

Unity Composition Interface

PiRho Soft

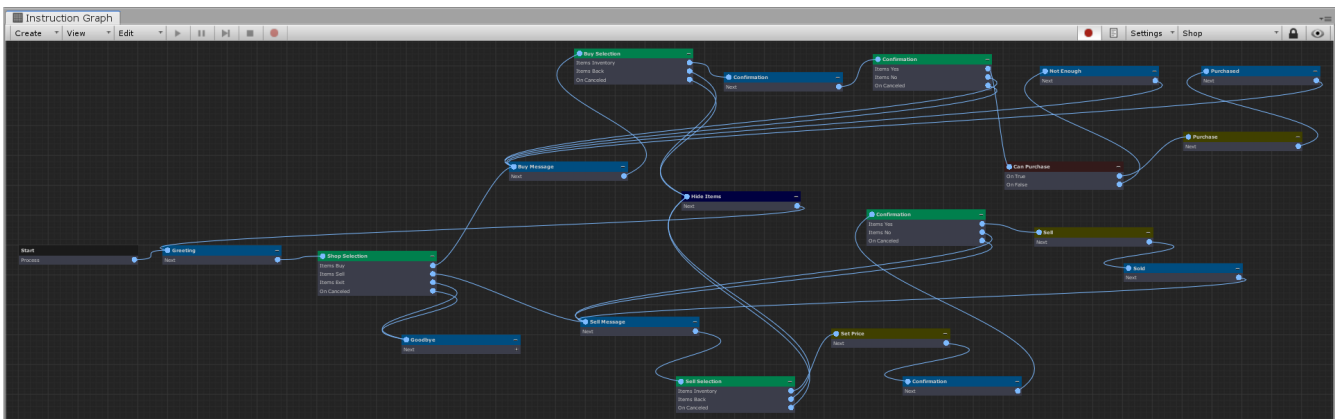
Overview.....	1
Workflow	1
Topics	2
Interface Controls.....	3
Message Controls	4
Message Node	4
Message	4
Input	5
Custom Message Nodes.....	5
Menus	8
Selection Controls.....	8
Menu Items	8
Selection Nodes.....	8
Menu Item Templates	8
Other Helpful Behaviours	9

Overview

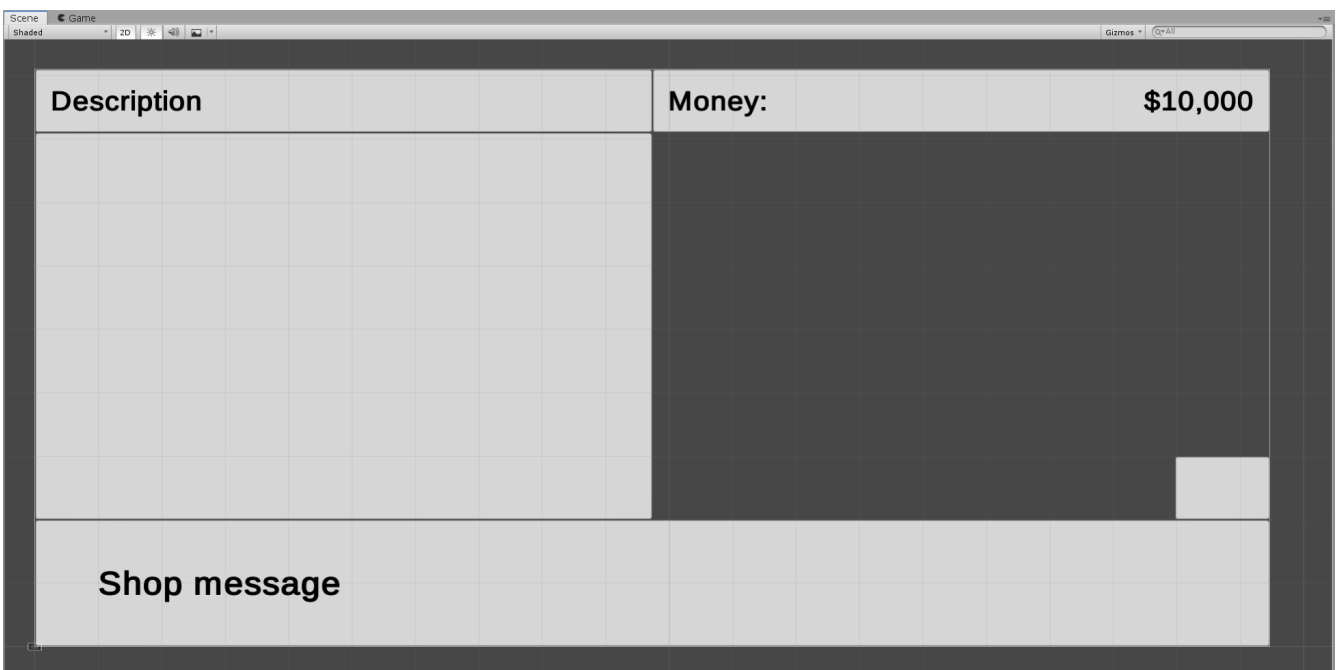
The interface system is used in conjunction with [Unity's UI](#) system make creating and interacting with common UI elements in a game much simpler. Interfaces work with [graphs](#) for sequencing things like dialogs, selections, menus, etc, without the need for extensive scripting. The main components of the interface system are Interface Controls. These are activated and deactivated from graphs and control the behaviour of UI elements in the scene. Messages and Selections are the two built in controls provided. Interface controls are designed with extensibility in mind; each can be customized for desired behaviour and other custom controls can be created.

Workflow

The following is an example of a [graph](#) the defines a dialog sequence constructed almost entirely of [Message Nodes](#) and [Selection Nodes](#) to create a simple shop.



And the scene it references which has a [Message](#) controlling the text at the bottom, and [Selection Controls](#) controlling the selection boxes on the sides.



See the "Shop" scene in the Shop project to view this example.

Topics

1. [Controls](#)
2. [Messages](#)
3. [Menus and Selections](#)

Interface Controls

[Interface Controls](#) are a [MonoBehaviour](#) that acts as the base class for all controls in the interface system. They should be attached to objects in a UI scene that are to be shown and hidden by the composition system (like dialog boxes, and menus). When loaded an Interface Control always starts inactive until the `Activate()` method is called. This usually happens from a [Show Control Node](#) on a [graph](#). Interface Controls also maintain a list of other [GameObjects](#) in a scene that should be activated and deactivate along with this control. To deactivate a control use a [Hide Control Node](#) or manually call the `Deactivate()` method from script.

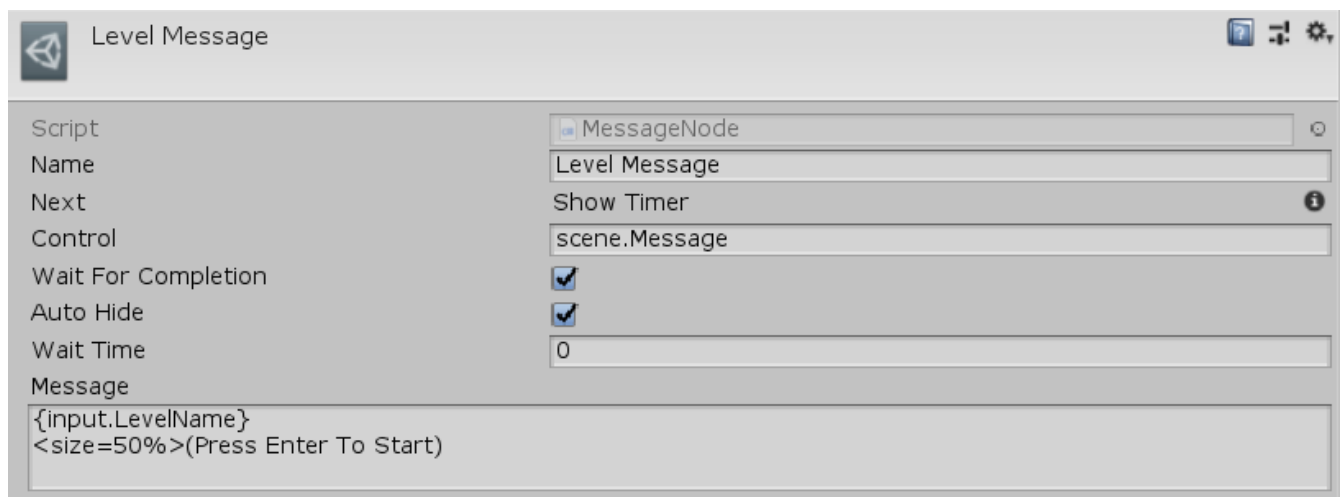
Usually, an interface control is derived from to implement custom behaviour, such as a a Message Control a Selection Control, detailed in the next sections.

Message Controls

Message Controls are an [Interface Control](#) used to show a [message](#), on a [TextMeshPro](#) component. Use a [Message Node](#) to activate a message control and a [Message Input](#) behaviour to advance the text with input. When shown a message controls starts a [Coroutine](#) that will run until the message is finished. Utilizing this coroutine, custom messages can be implemented to show certain pages that advance with input, scrolling, letter by letter typewriting, etc. A [Message Node](#) may also optionally wait for the message to complete before moving on to subsequent nodes. A message control's *IsRunning* flag will be set from when it is shown to when it is finished.

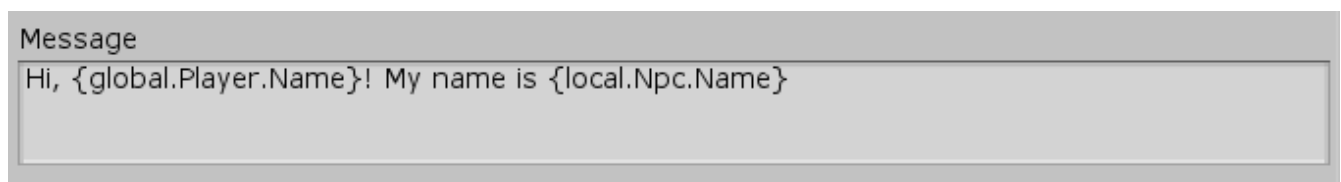
Message Node

Create a Message Node in a [graph](#) using the **Create › Interface › Message** menu of the Instruction Graph Window. A Message Node will activate a message control, which enables the referenced [TextMeshPro](#) component and set its text to a resolved [Message](#). The node can automacally deactivate message control when it is finished is complete if *AutoHide* is true. If *AutoHide* is false then a [Hide Control Node](#) must be used to deactivate it. Message nodes can also delay the amount of time to wait before they deactivate the message control with the *WaitTime* field.



Message

A [message](#) is the string that will be displayed and can be dynamically formatted using [variables](#). To add formatted variables insert a variable reference between braces ({}). For example in following message:



The variables `global.Player.Name` and `local.Npc.Name` will be resolved at runtime with the [instruction store](#) on the message node, and the message control will display the text accordingly (provided those variables exist). Variables that are not strings will have `.ToString()` called on them so numbers can be displayed as well.

Input

In order to maintain modularity of behaviours, responding to input in message controls should be done with separate components. Use a [Message Input](#) component to wait for the specified button, *AcceptButton* to be pressed to finish a message.

Custom Message Nodes

The Message Control provides basic functionality for displaying text, however, most games display messages with much more custom behaviour. Message controls are designed to be extensible desired behaviour can be achieved with little difficulty. The following example shows how to derive from [Message Control](#) to create a paged, typewriter-like effect that responds to two button presses - to fast forward, and to advance:

```
public class TypewriterControl : MessageControl
{
    public float CharactersPerSecond = 25.0f;

    [NonSerialized]
    public bool FastForward = false;

    protected override IEnumerator Run()
    {
        DisplayText.maxVisibleCharacters = 0;

        yield return null; // consume the press that opened the message

        for (var page = 0; page < DisplayText.textInfo.pageCount; page++)
        {
            yield return ShowPage(page);

            IsAdvancing = false;

            while (!IsAdvancing)
                yield return null;
        }

        yield return null;
    }

    private IEnumerator ShowPage(int index)
    {
        var page = DisplayText.textInfo.pageInfo[index];
        var characterCount = page.lastCharacterIndex - page.firstCharacterIndex +
1;
        var characterDelay = CharactersPerSecond <= 0.0f ? 0.0f : 1.0f /
CharactersPerSecond;
        var delay = characterDelay;
```

```

        DisplayText.maxVisibleCharacters = page.firstCharacterIndex;
        DisplayText.pageToDisplay = index + 1;

        while (DisplayText.maxVisibleCharacters < characterCount)
        {
            if (FastForward)
            {
                // fast forward to the end of the text (one character per frame)
                characterDelay = 0.0f;
                delay = 0.0f;
            }
            else if (IsAdvancing)
            {
                // skip to the end of the page
                DisplayText.maxVisibleCharacters = characterCount;
            }
            else if (delay <= 0.0f)
            {
                delay += characterDelay;
                DisplayText.maxVisibleCharacters++;
            }

            delay -= Time.deltaTime;

            yield return null;
        }
    }
}

public class TypewriterInput : MonoBehaviour
{
    public string FastForwardButton = "Cancel";
    public string NextButton = "Submit";

    private TypewriterControl _typewriter;

    void Awake()
    {
        _typewriter = GetComponent<TypewriterControl>();
    }

    void Update()
    {
        _typewriter.FastForward = InputHelper.GetButtonDown(FastForwardButton);

        if (InputHelper.GetWasButtonPressed(NextButton))
            _typewriter.Advance();
    }
}

```


Simply override the `Run()` method and implement your custom behaviour. Notice how `Run()` returns an [IEnumerator](#). As mentioned before, this is because message controls are implemented as a [Coroutine](#). When the method ends *IsRunning* will be set to false.

Menus

Menus are [MonoBehaviours](#) that provide an interface for adding, removing, selecting, and focusing of child [GameObjects](#). On their own, Menus are simply a container for child [Menu Items](#) which are created externally. They can be created manually in the editor or populated automatically using a [List Binding](#) or in conjunction with a [Selection Control](#) and a [Selection Node](#).

Selection Controls

Selection Controls are [Interface Controls](#) used to dynamically populate a menu with items to be selected from when prompted. Selection Controls are activated with the `Show()` method which takes a list of [MenuItemTemplates](#). Each template contains the info needed to create items in the menu: a name, the [variables](#) to assign to the item, the prefab to instantiate, etc. Selection controls can optionally require a selection to be made so they cannot be cancelled, and they can optionally maintain their focused index in case the selection is being returned to from a subsequent menu.

Menu Items

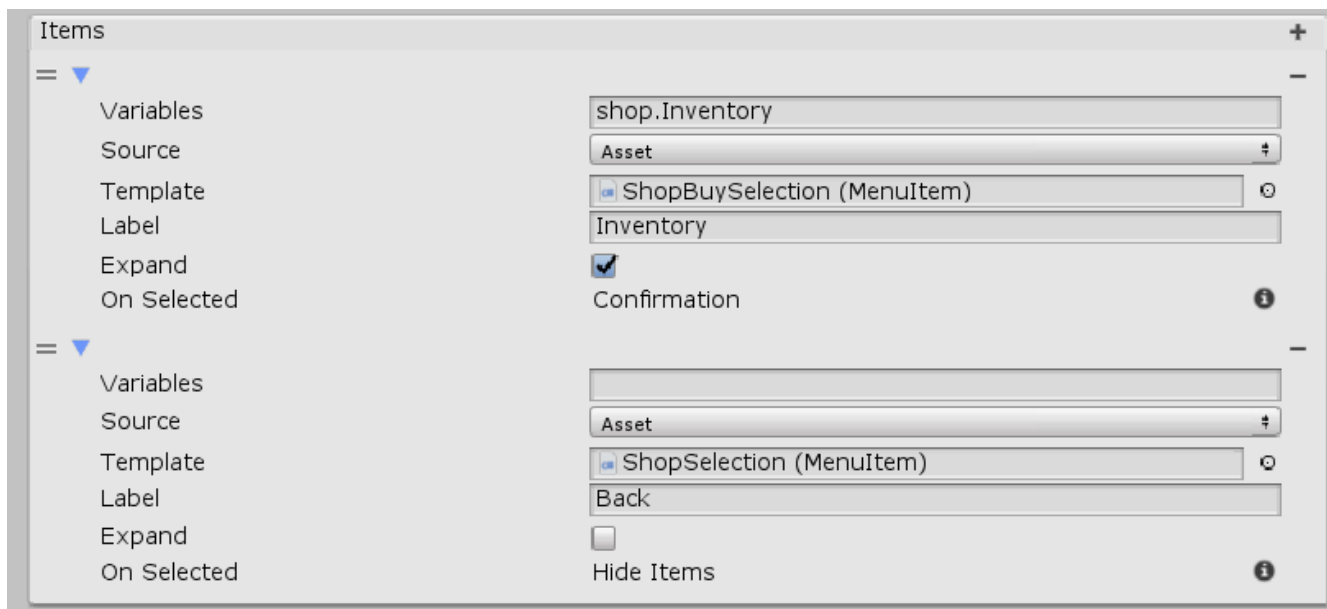
A Menu Item component must be attached to each object in a menu, whether it be a prefab to be instantiated or an existing item in the scene. They are a [binding root](#) which exposes an *ItemName* field which gives access to the data of the Menu Item itself. Variables that can be accessed by child [bindings](#) through *ItemName* are: *Index*, *Column*, *Row*, *Label*, and *Focused*. The *ValueName* field on a menu item exposes the variables specified by the menu item template used to create the menu item.

Selection Nodes

Create a Selection Node in a [graph](#) using the **Create › Interface › Selection** menu of the Instruction Graph Window. A Selection Node will activate a selection control and give it a list of [MenuItemTemplates](#) to create. A Selection Node will deactivate the control once a selection has been made if *AutoHide* is true. If *AutoHide* is false then a [Hide Control Node](#) must be used to deactivate it. When a selection is made the selected item and index will be assigned to [variables](#) specified by *SelectedItem* and *SelectedIndex*. The graph will then branch to the corresponding node of the [selected item](#). Selection Nodes tell the selection control via the *IsSelectionRequired* flag whether it can be cancelled or not. If this is the first iteration of the selection node then it will also tell the selection control to reset its focus.

Menu Item Templates

Each item will be created from a Menu Item Template which has the following properties:



Name	Description
Variables	The variable that should be used as the Binding Root Value for the menu item.
Source	Specifies whether the menu item should be looked up in the scene using <i>Name</i> (Scene) or created from a prefab using <i>Template</i> (Asset)
Name	When <i>Source</i> is Name, the name of the GameObject containing the menu item in the loaded scenes.
Template	When <i>Source</i> is Asset, the prefab to create the menu item from.
Label	When <i>Source</i> is Asset, the label to assign to the menu item
Expand	When <i>Source</i> is Asset, this is true, and <i>Variables</i> references a List , a menu item will be created from <i>Template</i> for each item in the list

Other Helpful Behaviours

Menu Input

In order to maintain modularity of behaviours, responding to input in menus should be done with separate components. Use a [Menu Input](#) component to handle the behaviour of input, focusing, selecting, and scrolling, through menu items.

Focus Binding Root

Use a Focus Binding Root to bind data to the currently focused menu item in a menu. This can be useful for displaying information like a description of an item in separate UI objects that are not actually part of the selection. See the "*Shop*" scene for an example.