplete"/>
```

Required Parameters

ONE *(and only one) optional parameter is required.*

Optional Parameters

seconds

type: number
value: "X"

Specifies a duration (in seconds) to pause before running additional scripts queued within the current process.

Examples:

```
<wait seconds="1.5"/>
<wait seconds="2"/>
```

---

stageEvent

type: string
value: "eventType"

Registers an event listener on the main Flash stage for the specified event type. The <wait> command will then wait for the specified event to be called on the stage. A stage event can

be called using Lassie's <flash stageEvent> command; OR – custom Flash media can call stage events by dispatching through stage or bubbling events up through the display tree.

Examples:

```
<wait stageEvent="myAnimationComplete"/>
```

Controlling Game Flow with Event Scripts

The Lassie Player has several built-in script calls which are triggered in response to major events within the Lassie Player. Major events include start up, creating a new game, loading a game, etc. By creating global scripts with IDs that match these event names, you may define custom scripts which control the flow of a game within the Lassie Player. Event calls include:

_onStartUp

Called immediately after the Lassie Player application has finished loading. Hook into this event to define how your game launches (opening the game menu, going straight into a game room, etc). If you do not define a custom startup script, then the Lassie Player will default to opening the game menu. If you DO define a custom script for this event, then the Lassie Player takes no further action.

_onStartGame

Called each time a new game is started – either by means of a "new" or "load" request within the game menu. This call is useful for establishing constant game behavior such as keyboard behaviors, or resetting configuration from a previously loaded game. If no corresponding script is defined, then no substitute action is taken by the Lassie Player.

_onNewGame

Called upon creating a new game from within the game menu. Use this event to define where a new game starts (starting room, actor, etc). If you do not define a custom script for this event, then the Lassie Player will default to loading the starting actor and starting game room that were defined within the Shepherd Editor. If you DO define a custom script for this event, then the Lassie Player takes no further action.

_onLoadGame

Called upon loading a new game from within the game menu, but before the Lassie Player sets the loaded game's configuration. The _onLoadGame script will run (if it exists), and then the Lassie Player will initialize the loaded actor and room configuration.

Again – to hook a global script into one of these events, just create a global script with an ID that matches the event name. Defining global scripts which hook into Lassie Player events is completely optional. There will be no negative result if you choose not to define an event script.

Calling "Animation Complete"

There are several time-based commands within the Lassie Player's script API that provide a "waitForComplete" option. This option instructs the command to wait for its resulting sequence (fade to black, sound playback, etc) to play out before proceeding on to additional scripts waiting in the queue. In most cases, the Lassie Player is able to natively monitor the progress of a time-based sequence, so you don't need to do anything but specify as "waitForComplete" as true.

However, in the case of playing timeline animations using the "layerClip" command, the Lassie Player has no way of (reliably) determing the length of the animation, so has no way of automatically waiting for the timeline animation to complete. Therefore, when enabling "waitForComplete" in conjunction with a timeline animation's playback, you'll need to build an event into your timeline animation that specifically tells the Lassie Player when the animation has completed.

Signaling the completion of a timeline animation is very easy to call. It only requires two lines of ActionScript placed at the end of your animation's timeline. To call the completion of an animation, do the following:

1) Make sure your FLA can access Lassie's script resources ("com").
Your FLA document must have access to the provided Lassie script library ("com") so that is can access one of the provided Lassie ActionScript files. For your FLA to access the Lassie script library, it just needs to be saved (emphasis: SAVED) into the same folder that contains the Lassie "com" resource folder. You should already be working within this directory given that all of your Flash assets must be contained within a Lassie media library, which also requires access to Lassie's "com" resources. For more information about media libraries, see the Media Library tutorial.

2) Import and call the "ANIMATION_COMPLETE" event.
Once your FLA is saved (emphasis: SAVED) into the directory containing Lassie's "com" resource folder, place the following ActionScript on a keyframe at the end of your Flash animation timelines:

```
import com.lassie.events.LassieEvent;
dispatchEvent( new LassieEvent(LassieEvent.ANIMATION_COMPLETE) );
```

That should be it! Once the above script is installed on your animation's timeline, then the Lassie Player will be able to listen for the completion of that animation and proceed accordingly. Note that if you instruct the Lassie Player to wait for an animation to complete but do not add in this completion event into the animation's timeline, then the Lassie Player will wait indefinitely for the event to occur, and your game will appear to have "hung". This is not a bug in the Lassie Player; it is simply a configuration error within your media.

The only scenario to watch out for when calling animation complete is when you also want the MovieClip change its own timeline frame at the end of an animation. Consider the

following INCORRECT script:

```
// For demonstration only. Do NOT use this script, see proper
implementation below...
import com.lassie.events.LassieEvent;
stop();
MovieClip(parent).gotoAndStop("rest");
dispatchEvent( new LassieEvent(LassieEvent.ANIMATION_COMPLETE) );
```

In Flash 9, the above script would behave as you would expect: at the end of an animation, the MovieClip stops playback, the parent timeline directs itself to a new visual state, and then animation-complete is called. Unfortunately, this process no longer works in Flash 10 and greater. It seems Adobe revised timeline behavior so that timeline scripts now abort processing when their frame is exited. So, by redirecting the parent timeline in the above example, the animation-complete dispatch is never performed and your animation will effectively hang. To perform this behavior in Flash 10 and later, you'll need to set up a function closure to disassociate actions from the MovieClip timeline. The following script will work to both set a new timeline frame and call animation-complete:

```
// Correct implementation of changing timeline frame and calling
animation-complete
import com.lassie.events.LassieEvent;
stop();
var $scope:MovieClip = MovieClip(parent);
(function(){
    $scope.gotoAndStop("rest");
    $scope.dispatchEvent( new
LassieEvent(LassieEvent.ANIMATION_COMPLETE) );
  $scope = null;
})();
```

Calling XML Scripts from within Flash Animations The most common way to call script within the Lassie Player is to compose all your scripts within predefined locations around the Shepherd editor. These scripts will be parsed and processed by the Lassie Player's native process. However, it is not uncommon to need to fire off a Lassie Player script at a specific point within a Flash animation placed into your game.

These custom script calls are very easy to perform; they just require an XML script to be sent to the Lassie Player using an "Event". If you're unfamiliar with the concept of events, just think of an event as a vehicle for delivering messages between Flash objects. The basic process for calling an XML script from a Flash timeline looks like this:

```
// import the LassieEvent class from the provided Lassie script
library.
import com.lassie.events.LassieEvent;

// Define the XML script.
var $xmlScript:XML = <script><inventory add="apple"/></script>;

// Send the XML script to the Lassie Player using an event.
dispatchEvent( new LassieEvent(LassieEvent.SCRIPT, $xmlScript) );
```

In the above example, note the formatting of the XML script. The <inventory> script is contained within an opening and closing <script> node. This is extremely important because all Lassie XML commands need to be contained within a single XML node. So, a list of Lassie XML commands would look like this:

```
var $xmlScript:XML = <script><inventory add="apple"/><inventory
remove="banana"/></script>;
```

Also note that the list of commands are formatted on a single line. If you try to compose XML across multiple lines within Flash's script window, then the Flash player will throw and error when you compile your Flash movie... this occurs because XML with line breaks causes an ActionScript parsing error. If you have a long set of XML commands that you would like to compose across multiple lines of code for human legibility, you must format your script call as follows:

```
import com.lassie.events.LassieEvent;

var $xml:String = '<script>';
$xml += '<layerClip play="reach_out" waitForComplete="1"/>';
$xml += '<inventory add="apple"/>';
$xml += '<layerClip play="reach_in" waitForComplete="1"/>';
$xml += '</script>';
```

```
dispatchEvent( new LassieEvent(LassieEvent.SCRIPT, new XML($xml))
);
```

In the above example, the XML script is composed as basic text with each line of XML being appended using a separate line of code. The script is then converted to true XML when it is fed into the LassieEvent. This trick is very handy for keeping your XML scripts easy to manage within the Flash Actions window; however, it does add a large margin of potential for syntax error that you will NOT be warned about while publishing your Flash movie. So, be sure to double check your own syntax before publishing your Flash movie!