Flash to Unity UI Components

Monday, October 7th, 2013

Abstract:

Flash to Unity offers several utilities when it comes to creating components, of which the most important are the UI components. The following document will focus on detailing each and every IU component, alongside its required and optional attributes.

Table of Contents

Introduction

The user may feel free to use Flash to Unity's basic components only, as they provide the base functionality; a complete game can be made by using exclusively those components. The full API, however, offers more functionality than that of the base components. The following sections will focus on several complementary components that the user may feel free to use for their applications.

The Hit Area Sprite

Before talking about the components per se, it is important to take into consideration that several of the following components depend on the existence of a common layer, aptly named "hit_area," that contains a single element to a "hit_area" animation. This layer is shared amongst every component that involves human interaction, and its only component should be an invisible object set around the area the user is allowed to interact with the component. When you assign a script to a game object, Flash to Unity will search for the "hit_area" layer, if it applies, in order to create the corresponding area of interaction. If functionality doesn't matter, most UI components can be reduced, at the very minimum, to an animation with a single "hit_area" layer. It is important for this layer to have a reference to an animation per se.

Common Labels

In addition to the hit_area label, certain components need for their Flash counterpart to contain labels in order to define the frame to be played whenever certain states are activated. For instance, a check box component may need to have the “checked”, “unchecked” and “disabled” labels in order to function properly. Along this manual, every common label will be explained in the section corresponding to their component. It is to be noted that the label names should be written verbatim. For example, a “checked” label must be named as “checked”, minus the quotes, on the Flash project. In order to learn more about the components required on the several structures, you may check the components scene example (F2UUIComponentsScene) and its corresponding .XFL file (f2u_ui_components_scene.xfl).

Style Keys

Most components have a “style key” property, which can be defined in order to assign properties to said components without having to modify each and every component on the scene. The latter allows for component customization without necessarily having to modify the properties of each and every object.

User Interface Components

Flash to Unity has the ability to create fully functional and customizable UI components capable of several standard actions, such as getting pressed or checked, being scrolled, dragged or slided, among others. The following subsections will delve into the components that can be used in a common project.

Buttons

Buttons are basic clickable components, normally used to confirm selections or when presenting a choice to the player. Buttons in Flash to Unity are defined in the script F2UButton, and they have an inherent method that executed whenever the button is clicked. In order to create a simple button, it's enough to make an empty game object and embed the F2UButton script onto it. The button needs a sprite object in order to draw itself; the user must attach an F2USprite script to the button if an animation script is not already in effect. In other words, the button must also be its own sprite. Buttons may also have separate sounds to play when they are clicked, and can also be set to wait for their related sound to finish playing before executing their actions.

Buttons may throw an event when they are clicked. The event itself may be set to whatever the user would like it to do, but in order to actually execute the event, the button's "Throw Event on Press" property must be set to true. Events are passed by the onClick function on the button class, and it may be reassigned at will.

Buttons have a series of properties that can be used to tweak their behavior. Said properties are listed as follows:

- Style Key: The key to the button's style.
- Text: The text the button displays. It receives an actual "F2UText" object.
- Throw Event on Press: Determines if the button will throw an event when pressed.
- Shortcut: Defines a keyboard shortcut for the button.
- Sound Group Name: The sound group that will play the button's sounds.
- Sound Priority: The priority of the button's sounds.
- On Click Sound: The sound to be played whenever the button is clicked.
- Pressed Color: The color the button will be tinted with when pressed.
- Disabled Color: The color the button will be tinted with when disabled.

- Rollover Color: The color the button will be tinted with when the mouse passes over it.
- Rollover Scale: The scale the button will suffer when the mouse passes over it.
- Enable Text Transformations: Determines whether the associated text should suffer from transformations along the button.
- Pressed Text Color: The color of the text when the button is pressed.
- Disabled Text Color: The color of the text when the button is disabled.
- Wait for Sound: Determines whether the button should wait for its associated sound to finish playing before executing its action.

Animated buttons

Sometimes, the user may want a button that animates when certain actions are made over them, as opposed to just change color. Flash to Unity allows animations with a determined format to become fully functional animated buttons. Additionally, animated buttons may also be animations, where several labels may be used to determine which parts of the animation to play at which moments. The base required label are listed as follows:

- idle: Determines the part of the animation to play when the button is idle.
- down: Defines the part of the animation that plays when the button is pushed.
- over: Declares the part of the animation that plays whenever the cursor passes over the button. Has no effect on purely tactile inputs.
- up: Defines the part of the animation that plays whenever the button is un-pressed.
- disabled: This label determines the part of the animation to be played when the button gets disabled.

Due to their nature, only the following behavior is customizable for animated buttons:

- Style Key
- Text
- Throw Event on Press
- Shortcut
- Wait for Animation

All properties work identically to how they work on regular buttons.

Check boxes

Check boxes are components that allow a player to choose from multiple options. Depending on what they will be used for, the check boxes may allow for multiple choices to be made, or may only allow for a single choice to be made, on which case they would be functionally identical to radio buttons. Check boxes may also be extended into fully functional tab panels. In order to make a checkbox out of an object, the F2UUICheckBox script must be attached to it. Additionally, it must be given a showable area and a hit area, both represented by sprites. Check boxes have three basic states, “checked”, “unchecked” and “disabled”, all of which should appear as identically named labels on the Flash animation for Flash to Unity to parse.

Check boxes have a base event, which fires up when the component transitions between a state and another. Any transition fires up the event, so it’s advisable to take care whenever using or firing this event. “OnCheckBoxStateChanged” is the name of this event.

Popups

Popups are small window-like entities that appear whenever a certain event is fired, or even automatically at the start of a scene. Popups exist since the scene is created,

and are hidden from the player until an event forces them to become visible. In order to create a popup, the user should add the F2UUIPopup script to a game object. The user may then attach other objects to the popup component as children in order to control them along with the window; all children objects will disable when the popup disappears, and will enable when the popup emerges; the opposite is true for objects that are not related to the popup. The common labels associated with this component are listed thusly:

- show: Defines the part of the animation to be played whenever the popup becomes visible.
- shown: Defines the part of the animation to be played while the popup remains visible.
- hide: Defines the part of the animation to be played whenever the popup becomes invisible.
- hidden: Defines the part of the animation to be played while the popup remains invisible.

Please do note that the “hidden” label should exist on one frame after the rest of the animation, along with any action related to this state.

The properties associated to this component are listed as follows:

- Sound Group Name: The name of the sound group that will play the popup’s sounds.
- Sound Priority: The priority of the sounds to be played.
- Preload Sounds: Determines if the sounds should be preloaded.
- Use Batched Sprites: Determines whether the sprites for the popup are candidates for sprite batching.

Progress Bars

Progress bars are components whose sole objective is to show progress over an action. They may be freely attached to an object in the scene and their Progress property bound to an event in order to update it automatically. Progress bars use sprites to show percentages; the more of a sprite is shown, the more the progression converges to 1. The progress bar does not impose restrictions on how should the progress be modified, so negative progress is a possibility that you may or may not want to avert. It's important to note that the whole progress bar animation should be created in Flash prior to importing them. The first frame should contain an "idle" label, which will point to the part of the animation to be played at the start, when the progress bar has not made any progress. The properties available on this component are detailed as follows:

- Style Key: The key to the bar's style.
- Progress: The current progress of the bar, defined as a float value.

Sliders

Sliders are objects whose objective is to make a component to scroll either horizontally or vertically. The orientation of the scroll can be customized, and they may be bound by a hit area sprite. In order to create a slider, it's usually enough to attach the F2UUISlider script to an object and set its hit area. Sliders may also become sprites in order to show on-screen. Sliders have three main common labels:

- idle: Part of the animation to show whenever the slider is idle.
- scrolling: Part of the animation to show whenever the slider is scrolling.
- disabled: Part of the animation to show whenever the slider is disabled.

Additionally, the "scroll_sprite" layer should be added, separated from other objects, in order to define the object that will scroll along with the dragging. The hit_area layer, however, should encompass the entirety of the slider, not just the scroll sprite. The properties accessible for this component are listed as follows:

- Style Key: The key to the slider's style.
- Orientation: The direction of the scroll. It can be either horizontal or vertical.
- Scroll Percentage: Determines the relative position of the object to be scrolled as a percentage.

Checkbox Groups

Checkbox groups are a way to transform several independent check boxes into a collection of related radio buttons. Checkbox groups encapsulate some previously defined check boxes and assign them behavior so that one and only one check box can be selected at the time. Even though they use checkbox components to build the functionality, their behavior is identical to the one of regular radio buttons. The Flash project must have a sole animation for the checkbox group in order to implement it. Once the previous step is achieved, the user may just create an object, attach the F2UUICheckboxGroup script to it and begin to assign child check boxes onto its checkbox list. While checkbox groups don't have any particular common tags and may lack hit areas, it's advisable to group the checkboxes under a layer folder named "checkboxes." After importing and successfully creating your checkbox group, make sure you correctly assign the checkboxes to the group via the "checkboxes" property.

Checkbox groups have the following properties:

- Check Boxes: This property holds information about the checkboxes associated with this component. It maintains its own size and a reference to each checkbox.

Tab Panels

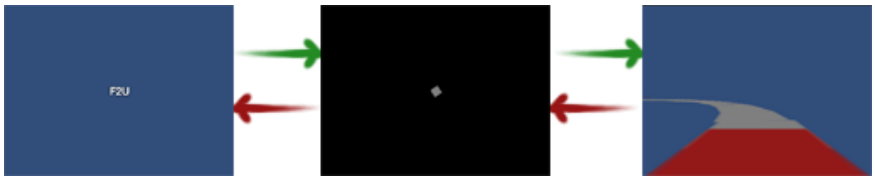
Similar to checkbox groups, tab panels allow a group of checkboxes to behave as a single entity. Tab panels contain a collection of checkboxes that behave as tabs for a single panel. The structure is similar to the one of checkbox groups, with the addition of a new layer folder: "panels". The "panels" folder holds the layers for every panel to be shown. Ideally, each panel will be shown just at a specific frame, which should be

labeled with the same name as the checkbox that shows it. The properties of this component are as follows:

- Sound Group Name: The sound group where the panel's sounds will be played.
- Sound Priority: The priority of the sounds.
- Preload sounds: Determines whether the sounds should be preloaded.
- Use Batched Sprites: Determines whether the panel's sprites are candidates for batching.

Loading Screens

Loading screens are complex structures that show a single, lightweight scene while another pair of scenes is being switched around. Loading scenes are useful when switching into a heavy scene, and they serve to let the user know the game hasn't frozen, but it's waiting for some operation to complete. Loading screens use a single prefab that is instantiated and preserved until the end of the program's execution that shows a single animation in a hopefully lightweight intermediary scene.



Basic loading transition routine between two scenes, using a loading scene as an intermediary.

Due to how Unity works, two switching scenes overlap on memory, so at one point both scenes are active at the same time. While this is a non-issue on relatively small scenes, larger scenes may make the computer lag due to the sheer weight of both scenes. When Flash to Unity's loading manager processes a request to switch scene A with scene B, it follows a four-step process:

1. The manager fires up the loading animation and fades it in.
2. Scene A gets unloaded and replaced with a dummy scene θ. As θ is supposed to be almost empty, switching between θ and A becomes a non-issue.
3. The manager waits for A to get completely unloaded from memory.
4. Scene I gets replaced with scene B. Again, since θ is by no means a heavy scene, performance is maintained when switching from θ to B.

Of course, the loading animation is optional and only serves to fulfill instant gratification standards, but it is recommended to use the manager when switching between scenes, especially heavier ones, as this may allow for faster transitions between scenes.

Gestures

Gestures are objects that allow Unity to detect several user movements, such as swipes or taps (in devices with touch screen), as well as to allow the user to freely rotate or scale an object. The following subsections will focus on the gestures Flash to Unity allows for.

Rotate Gesture

Rotate gestures are used to rotate an object around a pivot. Both the pivot and the rotatable transform must exist in the scene, and the pivot will contain an F2URotateGesture script. This script must know which object to affect; in order to do so, just reference the rotatable object's transform on the script as at the Rotable Transform property. Moreover, the pivot must also become a sprite in order to be both visible and interactable. Once the pivot and the object are created, the user may run the script and

freely test the rotation of each screen. The properties currently supported by rotate gestures include:

- Rotate pivot: Determines whether the hit area gets rotated along the gesture.

Scale Gesture

Scale gestures, much like rotate gestures, are used to transform an object. Using a scale gesture, you may let a player scale an object between a minimum and a maximum size. Due to the nature of this gesture, the only way to scroll up or down a component in a desktop device is to scroll with the mouse wheel over it. The scalable object must have a scalable area on its hierarchy, and in return that area must reference the scalable object's transform on its F2UScaleGesture script in order to act upon it. Scale may be achieved by dragging opposite ends of an object. The modifiable properties for this component include:

- Min scale: Determines the minimum value the object can be scaled down.
- Max Scale: Determines the maximum value the object can be scaled up.

Swipe Gesture

Swipe gestures are used to detect whenever the player performs a swiping motion on a touch screen. Swipe motions initiate when the user presses their finger against the screen, and may only be actually registered if the finger travels through the screen a certain distance, which is customizable. The user may want to attach a Swipe Gesture to an object in order for it to detect swiping motions, and it may also be annexed an F2USprite script in order to make it viewable. Swiping gestures need hit area sprites in order to be interactable. They can be made to detect swipes either in one or all of the

four cardinal directions, horizontally, or vertically. The following properties belong to swipe gestures:

- Min Swipe Distance: Defines the minimum amount of distance the dragging has to travel before the gesture detects the swiping.
- Detect On Drag Distance: Defines the distance the dragging has to travel before firing the OnDrag method.

Tap Gesture

Tap gestures are used to detect simple taps on a touch screen. As a matter of fact, a tap is a quick pressing motion initiated by the user's finger. There may be several tappable components in a scene, each with its own hit area. An object can be attached with an F2UTapgesture in order for it to become tappable. The following is a list of the properties owned by tap gestures:

- Throw Event On Press: Determines whether to throw an event when the gesture is tapped.

Resolution Controllers

Anchor

An anchor is defined as a point where an object is fixed. Currently, an object can be fixed to nine possible points: the top right of the screen, the center right, the left right, the middle left, the middle center, the middle right, the bottom right, the bottom center or the bottom right. In order to anchor an object to a specific point, attach an F2UUIAnchor script on the object you want to fix, and then select the anchor point by modifying the UIAnchor property. You may also modify the side offset so that the object gets fixed a certain distance away from the actual point of anchoring. For the anchoring to work as

expected, they must be attached to an animation with a “hit_area” layer. Moreover, the layer itself must reside at point (0, 0) on the animation.

Resolution Controller

A resolution controller is a utility made to make sure every single object fits onto the proposed screen coordinates and size, given their world coordinates and size. This utility is mostly used to crop out backgrounds and the like, but may also be used to scale other types of objects. The related script only works properly if F2UUIResolutionController is added to the script execution order before the default execution time. By default, this class will scale up or down the images so that the related camera’s background is shown as little as possible. For the controller to work as expected, they must be attached to an animation with a “hit_area” layer. Moreover, the layer itself must reside at point (0, 0) on the animation.

Text Inputs

Text inputs are the way Flash to Unity provides to capture text written by the user. By default, text inputs are unrestricted and hold a finite amount of characters. Text inputs need to have focus before the user is allowed to write on them, and only a single text input controller may have focus. Text inputs change their color when they come into focus, and they may turn red whenever if the user writes up invalid characters. Text inputs have several write modes that accept a different input pattern each; the following list will enumerate such write modes:

1. Default: Unrestricted mode. The user may write any character on this mode.

2. ASCIICapable: ASCII mode. The user may only write ASCII characters up to 0x7F.
3. Numbers and punctuation: Numerical mode: Allows the user to write numbers alongside some special characters such as dot, comma, minus and plus.
4. URL: Link mode. Allows the user to write a valid URL address. Currently unrestricted for desktop users.
5. Number pad: Basic numerical mode. Allows the user to write numbers.
6. Phone pad: Telephone mode. Allows the user to write up valid phone numbers. Numbers may have country code, but no extensions.
7. Email address: Mail mode. Allows the user to write up valid email addresses according to the RFC-2822 standard. The format currently does not support IPv6 address domains, nor it supports the usage of double quotes or backslashes on literals (double quote enclosed portions).

The components related to the text inputs are listed as follows:

- Style Key: The key used for the text's style.
- Throw Event on Press: Determines whether an event is thrown when the text input comes into focus.
- Max Text Length: The maximum amount of characters allowed for the text input.
- Type: The type of input field. The field can be set to be a regular textbox with single or multiple lines or a password field.
- Selected Color: The color the field is tinted with when it's on focus.
- Keyboard Type: The kind of input validation the text field allows for. It also defiens the type of keyboard that appears on-screen when using purely tactile devices. They are enumerated as follows:
    - Default: Default keyboard. Anything can be typed on this mode.
    - ASCIICapable: Keyboard capable of ASCII symbols. Validation fails when the user types a character not included in the regular ASCII character set.

  - NumbersAndPunctuation: Keyboard capable of numerical input, plus some punctuation. Validation fails when a non-numerical, non-special character symbol is written.
  - URL: Keyboard capable of URL input. Validation fails when a non-conforming URL is written.
  - Number: Keyboard capable of numerical input. Validation fails when a non-numerical character is written.
  - Phone: Keyboard capable of phone inputs. Validation fails when a non-standard phone number is written.
  - NamePhone: Keyboard capable of phone and name inputs. Identical to Default.
  - Email: Keyboard capable of email inputs. Validation fails when a mail address that does not conform to the RFC-2822 standard is typed.
- Keyboard Hide Input: Determines whether to hide the keyboard. Has no effect on non-mobile devices.
- Keyboard Auto Correction: Determines whether to turn on the keyboard's auto-correction feature. Has no effect on non-mobile devices.
- Keyboard Secure: Determines whether to use secure keyboards. Has no effect on non-mobile devices.
- Keyboard Alert: Determines whether to alert the keyboard. Has no effect on non-mobile devices.
- Keyboard Placeholder text: Determines a placeholder text for the keyboard. Has no effect on non-mobile devices.