Autodesk®
Scaleform®

Scaleform CLIK User Guide

This document contains detailed implementation instructions for the Scaleform CLIK framework and the prebuilt components provided with it.

Authors: Prasad Silva, Matthew Doyle

Version: 2.0
Last Edited: August 19, 2010

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.3

LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS.

How to Contact Autodesk Scaleform:

Document	Scaleform 4.3 CLIK User Guide
Address	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
Website	www.scaleform.com
Email	info@scaleform.com
Direct	(301) 446-3200
Fax	(301) 446-3199

Table of Contents

1 Introduction

This document provides a detailed implementation guide for the Scaleform® Common Lightweight Interface Kit (CLIK™) framework and components. Before diving too deep into the CLIK User Guide, developers are encouraged to go through Getting Started with CLIK and Getting Started with CLIK Buttons. These two introductory guides walk through the steps needed to get Scaleform CLIK up and running, introduce the basic core concepts and provide a detailed tutorial on creating and skinning CLIK components. However, more advanced users may prefer to dive straight into this document or reference it in tandem while studying the introductory documents.

1.1 Overview

Scaleform CLIK is a set of ActionScript™ 2.0 (AS2) User Interface (UI) elements, libraries and workflow enhancement tools to enable Scaleform 4.3 users to quickly implement rich and efficient interfaces for console, PC, and mobile applications. The main goals of the framework were to make it lightweight (in terms of memory and CPU usage), easily skinnable and highly customizable. In addition to a base framework of UI core classes and systems, CLIK ships with over 15 prebuilt common interface elements (e.g., buttons, sliders, text inputs) that will help developers rapidly create and iterate user interface screens.

1.1.1. What's included in Scaleform CLIK

Components: Simple extensible prebuilt UI controls

Button	Slider
ButtonBar	StatusIndicator
CheckBox	ProgressBar
RadioButton	UILoader
Label	ScrollingList
TextInput	TileList
TextArea	DropdownMenu
ScrollIndicator	Dialog
ScrollBar	NumericStepper

Classes: Core system APIs

InputManager	Locale
FocusManager	Tween
DragManager	DataProvider
PopUpManager	IDataProvider
EventDispatcher	IList
Delegate	IListItemRenderer
Constraints	GameDataProvider

Documentation and Example Files:

Getting Started with CLIK
Getting Started with CLIK Buttons
CLIK API Reference
CLIK User Guide
CLIK Flash Samples
CLIK Video Tutorials

1.2 Understanding Components

Before getting started, it is important that developers understand what exactly a Flash component is. Flash ships with a series of default interface creation tools and building blocks called components. However, in this document, "components" refer to the prebuilt components created using the Scaleform CLIK framework, which was developed by Scaleform in collaboration with the world renowned gskinner.com team.

gskinner.com is led by Grant Skinner, who was commissioned by Adobe to create the components for Flash Creative Suite® 4 (CS4), and is known as one of the leading Flash teams in the world. For more information on gskinner.com, visit http://gskinner.com/blog.

To gain a better understanding of what the prebuilt CLIK Components are, open up the default CLIK file palette in Flash studio:

C:\Program Files\Scaleform\GFx 4.3\Resources\AS2\CLIK\components\CLIK_Components.fla

1.3 UI Considerations

The first step to creating a good UI is to plan out the concept on paper or with a visual diagram editor such as Microsoft Visio® . The second step is to begin prototyping the UI in Flash, in order to work out all of the

interface options and flow before committing to a specific graphic design. Scaleform CLIK is specifically designed to allow users to rapidly prototype and then iterate to completion.

There are many different ways to structure a front end UI. In Flash, the concept of pages doesn't exist, such as is the case in Visio or other flowchart programs. Instead, key frames and movie clips take their place. If you have all of your pages in the same Flash file, the file can take up more memory, but may be faster and easier to transition between pages. If each page is in a separate file, overall memory usage may be lower, but load times may be longer. Having each page as a separate file also has the advantage of allowing multiple designers and artists to simultaneously work on the same interface. In addition to the technical and design requirements, make sure to consider the workflow when determining the structure of your UI projects.

Unlike traditional desktop or web applications, it is important to understand that there are many different ways to create rich multimedia game interfaces. Every engine, and even several platforms, may have slightly different approaches that could be better—or worse—to utilize. For instance, consider putting a multipage interface into a single Flash file. This can make it easier to manage, and make transitions easier to create, but it can also increase memory consumption which can have a negative impact on older consoles or mobiles. If everything is done in a single Flash file, you can't have multiple designers simultaneously working on different parts of the interface. If the project calls for a complex interface, has a large design team, or has other specific technical or design requirements, it may be better to put each page of the UI into a different Flash file. In many cases both solutions will work, but one may offer significant advantages over the other for the project. For instance, screens may need to be loaded and unloaded on demand or dynamically updated and streamed over the network. The bottom line is that asset management for the UI should always be well thought through, from the logical layout of the UI to implementation workflow to performance considerations.

While Scaleform highly recommends developers use Flash and ActionScript for the majority of the UI behavior, there's no perfect answer, and some teams may prefer to use the application engine's scripting language (such as Lua or UnrealScript) to do the majority of the heavy lifting. In these cases, Flash should be used primarily as an animation presentation tool with only a minor amount of ActionScript and interactivity within the Flash file itself.

It is important to understand the technical, design, and workflow requirements early on, then continue evaluating the overall approach throughout the process, particularly before getting too far into the project, to ensure a smooth and successful result.

2 The Prebuilt Components

At first glance, Scaleform CLIK appears to be a basic set of prebuilt UI components, but the true intent of CLIK is to provide a framework for creating richer components and interfaces. Developers are free—and to some extent expected—to create custom components to suit their needs and to build upon the CLIK framework.

The prebuilt CLIK components provide standard UI functionality that ranges from basic buttons and checkboxes, to complex composite components such as list boxes, drop down menus and modal dialog boxes. Developers can easily extend these standard components to add more features or simply use them as a reference when creating a custom component from scratch.

The following sections describe each prebuilt component in detail. They are grouped by complexity and functionality. Each component is described using the subsections listed here.

- ***User interaction***: How a user can interact with the component.
- ***Component setup***: Elements required when constructing the component in the Flash authoring environment.
- ***States***: Different visual states (keyframes) required by the component to function.
- ***Inspectable properties***: Properties exposed in the Flash authoring environment to easily configure certain aspects of a component without the use of code.
- ***Events***: List of events fired by the component that can be listened to by other objects in the UI.
- ***Tips and tricks***: Helpful code samples for accomplishing various tasks related to the component in question.

2.1 Basic Button and Text Types

The basic types consist of the Button, CheckBox, Label, TextInput and TextArea components. The Button forms the backbone of most user interfaces, and the CheckBox derives its functionality from the Button. The Label is a static label type, while the TextInput and TextArea represent single line and multiline textField types.

Figure 1: Button examples from *Free Realms*.

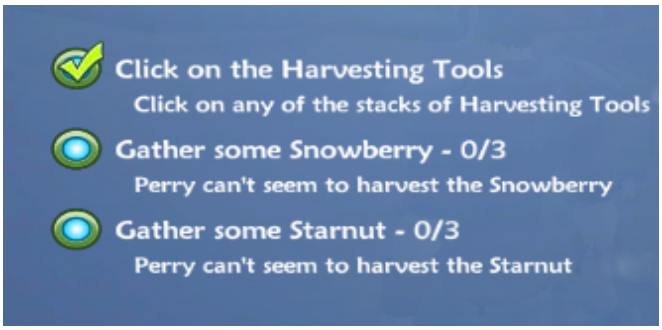

Figure 2: Check box examples from *Free Realms*.

Figure 3: Label example from *Free Realms*.

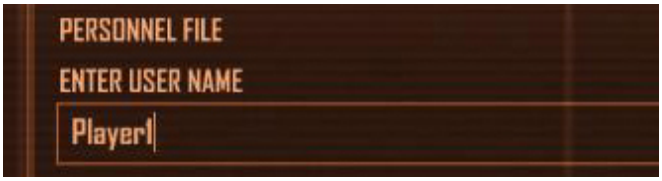

Figure 4: Text input example from Crysis Warhead.

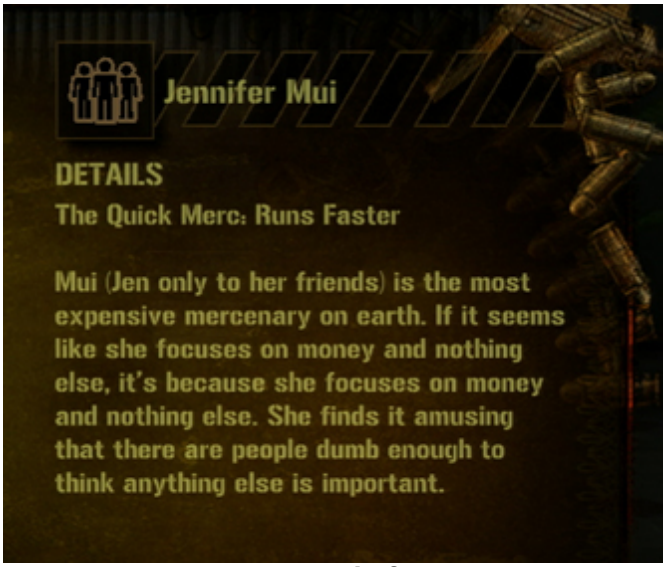

Figure 5: Text area example from *Mercenaries 2*.

2.1.1 Button

Figure 6: Unskinned button.

Buttons are the foundation component of the CLIK framework and are used anywhere a clickable interface control is required. The default Button class (gfx.controls.Button) supports a textField to display a label, and states to visually indicate user interaction. Buttons can be used on their own, or as part of a composite component, such as ScrollBar arrows or the Slider thumb. Most interactive components that are click-activated compose or extend Button.

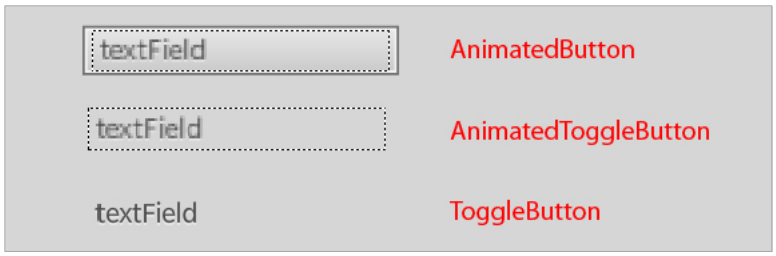

Figure 7: AnimatedButton, AnimatedToggleButton, and ToggleButton

The CLIK Button is a general purpose button component, which supports mouse interaction, keyboard interaction, states and other functionality that allow it to be used in a multitude of user interfaces. It also supports toggle capability, as well as animated states. The ToggleButton, AnimatedButton and AnimatedToggleButton provided in the Button.fla component source file all use the same base component class.

2.1.1.1 User interaction

The Button component can be pressed using the mouse or any equivalent controller. It can also be pressed via the keyboard when it has focus.

In general, only a focused component receives keyboard events. There are many ways to set the focus on a component. Several methods are described in the Programming Details chapter of this document. Most CLIK components also receive focus when interacted with, especially when the left mouse button or equivalent control is pressed (clicked) over them. The (Tab) and (Shift+Tab) keys (or equivalent navigation controls) move focus throughout the displayed focusable components. This is the same behavior you find in most desktop

applications and websites. Note that the focus used by non-CLIK elements is used by CLIK components. This means that a Flash developer is able to mix and match CLIK elements and non-CLIK elements in the scene and have the focus work as intended, especially when using (Tab) and (Shift+Tab) keys to move the focus around the scene.

By default the Button responds to the (Enter) key and the spacebar. Rolling the mouse cursor over and out of the Button also affects the component, as well as dragging the mouse cursor in and out of it.

Developers can easily map gamepad controls to keyboard and mouse controls. For example, the (Enter) key is typically mapped to the (A) or (X) button on a Xbox360 or PS3 console controller respectively. This mapping allows UIs built with CLIK to work in a wide variety of platforms.

2.1.1.2 Component setup

A MovieClip that uses the CLIK Button class must have the following named subelements. Optional elements are noted accordingly.

- ***textField***: (optional) TextField type. The button's label.
- ***focusIndicator***: (optional) MovieClip type. A separate MovieClip used to display the focused state. If it exists, this MovieClip must have two named frames: "show" and "hide". By default the *over* state is used to denote a focused Button component. However in a few cases, this behavior can be too restrictive as artists may prefer to separate the focused state and the mouse over state. The focusIndicator subelement is useful in such cases. Adding this subelement frees the Button states from being affected by its focused state. For more information on the component states, see the next section.

2.1.1.3 States

The CLIK Button component supports different states based on user interaction. These states include:

- an ***up*** or default state;
- an ***over*** state when the mouse cursor is over the component, or when it is focused;
- a ***down*** state when the button is pressed;
- a ***disabled*** state.

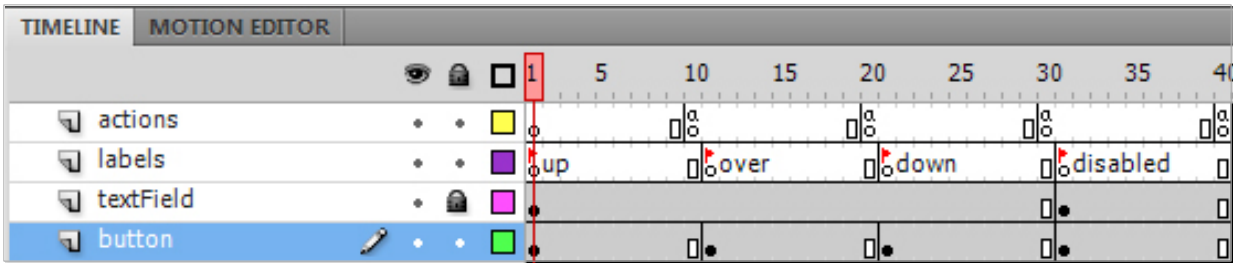

Figure 8: Button timeline.

These states are represented as keyframes in the Flash timeline, and are the minimal set of keyframes required for the Button component to operate correctly. There are other states that extend the capabilities of the component to support complex user interactions and animated transitions, and this information is provided in the Getting Started with CLIK Buttons document.

2.1.1.4 Inspectable properties

The important properties of a component can be configured via the Flash IDE's Component Inspector panel or the Parameters tab. To open this panel in CS4, select the Window drop down menu on the top toolbar and enable the Component Inspector window by clicking on it, or press (Shift+F7) on the keyboard. This will open the Component Inspector Panel. These are called "inspectable properties." This provides a convenient way for artists and designers unfamiliar with AS programming to configure a component's behavior and functionality.

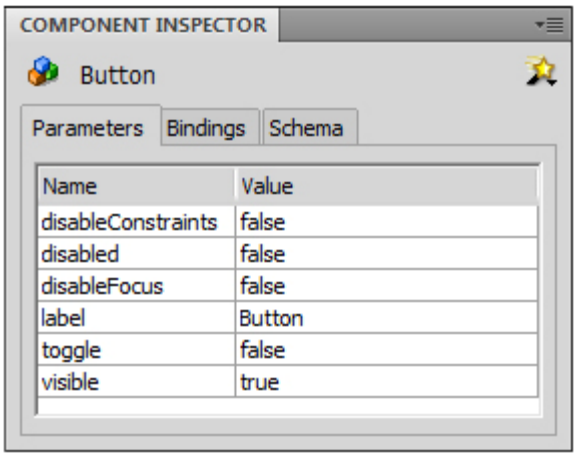

Figure 9: Button component inspectable properties in the CS4 component inspector.

The inspectable properties of the Button component are:

disableConstraints	The Button component contains a constraints object that determines the placement and scaling of the textField inside of the button when the component is resized. Setting this property to true will disable the constraints object. This is particularly useful if there is a need to resize or reposition the button's textField via a timeline animation and the button component is never resized. If not disabled, the textField will be moved and scaled to its default values throughout its lifetime, thus nullifying any textField translation/scaling tweens that may have been created in the button's timeline.
disabled	Disables the button if set to true. Once disabled, the button will not receive focus.
disableFocus	By default buttons receive focus for user interactions. Setting this property to true will disable focus acquisition.
label	Sets the text displayed inside the Button.
toggle	Sets the toggle mode of the Button. If set to true, the Button will act as a toggle button.
visible	Hides the button if set to false.

Changes to these properties will only be noticeable after publishing the SWF file that contains the component. The Flash IDE will not display any changes on the Stage at design time since the CLIK components are not compiled clips. This is necessary to ensure that the components are easily accessible and skinnable.

2.1.1.5 Events

Most components produce events for user interaction, state changes and focus management. These events are important to the creation of a functional user interface using CLIK components.

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the Button component are listed below. The properties listed next to the event are provided in addition to the common properties.

show	The visible property has been set to true at runtime.
hide	The visible property has been set to false at runtime.
focusIn	The button has received focus.

focusOut	The button has lost focus.
select	The selected property has changed. *selected*: The selected state of the Button, true for selected. Boolean type.
stateChange	The button's state has changed. *state*: The Button's new state. String type, Values "up", "over", "down", etc. See Getting Started with CLIK Buttons document for full list of states.
rollOver	The mouse cursor has rolled over the button. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse-cursor environments). Number type. Values 0 to 3.
rollOut	The mouse cursor has rolled out of the button. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
press	The button has been pressed. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
doubleClick	The button has been double clicked. Only fired when the *doubleClickEnabled* property is true. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
click	The button has been clicked. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
dragOver	The mouse cursor has been dragged over the button (while the left mouse button is pressed). *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
dragOut	The mouse cursor has been dragged out of the button (while the left mouse button is pressed). *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.

releaseOutside	The mouse cursor has been dragged out of the button and the left mouse button has been released. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.

A snippet of ActionScript code is required to catch or handle these events. The following example shows how to handle a button click:

```
myButton.addEventListener("click", this, "onButtonPress");
function onButtonPress(event:Object) {
	// Do something
}
```

The first line installs an event listener for the `"click"` event with the button named 'myButton', and tells it to call the `onButtonPress` function when the event occurs. The same code pattern can also be used to handle the other events. The Object parameter returned in the event handler (named 'event' in the example) contains relevant information about the event. This information is returned as properties of the Object parameter and differs for most events.

2.1.1.6 Tips and tricks

Create a Button component at runtime with custom properties:

```
import gfx.controls.Button;
attachMovie("Button", "btnInstanceName", someDepth, {_x:33, _y:262, …});
```

Set up a CLIK Button instance to toggle between the unselected and selected states. This is not required if the toggle property is set via the inspectable properties:

```
btn.toggle = true;
```

Enable double clicking with the left mouse button:

```
btn.doubleClickEnabled = true;
btn.addEventListener("doubleClick", this, "onDoubleClick");
function onDoubleClick(e:Object):Void {
	// Button was double clicked!
}
```

Enable firing of click events repeatedly when the button is held down:

```
btn.autoRepeat = true;
```

2.1.2 CheckBox

Figure 10: Unskinned CheckBox.

The CheckBox (gfx.controls.CheckBox) is a Button component that is set to toggle the selected state when clicked. CheckBoxes are used to display and change a true/false (Boolean) value. It is functionally equivalent to the ToggleButton, but sets the toggle property implicitly.

2.1.2.1 User interaction

Clicking on the CheckBox component using the mouse or any equivalent keyboard control will continuously select and deselect it. In all other respects, the CheckBox behaves the same as the Button.

2.1.2.2 Component setup

A MovieClip that uses the CLIK CheckBox class must have the following named subelements. Optional elements are noted appropriately:

- ***textField***: (optional) TextField type. The button's label.
- ***focusIndicator***: (optional) MovieClip type. A separate MovieClip used to display the focused state. If it exists, this MovieClip must have two named frames: "show" and "hide".

2.1.2.3 States

Due to its toggle property, the CheckBox requires another set of keyframes to denote the selected state. These states include:

- an ***up*** or default state;
- an ***over*** state when the mouse cursor is over the component, or when it is focused;

- a ***down*** state when the button is pressed;
- a ***disabled*** state;
- a ***selected_up*** or default state;
- a ***selected_over*** state when the mouse cursor is over the component, or when it is focused;
- a ***selected_ down*** state when the button is pressed;
- a ***selected_disabled*** state.

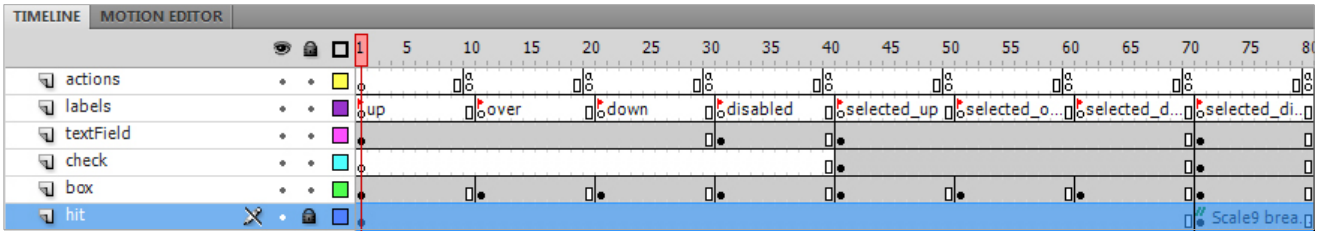

Figure 11: CheckBox timeline.

This is the minimal set of keyframes that should be in a CheckBox. The extended set of states and keyframes supported by the Button component, and consequently the CheckBox component, are described in the Getting Started with CLIK Buttons document.

2.1.2.4 Inspectable properties

Since it derives from the Button control, the CheckBox contains the same inspectable properties as the Button with the omission of the toggle and disableFocus properties.

data	Custom string that can be attached to the component as separate data than the CheckBox's label.
disableConstraints	The Button component contains a constraints object that determines the placement and scaling of the textField inside of the button when the component is resized. Setting this property to true will disable the constraints object. This is particularly useful if there is a need to resize or reposition the button's textField via a timeline animation and the button component is never resized. If not disabled, the textField will be moved and scaled to its default values throughout its lifetime, thus nullifying any textField translation/scaling tweens that may have been created in the button's timeline.
disabled	Disables the button if set to true.
label	Sets the label of the Button.
selected	Checks (or selects) the CheckBox when set to true.
visible	Hides the button if set to false.

2.1.2.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the CheckBox component are listed below. The properties listed next to the event are provided in addition to the common properties.

show	The visible property has been set to true at runtime.
hide	The visible property has been set to false at runtime.
focusIn	The component has received focus.
focusOut	The component has lost focus.
select	The component's selected property has changed. *selected*: The selected property of the Button. Boolean type.
stateChange	The button's state has changed. *state*: The Button's new state. String type, Values "up", "over", "down", etc. See Getting Started with CLIK Buttons document for full list of states.
rollOver	The mouse cursor has rolled over the button. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
rollOut	The mouse cursor has rolled out of the button. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
press	The button has been pressed. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
doubleClick	The button has been double clicked. Only fired when the *doubleClickEnabled* property is true. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
click	The button has been clicked. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0

	to 3.
dragOver	The mouse cursor has been dragged over the button (while the left mouse button is pressed). *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
dragOut	The mouse cursor has been dragged out of the button (while the left mouse button is pressed). *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
releaseOutside	The mouse cursor has been dragged out of the button and the left mouse button has been released. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.

The following example code reveals how to handle a CheckBox toggle:

```
myCheckBox.addEventListener("select", this, "onCheckBoxToggle");
function onCheckBoxToggle(event:Object) {
      // Do something
}
```

2.1.3 Label

Figure 12: Unskinned Label.

The CLIK Label component (gfx.controls.Label) is simply a noneditable standard textField wrapped by a MovieClip symbol, with a few additional convenient features. Internally, the Label supports the same properties and behaviors as the standard textField, however only a handful of commonly used features are exposed by the component itself. Access to the Label's actual textField is provided if the user needs to

change its properties directly. In certain cases, such as those described below, developers may use the standard textField instead of the Label component.

Since the Label is a MovieClip symbol, it can be embellished with graphical elements, which is not possible with the standard textField. As a symbol, it does not need to be configured per instance like textField instances. The Label also provides a disabled state that can be defined in the timeline. Whereas, complex AS2 code is required to mimic such behavior with the standard textField.

The Label component uses constraints by default, which means resizing a Label instance on the stage will have no visible effect at runtime. If resizing textFields is required, developers should use the standard textField instead of the Label in most cases. In general, if consistent reusability is not a requirement for the text element, the standard textField is a lighter weight alternative than the Label component.

2.1.3.1 User interaction

No user interaction is possible with the Label.

2.1.3.2 Component setup

A MovieClip that uses the CLIK Label class must have the following named subelements. Optional elements are noted appropriately:

- ***textField***: TextField type. The Label text.

2.1.3.3 States

The CLIK Label component supports two states based on the disabled property:

- a ***default*** or enabled state;
- a ***disabled*** state.

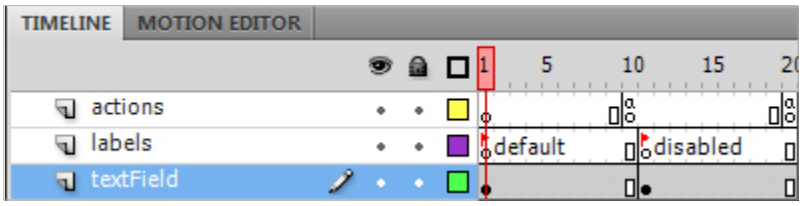

Figure 13: Label timeline.

2.1.3.4 Inspectable properties

The inspectable properties of the Label component are:

text	Sets the text of the label.
visible	Hides the label if set to false.
disabled	Disables the label if set to true.

2.1.3.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the Label component are listed below. The properties listed next to the event are provided in addition to the common properties.

show	The component's visible property has been set to true at runtime.
hide	The component's visible property has been set to false at runtime.
stateChange	The Label's state has changed. *state*: The new Label state. String type. Values "default" or "disabled".

The following example shows how to listen for a Label state change:

```
myLabel.addEventListener("stateChange", this, "onStateChange");
function onStateChange(event:Object) {
      if (event.state == "default") {
            // Do something
      }
}
```

2.1.3.6 Tips and tricks

Display HTML text in the Label component. See Flash 8 documentation for the HTML tags supported by the standard textField:

```
lbl.htmlText = "<b>foo</b>bar";
```

2.1.4 TextInput

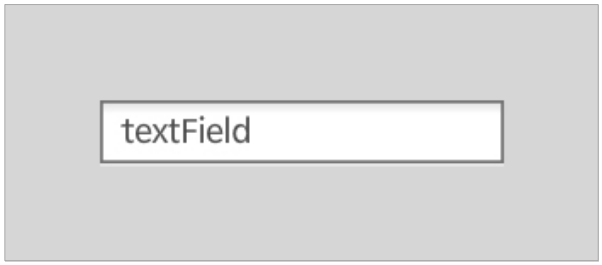

Figure 14: Unskinned TextInput.

TextInput (gfx.controls.TextInput) is an editable textField component used to capture textual user input. Similar to the Label, this component is merely a wrapper for a standard textField, and therefore supports the same capabilities of the textField such as password mode, maximum number of characters and HTML text. Only a handful of these properties are exposed by the component itself, while the rest can be modified by directly accessing the TextInput's textField instance.

The TextInput component should be used for input, since noneditable text can be displayed using the Label. Similar to the Label, developers may substitute standard textFields for TextInput components based on their requirements. However, when developing sophisticated UIs, especially for PC applications, the TextInput component provides valuable extended capabilities over the standard textField.

For staters, TextInput supports the focused and disabled state, which are not easily achieved with the standard textField. Due to the separated focus state, TextInput can support custom focus indicators, which are not included with the standard textField. Complex AS2 code is required to change the visual style of a standard textField, while the TextInput visual style can be configured easily on the timeline. The TextInput inspectable properties provide an easy workflow for designers and programmers who are not familiar with Flash Studio. Developers can also easily listen for events fired by the TextInput to create custom behaviors.

The TextInput also supports the standard selection and cut, copy, and paste functionality provided by the textField, including multi paragraph HTML formatted text. By default, the keyboard commands are select (Shift+Arrows), cut (Shift+Delete), copy (Ctrl+Insert), and paste (Shift+Insert).

The TextInput can support highlighting on mouse cursor roll over and roll out events. The special actAsButton inspectable property can be set to enable this mode which provides support for two extra keyframes that will be played for the two mouse events. These frames are named "over" and "out", and represent the rollOver and rollout states respectively. If the actAsButton mode is set, and the "over"/"out" frames do not exist, then the TextInput will play the "default" keyframe for both events. Note that these frames do not exist by default for the prebuilt TextInput component shipped with CLIK. They are meant to be added by developers depending on specific requirements.

This component also supports default text that is displayed when no value is set or entered by the user. The defaultText property can be set to any String. The theme (color and style) of default text will be light gray (0xAAAAAA) and italics. Custom styles can be set by assigning a new TextFormat object to the defaultTextFormat property.

2.1.4.1 User interaction

Clicking on the TextInput gives focus to it, which in turns causes a cursor to appear in the textField. When this cursor is visible, the user is able to type in characters via the keyboard or equivalent controller. Pressing the left and right arrow keys moves the cursor in the appropriate direction. If the left arrow key is used when the cursor is already at the left edge of the textField, then focus will be transferred to the control on the left. The same goes for the right arrow key.

2.1.4.2 Component setup

A MovieClip that uses the CLIK TextInput class must have the following named subelements. Optional elements are noted appropriately:

- ***textField***: textField type.

2.1.4.3 States

The CLIK TextInput component supports three states based on its focused and disabled properties:

- a ***default*** or enabled state;
- a ***focused*** state, typically a represented by a highlighted border around the textField;
- a ***disabled*** state.

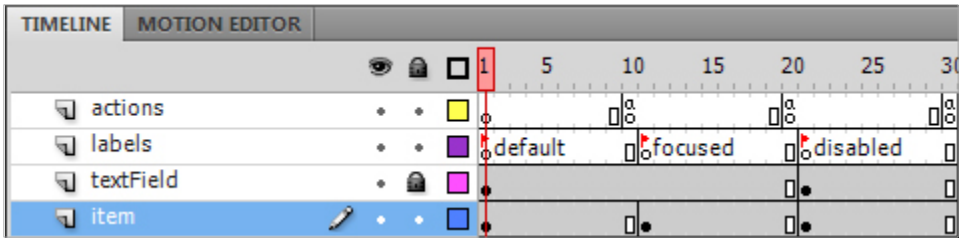

Figure 15: TextInput timeline.

2.1.4.4 Inspectable properties

The inspectable properties of the TextInput component are:

text	Sets the text of the textField.
visible	Hides the component if set to false.
disabled	Disables the component if set to true.
editable	Makes the TextInput noneditable if set to false.
maxChars	A number greater than zero limits the number of characters that can be entered in the textField.
password	If true, sets the textField to display '*' characters instead of the real characters. The value of the textField will be the real characters entered by the user, returned by the text property.
defaultText	Text to display when the textField is empty. This text is formatted by the defaultTextFormat object, which is by default set to light gray and italics.
actAsButton	If true, then the TextInput will behave similar to a Button when not focused and support rollOver and rollOut states. Once focused via mouse press or tab, the TextInput reverts to its normal mode until focus is lost.

2.1.4.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the TextInput component are listed below. The properties listed next to the event are provided in addition to the common properties.

show	The component's visible property has been set to true at runtime.
hide	The component's visible property has been set to false at runtime.
focusIn	The component has received focus.
focusOut	The component has lost focus.
textChange	The textField contents have changed.
rollOver	The mouse cursor has rolled over the component when not focused. Only fired when the actAsButton property is set. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
rollOut	The mouse cursor has rolled out of the component when not focused. Only fired when the actAsButton property is set. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.

The following code example reveals how to listen for changes to the textField contents:

```
myTextInput.addEventListener("textChange", this, "onTextChange");
function onTextChange(event:Object) {
      // Do something
}
```

2.1.4.6 Tips and Tricks

Stop auto selecting the text when focused:

```
_global.gfxExtensions = true;
textinput.textField.noAutoSelection = true;
```

Display HTML text in the TextInput component. See Flash 8 documentation for the HTML tags supported by the standard textField:

```
textinput.htmlText = "<b>foo</b>bar";
```

2.1.5 TextArea

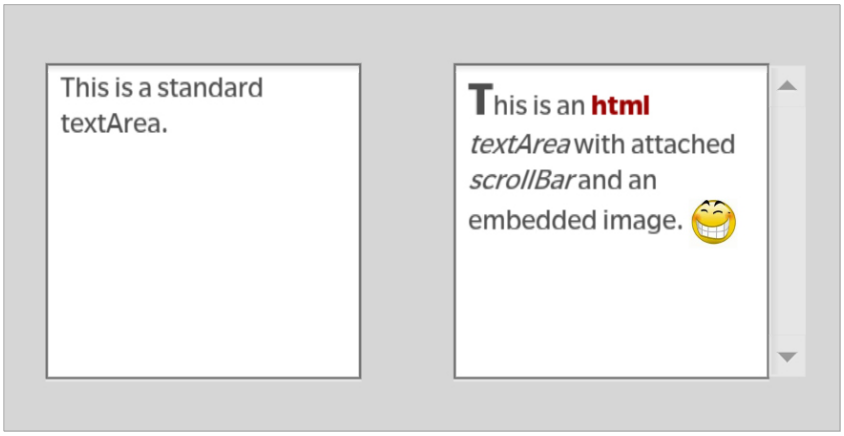

Figure 16: Unskinned TextArea.

TextArea (gfx.controls.TextArea) is derived from the CLIK TextInput, sharing the same functionality, properties and states, but with an optional ScrollBar for multiline editable scrollable text input. Please refer to the TextInput description to learn more about special functionalities shared between TextInput and TextArea.

Similar to the Label and TextInput, TextArea is also a wrapper for a standard multiline textField, and therefore supports the properties and behavior of a textField such as HTML text, word wrapping, selection, and cut, copy, paste. Developers are free to substitute TextArea with a standard textField, however, using this component is highly recommended for its extended functionality, states, inspectable properties and events.

Although the standard textField can be connected to a ScrollIndicator or ScrollBar, TextArea provides a tighter coupling with those components. Unlike the standard textField, TextArea can be scrolled when focused using the keyboard or equivalent controller even when it is noneditable. Since the scroll components cannot be focused, TextArea is able to present a more elegant focused graphical state that encompasses both itself and the scroll component in its focus state.

2.1.5.1 User interaction

Clicking on the TextArea gives focus to it, which in turns causes a cursor to appear in the textField. When this cursor is visible, the user is able to type in characters via the keyboard or equivalent controller. Pressing the four arrow keys moves the cursor in the appropriate direction. If the arrow keys are used when the cursor is already at an edge, then focus will be transferred to an adjacent control in the direction of the arrow key.

2.1.5.2 Component setup

A MovieClip that uses the CLIK TextArea class must have the following named subelements. Optional elements are noted appropriately:

- ***textField***: TextField type.

2.1.5.3 States

Similar to its parent, TextInput, the TextArea component supports three states based on its focused and disabled properties:

- a ***default*** or enabled state;
- a ***focused*** state, typically a represented by a highlighted border around the textField;
- a ***disabled*** state.

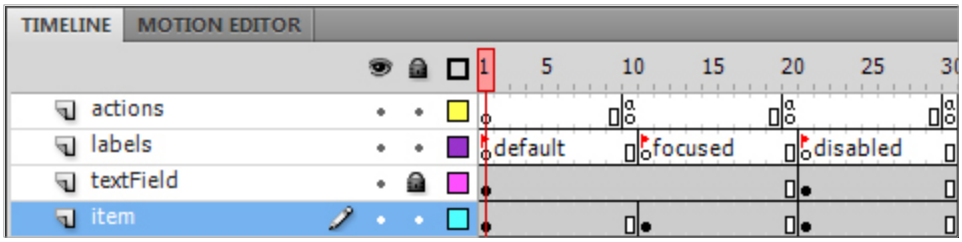

Figure 17: TextArea timeline.

2.1.5.4 Inspectable properties

The inspectable properties of the TextArea component are similar to TextInput with a couple of additions and the omission of the password property. The additions are related to the CLIK ScrollBar component, which is described in section 2.4:

Text	Sets the text of the textField.
Visible	Hides the component if set to false.
disabled	Disables the component if set to true.
editable	Makes the TextInput noneditable if set to false.
maxChars	A number greater than zero limits the number of characters that can be entered in the textField.
scrollBar	Instance name of the CLIK ScrollBar component to use, or a linkage ID to the ScrollBar symbol (an instance will be created by the TextArea in this case).
scrollPolicy	When set to "auto" the scroll bar will only show if there is enough text to scroll. The ScrollBar will always display if set to "on", and never display if set to "off". This property only affects the component if a ScrollBar is assigned to it (see the ScrollBar property).
defaultText	Text to display when the textField is empty. This text is formatted by the defaultTextFormat object, which is by default set to light gray and italics.
actAsButton	If true, then the TextInput will behave similar to a Button when not focused and support rollOver and rollOut states. Once focused via mouse press or tab, the TextArea reverts to its normal mode until focus is lost.

2.1.5.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the TextArea component are listed below. The properties listed next to the event are provided in addition to the common properties.

Show	The component's visible property has been set to true at runtime.
Hide	The component's visible property has been set to false at runtime.
focusIn	The component has received focus.
focusOut	The component has lost focus.
textChange	The textField contents have changed.
Scroll	The text area has been scrolled.
rollOver	The mouse cursor has rolled over the component when not focused. Only fired when the actAsButton property is set. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
rollout	The mouse cursor has rolled out of the component when not focused. Only fired when the actAsButton property is set. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.

The following example shows how to listen for TextArea scroll events:

```
myTextArea.addEventListener("scroll", this, "onTextScroll");
function onTextScroll(event:Object) {
      // Do something
}
```

2.1.5.6 Tips and tricks

Stop auto selecting the text when focused:

```
_global.gfxExtensions = true;
textinput.textField.noAutoSelection = true;
```

Display HTML text in the TextArea component. See Flash 8 documentation for the HTML tags supported by the standard textField:

```
textinput.htmlText = "<b>foo</b>bar";
```

2.2 Group Types

The group types consist of the RadioButton, ButtonGroup, and ButtonBar components. The ButtonGroup is a manager type that has special logic to maintain groups of Button components. It has no visual appearance

and does not exist on the Stage. However, the ButtonBar does exist on the Stage and also maintains groups of Buttons. The RadioButton is a special Button that is automatically grouped with its siblings into a ButtonGroup.

Figure 18: Radio button group example from *Dawn of War II*.

2.2.1 RadioButton

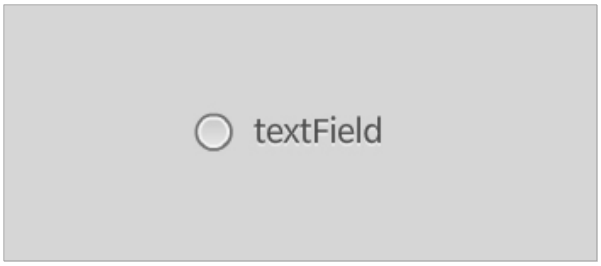

Figure 19: Unskinned RadioButton.

RadioButton (gfx.controls.RadioButton) is a button component that usually belongs in a set to display and change a single value. Only one RadioButton in a set can be selected, and clicking another RadioButton in the set selects the new component and deselects the previously selected component.

The CLIK RadioButton is very similar to the CheckBox component and shares its functionality, states and behavior. The main difference is that the RadioButton supports a group property, to which a custom ButtonGroup (see the next section) can be assigned. RadioButton also does not inherently set its toggle property since toggling is performed by the ButtonGroup instance that manages it.

2.2.1.1 User interaction

Clicking on the RadioButton component using the mouse or any equivalent control will continuously select it if not already selected. If a RadioButton is selected, and it belongs to the same ButtonGroup as another RadioButton that was just clicked, then the first radio will be deselected. In all other respects, the RadioButton behaves the same as the Button.

2.2.1.2 Component setup

A MovieClip that uses the CLIK RadioButton class must have the following named subelements. Optional elements are noted appropriately:

- ***textField***: (optional) TextField type. The button's label.
- ***focusIndicator***: (optional) MovieClip type. A separate MovieClip used to display the focused state. If it exists, this MovieClip must have two named frames: "show" and "hide".

2.2.1.3 States

Since the RadioButton is able to toggle between selected and unselected states, it, similar to the CheckBox, requires at least the following states:

- an ***up*** or default state;
- an ***over*** state when the mouse cursor is over the component, or when it is focused;
- a ***down*** state when the button is pressed;
- a ***disabled*** state;
- a ***selected_up*** or default state;
- a ***selected_over*** state when the mouse cursor is over the component, or when it is focused;
- a ***selected_down*** state when the button is pressed;
- a ***selected_disabled*** state.

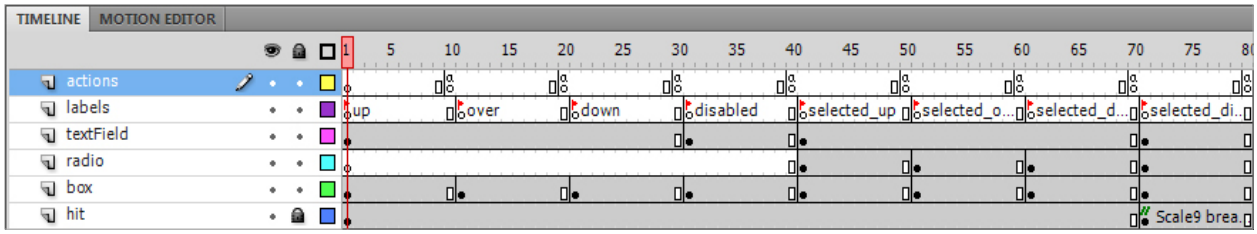

Figure 20: RadioButton timeline.

This is the minimal set of keyframes that should be in a RadioButton. The extended set of states and keyframes supported by the Button component, and consequently the RadioButton component, are described in the Getting Started with CLIK Buttons document.

2.2.1.4 Inspectable properties

Since it derives from the Button control, the RadioButton contains the same inspectable properties as the Button with the omission of the toggle and disableFocus properties.

Label	Sets the label of the Button.
Visible	Hides the button if set to false.
disabled	Disables the button if set to true.
disableConstraints	The Button component contains a constraints object that determines the placement and scaling of the textField inside of the button when the component is resized. Setting this property to true will disable the constraints object. This is particularly useful if there is a need to resize or reposition the button's textField via a timeline animation and the Button component is never resized. If not disabled, the textField will be moved and scaled to its default values throughout its lifetime, thus nullifying any textField translation/scaling tweens that may have been created in the Button's timeline.
selected	Checks (or selects) the RadioButton when set to true.
Data	Custom string that can be attached to the component as separate data than the RadioButton's label.
Group	A name of a ButtonGroup instance that exists or should be created automatically by the RadioButton. If created by the RadioButton, the new ButtonGroup will exist inside the container of the RadioButton. For example, if the RadioButton exists in *_root*, then its ButtonGroup object will be created in *_root*. All RadioButtons that use the same group will belong to one set.

2.2.1.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the RadioButton component are listed below. The properties listed next to the event are provided in addition to the common properties.

show	The component's visible property has been set to true at runtime.
hide	The component's visible property has been set to false at runtime.
focusIn	The component has received focus.
focusOut	The component has lost focus.
select	The component's selected property has changed. *selected*: The selected property of the Button. Boolean type.
stateChange	The button's state has changed. *state*: The button's new state. String type, Values "up", "over", "down", etc. See Getting Started with CLIK Buttons document for full list of states.
rollOver	The mouse cursor has rolled over the button. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
rollOut	The mouse cursor has rolled out of the button. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
press	The button has been pressed. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
doubleClick	The button has been double clicked. Only fired when the *doubleClickEnabled* property is true. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
click	The button has been clicked. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
dragOver	The mouse cursor has been dragged over the button (while the left mouse button is pressed). *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
dragOut	The mouse cursor has been dragged out of the button (while the left mouse button is pressed). *mouseIndex*: The index of the mouse cursor used to generate the event

	(applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
releaseOutside	The mouse cursor has been dragged out of the button and the left mouse button has been released. *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.

The following example shows how to handle a RadioButton toggle:

```
myRadio.addEventListener("select", this, "onRadioToggle");
function onRadioToggle(event:Object) {
      // Do something
}
```

2.2.1.6 Tips and tricks

Group RadioButtons not on the same level:

```
import gfx.controls.ButtonGroup;
var myGroup:ButtonGroup = new ButtonGroup("groupName");
radio1.group = myGroup;
radio2.group = myGroup;
someMovie.radio1.group = myGroup;
someMovie.radio2.group = myGroup;
```

2.2.2 ButtonGroup

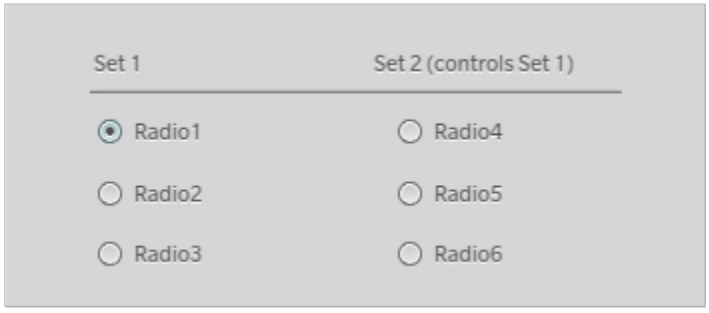

Figure 21: Unskinned ButtonGroup.

The CLIK ButtonGroup (gfx.controls.ButtonGroup) is not a component per se, but is important nonetheless because it is used to manage sets of buttons. It allows one button in the set to be selected, and ensures that

the rest are unselected. If the user selects another button in the set, then the currently selected button will be unselected. Any component that derives from the CLIK Button component (such as CheckBox and RadioButton) can be assigned a ButtonGroup instance.

2.2.2.1 User interaction

The ButtonGroup does not have any user interaction since it has no visual representation. However, it is indirectly affected when RadioButtons belonging to it are clicked.

2.2.2.2 Component setup

A MovieClip that uses the CLIK ButtonGroup class does not require any subelements because it does not have a visual representation.

2.2.2.3 States

The ButtonGroup does not have a visual representation on the Stage. Therefore no states are associated with it.

2.2.2.4 Inspectable properties

The ButtonGroup does not have a visual representation on the Stage. Therefore no inspectable properties are available.

2.2.2.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the ButtonGroup component are listed below. The properties listed next to the event are provided in addition to the common properties.

change	A new button from the group has been selected. • *item*: The selected button. CLIK Button type. • *data*: The data value of the selected button. AS2 Object type.
itemClick	A button in the group has been clicked. • *item*: The button that was clicked. CLIK Button type.

The following example shows how to determine which button in the ButtonGroup was selected:

```
myGroup.addEventListener("change", this, "onNewSelection");
function onNewSelection(event:Object) {
      if (event.item == myRadio) {
            // Do something
      }
}
```

2.2.2.6 Tips and tricks

Find out the currently selected RadioButton from a group:

```
var selectedRadio:Button = group.selectedButton;
```

Install listeners to the default ButtonGroup attached to a set of RadioButtons on the Stage:

```
radioBtn1.group.addEventListener("itemClick", this, "onClick");
```

2.2.3 ButtonBar

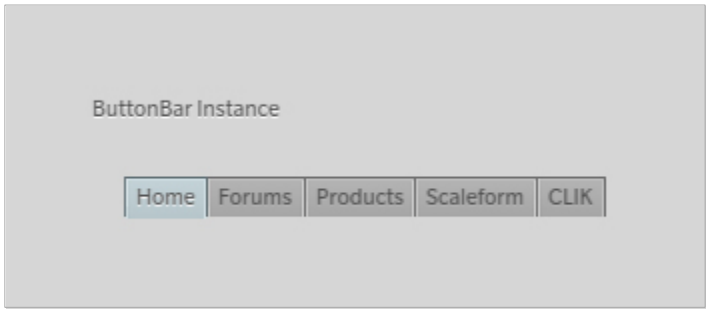

Figure 22: Unskinned ButtonBar.

The ButtonBar (gfx.controls.ButtonBar) is similar to the ButtonGroup, although it has a visual representation. It is also able to create Button instances on the fly, based on a dataProvider (see the Programming Details section for a description of the dataProvider). The ButtonBar is useful for creating dynamic tab-bar-like UI elements. This component is populated by assigning a dataProvider:

```
buttonBar.dataProvider = ["item1", "item2", "item3", "item4", "item5"];
```

2.2.3.1 User interaction

The ButtonBar has the same behavior as the ButtonGroup, and while users cannot directly interact with it, the ButtonBar is also indirectly affected when a Button maintained by it is clicked.

2.2.3.2 Component setup

A MovieClip that uses the CLIK ButtonBar class does not require any subelements because it creates them at runtime.

2.2.3.3 States

The CLIK ButtonBar does not have any visual states because its managed Button components are used to display the group state.

2.2.3.4 Inspectable properties

Although the ButtonBar component has no content (represented simply as a small circle on the Stage in the Flash IDE), it does contain several inspectable properties. The majority of them deal with the placement settings of the Button instances created by the ButtonBar.

visible	Hides the ButtonBar if set to false.
disabled	Disables the ButtonBar if set to true.
itemRenderer	Linkage ID of the Button component symbol. This symbol will be instantiated as needed based on the data assigned to the ButtonBar.
direction	Button placement. Horizontal will place the Button instances side-by-side, while vertical will stack them on top of each other.
spacing	The spacing between the Button instances. Affects only the current direction (see *direction* property).
autoSize	If set to true, resizes the Button instances to fit the displayed label.
buttonWidth	Sets a common width to all Button instances. If autoSize is set to true this property is ignored.

2.2.3.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the ButtonBar component are listed below. The properties listed next to the event are provided in addition to the common properties.

show	The component's visible property has been set to true at runtime.
hide	The component's visible property has been set to false at runtime.
focusIn	The component has received focus.
focusOut	The component has lost focus.
change	A new button from the group has been selected. • *index*: The selected index of the ButtonBar. Number type. Values 0 to number of buttons minus 1. • *renderer*: The selected Button. CLIK Button type. • *item*: The selected item from the dataProvider. AS2 Object type. • *data*: The data value of the selected dataProvider item. AS2 Object type.
itemClick	A button in the group has been clicked. • index: The ButtonBar index of the Button that was clicked. Number type. Values 0 to number of buttons minus 1. • item: The selected item from the dataProvider. AS2 Object type. • data: The data value of the selected dataProvider item. AS2 Object type. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.

The following example shows how to detect when a Button instance inside the ButtonBar has been clicked:

```
myBar.addEventListener("itemClick", this, "onItemClick");
function onItemClick(event:Object) {
      processData(event.data);
      // Do something
}
```

2.3 Scroll Types

The scroll types consist of the ScrollIndicator, ScrollBar and Slider components. The ScrollIndicator is non-interactive and simply displays the scroll index of a target component using a thumb, while the ScrollBar allows user interactions to affect the scroll position. The ScrollBar is composed of four Buttons: the thumb or grip, the track, the up arrow and the down arrow. The Slider is similar to the ScrollBar, but it only contains an interactive thumb and track, and does not resize the thumb based on the number of elements in the target

component. The Slider is similar to the ScrollBar, but it only contains an interactive thumb and track, and does not resize the thumb based on the number of elements in the target component.

Figure 23: Audio menu slider examples from *Mercenaries 2*.

2.3.1 ScrollIndicator

Figure 24: Unskinned ScrollIndicator.

The CLIK ScrollIndicator (gfx.controls.ScrollIndicator) displays the scroll position of another component, such as a multiline textField. It can be pointed at a textField to automatically display its scroll position. All list-based components, as well as the TextArea, have a scrollBar property that can be pointed to a ScrollIndicator or ScrollBar (see next section) instance or linkage ID.

2.3.1.1 User interaction

The ScrollInidicator does not have any user interaction. It is intended for display purposes only.

2.3.1.2 Component setup

A MovieClip that uses the CLIK ScrollIndicator class must have the following named subelements. Optional elements are noted appropriately:

- ***thumb***: CLIK Button type. The scroll indicator grip.
- ***track***: MovieClip type. The scroll indicator track. The bounds of this track determine the extents to which the grip can travel.

2.3.1.3 States

The ScrollIndicator does not have explicit states. It uses the states of its child elements: the thumb and track Button components.

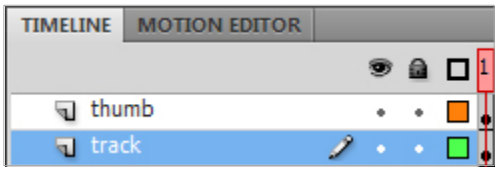

Figure 25: ScrollIndicator timeline.

2.3.1.4 Inspectable properties

The inspectable properties of the ScrollIndicator are:

scrollTarget	Set a TextArea or normal multiline textField as the scroll target to automatically respond to scroll events. Non-textField types have to manually update the ScrollIndicator properties.
Visible	Hides the component if set to false.
disabled	Disables the component if set to true.
offsetTop	Thumb offset at the top. A positive value moves the thumb's top-most position higher.
offsetBottom	Thumb offset at the bottom. A positive value moves the thumb's bottom-most position lower.

2.3.1.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the ScrollIndicator component are listed below. The properties listed next to the event are provided in addition to the common properties.

Show	The component's visible property has been set to true at runtime.
Hide	The component's visible property has been set to false at runtime.
Scroll	The scroll position has changed. • *position*: The new scroll position. Number type. Values minimum position to maximum position.

The following example shows how to listen for scroll events:

```
mySI.addEventListener("scroll", this, "onTextScroll");
function onTextScroll(event:Object) {
       trace("scroll position: " + event.position);
      // Do something
}
```

2.3.1.6 Tips and tricks

Set up a standalone scroll indicator instance manually:

```
scrollInd.setScrollProperties(pageSize, minPos, maxPos, pageScrollSz);
scrollInd.positon = currPos;
```

Set the scrolling direction. This is set automatically for components created on the Stage using their _rotation property. If the ScrollIndicator component is not rotated and is horizontal by default, then this value should be set explicitly:

```
scrollInd.direction = "horizontal";
```

2.3.2 ScrollBar

Figure 26: Unskinned ScrollBar.

The CLIK ScrollBar (gfx.controls.ScrollBar) displays and controls the scroll position of another component. It adds interactivity to the ScrollIndicator with a draggable thumb button, as well as optional up and down arrow buttons, and a clickable track.

2.3.2.1 User interaction

The ScrollBar thumb can be gripped by the mouse or equivalent control and dragged between the bounds of the ScrollBar track. Moving the mouse wheel while the mouse cursor is on top of the ScrollBar performs a scroll operation. Clicking on the up arrow moves the thumb up, and clicking the down arrow moves the thumb down. Clicking the track can have two behaviors: the thumb continuously scrolls towards the clicked point, or immediately jumps to that point and is set to drag. This track mode is determined by the *trackMode* inspectable property (see Inspectable properties section). Regardless of the trackMode setting, pressing the (Shift) key and clicking on the track will immediately move the thumb to the cursor and set it to drag.

2.3.2.2 Component setup

A MovieClip that uses the CLIK ScrollBar class must have the following named subelements. Optional elements are noted appropriately:

- ***upArrow***: CLIK Button type. Button that scrolls up; typically placed at the top of the scrollbar.
- ***downArrow***: CLIK Button type. Button that scrolls down; typically placed at the bottom of the scrollbar.
- ***thumb***: CLIK Button type. The grip of the scrollbar.
- ***track***: CLIK Button type. The scrollbar track whose boundary determines the extent to which the grip can move.

2.3.2.3 States

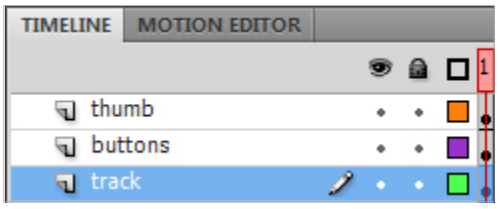

Figure 27: ScrollBar timeline.

The ScrollBar, similar to the ScrollIndicator, does not have explicit states. It uses the states of its child elements: the thumb, up, down and track Button components.

2.3.2.4 Inspectable properties

The inspectable properties of the ScrollBar are similar to ScrollIndicator, with one addition:

scrollTarget	Set a TextArea or normal multiline textField as the scroll target to automatically respond to scroll events. Non-textField types have to manually update the ScrollIndicator properties.
trackMode	When the user clicks on the track with the cursor, the scrollPage setting will cause the thumb to continuously scroll by a page until the cursor is released. The scrollToCursor setting will cause the thumb to immediately jump to the cursor and will also transition the thumb into a dragging mode until the cursor is released.
Visible	Hides the component if set to false.
disabled	Disables the component if set to true.
offsetTop	Thumb offset at the top. A positive value moves the thumb's top-most position higher.
offsetBottom	Thumb offset at the bottom. A positive value moves the thumb's bottom-most position lower.

2.3.2.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the ScrollBar component are listed below. The properties listed next to the event are provided in addition to the common properties.

Show	The component's visible property has been set to true at runtime.
Hide	The component's visible property has been set to false at runtime.
Scroll	The scroll position has changed. • *position*: The new scroll position. Number type. Values minimum position to maximum position.

The following code example reveals how to listen for scroll events:

```
mySB.addEventListener("scroll", this, "onListScroll");
function onListScroll(event:Object) {
      trace("scroll position: " + event.position);
      // Do something
```

```
    }
```

2.3.2.6 Tips and tricks

Set up a standalone scroll indicator instance manually:

```
    scrollBar.setScrollProperties(pageSize, minPos, maxPos, pageScrollSz);
    scrollBar.positon = currPos;
```

Set the scrolling direction. This is set automatically for components created on the Stage using their _rotation property. If the scroll bar component is not rotated and is horizontal by default, then this value should be set explicitly:

```
    scrollBar.direction = "horizontal";
```

Set the number of positions scrolled when the track is pressed in *scrollPage* mode. By default the value is 1:

```
    scrollBar.trackScrollPageSize = pageSize;
```

2.3.3 Slider

Figure 28: Unskinned Slider.

The Slider (gfx.controls.Slider) displays a numerical value in range, with a thumb to represent the value, as well as modify it via dragging.

2.3.3.1 User interaction

The Slider thumb can be dragged by the mouse or equivalent control between the Slider track bounds. Clicking on the track will immediately move the thumb to the cursor position and set it to drag. While focused, the left and right arrow keys move the thumb in the appropriate direction, while the home and end keys move the thumb to the beginning and end of the track.

2.3.3.2 Component setup

A MovieClip that uses the CLIK Slider class must have the following named subelements. Optional elements are noted appropriately:

- ***thumb***: CLIK Button type. The slider grip.
- ***track***: CLIK Button type. The slider track bounds determine the extents the grip can travel.

2.3.3.3 States

Similar to the ScrollIndicator and the ScrollBar, the Slider does not have explicit states. It uses the states of its child elements: the thumb and track Button components.

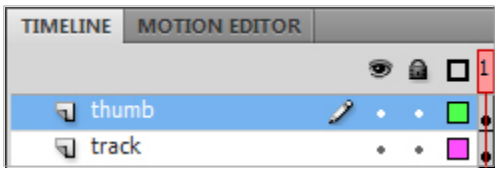

Figure 29: Slider timeline.

2.3.3.4 Inspectable properties

The inspectable properties of the Slider component are:

Visible	Hides the component if set to false.
disabled	Disables the component if set to true.
Value	The numeric value displayed by the Slider.
minimum	The minimum value of the Slider's range.
maximum	The maximum value of the Slider's range.
snapping	If set to true, then the thumb will snap to values that are multiples of snapInterval.
snapInterval	The snapping interval that determines which multiples of values the thumb snaps to. It has no effect if snapping is set to false.
liveDragging	If set to true, then the Slider will generate a change event when dragging the thumb. If false, then the Slider will only generate a change event after the dragging is over.
offsetLeft	Left offset for the thumb. A positive value will push the thumb inward.
offsetRight	Right offset for the thumb. A positive value will push the thumb inward.

2.3.3.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the Slider component are listed below. The properties listed next to the event are provided in addition to the common properties.

Show	The component's visible property has been set to true at runtime.
Hide	The component's visible property has been set to false at runtime.
focusIn	The component has received focus.
focusOut	The component has lost focus.
Change	The Slider value has changed.

The following example shows how to listen for Slider value changes:

```
mySlider.addEventListener("change", this, "onValueChange");
function onValueChange(event:Object) {
	trace("slider value: " + event.target.value);
	// Do something
}
```

2.4 List Types

The CLIK list types consist of the NumericStepper, OptionStepper, ScrollingList, TileList and DropdownMenu components. All of these components, except for the NumericStepper, work with a DataProvider to manage the displayed information. The ListItemRenderer component is also included in this category because it is used by the ScrollingList and TileList components to display the list items.

The NumericStepper and OptionStepper only display one element at a time, while the ScrollingList and TileList are capable of displaying more than one element. The two latter components can support either a ScrollIndicator or ScrollBar component. The DropdownMenu component displays one element in its idle state, but displays more elements using either a ScrollingList or TileList when opened.

Figure 30: Scrolling list with scroll indicator example from *Mercenaries 2*.

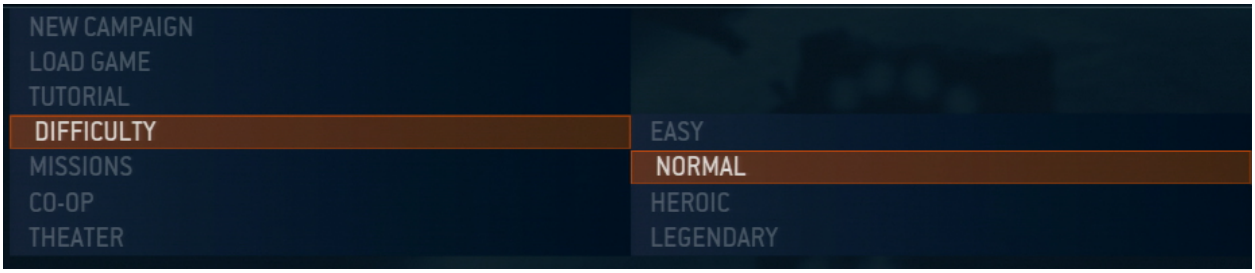

Figure 31: 'Difficulty Settings' dropdown menu example from *Halo Wars*.

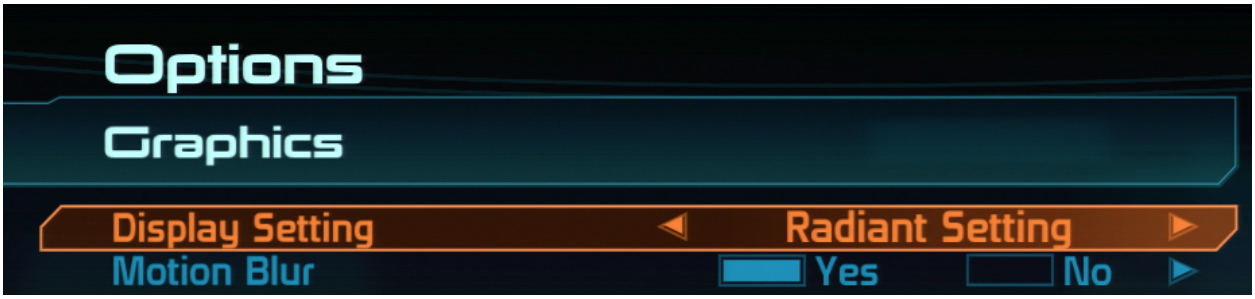

Figure 32: 'Display Setting' option stepper example from *Mass Effect*.

2.4.1 NumericStepper

Figure 33: Unskinned Numeric Stepper.

The NumericStepper (gfx.controls.NumericStepper) displays a single number in the range assigned to it, and supports the ability to increment and decrement the value based on an arbitrary step size.

2.4.1.1 User interaction

The NumericStepper includes two arrow buttons that can be clicked on via the mouse or equivalent controller to change its numerical value. When focused, the numerical value can also be changed via the keyboard using the left and right arrow keys or equivalent controls. These keys decrement and increment the current value by the step size. Pressing the (Home) and (End) keys or equivalent controls will change the numerical value to the minimum and maximum values respectively.

2.4.1.2 Component setup

A MovieClip that uses the NumericStepper class must have the following named subelements. Optional elements are noted appropriately:

- ***textField***: TextField type. Used to display the current value.
- ***nextBtn***: CLIK Button type. Clicking this will increment the current value by the step size.
- ***prevBtn***: CLIK Button type. Clicking this will decrement the current value by the step size.

2.4.1.3 States

The NumericStepper component supports three states based on its focused and disabled properties:

- a ***default*** or enabled state;
- a ***focused*** state, that highlights the textField area;
- a ***disabled*** state.

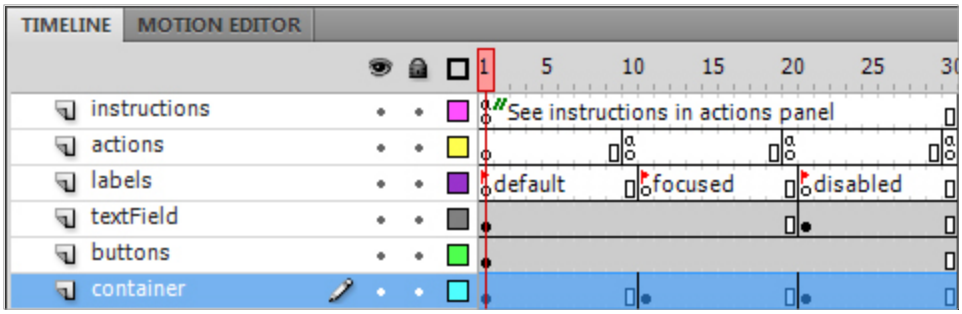

Figure 34: NumericStepper timeline.

2.4.1.4 Inspectable properties

A MovieClip that derives from the NumericStepper component will have the following inspectable properties:

Visible	Hides the component if set to false.
disabled	Disables the component if set to true.
Value	The numeric value displayed by the NumericStepper.
minimum	The minimum value of the NumericStepper's range.
maximum	The maximum value of the NumericStepper's range.

2.4.1.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the NumericStepper component are listed below. The properties listed next to the event are provided in addition to the common properties.

Show	The component's visible property has been set to true at runtime.
Hide	The component's visible property has been set to false at runtime.
Change	The NumericStepper's value has changed.
stateChange	The NumericStepper's focused or disabled property has changed. • *state*: Name of the new state. String type. Values "default", "focused" or "disabled".

The following example shows how to listen for a NumericStepper's value changes:

```
myNS.addEventListener("change", this, "onValueChange");
function onValueChange(event:Object) {
```

```
            trace("ns value: " + event.target.value);
            // Do something
      }
```

2.4.1.6 Tips and tricks

Change the increment/decrement interval:

```
      ns.stepSize = 0.5;
```

Add a decorator to the numeric value. Example:

```
      ns.labelFunction = function(value:Number) {
            switch(value) {
                        case 1:
                              return value + "st";
                        default:
                              return value + "th";
                  }
      }
```

2.4.2 OptionStepper

Figure 35: Unskinned OptionStepper.

The OptionStepper (gfx.controls.OptionStepper), similar to the NumericStepper, displays a single value, but is capable of displaying more than just numbers. It uses a dataProvider instance to query for the current value; therefore it is able to support an arbitrary number of elements of different types. The displayed value is set via code using the OptionStepper's selectedIndex property, which is a 0-based index into the attached dataProvider. The dataProvider is assigned via code, as shown in the example below:

```
      optionStepper.dataProvider = ["item1", "item2", "item3", "item4"];
```

2.4.2.1 User interaction

Similar to the NumericStepper, the OptionStepper includes two arrow buttons that can be clicked on via the mouse or equivalent controller to change its current value. When focused, the current value can also be changed via the keyboard using the left and right arrow keys or equivalent controls. These keys change the current value to the previous and next values. Pressing the (Home) and (End) keys or equivalent controls will change the current value to the first and last elements in its dataProvider.

2.4.2.2 Component setup

A MovieClip that uses the NumericStepper class must have the following named subelements. Optional elements are noted appropriately:

- ***textField***: TextField type. Used to display the current value.
- ***nextBtn***: CLIK Button type. Changes the current value to the next element in the data provider.
- ***prevBtn***: CLIK Button type. Changes the current value to the previous element in the dataProvider.

2.4.2.3 States

The OptionStepper component supports three states based on its focused and disabled properties:

- a ***default*** or enabled state;
- a ***focused*** state, that highlights the textField area;
- a ***disabled*** state.

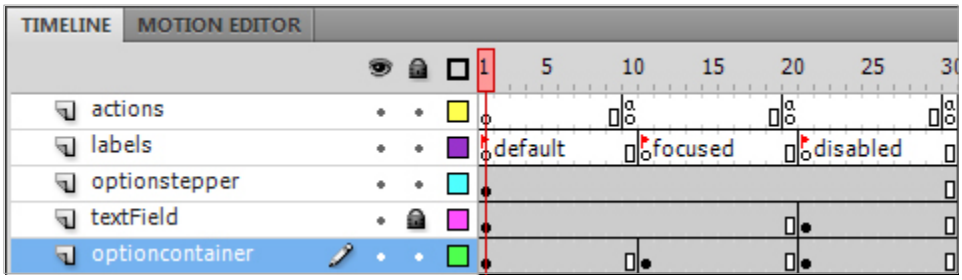

Figure 36: OptionStepper timeline.

2.4.2.4 Inspectable properties

A MovieClip that derives from the OptionStepper component will have the following inspectable properties:

Visible	Hides the component if set to false.
disabled	Disables the component if set to true.

2.4.2.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the OptionStepper component are listed below. The properties listed next to the event are provided in addition to the common properties.

Show	The component's visible property has been set to true at runtime.
Hide	The component's visible property has been set to false at runtime.
Change	The OptionStepper's value has changed.
stateChange	The OptionStepper's focused or disabled property has change. • *state*: Name of the new state. String type. Values "default", "focused" or "disabled".

The following example shows how to listen for an OptionStepper's value changes:

```
myOS.addEventListener("change", this, "onValueChange");
function onValueChange(event:Object) {
	trace("os value: " + event.target.selectedItem);
	// Do something
}
```

2.4.3 ListItemRenderer

Figure 37**: Unskinned ListItemRenderer.**

The ListItemRenderer (gfx.controls.ListItemRenderer) derives from the CLIK Button class and extends it to include list-related properties that are useful for its container components. However, it is not designed to be

a standalone component, instead it is only used in conjunction with the ScrollingList, TileList and DropdownMenu components.

2.4.3.1 User interaction

Since the ListItemRenderer derives from the Button component, it has similar user interactions as the Button such as being pressed using the mouse. Rolling the mouse cursor over and out of the ListItemRenderer also affects the component, as well as dragging the mouse cursor in and out of it. Keyboard or equivalent controller interactions are defined by the container component of the ListItemRenderer.

2.4.3.2 Component setup

A MovieClip that uses the CLIK ListItemRenderer class must have the following named subelements. Optional elements are noted appropriately:

- ***textField***: (optional) TextField type. The list item's label.
- ***focusIndicator***: (optional) MovieClip type. A separate MovieClip used to display the focused state. If it exists, this MovieClip must have two named frames: "show" and "hide".

2.4.3.3 States

Since it can be selected inside a container component, the ListItemRenderer requires the *selected* set of keyframes to denote its selected state. This component's states include:

- an ***up*** or default state;
- an ***over*** state when the mouse cursor is over the component, or when it is focused;
- a ***down*** state when the button is pressed;
- a ***disabled*** state;
- a ***selected_up*** or default state;
- a ***selected_over*** state when the mouse cursor is over the component, or when it is focused;
- a ***selected_ down*** state when the button is pressed;
- a ***selected_disabled*** state.

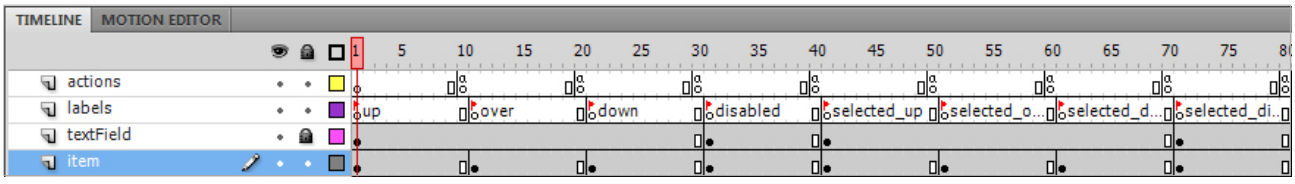

Figure 38: ListItemRenderer timeline.

This is the minimal set of keyframes that should be in a ListItemRenderer. The extended set of states and keyframes supported by the Button component, and consequently the ListItemRenderer component, are described in the Getting Started with CLIK Buttons document.

2.4.3.4 Inspectable properties

Since the ListItemRenderer's are controlled by a container component and never configured manually by a user, they contain only a small subset of the inspectable properties of the Button.

Label	Sets the label of the ListItemRenderer.
Visible	Hides the button if set to false.
disabled	Disables the button if set to true.

2.4.3.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the ListItemRenderer component are the same as the Button component. The properties listed next to the event are provided in addition to the common properties.

show	The component's visible property has been set to true at runtime.
hide	The component's visible property has been set to false at runtime.
focusIn	The component has received focus.
focusOut	The component has lost focus.
select	The component's selected property has changed. • *selected*: The selected property of the Button. Boolean type.
stateChange	The button's state has changed. • *state*: The Button's new state. String type, Values "up", "over", "down", etc. See Getting Started with CLIK Buttons document for full list of states.
rollOver	The mouse cursor has rolled over the button. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
rollOut	The mouse cursor has rolled out of the button. • *mouseIndex*: The index of the mouse cursor used to generate the event

	(applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
press	The button has been pressed. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
doubleClick	The button has been double clicked. Only fired when the *doubleClickEnabled* property is true. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
click	The button has been clicked. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
dragOver	The mouse cursor has been dragged over the button (while the left mouse button is pressed). • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
dragOut	The mouse cursor has been dragged out of the button (while the left mouse button is pressed). • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
releaseOutside	The mouse cursor has been dragged out of the button and the left mouse button has been released. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.

2.4.4 ScrollingList

Figure 39: Unskinned ScrollingList.

The ScrollingList (gfx.controls.ScrollingList) is a list component that can scroll its elements. It can instantiate list items by itself or use existing list items on the Stage. A ScrollIndicator or ScrollBar component can also be attached to this list component to provide scroll feedback and control. This component is populated via a dataProvider. The dataProvider is assigned via code, as shown in the example below:

```
scrollingList.dataProvider = ["item1", "item2", "item3", "item4"];
```

By default the ScrollingList uses the ListItemRenderer component for its contents. Therefore the ListItemRenderer must also exist in the FLA file's Library for it to work, unless the itemRenderer inspectable property is changed to another component. See the Inspectable properties section for more information.

2.4.4.1 User interaction

Clicking on a list item or an attached ScrollBar instance will transfer focus to the ScrollingList component. While focused, pressing the keyboard up and down arrows or equivalent controls scrolls the list selection by a single element. If no element is selected, the topmost element is automatically selected for this action. The mouse wheel scrolls the list if the cursor is on top of the ScrollingList boundary.

The scrolling behavior at the list boundaries is determined by the ScrollingList's *wrapping* property and is not inspectable. If *wrapping* is set to "normal", focus will leave the component when selection reaches the beginning or end of the list. If *wrapping* is set to "wrap", then selection will wrap to the beginning or end. If *wrapping* is set to "stick", then selection will stop when it reaches the end of data and focus is not transferred to adjacent components.

Pressing the keyboard (Page Up) and (Page down) keys or equivalent controls will scroll the selection by a page, i.e., the number of visible items in the list. Pressing the (Home) and (End) keys, or equivalent controls,

will scroll the list to the first and last elements respectively. Interacting with an attached ScrollBar component will affect the ScrollingList as expected. See the ScrollBar section to learn about its own user interaction.

Developers can easily map gamepad controls to keyboard and mouse controls. For example, the keyboard arrow keys are typically mapped to the D-Pad on console controllers. This mapping allows UIs built with CLIK to work in a wide variety of platforms.

2.4.4.2 Component setup

The ScrollingList does not require any named subelements. However, a visible background is helpful when placing and resizing an instance of the ScrollingList component on the Stage.

2.4.4.3 States

The ScrollingList component supports three states based on its focused and disabled properties:

- a ***default*** or enabled state;
- a ***focused*** state, that typically highlights the component's border area;
- a ***disabled*** state.

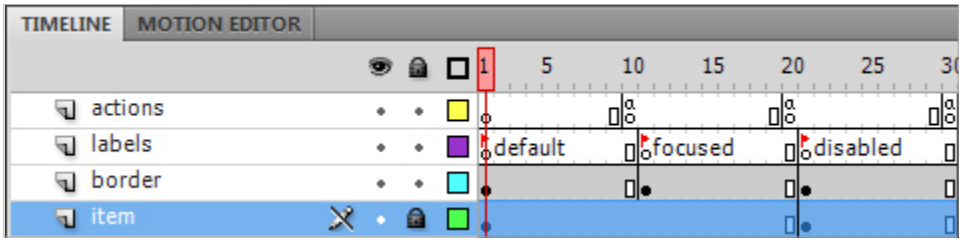

Figure 40**: ScrollingList timeline.**

2.4.4.4 Inspectable properties

A MovieClip that derives from the ScrollingList component will have the following inspectable properties:

visible	Hides the component if set to false. This does not hide the attached scrollbar or any external list item renderers.
disabled	Disables the component if set to true. This does disable both the attached scrollbar and the list items (both internally created and external renderers).
itemRenderer	The symbol name of the ListItemRenderer. Used to create list item instances internally. Has no effect if the *rendererInstanceName* property is set.
renderInstanceName	Prefix of the external list item renderers to use with this ScrollingList

	component. The list item instances on the Stage must be prefixed with this property value. If this property is set to the value 'r', then all list item instances to be used with this component must have the following values: 'r1', 'r2', 'r3',… The first item should have the number 1.
scrollBar	Instance name of a ScrollBar component on the Stage or a symbol name. If an instance name is specified, then the ScrollingList will hook into that instance. If a symbol name is specified, an instance of the symbol will be created by the ScrollingList.
margin	The margin between the boundary of the list component and the list items created internally. This value has no effect if the *rendererInstanceName* property is set.
rowHeight	The height of list item instances created internally. This value has no effect if the *rendererInstanceName* property is set.
paddingTop	Extra padding at the top for the list items. This value has no effect if the rendererInstanceName property is set. Does not affect the automatically generated scrollbar.
paddingBottom	Extra padding at the bottom for the list items. This value has no effect if the rendererInstanceName property is set. Does not affect the automatically generated scrollbar.
paddingLeft	Extra padding on the left side for the list items. This value has no effect if the rendererInstanceName property is set. Does not affect the automatically generated scrollbar.
paddingRight	Extra padding on the right side for the list items. This value has no effect if the rendererInstanceName property is set. Does not affect the automatically generated scrollbar.
thumbOffsetTop	Scrollbar thumb top offset. This property has no effect if the list does not automatically create a scrollbar instance.
thumbOffsetBottom	Scrollbar thumb bottom offset. This property has no effect if the list does not automatically create a scrollbar instance.
thumbSizeFactor	Page size factor for the scrollbar thumb. A value greater than 1.0 will increase the thumb size by the given factor. A positive value less than 1.0 will decreate the thumb size. This value has no effect if a scrollbar is not attached to the list.

2.4.4.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the ScrollingList component are listed below. The properties listed next to the event are provided in addition to the common properties.

show	The component's visible property has been set to true at runtime.
hide	The component's visible property has been set to false at runtime
focusIn	The component has received focus.
focusOut	The component has lost focus.
change	The selected index has changed. • *index*: The new selected index. Number type. Values 0 to number of list items minus 1.
itemPress	A list item has been pressed down. • *renderer*: The list item that was pressed. CLIK Button type. • *item*: The data associated with the list item. This value is retrieved from the list's dataProvider. AS2 Object type. • *index*: The index of the list item relative to the list's dataProvider. Number type. Values 0 to number of list items minus 1. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
itemClick	A list item has been clicked. • *renderer*: The list item that was clicked. CLIK Button type. • *item*: The data associated with the list item. This value is retrieved from the list's dataProvider. AS2 Object type. • *index*: The index of the list item relative to the list's dataProvider. Number type. Values 0 to number of list items minus 1. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
itemDoubleClick	A list item has been double clicked. • *renderer*: The list item that was double clicked. CLIK Button type. • *item*: The data associated with the list item. This value is retrieved from the list's dataProvider. AS2 Object type. • *index*: The index of the list item relative to the list's dataProvider. Number type. Values 0 to number of list items minus 1. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number

	type. Values 0 to 3.
itemRollOver	The mouse cursor has rolled over a list item. • *renderer*: The list item that was rolled over. CLIK Button type. • *item*: The data associated with the list item. This value is retrieved from the list's dataProvider. AS2 Object type. • *index*: The index of the list item relative to the list's dataProvider. Number type. Values 0 to number of list items minus 1. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
itemRollOut	The mouse cursor has rolled out of a list item. • *renderer*: The list item that was rolled out of. CLIK Button type. • *item*: The data associated with the list item. This value is retrieved from the list's dataProvider. AS2 Object type. • *index*: The index of the list item relative to the list's dataProvider. Number type. Values 0 to number of list items minus 1. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.

The following example shows how to listen to a list item being clicked:

```
myList.addEventListener("itemClick", this, "onItemClicked");
function onItemClicked(event:Object) {
      trace("list item was clicked: " + event.renderer);
      // Do something
}
```

2.4.4.6 Tips and tricks

Displaying a single label from a set of complex objects:

```
// The ScrollingList automatically will use the property named 'label'
// if found in the item object:
list.dataProvider = [{label: "one", data:1}, {label: "two", data:2}];

// However if the item object has a different label property, such as
// the following, then the list can be configured to use that property
// instead:
list.labelField = "name";
list.dataProvider = [{name: "one", data:1}, {name: "two", data:2}];
```

```
    // If the logic to construct a label from the item object is
    // complicated and requires a function, then set the labelFunction
    // property:
list.labelFunction = function(itemObj:Object):String {
    // Logic to construct a label
}
list.dataProvider = [{p1: "foo", p2: 1}, {p1: "bar", p2: 2}];
```

2.4.5 TileList

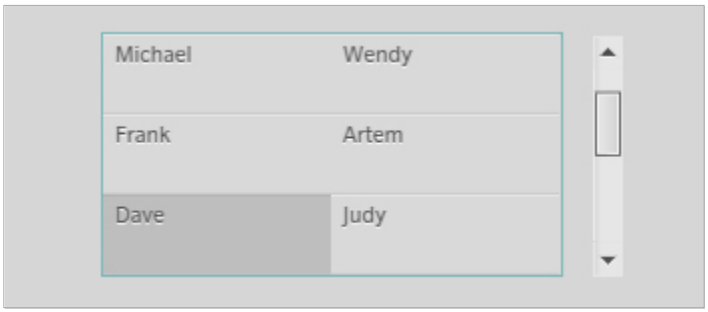

Figure 41: Unskinned TileList.

The TileList (gfx.controls.TileList), similar to the ScrollingList, is a list component that can scroll its elements. It can instantiate list items by itself or use existing list items on the Stage. A ScrollIndicator or ScrollBar component can also be attached to this list component to provide scroll feedback and control. The difference between the TileList and the ScrollingList is that the TileList can support multiple rows and columns at the same time. List item selection can move in all four cardinal directions. This component is populated via a dataProvider. The dataProvider is assigned via code, as shown in the example below:

```
    tileList.dataProvider = ["item1", "item2", "item3", "item4", "item5"];
```

By default the TileList uses the ListItemRenderer component for its contents. Therefore the ListItemRenderer must also exist in the FLA file's Library for it to work, unless the itemRenderer inspectable property is changed to another component. See the Inspectable properties section for more information.

2.4.5.1 User interaction

Clicking on a list item or an attached ScrollBar instance will transfer focus to the TileList component. While focused, pressing the keyboard up and down arrows, or equivalent controls, scrolls the list selection vertically by a single element if it contains multiple rows, and the left and right arrows, or equivalent controls, scrolls

the list selection horizontally if it contains multiple columns. If the TileList contains multiple rows and columns, then all four arrow keys or equivalent controls can be used to navigate list selection. If no element is selected, the topmost element is automatically selected for this action. The mouse wheel scrolls the list if the cursor is on top of the TileList boundary.

Pressing the keyboard (Page Up) and (Page Down) keys or equivalent controls will scroll the selection by a page, i.e., the number of visible rows in the list. Pressing the (Home) and (End) keys or equivalent controls will scroll the list to the first and last elements respectively. Interacting with an attached ScrollBar component will affect the ScrollingList as expected. See the ScrollBar section to learn about its own user interaction.

2.4.5.2 Component setup

The TileList does not require any named subelements. However, a visible background is helpful when placing and resizing an instance of the TileList component on the Stage.

2.4.5.3 States

The TileList component supports three states based on its focused and disabled properties:

- a ***default*** or enabled state;
- a ***focused*** state, that typically highlights the component's border area;
- a ***disabled*** state.

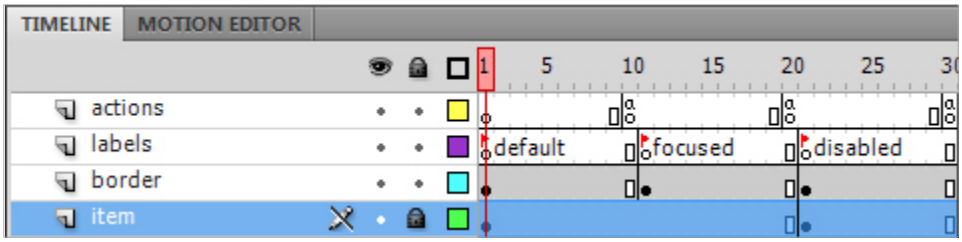

Figure 42: TileList timeline.

2.4.5.4 Inspectable properties

A MovieClip that derives from the TileList component will have the following inspectable properties:

visible	Hides the component if set to false. This does not hide the attached scrollbar or any external list item renderers.
disabled	Disables the component if set to true. This does disable both the attached scrollbar and the list items (both internally created and external renderers).
itemRenderer	The symbol name of the ListItemRenderer. Used to create list item

	instances internally. Has no effect if the *rendererInstanceName* property is set.
renderInstanceName	Prefix of the external list item renderers to use with this TileList component. The list item instances on the Stage must be prefixed with this property value. If this property is set to the value 'r', then all list item instances to be used with this component must have the following values: 'r1', 'r2', 'r3',… The first item should have the number 1.
scrollBar	Instance name of a ScrollBar component on the Stage or a symbol name. If an instance name is specified, then the TileList will hook into that instance. If a symbol name is specified, an instance of the symbol will be created by the TileList.
margin	The margin between the boundary of the list component and the list items created internally. This value has no effect if the *rendererInstanceName* property is set.
rowHeight	The height of list item instances created internally. This value has no effect if the *rendererInstanceName* property is set.
columnWidth	The width of list item instances created internally. This value has no effect if the *rendererInstanceName* property is set.
externalColumnCount	When the *rendererInstanceName* property is set, this value is used to notify the TileList of the number of columns used by the external renderers.
direction	The scrolling direction. The semantics of rows and columns do not change depending on this value.

2.4.5.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the TileList component are listed below. The properties listed next to the event are provided in addition to the common properties.

show	The component's visible property has been set to true at runtime.
hide	The component's visible property has been set to false at runtime.
focusIn	The component has received focus.
focusOut	The component has lost focus.
change	The selected index has changed.

	• *index*: The new selected index. Number type. Values 0 to number of list items minus 1.
itemPress	A list item has been pressed down. • *renderer*: The list item that was pressed. CLIK Button type. • *item*: The data associated with the list item. This value is retrieved from the list's dataProvider. AS2 Object type. • *index*: The index of the list item relative to the list's dataProvider. Number type. Values 0 to number of list items minus 1. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
itemClick	A list item has been clicked. • *renderer*: The list item that was clicked. CLIK Button type. • *item*: The data associated with the list item. This value is retrieved from the list's dataProvider. AS2 Object type. • *index*: The index of the list item relative to the list's dataProvider. Number type. Values 0 to number of list items minus 1. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
itemDoubleClick	A list item has been double clicked. • *renderer*: The list item that was double clicked. CLIK Button type. • *item*: The data associated with the list item. This value is retrieved from the list's dataProvider. AS2 Object type. • *index*: The index of the list item relative to the list's dataProvider. Number type. Values 0 to number of list items minus 1. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
itemRollOver	The mouse cursor has rolled over a list item. • *renderer*: The list item that was rolled over. CLIK Button type. • *item*: The data associated with the list item. This value is retrieved from the list's dataProvider. • *index*: The index of the list item relative to the list's dataProvider. Number type. Values 0 to number of list items minus 1. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.

itemRollOut	The mouse cursor has rolled out of a list item. • *renderer*: The list item that was rolled out of. CLIK Button type. • *item*: The data associated with the list item. This value is retrieved from the list's dataProvider. AS2 Object type. • *index*: The index of the list item relative to the list's dataProvider. Number type. Values 0 to number of list items minus 1. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.

The following example shows how to determine when the TileList has received focus:

```
myList.addEventListener("focusIn", this, "onListFocused");
function onListFocused(event:Object) {
      trace("tile list was focused!");
      // Do something
}
```

2.4.5.6 Tips and tricks

Displaying a single label from a set of complex objects:

```
// The TileList automatically will use the property named 'label'
// if found in the item object:
list.dataProvider = [{label: "one", data:1}, {label: "two", data:2}];

// However if the item object has a different label property, such as
// the following, then the list can be configured to use that property
// instead:
list.labelField = "name";
list.dataProvider = [{name: "one", data:1}, {name: "two", data:2}];

// If the logic to construct a label from the item object is
// complicated and requires a function, then set the labelFunction
// property:
list.labelFunction = function(itemObj:Object):String {
    // Logic to construct a label
}
list.dataProvider = [{p1: "foo", p2: 1}, {p1: "bar", p2: 2}];
```

2.4.6 DropdownMenu

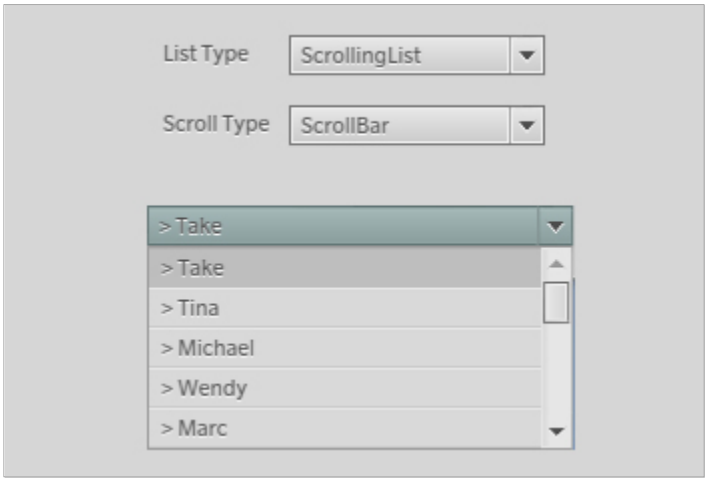

Figure 43: Unskinned DropdownMenu.

The DropdownMenu (gfx.controls.DropdownMenu) wraps the behavior of a button and a list. Clicking on this component opens a list that contains the elements to be selected. The DropdownMenu displays only the selected element in its idle state. It can be configured to use either the ScrollingList or the TileList, to which either a ScrollBar or ScrollIndicator can be paired with. The list is populated via an installed dataProvider. The DropdownMenu's list element is populated via a dataProvider. The dataProvider is assigned via code, as shown in the example below:

```
dropdownMenu.dataProvider = ["item1", "item2", "item3", "item4"];
```

By default the DropdownMenu uses the ScrollingList component for its contents. Therefore the ScrollingList and ListItemRenderer must also exist in the FLA file's Library for it to work, unless the dropdown inspectable property is changed to another component. See the Inspectable properties section for more information.

Also note that by default the DropdownMenu does not attach a scrollbar to its list element. The ScrollBar or ScrollIndicator must be attached via code to the DropdownMenu's list element. See the Tips and tricks section for more information.

2.4.6.1 User interaction

Clicking on a DropdownMenu instance or pressing the (Enter) key or equivalent control will open the list of selectable elements. Focus is also transferred to the list when it is opened. The user can interact with the list as described in the User interaction sections of the ScrollingList, TileList and ScrollBar. Clicking on a list item will select it, close the list and display the selected item in the DropdownMenu component. Clicking outside

the list boundary will automatically close the list and focus will be transferred back to the DropdownMenu component.

2.4.6.2 Component setup

The DropdownMenu derives most of its functionality from the Button component. Consequently a MovieClip that uses the DropdownMenu class must have the following named subelements. The list and scrollbar are created dynamically. Optional elements are noted appropriately:

- ***textField***: (optional) TextField type. The button's label.
- ***focusIndicator***: (optional) MovieClip type. A separate MovieClip used to display the focused state. If it exists, this MovieClip must have two named frames: "show" and "hide".

2.4.6.3 States

The DropdownMenu is toggled when opened, and therefore needs the same states as a ToggleButton or CheckBox that denote the selected state. These states include:

- an ***up*** or default state;
- an ***over*** state when the mouse cursor is over the component, or when it is focused;
- a ***down*** state when the button is pressed;
- a ***disabled*** state;
- a ***selected_up*** or default state;
- a ***selected_over*** state when the mouse cursor is over the component, or when it is focused;
- a ***selected_ down*** state when the button is pressed;
- a ***selected_disabled*** state.

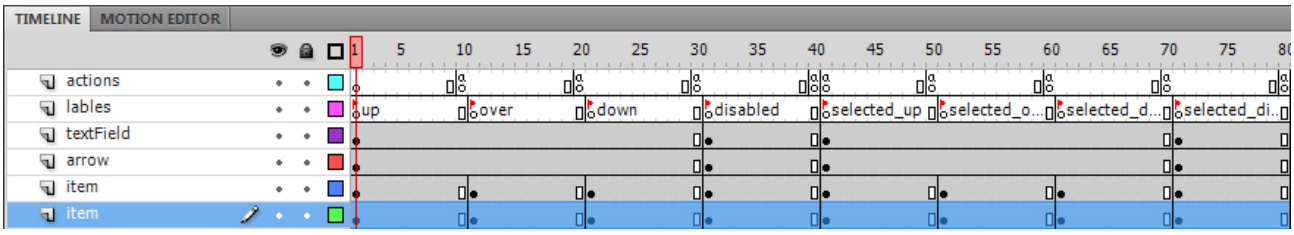

Figure 44: DropdownMenu timeline.

This is the minimal set of keyframes that should be in a DropdownMenu. The extended set of states and keyframes supported by the Button component, and consequently the DropdownMenu component, are described in the Getting Started with CLIK Buttons document.

2.4.6.4 Inspectable properties

The inspectable properties of the DropdownMenu component are:

visible	Hides the component if set to false.
disabled	Disables the component if set to true.
dropdown	Symbol name of the list component (ScrollingList or TileList) to use with the DropdownMenu component.
dropdownWidth	Width of the dropdown list. If this value is -1, then the DropdownMenu will size the list to the component's width.
itemRenderer	Symbol name of the dropdown list's item renderer. Created by the dropdown list instance.
scrollBar	Symbol name of the dropdown list's scrollbar. Created by the dropdown list instance. If empty, then the dropdown list will have no scrollbar.
margin	The margin between the boundary of the list component and the list items created internally. This margin also affects the automatically generated scrollbar.
paddingTop	Extra padding at the top for the list items. Does not affect the automatically generated scrollbar.
paddingBottom	Extra padding at the bottom for the list items. Does not affect the automatically generated scrollbar.
paddingLeft	Extra padding on the left side for the list items. Does not affect the automatically generated scrollbar.
paddingRight	Extra padding on the right side for the list items. Does not affect the automatically generated scrollbar.
thumbOffsetTop	Scrollbar thumb top offset. This property has no effect if the list does not automatically create a scrollbar instance.
thumbOffsetBottom	Scrollbar thumb bottom offset. This property has no effect if the list does not automatically create a scrollbar instance.
thumbSizeFactor	Page size factor for the scrollbar thumb. A value greater than 1.0 will increase the thumb size by the given factor. This value has no effect if a scrollbar is not attached to the list.
offsetX	Horizontal offset of the dropdown list from the dropdown button position. A positive value moves the list to the right of the dropdown button horizontal position.
offsetY	Vertical offset of the dropdown list from the dropdown button. A positive value moves the list away from the button.
extent	An extra width offset that can be used in conjunction with offsetX. This value has no effect if the dropdownWidth property is set to a value other than -1.
direction	The list open direction. Valid values are "up" and "down".

2.4.6.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the DropdownMenu component are listed below. They are the same as the Button component, with the exception of the *change* event. The properties listed next to the event are provided in addition to the common properties.

Show	The component's visible property has been set to true at runtime.
Hide	The component's visible property has been set to false at runtime.
focusIn	The component has received focus.
focusOut	The component has lost focus.
change	The selected index has changed. • *index*: The new selected index. Number type. Values 0 to number of list items minus 1. • *data*: The data associated with the list item at the selected index. AS2 Object type.
select	The component's selected property has changed. • *selected*: The selected property of the Button. Boolean type.
stateChange	The button's state has changed. • *state*: The button's new state. String type, Values "up", "over", "down", etc. See Getting Started with CLIK Buttons document for full list of states.
rollOver	The mouse cursor has rolled over the button. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
rollOut	The mouse cursor has rolled out of the button. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
Press	The button has been pressed. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.

doubleClick	The button has been double clicked. Only fired when the *doubleClickEnabled* property is true. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
Click	The button has been clicked. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
dragOver	The mouse cursor has been dragged over the button (while the left mouse button is pressed). • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
dragOut	The mouse cursor has been dragged out of the button (while the left mouse button is pressed). • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.
releaseOutside	The mouse cursor has been dragged out of the button and the left mouse button has been released. • *mouseIndex*: The index of the mouse cursor used to generate the event (applicable only for multi-mouse cursor environments). Number type. Values 0 to 3.

2.4.6.6 Tips and tricks

Determine if the drop down menu is open or closed:

```
    dropdown.addEventListener("click", this, "onClick");
function onClick(e:Object) {
    if (e.target.isOpen) {
        // Do something when open
    }
}
```

Creating a DropdownMenu dynamically at runtime:

```
    // Make sure the linkage IDs used in the code are available;
    // meaning the referenced symbols/components exist in the .fla library.
    // Example: requiring the DropdownMenu, ScrollingList and ScrollBar.
```

```
    attachMovie("DropdownMenu", "dd", this.getNextHighestDepth(),
                                      {dropdown: "ScrollingList"});
    dd.dataProvider = ["one", "two", "three", "four", "five", "six"];

    // Installing a scroll bar or scroll indicator to the dropdown's
    // list requires a delayed property set using a trick presented in
    // the following code. The delay is required to allow the dropdown to
    // instantiate the list first, before attaching a scroll bar to the list:
  onEnterFrame = function() {
      dd.dropdown.scrollBar = "ScrollBar";
      onEnterFrame = null;
    }
```

Displaying a single label from a set of complex objects:

```
    // The DropdownMenu automatically will use the property named 'label'
    // if found in the item object:
    list.dataProvider = [{label: "one", data:1}, {label: "two", data:2}];

    // However if the item object has a different label property, like below
    // then the list can be configured to use that property instead:
    list.labelField = "name";
    list.dataProvider = [{name: "one", data:1}, {name: "two", data:2}];

    // If the logic to construct a label from the item object is
    // complicated and requires a function, then set the labelFunction:
    list.labelFunction = function(itemObj:Object):String {
      // Logic to construct a label
  }
  list.dataProvider = [{p1: "foo", p2: 1}, {p1: "bar", p2: 2}];
```

2.5 Progress Types

The progress types are used to display the status or progress of an event or action. The Scaleform CLIK framework contains two components that belong to this category, the StatusIndicator and ProgressBar. The StatusIndicator component is used to display the status of an event or action. While the ProgressBar component has the same semantics as the StatusIndicator, but it includes additional functionality that listens to other components or actions that generate progress events.

Figure 45: Faction status indicator example from *Mercenaries 2*.

2.5.1 StatusIndicator

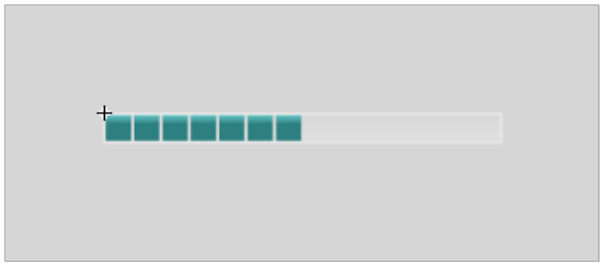

Figure 46: Unskinned StatusIndicator.

The StatusIndicator component (gfx.controls.StatusIndicator) displays the status of an event or action using its timeline as the visual indicator. The value of the StatusIndicator will be interpolated with the minimum and maximum values to generate a frame number that will be played in the component's timeline. Since the component's timeline is used to display the status, it provides absolute freedom in creating innovative visual indicators.

2.5.1.1 User interaction

The StatusIndicator does not have any user interaction.

2.5.1.2 Component setup

A MovieClip that uses the CLIK StatusIndicator class does not require any named subelements. However, the StatusIndicator is required to have at least two frames for correct operation. Make sure to insert a stop() command in the first frame to avoid playing the frames. The StatusIndicator component will gotoAndStop() on the relevant frame generated by its value property.

2.5.1.3 States

There are no states for the StatusIndicator component. The component's frames are used to display the status of an event or action.

2.5.1.4 Inspectable properties

A MovieClip that derives from the StatusIndicator component will have the following inspectable properties:

Visible	Hides the component if set to false.
disabled	Disables the component if set to true.
Value	The status value of an event or action. It is interpolated between the minimum and maximum values to generate a frame number to be played.
minimum	The minimum value used to interpolate the target frame.
maximum	The maximum value used to interpolate the target frame.

2.5.1.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

No special events are generated by the StatusIndicator. The properties listed next to the event are provided in addition to the common properties.

Show	The component's visible property has been set to true at runtime.
Hide	The component's visible property has been set to false at runtime.

2.5.2 ProgressBar

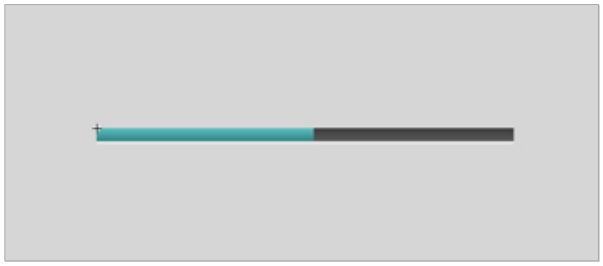

Figure 47: Unskinned ProgressBar.

The ProgressBar (gfx.controls.ProgressBar) is similar to the StatusIndicator in that it also displays the status of an event or action using its timeline, however it is also intended to be used in conjunction with a component or event that generates progress events. By assigning a target and setting its mode appropriately, the ProgressBar component will automatically change its visual state based on the loaded values (bytesLoaded and bytesTotal) of its target.

2.5.2.1 User interaction

The ProgressBar does not have any user interaction.

2.5.2.2 Component setup

Similar to the StatusIndicator, a MovieClip that uses the CLIK ProgressBar class does not require any named subelements. However the ProgressBar is required to have at least two frames for correct operation. Make sure to insert a stop() command in the first frame to avoid playing the frames. The ProgressBar component will gotoAndStop() on the relevant frame generated by its value property.

2.5.2.3 States

There are no states for the ProgressBar component. The component's frames are used to display the status of an event or action.

2.5.2.4 Inspectable properties

A MovieClip that derives from the ProgressBar component will have the following inspectable properties:

Visible	Hides the component if set to false.
disabled	Disables the component if set to true.
Target	The target the ProgressBar will "listen" to, to determine the bytesLoaded and bytesTotal values.
Mode	Listening mode of the ProgressBar. In "manual" mode, the progress values must be

	set using the setProgress method. In "polled" mode, the target must expose bytesLoaded and bytesTotal properties, and in "event" mode, the target must dispatch "progress" events, containing bytesLoaded and bytesTotal properties.

2.5.2.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the ProgressBar component are listed below. The properties listed next to the event are provided in addition to the common properties.

Show	The component's visible property has been set to true at runtime.
Hide	The component's visible property has been set to false at runtime.
progress	Generated when the ProgressBar value changes.
complete	Generated when the ProgressBar value is equal to its maximum.

The following example shows how to listen for progress events:

```
myProgress.addEventListener("progress", this, "onProgress");
myProgress.addEventListener("complete", this, "onProgress");
function onProgress(event:Object) {
      if (event.type == "progress") {
            // Do something
      } else
      {
            trace("Loading complete!");
      }
}
```

2.6 Other Types

The Scaleform CLIK framework also consists of several components that cannot be easily categorized, but nevertheless provide valuable functionality for UI developers. They are the Dialog, UILoader and ViewStack components. The Dialog component allows the display of modal and modal-free dialog boxes. The UILoader provides a convenient interface to load content, and the ViewStack can be used to manage a set of forms.

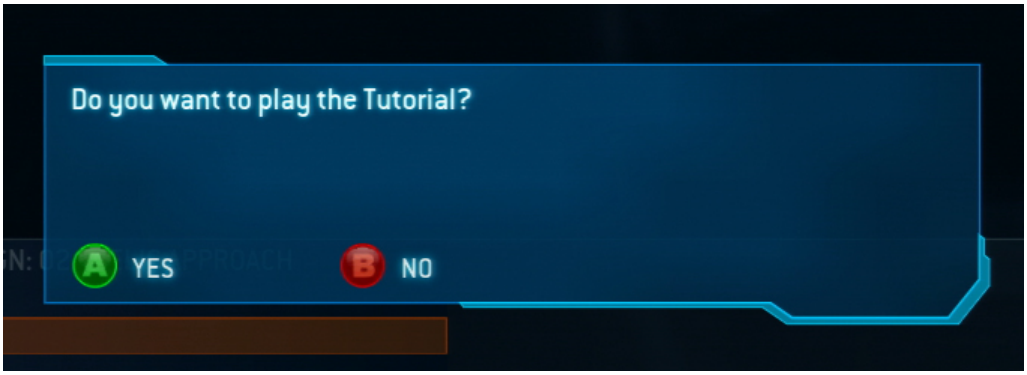

Figure 48: Dialog example from *Halo Wars*.

2.6.1 Dialog

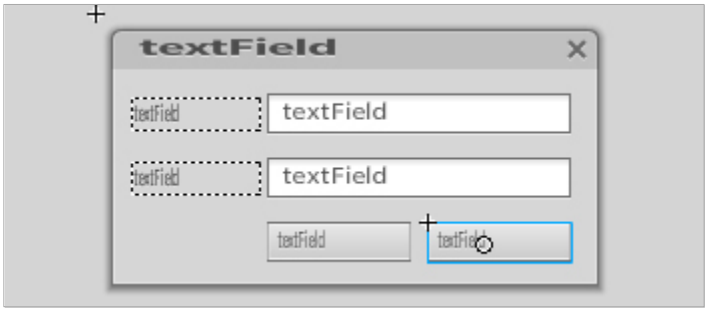

Figure 49: Sample unskinned Dialog.

The CLIK Dialog component (gfx.controls.Dialog) displays a dialog view, such as an Alert dialog, on top of the rest of the application. It provides a static interface to instantiate, show and hide any MovieClip as a dialog, as well as a base class that can be used (extended) as the actual dialog MovieClip. To ensure only a single dialog is open at once, new Dialog.show() calls will close the currently open dialog.

Note that the prebuilt component does not have any content since it is designed to be completely user defined. However, it is quite easy to add content to it by simply editing the component symbol.

2.6.1.1 User interaction

The user interaction of a Dialog is defined inside the dialog view it creates.

2.6.1.2 Component setup

A MovieClip that uses the CLIK Dialog class must have the following named subelements. Optional elements are noted appropriately:

- ***closeBtn***: (optional) CLIK Button type. Similar to a window close button.
- ***cancelBtn***: (optional) CLIK Button type. The cancel button that closes the dialog without submitting.
- ***submitBtn***: (optional) CLIK Button type. The OK button that closes the dialog after submitting.
- ***dragBar***: (optional) MovieClip type. A draggable title bar for the dialog.

2.6.1.3 States

There are no states for the Dialog component. The MovieClip displayed as the dialog view may or may not have its own states.

2.6.1.4 Inspectable properties

A MovieClip that derives from the Dialog component will have the following inspectable properties:

Visible	Hides the component if set to false.
disabled	Disables the component if set to true.

2.6.1.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the Dialog component are listed below. The properties listed next to the event are provided in addition to the common properties.

Show	The component's visible property has been set to true at runtime.
Hide	The component's visible property has been set to false at runtime.
Submit	The submit button of the Dialog has been clicked. • *data*: The submitted data of the Dialog. The Dialog component's

	getSubmitData method is invoked for this property, and should be overridden by custom dialog boxes. The default value returned is true (Boolean). The return value of getSubmitData is AS2 Object.
Close	The close button of the Dialog has been clicked.

The following example shows how to handle a dialog submit event:

```
myDialog.addEventListener("submit", this, "onLoginSubmit");
function onLoginSubmit(event:Object) {
	trace("Received data from dialog: " + event.data);
}
```

2.6.2 UILoader

The CLIK UILoader (gfx.controls.UILoader) loads an external SWF/GFX or image using only the path. UILoaders also support auto-sizing of the loaded asset to fit in its bounding box. Asset loading is asynchronous if both Scaleform and the platform running it has threading support.

2.6.2.1 User interaction

There is no user interaction for the UILoader component. If a SWF/SCALEFORM file is loaded into the UILoader, then it may have its own user interaction.

2.6.2.2 Component setup

A MovieClip that uses the CLIK UILoader class must have the following named subelements. Optional elements are noted appropriately:

- ***bg***: (optional) MovieClip type. Background of the UILoader. It has no functional purpose, other than for visual representation of the parent component on the Stage. This background is removed at runtime.

2.6.2.3 States

There are no states for the UILoader component. If a SWF/SCALEFORM file is loaded into the UILoader, then it may have its own states.

2.6.2.4 Inspectable properties

A MovieClip that derives from the UILoader component will have the following inspectable properties:

Visible	Hides the component if set to false.
autoSize	If set to true, sizes the loaded to content to fit in the UILoader's bounds.
maintainAspectRatio	If true, the loaded content will be fit based on its aspect ratio inside the UILoader's bounds. If false, then the content will be stretched to fit the UILoader bounds.
Source	The SWF/GFX or image file name to load.
timeout	Timeout in number of milliseconds to wait for the content to be loaded. If the timeout is reached and the content has not been loaded, an 'ioError' event will be generated by the UILoader.

2.6.2.5 Events

All event callbacks receive a single Object parameter that contains relevant information about the event. The following properties are common to all events.

- ***type***: The event type.
- ***target***: The target that generated the event.

The events generated by the UILoader component are listed below. The properties listed next to the event are provided in addition to the common properties.

Show	The component's visible property has been set to true at runtime.
Hide	The component's visible property has been set to false at runtime.
progress	Content is in the process of being loaded regardless whether the content can or cannot be loaded. This event will be fired continuously until either a) the content is loaded or b) the loading timeout has been reached.
Loaded	The percentage of data loaded. This property's value is between 0 and 100.
complete	Content loading has been completed.
ioError	Content specified in the source property could not be loaded.

The following example shows how to be notified on a loading error:

```
myUILoader.addEventListener("ioError", this, "onLoadingError");
function onLoadingError(event:Object) {
      displayErrorMessage("Could not load" + event.target.source);
}
```

2.6.3 ViewStack

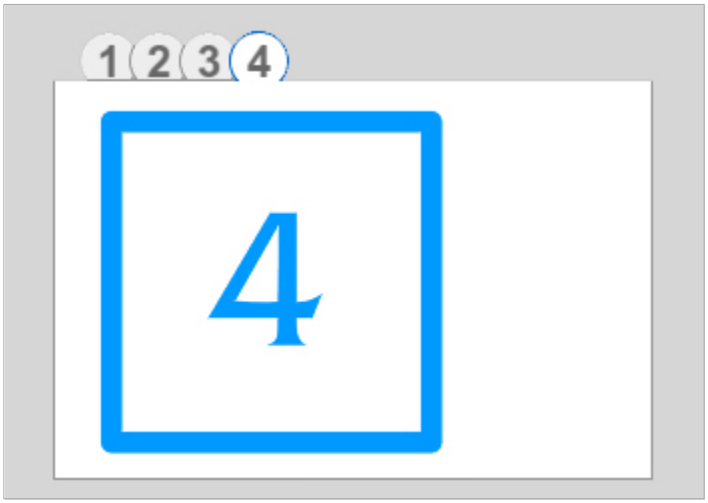

Figure 50: Unskinned ViewStack.

The CLIK ViewStack (gfx.controls.ViewStack) displays a single view from a set that is loaded and optionally cached internally. This component can be used for multiview components such as a TabBox, or Accordion (samples of each of these examples can be found in the demos folder). A ViewStack can also be pointed at another component such as a RadioButton group to automatically change views when the component changes.

2.6.3.1 User interaction

There is no user interaction for the ViewStack component. Views loaded and displayed by the ViewStack may have their own user interaction.

2.6.3.2 Component setup

A MovieClip that uses the CLIK ViewStack class does not require any named subelements. It however must be sized accordingly to fit the contents to be loaded.

2.6.3.3 States

There are no states for the ViewStack component. Views loaded and displayed by the ViewStack may have their own states.

2.6.3.4 Inspectable properties

A MovieClip that derives from the ViewStack component will have the following inspectable properties:

Visible	Hides the component if set to false.
Cache	If set to true, loaded views will be cached internally. This saves on processing time to create the views, but requires an immutable ViewStack targetGroup (see below).
targetGroup	A name of a valid group object, such as ButtonGroup, that generates 'change' events. The current element in the group object must have a data property containing a linkage ID for the view to be loaded and displayed. RadioButtons, for example, have a data property that can be assigned a linkage ID via the Flash IDE Component Inspector.

2.6.3.5 Events

The ViewStack does not produce any events.

3 Art Details

This section will aid artists to develop skins for Scaleform CLIK components and includes details on skinning a small sample of components as well as best practices, animation and font embedding.

3.1 Best Practices

This section contains best practices when creating skins for Scaleform CLIK components.

3.1.1 Pixel-perfect Images

When developing vector based skins for CLIK components, it is recommended that all assets be pixel perfect. A pixel perfect asset is one which lines up perfectly on the Flash grid.

To enable and modify the grid:

1. Select View from the top Flash menu.
2. Select Grid, then enable Show Grid by clicking on it.
3. Select View from the top Flash menu again.
4. Select Grid, then click on Edit Grid.
5. Enter 1 px for the vertical and horizontal distribution.
6. Press OK.

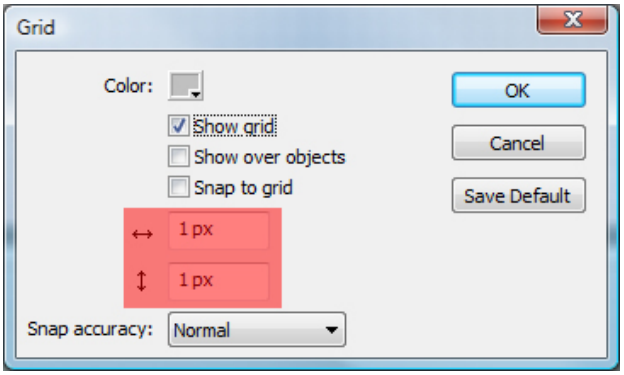

Figure 51: Edit Grid window.

You should now see a grid overlay on the Stage. Each grid square represents exactly one pixel. Be sure when creating art assets that they snap to this grid. This will ensure final SWFs which are not blurry. **NOTE:** Be sure to keep all image sizes to a power of two; otherwise, they may still be blurry. See the Power of two subheading below.

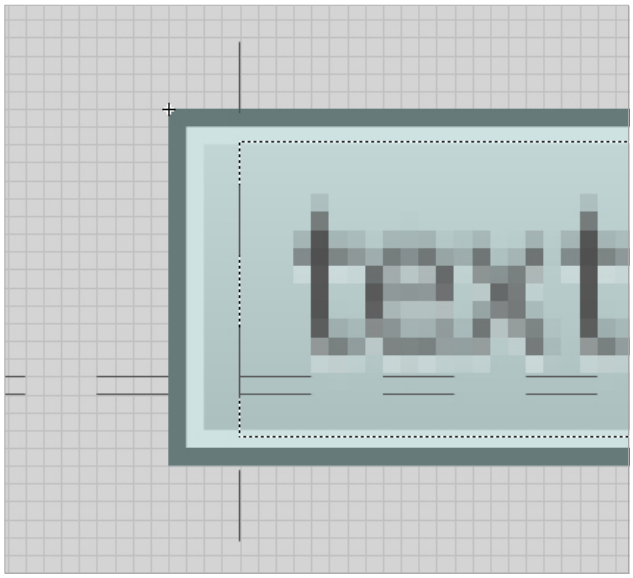

Figure 52: A pixel-perfect vector graphic skin.

3.1.2 Masks

Masks in Flash allow artists to hide parts of a graphic at runtime. Masks are often used for animation effects. However, masks can be very expensive at runtime. Scaleform recommends avoiding masks as much as possible. If masks must be used, keep them in the single digits and test the performance of the movie with and without the mask.

One possible alternative to using a mask in Flash is to create a PNG in Photoshop® with alpha blend to the areas that need to be masked out. However, this only works for an image in which there is no animation of the transparent area.

3.1.3 Animations

It is best to avoid animations that morph one vector shape into another, such as a morph of a square into a circle. These types of animations are very costly, as the shape must be reevaluated every frame.

Avoid scaling animations for vector graphics if possible, as they can have an impact on memory and performance due to extra tessellations – the process of converting vector shapes into triangles. The least expensive animations to use are translations and rotations, as they do not require extra tessellation.

Avoid programmatic tweens and opt for timeline based tweens as timeline tweens are much cheaper in terms of performance. For timeline tweens, try to keep the number of keyframes to the minimum required to achieve a smooth animation at the desired framerate.

3.1.4 Layers and Draw Primitives

Try to use as few layers as possible when creating a skin in Flash. Every layer used adds at least one draw primitive. The more draw primitives used, the more memory is required, and performance takes a hit as well.

3.1.5 Complex Skins

It is best to use bitmaps for complex skins; however, for simpler skins vector graphics will save more memory and be scalable to any resolution without a loss in quality (blurring).

3.1.6 Power of Two

Ensure that all bitmaps remain power of two in size. Examples of power of two-bitmap sizes are:

- 16x16
- 32x32
- 64x64
- 128x128
- 128x256
- 256x256
- 512x256
- 512x512

3.2 Known Issues and Recommended Workflow

This section contains a list of known art-related Flash issues and workflow recommendations when working with Scaleform CLIK.

3.2.1 Duplicating Components

There are issues when duplicating components in Flash, which cause the linkage information and component definition information to not be copied into the new component form the original one. As such, any component without this linkage information will not function. This section details two methods to work around this.

3.2.1.1 Duplicating components (method 1)

The quickest method for duplicating an unmodified (unksinned) CLIK component, such as Button, from an external FLA file into a destination FLA file is as follows:

1. Open the *CLIK_Components.fla* file.

2. Copy the component (e.g., Button) from that file's Library into the destination FLA by right clicking on it and choosing *Copy*, then right click in the Library of the destination FLA and choose *Paste*.
3. Right click on the component in the destination FLA Library and choose *Rename* to rename it to something other than 'Button'.
4. Right click on the component in the destination FLA Library again, and select *Properties*.
5. Change the *Identifier* field to match the new name chosen for the component in step 3.
6. Right click on a blank area in the Library window and choose *Paste*. A new copy of the original component will be pasted in with all linkage information intact.

3.2.1.2 Duplicating components (method 2)

When duplicating a symbol (component) in the same library, the linkage information will not be copied to the new duplicate symbol. This information will need to be entered in order for the component to function. To do this:

1. Right click the component to be duplicated in the *library* pane and select *Properties*.
2. In the *Properties* window, highlight the *Class* textfield by double clicking the text (e.g.: gfx.controlsButton) and press (CTRL+C) on the keyboard to copy it.
3. Press *Cancel*.
4. Right click the component to be duplicated again, and select *Duplicate*.
5. In the *Duplicate Symbol* window, click on the *Export for ActionScript* check box to enable it.
6. Click on the *Class* field and press (CTRL+V) to paste the linkage information into this textField.
7. Press *OK*.
8. Right click on the new copy of the component in the *library* pane and select *Component Definition*.
9. Click on the blank *Class* textField and press (CTRL+V) to paste the linkage information into the textField.
10. Press *OK*.

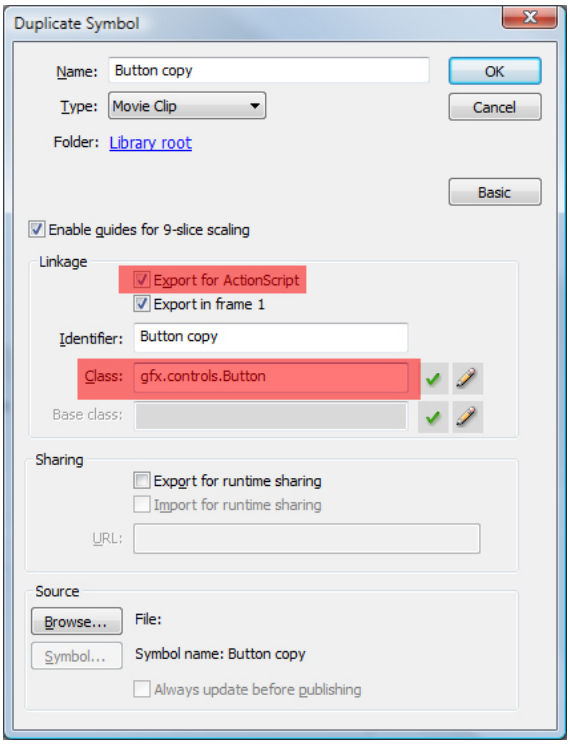

Figure 53: Duplicate Symbol Linkage (must fill in red highlighted areas).

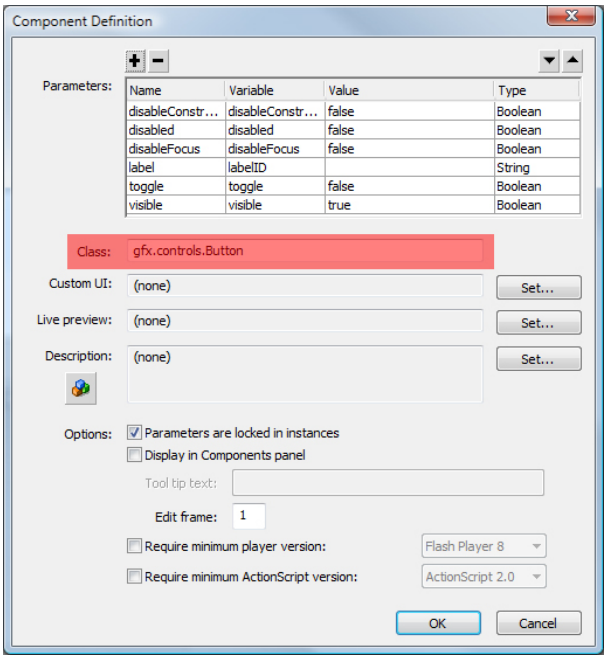

Figure 54: Component Definition (class must be filled in).

3.3 Skinning Examples

The major advantage of Scaleform CLIK is the separation of the visual and functional layers. This separation for the most part allows programmers and artists to work independently from each other. Easily customizing the look and feel is one of the primary benefits of this separation.

Although the prebuilt CLIK components have a standard look and feel, they are meant to be customized by users to match their own needs. The following sections describes the approaches one can take to customize the prebuilt components

The CLIK component set is as easy to skin as any standard Flash symbol. Simply double click a component in a FLA's library to gain access to the component's timeline and either:

- modify the default skin in Flash;
- create a custom skin from scratch in Flash; or
- import art assets created in Photoshop or Illustrator® into Flash.

Please review the document Getting Started with CLIK for a detailed skinning tutorial.

3.3.1 Skinning a StatusIndicator

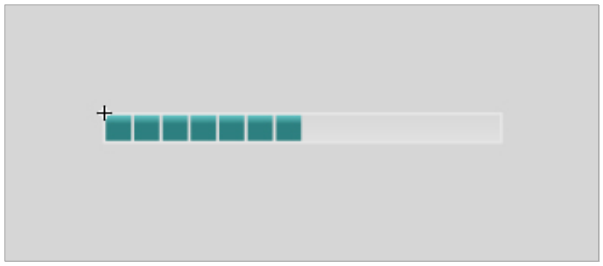

Figure 55: Unskinned StatusIndicator.

The StatusIndicator is a unique component, requiring a slightly different approach to skinning than most of the other components which are based upon Buttons. Open the StatusIndicator component, and take note of the timeline. There are two layers: *indicator* and *track*. *Track* is used to visually represent the background graphic; it serves no other purpose. The *indicator* layer uses several keyframes and either bitmap or vector graphics to represent the status in incremental steps from lowest to highest.

This tutorial uses several bitmaps created in a single Photoshop file on multiple layers to represent the status indicator. This walk through provides one way to skin the StatusIndicator; however, there are other methods which may be used to create and assemble the graphics on Stage. The end result should remain the same in order for the StatusIndicator to function properly.

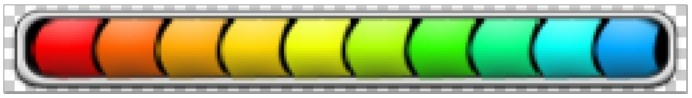

Figure 56: The StatusIndicator PSD file.

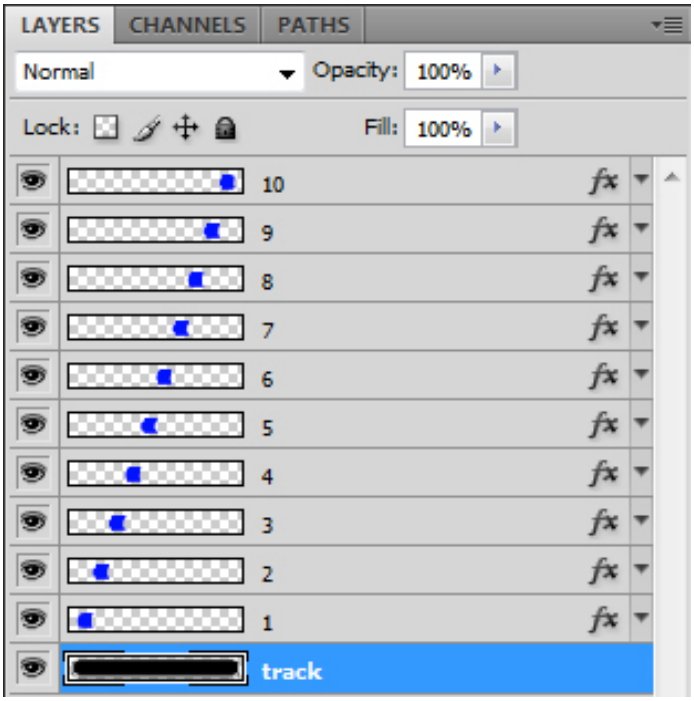

Figure 57: The layers setup in Photoshop for the StatusIndicator PSD file.

1. Create a StatusIndicator bitmap in Photoshop similar to the one pictured above. Take note of the layer order and numbering, as well as the position of each image on the layer. Ensure that the background is transparent. Create a similar file for this tutorial.
2. Save the file as a PSD.
3. In Flash, select *File* from the top Menu, and then choose *Import -> Import to Stage*.
4. Browse to and select the StatusIndicator skin PSD file.
5. In the *Import* window, ensure the *Convert layers to*: dropdown is set to 'Layers.'
6. Press *OK*.
7. A new Flash timeline layer should have been created for each layer in the PSD file. In the case of the image above, 10 layers were created, because the PSD file has 10 layers in it (one for each flame). Each layer was labeled from 1 to 10, with 1 being the bottom most layer and 10 being the topmost. Select *layer 1*.
8. Draw a selection box around all the bitmap images on the Stage.
9. Move the images into position over the old unskinned track, and scale to fit as necessary.
10. Select the first keyframe on *layer 1* and drag it to frame 6.
11. Add a new keyframe on *layer 1* at frame 11 by right clicking on that frame and selecting *Insert Keyframe*.
12. Select the bitmap on *layer 2* and press (Ctrl+X) to cut it.

13. Select the new keyframe at frame 11 of *layer 1* and press (Ctrl+Shift+V) to place the bitmap at the exact same spot on *layer 1*.
14. Repeat this process until all ten bitmaps are on the same layer (*layer 1*), at the correct keyframe. Each subsequent bitmap should be copied to a keyframe five frames after the last keyframe. The bitmap from *layer 2* should be copied to keyframe 11; the bitmap from *layer 3* should be copied to frame 16; the bitmap from *layer 4* should be copied to frame 21, etc. Using a 10 image PSD, the last keyframe should be on frame 51.
15. Delete the empty layers (2–10) to clean up.
16. If there are additional frames on the timeline after the last bitmap keyframe, count up another five keyframes, then select the remaining frames and delete them by first selecting them then right clicking on them and selecting *Remove Frames*. In the case of this tutorial, the last frame should be on frame 55.
17. Select the old *indicator layer* and delete it.
18. Select the old *track* layer and delete it. Be sure not to delete the new *track* layer that was imported.
19. Select *layer 1* and rename it to 'indicator.'
20. Drag the *indicator* layer and the *track* layer down below the *actions layer*, ensuring the *track* layer is below *indicator.*

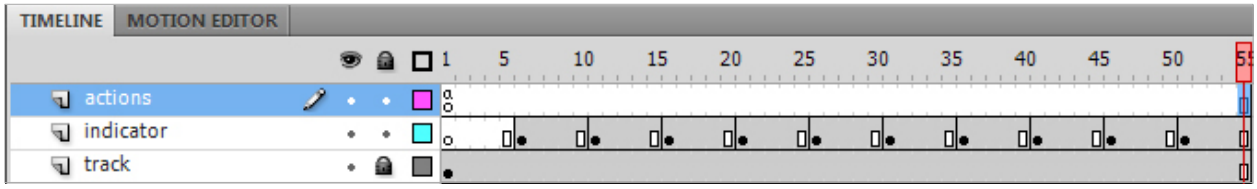

Figure 58: The final timeline (take note of the keyframe locations and final frame location).

21. Exit the timeline of the StatusIndicator.
22. Set the inspectable parameter value to any number between 1 and 10.
23. Save the file.
24. Publish the file to see the newly skinned indicator.

Figure 59: Skinned StatusIndicator.

3.4 Fonts and Localization

This section details font use in Scaleform CLIK components.

3.4.1 Overview

In order for fonts to render correctly in Scaleform, the required glyphs (font characters) must either be embedded in the SWFs or be set up using the Scaleform localization system. The following area explores ways to manage fonts in Flash UIs.

3.4.2 Embedding Fonts

Unlike the Flash Player, Scaleform requires fonts to be embedded properly for them to appear. Scaleform player will display empty rectangles if fonts are not embedded, even if they reside on the system. The benefit of this is you can easily see if your fonts are embedded correctly in Scaleform Scaleform. In the prebuilt component set, Slate Mobile is embedded in each textField instance. Note that Slate Mobile does not contain any Chinese, Japanese or Korean (CJK) characters or glyphs, and the prebuilt components only embed the ASCII glyphs. This can be changed by substituting Slate Mobile with a font containing CJK glyphs and setting the appropriate embedding options.

Note that embedding fonts directly into the textFields is not compatible with the Scaleform localization system (described in section 3.4.4). Scaleform in fact recommends using the Scaleform localization system for setting up fonts as it provides many benefits over direct embedding. However in a few cases, such as where localization is not required, embedding the fonts may be a better alternative. Similar to UI management, font management also requires several considerations such as localization and memory management.

3.4.3 Embedding Fonts in a textField

You only need to embed fonts in dynamic or input textFields. Static text is automatically converted to outlines (raw vector shapes) when you compile. To embed a font in a dynamic textField, select the textField on the Stage and click the *Embed* button in the property inspector (Window > Property Inspector). Choose the characters, or character sets, that should be embedded and choose *OK*. Once you have completed those steps, the font will be available for use in your FLA. You only need to embed a font this way once if the same font is used in multiple textFields in the same FLA. Conversely, embedding it multiple times will not increase the file size or memory usage. Note that character sets with a large number of glyphs such as Chinese can occupy a large memory footprint when loaded.

3.4.4 Scaleform Localization System

The Scaleform localization system uses a hot-swapping mechanism to load and unload font libraries whenever the current locale changes. These font libraries are themselves SWF files that contain embedded font glyphs that are used by the content SWFs. Scaleform is able to use glyphs from the font libraries to provide a powerful way to dynamically change fonts on demand.

Since the Scaleform localization system cannot swap fonts if a textField directly embeds the glyphs, the textFields must use an imported font symbol. Typically the font symbol is created in a file called gfxfontlib.fla and imported into the content swfs. This imported font is then set on all textFields that will support font swapping. Scaleform is now able to hijack this font symbol link and load different fonts based on the current locale.

The font libraries do not need to export any fonts. They should only embed the appropriate fonts (see the previous sections on how to embed fonts). The Scaleform localization system uses a fontconfig.txt file to define the font libraries to use per locale, as well as font mapping between the font symbols used by the textFields and the physical font embedded in the font libraries.

The fontconfig.txt file also contains the translation maps. The Scaleform localization system performs on the fly translation of text displayed in every dynamic or input textField on the Stage. For example, if a textField contains the text '$TITLE' and the translation map has a mapping such as $TITLE=Scaleform, then the textField will automatically display 'Scaleform' instead.

The information presented in this section is meant as a high level overview of the Scaleform font and localization system. To fully understand fonts and localization in Scaleform, please refer to the Font Overview document.

4 Programming Details

This chapter will describe the nuts and bolts of the framework and highlight each subsystem to provide a high-level understanding of the Scaleform CLIK component architecture.

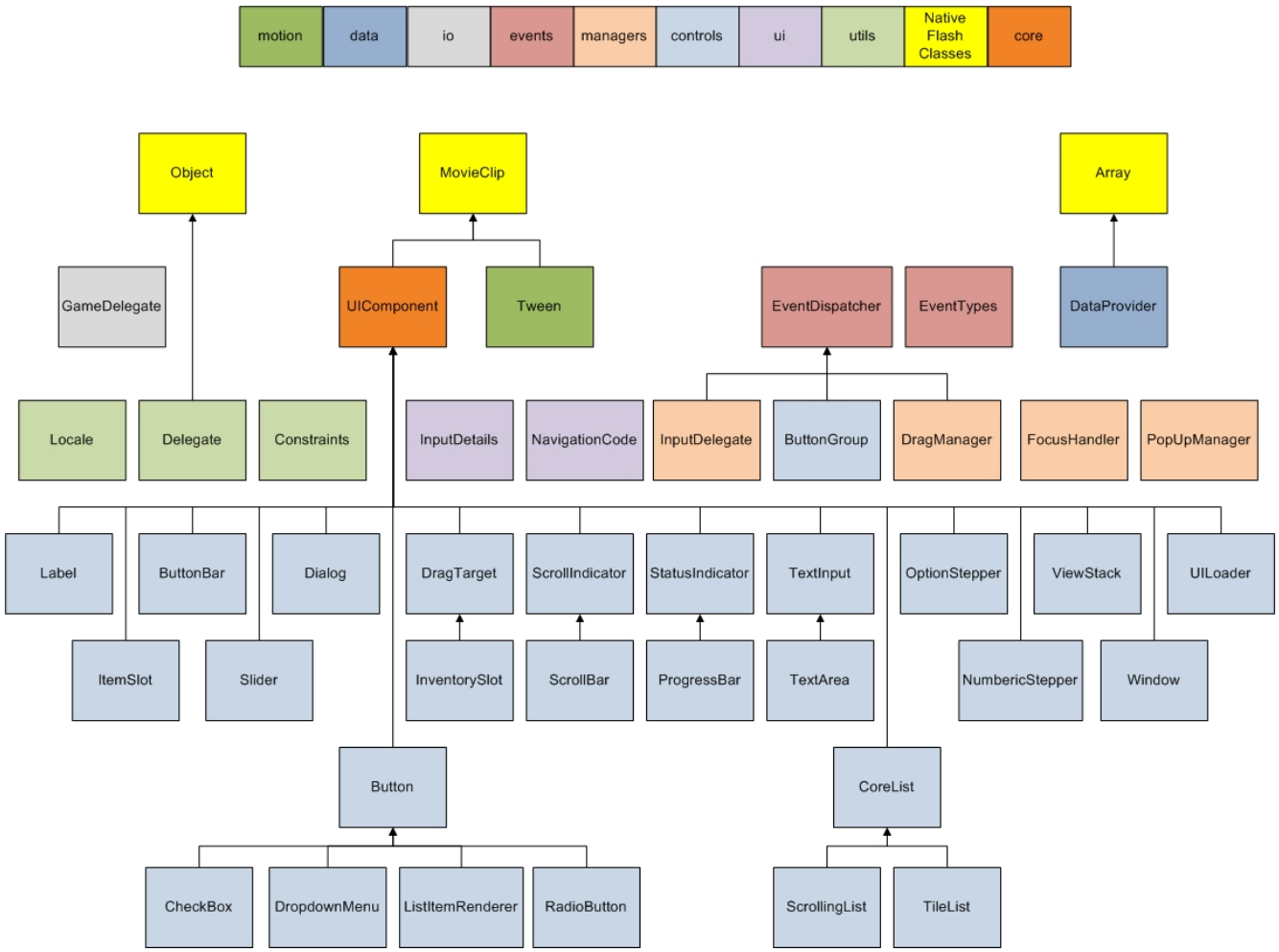

Figure 60: Scaleform CLIK class hierarchy.

4.1 UIComponent

The Prebuilt Components chapter described the classes used by the concrete components bundled with Scaleform CLIK. All of these components inherit core functionality from the UIComponent class (gfx.core.UIComponent). This class is the foundation of all CLIK components, and Scaleform recommends custom components be subclassed from UIComponent.

The UIComponent class itself derives from the Flash 8 MovieClip class, and thus inherits all properties and methods of a standard Flash MovieClip. Several custom read/write properties are supported by the UIComponent class:

- ***disabled***;
- ***visible***;
- ***focused***;
- ***width***;
- ***height***;
- ***displayFocus***: Set this property to true if the component should display its focused state. For more information see the Focus Handling section.

UIComponent has several empty methods that are meant to be implemented by subclasses. These include:

- ***configUI***: Called to perform component configuration.
- ***draw***: Called when the component is invalidated. For more information, see the Invalidation section.
- ***changeFocus***: Called when the component receives or loses focus.
- ***scrollWheel***: Called when the scroll wheel is used when the cursor is over the component.

The UIComponent mixes in the EventDispatcher class to support event subscribing and dispatching. Therefore, all subclasses support event subscribing and dispatching.
For more information, see the Event Model section.

4.1.1 Initialization

This class performs the following initialization steps inside an onLoad() event handler:

- Sets the default size to the MovieClip dimensions.
- Perform component configuration. This steps calls the configUI() method.
- Draws contents. This step calls the validateNow() method, which performs a redraw immediately, i.e., calls draw().

4.2 Component States

Almost all Scaleform CLIK components support visual states. States are set up by navigating to a specific keyframe in the component timeline, or passing state information onto subelements. There are three common state setups, detailed below.

4.2.1 Button Components

Any component that behaves similar to a button, which responds to mouse actions, and can optionally be selected fall into this category. Examples of CLIK components that use this schema are Button and its variants, ListItemRenderer, RadioButton and CheckBox. Subelements of complex components such as ScrollBar are also Button components. CLIK components that fall into this category either directly use the Button class (gfx.controls.Button) or use a class that is derived from the Button class.

The basic states supported by button components are:

- an ***up*** or default state;
- an ***over*** state when the mouse cursor is over the component, or when it is focused;
- a ***down*** state when the mouse has been pressed on the component;
- a ***disabled*** state when the component has been disabled.

Button components also support prefixed states, which can be set depending on the value of other properties. By default, the core CLIK Button component only supports a "selected_" prefix, which is appended to the frame label when the component is in a selected state.

The basic states supported by Button components, including selected states, are:

- an ***up*** or default state;
- an ***over*** state when the mouse cursor is over the component, or when it is focused;
- a ***down*** state when the mouse has been pressed on the component;
- a ***disabled*** state when the component has been disabled;
- a ***selected_up*** or default state;
- a ***selected_over*** state when the mouse cursor is over the component, or when it is focused;
- a ***selected_ down*** state when the button is pressed;
- a ***selected_disabled*** state.

Note: The states mentioned in this section are only a handful of the full complement of states supported by CLIK Button components. Please refer to the Getting Started with CLIK Buttons document for the complete list of states.

The CLIK Button class provides a getStatePrefixes() method, which allows developers to change the list of prefixes depending on the component's properties. This method is defined as follows:

```
private function getStatePrefixes():Array {
      return (_selected) ? ["selected_",""] : [""];
}
```

As mentioned earlier, the CLIK Button by default only supports the "selected_" prefix. The getStatePrefixes() method returns a different array of prefixes depending on its selected property. This prefix array will be used internally in conjunction with the appropriate state label to determine the frame to play.

When a state is set internally, for example on mouse rollover, a lookup table is queried for a list of frame labels. The stateMap property in the Button class defines the state to frame label mapping. The following is the state mapping defined in the CLIK Button class:

```
private var stateMap:Object = {
      up:["up"],
      over:["over"],
      down:["down"],
      release: ["release", "over"],
      out:["out","up"],
      disabled:["disabled"],
      selecting: ["selecting", "over"],
      kb_selecting: ["kb_selecting", "up"],
      kb_release: ["kb_release", "out", "up"],
      kb_down: ["kb_down", "down"]
}
```

Each state may have more than one target label. The values returned from the state map is combined with the prefix returned by getStatePrefixes() to generate a list of target frames to be played. The following figure describes the complete process used to determine the correct frame to play:

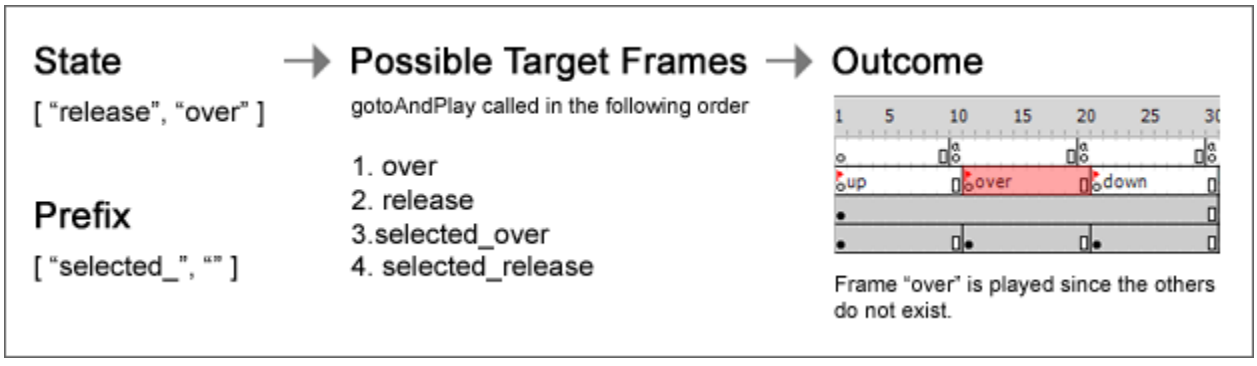

Figure 61: Process used to determine the correct keyframe to play.

The playhead will always jump to the last available frame, consequently if a certain prefixed frame is not available, the component will default to the previously requested frame. Developers can override the state map to create custom behaviors.

4.2.2 Non-button Interactive Components

These refer to any components that are interactive, and can receive focus, but do not respond to mouse events. Examples of CLIK components that use this schema are ScrollingList, OptionStepper, Slider and TextArea. These components may contain child elements that respond to mouse events. The states supported by the nonbutton interactive components are:

- *a **default*** state;
- a ***focused*** state;
- a ***disabled*** state.

4.2.3 Noninteractive Components

These refer to any component that is not interactive, but can be disabled. The Label component is the only noninteractive component in the default component set that support states. The states supported by the non-interactive components are:

- a ***default*** state;
- a ***disabled*** state.

4.2.4 Special Cases

There are several components which do not abide by the staterules described above. StatusIndicator, and its subclass ProgressBar, use the timeline to display the value of the component. The playhead will be set to the frame that represents the percentage value of the component. For instance, a 50-frame timeline in a StatusIndicator with a minimum value of 0, a maximum value of 10, and a current value set to 5 (50%) will gotoAndStop() on frame 25 (50% of 50 frames). It is fairly simple to extend these components to manage the display programmatically. The updateValue() method can be modified or overridden to change this behavior.

In some cases, components may have special modes in addition to their default behavior that provide support for additional component states. The TextInput and TextArea components enable the ***over*** and ***out*** states when their actAsButton property is set to support mouse cursor roll over and roll out events when not focused.

4.3 Event Model

The Scaleform CLIK component framework uses a communication paradigm known as the *event model*. Components "dispatch" events when they change or are interacted with, and container components can subscribe to the different events. This allows multiple objects to be notified of a change, instead of just one, which is how ActionScript 2 callbacks work.

The EventDispatcher class (gfx.events.EventDispatcher) provides an easy to use API that supports the event model. A CLIK component can either extend this class or use the EventDispatcher.initialize() method to mix in its behavior. However, if the CLIK component derives from UIComponent, then it only needs to call super() in its contructor since UIComponent already does a mix in of EventDispatcher. An EventDispatcher subclass or mix in provides support for subscribing and dispatching of events.

4.3.1 Usage and Best Practices

4.3.1.1 Subscribing to an Event

To subscribe to an event, use the addEventListener() method, with a type parameter that specifies the type of interaction to listen for. Conversely, removeEventListener() will unsubscribe from an event. If multiple listeners are added with the exact same parameters, only one event will be fired. Each of these methods also requires a scope parameter, which is the listening object, and a callBack, which is the String name of the function that is called when the event is dispatched.

The EventTypes class (gfx.events.EventTypes) contains an enumeration of commonly used events. They can be used instead of strings denoting the event type (EventTypes.SHOW instead of "show").

```
buttonInstance.addEventListener("itemClick", this, "callBack");
function callBack(eventObj:Object):Void {
      buttonInstance.removeEventListener("itemClick", this, "callBack");
}
```

In this example the buttonInstance is set to listen for "itemClick" events and to execute the function "callback" when it receives the event. The callback function then removes the event listener.

The properties of the event Object differs based on its origin and type. For a complete list of event objects (and their properties) generated by the CLIK components, please refer to the chapter on the Prebuilt Components.

4.3.1.2 Dispatching an Event

CLIK components that wish to notify subscribed listeners of a change or interaction use the dispatchEvent() method. This method requires a single argument: an object containing relevant data, including a mandatory

type property which specifies the type of event to be dispatched. The component framework automatically adds a target property, which is a reference to the object dispatching the event, but it can also be set manually to override it with a custom target.

```
dispatchEvent({type:"itemClick", item:selectedItem});
```

4.4 Focus Handling

The Scaleform CLIK components use a custom focus handling framework, which is implemented in most components, and should work well with nonframework components and symbols. All focus changes happen at the Scaleform player level, either by mouse or keyboard (gamepad) focus changes, or by calling Selection.setFocus(instance) in ActionScript. The FocusHandler manager (gfx.managers.FocusHandler) is instantiated as soon as a single component class is created. There is no need to instantiate the FocusHandler directly.

4.4.1 Usage and Best Practices

Currently focus is required to be set on a CLIK component instance; otherwise focus will default to the player. If not set, the focus management system will not behave correctly (such as the Tab key not switching focus, etc.). Focus can be applied by setting the focused property on any component to true. Another method that works to apply focus, which also works with non-component elements, is the following:

```
Selection.setFocus(firstComponentOrMovieClip);
```

MovieClips without mouse handlers (e.g., onRelease, onRollOver) cannot be focused by the Scaleform or Flash player, and will not generate focus change events. Button components automatically add mouse handlers. Other components will set a combination of the standard MovieClip focusEnabled and tabEnabled properties.

Components with focusable subelements, such as ScrollBar, pass focus onto their owner component using their focusTarget property. When focus changes to a component, the FocusHandler will recursively search the focusTarget chain, and give focus to the last component that doesn't return a focusTarget.

Sometimes a component needs to appear focused when it is not the actual focus of the player or application. The displayFocus property can be set to true to tell a component to behave as if it is focused. For example, when a Slider is focused, the track of a Slider needs to appear to also be focused. Note that the UIComponent.changeFocus() method is called on a component when the focused property changes.

```
function changeFocus():Void {
      track.displayFocus = _focused;
```

```
    }
```

Conversely, sometimes a component needs to be clickable, but not focusable, such as a draggable panel, or any other component that would benefit from mouse-only control. In this case, set the tabEnabled property false.

```
    background.tabEnabled = false;
```

4.4.2 Capturing Focus in Composite Components

Composite components are those that are made up of other components, such as a ScrollingList, OptionStepper, or ButtonBar. The component itself will likely have mouse handlers on the sub-components, but not have any mouse handlers of its own. This means that the Selection engine in Flash and Scaleform cannot focus the item and the built-in navigation support will have trouble identifying the component, and instead inspect its children in error.

To make the component behave as a single entity despite its composition, the following steps can be taken:

1. Set tabEnabled = false on all subcomponents that have mouse-handlers (such as the arrow buttons in OptionStepper).
2. Set the focusTarget property on all subcomponents that have mouse-handlers to the container component. Make sure to set focusEnabled = true in the container component.
3. If necessary set the displayFocus property in the subcomponents to true inside the container component's changeFocus method.

Now when the subcomponent is focused, the focus will be transferred to the container component.

4.5 Input Handling

Since the Scaleform CLIK components derive from the Flash 8 MovieClip class, they mostly behave the same as any other MovieClip instance when receiving user input. Mouse events are caught by components if they install mouse handlers. However, there is a major conceptual change in CLIK related to how keyboard or equivalent controller events are handled.

4.5.1 Usage and Best Practices

All non-mouse input is intercepted by the InputDelegate manager class (gfx.managers.InputDelegate). The InputDelegate converts input commands into InputDetails objects (gfx.ui.InputDetails) either internally, or by

requesting the value of an input command from the game engine. The latter involves modifying the InputDelegate.readInput() method to support the target application. Once InputDetails have been created, an input event is dispatched from the InputDelegate.

InputDetails consist of the following properties:

- a type, such as “key”;
- a code, such as the key code of the pressed key;
- a value, which is extra information about the input such as button vector, or key-press type. Key events are generated for both key up and key down actions, and the corresponding InputDetails’ value parameter will be either “keyUp” or “keyDown” respectively;
- a navEquivalent (navigation equivalent), which defines a human-readable navigation direction such as “up”, “down”, “left” or “right” if such a mapping is possible. The NavigationCode class (gfx.ui.NavigationCode) provides a handy enumeration of common navigation equivalents.

The FocusHandler listens for input events from the InputDelegate and passes them to the components through the focus path. The focused component is used to determine the focus path, which is a top-down list of components in the display-list hierarchy that implement the handleInput() method.

This method is called on the top-most component in the focus chain, and the InputDetails and pathToFocus array are passed as parameters.

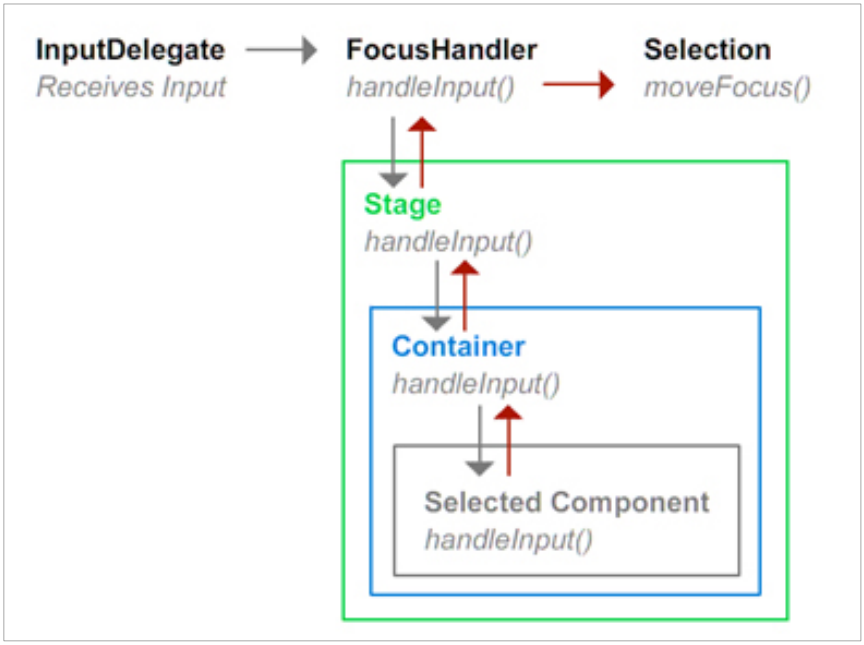

Figure 62: The handleInput() chain.

The FocusHandler expects a Boolean response to the handleInput() call, which indicates if the component, or any components in the focus path, has handled the input. If it receives a false response, and the input has a navEquivalent that is not null, then the input is passed on to the Scaleform player.

```
function handleInput(details:InputDetails, pathToFocus:Array):Boolean {
	if (details.navEquivalent == "left")
	{  // or NavigationCode.LEFT
		doSomething();
		return true;
	}
	return false;
}
```

It is up to each component handling input to bubble the event to the next component in the pathToFocus array, and to return true or false if the input is handled. It is a good idea not to assume that the next component in the focus path will implement handleInput correctly, therefore the method should ensure that it passes a Boolean value back and not just the return value of the bubbled input.

```
function handleInput(details:InputDetails, pathToFocus:Array):Boolean {
	var nextItem:MovieClip = pathToFocus.shift();
	var handled:Boolean = nextItem.handleInput(details, pathToFocus);
	if (handled) { return true; }
	// custom handling code
	return false; // or true if handled
}
```

Input can also be passed on to components not in the focus path. For instance, in DropdownMenu, the handleInput is passed on to the dropdown list component, even though it is not in the pathToFocus array. This enables the list to respond to key commands.

```
function handleInput(details:InputDetails, pathToFocus:Array):Boolean {
	var handled:Boolean = dropdown.handleInput(details);
	if (handled) { return true; }
	// custom handling code
	return false; // or true if handled
}
```

It is also possible to listen for the input event manually by adding an event listener to the InputDelegate.instance, and handle the input that way. Note that the input will still be captured by the FocusHandler and passed down the focus hierarchy.

```
InputDelegate.instance.addEventListener("input", this, "handleInput");
function handleInput(event:Object):Void {
	var details:InputDetails = event.details;
```

```
            if (details.value = Key.TAB) { doSomething(); }
      }
```

The InputDelegate should be tailored to the game to manage the expected input. The default InputDelegate handles keyboard control, and converts the arrow keys, as well as the common game navigation keys W, A, S, and D into directional navigation equivalents.

4.5.2 Multiple Mouse Cursors

A few systems, such as the Nintendo Wii™, support multiple cursor devices. The CLIK component framework supports multiple cursors, but will not allow multiple components to be focused in a single SWF. If two users click on separate buttons, the last clicked item will be the focused item.

All mouse events dispatched in the framework contain a mouseIndex property, which is the index of the input device that generated the event.

```
      myButton.addEventListener("click", this, "onCursorClick");
      function onCursorClick(event:Object):Void {
            switch (event.mouseIndex) {
            …
            }
      }
```

4.6 Invalidation

Invalidation is the mechanism used by the components to limit the number of times the component redraws when multiple properties are changed. When a component is invalidated, it will redraw itself on the next frame. This enables developers to throw as many changes at once to the components, and only have the component update once. There are a few exceptions where the component needs to redraw immediately; however for most cases, invalidation is sufficient.

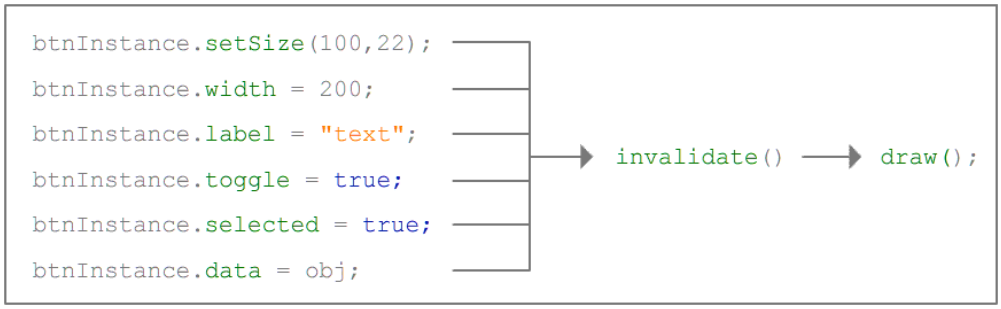

Figure 63: CLIK components automatically invalidate when certain properties are changed.

4.6.1 Usage and Best Practices

After any internal component change (usually caused by setter functions), UIComponent.invalidate() should be called, which ultimately calls UIComponent.draw()in the component. The invalidate() method generates the draw() call using a timer-based delay to avoid unnecessary updates. Developers using the existing components will likely not need to manage invalidation, but should at least be aware of it.

For cases where an immediate redraw is required, developers can use the UIComponent.validateNow() method.

4.7 Component Scaling

Scaleform CLIK components scale in two ways:

1. Using a reflowing layout, in which the component's scale is reset and then its elements resized to match the original size. Components with subelements and no background take this approach.
2. Counterscaling the elements to maintain the aspect ratio, in which the component remains scaled, but its elements are counter-scaled to give the appearance of not scaling. This approach enables components with a graphics background, and a scale9grid to be stretched, and the context to scale inside instead of becoming distorted.

Typically components use the reflow method due to Flash scale9Grid limitations. This forces developers to build "skin" symbols, similar to Flash/Flex components. Reflowing works best if components have sub-elements that may also scale, therefore it is intended for containers and similar entities.

The counter-scaling method was created specifically for CLIK, mainly due to the extended scale9Grid functionality available in Scaleform. This allowed for the creation of single-asset components using frame states instead of the more intensive layered approach used by other component sets. The base CLIK components are minimalistic in nature, usually containing a background, a label and an optional icon or sub-button, and therefore ideal for the counter-scaling method. However, counter-scaling is not intended for container-like setups (panel layout, etc.). In such cases, the reflow method with a background that is constrained along with the rest of the sub-elements is the recommended approach.

Components can be scaled on the Stage in the Flash IDE, or dynamically scaled using the width and height properties, or the setSize() method. The appearance of scaled components may not look accurate in the Flash IDE. This is a limitation of delivering the components as un-compiled MovieClips without LivePreviews. Testing the movie in Scaleform Player is the only way to get an accurate representation of how the scaled

component will appear in-game. The CLIK extension bundles a Launcher panel to improve this part of the workflow.

4.7.1 Scale9Grid

Most components use the second scaling method. MovieClip assets inside of a Scale9Grid in the Scaleform Player will adhere to the scale9grid for the most part, whereas Flash will discard the grid if it contains MovieClips. This means that even though a scale9grid does not work in Flash Player, it may work perfectly in Scaleform.

One item to note is that when a MovieClip has a scale9grid, subelements will also scale under that rule. Adding a Scale9grid to the subelement will cause it to ignore the parent's grid, and draw normally.

4.7.2 Constraints

The Constraints utility class (gfx.utils.Constraints) assists in the scaling and positioning of assets inside a component that scales. It enables developers to position assets on the Stage, and for the assets to retain their distance from the edges of the parent component. For example, the ScrollBar component will resize and position the track and down arrow buttons according to where they are dropped on the Stage. Constraints work with both scaling methods used in the components.

The following code adds the ScrollBar assets to a constraints object in the configUI() method, aligning the down arrow button to the bottom, and scaling the track to stretch with its parent. The draw() method contains code to update the constraints, and consequently any elements registered with it. This updating is done inside draw() because it is called after component invalidation, commonly after the component's dimensions have changed.

```
private function configUI():Void {
      …
      constraints = new Constraints(this);
      // The upArrow already sticks to top left.
      constraints.addElements(downArrow, Constraints.BOTTOM);
      constraints.addElenent(track, Constraints.TOP|Constraints.BOTTOM);
      …
}

private function draw():Void {
      …
     constraints.update(__width, __height);
      …
}
```

4.8 Components and Data Sets

Components that require a list of data use a dataProvider approach. A dataProvider is a data storage and retrieval object that exposes all or part of the API defined in the Scaleform CLIK IDataProvider class (gfx.interfaces.IDataProvider).

The dataProvider approach uses a request model with callbacks, instead of direct property access. This allows the dataProvider to reach out to the game engine for data as it is needed. This provides memory benefits, as well as the ability to chunk large data sets into small, manageable ones.

Components that use a dataProvider include anything that extends CoreList (ScrollingList, TileList), OptionStepper and DropdownMenu. None of the components in the CLIK framework require the use of the IDataProvider interface; only the inclusion of the methods in its API. The IDataProvider class is only provided for reference.

4.8.1 Usage and Best Practices

The DataProvider class (gfx.data.DataProvider) included in the framework uses its static initialize() method to add the dataProvider methods to any ActionScript array. Components that use a dataProvider will automatically initialize arrays making them accessible using the dataProvider approach. This means the following syntax initializes a statically declared array as a fully operational dataProvider with the methods described in IDataProvider:

```
myComponent.dataProvider = [ "data1",
                             4.3,
                             {label:"anObjectElement", value:6} ];
```

The methods that a dataProvider should implement are:

- *length*: Either as a property or a getter function to return the length of the data set;
- *requestItemAt*: Request a specific item from the dataProvider. Typically used by list components that display a single item at any time, such as the OptionStepper;
- *requestItemRange:* Request a range of items from the dataProvider with a start and end index. Typically used by list components that displays more than one item, such as the ScrollingList;
- *indexOf:* Return the index of an item;
- *invalidate:* Flag the dataProvider as changed, and provide a new length of the data. This method also should dispatch a "dataChange" event to notify the component that the data has been updated. The dataProvider should support a length property that is publically accessible and reflects the data size.

Instances requiring short lists of data can use an array as a dataProvider. Developers should be aware that storing data in ActionScript 2 is much more expensive in terms of memory and performance that if stored natively in the application. For large data sets it is recommended to tie the dataProvider with ExternalInterface.call to poll data from the game engine on a need basis.

4.9 Dynamic Animations

Scaleform CLIK provides a custom Tween class (gfx.motion.Tween) that has similar behavior to the Flash 8 Tween class, but is fully compatible with Scaleform. Both Tween classes support the same types of easing functions, such as the ones under the mx.transitions.easing.* package.

However, the CLIK Tween class has significant differences to its counterpart. Firstly, it installs tween methods to the Flash 8 MovieClip class prototype. This exposes tween functionality to all MovieClips, not just CLIK components. Secondly, the CLIK Tween class uses onEnterFrame and therefore only one tween can be active per MovieClip at one time. If a new tween is created for a MovieClip that already has an active tween, then the new tween will stop the previous one. However the tween methods support multiple properties per MovieClip, allowing different properties of the same object to be changed at the same time. The examples in the following section show how to create tweens that affect multiple properties at the same time.

4.9.1 Usage and Best Practices

The Tween class exposes two main methods in MovieClip: tweenTo() and tweenFrom(). The tweenTo() method tweens from the current property value of the MovieClip to one specified in the function parameter list. The tweenFrom() method tweens from a property value specified in the parameter list to the current value.

Before using the tween methods, they must be installed with the MovieClip prototype. The Tween class provides a helper static function to perform this action: Tween.init().

```
import mx.transitions.easing.*;
import gfx.motion.Tween;
Tween.init();     // Install tween methods to MovieClip prototype
// Perform a 1 second tween from the current horizontal position and
  alpha values to the ones specified
myMovieClip.tweenTo(1, {_x: 200, _alpha: 0}, Strong.easeIn);
```

To be notified when the tween has completed, simply create an onTweenComplete() function in the object being tweened.

```
import mx.transitions.easing.*;
```

```
import gfx.motion.Tween;
Tween.init();
mc.onTweenComplete = function() {
      trace("Tween has finished!");
}
mc.tweenTo(5, {_rotation: 20},   Bounce.easeOut);
```

4.10 PopUp Support

Scaleform CLIK includes the PopUpManager class (gfx.managers.PopUpManager) to provide support for popups such as dialogs and tooltips. The Dialog component uses the PopUpManager to display its content.

4.10.1 Usage and Best Practices

The PopUpManager has several static methods to assist in the creation and maintenance of popups. The createPopUp() method can be used to create a popup using a linkage ID to any MovieClip symbol (including CLIK components). The context parameter is used to create an instance of the popup, and the relativeTo parameter is used to position the popup. Both parameters are expected to be MovieClips.

```
import gfx.managers.PopUpManager;
PopUpManager.createPopUp(context, "MyToolTipLinkageID",
                                  {_x: 200, _y:200}, relativeTo);
```

The Scaleform extension topmostLevel is used to guarantee that the popups are always displayed on top of all other elements on the Stage, This scheme was used instead of trying to create the popup at the root level because of problems related to library scopes. If a child SWF was loaded inside a parent, and it tried to create an instance of a symbol defined in its library in a path that existed in the parent context, then a symbol lookup error would occur because the parent is unable to access the child's library. Unfortunately, there is no clean workaround for this problem and the topmostLevel scheme is the best alternative.

Note that topmostLevel can be applied to non-popups as well, and the z-ordering of such elements in the Stage is maintained. Therefore, to draw a custom cursor on top of popups, the cursor can be created at the highest depth or level and set its topmostLevel property to true.

4.11 Drag and Drop

The DragManager class (gfx.managers.DragManager) provides support to initiate and manage drag operations. The class behaves similar to a singleton, and provides an instance property to access the one and only object. Using this object, developers can start and stop drag operations.

The startDrag() method can be invoked to start the drag operation using either a MovieClip on the Stage or a symbol ID, which will create an instance of the symbol for dragging. The stopDrag() method ends the drag-

operation and also notifies all listeners. The DragManager also registers itself with the InputDelegate to allow drag-operations to be cancelled using the keyboard or equivalent controller.

4.11.1 Usage and Best Practices

Figure 64: DragDemo in action.

DragManager only performs drag management. It is up to developers to create components that utilize the DragManager to provide true drag and drop support. Scaleform CLIK includes a sample drop target class called DragTarget (gfx.controls.DragTarget). The DragTarget extends UIComponent and thus is classified as a CLIK component. It has several unique features that complement the DragManager.

The DragTarget has a concept of drag types. These drag types can be configured per DragTarget to enables the component to allow or disallow a drop operation. To complement these drag types, the DragTarget also installs event listeners for dragBegin and dragEnd with the DragManager. This enables the DragTarget to visually display whether or not it can accept a drop operation.

The Inventory demo bundled with CLIK uses two components that subclasses or composes the DragTarget to demonstrate drag and drop operations using the DragManager and the DragTarget. These two components are the InventorySlot and the ItemSlot. These component classes are provided as part of the CLIK framework; however, note that they are meant to be a sample of what can be accomplished and not an end-all solution to all use cases.

The InventorySlot class (gfx.controls.InventorySlot) supports the display of a set of icons that denote inventory items, such as those commonly found in role playing games. It subclasses the DragTarget class to provide drop support, and contains code to generate drag operations using the DragManager. When the drag operation begins, a copy of the icon is created to follow the cursor. On a valid drop operation, the target InventorySlot will change its icon to be the one that was dragged.

The ItemSlot class (gfx.controls.ItemSlot) is very similar to the InventorySlot, but contains extra functionality to support mouse events such as clicking. Instead of subclassing, the ItemSlot contains an instance of DragTarget. It also contains an instance of a CLIK Button. These two subelements provide the necessary functionality for ItemSlot to support both drag and drop and button operations.

4.12 Miscellaneous

4.12.1 Delegate

The Delegate class (gfx.utils.Delegate) provides a static method to create a function delegate with the correct scope. It is not used by any Scaleform CLIK components, and is mainly used by the DragManager. Usage example:

```
eventProvider.onMouseMove = Delegate.create(this, doDrag);
```

4.12.2 Locale

The Locale class (gfx.utils.Locale) provides an interface for custom localization schemes. The Scaleform localization scheme performs translation lookup internally and does not require an explicit lookup from ActionScript. However, custom translation providers may require this lookup. The Button, Label and TextInput components and their subclasses all query for a localized string via the Locale.getTranslatedString() static method. The default implementation simply returns the same value. It is up to the developer to modify the Locale class to support the custom localization scheme.

5 Examples

The following sections contain several examples of use cases that are implemented using Scaleform CLIK components.

5.1 Basic

The following examples are simple in scope and complexity, but demonstrate the ease of use of the Scaleform CLIK framework to perform general tasks.

5.1.1 A ListItem Renderer with two textFields

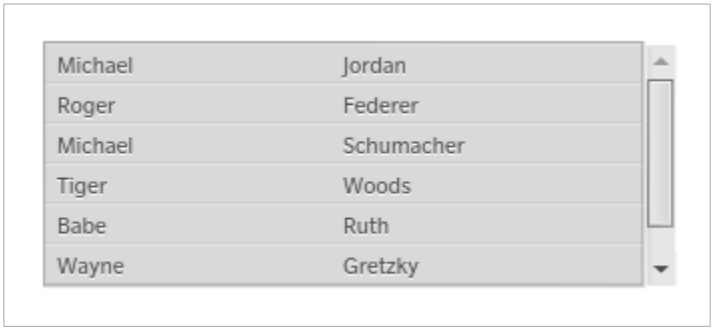

Figure 65: ScrollingList showing list items containing two labels.

The ScrollingList component by default uses the ListItemRenderer to display the row contents. However the ListItemRenderer only supports a single textField. There are many cases in which a list item may have more than one textField to display, or even non-textField resources such as icons. This example demonstrates how to add two textFields to a list item.

First, let us define the requirements. The objective will be to create a custom ListItemRenderer that will support two textFields. This custom ListItemRenderer should also be compatible with the ScrollingList. Since the plan is to have two textFields per list item, the data should also be more than just a list of single strings. Let us use the following dataProvider for this example:

```
list.dataProvider = [{fname: "Michael", lname: "Jordan"},
                     {fname: "Roger",   lname: "Federer"},
                     {fname: "Michael", lname: "Schumacher"},
                     {fname: "Tiger",   lname: "Woods"},
                     {fname: "Babe",    lname: "Ruth"},
                     {fname: "Wayne",   lname: "Gretzky"},
                     {fname: "Usain",   lname: "Bolt"}];
```

This data provider contains objects with two properties: fname and lname. These two properties will be displayed in the two list item textFields.

Since the default ListItemRenderer only supports one textField, it will need to functionality to support two textFields. The easiest way to accomplish this is to subclass the ListItemRenderer class. The following is the source code to a class called MyItemRenderer, which subclasses ListItemRenderer and adds in basic support for two textFields. (Put this code in a file called *MyItemRenderer.as* in the same directory as the FLA being worked on):

```
import gfx.controls.ListItemRenderer;

class MyItemRenderer extends ListItemRenderer {

	public var textField1:TextField;
	public var textField2:TextField;

	public function MyItemRenderer() { super(); }

	public function setData(data:Object):Void {
		this.data = data;
		textField1.text = data ? data.fname : "";
		textField2.text = data ? data.lname : "";
	}

	private function updateAfterStateChange():Void {
		textField1.text = data ? data.fname : "";
		textField2.text = data ? data.lname : "";
	}
}
```

The setData and updateAfterStateChange methods of ListItemRenderer are overridden in this subclass. The setData method is called whenever the list item receives the item data from its container list component (ScrollingList, TileList, etc.). In the ListItemRenderer, this method sets the value of the one textField. MyItemRenderer instead sets the values of both textFields and also stores a reference to the item object internally. This item object is reused in updateAfterStateChange, which is called whenever the ListItemRenderer's state changes. This state change may require the textFields to refresh their values.

Thus far, the dataProvider with the complex list items and a ListItemRenderer class that supports rendering of those list items have been defined. To hook up everything with the list component, a symbol must be created to support this new ListItemRenderer class. For this example, the fastest way to accomplish this would be to modify the ListItemRenderer symbol to have two textFields called 'textField1' and 'textField2'. Next this symbol's identifier and class must be changed to MyItemRenderer. To use the MyItemRenderer component

with the list, change the itemRenderer inspectable property of the list instance from 'ListItemRenderer' to 'MyItemRenderer'.

Run the FLA now. The list should be visible containing list elements that display two labels: the fname and lname properties that were set in the dataProvider.

5.1.2 A Per-pixel Scroll View

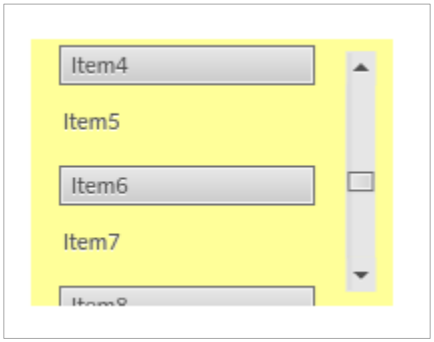

Figure 66: Per-pixel scroll view with content containing CLIK elements.

Per pixel scrolling is a common use case in complex user interfaces. This example demonstrates how to achieve this easily using the CLIK ScrollBar component.

To begin, create a new symbol for the scroll view. This provides a useful container that simplifies the math required to compute content offsets. Inside this scroll view container, set up the following layers in top to bottom order. These layers are not required, but are recommended for clarity:

- ***actions***: Contains the ActionScript code to make the scroll view work;
- ***scrollbar***: Contains an instance of the CLIK ScrollBar. Call this instance 'sb';
- ***mask***: The mask layer defined by a rectangular shape or MovieClip. Set the layer property to be a mask as well;
- ***content***: Contains the content to be scrolled;
- ***background***: Optional layer containing a background to highlight the unmasked area.

Add the appropriate elements to the content layer and create a symbol that holds all of them. By creating a symbol that holds all of the content, the task of scrolling the content becomes easier. Call this content instance 'content' and set its y-value to 0. This ensures that the scrolling logic does not need to account for different offsets. However, such offsets may be required depending on the complexity of your scroll view.

Thus far the structure necessary to create a simple scrollview, as well as named elements that need to be hooked together have been defined. Put the following ActionScript code in the first frame of the *code* layer:

```
// 133 is the view size (height of the mask)
sb.setScrollProperties(1, 0, content._height - 133);
sb.position = 0;

sb.addEventListener("scroll", this, "onScroll");
function onScroll() {
      content._y = -sb.position;
}
```

Since the scrollbar is not connected to another component, it needs to be configured manually. The setScrollProperties method is used to make it scroll by one position, and set its minimum and maximum values to completely display the content. The scrollbar positions in this case are in pixels. Run the file now. The scrollview may now be changed by interacting with the scrollbar.

Note that extensive use of masks in Scaleform can degrade performance significantly, especially for HUDs. Scaleform recommend that developers profile their content for performance metrics.

5.2 Complex

The following sections describe several complex demos that bundled with Scaleform CLIK. Only high-level implementation details are provided. Note that these sections require a fairly thorough knowledge of Flash and ActionScript 2.

5.2.1 A TreeView using the ScrollingList

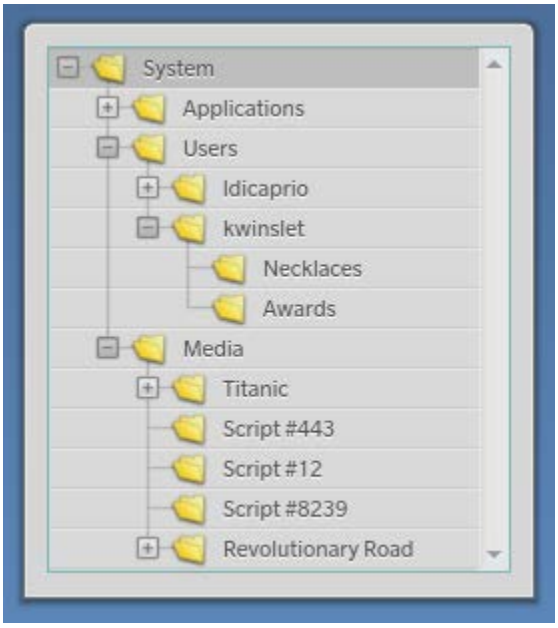

Figure 67: TreeView demo.

The TreeView Demo (*Resources/AS2/CLIK/demos/TreeViewDemo.fla*) implements a tree view control using the ScrollingList component as a base along with a custom ListItemRenderer and DataProvider.

Let's consider the differences between a normal ScrollingList and a tree view:

- The number of elements in the tree view can change depending on the state of its item hierarchy, but they are still a linear sequence of elements.
- The tree view item must display its depth in the hierarchy tree using a visual indicator; in most cases this would be spaces to provide a 'tabbed' look. This is easily possible by modifying a ListItemRenderer to 'tab' its contents.
- The tree view must contain a mechanism to allow users to expand and contract each item. Again, a ListItemRenderer can be modified to include a button subelement to provide this functionality.

The ScrollingList is an effective surrogate for a tree view built from scratch. For the demo, a custom ListItemRenderer called TreeViewItemRenderer and a custom data provider called TreeViewDataProvider are defined. The demo also uses a small utility class called TreeViewConstants to provide common enumerations. All of these classes can be found under the *Resources/AS2/CLIK/demos/com/scaleform* directory.

The first item to decide when implementing a tree view is its data representation. ActionScript 2 Objects are inherently hash tables, and this demo exploits this property. The tree is constructed using a tree of Objects, such as the following:

```
var root:Object =
{
      label: "System",
      nodes:[
               {label: "Applications",
                nodes:[
                           {label: "CGStudio",
                            nodes:[
                                       {label: "Data"},
                                       {label: "Samples"},
                                       {label: "Config"}
                                      ]
                              },
                           {label: "Money Manager"},
                           {label: "Role Call v6"},
                           {label: "Super Draft",
                            nodes:[
                                       {label: "Templates"},
 …
```

Each item in the tree is represented as an Object with two properties: label and nodes, which is an Array containing potential child nodes. To bridge this hierarchical representation with the ScrollingList, a custom data provider to parse and maintain the tree structure and expose the appropriate data to the list are required.

The TreeViewDataProvider takes in this tree object in its constructor and performs a preprocess step to annotate each item with special properties only useful for the data provider and the custom ListItemRenderer. These properties include the item type, its expanded/contracted state, parent/sibling links and an array of line graphic descriptors that describe which line graphics to use during rendering. The TreeViewDataProvider implements the appropriate public methods required by a CLIK data provider, but defers most of the work of item retrieval to several helper functions. For example, item range retrieval first finds the tree object using the list's top most item index, and then collects a linear sequence of items starting from the top-most item via tree traversal.

The TreeViewItemRenderer receives its data from the TreeViewDataProvider via the ScrollingList. The data object it gets contains all the meta data required to render the item correctly, without any knowledge of the tree structure. The unmodified ListItemRenderer component symbol is reused in this demo, with its class linkage set to the TreeViewItemRenderer. The list item's folder icons and line connector graphics are created dynamically. These graphical elements are pre-cached to avoid duplicate instantiations of the same symbol at the same 'tab' slot. The textField element of the ListItemRenderer is shifted to the right, based on the item's depth.

The TreeViewConstants class contains enumerations for the following:

- type of node in the hierarchy: open node, closed node, leaf node;
- type of folder icon;
- type of line connector.

The latter enumerations are used for display purposes only. This demo uses folder icon and line graphics to provide seamless connections between items regardless of depth. To achieve this effect, the folder icons and line graphics must cover all different configurations of expanded and contracted views.

Since this tree view is based on a ScrollingList, a ScrollBar component can be connected to it. The scroll bar changes its appearance based on the state of the tree hierarchy; the expanded/contracted state of the tree affects the scrollbar's grip size. Interacting with the scroll bar modifies the ScrollingList view as expected.

5.2.2 A Reusable Window

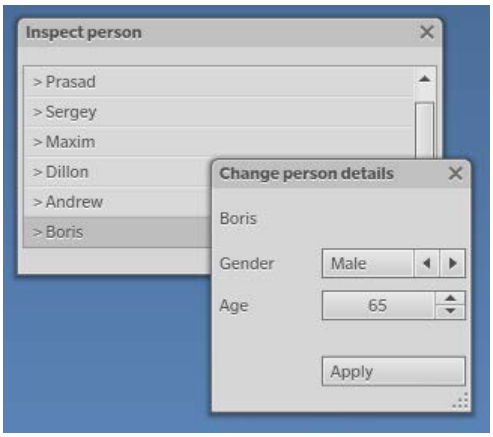

Figure 68: Window demo.

The Window Demo (*Resources/AS2/CLIK/demos/WindowDemo.fla*) provides a simple reusable window component capable of displaying arbitrary content. The Window component class can be found under *Resources/AS2/CLIK/demos/com/scaleform/*.

This Window component subclasses the core UIComponent class and thus is classified as a CLIK component. It does not exist under the core framework classes because a general purpose Window component is hard to define, let alone be implemented efficiently. The more features one adds to a component, the more cumbersome it becomes in terms of memory and performance.

Therefore, this class only implements a core set of common features expected from a Window component. They are:

- modifiable title bar;
- draggable title bar and window background;
- close button;
- resizable border (bottom right only);
- minimum and maximum dimensions.

The Window class exposes several inspectable properties:

Title	The title of the Window.
formPadding	The amount of padding between the Window border and its contents.
formType	Type of content to load. This value can be either 'symbol' or 'swf'. If 'symbol', then the content is loaded from the library. If 'swf', then an external SWF file is loaded.
formSource	The source of the content. This can either by a symbol name (if *formType* is 'symbol') or SWF file name (if formType is 'swf').
allowResize	Shows or hides the resize button.
minWidth, maxWidth, minHeight, maxHeight	The resizing dimensions of the window. If the max values are set to a negative value, then they will be clamped to its min counterparts. Setting both max values to negative values has the same effect as hiding the resize button, which disables resizing completely. If the max values are set to 0, then the component can be resized without a max limit. The minimum will always be the initial size of the content.

Adding an inspectable property to a custom component class is quite easy. If a public property does not require any code to execute when getting or setting the property, then it can be declared as follows:

```
	[Inspectable(name="formType", enumeration="symbol,swf"]
	private var _formType:String = "symbol";
```

For more information on the Inspectable meta data syntax, please refer to the Flash documentation. Note that the Inspectable meta data provides support for aliasing the actual property; in this case the _formType property is aliased as formType for readability.

If the property requires code to be executed after setting or getting its value, then the property should define getter and setter functions:

```
	[Inspectable(defaultValue="Title")]
	public function get title():String { return _title; }
	public function set title(value:String):Void {
		_title = value;
```

```
            invalidate();
      }
```

This Window component inherently supports several subelements, such as the title button, close button and resize button. Each of these buttons is a CLIK Button component. Since these buttons are required by the class, the component symbol must reflect this requirement by including the appropriate subelements.

Since the Window class subclasses the UIComponent class, the two main methods are overriden: configUI() and draw(). The configUI() method sets up the button listeners and the constraints object for resizing. The draw() method generally performs reflowing of the component's layout either manually or using the constraints object. However, it also contains logic to load the content using a symbol name or file path. After loading a private method, configForm(), is called to set up the contents.

File loading is performed using the Flash 8 MovieClipLoader. Since file loading is performed asynchronously in Scaleform when threading support is enabled, the component invokes configForm() from within a MovieClipLoader.onLoadComplete() callback that is triggered when file loading has completed.

The form content loaded either via symbol or file path are added to the internal constraints object. Whenever the Window component is resized, it is invalidated. This causes the draw() method to be invoked, which consequently performs an update on its constraints object with the new dimensions. Since the form content is registered with the constraints object, it is notified of the change via the validateNow() method.

If the form content derive from UIComponent, then its own draw() method will be invoked. Layout management logic for the form content can be executed when the draw() method is called. If the form content does not derive from UIComponent, then it can define a validateNow() method to reflow its own elements. The Window Demo contains several examples of symbols and SWF files that use the draw() method and the validateNow() method to maintain form layout.

The Window component also contains special logic to move the window that was pressed by the mouse cursor to the foreground. The logic näively assumes that all Window component instances exist in the same level, and bubbles the component in focus to the next highest depth.

```
      private function onMouseDown() {
            var targetObj:Object = Mouse.getTopMostEntity();
            while (targetObj != null && targetObj != _root)
            {
                  if (targetObj == this) {
                        swapDepths(_parent.getNextHighestDepth());
                        return;
                  }
            targetObj = targetObj._parent;
```

```
            }
        }
```

This is one way of implementing such behavior. Another way is to create a WindowManager to maintain the z-indices of all window components on the Stage. This WindowManager approach is in fact more flexible than the way implemented by this sample Window component.