Tutorial – Unity Event Systems

This tutorial series will take you through creating a moderately complex GUI in Unity.

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

Making an interactive UI

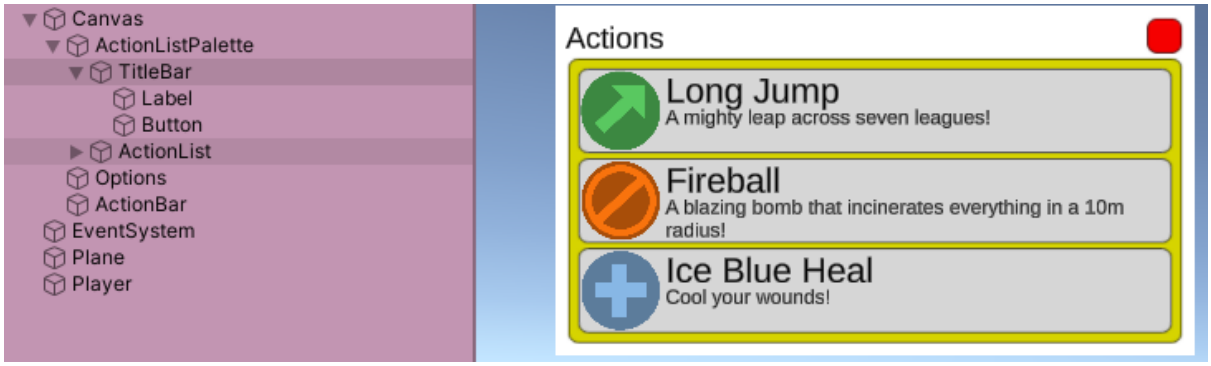
We've previously created a programmatic UI that creates a front-end object for every back-end object in a list. These were just displaying information. It's common to want to make these elements interactive.

Using UnityEvents

A UnityEvent behaves like a delegate. You can attach functions to it both in the inspector and via code.

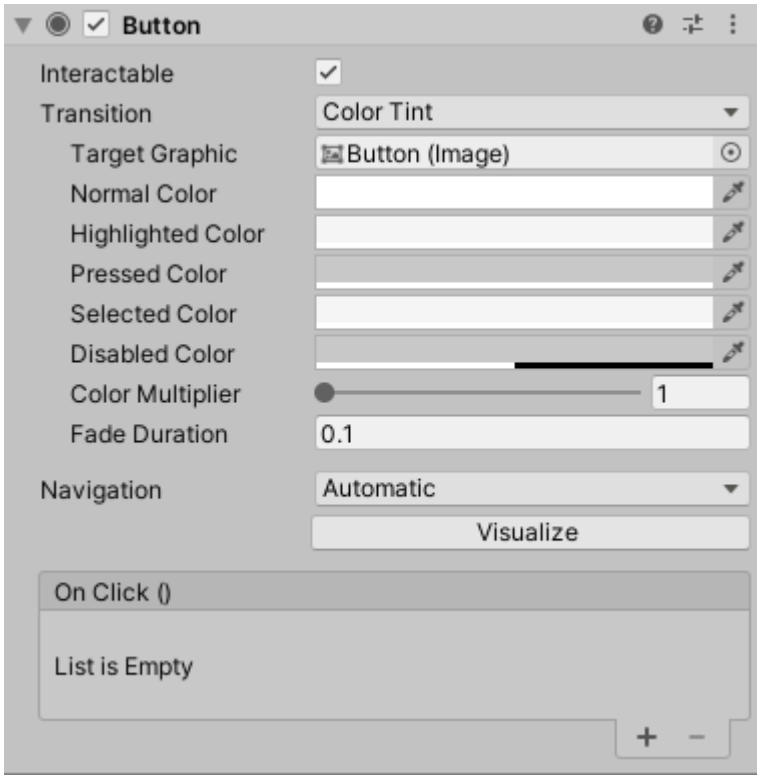
Lets start by setting up a simple button to collapse or expand our item list. We'll re-organise the ActionListUI to have a parent Image with a VerticalLayoutGroup and ContentSizeFitter, which contains two children.

- A TitleBar that contains a TextMeshProUGUI label and a small button in the top right
- The existing ActionList object, which has its own VerticalLayoutGroup for organising its children.

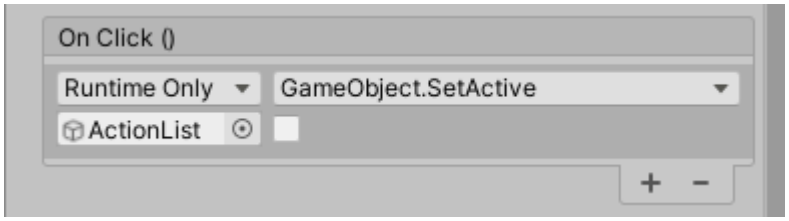


The hierarchy should look like this, and give this result when run. You'll have to work out how to anchor and position the title and button in the TitleBar. Note that when you create a button it automatically adds a child text. We can delete it for a button like this which is purely an image.

The Button component has an OnClick Unity event which we can attach functions of any object to, which looks like a small tab in the inspector.



Use the + button on the OnClick() tab to add a new event. For now, let's make the ActionList collapse when we click the button by dragging in the ActionList. We get a droplist of all components, each of which shows up all fucntions that we can call on them. Choose GameObject, SetActive with a false checkbox to deactivate it.



Press Play and you can now collapse the ActionList by clicking on the red button.

We'd really like to toggle it on and off instead. To do this we have to write a short script, call it Toggle Active, which will have a single function in it.

```
public class ToggleActive : MonoBehaviour
{
    public void Toggle()
    {
        gameObject.SetActive(!gameObject.activeSelf);
    }
}
```

Attach this to the ActionList object, and then call the Toggle function from the Button's OnClick event. You can now open and close the ActionList with the red button.

Programming with UnityEvents

Our next task will be to turn the icons on the ActionList into buttons.

First, let's create a script on the Player object with a stub function for the player to perform a particular Action.

Create a Script called Player, and give it a single function that for now prints out a debug log statement.

```
    public void DoAction(Action action)
    {
        Debug.Log("Doing " + action.actionName);
    }
```

Now open the prefab for the ActionUI and add a Button component to it. If you run, you'll now see that the icons behave like buttons, dimming when you click them. They don't do anything yet.

We can't link the Button's onClick event to an object in the scene because it lives in the assets folder, and we want to make each button tell the player to do its own particular action.

We'll need to add something to the onClick event programmatically.

We can do this by adding a lambda function to the button's onClick that calls the Player.DoAction with the correct action like so:

```
public class ActionUI : MonoBehaviour
{
    …

    Player player;

    public void Init(Player p)
    {
        // store the player ref for use in our lambda function below
        player = p;
        // find the button wherever we've placed it in the prefab
        // for more complicated types of prefabs with multiple buttons, we'd make
this a public member
        // and hook it up in the Unity editor
        Button button = GetComponentInChildren<Button>();
        if (button)
            button.onClick.AddListener(() => { player.DoAction(action); });
    }
```

The lambda function here creates a local unnamed function that gets called when the button is pressed, with a copy of the stack at the time it was set up.

Lambda functions can look a bit strange at first, but are extremely useful for this kind of UI programming, and should be incorporated into your programming repertoire. Practice them until they become second nature.

Call this when we create each button in ActionListUI.

```
    void Start()
    {
        player = actionList.GetComponent<Player>();
        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);
            ui.Init(player);
        }
    }
```

Run the game, and you should see some debug log messages print out when you click on each action button.

Creating an ActionBar

We've made a collapsible action list here, which can be turned on or off and provides details of the available actions. It takes up a lot of Screen real estate though, and might not be so great in game.

We can use the code we've written to make an ActionBar in the bottom left of the screen that looks like this:

1. Create a new prefab called ActionButtonUI, with an Image and Button component, and an ActionUI component that refers to its own Image, but has null entries for the nameTag and description.
2. Add the following components to the ActionBar Image we placed on the main Canvas: HorizontalLayoutGroup, ContentSizeFitter (with Preferred sizes in width and height) and ActionListUI
3. Link up the Player in the ActionListUI to the Player in the scene
4. Link up the ActionButtonUI as the Prefab

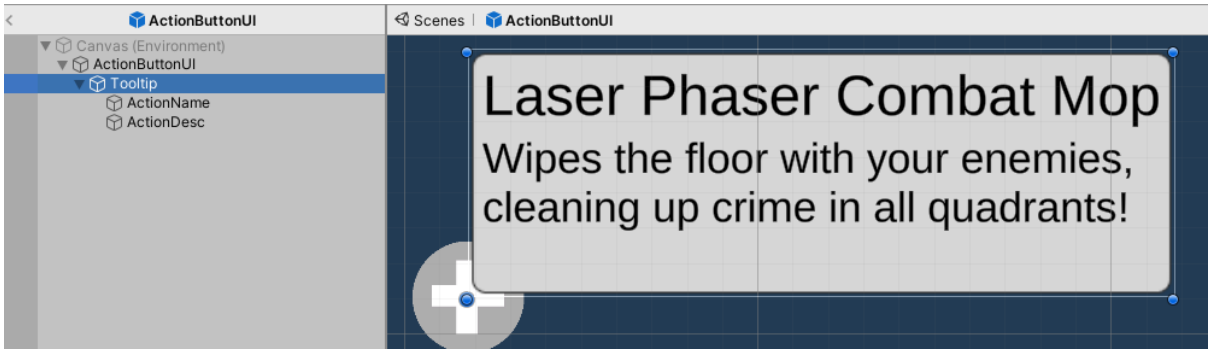
You should now get a similar looking action bar with clickable buttons when you run. Note that we haven't written any new code for this. The null pointer checks in ActionUI.SetAction mean that the text members are purely optional.

Adding Tooltips

The action bar is compact, but in the heat of the game, we might forget which button does what. Tooltips are a great way to remind the player.

There are a number of different approaches we can take to tooltips. Here we'll add a tooltip panel to the ActionButtonUI prefab with text members that hold the name and description, and turn it on and off when the mouse enters or leaves the button.

First, add a tooltip Image with a couple of TextMeshProUGUI children to the button, and position them where we want the tooltip to appear. We want the bottom left corner of the tooltip to sit on the centre of the button roughly, so figure out how to express that in terms of anchors and pivots in the RectTransform



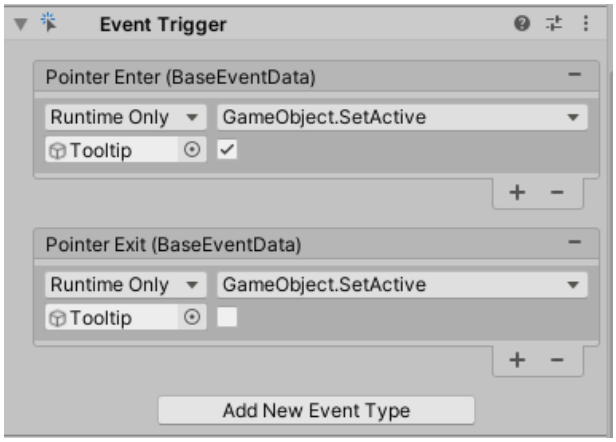
Hook the ActionName and ActionDesc objects up to the parent ActionUI.

In the parent button we now ad an EventTrigger, which is a Unity component that allows us to set up less common events for any UI object.

Click the **Add New Event Type** button and choose PointerEnter and PointerExit from the dropdown. Each will add a new UnityEvent with the corresponding name.

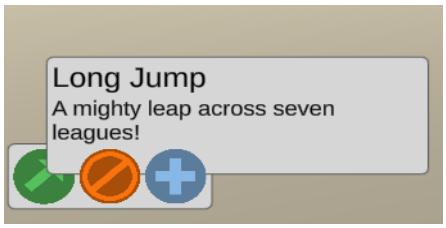
We want to make the Tooltip GameObject set itself to active in the Pointer Enter event, and set itself to inactive in the Pointer Exit event.

Your EventTrigger should look like this:



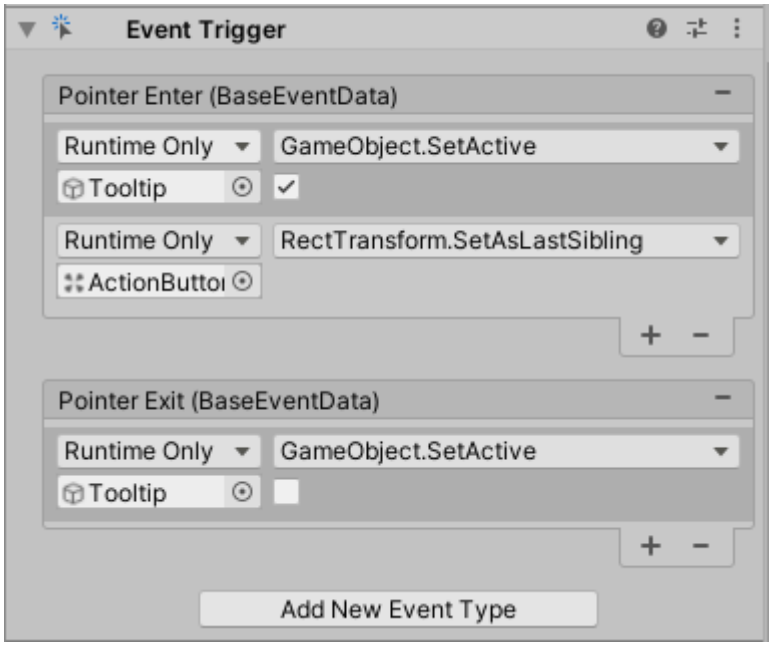
Turn the Tooltip off at the GameObject level in the prefab, and now run. You should find that the tooltips appear when you mouseover each button.

There is a problem with rendering order though, since the later buttons are rendered after the first one's tooltip.



We can make the button we've highlighted the last one to draw using Transform.SetAsLastSibling(). That will have the unfortunate side-effect of re-ordering the buttons via the Horizontal Layout Group. We could work around that by turning the LayoutGroup off once the control has initialised.

Add the call to SetAsLastSibling to your PointerEnter UnityEvent in the ActionButton prefab:



And add some code to turn off the LayoutGroup and ContentSizeFitter once we've instantiated all the children.

We don't want to do this straight away, because these components need to be turned on for a frame in order to reposition the children. We can make this happen by turning the Start function into a coroutine, and turning off the LayoutGroup and ContentSizeFitter after a single frame has elapsed.

```
IEnumerator Start()
    {
        Player player = actionList.GetComponent<Player>();

        foreach (Action a in actionList.actions)
        {
            …
        }

        yield return new WaitForEndOfFrame();

        GetComponent<ContentSizeFitter>().enabled = false;
        GetComponent<LayoutGroup>().enabled = false;
    }
```

We can now mouseover the buttons and the tooltips pop up over all the buttons without them changing order.

Note that this isn't the only way to do tooltips.

As an alternative, you could have a single component that polls the screen every frame to see what object is under the mouse using EventSystem.RaycastAll(). It checks the object it hits for a IToolTip interface that you write, which returns the tooltip string to display in a single image. Research this or other methods if you want to experiment.

Creating our own Events

We've now got a component for displaying a single Action, or a static list of Actions which doesn't change throughout the game. In a real game, the list of backend actions may change. You level up and get a new Action or Power. You buy an item from a store and it disappears from the store's inventory and appears in your own.

We'll now look at how to deal with a list of back-end objects that changes dynamically over time.

We want a solution that obeys the following good practices:

- The backend object doesn't know anything about the UI components, so ActionList can't call any functions of ActionListUI. This may seem restrictive, but we may want to be able to use our backend classes in a UI-free tool project.
- The UI is not polling for updates, but responds to events when the user adds or removes an item to the ActionList.
- The ActionListUI may need to add new components and recycle some old ones, and keep some existing ones when it updates itself.

To satisfy these we should create an event on the ActionList that the ActionListUI can subscribe to. When the contents of the ActionList change, it fires off this event and the UI will respond, without the ActionList needing to know anything about the ActionListUI class.

When the ActionListUI updates itself it should:

1. Go through the list of Actions. If we don't have enough ActionUI's created, create some new ones.
2. Go through the ActionUIs we have and turn any spare ones off.
3. Make sure everything's in the correct order.

It should also reactivate the LayoutGroup and ContentSizeFitter, and then deactivate them a frame later.

We can keep track of the prefabs ActionUIs we've cloned in a list. Like all collection classes in C#, the List itself is a reference, and initialises to null, so allocate it on declaration of the member variable.

```
    List<ActionUI> uis = new List<ActionUI>();
```

We'll then want to step through the List and assign Actions to each element, and turn off any spare ones we may have created if the list has shrunk

```
// step through the dictionary, and remove any uis associated with actions no
longer in our list
        for (int i = 0; i < actionList.actions.Length; i++)
        {
            // if we need to add another UI to our list, create it here
            if (i >= uis.Count)
            {
                // make this a child of ours on creation.
                // Don't worry about specifying a position as the LayoutGroup
handles that
                uis.Add(Instantiate(prefab, transform));

                // pass the player ref through and hook up any buttons
                uis[i].Init(player);
            }
            uis[i].gameObject.SetActive(true);
            uis[i].SetAction(actionList.actions[i]);
            // make sure they all appear in order again
            uis[i].transform.SetAsLastSibling();
        }

        // disable any remaining UIs if the list has shrunk on us
        for (int i = actionList.actions.Length; i < uis.Count; i++)
            uis[i].gameObject.SetActive(false);
```

We now want to move this out of the Start function into its own function, so that we can call it repeatedly whenever the back-end list changes. Since we're repeatedly referencing the LayoutGroup and ContentSizeFitter we'll make them member variables, and initialize them in Start().

```
    LayoutGroup layoutGroup;
    ContentSizeFitter contentSizeFitter;

    // Start is called before the first frame update
    void Start()
    {
        layoutGroup = GetComponent<LayoutGroup>();
        contentSizeFitter = GetComponent<ContentSizeFitter>();
        StartCoroutine(UpdateUI());
    }

    IEnumerator UpdateUI()
    {
        contentSizeFitter.enabled = true;
        layoutGroup.enabled = true;
        yield return new WaitForEndOfFrame();

        player = actionList.GetComponent<Player>();
        …
        yield return new WaitForEndOfFrame();

        contentSizeFitter.enabled = false;
        layoutGroup.enabled = false;
    }
```

Now we're ready to create our own custom event.

Add a public UnityEvent member to the ActionList (the backend object) that we can fire off when anything changes in the list.

UnityEvent is part of the UnityEngine.Events namespace, so you'll have to add a using declaration to that to ActionList.cs

Write a little test function that deletes the first Action in the array, and hook it up to a ContextMenu in the editor using the ContextMenu attribute. Not that we call Invoke() on the UnityEvent to call any functions that have been attached to it with AddListener.

```
using UnityEngine.Events;

public class ActionList : MonoBehaviour
{
    public UnityEvent onChanged;

    …

    [ContextMenu("Delete First")]
    void DeleteFirst()
    {
        List<Action> actions = new List<Action>(_actions);
        actions.RemoveAt(0);
        _actions = actions.ToArray();
        onChanged.Invoke();
    }
}
```

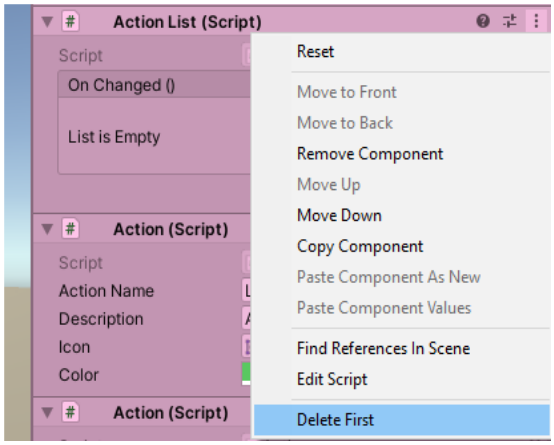
In ActionListUI, we'll subscribe to this event in Start, and get it to call our UpdateActions() function.

```
    void Start()
    {
        layoutGroup = GetComponent<LayoutGroup>();
        contentSizeFitter = GetComponent<ContentSizeFitter>();
        StartCoroutine(UpdateUI());

        actionList.onChanged.AddListener(() => { StartCoroutine(UpdateUI()); });
    }
```

Run the game. You should see the three items in the list. Select the Player object. In the ContextMenu in the top right, indicated by three dots, you'll see the test function we put in with the ContextMenu tag.

Activate it, and the first Action will disappear from both lists smoothly.



Summary

- We can attach public functions to the onClick event of a Button in the Unity editor
- We can also attach functions programmatically, with UnityEvent.AddListener()
- Lambda functions can be very useful when dealing with the onClick events of multiple cloned Buttons
- We can create our own UnityEvents in backend objects and use them to communicate with the UI components without them being aware of the front end class.

Further Development

Add another test function to ActionList to duplicate a random element from the list and stick it on the end, so you can test that your UI can manage a growing list too.

Use this pattern of (Object, ObjectUI), (ObjectList, ObjectListUI) to create something more functional, like a shop, a set of actions for a character to perform, or an inventory.