### 

Lucine.UISystem

**UI manager**



Lucine.UISystem

## Introduction

This documents explains how the Lucine.UISystem works. You can also find a doxygen documentation of alls classes in the Documentation folder.

## Some definition

### Screens

A screen is the base class for all UISystem. A screen is something that need to be displayed. Ideally it is a GameObject with all the needed children to display something that may be interactable or not. This root game objects has a script on it that inherit from UIScreenController<>. It is responsible to handle interactions on these elements. A Screen has parameters that are inherited from UIScreenParameters and contains in and out transition that are inherited from UITransition to handle the way they appears and disappear when asked by the user. A Screen is supported on a layer.

Screens have an unique id (the name of the GameObject supporting it). This id will be used to indicate to which screen orders are given. Screens keep trace of their visibility. A screen implements also 4 callback on which other can register. OnInTransitionFinished, OnOutTransitionFinished, OnCloseRequest, OnScreenDestroyed. All these information are implemented on screen using the IUIScreenController Interface.

### Panels

Panels are the simple type of screen, it is aimed to support information that will be displayed with no interaction, such as life counter, what you have in hand and so on. It is inherited from UIScreenController with UIPanelParameters. Panels can be showed or hidden. Panels are to be stored in UIPanelLayer hierarchy. There are not supposed to be interactive but nothing impeach you to make it so. They are always on the position of the hierarchy you defined. If you want to change them in the hierarchy of the panel layer, you have to do it yourself.

### Windows

Window are inherited from UIScreenControllers with UIWindowParameters. Windows can be Opened and Closed. When opened a windows always go to the top of other displayed windows. Windows have basic parameters that tells if it should be closed when loosing foreground and if they are popup. If they are popup they will set to front when opened and a BlackBackground GameObject (prefab furnished in sample) will be placed just below it in order to make everything below grayed.. Window are all stored in UIWindowLayer Hierarchy. Currently there’s no history management, but UIWindowLayer may be used to implement that kind of features if needed.

UIWindowLayer if front of UIPanelLayer by specification. UIWindowLayer and UIPanelLayer are specialization of UILayerController.

We never speak to screens directly (even if we could). To Open/Close windows and Show/Hide panels we use the UIController. UIController is the manager of UISystem. It requires a Canvas and a graphics raycaster. UIController owns two layers reference, the UIPanelsLayer and the UIWindowsLayer. It finds them automatically in its children on initialization. UIController is also responsible to disable interaction with all Unity UI elements, so it need to register to some events to be notified when interaction need to be disabled or enabled. UIController script need to be called before everything on startup to initialize everything (UIPanelsLayer & UIWindowsLayer).

### Parameters

UIScreen are templated with parameters, UIWindow are UIScreen with UIWindowParameters, UIPanel are UIScreen with UIPanelParameters. Right now only UIWindow defines parameters specific to windows. You will have to define specific UIScreen types with their appropriate UIParameters inherited from UIScreen, UIWindow or UIPanels while developing the controller for each screen of your interface.

### Events

UIController needs to know when it needs to disable or enable Unity UI events. In UISystem windows disable interaction during opening and closing transitions. The window layer is responsible to inform the listeners of these events that a windows is closing or opening. That’s why UIWindowsLayer expose two Action : DisableInteractionRequest and EnableInteractionRequest that UIController will listen to do the job.

The UIWindowsLayer itself need to know when screen start to appear and disappear to handle this case. That’s why it register to screen Actions : OnInAnimationFinished and OnOutAnimationFinished to do the job. It also registers to screen OnCloseRequest action to handle the case where a windows close is asked by the window itself (press a button for instance, public UnityClose function of UIWindow that send the event could be linked in button event trigger)

UIController

We’re going through him to do things

OpenWindow/CloseWindow => this will be asked to be done by the UIWindowLayer

ShowPanel/HidePanel => this will be done by the UIPanelLayer

Can disable/enable user interaction (raycaster of the main canvas). This behaviour should be asked by UIWindowLayer when a window is blocking window (such as a popup). It could be done by registering to an event, OnBlockScreenEnable, OnBlockScreenDisable

Can give the Main Canvas and the render camera. This camera should only display UI layer

This object root should be conserved when loading scene that’s why it is a Singleton

Make it a prefab

On initialization the UIController finds in his children the UIPanelLayerController and the UIWindowLayerController and initialize them

The UIController can close the current window asking to the UIWindowLayer to do it. So we can ask to close the current window.

We can ask if a panel or window is open

We can maybe have a HideAll function to hide all panels and windows

HideAllPanels only

CloseAllWindows only

UILAYER<TScreenController>

Registered screens

Show/Hide screen with or without UIParameters

Initialize created the registered screens

Register/unregister screen

IsScreenRegistred

Subscribe to Screen callback OnScreenDestroyed to remove register screens

UIWindowLayer = UILayer<UIWindowController>

Current window is stored here

List of active transition

Function to know if a transition is running

ShowScreen with or without UIparameters

HideScreen with or without UIparameters

HideAll close all windows registered to this layer

CloseCurrentWindow close the current window

Should register/unregister to UIScreen OnInOutTransitionFinished to store/remove currenttransition (transition are started by UIScreen with showscreen/hidescreen methods)

Should register/unregister to UIScreen OnCloseRequest to be able to handle to window close request sent by the UIScreen

UIPanelLayer = UILayer<UIPanelController>

ShowScreen with or without UIparameters

HideScreen with or without UIparameters

IsPanelVisible

UIScreen<TParameters>

Show

Play UITransition when showing

Call intransitiondone when finish in transition

Hide

Play UITransition when closing

Call outtransitiondone when finish out transition

Callback for Transition In done, callback for transition out done, callback for close request, callback for screen destroy

UIWindow = UIScreen<UIWindowParameters>

Hide window when foreground is lost ?

Is Popup indicates that should be drawn before all with dark screen

Callback closerequest when Close function called by Unity events is called (UnityClose)

UIPanel = UIScreen<UIPanelParameters>

Just like a UIScreen right now may think about some specialization later

UITransition

Base class for UITransition

These are monobehaviour with only one function to do wanted animation.

Parameter of this function is the transform to move, and a callback to call when finished

The Event System

The event system is a tiny tools to help sending and receiving notifications of events.

You define an eventType inherited from Event (event with no parameter)

public class ApplicationQuitEvent : Event { }

Or from the templated version which accept one parameter

public class OnPanelClosedEvent : Event<DemoPanel> { }

Then elsewhere you Add a listener to the defined type and set a function that will be called when the event will be fired. The function should have no parameter or the templated parameter type

Events.Instance.TypeOf<DemoPanel.OnPanelClosedEvent>().AddListener(OnPanel1Closed);

When no more need of the notification you can remove the listener

Events.Instance.TypeOf<DemoPanel.OnPanelClosedEvent>().RemoveListener(OnPanel1Closed);

When you want to fire the event you dispatch it !

Events.Instance.TypeOf<OnPanelClosedEvent>().Dispatch(this);

The Events class is a singleton that let you have a global pool of events. But you also can have pools dedicated to special features if you want. It this case you have to define your own pool using the class EventPool (that’s what Events do)

That’s so easy so use it !

The Sample

The sample shows a main window with an input field and 3 buttons.

One button to show a first panel, another button to show a second panel, then a quit button that when clicked open a popup to confirm that you want to exit or cancel.

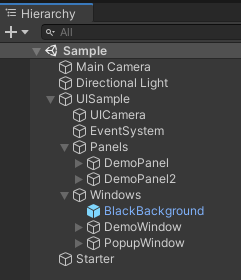
When a panel is open the corresponding button in the main window should be disabled to avoid to click one more time.

When a panel is closed, the corresponding button in the main window should be enabled to permit to open the panel once again.

When clicking on the quit button of the main window, the starter GameObject should kill the application.

When click on cancel the dialog box should close and enable background windows to become active once again.

Here is the hierarchy of the Sample scene



You can see a Main Camera for 3D rendering, the directional light to light the scene and the UISample which is the UIController of the demo. Under the UIController you can find a UICamera that render only UI layer with no clear but depth so be on top of 3D scene. The Unity EventSystem is in there but it could be anywhere. It must be unique. Next you have the Panel Layer (in which we can find all panel1 and panel2 ) on top of the Windows Layer where you can find DemoWindow and PopupWindow.

The Starter GameObject starts the demo with its UIStarter script, it opens the DemoWindow window using the referenced UIController.

void **Start**()  
{  
 *// the UIManager should be initialized before opening windows  
 // so UIController script execution order is before other scripts* uiController.OpenWindow("DemoWindow");  
   
 *// register to ApplicationQuitEvent* Events.Instance.TypeOf<ApplicationQuitEvent>().AddListener(OnCloseMainWindow);  
}

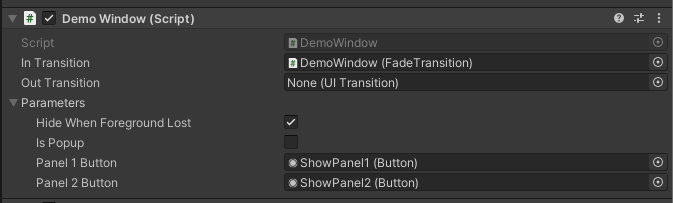
It also register to an event (ApplicationQuitEvent) that will be sent by the popup when validation of quit will be done by the user. The callback function is OnCloseMainWindow() that unsubscribe to event and quit application.

void OnCloseMainWindow()  
 {  
 *// unsubscribe to ApplicationQuitEvent* Events.Instance.TypeOf<ApplicationQuitEvent>().RemoveListener(OnCloseMainWindow);  
   
#if UNITY\_EDITOR  
 UnityEditor.EditorApplication.isPlaying = false;  
#else  
 Application.Quit();  
#endif  
 }



The window DemoWindow is handled by the demowindow script. Demo window will have to know which show panel button has been pressed in order to deactivate it when the associated panel opens. So DemoWindow script is a UIWindow<DemoWindowParameters>. The DemoWindowParameters extends UIWindows parameters to define two Button that will be set in inspector.

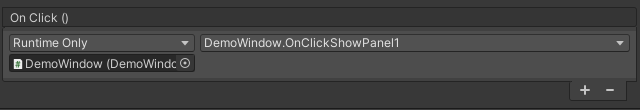
*/// <summary>  
/// Custom parameters to have buttons that launch panel 1 and panel 2 in order to disable them  
/// while panels are open  
/// </summary>*[Serializable]  
public class DemoWindowParameters : UIWindowParameters  
{  
 public Button **panel1Button**;  
 public Button **panel2Button**;



We can see here the link to the two buttons. We also see that this window has an in transition but no out transition that means that the window will fade according to the associated fade transition when opened. This window is close when another window is opened (get foreground) but is not a popup (these two parameters are the default parameters of all UIWindows)

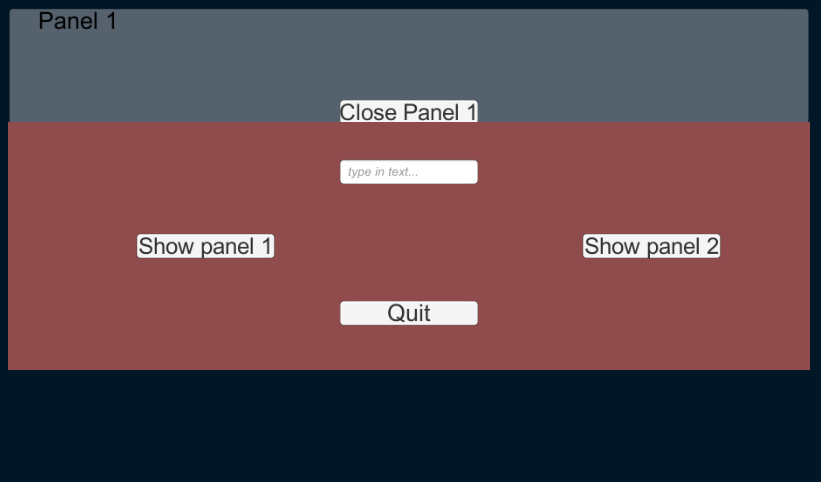


A fadein transition in 0.5s with a in sine ease.

In the inspector we set the callback of the show panels buttons to DemoWindow:OnClickShowPanel1 and DemoWindow.OnClickPanel2 so that this functions are called by Unity on click event. 

*/// <summary>  
/// this is called by Unity when Click on OpenPanel1 button  
/// </summary>*public void **OnClickShowPanel1**()  
{  
 *// disable the corresponding button* Parameters.panel1Button.enabled = false;  
 *// add a listener to the OnPanelClosedEvent (OnPanel1Closed)* Events.Instance.TypeOf<DemoPanel.OnPanelClosedEvent>().AddListener(OnPanel1Closed);  
 *// open the demoPanel* UIController.Instance.ShowPanel("DemoPanel");  
}

We can see that the panel1button referenced in the DemoWindowParameter is disabled to avoid another click. Then DemoWindow register to the DemoPanel.OnPanelClosedEvent that will be sent by the DemoPanel script when panel is closed. Then it Show the DemoPanel panel. The same is done in OnClickShowPanel2 but with panel2 button.



Here we can see the panel 1 opened after the click on show panel1

Each panel is handled by the same script DemoPanel.

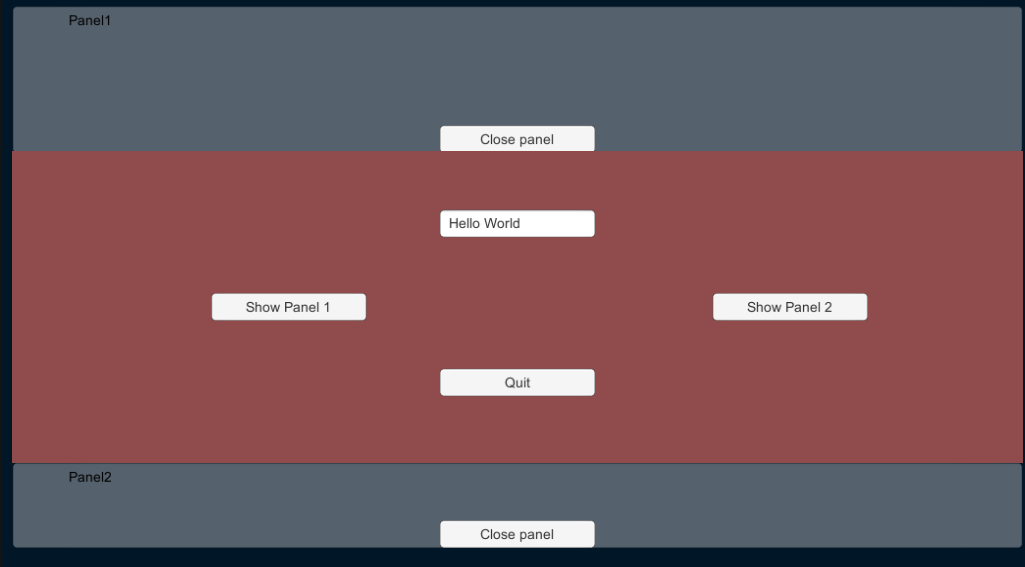
Unity callback for close panel button is linked to DemoPanel:OnClosePanel.

*/// <summary>  
/// Called by unity when the close panel button is pressed  
/// </summary>*public void **OnClosePanel**()  
{  
 *// dispatch the event to all listener giving in parameter the panel which is closed* Events.Instance.TypeOf<OnPanelClosedEvent>().Dispatch(this);  
   
 *// hide the panel* Hide();  
}

This function dispatch the OnPanelClosedEvent with one parameter which is the instance of the script (this may permit to identify which panel close button has been pressed to the listeners in our case no need since we registered one listener for each button). So all registered listener will receive the message that the panel close button has been pressed. Then the OnClosePanel only Hide the panel.

The DemoWindow:OnPanel1Closed then remove its listener and enable the show panel 1 button. Same for panel 2.

*/// <summary>  
/// Call by DemoPanel script when dispatching events  
/// </summary>  
/// <param name="panel">The panel that throw the event</param>*void OnPanel1Closed(DemoPanel panel)  
{  
 *// we can now remove the listener for this fonction* Events.Instance.TypeOf<DemoPanel.OnPanelClosedEvent>().RemoveListener(OnPanel1Closed);  
 *// enable back the button* Parameters.panel1Button.enabled = true;  
}



Here is the app with the two panels opened. You may have noticed that in and out transition are not the same in the both cases. It is possible since each gameobject has fadetransition linked to its in transition and out transition and that they are parametred the same way.

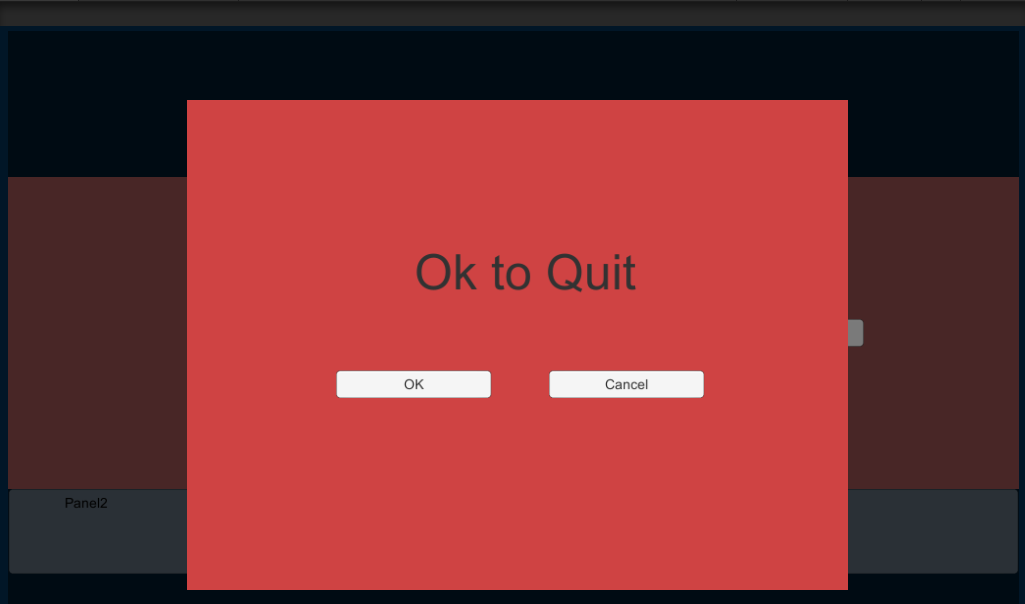
Now only the panel 2 is opened.

When click on quit a popup appears that ask for confirmation of the action.

You may notice the dark background, and that the DemoWindow is still here even if it is indicated in the parameters that the windows hide when it lost foreground.

This is because the popup window is a popup and that popups never close the window under it.

The dark screen is automatically added when popups are opened to avoid being possible to click on the underlying windows. This is a prefab name Blackbackground and hidden by default but appears at the last sibling just before the popup appears at the last sibling.



The popup window is handled by DemoPopup script. The demo popup is a simple window so DemoPopup extends UIWindow with no special custom parameters.

Unity will call the OnClickOnOk() using its event system when the button Ok is pressed

public void **OnClickOnOk**()  
{  
 UIController.Instance.HideAllPanels();  
 UIController.Instance.CloseAllWindows();  
 Events.Instance.TypeOf<ApplicationQuitEvent>().Dispatch();  
}

This lead to close all panels and all windows and then dispatch the ApplicationQuitEvent

ApplicationQuitEvent Class is defined in UIStarter because it’s it that want to listen to this event. It extends Lucine.Helpers.Event class (event with no parameter) just like the following.

In the case of cancel click Unity event system calls DemoPopup:OnClickOnCancel

public void **OnClickOnCancel**()  
{  
 UIController.Instance.CloseWindow("PopupWindow");   
}

public class ApplicationQuitEvent : Event { }

Notice that the function only close the PopupWindow. Nothing else to do, last window will be on top of the screen and can be interacted with.

Notice that all texts in the application are using TextManager to get corresponding text of Ids. No text in Text components in the application interface, but UIText component just added on each text gameobject with an id. See TextManager documentation for more information.

Lucine.Helpers.TextManager

The role of the text manager is to keep trace of all text of the application.

All texts are defined in an xml file using Ids and Text with the following format

<?xml version="1.0" encoding="utf-16"?>

<TextDatabase xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<Texts>

<Entry Id="ID\_SHOWPANEL1" Text="Show panel 1" />

<Entry Id="ID\_SHOWPANEL2" Text="Show panel 2" />

<Entry Id="ID\_TITLEPANEL1" Text="Panel 1" />

<Entry Id="ID\_TITLEPANEL2" Text="Panel 2" />

<Entry Id="ID\_CLOSEPANEL1" Text="Close Panel 1" />

<Entry Id="ID\_CLOSEPANEL2" Text="Close Panel 2" />

<Entry Id="ID\_QUIT" Text="Quit" />

<Entry Id="ID\_QUITMSG" Text="Ok to quit ?" />

<Entry Id="ID\_OK" Text="OK" />

<Entry Id="ID\_CANCEL" Text="Cancel" />

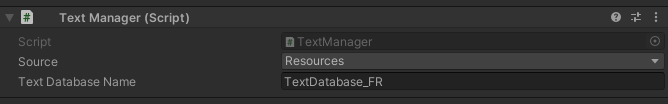
<Entry Id="ID\_ENTERTEXT" Text="type in text..." />

</Texts>

</TextDatabase>

This xmlfile can loaded by the TextManager from a file in StreamingAssets folder, or from a Text resource in a Resources directory (in which case the extension of the file has to be .txt even if xml)

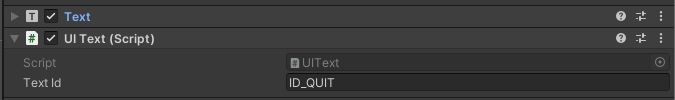
TextManager is a singleton that can be place where you want. In the sample it is located on the Starter GameObject.



You can select the TextDatabase name (no extension has to be given when using resources) and where to load it from. Resources => Resources folder, StreamingAssets, streaming assets folder (in this case you have to include the extension of the file). In the sample the file is present both in streamingassets and resources, in order you can test both methods.

When the file is loaded it fires an OnTextDabaseChanged event, you can register to this event to be notified when something changed in the texts.

That’s what does UIText component



This component requires that the gameobject on which it is, has also a TextComponent.

The only information to set on it is the id of the text in the textdatabase that need to be set in the text component.

On startup the TextComponent is cached and UIText register itself to the OnTextDataBaseChanged event. That way when the text is ready, it is alerted and it can just ask to the TextManager the text that correspond to the id it has and set it in the text component.

*/// <summary>  
/// Change the text when event fired  
/// </summary>*public void OnTextChanged()  
{  
 m\_Text.text = TextManager.Instance.GetText(m\_TextId);  
}

The TextManager is useful to avoid having text in the application. All texts are outside the application.

This way translation are made easier.