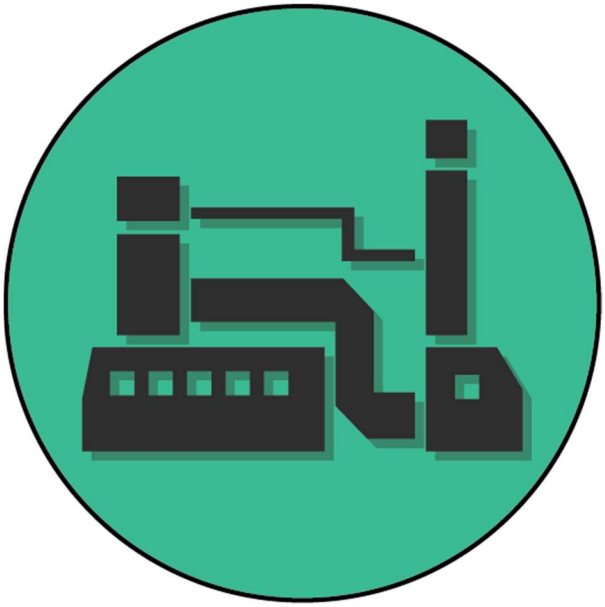
ANNONA SOFTWARE
https:/games.annonasoftware.com
annonasoftware@gmail.com



UI AUTOMATOR

MANUAL AND API REFERENCE

UI Automator is a lightweight UI management and automation system for Unity 3D, designed to simplify the use of Unity's UI system and streamline the workflow for both new and experienced devs.

TABLE OF CONTENTS

Contents

Introduction and Package Summary

INTRODUCTION

Congratulations! You are now the proud owner of **UI Automator**, a lightweight system designed to quickly bootstrap a new project with a UI system that is reliable and easy to understand. This manual will provide the references and explanations to get you started. If you have any questions about how this asset works, have found a bug, or you have an idea for a useful feature, don't hesitate to let us know at annonasoftware@gmail.com.

PACKAGE SUMMARY

There are three cooperative pieces of the **UI Automator**: the `UIManager`, `ModalManager`, and Script Automation, called `UIAutomatorEditor` in the scripts. This is a full source project, and so if you have any questions about internal functionality or simply want to modify the scripts, you can do so to your heart's content in an IDE of your choice. All scripts are included in the namespace `UIAutomator` and can be accessed by including `using UIAutomator;` before your class declarations.

UIManager

The `UIManager` is a singleton class that inherits from `MonoBehaviour` and must be placed on a Game Object in your scene. If you change scenes in your game design but want to maintain the UI throughout, simply add the `[DontDestroyOnLoad]` attribute to the `UIManager` class declaration. This class tracks the active UI elements in the scene and handles their creation and destruction, as well as serving as a convenient place to access UI elements if you need to do so. Each UI element works by inheriting from the abstract class `Element`, and the required overrides give the data necessary for the `UIManager` to do its job.

ModalManager

The `ModalManager` acts very similarly to the `UIManager`, but instead of dealing with all UI elements it handles one, the Modal. Modals are the dialog boxes that appear in all operating systems and in many games that warn, inform, or otherwise draw attention to some aspect of the game or program and require user action to remove, typically by clicking one of one or more buttons on the modal itself. There are two types of modal included in this asset: `OneButtonModal` and `TwoButtonModal`. The dialog or description on the modal, the button names, and the actions the buttons perform `onClick` are all completely configurable, as is the design of the modal itself, which is drawn from a prefab. This asset includes a basic prefab to get you started; if you want to modify the design or layout, make sure to either use the same labels for the Game Objects holding the UI components, or re-assign those references in the `ModalManager` script.

INTRODUCTION AND PACKAGE SUMMARY

Script Automation

The second half of this asset, after the managing behaviors at runtime, is the editor-based script automation. In our own game designs, the creation and assignment of variables always was a drag on the workflow; creating a new class is a repetitive process, and while copying and pasting the relevant methods saves time, it can also lead to bugs, particularly Null Reference Errors if a mistake is made. This editor window, available under the Tools->UI Automation menu, takes a selected object or one provided in the object field and creates a new UI class script, shells of its methods, and even names and assigns variables based off the UI components in the element. The use of the editor automation is covered in detail in the "UI Automation" portion of this manual.

Miscellaneous

Also included are the handful of scripts needed for the asset to work: the abstract class that all UI elements must inherit, called `Element`; a `Util` class that includes two very useful methods, used in the **UIAutomator** scripts and available for your own use wherever you would like; and the `Singleton` class, which implements the singleton design pattern to ensure that only a single instance of `UIManager` and `ModalManager` are active at any one time.

Getting Started and Workflow

PREPARE SCENE

- Create a new Game Object as a "Manager", or use one you have already set aside for that purpose, and attach the `UIManager` and `ModalManager` scripts as components on the Game Object.
- Don't forget to assign one of the scripts on the Game Object the `[DontDestroyOnLoad]` attribute (put just above the class declaration in the script file) if you intend for the managers to persist through scene changes.

CREATE YOUR UI ELEMENT PREFABS

- Design your UI elements however you'd like, focusing on the overall look first.
- Once you've designed the UI element to your satisfaction, use UI Automation (see below) to create most of the script's framework for you!

WRITE YOUR SCRIPTS

- Open the script file and make the UI functional by adding callbacks and any additional methods.
  Make sure once you've finished your script to drag it onto your designed prefab.
- In the public fields on the managers, assign your UI element prefabs in the `Element Prefabs` array field on `UIManager` and your modal prefab on the `Modal` field. Expand the `Element Prefabs` array as necessary for all your prefabs.
- As usual, once you've saved the prefab to your Assets folder delete the prefab instance in your scene.

TEST IN PLAY MODE

- Everything should be good to go! Test your scene and fix any errors that occur.

UI Design

OVERALL DESIGN

UI Automator uses a manager-run UI design, where a centralized singleton manager keeps track of all the various potential UI elements and their sorting order, activation, and destruction. Using prefabs means highly customizable UIs without a lot of coding legwork.

For a UI element to be manipulated by the manager, it must inherit from and override the public methods and properties of the `Element` class.

```
public class MyUIClass : Element
{
}
```

Properties

`Element` provides three properties that must be set in code (done automatically by the UI Automation, or manually as below): `Category`, `Prefab`, and `CanEscape`. Prefab is simply a reference to the prefab attached to the instantiated class, often simply `this.gameObject`; `CanEscape` is a bool that, if true, allows the player to close out the UI element with a keyboard shortcut. Category has five possible values for its enum, and these are largely open frameworks that you can build your own structure into. You can treat them however you would like in your design, but there are some built-in ways they're handed that can, of course, be rewritten if desired.

CATEGORY DESCRIPTIONS

NAME	BEHAVIOR	USE
Element	Cannot co-exist with other elements; if a new element is called, the old will be destroyed.	The root or base UI element. Used for your standard run-of-the-mill in-game UI. Sorted lowest of all the UI elements.
Subelement	Multiple subelements can exist on screen simultaneously. Subelements are children of the active element: if the element is destroyed, all active subelements will be as well.	Used for smaller elements dependent on the larger parent element, like color pickers, name choosers, or other simple purposes. Sorted higher than elements.
Popup	Cannot co-exist with other popups; if a new	Typically used for popup or temporary messages; fundamentally identical to elements,

	popup is called, the old will be destroyed.	but a different category to allow for flexibility in UI design. Sorted higher than subelements.
Menu	Cannot co-exist with other menus; if a new menu is called, the old will be destroyed.	Like popups and elements but sorted higher than popups. In our designs used for utility and main menus, and often marked to pause the action or otherwise halt play.
Debug	Only one prefab can be a debug, and so toggling will create or destroy the instance.	The highest sorted element of all, and the only one for which there can be but a single prefab. Used for a debug, editing console, etc, and often bound to a single key typical for such windows, as "F1" in the DemoScene.

Methods

There are three methods (in addition to the two standard Unity methods, `Start()` and `Update()`) required as overrides and auto-generated by the UI Automation: `Assign()`, `Close()`, and `Refresh()`.

- `Assign()` is called by the `UIManager` immediately after the element's prefab is instantiated, and is the place to initialize and perform any housekeeping. It is automatically filled with variable assignments by the UI Automation script.
- `Close()` is called by the `DestroyElement(Element e)` method and its overload in `UIManager`, and is a place to destroy or otherwise clean up any threads before the UI element is destroyed.
- Refresh() is called automatically on every class that inherits from Element by the `UIManager.RefreshAll()` method, described in more detail in the "API Reference" section of this manual. Here is a good place for filling the data of your UI; many users call `Refresh()` from within `Assign()` after initialization to set the first frame of data.

DEMO SCENE

The DemoScene included with this asset shows a very basic UI and how seamlessly the system operates. Plese check it out for some good usage examples, but do not look to it for graphic design. It's a very simple system using only Unity defaults; the art and design of the system is up to you!

UI Automation

LOCATION AND SETUP

The two menu items are located under Tools->UI Automation and have shortcuts. The first option uses default settings to generate a script from the selected object; the second option creates an editor window that lets you drag and drop a prefab in for script generation with customization.

SCRIPT GENERATION OPTIONS

NAME	TYPE	DESCRIPTION
UI Game Object	Object Field	By default, set to the selected object in the hierarchy; drag and drop any prefab with a canvas attached into the field to generate a script for that object. Only objects with a Canvas component attached will work, otherwise the generation will cancel and a warning will be logged in the console.
Create a UI Folder	Toggle	If checked, the script generation will place the new script in the path "Assets/Scripts/UI/[new script name]". If unchecked, the script will simply be placed in "Assets/". If there is no "Assets/Scripts/UI" path, one will be created.
Element Category	Dropdown	The element category identifies the type of UI element the new script (and thus the prefab) will have. Choices are Element, Subelement, Popup, Menu, and Debug, explained in the "UI Design" section of this manual. These can be changed later by modifying the enum in the `Category` property of the generated script.
Variable Prefix Type	Dropdown	There are several different conventions for naming UI variables in C# scripts. Three choices are provided here: "Full word", "Initial and underscore", and "none". For an object named `Foo` containing a `Button` component, these three options would look like this: *Full word*: `Button buttonFoo;` *Initials and underscore*: `Button b_Foo;` *None*: `Button foo;`
Escapable	Toggle	If checked, the player can press the "Escape" key (defaulted to ESC in the `UIManager`, but can be modified to fit any design) to close out of the UI elements in descending order of sorting layer priority: debug, menu, popup, subelement, and element. If unchecked, the UI element is "unescapable," useful for designs where the player is required to interact with a UI element before proceeding.

API Reference

MODALMANAGER.CS

`TwoButtonModal(string dialog, string left, string right, UnityAction leftAction, UnityAction rightAction)`

Summary: A typical modal with dialog and two buttons.

Parameters:

	dialog	(`string`) The modal's descriptive text.
	left	(`string`) The left button's label.
	right	(`string`) The right button's label.
	leftAction	(`UnityAction`) The left button's method name to call on button click.
	rightAction	(`UnityAction`) The right button's method name to call on button click.

Remarks: Unity actions are called as methods without the closing parentheses; in other words like `MyClass.MyMethod`, rather than `MyClass.MyMethod()`. This obviously prevents use of method calls that require parameters. The solution is to use a delegate in place of the method. See below or the DemoScene that shipped with this package for an example.

```
Void Assign()
{
        buttonClose = transform.FindDeepChild("Close").GetComponent<Button>();
        buttonClose.onClick.AddListener(delegate
                {UIManager.Instance.DestroyElement(this); });
}
```

In the example above, `UIManager.Instance.DestroyElement(Element e)` requires a parameter of type `Element`. Therefore, a delegate pattern was used.

`OneButtonModal(string dialog, string left, UnityAction leftAction)`

Summary: A modal for use on warnings and dialogs with only a single button.

Parameters:

	dialog	(`string`) The modal's descriptive text.
	left	(`string`) The left (and only) button's label.
	leftAction	(`UnityAction`) The left button's method name to call on button click.

Remarks: Just as in the TwoButtonModal above, Unity actions are called as methods without the closing parentheses; in other words like `MyClass.MyMethod`, rather than `MyClass.MyMethod()`. This obviously prevents use of method calls that require parameters. The solution is to use a delegate in place of the method. See below or the DemoScene that shipped with this package for an example.

```
Void Assign()
{
        buttonClose = transform.FindDeepChild("Close").GetComponent<Button>();
        buttonClose.onClick.AddListener(delegate
                {UIManager.Instance.DestroyElement(this); });
}
```

In the example above, `UIManager.Instance.DestroyElement(Element e)` requires a parameter of type `Element`. Therefore, a delegate pattern was used.

`Cancel()`

Summary: Assigned to one or both modal buttons to destroy it.

Parameters: None.

Remarks: Very straightforward: just kills the modal Game Object. This serves as a demonstration of the most basic method that can be used as a UnityAction with any part of **UI Automator**'s management system.

UIMANAGER.CS

`ToggleElement(Element e)`

Summary: The heart of the UI management system for **UI Automator**; a publicly accessible method that determines which of three actions should occur: create menu, destroy old and create new, do nothing.

Parameters: *e* (`Element`) Takes the `Element` class of a particular UI prefab.

Returns: `bool`; `true` if UI element toggled on, `false` if toggled off.

Remarks: This is the central method around which the UI system revolves. It functions on a basic assumption that is typical of modern UI systems: when the user attempts to toggle a UI element, if that element already exists it should be removed; if the user intends to toggle a different element at the same level of the hierarchy, the old should be removed and the new activated; and if no active element exists, the desired element should be activated. Specific behavior for the different categories of elements is found elsewhere in this manual.

`ToggleElement(GameObject o)`

Summary: Overload for `ToggleElement(Element e)`. See above.

Parameters: *o* (`GameObject`) The UI element's prefab.

Returns: `bool`; `true` if UI element toggled on, `false` if toggled off.

Remarks: Overload for the `ToggleElement` method. Takes the prefab itself rather than an element, which allows for a more streamlined call using the UIManager's `public Element[] ElementPrefabs` array.

Example:

```
buttonSubElement.onClick.AddListener(delegate
{UIManager.Instance.ToggleElement(UIManager.Instance.ElementPrefabs[1]); });
```

`DestroyElement(Element e)`

Summary: Forces the given UI element active in the scene to close.

Parameters: *e* (`Element`) Takes the `Element` class of a particular UI prefab.

Remarks: While you can use `ToggleElement` for destruction or creation, and it will handle the various scenarios that could arise, sometimes it's beneficial to ensure that the element will always be destroyed—on a "Close" button, for instance. This method serves that need.

`DestroyElement(GameObject o)`

Summary: Overload for `DestroyElement(Element e)`. See above.

Parameters: *o* (`GameObject`) The UI element's prefab.

Remarks: Overload for the `DestroyElement` method. Takes the prefab itself rather than an element, which allows for a more streamlined call using the UIManager's `public Element[] ElementPrefabs` array.

Example:

```
buttonClose.onClick.AddListener(delegate
{UIManager.Instance.DestroyElement(UIManager.Instance.ElementPrefabs[1]); });
```

`RefreshAll()`

Summary: Refreshes all active UI elements that inherit from `Element`.

Parameters: None.

Remarks: While not used in the DemoScene, this is a very useful method for use in the more complicated UI setups. Common use would be if one UI element includes an interactive element to change player data, for instance, and other existing UI elements need to be refreshed to display the new data. Not recommended for use in an `Update()` loop, as it forces a redraw of all active UI elements.

UTIL.CS

`FindDeepChild(this Transform aParent, string aName)`

Summary: Locates and returns a child transform that matches the provided `string` parameter regardless of nested depth.

Parameters: *aName* (`string`) The child you are attempting to locate.

Returns: `Transform`.

Remarks: This useful `Transform` extension method is not specifically UI-related, but is included because of its great utility. It searches within the parent transform to locate a child nested within any number of children; in other words, a grand-n child. Used by default by the **UIAutomator** Editor window when assigning UI elements to their script variables, which reduces the need for expensive `GameObject.Find()` calls.

`SetLayerRecursively(GameObject parent, int layer)`

Summary: Sets the sorting layer for the game object and all of its children and nested objects.

Parameters: *parent* (`GameObject`) The parent for all the nested objects to be layered.

Layer (`int`) The Unity-designated layer number to be assigned.

Returns: None.

Remarks: Another useful method that overrides default behavior. Used in the **UI Automator** for reassigning a UI element and all of its nested children and components to the appropriate sorting layer.

Common Mistakes

1. **Failing to use `[Class].Instance.[Method]` to refer to singleton classes**

Because they need to access `MonoBehaviour` while ensuring there is one and only one instance of the class in the scene at any moment in time, both `UIManager` and `ModalManager` are singletons. When accessing their methods, this means you need to qualify your method call with `UIManager.Instance.[method call]` and `ModalManager.Instance.[method call]` respectively.

2. **Duplicate Game Object names when automating script creation**

One of **UI Automator**'s most useful time-saving aspects is the auto-generation of UI scripts. However, because the automated script builder uses the Game Object's name as its identifier for the script variable, and because Unity allows the creation of identically named Game Objects, this can cause compiler errors in your automated script. The solution is simple: give your Game Objects unique names! The most common tripping point is with Unity's `Button`, `Dropdown`, and `Slider` components, since they automatically name the child Game Objects "Text", "Label", and so on.

3. **Forgetting to put the auto-generated script on to the UI prefab**

This asset does not, for a number of reasons, automatically put the generated script onto the UI prefab that it is meant to accompany. You will need to do that yourself, and a failure to do this will make that particular UI element fail to operate.

Bugs? Questions? Feature Requests?

Annona Software strives to produce seamless, bug-free, and efficient assets. However, we know that glitches can occur, and we are quick to address any issues that arise.

If you have any questions or concerns about how the **UI Automator** asset works, or would like to make a bug report or feature request, please contact us via the contact information below. We will work diligently to address the issue.

Thank you for your purchase, and happy automation!

ANNONA SOFTWARE
https:/games.annonasoftware.com
annonasoftware@gmail.com