Tutorial – Options Menu

This tutorial covers a potential project brief for your assessment.

Make sure you've completed the following series as it makes use of the concepts introduced there.

- *Tutorial – Unity User Interface*
- *Tutorial – Unity UI Layouts*
- *Tutorial – Unity Event Systems*

Designing an Options Menu

An options menu will typically present a number of values to the player. Each of these allows you to modify a primitive variable in some back-end settings object. These could include:

- Volume settings for music, sound fx and a master volume for both (floats)
- Camera Invert options (possibly horizontal and vertical) (bools)
- Graphical resolution (Vector2Int)

The Audio Options for Bioshock below shows three sliders for the different volume settings.

The way you edit these variables can vary. You could modify a single number with a slider or a text input field, or both.

You could edit a bool with a checkbox or a button that changes text when clicked.

You could edit a Vector2 by having 2 InputFields, or a droplist of pre-determined values (which is far more common for screen resolution)

Creating a basic Options Dialog

Start off making as Settings object, which can be either a Monobehaviour or a ScriptableObject. Give it a set of public float member variables such as musicVolume and soundFxVolume.

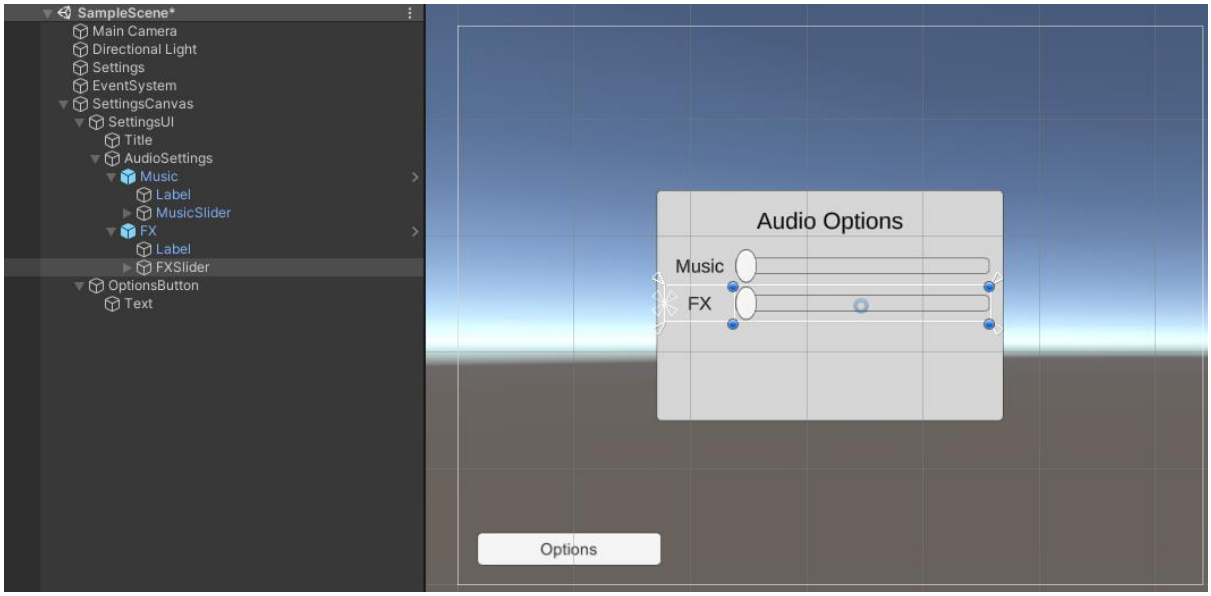
```
public class Settings : MonoBehaviour
{
    public float musicVolume;
    public float soundFxVolume;
}
```

Create a class called SettingsUI to display all these to the player as Sliders

We'll need a member variable for each backend-setting variable. When you show the SettingsUI, you assign these sliders values from the back-end data. When you move the sliders, you'll want to write to the backend variables.

Sliders have an onValueChanged event. You can attach functions in your SettingsUI to the sliders in the Unity editor, but it's better practice to attach these via code.

Here's an example setup for an OptionsScreen demo



The Settings object just holds our Settings script

We have a Canvas, set to Scale with Screen Size with a reference resolution of 800 x 600

The OptionsButton is anchored to the bottom left of the screen.

The AudioSettings object is an Image, anchored to the centre and around 300 x 200 in size.

AudioSettings has a VerticalLayoutGroup, and the Music and FX objects are two instances of the same prefab, which has a Text label and a Slider. The prefab is 280 x32 in size. The slider is set to fill its parent with a 60 pixel border on the left. The label is set to 60 pixels in width and fills its parent horizontally, so the two children fill the parent completely.

These are the terms to think in when you create a UI – prefabs, anchors, pivots and border around objects that fill their parents in one or both direction.

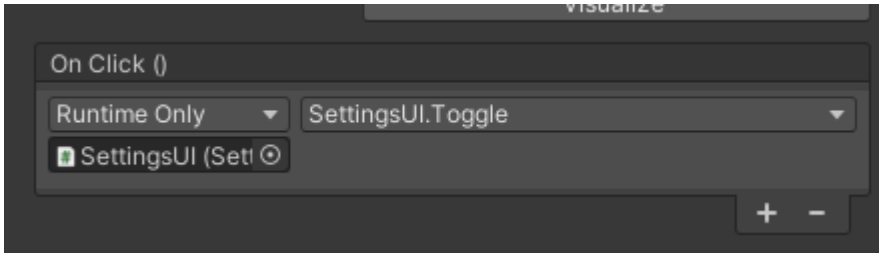
To get this functional, we'd want to complete the following tasks:

1. The OptionsButton makes the SettingsUI appear or disappear when clicked.
2. When the SettingsUI appears, its sliders set to the correct positions based on the Settings object.
3. When we move the sliders, they adjust the appropriate value in the Settings object.

For Task 1 we want a function on the SettingsUI that toggles the active state of its gameObject on and off. This can look something like this:

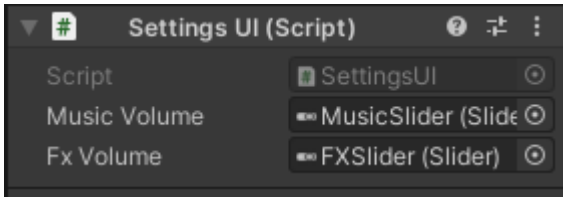
```
    public void Toggle()
    {
        gameObject.SetActive(!gameObject.activeSelf);
    }
```

We can link this function to the OptionButtons onClick event in the editor.



For Task 2, we want to give the SettingsUI a reference to a Settings object, which we can just hook up in the editor for now. This will allow us to read values from it, and apply them to the Sliders. We will of course need to have variables to represent the Sliders, which we set up in the inspector.

```
public class SettingsUI : MonoBehaviour
{
    public Slider musicVolume;
    public Slider fxVolume;
```



Add some code to the Toggle function so that writes the volume settings of the Settings objects into the Sliders .value members.

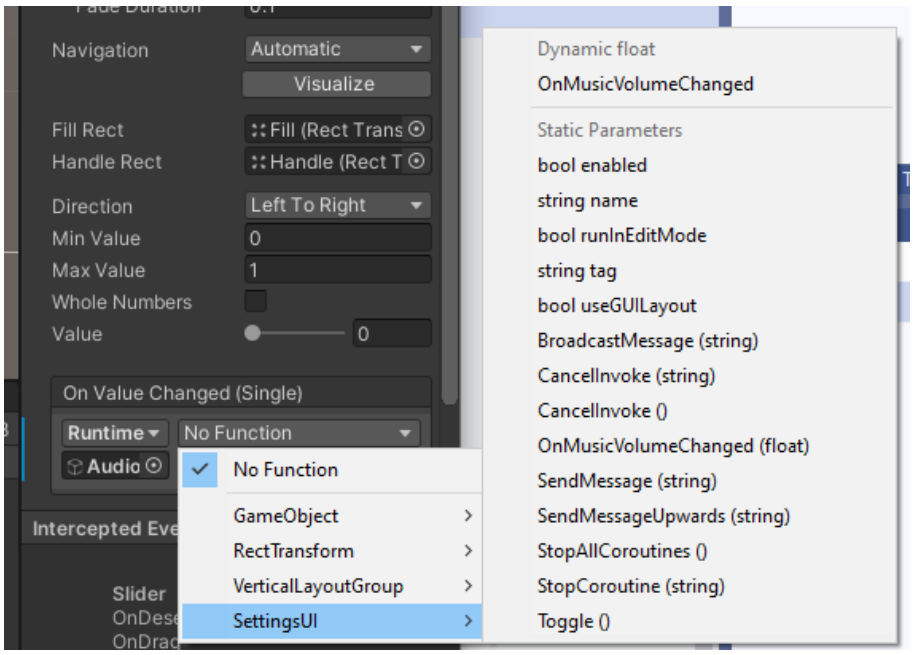
Events with Parameters

We can attach a function to when the slider changes using AddListener. The onValueChanged event is different from a Button's onClick in that it passes a floating point value through as a parameter to any function that is attached.

To demonstrate this, create a function that takes a float parameter in your SettingsUI.

```
    public void OnMusicVolumeChanged(float volume)
    {
        settings.musicVolume = volume;
    }
```

If you try to attach this to the onValueChanged of the MusicSlider in the editor, you'll see the following options:



OnMusicChanged appears with all the other functions under the **Static Parameters** heading. If you select any of these, you have to provide a constant floating point value that will always be passed through.

If you choose OnMusicVolumeChanged from the **Dynamic float** list at the top, then the value of the slider is passed through to the OnMusicVolumeChanged function.

To set this up in code, we can call AddListener and pass in any function that has a single float parameter.

For example:

```
    private void Start()
    {
        musicVolume.onValueChanged.AddListener(OnMusicVolumeChanged);
    }
```

If you want to use a lambda function for events with parameters, you can pass the parameters through like this:

```
    private void Start()
    {
        musicVolume.onValueChanged.AddListener((float value) => {
           settings.musicVolume = value;
        });
    }
```

You should now be able to put together an options screen that reads the values from Settings when it appears, and writes to the values in Settings when the sliders are moved.

Adding Checkboxes

Checkboxes (called Toggles in Unity) also have a onValueChanged which takes functions with a single bool parameter. Try adding a Boolean value to Settings for stereo, and linking it with a Toggle in the SettingsUI.

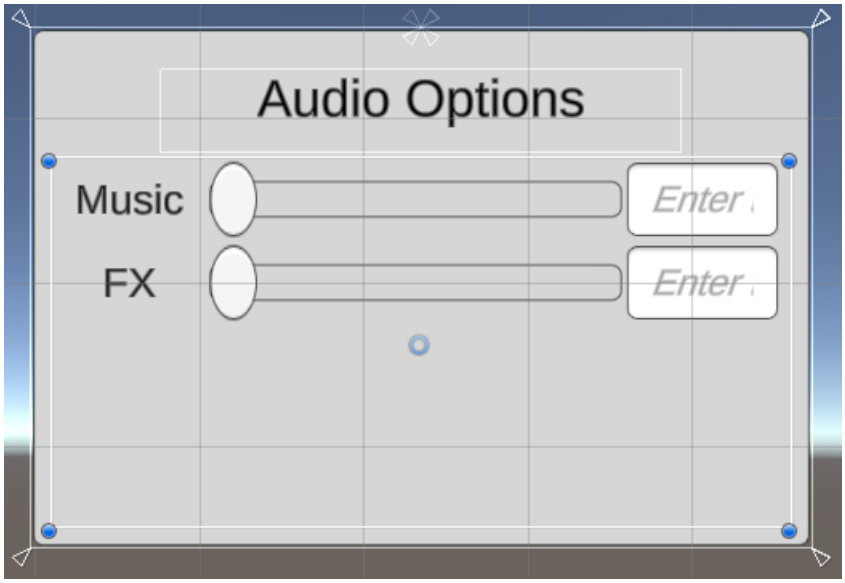
Extending the Sliders (Advanced)

Are sliders the best choice for an options screen? Sometimes you'll want to represent a number with a slider, like a volume meter. Sometimes you'll want a display of the number next to the Slider. This could be read-only, or be an input field where you can enter a number, and the slider adjusts accordingly.
Sometimes, you may want to edit a number via a dropdown list, for example for anti-aliasing settings which can only be 1, 2, 4, or a Field of View setting that can be limited to certain fixed values.

If you add an InputField to the prefab with the Slider and Label, it could look something like this, allowing you to adjust the Slider or enter a value in the InputField to change the volume.

To keep this all consistent we need to perform a number of extra actions.

- When the Slider is updated, update the InputField and the backend value
- When the InputField is update, update the slider and the backend value
- When the dialog starts up, write to both the Slider and the InputField



As your project progresses, you may receive design input to remove the Sliders, or remove the InputFields, or to keep one of them and add a droplist.

It would be nice to be able to do so by modifying prefab data and not have to rewrite the client code that uses your components.

A FloatEditor class

Sliders have an onValueChanged event.

TMP_InputField's have an onValueChanged event that fires off whenever the text changes, and an onEndEdit event that fires off when you're finished editing, either by the input field losing focus, or by pressing Enter.

Rather than hook every piece of client code up to both these events, we could make a FloatEditor class that manages both of these and has its own event for client code to attach to.

We'll need a private float variable to hold the float value, and a public property whose setter updates the UI controls.

```
using UnityEngine;
using UnityEngine.Events;
using UnityEngine.UI;
using TMPro;

public class FloatEditor : MonoBehaviour
{
    [Header("Components")]
    public Slider slider;
    public TMP_InputField input;
    // string used to format the text field when you move the slider
    // see https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-
numeric-format-strings for details
    public string formatString = "0.0";

    // property for our float value that sets any slider or input we may have
attached
    float _floatValue;
    public float floatValue
    {
        get { return _floatValue; }
        set
        {
            // update our internal variable
            _floatValue = value;

            // make sure all our controls are visually consistent
            if (slider)
                slider.value = value;
            if (input)
                input.text = value.ToString(formatString);
        }
    }
```

This way, our client code can just go

```
        editor.floatValue = 0.7f;
```

and all the sub-components (slider and input field here) get updated correctly.

We'll want to create our own event that fires off when we change the value. This event should also pass the new value through.

We do this by creating a local class derived from UnityEvent, with the generic types (float in this case) detailing the parameters of any functions we attach to the event.

```
    [System.Serializable]
    public class FloatEvent : UnityEvent<float> { }

    public FloatEvent onValueChanged;
```

In order for the event to appear in the inspector, we add the System.Serializable attribute.

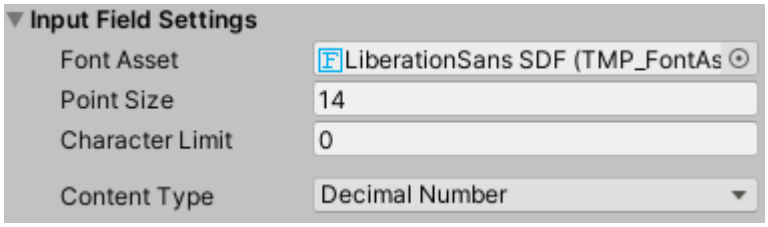
We then call this event from our set property like so:

```
public float floatValue
    {
        get { return _floatValue; }
        set
        {
            // update our internal variable
            _floatValue = value;

            // make sure all our controls are visually consistent
            if (slider)
                slider.value = value;
            if (input)
                input.text = value.ToString(formatString);

            // update any client code that has registered with our event
            onValueChanged.Invoke(_floatValue);
        }
    }
```

We automatically attach the events for the slider and input field in Start(), so that when the slider is moved or the input field is edited we invoke the setter for our floatValue property.

```
    void Start()
    {
        if (slider)
            slider.onValueChanged.AddListener((float value) => {
              floatValue = value; });
        if (input)
            input.onEndEdit.AddListener((string text) =>
            {
                float parsedValue;
                if (float.TryParse(text, out parsedValue))
                    floatValue = parsedValue;
            });
    }
```

We're using float.TryParse to turn the text in the input field into a number. You can set the TMP_InputField to only accept numbers (and not alphabetical characters or symbols) by setting its **Content Type** to **Decimal Number** in the inspector.



When the FloatEditor class is finished, we can change the references to musicVolume and soundFXVolume in the SettingsUI class into FloatEditors, and attach the callbacks for updating the Settings back-end object to the onValueChanged event of our FloatEditor class.

```
    public FloatEditor musicVolume;
    public FloatEditor fxVolume;

    private void Start()
    {
        if (musicVolume)
        {
            musicVolume.floatValue = settings.musicVolume;
            musicVolume.onValueChanged.AddListener((float value) => {
settings.musicVolume = value; });
        }
```