Tutorial – Unity UI Layouts

This tutorial series will take you through creating a moderately complex GUI in Unity. We'll start with laying out some basic components, and build up to creating lists of components through scripting.

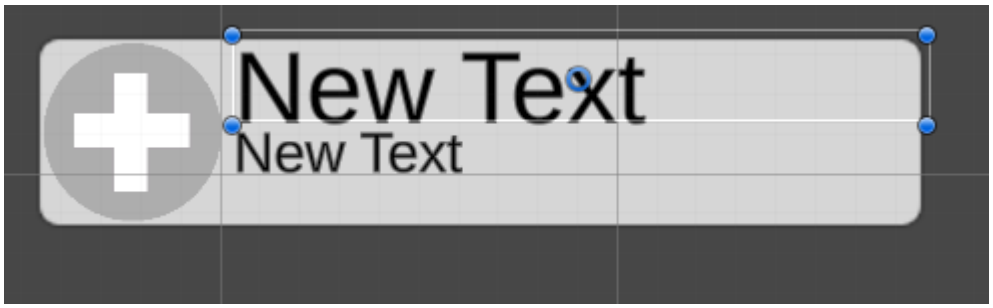
Make sure you've completed *Tutorial – Unity User Interface* before starting this one.

Creating a UI Prefab

Lets start by making a prefab to go into our UI action list.

Prefabs (short for pre-fabricated) are Unity components that you can create to live in the Assets folder, and re-use multiple times in your scene. If you modify the prefab, then all instances of the prefab update. Individual prefabs can have different data from the base object, known as overrides.

Grab the UnityUI.unitypackage and import it into your project. This has some basic icons. We'll now make a simple object that displays information to the player about an action they might perform. The end result will look a bit like this:



How would you describe this to someone in words?

The icon sits at the left, with a small border between it and the edge of the container. The larger header text and smaller description text fill the remainder of the container horizontally, each taking up about half of the available vertical space.

First, create a new Image in the centre of the screen, and give it the "Background" sprite. Make it 160 pixels wide and 36 tall. Rename it as **ActionUI**

Add another Image as a child, and call it **Icon**

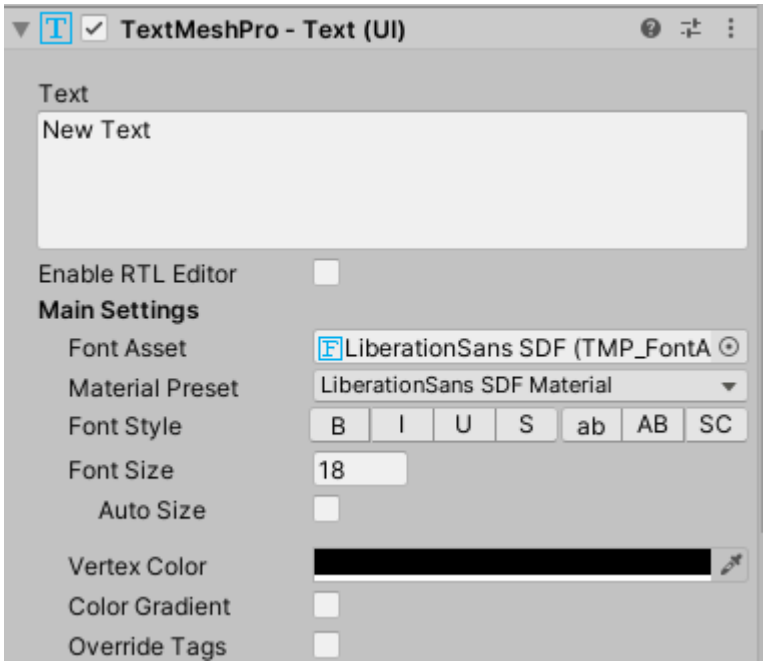
Use the description above to inform your decisions on how to set up the icon. It should be anchored to the mid left of the panel, with a width and height that fits in inside vertically. I made my icon 32 x 32, and the container 160 x 36, so we have a 2 pixel border above and below the icon. The icon has a pivot of 0 and position of (2,0) to give a 2 pixel border on the left, and centre it vertically.

The text should use up the remaining horizontal space.

Create a **UI->Text-TextMeshPro** object for the header text, as a child of the parent container. Call it **Name**.

TextMeshPro is an Asset Store item that unity have integrated into the main editor. It provides higher quality text that's rendered faster than normal unity Text, so there's no reason not to use it. You'll get a pop-up when you first use it in a project, click the Import button and keep on going.

The text will initially be quite large and white. Make it Size 20 and Black in the TextMeshPro Text (UI) component in the inspector



To anchor it and size it, we want it to use whatever horizontal space is available. In its RectTransform, click on the anchor box and select the bottom right icon with the arrows stretching to fill the whole parent. This means we now express our size and position by specifying borders to the left, right, top and bottom in pixels. We want zero border on top and right, maybe a 20 pixel border below (so we're using half of the 36 pixels available) and 36 pixels on the left, so as not to overlap the icon.

Pivot doesn't matter with these settings, so leave it at the default (0.5, 0.5)

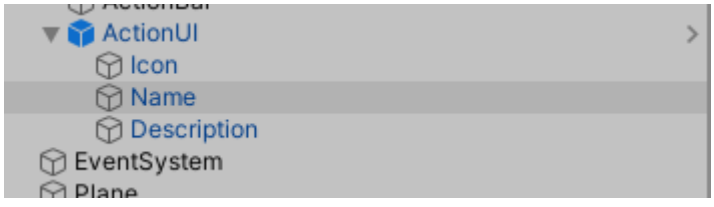
Create another TextMeshPro text item called **Description** and set it up in a similar manner to the Name text, but occupying the bottom half of the available space. Give it a font size of 10.

When making a component like this, think of what the longest text you might display will be, and type something in there. It doesn't have to make sense, but it's a good testing step in UI development.

You may decide at this point to reduce your text size, or increase the width of the container or both. Note that if you increase the width of the container, the Text items resize automatically.

I've made the parent object 240 pixels wide, and the text sizes 14 and 8 here, and I can be reasonably confident that this will work for any Action I want to display.

Now make a folder called Prefabs in your Assets pane, and drag the ActionUI into there. It will turn blue in the Heriarchy, indicating that it's now a prefab.

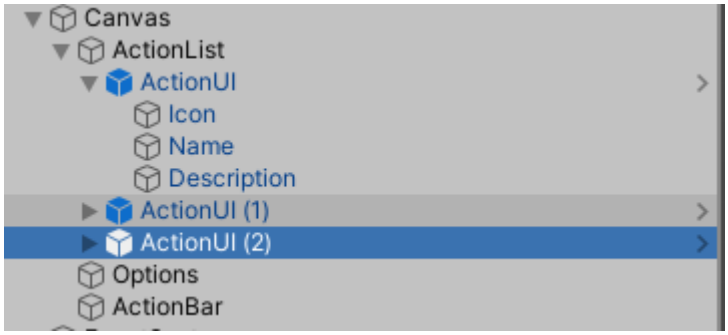


The arrow on the right of the word ActionUI opens up the prefab editor, where you can edit your prefab object in it's own separate scene.

Using a LayoutGroup

Let's make the ActionUI a child of the ActionList image.

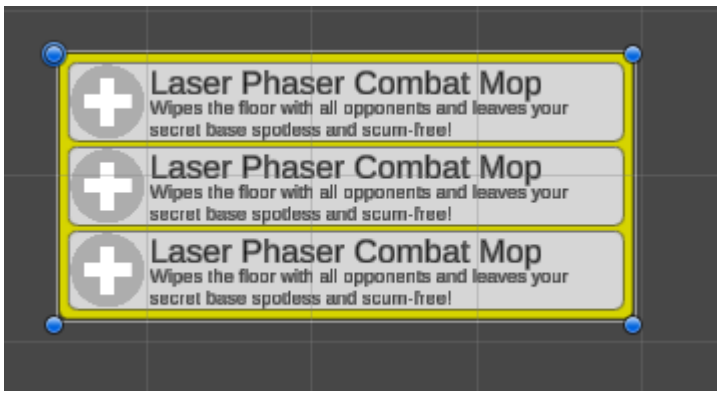
Find the ActionUI prefab in your assets folder and drag a couple of other ones onto the ActionList object in the Heriarchy, and you should get three of them.



Add a **VerticalLayoutGroup** component to the ActionListUI, and the three child objects will automatically line up on top of each other.

Add a **ContentSizeFitter** and set the Horizontal and Vertical Fit to Preferred, and the background image will resize to fit around the child components.

Give the background a clearer color to highlight it. In the Vertical Layout Group, change the padding and spacing and see how it reacts.



Programming the UI

It's time to do some programming!

Create a script called Action, with the following public members: a string action name, a string description, a Sprite icon and a Color (to tint the icon).

Create another script called ActionUI, which will take one of these Actions and show it in our prefab.

This follows a pattern to adopt throughout any UI programming exercise. The Action is a backend object holding the important data, and doesn't need to know anything about the front-end UI. The ActionUI is that frontend UI, and it can take a Action and show it to the player.

```
public class Action : MonoBehaviour
{
    public string actionName;
    public string description;
    public Sprite icon;
    public Color color = Color.white;;
}
```

The ActionUI should have a public Action member, and a series of references to child components so we can tell it which piece of information goes where.

We'll want an Image member and some TextMeshProUGUI members. Both of these classes exist in namespaces, so we'll have to add using declarations for them.

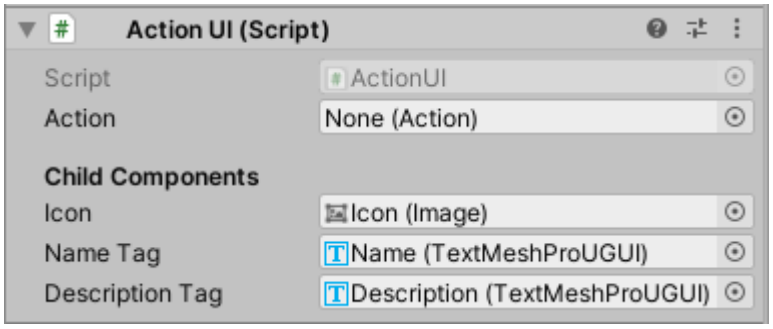
```
using UnityEngine.UI;
using TMPro;

public class ActionUI : MonoBehaviour
{
    public Action action;

    [Header("Child Components")]
    public Image icon;
    public TextMeshProUGUI nameTag;
    public TextMeshProUGUI desccriptionTag;
}
```

We now want to have a function to set our action member, and update the child components. We should check each child is not null before using them.

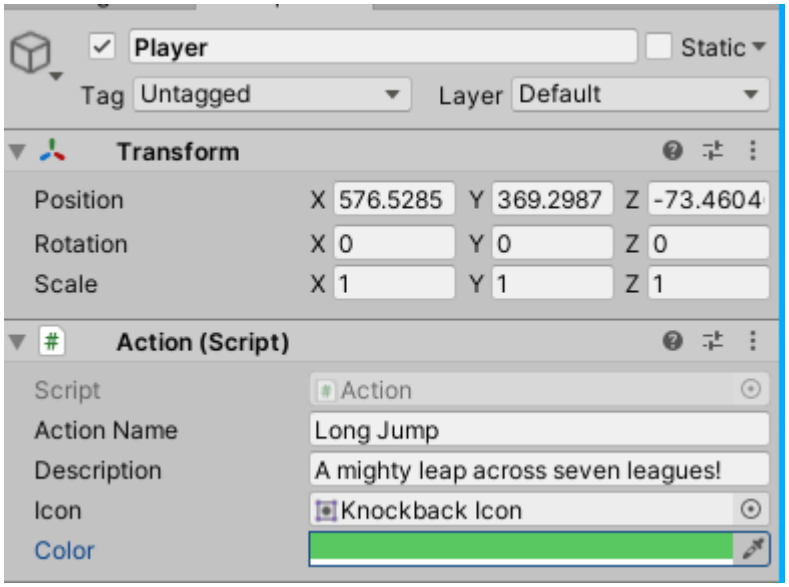
```
public void SetAction(Action a)
    {
        action = a;
        if (nameTag)
            nameTag.text = action.actionName;
        if (descriptionTag)
            descriptionTag.text = action.description;
        if (icon)
        {
            icon.sprite = action.icon;
            icon.color = action.color;
        }
    }
```

Let's text this code.

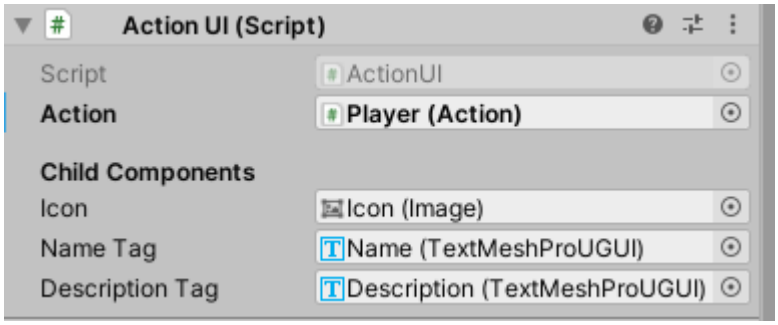
Open the ActionUI prefab and add an ActionUI script to the top-level component. Drag in the child elements into the correct slots.



Back in the main scene, create an empty GameObject called Player, an add an Action component to it. Choose an icon and set some text in the inspector.



Drag this object into the first ActionUI in the list. Notice that the word Action is now in bold, as it overrides the default value of null for the prefab.



In your ActionUI class, write a Start function that calls SetAction on the action.

```
    private void Start()
    {
        SetAction(action);
    }
```

When you press play, you should now see Long Jump with a green icon at the top of your list!

You'll also get a couple of errors, because we didn't check if action was a null reference before using it in ActionUI.SetAction. Put a check in the top here to make sure that we deal with this case.

Filling the List procedurally

We often have a list or array of back end items that we want to display on in the UI. At design time we don't know how many there will be. We'll now set up a couple of classes to do this, so make two new scripts called ActionList and ActionListUI

ActionList is responsible for managing an array or list of Actions. It's a back-end class, that doesn't need to know about the UI, and could represent a player's inventory, their powers, the items in a shop or any other list of things.

Here we want to be able to get an array of Actions from it, and we'll do this by finding all sibling components that are an Action. (note the plural GetComponent**s** returns an array of objects.

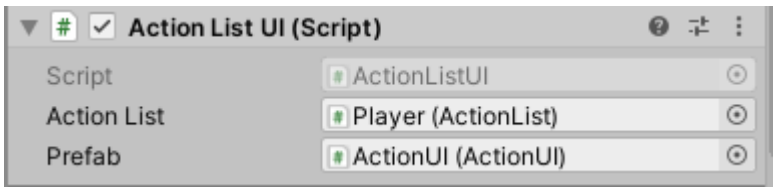
```
public class ActionList : MonoBehaviour
{
    public Action[] actions;

    // Start is called before the first frame update
    void Start()
    {
        actions = GetComponents<Action>();
    }
}
```

Put this class on the Player GameObject and add a couple of extra Actions with different data on them.

The ActionListUI should have a reference to an ActionList, and a reference to a prefab UI object with an ActionUI script. For each Action in the ActionList, its clones the prefab as a child of itself, and sets the Action into the clone's Action UI.

```
public class ActionListUI : MonoBehaviour
{
    public ActionList actionList;
    public ActionUI prefab;

    // Start is called before the first frame update
    void Start()
    {
        foreach (Action a in actionList.actions)
        {
            // make this a child of ours on creation.
            // Don't worry about specifying a position as the LayotGroup handles
that
            ActionUI ui = Instantiate(prefab, transform);
            ui.SetAction(a);
        }
    }
}
```

To test this, add an ActionListUI to your ActionList object with the HorizontalLayoutGroup on it. Delete all its children, and link up the prefab ActionUI from your prefabs folder and the Player in the scene in the ActionListUI component.



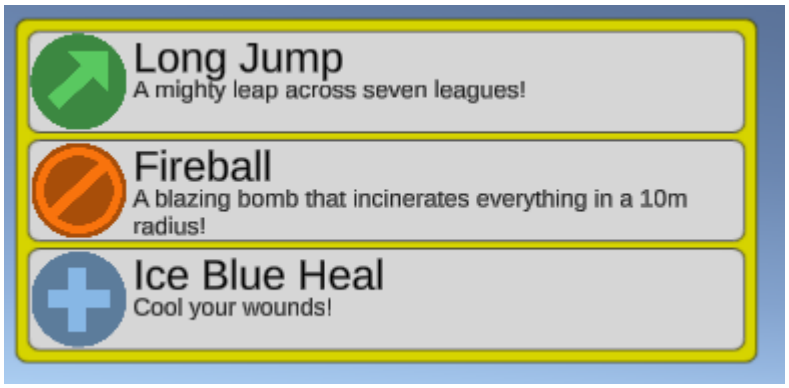
This may not work.

The Start() function of ActionListUI may be getting called before the Start() function of ActionList, so we have an empty list of Actions to clone from when we start the ActionListUI.

We can get around this by making a property in ActionList that gathers the list of Actions when required. This pattern is referred to as “lazy initialisation” where you calculate a value on demand at the last possible minute.

```
public class ActionList : MonoBehaviour
{
    Action[] _actions = null;
    // lazy initialisation public accessor
    public Action[] actions
    {
        get
        {
            if (_actions == null)
                _actions = GetComponents<Action>();
            return _actions;
        }
    }
}
```

You should now see your Actions in the UI when you run.



Summary

Here we've established a few patterns that you should keep in mind when designing any UI.

- A simple back-end object (Action) is represented by a corresponding UI object (ActionUI)
- We have another UI object (ActionListUI) that represents a list of simple back-end objects
- The back-end object code makes no reference to the UI objects
- The list Ui class instantiates copies of a prefab
- It doesn't concern itself with the details of each object, that is all done through a simple function in the individual UI object class (Action.SetObject)
- The visual layout of the list of items is handled by Unity's LayoutGroup class