DIALOGS, MENUS, AND NOTIFICATIONS FOR THE UNITY GUI

USER GUIDE

V1.00 (March 2016)

TABLE OF CONTENTS

1 - Getting Started 4
1.1 What can you do with uDialog? 4
1.2 Quick Start – In Editor 4
1.3 Quick Start – Using Code 5
2 – The uDialog Component 8
2.1 Controls 8
2.2 Basic 8
2.3 Title 8
2.4 Content 9
2.5 Modal 10
2.6 Auto Close 10
2.7 Theme 10
2.8 Events 13
2.9 Animation 13
2.10 Audio 14
2.11 Dragging and Resizing 14
2.12 Focus 15
2.13 Misc 15
2.14 References 15
3 - Notifications 16
3.1 Notification Panels 16
3.2 Adding Notifications to a Notification Panel 17
4 – Wrapping existing content 19
4.1 What can be wrapped? 19
4.2 Wrapping content in the editor 19
4.3 Wrapping content in code 20
5 – Fluent API 21
5.1 Overview 21

1 - Getting Started

1.1 What can you do with uDialog?

uDialog allows you to easily:

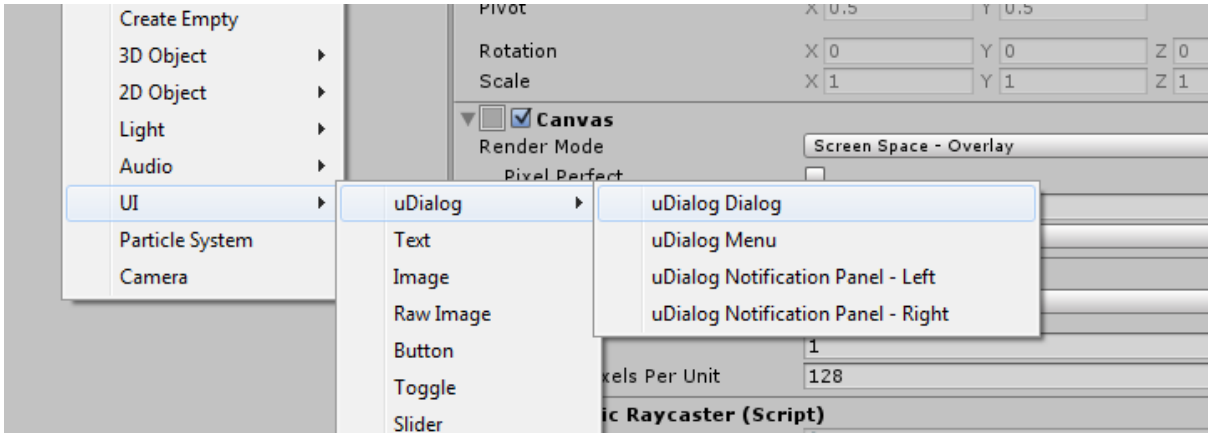
- Create message boxes
- Create confirmation dialogs
- Create menus
- Create notification messages
- Wrap existing UI content in themed, draggable, and resizable windows

1.2 Quick Start – In Editor

Adding a new **uDialog** to a scene is easy – simply open the **GameObject** menu (or right click the **Canvas** or other parent object in the scene hierarchy) then select a prefab from the **UI -> uDialog** list. Several prefabs are available by default:

- **uDialog Dialog**: This prefab is a standard simple dialog.
- **uDialog Menu**: This prefab is a dialog configured as a vertical menu.
- **uDialog Notification Panel Left / Right**: These prefabs allow you to add a new **Notification Panel** to the scene (preconfigured on either the left or right side of its parent).

If there is no **Canvas** and/or **EventSystem** in your scene, they will automatically be added for you.



For details on how to customize the uDialog created by this menu, see section 2 **– The uDialog Component.**

Once a **uDialog** has been added in this manner, it will (by default) be visible when the scene starts. It can be set to be hidden by default by changing the value of its **Visible On Start** property. If it is not visible when the scene starts, then it can be shown at any time by calling its **Show()** function (either through code, or via a Unity Event).

1.3 Quick Start – Using Code

Creating uDialog instances using code is easy.

Start by including the UI.Dialogs namespace, e.g.:

```
using UnityEngine;
using UI.Dialogs;
```

This will grant you access to the following static functions:

- `uDialog.NewDialog()`
  This function will instantiate a new uDialog instance. By default, the instance will be based on the **uDialog_Default** prefab, but you can specify other custom prefabs by passing their name as an argument. Please note that any custom prefabs must be in a **Resources/Prefabs** folder.

- `uDialog.NewMenu()`
  This function will instantiate a new uDialog instance, using the **uDialog_Menu** prefab. The primary difference between this prefab and a regular dialog is that the buttons are arranged vertically instead of horizontally.

- `uDialog.NewNotification()`
  This function will instantiate a new uDialog instance, using the **uDialog_Notification** prefab.
  This function is best used in combination with a **uDialog_NotificationPanel** object.

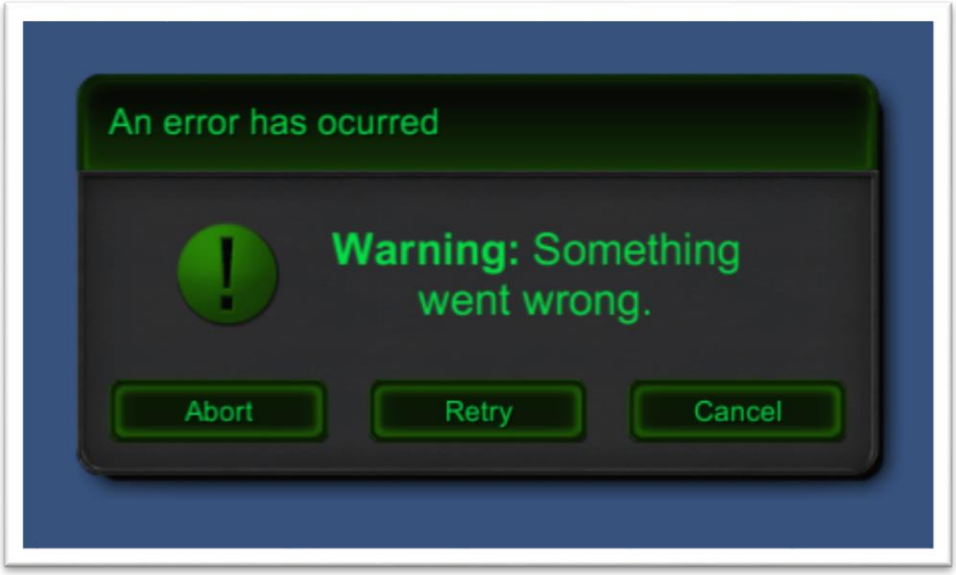
Here are a few simple examples of these functions in action:

A simple dialog:

```
public void ShowSimpleDialog()
{
    uDialog.NewDialog()
           .SetTitleText("Simple Message Dialog")
           .SetContentText("This is a simple Message Dialog")
           .AddButton("Close", (dialog) => dialog.Close());
}
```

A simple error/confirm dialog:

```
public void ShowConfirmDialog()
{
    uDialog.NewDialog()
           .SetColorScheme("Green Highlight")
           .SetThemeImageSet(eThemeImageSet.SciFi)
           .SetIcon(eIconType.Warning)
           .SetTitleText("An error has ocurred")
           .SetShowTitleCloseButton(false)
           .SetContentText("<b>Warning:</b> Something went wrong.")
           .SetDimensions(400, 200)
           .AddButton("Abort", () => { /* Call abort functionality */ })
           .AddButton("Retry", () => { /* Call retry functionality */ })
           .AddButton("Cancel", () => { /* Call cancel functionality */})
           .SetCloseWhenAnyButtonClicked(true);
}
```



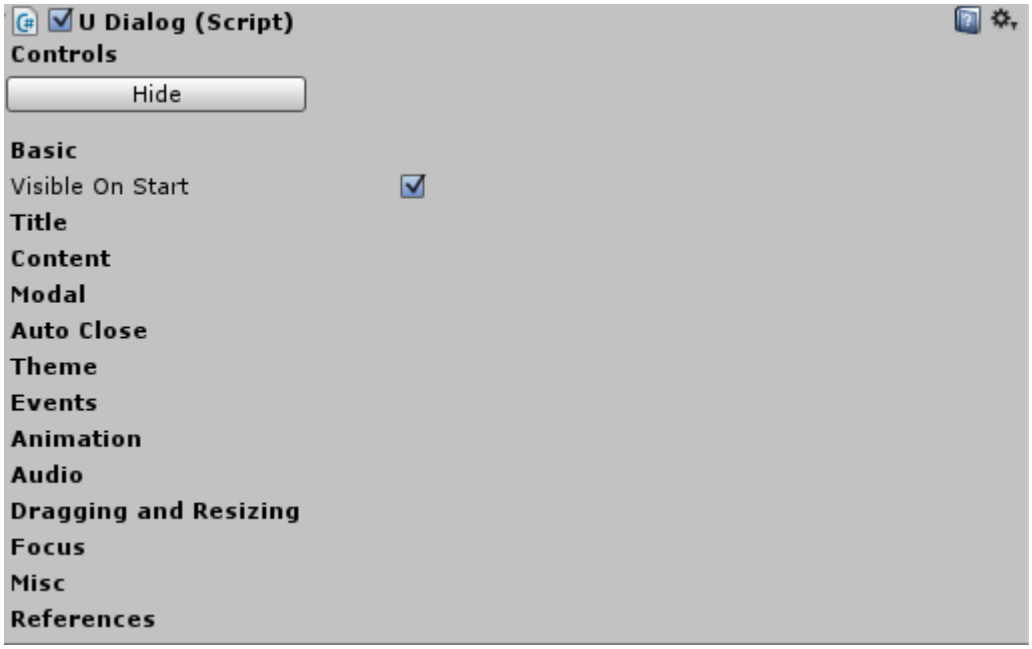
A simple menu:

```
public void ShowSimpleMenu()
{
    uDialog.NewMenu()
           .SetTitleText("Simple Menu")
           .SetColorScheme("Light")
           .AddButton("Button 1", (dialog, button) => { button.ButtonText = "1 Clicked!"; })
           .AddButton("Button 2", (dialog, button) => { button.ButtonText = "2 Clicked!"; })
           .AddButton("Button 3", (dialog, button) => { button.ButtonText = "3 Clicked!"; })
           .AddButton("Button 4", (dialog, button) => { button.ButtonText = "4 Clicked!"; })
           .AddButton("Close", (dialog) => dialog.Close());
}
```

A simple menu after 'Button 3' has been clicked.

2 – The uDialog Component

The **uDialog** component provides a lot of customization options for you to use when setting up your own dialogs, menus, notifications, or content windows.



The default view of a uDialog component

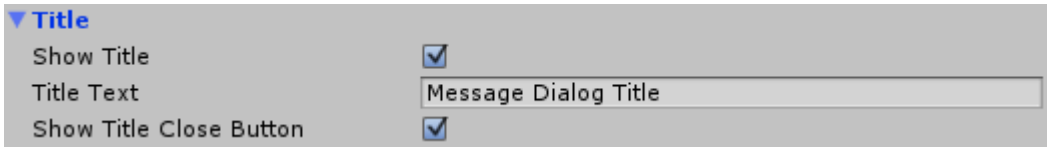
2.1 Controls

The **Controls** section simply provides a button which can be used to show or close the dialog in the editor. Please be advised that when a dialog is closed, its GameObject is made inactive, and as such events such as `Start()` will not be fired – what this means is that, should a **uDialog** object not be active when the scene is loaded (or when it is added to the scene), it will be necessary to show it manually (via its `Show()` function).

2.2 Basic

Visible On Start – If this property is cleared, then this **uDialog** will be hidden as soon as the scene is loaded or when it is added to the scene in play mode. As per 2.1, please be advised that a hidden dialog will **not** be shown on start until `dialog.Show()` is called.

2.3 Title

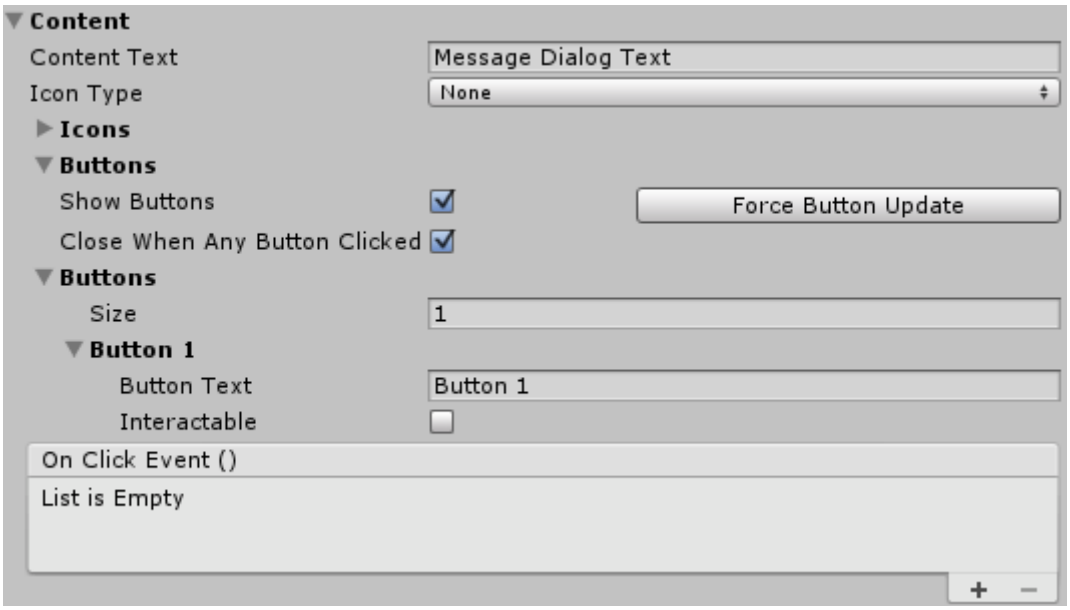


Show Title – If this property is set, then the title bar will be shown. If not, it will be hidden.

Title Text – This property controls what text is shown on the title bar (if any).

Show Title Close Button – This property controls whether or not the title bar should have a close button (on the right-hand side).

2.4 CONTENT



Content Text – This property specifies what text should be shown in the dialogs content window. If no text is specified here, then the space normally taken up by the text will be made available to other components in the window, e.g. buttons.

Icon Type – This property specifies what type of icon should be shown by this dialog (if any). If no icon is shown, then the space normally taken up by the icon will be made available to other components in the window, e.g. content text.
If you wish to take manual control over the icon, you can change the Icon Type to *Custom*. You can then set the icon image manually (by locating its **Image** component in the hierarchy and setting the sprite).

Icons – This property allows you to override the default uDialog icons (Information, Warning, Question) and replace them with your own.

2.4.1 BUTTONS

Show Buttons – If this property is set, then any buttons set for this **uDialog** will be visible. If not, they will be hidden.

Close When Any Button Clicked - If this property is set, clicking any of the buttons in this **uDialog** will close the dialog once the action associated with the button has finished executing.

Force Button Update – When this button is clicked, the buttons used by this **uDialog** will be completely rebuilt based on their template. This is useful if, for example, you edit the template and wish to update existing buttons. Editing the button template will be discussed in further detail in **Section 3 – uDialog Structure.**

Buttons – This property specifies a list of buttons to be shown on the dialog. To add new button, either set the **Size** property of the array to the number of buttons you need, or right-click on an existing button and select

"**Duplicate Array Element"**. Similarly, you can remove existing buttons by right-clicking on them and selecting **"Delete Array Element"**.

Button Properties

Button Text – This specifies what text should be visible on the button.

Interactable - This specifies whether or not the button should be *Interactable*. If this value is set to false, the button will not be able to be clicked.

On Click Event() - This is a list of standard unity events to trigger when this button is clicked.

2.5 MODAL

Modal - If this property is set, this **uDialog** will become modal. An overlay will be placed behind the dialog to prevent users from clicking on anything else. The appearance of this overlay can be customised as part of the color scheme.

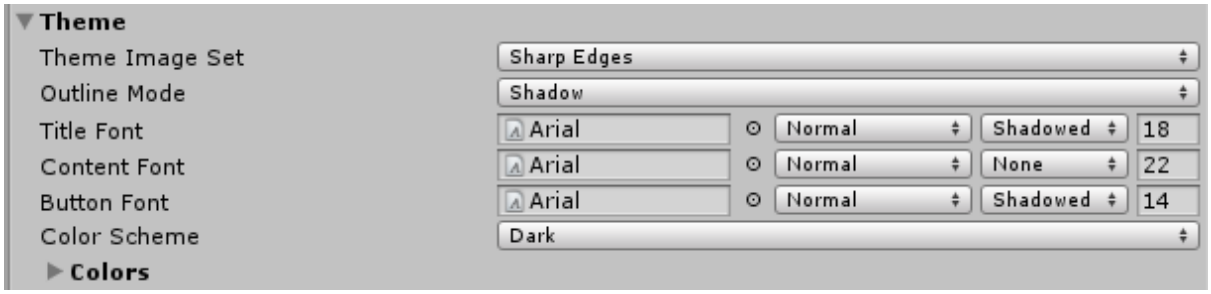
Cose When Screen Overlay Is Clicked – If this property is set, then this **uDialog** will be closed if the user clicks anywhere on the screen overlay (only applicably to modal dialogs, menus, and windows).

2.6 AUTO CLOSE

Auto Close – If this property is set, then this **uDialog** will automatically close **Auto Close Time** seconds after it is shown. This is primarily used by notifications, but can be used by any type of **uDialog**.

Auto Close Time – Specifies how long to wait before closing this dialog if **Auto Close** is set.

2.7 THEME



Theme Image Set – This property allows you to select what base image set should be used by this **uDialog**. At the time of writing, the options available are: *Rounded Edges, Sharp Edges, No Images, Fantasy, Sci-Fi, Angular, and Custom*. More sets may be added in the future.

The *No Images* set is precisely that, it does not use any images at all, instead it will simply use colors as specified by the color scheme.

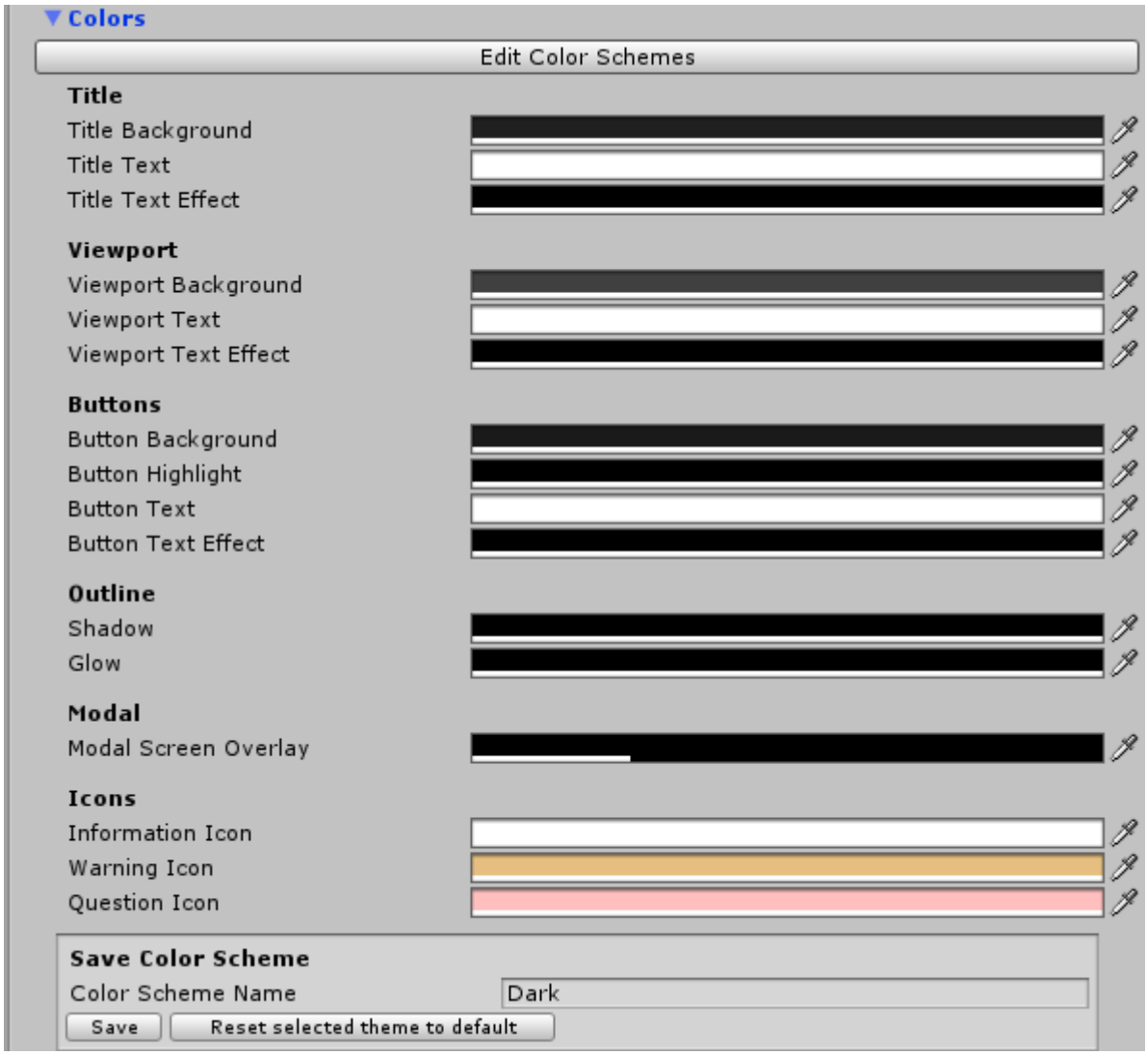
Selecting the *Custom* set will instruct the **uDialog** to leave the images it uses alone – allowing you to manually switch them out with images of your choice.

Outline Mode – This property allows you to specify how the dialog is outlined, if at all. The options available are: *Shadow, Glow, and None*. The colors used by the *Shadow* and *Glow* options are specified by the color scheme.

Title, Content, and Button Font – These properties control the font, style, text effect, and size of the title, content, and button text respectively. The text effects available are *Shadowed, Outlined, and None*. The color used by these text fields and any text effects used is specified by the color scheme.

Color Scheme – This property allows you to select a color scheme to be used by this **uDialog**. By default, there are several color schemes for you to choose from, but it is also possible for you to edit them, or even add your own custom schemes. More details on this on the following page.

2.7.1 Color Schemes



The color scheme editor allows you to select the colors used for a particular color scheme. **Please be advised that any *unsaved* changes to a color scheme will be lost if the scene is started. If you wish to edit a color scheme, it is highly recommended that you a) Select the color scheme you wish to edit from the *Color Scheme* list, then b) Change the color scheme to *Custom*. This will prevent your changes from being lost, and, once you are satisfied with the changes, you can type the name of the desired color scheme in the *Color Scheme Name* field and click *Save* to override it.**

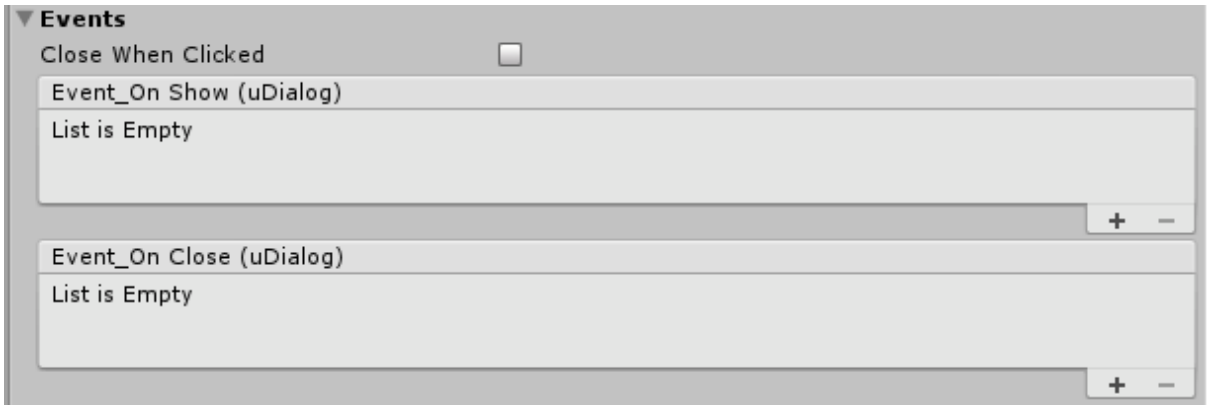
Any changes made to colors here will be shown on the dialog immediately. If you wish to create a color scheme to be used by a single dialog (or template prefab) only, then you can select the *Custom* color scheme and set the values as needed, and they will not be overridden. If, however, you wish to create or edit a color scheme used by multiple dialogs/windows/etc., then it is recommended that you add a new color scheme (by typing a new name in the **Color Scheme Name** input field and clicking **Save**). When a color scheme is saved, all other dialogs which use the scheme in the current scene will be updated to match.

Reset selected theme to default – this button will allow you to reset the selected theme to its default values. This is only available for a select few default themes, namely *Light*, *Dark*, and *Plain*. For other themes (including

themes you create yourself), this button will be replaced with a **Delete selected theme** button instead which will allow you to remove a color scheme entirely (any dialogs using the selected theme will have their color scheme changed to *Custom* and will otherwise be unaffected).

The **Edit Color Schemes** button:
uDialog color schemes are saved in a serialized asset located in the **uDialog/Resources/Configuration** folder. Clicking this button will select this asset in the editor and allow you to edit it directly.
As with the editor provided, you can add new schemes, delete and rename existing schemes, and change their color values through Unitys standard inspector interface. In general, it is probably easier to edit themes directly through a **uDialog** instance, but in some cases it may be preferable to use this interface (as no **uDialog** instance is required). You can access this file directly by locating it in the **Project** window and selecting it.

2.8 EVENTS

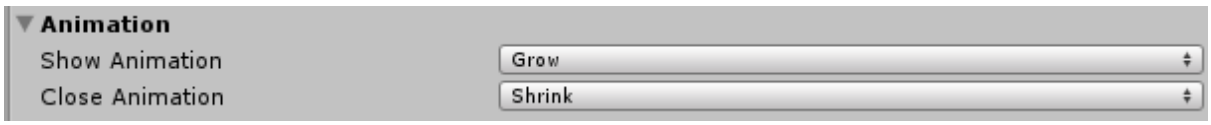


Close When Clicked - If this property is set, this **uDialog** will be closed if the user clicks anywhere within the dialog area.

Event_On Show – This is a standard Unity Event to be called when this dialog is shown. An optional argument (consisting of the **uDialog** object itself) may be used.

Event_On Close – This is a standard Unity Event to be called when this dialog is closed. An optional argument (consisting of the **uDialog** object itself) may be used.

2.9 ANIMATION



Show Animation – This property specifies what animation (if any) to play when the dialog is shown. The options available are: *Slide In (from any direction), Fade In, and Grow*.

Close Animation – This property specifies what animation (if any) to play when the dialog is closed. The options available are: *Slide Out (in any direction), Fade Out, and Shrink*.

2.10 AUDIO

On Show Sound - Specifies an audio clip to be played when this **uDialog** is shown.

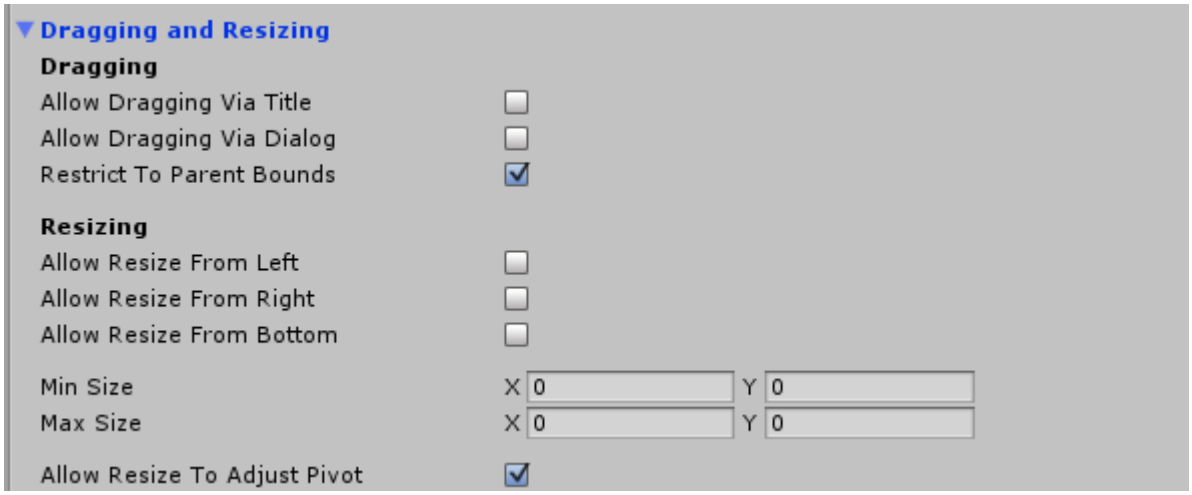
On Close Sound – Specifies an audio clip to be played when this **uDialog** is closed.

On Button Click Sound – Specifies an audio clip to be played whenever any button in this **uDialog** is clicked.

Audio Volume – Controls the volume of all sounds played by this **uDialog**.

Audio Mixer Group – Allows you to assign this **uDialog** to a Unity Audio Mixer Group (optional).

2.11 DRAGGING AND RESIZING



2.11.1 DRAGGING

Allow Dragging Via Title – If this option is enabled, this **uDialog** can be dragged by its title.

Allow Dragging Via Dialog – If this option is enabled, this **uDialog** can be dragged from any point within the dialog.

Restrict To Parent Bounds – If this option is enabled, then this **uDialog** will be not be able to be dragged outside of its parents bounds – e.g. if the **uDialog** is contained directly within a **Canvas** element, then it will not be allowed to be dragged off the edge of the screen. If it is contained within another object, it will not be permitted to be dragged outside of that objects bounds (as defined by its **RectTransform**).

2.11.2 RESIZING

Allow Resize From Left/Right/Bottom – Each of these properties controls whether or not the **uDialog** can be resized by dragging the left/right/bottom borders respectively. If both the Right and Bottom options are enabled, then this **uDialog** will also be able to be resized from the bottom-right corner. Similarly, if both the Left and Bottom options are enabled, then this **uDialog** will be able to be resized from the bottom-left corner.

Min Size – This property specifies the minimum size this **uDialog** can be resized to.

Max Size – This property specifies the maximum size this **uDialog** can be resized to. If the default value of (0, 0) is used, then the **uDialog** can be increased in size with no restrictions.

Allow Resize To Adjust Pivot – If this property is enabled, then the **uDialog** will be permitted to adjust its pivot value in order to make resizing easier. With this property enabled, for example, dragging the right border of the **uDialog** will move that side of the window while leaving the other sides exactly where they are. If this property is disabled, then the uDialog will be resized according to its current pivot setting. By default, the pivot of (0.5, 0.5) means that the dialog would be resized in all directions equally.

2.12 FOCUS

Focus On Click – If this property is set, then this **uDialog** will be brought to the front whenever it is clicked. In Unity terms, this basically means that it will be made the last child of its parent object.

Focus On Mouse Over – If this property is set, then this **uDialog** will be brought to the front whenever the mouse enters its borders.

2.13 MISC

Destroy After Close – If this property is set, then this **uDialog** object will be destroyed once it is closed. If you intend to re-use the object, then this value should be left unset.

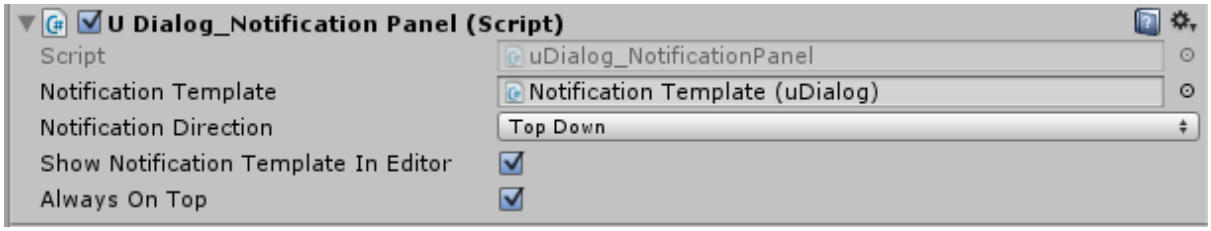
2.14 REFERENCES

This section defines various referenced GameObjects/etc. which **uDialog** needs in order to function. Under normal circumstances, there should be no need to modify any of these values.

3 - NOTIFICATIONS

3.1 NOTIFICATION PANELS

Notification Panels define the area in which notification messages are shown. They consist of a RectTransform, a **uDialog** template for the notification messages, and several options.



As with other **uDialog** components, notification panels can be added to the scene through the **GameObject** menu (located in the **UI -> uDialog** section). There are two default options available, **Left**, and **Right**. These prefabs are set up to show notification messages on the left and right side of the screens respectively (although you are free to reposition them wherever you wish).

Notification Template – This field references the template used for this Notification Panel. Under normal circumstances, there should be no need to change this value, although if you wish you can substitute another **uDialog** object in place of the existing template.

Notification Direction – This field specifies which direction notifications should stack. The options available are *Top Down*, and *Bottom Up*.

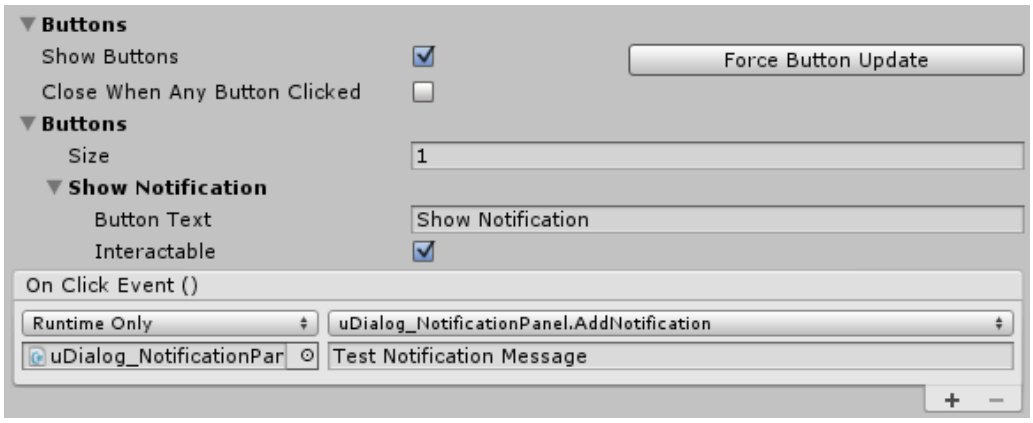
Show Notification Template In Editor - If this field is set, then the **Notification Template** will be visible in edit mode (but not in play mode). This may make it easier to customise the appearance of the template.

Always On Top – If this field is set, this Notification Panel will always set itself as the last child of its parent container so as to always be on top of anything else in the scene.

By default, the Notification Panel has no appearance of its own, but if you wish you can add an **Image** component to provide a background, or customise it in any other way you wish.

3.2 Adding Notifications to a Notification Panel

3.2.1 Using The Editor



Notifications can be added to a Notification Panel by setting up a Unity Event which calls the **AddNotification** function. In the example above, a **uDialog** button has been set to add a simple notification message when it is clicked.

Please note that only very simple notifications can be shown in this manner. For control over additional properties, such as the icon, you will need to show notifications using code instead.

3.2.2 Using Code

The Notification Panel provides several overloads of the **AddNotification** function:

`void AddNotification(string notificationText)`
This function adds a new notification message using the notification text provided.

`uDialog AddNotification(string notificationText, eIconType? iconType)`
This function adds a new notification message using the notification text provided and sets the icon to the specified type. This function returns a reference to the **uDialog** (Notification) so that you can customise it further if you wish.

`uDialog AddNotification(string notificationText, Sprite iconType)`
This function adds a new notification message using the notification text provided and sets the icon to the specified **Sprite**. This function returns a reference to the **uDialog** (Notification) so that you can customise it further if you wish.

Here is a simple example of a notification message being added to a notification panel:

```
public void ShowSimpleNotificationMessage()
{
    NotificationPanel.AddNotification("Test Notification", eIconType.Warning)
                     .SetContentFontSize(10, 20); // Set the font size to best fit,
                                                  // between size 10 and 20
}
```

A simple test notification

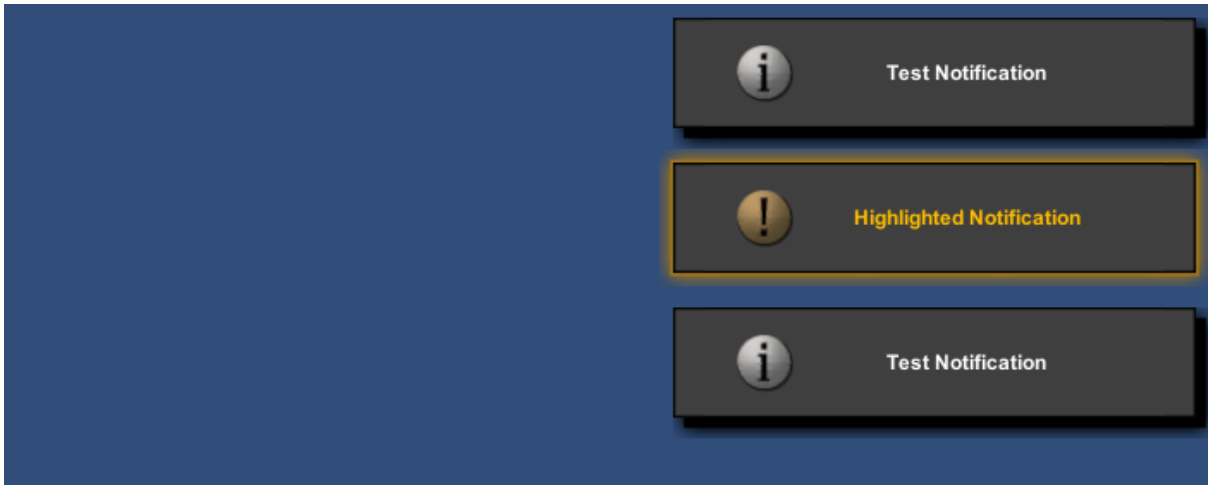
Here is a second example, of a notification message which has been highlighted (in between two other notifications):

```
public void ShowNotifications()
{
    ShowSimpleNotificationMessage();
    ShowHighlightedNotificationMessage();
    ShowSimpleNotificationMessage();
}

public void ShowSimpleNotificationMessage()
{
    NotificationPanel.AddNotification("Test Notification", eIconType.Information);
}

public void ShowHighlightedNotificationMessage()
{
    var notification = NotificationPanel.AddNotification("Highlighted Notification",
                                                         eIconType.Warning)
                                        .SetOutlineMode(eOutlineMode.Glow);

    notification.Color_Glow = new Color(0.75f, 0.5f, 0.0f, 0.75f);
    notification.Color_ViewportText = new Color(1f, 0.75f, 0.0f);
}
```



4 – WRAPPING EXISTING CONTENT

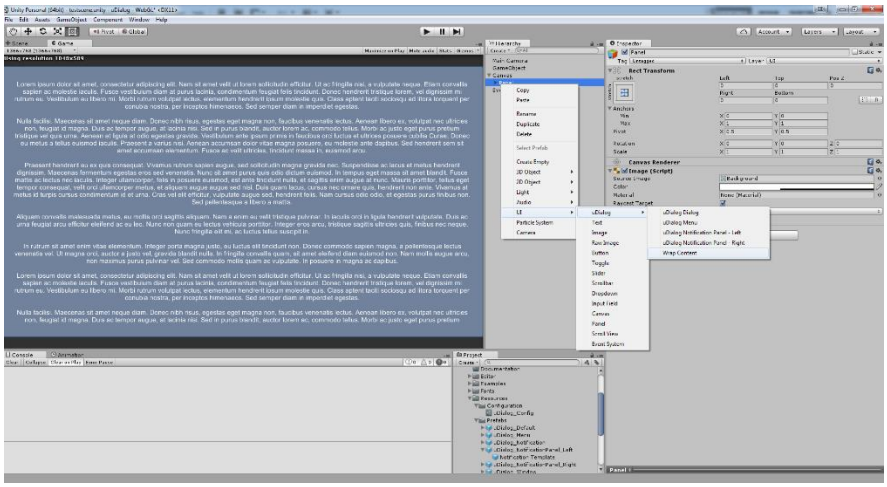
uDialog allows you to wrap existing UI content within a **uDialog** window – for example, you could create a form in the editor, and then place it within a **uDialog** window.

4.1 WHAT CAN BE WRAPPED?

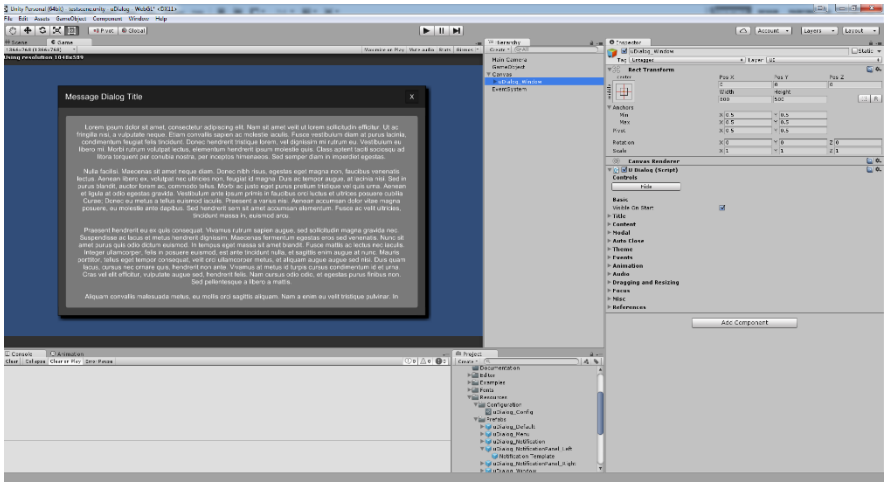
Essentially, any UI element with a **RectTransform** can be wrapped. In some cases, however, there may be layout issues – for example, wrapping a ScrollRect may trigger Unity Exceptions when the **uDialog** is resized (although these exceptions do not prevent it from working).

4.2 WRAPPING CONTENT IN THE EDITOR

Content can be wrapped in the editor by first selecting it in the **Hierarchy**, then right clicking and selecting the **UI -> uDialog -> Wrap Content** menu item.



Wrapping Content



Result

4.3 WRAPPING CONTENT IN CODE

Content can easily be wrapped by passing a reference to it to **uDialog.SetContent()**. For example:

```
public RectTransform Content;
public void ShowContentDialog()
{
  // You can set the content value in code by finding/instantiating the object you wish to wrap,
  // or you can set the reference in the editor

    uDialog.NewDialog()
           .SetDimensions(800, 600)
           .SetAllowDragging()
           .SetResizeable()
           .SetContent(Content);
}
```

5 – FLUENT API

5.1 OVERVIEW

You can find the API documentation online here.

Most of the functions you will need are located in the static uDialog_Fluent class. These are extension functions, which means that they can be called directly on **uDialog** objects.

The Fluent API is designed to make working with **uDialog** as simple as possible, and keep the code as readable as possible. In some cases, the functionality provided by the Fluent API can be accessed by modifying properties of the **uDialog** object directly, but generally it is better to use the API functions as they will ensure that any updated values are put to use correctly.

The Fluent API is modelled after C# functionality such as LINQ – each function in the API returns the **uDialog** object, so that functions can be chained one after the other, allowing you to configure the majority of **uDialog** properties in a clear and easy to read manner.

For example, you could configure a simple **uDialog** object like this:

```
public void ShowSimpleDialog()
{
    var dialog = uDialog.NewDialog();

    dialog.SetModal(true);
    dialog.SetContentText("Content Text");
    dialog.SetColorScheme("Light");
    dialog.SetThemeImageSet(eThemeImageSet.RoundedEdges);
}
```

But, because each of the Fluent functions returns the uDialog object, you could also configure it like this:

```
public void ShowSimpleDialog()
{
    uDialog.NewDialog()
           .SetModal(true)
           .SetContentText("Content Text")
           .SetColorScheme("Light")
           .SetThemeImageSet(eThemeImageSet.RoundedEdges);
}
```

Which method you choose is up to you; either should work fine. You can also use a combination of them, in order to customise properties not covered by the API (such as specific color values used by the **uDialog**):

```
public void ShowSimpleDialog()
{
    var dialog = uDialog.NewDialog()
                        .SetModal(true)
                        .SetContentText("Content Text")
                        .SetColorScheme("Light")
                        .SetThemeImageSet(eThemeImageSet.RoundedEdges);

    dialog.Color_ViewportText = Color.gray;
}
```