

Input Icons for Input System

Requirements (from package manager. Window → Package Manager):

- **TextMeshPro 2.1.6 or higher**
- **Input System 1.2.0 or higher (Package Manager → Unity Registry) (1.6.3 or higher recommended)**
- **2D Sprite 1.0.0 or higher (Package Manager → Unity Registry)**

Input Icons allows for easy display of input bindings in SpriteRenderers, UI Images and TMPro texts via components. You can also display bindings in TMPro texts using the style tag

(**<style=NameOfActionMap/NameOfAction>**). All displayed bindings get updated dynamically if the used input device changes.

For local multiplayer games we can use special components to display user specific bindings in SpriteRenderers and UI images.

The package includes sprites for keyboard/mouse and Xbox, PlayStation, and Nintendo controllers. If using a different controller, the Xbox sprites will be displayed by default (based on Steam statistics showing most PC players use Xbox controllers).

Video guide: <https://youtu.be/q9czf-6acsE>

1 Overview

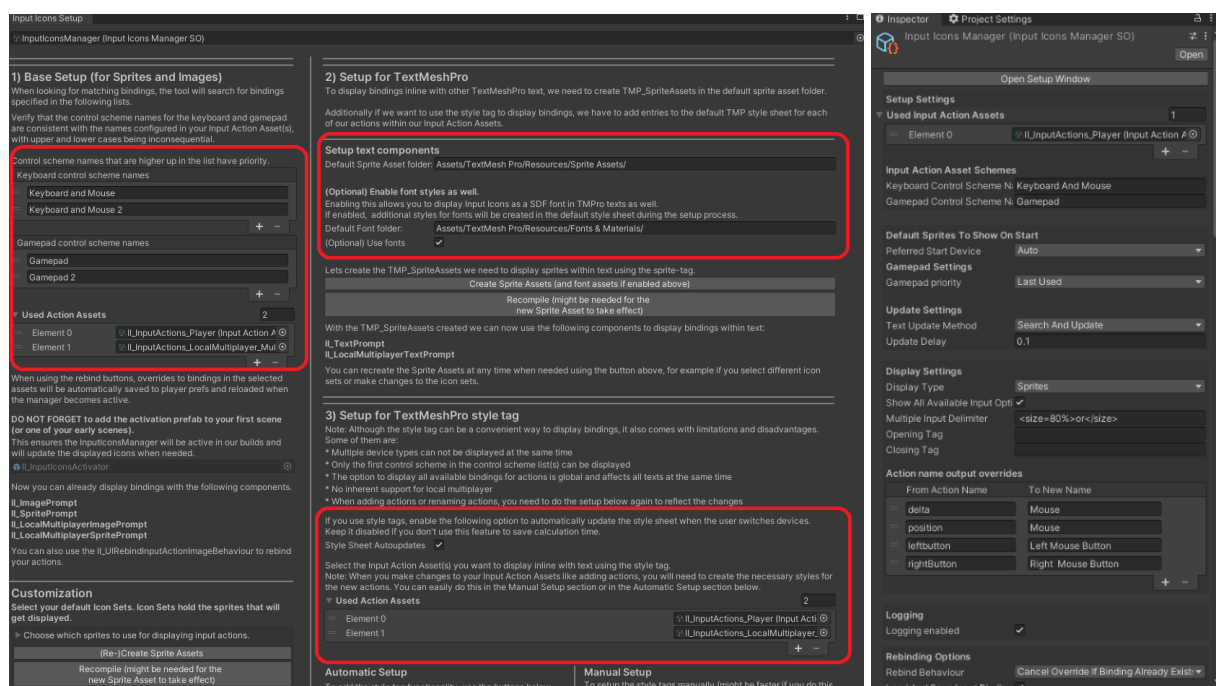
1.1	Updating From An Earlier Version.....	2
1.	Getting Started.....	3
1.2	The Base Setup (for Sprites and UI Images).....	4
1.3	The Setup for TextMeshPro	4
1.4	The Setup for TextMeshPro style tag	5
2	IMPORTANT FOR BUILDS: Add Input Icons Activator Prefab to Scene	6
3	Setting Up The Input Action Asset	7
3.1.1	Verifying Control Scheme Names.....	7
3.1.2	Assigning Devices To Control Schemes.....	7
4	How To Use Input Icons	9
4.1	Displaying Input Bindings.....	9
4.1.1	Displaying Bindings with Sprites and Images	9
4.1.2	Displaying Bindings in TMPro	11
4.1.3	Changing displayed Bindings in II_TextPrompts	13
4.1.4	Displaying Bindings in TMPro using the style tag.....	14
4.1.5	Displaying Bindings as an SDF Font in TMPro using the style tag	15
4.1.6	Displaying Bindings in Local Multiplayer Games.....	15
4.1.7	Changing displayed Bindings in II_LocalMultiplayerTextPrompts	18
4.1.8	Displaying Bindings in UI Toolkit	18
4.2	Quality Settings for Small Sprites.....	19
4.3	Keyboard And Gamepad Support	19
4.4	Rebind Buttons.....	19

4.4.1	Rebinding Using A Generated C# Class	20
4.4.2	Rebinding By Using Your Own Methods	20
4.5	Stopping/Starting The Automatic Update Behavior	20
4.6	Displaying A Specific Device Manually	21
4.7	Limitations.....	21
5	Customization	22
5.1	Using Different Sprites.....	22
5.2	Adding Custom Context Sprites	22
5.3	Performance Options	23
5.4	Display Settings	23
6	Steam Input Troubleshooting	24
6.1	User Preferences – Steam Input enabled.....	25
6.2	Local Multiplayer with Steam Input	26
7	Troubleshooting	27
7.1	Troubleshooting TextMeshPro style tags	28

1.1 Updating From An Earlier Version

When updating, it's essential to have a backup prepared in case of any unforeseen issues. Upgrading to a new version may result in your settings being reset to default values. Any modifications made to the InputIconManager and the InputIconConfigurator could potentially be lost and may need to be recreated. If you haven't made extensive changes, updating the references in the setup window (Tools → Input Icons → Input Icons Setup) should be sufficient. However, it's advisable to also inspect the Input Icons Manager (located in Assets/InputIcons/Resources/InputIcons/) for a comprehensive overview.

If you've made alterations to the icon sets that include the displayed sprites, consider creating a copy of them before the update. This allows you to reuse your custom icon sets after the update. Make sure to reassign your personalized icon sets to the InputIconSetConfigurator.



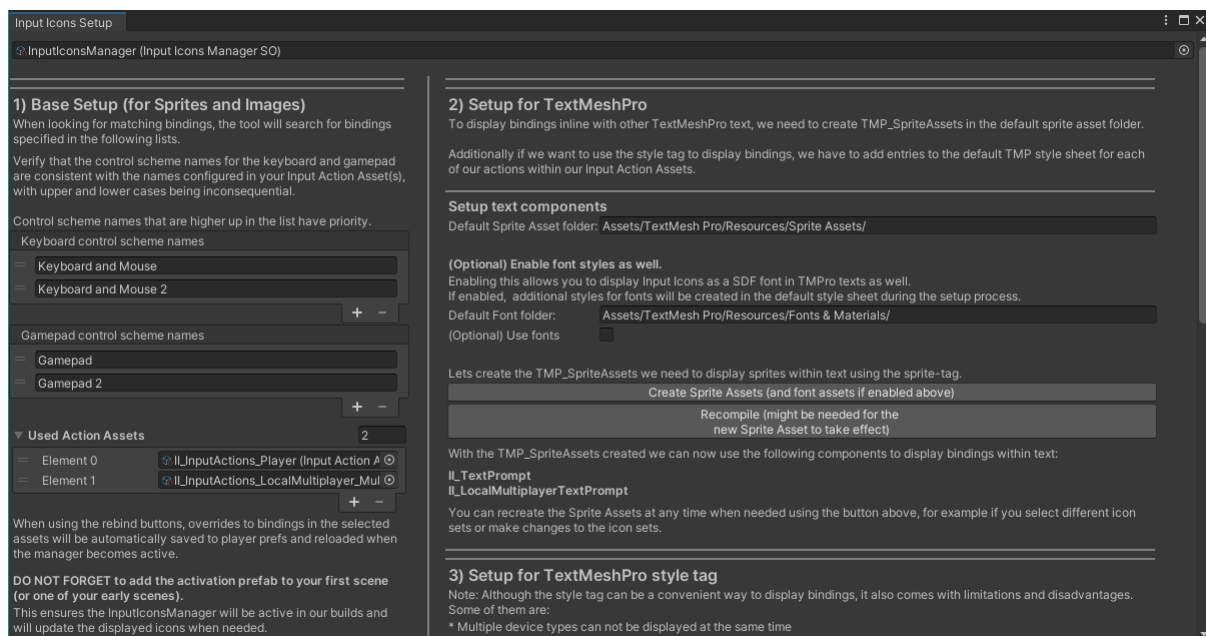
1. Getting Started

To start, open the setup window from the toolbar: “**Tools/Input Icons/Input Icons Setup**”

In the setup you find the necessary fields to configure Input Icons. Some customization options are also available in this window, but you might also want to check out the InputIconsManager and the InputIconSetConfigurator. More on that later.

The setup window is split into two parts, namely the Base Setup and the Setup for TextMeshPro.

- The **Base Setup** is enough to display dynamic bindings with SpriteRenderers and UI Images
- In the **Setup for TextMeshPro** we will create TMP_SpriteAssets for each device and add a bunch of entries to the default TMP Style Sheet. This will allow us to also display dynamic bindings along with other text. (If you don't need this feature, you might want to skip this part as the created TMP_SpriteAssets can take up 10-50Mb of space which you could otherwise save)



1.2 The Base Setup (for Sprites and UI Images)

With the setup window open (**Tools/Input Icons/Input Icons Setup**) all we need to do is to define the control scheme names we want to use in our Input Action Assets (if you need help with that, have a look at: [Setting Up The Input Action Asset](#)). We have separate lists for keyboard and gamepad control schemes. This way the tool can better differentiate between these two device types.

With the control scheme names setup you should already be able to display bindings with sprites and images using the scripts which come with this tool.

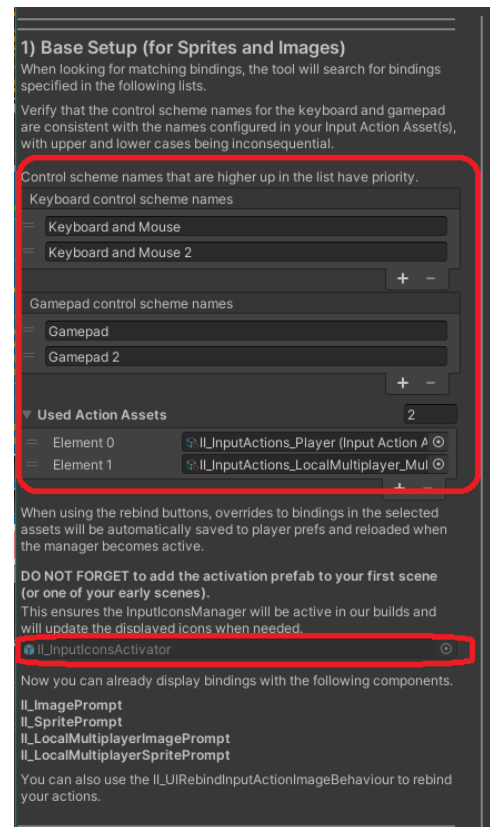
Note: You can define as many control scheme names as you want but I recommend using as few as possible and to reuse them in your Input Action Assets if you have several.

If you want to let users rebind their keys, you can fill the Used Action Assets list with your asset(s) and the tool will automatically save/load the overridden bindings when necessary.

For the final build, the InputIconsManagerSO needs to be referenced somewhere. We can easily achieve this, by dragging the `II_InputIconsActivator` prefab into an early scene.

With this done you can display the actions of your Input Action Assets with SpriteRenderers and UI Images (have a look at:

[Displaying Bindings with Sprites and Images](#))



1.3 The Setup for TextMeshPro

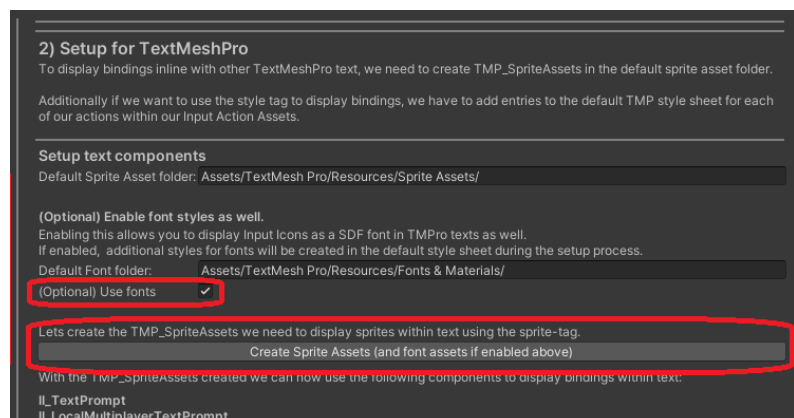
To display bindings in TMPPro texts, we use the sprite-tag to display our key sprites along with other text. But first we need to create `TMP_SpriteAssets` holding all sprites we will need.

If you moved the TextMeshPro folder to a different location, you need to update the path values before creating the `TMP_SpriteAssets`.

Enable the “Use fonts” checkbox, if you also want the ability to display bindings as a special SDF font. This is a font which holds symbols for each key/button instead of letters of the alphabet. The SDF font has the advantage that the edges will look sharp at all sizes while sprites can become blurry at a large size. But unfortunately they don’t have much detail.

After pressing the button in this setup area, the `TMP_SpriteAssets` will be created (and if we selected the “Use fonts” checkbox above, the SDF font assets will also be created. You might need to recompile for the new Sprite Assets to take effect.

Have a look at [Displaying Bindings in TMPPro](#) to see how you can display bindings along other text.



1.4 The Setup for TextMeshPro style tag

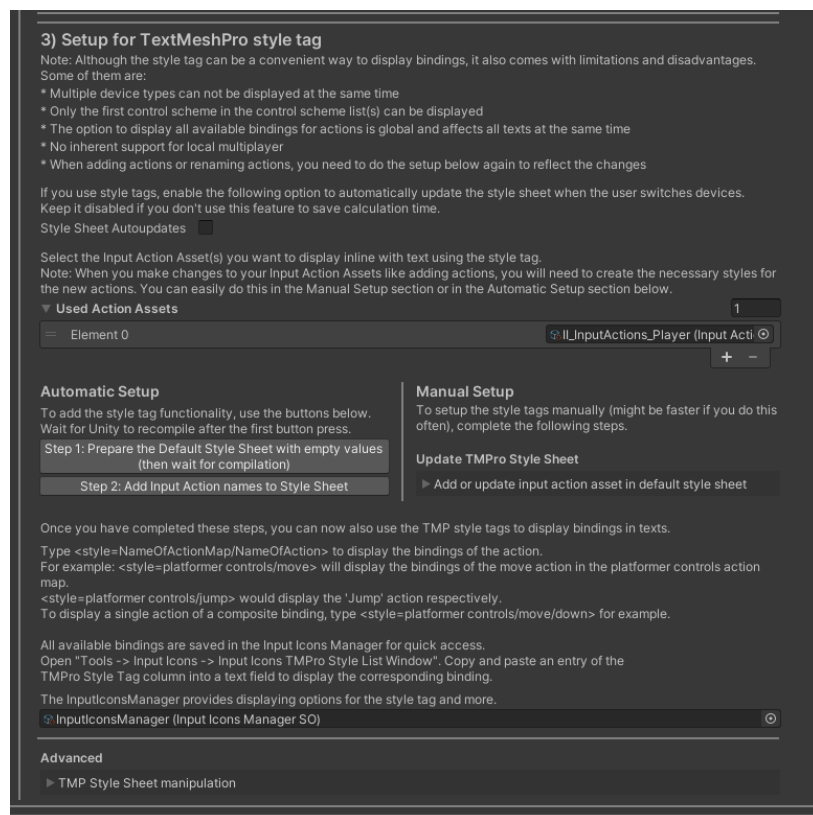
After completing the Base Setup and creating the TMP_SpriteAssets in the Setup for TextMeshPro we can also decide to make use of the style tag of TMPPro. The advantage of displaying bindings with the style tag is that we don't need any components and that this solution might work better in combination with other systems which also make use of tags.

However, this method also comes with limitations. Some of them are:

- Multiple device types can not be displayed at the same time
- Only the first control scheme in the control scheme list(s) can be displayed
- The option to display all available bindings for actions is global and affects all texts at the same time
- No inherent support for local multiplayer
- When adding actions or renaming actions, you need to do the setup below again to reflect the changes

Here is what we need to do in order to make the style tags work

- Create our Input Action Asset(s) and assign them to the “Used Action Assets” list in the setup window. Also define the same control scheme names we defined in the Base Setup (on how to do that look at: [Setting Up The Input Action Asset](#))
- Generate Style Sheet entries via either the Automatic Setup section or the Manual Setup section



And that's it! For more information on how to display bindings with the style tag look at: [Displaying Bindings in TMPPro using the style tag](#)

2 IMPORTANT FOR BUILDS: Add Input Icons Activator Prefab to Scene

It is important to know that Scriptable Objects only call their OnEnable methods when either being selected in the inspector or when being referenced by a MonoBehaviour or when being referenced by another Scriptable Object which is referenced.

Therefore the best way to ensure that the InputIconsManagerSO (which is a Scriptable Object) works properly in build as well, is to reference it in the scene. We can easily do this by just dragging the **II_InputIconsActivator prefab** into the first scene. Once the InputIconsManagerSO has been referenced it should stay active until you exit the game. There should be no need to keep the II_InputIconsActivator prefab through scene changes.

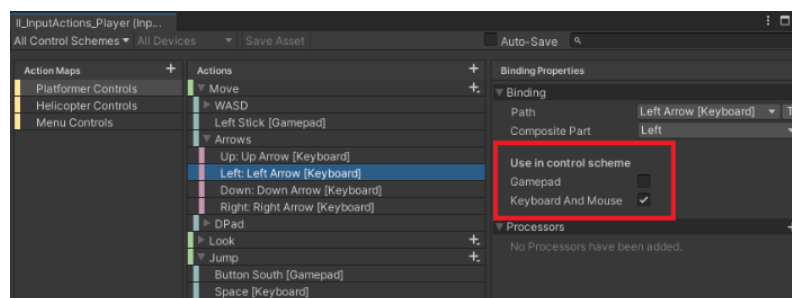
3 Setting Up The Input Action Asset

Our Input Action Assets (requires Input System 1.2.0 or higher ... Window → Package Manager) need control schemes for keyboard and gamepads (if we want to provide support for both). We have to add devices to the control schemes so the Input Icons Manager can decide which icons to display whenever we switch to gamepad or keyboard.

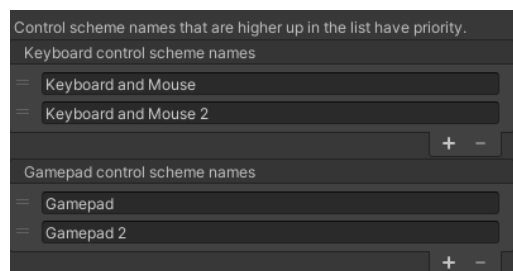
3.1.1 Verifying Control Scheme Names

Let's be sure that the Input Action Asset has **control schemes** set up for **keyboard and gamepad** control. Or only one of them, depending which device(s) we want to support.

IMPORTANT: I noticed, changing the control schemes on the Input Action Assets does not always yield immediate results for the displayed sprites, even after saving the Input Action Asset. Often, only a domain reload will update the sprites accordingly (just enter Play Mode with "Enter Play Mode Options" disabled in Project Settings → Editor → Enter Play Mode Settings)

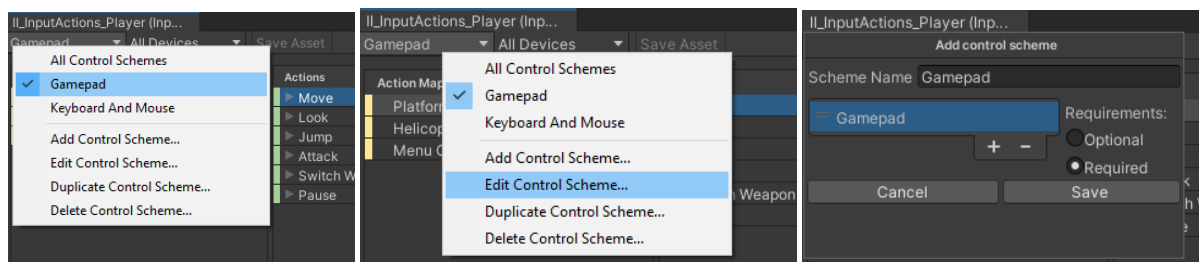


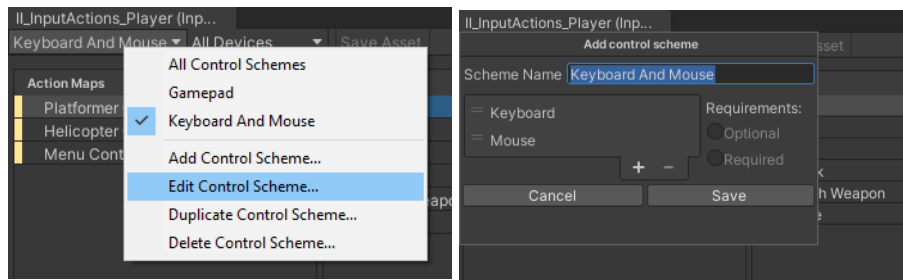
The InputIconsManager needs to know about these names. So if you have not already, fill in the names in the required lists in the setup window (Tools → Input Icons → Input Icons Setup) or in the Input Icons Manager scriptable object.



3.1.2 Assigning Devices To Control Schemes

We should now have two control schemes. One for gamepads, one for keyboard and mouse. We need to add devices to these control schemes in order for the manager to know which icons to display when the user switches to a different device.





4 How To Use Input Icons

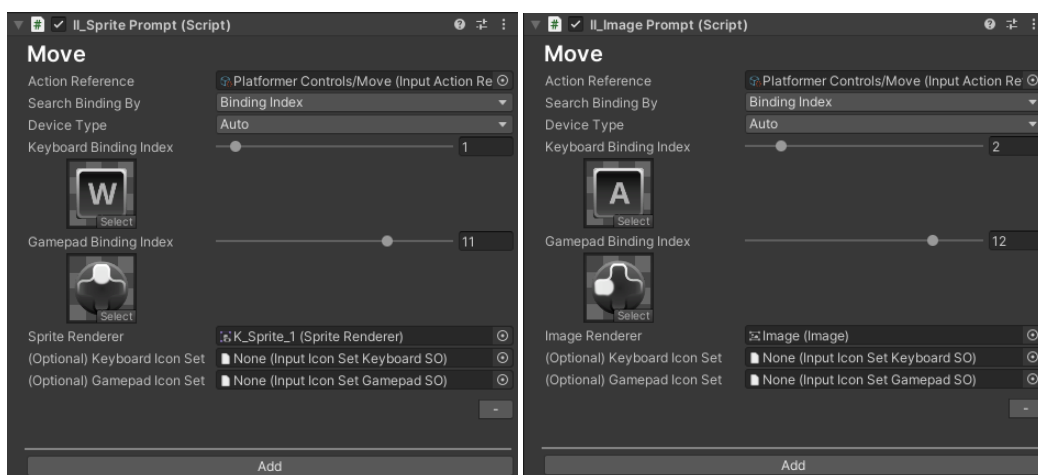
	Sprite Renderers and UI Images with components	TextMeshPro with components	TextMeshPro with style tag
Components	II_SpritePrompt II_ImagePrompt LocalMultiplayerSpritePrompt LocalMultiplayerImagePrompt (deprecated) II_SpritePromptBehaviour II_ImagePromptBehaviour	II_TextPrompt II_LocalMultiplayerTextPrompt	No components needed
Requirements	Matching Control Scheme Names (Input Action Assets and lists in the Setup Window) II_InputActionsActivator added to an early scene	TMP_SpriteAssets created in the default resources folder of TMPPro (by default: "Assets/TextMesh Pro/Resources/Sprite Assets/")	Input Action Assets added to the list of used Input Action Assets (Setup Window) Styles added to the default style sheet (through the Setup Window)
Info Sections	Displaying Bindings with Sprites and Images	Displaying Bindings in TMPPro	Displaying Bindings in TMPPro using the style tag

4.1 Displaying Input Bindings

Depending on which setups you did, there are several possible ways to display bindings.

4.1.1 Displaying Bindings with Sprites and Images

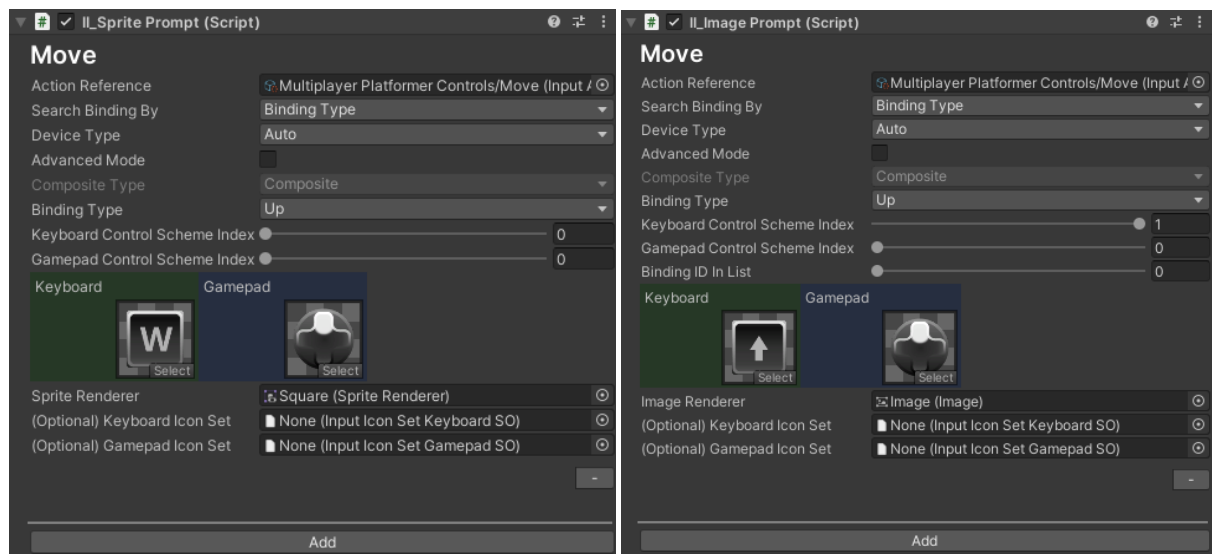
To show current bindings in **SpriteRenderers** or **UI Images** we can add the **II_SpritePrompt** or the **II_ImagePrompt** script to an object in the scene. Both components function in a similar way and use a custom inspector to automatically change their appearance and functionality based on the chosen settings.



Searching by "Binding Index"

Action Reference	The action we want to display. The tool will search for bindings fitting the defined control scheme names defined in the setup window.
Searching by Binding Index	The sliders below control which binding of the action to display.

	Simple, but beware: If you add/remove/move bindings in the Input Action Asset under that action, your displayed bindings might change.
Device Type	The device to display. Can be “Keyboard And Mouse”, “Gamepad” or “Auto”. When Auto is selected, the displayed sprite will switch between keyboard and gamepad depending on which device is being used.



Searching by “Binding Type” Normal Mode

Search Binding By “Binding Type”	This is generally the better option. It will give you additional fields depending on the input action you referenced.
Device Type	The device to display. Can be “Keyboard And Mouse”, “Gamepad” or “Auto”. When Auto is selected, the displayed sprite will switch between keyboard and gamepad depending on which device is being used.
Advanced Mode	Switches between normal and advanced mode. While in most situations normal mode is enough, advanced mode offers more flexibility – especially when we need to display composites (like WASD) and non composites (like a gamepad stick)
Composite Type (disabled)	In normal mode the composite type (composite or non-composite) gets determined automatically and will be composite if there is a composite binding assigned to the action. If keyboard and gamepad use different composite types (e.g. WADS and gamepad stick) you should use advanced mode instead.
Keyboard Control Scheme Index	If the tool detects several possible keyboard bindings but with different control schemes, this field lets you choose which control scheme to display
Gamepad Control Scheme Index	Same as the option above but for gamepad control schemes.
Binding ID in List	If there are several possible bindings in the chosen control scheme, choose which one to display. E.g. you have defined WASD and the arrow keys for the keyboard control scheme. This lets you choose between W and the up arrow for example.



Searching by “Binding Type” Advanced Mode

Control Scheme Index	Similar to the normal mode, you can select the control scheme index for keyboard and gamepad.
Composite Types	This lets you choose between displaying a composite (like WASD) or non composite (like a gamepad stick or a single button) binding. The field gets automatically set in the OnValidate method if there is only one type available on the referenced action.
Binding Types	If we chose a composite type in the field above, we can choose which direction (Up, Down, Left, Right, ...) should be displayed.
Available Binding Index	If there are still several options left, we can choose which one we want to display (e.g. W or the Up Arrow)

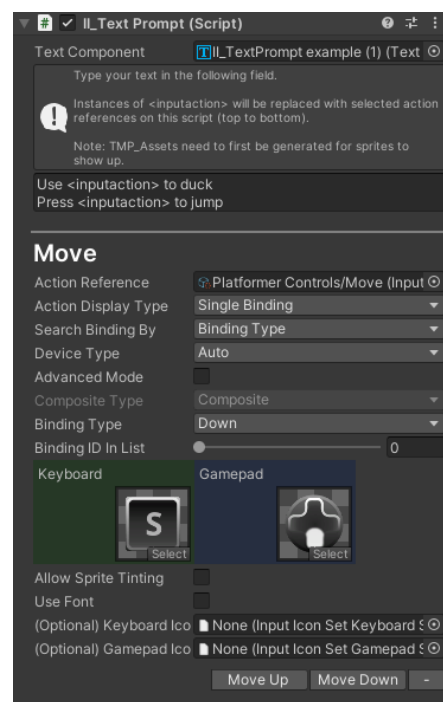
General settings

Renderer	The Sprite Renderer or UI Image component which will display the button prompt
(Optional) Keyboard Icon Set	If the displayed sprite is a keyboard sprite and this value is set, it will always display the sprite of the selected icon set instead of displaying the sprite of the currently used gamepad type.
(Optional) Gamepad Icon Set	Same as the option above, but for gamepads

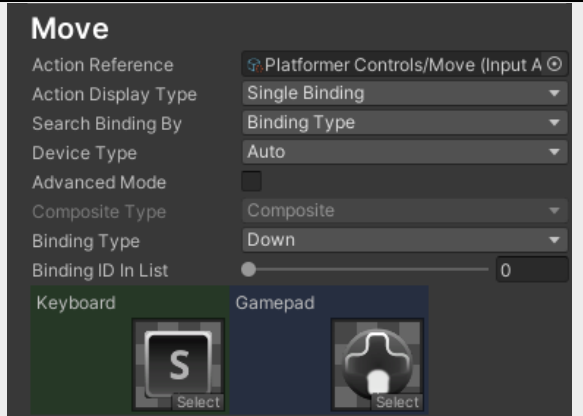
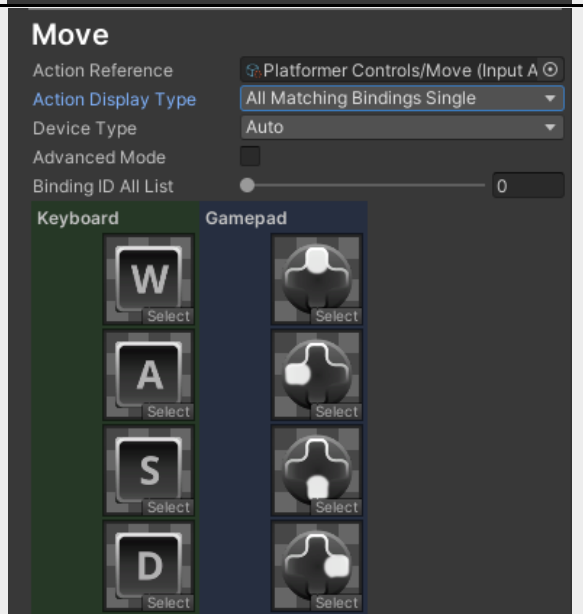
4.1.2 Displaying Bindings in TMPPro


We can use the `II_TextPrompt` script to display bindings along other text. In the inspector we need to assign a reference to a text component and an action reference. Then instead of typing the text into the `TextMeshPro` component, type it into the `II_TextPrompt` inspector instead. The `II_TextPrompt` script will apply the text to the `TextMeshPro` component and occurrences of `<inputaction>` will be replaced with sprite tags, displaying the bindings of the referenced action. You can display multiple action references with a single `II_TextPrompt` component by clicking the “Add” button at the bottom. The order of the action references matters. The first action reference will replace the first `<inputaction>`, the second reference will replace the second `<inputaction>` and so forth.

The base functionality is the same as discussed in [Displaying Bindings with Sprites and Images](#), but we also have the option to display several bindings at once. For example, to display the move keys (WASD) with a `II_SpritePrompt`, we would need four separate sprites. In text we can use a single text component to achieve this and more.



By changing the Action Display Type, we can select “**Single Binding**”, “**All Matching Bindings Single**” and “**All Matching Bindings With Delimiters**”

<p>Single Binding</p>	<p>This setting works the same as in the II_SpritePrompt and II_ImagePrompt and will display a single sprite.</p> <p>You can search the binding by Binding Type or by Binding Index</p> <p>Advanced Mode provides more controls just like with the other prompt components.</p>	
<p>All Matching Bindings Single</p>	<p>This can display multiple bindings and is useful for composite bindings like “WASD”.</p> <p>Binding ID All List: If multiple possible bindings are available, this slider can control which bindings to display. (e.g. WASD or the Arrow Keys)</p> <p>Advanced Mode provides more controls.</p>	

All Matching Bindings With Delimiters	<p>This option will display all available bindings. E.g. “WASD or Arrow Keys”</p> <p>Delimiter: This will be placed between the available bindings. It can be left empty or augmented with additional TMP tags like <size> or <color></p>	
--	---	---

General II_TextPrompt Settings

Allow Tinting	<p>This allows the displayed sprites for this action to be tinted, allowing you to display your sprites in the color you want, using the <color> tag before the <inputaction> tag.</p>
Use Font (experimental, icon switching not supported)	<p>If we tag this field, the displayed bindings will be displayed as a special SDF font instead of sprites. This ensures the displayed icons won't get pixelated at a large size.</p> <p>Unfortunately, TMP does not seem to support Unicode symbols very well when assigning a new text to a text field. The text field won't display the Unicode symbols appropriately, which means we can not use the font style in this manner when switching devices.</p> <p>Using <code>textComponent.OnRebuildRequested()</code> would fix the issue, but this method is not available in builds.</p> <p>If you know a fix or workaround, please let me know.</p>

4.1.3 Changing displayed Bindings in II_TextPrompts

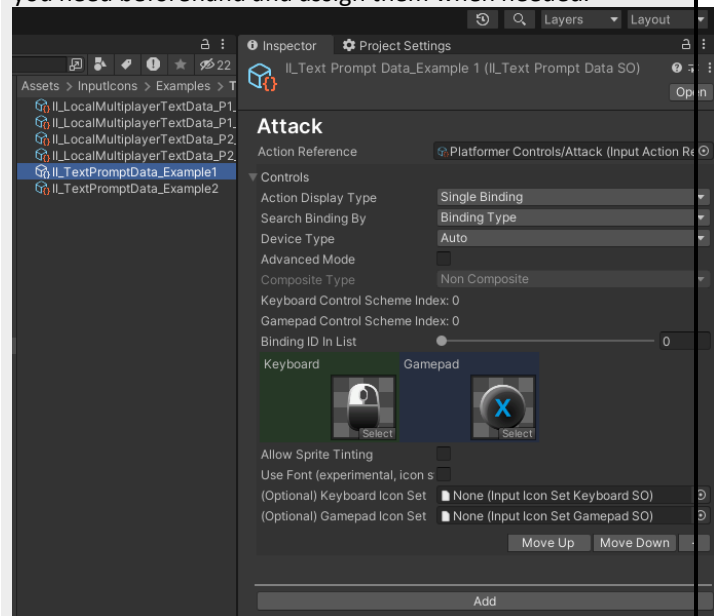
To change the displayed text and bindings in your text prompt components, it is best to use the following methods

SetText(string newText)	<p>This applies a new text and also refreshes the TMP component.</p>
SetTextPromptData(List<TextPromptData> newTextPromptDatas)	<p>Use this method to update the list of bindings to display.</p>

SetTextPromptData(IL_TextPromptDataSO textPromptDataSO)

Similar to the previous method, this can be used to update the list of bindings to display. First you need to create a Scriptable Object of type IL_TextPromptDataSO. This object manages a **TextPromptData** list, similar to the IL_TextPrompt component. Add the actions you want to display to that Scriptable Object, then call the SetTextPromptData method on the IL_TextPrompt to display the bindings defined in the Scriptable Object.

The best way to use this is to create all the Scriptable Objects you need beforehand and assign them when needed.



Also, have a look at the TextPromptsRuntimeUpdates example scene.

4.1.4 Displaying Bindings in TMPro using the style tag

This requires the completion of part three of the setup window. In this part we added the necessary styles to the Default Style Sheet of TMPro. After that, we can display bindings by using the TMPro-style tag (as can be found on the official website <http://digitalnativestudios.com/textmeshpro/docs/rich-text/#style>).

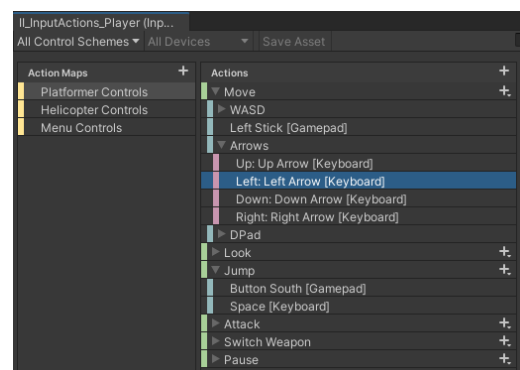
Although this method is not as flexible as using the IL_TextPrompt component, it can be handy to display bindings without the need of an extra component.

When we enter play mode, the opening tags in the style sheet will automatically be updated with the appropriate tags (given we enabled Style Sheet Autoupdates in the setup window), displaying the icons for the device currently in use.

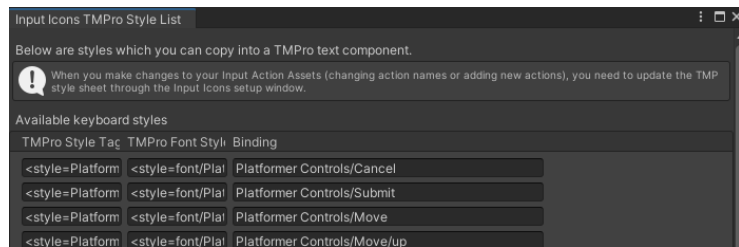
To display the bindings for a specific action, we can insert the following into a TMPro text field:

<style=NameOfActionMap/NameOfAction>. For example, to display the bindings for a "jump" action in the "Platform Controls" action map, we would insert **<style=Platform Controls/Jump>** into the text field. Similarly, to display the

bindings for a "move" action, we would insert **<style=Platform Controls/Move>** which would likely display either the "W A S D" keyboard keys or arrow key sprites, or the left stick sprite if a gamepad is being used. To display only one part of the move action we can write insert **<style=Platform Controls/Move/Down>**.



Open **Tools** → **Input Icons** → **Input Icons TMPPro Style List Window** to open a window which contains all the information needed to display bindings. To display a binding, simply copy and paste the entry in the first column (labeled "TMPPro Style Tag") into a text field. Or if you are using fonts, use the second column.



4.1.5 Displaying Bindings as an SDF Font in TMPPro using the style tag

Since version 2.0.0 it is possible to display a special font instead of sprites in TMPPro texts. The advantage is that SDF fonts will always appear crisp at any size whereas sprites can become pixelated when scaled up a lot. A disadvantage is that since it is a font, adding details to the displayed symbols can only be done by assigning a material to the text.

Displaying font input icons is very similar to displaying them as sprites. Just add "Font/" in front of the style. If you used "<style=Controls/Move>" to display the move action, you would write "<style=Font/Controls/Move>" now. The scheme is "<style=Font/ActionMap/Action>".

Just like with non font styles you can grab the style texts from the window: **Tools** → **Input Icons** → **Input Icons TMPPro Style List Window**

4.1.6 Displaying Bindings in Local Multiplayer Games

Achieving this the way you need in your game will very likely require some extra work. A good starting point is to have a look at the Local Multiplayer Prompts example scenes.

The InputIconsManagerSO manages a dictionary of users which you can access and manage through the static field **InputIconsManagerSO.localMultiplayerManagement**. Here are some important methods you can use.

AssignDeviceToPlayer (int playerId, InputDevice device, bool tryToReassign)	Use this to link a device to a specific playerId. If tryToReassign is set to true, this method will first search through all already available players and link the device to the original player if the device description matches the description of a player.
SetControlSchemeForPlayer (int playerId, string controlSchemeName)	Assign a control scheme to a player. This makes it easier for II_LocalMultiplayerSprite-/ and -ImagePrompts to display the correct sprites.
TryReassignDevice(InputDevice reconnectedDevice)	This will try to reassign a device based on the device description
UnassignDeviceFromPlayer(int playerId)	Sets the device for a specific playerId to null but keeps the playerId and the device description so the device may be reassigned in the future.

As I can not know how you want to treat your player IDs and possible use cases (like lost and reconnecting devices), I leave managing this list to you. Depending on how you spawn and how you control your players you have several options on assigning devices to players. In the following table is a short overview on how I did it in the example scenes. Of course you can also find your own solution.

Spawning players via Unity's Player Input Manager component	In this case I have a Player Input component attached to the player prefab which gets spawned. I also attached a script which reads some values from the Player Input component and sends them to the InputIconsManagerSO.localMultiplayerManagement. As every player has a unique device, there are only two control schemes we need to care about (1 for keyboard, 1 for gamepad)
--	---

```

private void Awake()
{
    if (playerPrompts)
        playerPrompts.SetActive(false);

    if (useAutomaticID)
    {
        playerID = nextID;
        nextID++;
    }
}

private void Start()
{
    PlayerInput input = GetComponent<PlayerInput>();
    if (input)
    {
        InputDevice myDevice = input.devices[0];
        usedControlScheme = controlSchemeNameKeyboard;
        if (myDevice is Gamepad)
            usedControlScheme = controlSchemeNameGamepad;

        InputIconsManagerSO.localMultiplayerManagement.AssignDeviceToPlayer(playerID, input.devices[0], false);
        SetControlSchemeName(usedControlScheme);
    }
}

public void SetControlSchemeName(string controlSchemeName)
{
    usedControlScheme = controlSchemeName;
    InputIconsManagerSO.localMultiplayerManagement.SetControlSchemeForPlayer(playerID, controlSchemeName);

    if (playerPrompts)
        playerPrompts.SetActive(true);

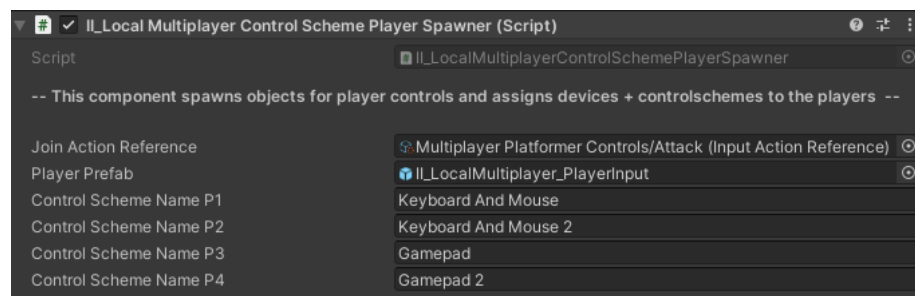
    InputIconsManagerSO.onInputUsersChanged?.Invoke();
}

private void OnDestroy()
{
    if (useAutomaticID)
        nextID--;
}

```

Spawning players through a self made player spawner

This method is more advanced, but allows several players to use the same keyboard through the use of separate control schemes. In the example scene we can spawn 2 keyboard players and 2 gamepad players (given that we have not tweaked the settings in the setup window of the tool)



The script listens for a join action, will then check if the control scheme which caused the join action is still available and not already in use and if it is available, it spawns a player in the AssignPlayerInput method.


```
private void HandleJoinAction(InputAction.CallbackContext ctx)
{
    InputControl controls = ctx.control;
    InputBinding binding = ctx.action.GetBindingForControl(controls).Value;

    string usedControlScheme = binding.groups;

    //control scheme already in use, do not spawn another player
    if (trackedControlSchemes.Contains(usedControlScheme))
        return;

    //control scheme not yet in use, spawn a player and assign the device + control scheme to that player
    if (usedControlScheme == controlSchemeNameP1)
    {
        AssignPlayerInput(controlSchemeNameP1, ctx.control.device);
    }
    else if (usedControlScheme == controlSchemeNameP2)
    {
        AssignPlayerInput(controlSchemeNameP2, ctx.control.device);
    }
    else if (usedControlScheme == controlSchemeNameP3)
    {
        AssignPlayerInput(controlSchemeNameP3, ctx.control.device);
    }
    else if (usedControlScheme == controlSchemeNameP4)
    {
        AssignPlayerInput(controlSchemeNameP4, ctx.control.device);
    }
}

}
```

In the AssignPlayerInput method we instantiate a new player through PlayerInput.Instantiate. From the spawned PlayerInput we can get the playerIndex and together with the device which triggered the spawn action, we can add the new player with its device to the InputIconsManagerSO dictionary by calling AssignDeviceToPlayer on the manager.

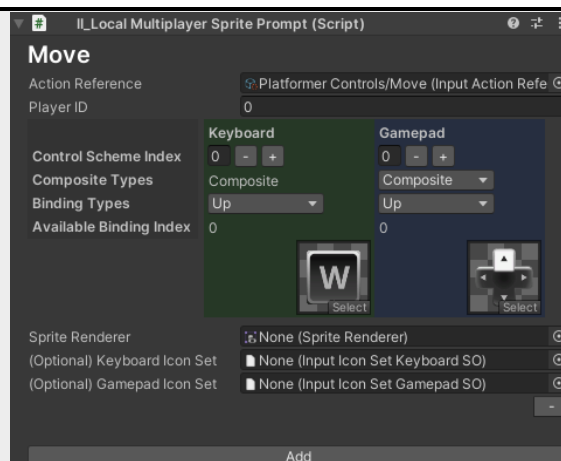
```
private void AssignPlayerInput(string controlScheme, InputDevice device = null)
{
    PlayerInput playerInput = PlayerInput.Instantiate(playerPrefab, controlScheme: controlScheme, pairWithDevice: device);
    InputIconsManagerSO.localMultiplayerManagement.AssignDeviceToPlayer(playerInput.playerIndex, device, false);

    playerInput.gameObject.name = "Player Input_" + playerInput.playerIndex;
    playerInput.GetComponent<II_LocalMultiplayerPlayerInputHandler>().controlScheme = controlScheme;

    trackedControlSchemes.Add(controlScheme);
    //Debug.Log("CONTROL SCHEME ADDED: " + controlScheme);
}
```

You can use the following components to display bindings in local multiplayer games for each player:

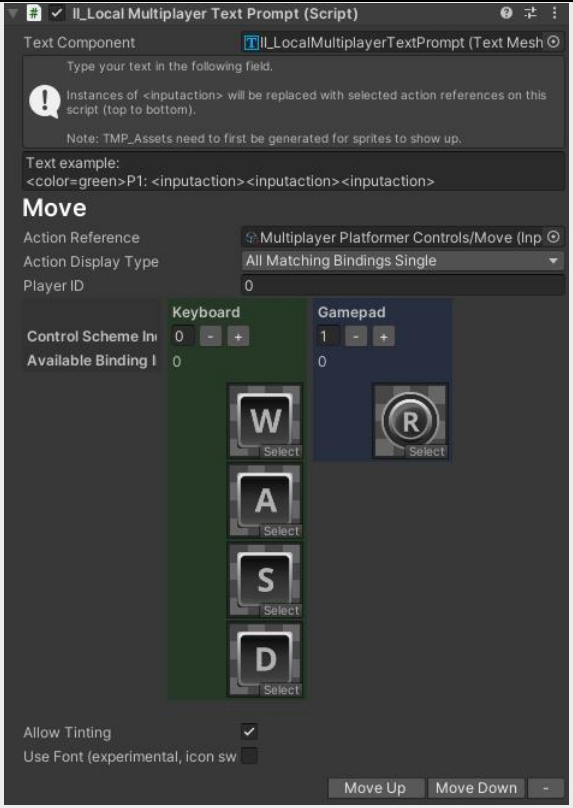
II_LocalMultiplayerSpritePrompt II_LocalMultiplayerImagePrompt



These components are somewhat similar to the II_SpritePrompt and II_ImagePrompt components. The difference is that we select a PlayerID we want to display. This PlayerID will be matched against the managed multiplayer dictionary on the InputIconsManagerSO to display the corresponding sprite.

During runtime, the Control Scheme Index can not be changed, as the scheme and the device are determined by the list on the InputIconsManagerSO dictionary. In the editor however, you can change the scheme to see which sprites will be displayed for each control scheme.

II_LocalMultiplayerTextPrompt



This component is similar to the II_TextPrompt component. But like the other multiplayer prompts, the control scheme and the device will be determined by the PlayerID.

The onInputUsersChanged event will be invoked within these methods, causing scripts like II_LocalMultiplayerSpritePrompt to update their displayed sprites.

Note this is quite a new feature and there might be bugs. If you think something important is missing, please let me know (support@octacube-studios.com)

4.1.7 Changing displayed Bindings in II_LocalMultiplayerTextPrompts

Updating text in local multiplayer text prompts does work very similar to updating general II_TextPrompt components. Have a look at the [4.1.3](#).

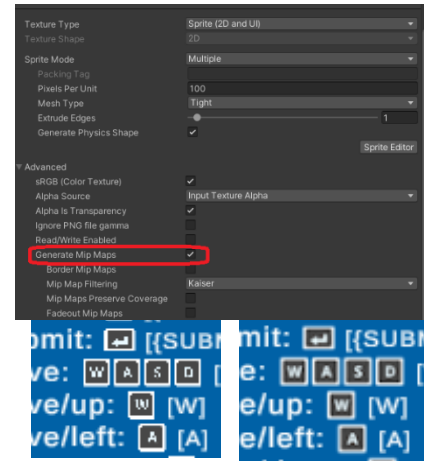
4.1.8 Displaying Bindings in UI Toolkit

There is an [extension asset](#) available which allows you to display Input Icons in UI Toolkit. That extension is available for free on the asset store as a separate package. (Some features like advanced mode for prompts, displaying input icons as fonts or displaying bindings for local multiplayer games is currently not possible in the UI Builder extension and would require additional work)

4.2 Quality Settings for Small Sprites

A common issue when displaying sprites at a very small size is that they can become quite unreadable as there are not enough pixels on the screen to render all the details. By changing the import settings of the sprites and the created sprite atlas in the Sprite Assets folder of TMPPro we can improve the readability of the letters quite a bit. Select the texture and enable **“Generate Mip Maps”** and hit Apply.

Since version 1.4.x this setting will be active for the generated sprite assets by default.



4.3 Keyboard And Gamepad Support

To support keyboards and gamepads, our Input Action Asset(s) need to be correctly setup (see section **“Setting Up The Input Action Asset”**)

In earlier versions, the Input Icons Manager would listen for the “Controls Changed” event on a Player Input component. Since Input Icons 1.3.0 this is no longer the case and changing between keyboard and gamepad icons now also works without a Player Input component in the scene.

4.4 Rebind Buttons

In the prefab folder we can find the `II_UI_RebindActionImageObject` which is a prefab we can use to rebind our input. You might want to create your own prefab with custom visuals but this prefab is a good starting point. (I recommend to create another prefab out of this prefab so your buttons don’t get messed up when you update this asset)



The script takes an Input Action as a reference which we can then rebind by using a button and then pressing the desired new key.

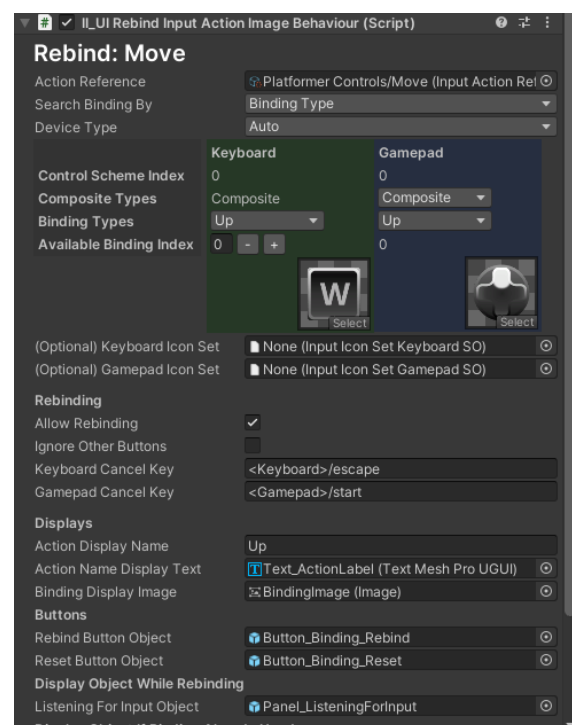
The settings to control which binding to display is very similar to the controls for Sprite-, Image-, and Text-Prompts.

The toggle **“Allow Rebinding”** can be turned off for input actions we don’t want to be rebound, but should still be visible to the player.

The toggle **“Ignore Other Buttons”** can be enabled if we want to keep the binding on that button, even if another rebind button gets bound to the same binding. This option is only available when the selected Rebind Behaviour for all buttons is set to **“Override Existing”**.

The text field **“Action Display Name”** provides a quick way to change the text of the action name label without having to select the text object in the hierarchy.

The last field **“Rebind Behavior”** can be used to choose the wanted behavior in case the player wants to bind a key to an already existing key. This option is global can also be found on the Input Icons Manager. The options are



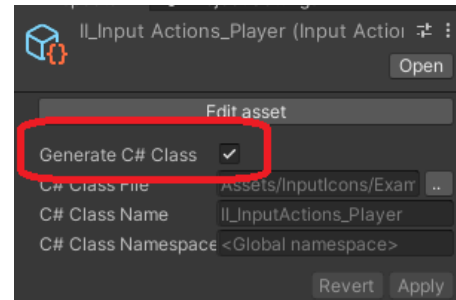
1. **“Override Existing”** in that case, the new binding gets accepted and the other action will be unbound.
2. **“Cancel Override If Binding Already Exists”** will display a message like “Binding already in use” to the player and cancel the rebind function. The player will have to first find and rebind the other action.

3. **“Always Apply And Keep Other Bindings”** will always accept the new binding. Other bindings will not lose their binding like in **“Override Existing”** when a conflict occurs.

4.4.1 Rebinding Using A Generated C# Class

If you use a generated C# class out of an Input Action Asset and you want to use the rebind button prefabs of this asset, you will have to do a little bit of additional work. The rebind prefabs which come with this tool can only rebind the actions of the Input Action Asset and can not rebind the actions of the generated C# class by default, therefore we have to write some code to keep the asset and the generated C# class in sync.

Whenever we change a binding using the prefab buttons, we also need to apply that new binding override to the generated C# class. I have already created a template class, called **“II_InputActions_ControllerTemplate”** to make this process easier. You can copy and paste the necessary code from the template into whichever class(es) you use for setting up the player controls.



Attention: In version 2.0.23 the following methods changed from 4 to 2 methods, making it simpler to implement, while also making the synchronization more reliable.

The template class contains the following important methods: **LoadSavedBindingOverrides** and **HandleAllBindingsReset**. The **LoadSavedBindingOverrides** method loads in the overrides we applied to the Input Action Asset (the overrides are saved in PlayerPrefs) and applies them to the instance of the generated C# class. The **HandleAllBindingsReset** method resets the bindings of the instance of the generated C# class to keep the bindings in sync.

In the **Awake** function we subscribe to events on the manager and call the above methods whenever needed.

For some more information you might also want to have a look at the example scene called **“InputIcons_ExampleScene3_GeneratedCode”**. This scene makes use of the template class to move the player object and to apply binding overrides to the generated C# class.

4.4.2 Rebinding By Using Your Own Methods

If you decide to not use the rebind buttons that come with this asset, you can use the **“InputIconsManagerSO.ApplyBindingOverridesToInputActionAssets”** method to apply binding overrides to the Input Action Assets managed by the InputIconsManager. The manger should then update the displayed input icons accordingly.

For example: You have generated a C# class out of an Input Action Asset and set binding overrides to that class using your own rebinding tools. In this case the InputIconsManager would not know about the changes made. You have to use the above mentioned method to sync up the changes every time you make changes, so the InputIconsManager can display the correct sprites.

4.5 Stopping/Starting The Automatic Update Behavior

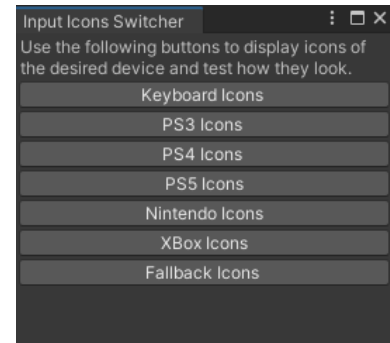
By default the tool will listen for device changes and automatically update the displayed icons when a change is detected. You can stop that behavior by calling the static **‘StopUpdatingIconsBehaviour’** method on the manager (**InputIconsManagerSO.StopUpdatingIconsBehaviour**). You can reenale the automatic update behavior by calling **StartUpdatingIconsBehaviour** respectively.

4.6 Displaying A Specific Device Manually

You can open a device switching window by going to **Tools → Input Icons → Input Icons Switcher**. Use the buttons in that window to change the displayed icons to whichever device is supported (Keyboard, PS3, PS4, PS5, Switch, XBox, Fallback).

This can be very useful to test how the buttons look in the scene without needing to enter play mode.

These buttons make use of a static method on the input icons manager called **InputIconsSO.SetDeviceAndRefreshDisplayedIcons** (**InputIconSetConfiguratorSO.InputIconsDevice inputIconsDevice**)



This is not fully fledged out and might display wrong keys if some bindings exist in the keyboard control scheme and not in the gamepad control scheme and vice versa.

In the example scene 10 (ChangeDisplayedSpritesManually) is a panel which provides the same functionality, but the custom editor window makes that panel somewhat obsolete.

4.7 Limitations

- Supported keyboard layouts (with the exception of special characters) are **QWERTY**, **QWERTZ** and **AZERTY**. For all other keyboard layouts, the QWERTY layout (which is the most commonly used layout) will be used to display action bindings.
- If we use the TextMeshPro Style tag and not the Prompt Components to display bindings: If we use more than one Input Action Asset, we should not use the same names for the action maps across these Input Action Assets. Doing so might cause the manager to override style strings in the TMPro Default Style Sheet.

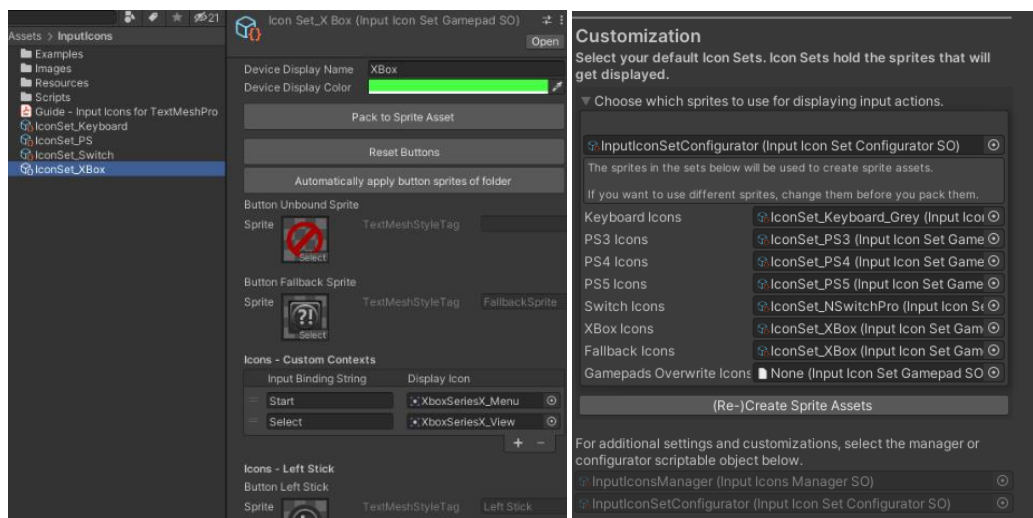
5 Customization

There are several ways to customize the behavior of the tool and the appearance of the bindings we want to display.

5.1 Using Different Sprites

The following steps describe how to use different sets of sprites:

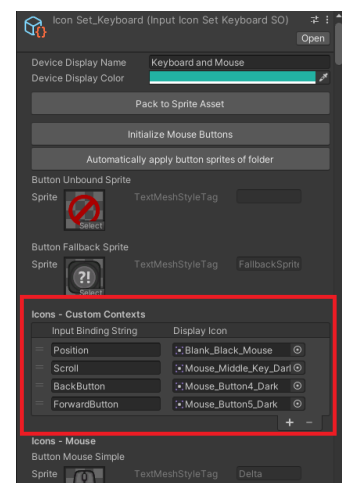
1. In the **Assets/InputIcons** folder are 2 types of Scriptable Objects that store the sprites used by the tool: one for keyboard sprites (InputIconSetKeyboardSO) and one for gamepad sprites (InputIconSetGamepadSO).
2. To use a different set of sprites, we can duplicate an IconSet and assign our desired sprites to the new Scriptable Object. One efficient way to do this is to drag the Scriptable Object into a folder containing our sprites and then click on “Automatically apply button sprites of folder” in the inspector of the Scriptable Object. This will try to assign as many sprites as possible (the success of this process depends on the names of the sprites).
3. For the “Custom Context” sprites, we have to apply the sprites manually as they cannot be assigned automatically.
4. Once we have our Icon Sets ready, we can drag them into the **Customization area of the Setup Window** and then hit the “(Re-)Create Sprite Assets” button.
5. Sometimes, we may need to enter play mode or recompile the code for the new icons to be displayed.



5.2 Adding Custom Context Sprites

The preset Input Icon Sets that come with this package (in the folder **Assets/InputIcons/**) already have sprites defined for all common input keys, gamepad buttons and joysticks. In case we need more input controls, we can add them in the Custom Contexts list in the Input Icon Sets. We need the input binding string and a display icon, to add a new binding.

An easy way to find the input binding string for an input is to use the script “II_UITextDisplayAllActions” which is in the example folder of this package. Add it to a TextMeshProUGUI object to display current input bindings. Then we can use a rebind button to override the current binding with the new binding we want to add. The input binding string for that binding will be displayed in the brackets. For the mousewheel, the input binding string would be “Scroll” for example.



Important: Whenever we make changes to an Input Icon Set, we will have to create a new Sprite Asset out of the Input Icon Set. Otherwise, our changes only exist in the Input Icon Set, but the desired sprite will not be available in the Sprite Asset used to display sprites within TPro texts.



5.3 Performance Options

We can improve the performance when the user switches devices by selecting the Text Update Method “Via Input Icons Text Components” in the Input Icons Manager.

The default setting is “Search and Update”, which will search for all text objects in the scene and update them when the user switches devices.

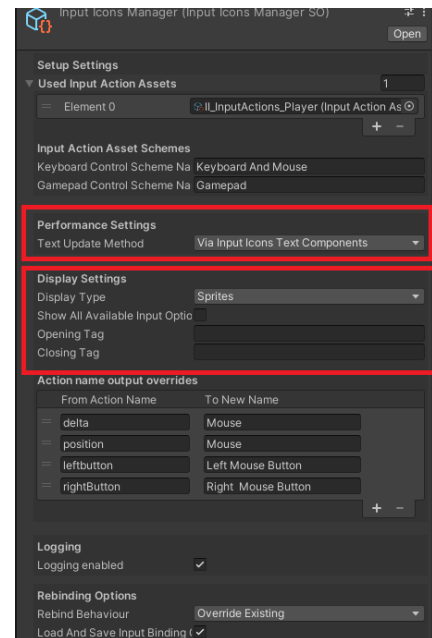
By using the “Via Input Icons Text Components” method, the manager won’t have to search through the whole scene every time the user switches devices. But we have to ensure that the required text objects have a InputIconsText component attached to them.

5.4 Display Settings

The InputIconsManager provides options for how we want to display the input action bindings.

Show All Available Input Option: We can switch between showing only the first available binding, e.g. “WASD” or all available bindings, e.g. “WASD or Up Left Down Right” for a move action.

Multiple Input Delimiter: If we show all available options, we can define a delimiter between these actions. This delimiter can have TPro tags for customization. The standard delimiter is “<size=80%>or</size>” and will therefore display a slightly smaller “or” between different bindings.



Opening and Closing Tag: Here we can add general styling options for the displayed icons. For example we can write <size=120%> in the opening tag and </size> into the closing tag to make all displayed icons a little bit bigger.

Display Type: input actions as sprites, text, or text in brackets.

Text Display For Unbound Actions: If we display bindings as Text or TextInBrackets, we can choose which text should be displayed for an unbound action.

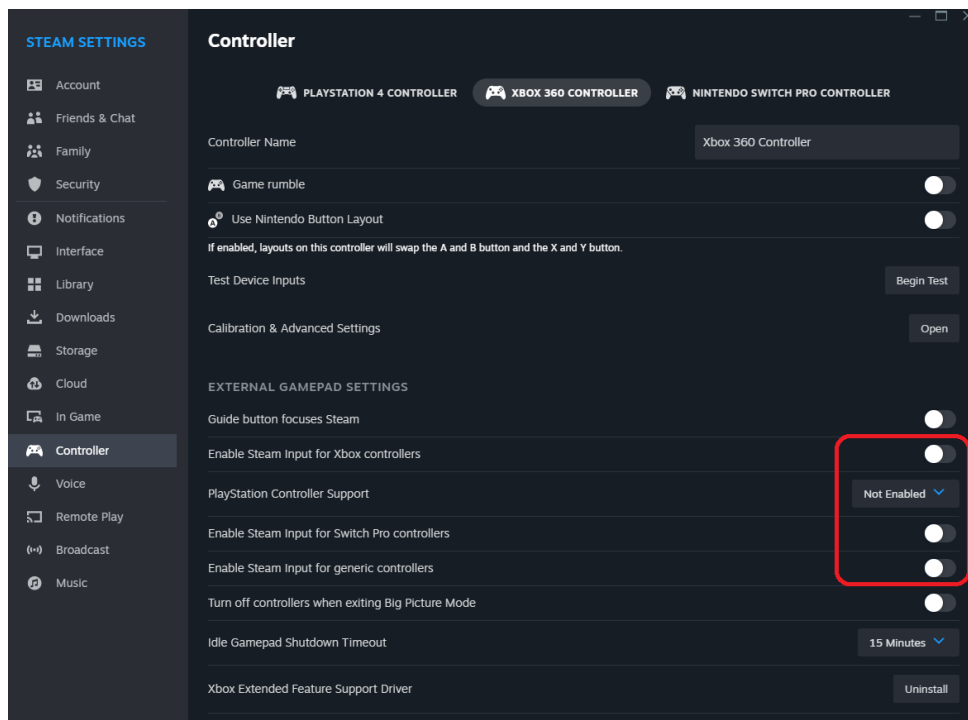
Text Display Language: If we display bindings as Text or TextInBrackets, we can decide if we want to display this text in English or in System Language (the language currently used by the device, might produce very different results in text length and is generally not recommended).

Action name output overrides: If we use Text or TextInBrackets as the option to display input actions, we can override the text which would be displayed. For example it makes more sense to display [MOUSE] instead of [POSITION] for pointer controls.

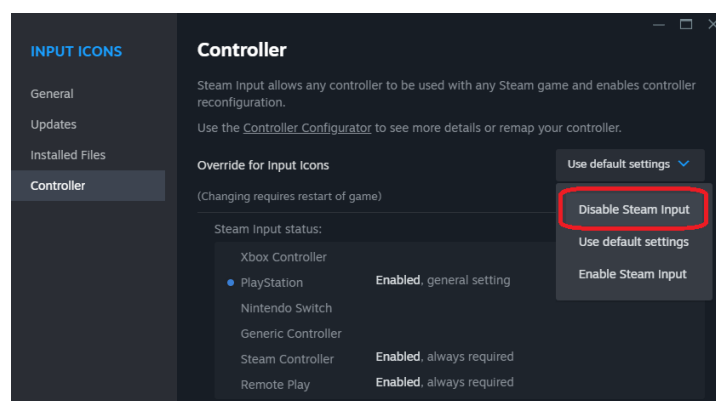
6 Steam Input Troubleshooting

Integrating Steamworks.NET (<https://steamworks.github.io/>) into the project can cause some problems for the new Input System as in specific cases the Steam launcher will take over XInput, which might cause the new Input System to mistake gamepad devices for a different type. If you use a PS4 controller, Input System might think it is a Xbox controller for example. It all depends on the preferences of the player and which settings they enabled in the Steam launcher.

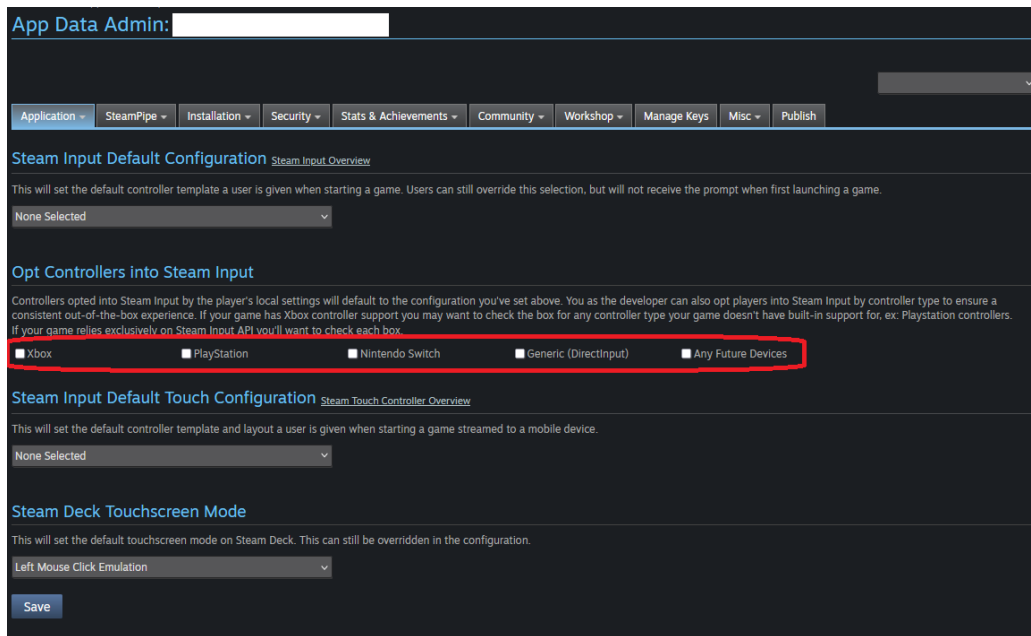
The easiest way to ensure the same behavior as in the Unity Editor and Non-Steam games is to go into the steam settings (from the Steam launcher: Steam -> Settings -> Controller) and ensure that (Steam Input-) controller support is disabled for each controller type. I also noticed if one of these is enabled, it can happen that the gamepads are not responsive in the Unity Editor if Steam is running. So for development I recommend disabling these settings.



These settings can also be overridden per Steam game by right clicking the game in the library and opening “Properties... -> Controller” and selecting “Disable Steam Input” in the dropdown menu. You might want to recommend this setting to your players, although most people will have Steam Input disabled by default anyway.

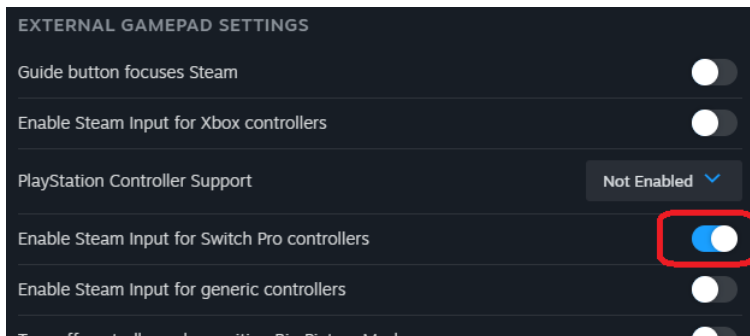


Your games Steamworks page also allows you to opt controllers into Steam Input. I recommend leaving them unchecked so Steam Input won't take over by default and won't mess with Unity's Input System as described above. So the setting should look like this:



6.1 User Preferences – Steam Input enabled

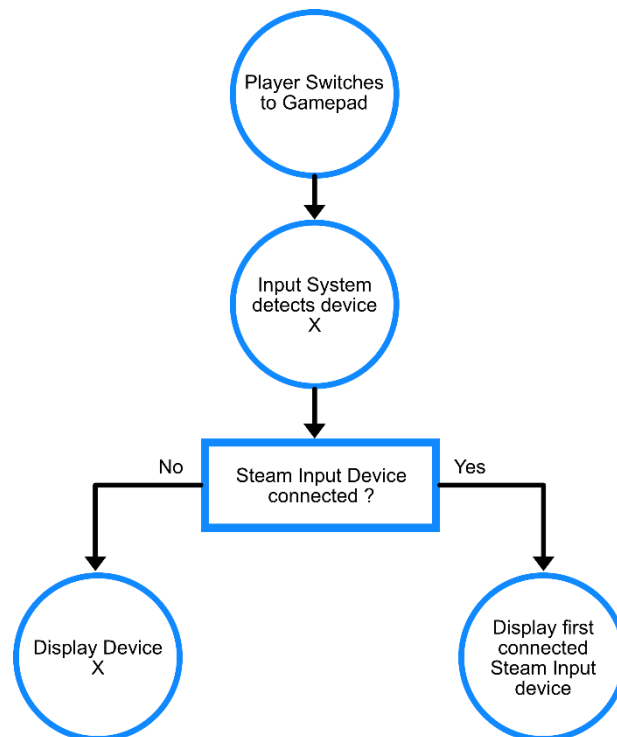
Your players might still have their own preferences and might decide to enable Steam Input for a specific device. In the example below, I enabled Stem Input for Switch Pro Controllers.



In this case, the Unity Input System won't recognize a connected Switch Pro controller as such anymore as Steam Input takes over for this controller. The behavior changes to the following:

	Switch Controller	Play Station Controller	XBox Controller
Switch Pro Controller is connected	Input System detects XBox controller. Showing Switch Icons	Input System detects PS controller. Showing Switch Icons	Input System detects XBox controller. Showing Switch Icons
Switch Pro Controller is NOT connected		Input System detects PS controller. Showing Play Station Icons	Input System detects XBox controller. Showing XBox Icons

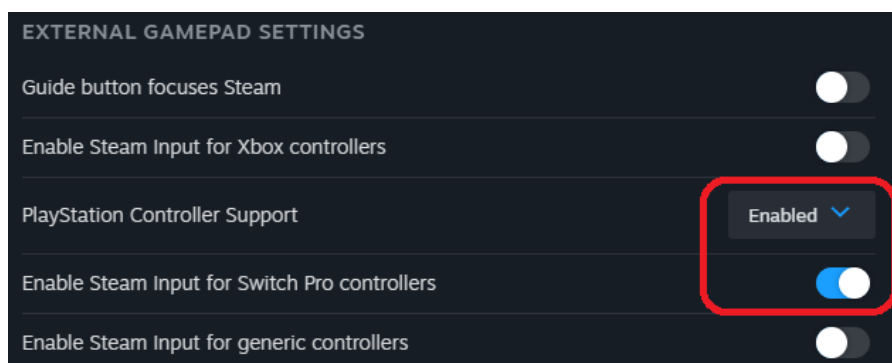
This behavior is a result of the InputIconsSteamworksExtensionSO.cs script. When it detects a controller for which Steam Input is enabled, it will override the Icon Set detected by Unity's Input System with the Icon Set matching the first connected Steam Input controller. The behavior tree looks like the following.



You can remove this behavior by removing the `InputIconsSteamworksExtensionSO.cs` script or the code within this script and by also removing the creation of the Instance of this scriptable object in the `OnEnable` method of the `InputIconsManagerSO.cs` script. But note if you do so, players who have enabled Steam Input for a specific device will likely not get displayed the correct icons while using that device and user preference should always go first.

6.2 Local Multiplayer with Steam Input

As of now I have not found a way to display the correct icons to players who have Steam Input enabled for a specific gamepad types. If Steam Input support is enabled for the PlayStation controller and the Switch Pro controller like in the example below, people using these controllers will also see Xbox icons, as the new Input System will detect these devices as Xbox controllers.



However, I strongly believe that most people won't have these settings enabled and as of my knowing, these settings are disabled by default in the Steam settings. So we probably won't have to worry about that too much.

But should issues with non correct icons arise, this is a good place to start your investigation and kindly ask players to disable Steam Input for this specific game if they have it enabled.

If you are facing troubles or know a way to provide better support for Steam Input while using the new Input System, please reach out to me, either by email: tobias.froihofer@gmx.at or through the forum thread of this tool: <https://forum.unity.com/threads/input-icons-for-tmpro-easily-display-action-bindings.1253292/>

7 Troubleshooting

Compile errors on import

- In an empty project with TMPro and the new Input System installed, you should not face compile errors. If you do however, please contact me as there might be some issues I don't know of in specific Unity versions.
- **Steamworks / InputIconsSteamworksExtensionSO**: If you get a compile error stating that the type or namespace name 'Steamworks' could not be found, then you probably have STEAMWORKS_NET in your Scripting Define Symbols in the Project Settings and are using some type of plugin to make calls to the Steam API. For Input Icons I used the open source framework Steamworks.NET (<https://github.com/rlabrecque/Steamworks.NET/releases>), and if you don't have that package in your project (while having STEAMWORKS_NET in your Scripting Define Symbols), you might face the above mentioned compile error(s). Adding that package should fix the problems, if not, also check the Assembly Definition called "InputIcons" in the "Assets/InputIcons/Scripts/Runtime/" folder: com.rlabrecque.steamworks.net should be added to the Assembly Definition References. **If you want to use a different framework** and not Steamworks.NET, you might have to add some custom code for Steam games. The **InputIconsSteamworksExtensionSO** class listens for an event named **InputIconSetConfiguratorSO.onIconSetUpdated** and will override the current Icon Set with whichever device is currently detected by Steam. You probably have to add this functionality through the framework you are using.

Most other problems appear from an incorrect setup. Remember, whenever you make changes to an Input Action Asset, to also update the style sheet by using the setup tool (Tools → Input Icons Setup).

Also make sure, the control scheme names in the setup window match the control scheme names in your Input Action Assets (sometimes an empty space slips in at the end of a string, it can happen to everyone)

If you have updated to a new version of Input Icons, the values in the InputIconsManager might very likely have changed to the default ones. Open the setup window again and reassign your Input Action Asset and the control scheme names.

Error: "... The type or namespace name '...' does not exist in the namespace '...' (are you missing an assembly reference?)"

- Assembly Definitions got added in v1.3.22. To properly include them, the folder structure of the asset needed to be changed. When updating to the latest version you could end up with a wrong folder structure, making the Assembly Definitions not work correctly. To solve the issue, **delete the content of the "Scripts" folder in "InputIcons/Scripts" and reimport the asset** or the Scripts folder again.

When you rebind using the rebind prefabs, the icons are updated correctly, but the controls don't change.

- This can happen when you use a generated C# class to handle player input. The rebind prefabs can only rebind the actions of the Input Action Asset and not the actions of a generated C# class. Check section "[Rebinding Using A Generated C# Class](#)"

Controls are weird or do not work anymore

- The reason might be that you have several active Player Input components in the scene. Search the hierarchy for "Player Input".
- Another reason could be that you run the Unity Player via Steam. Steam can mess up your controls if run simultaneously.

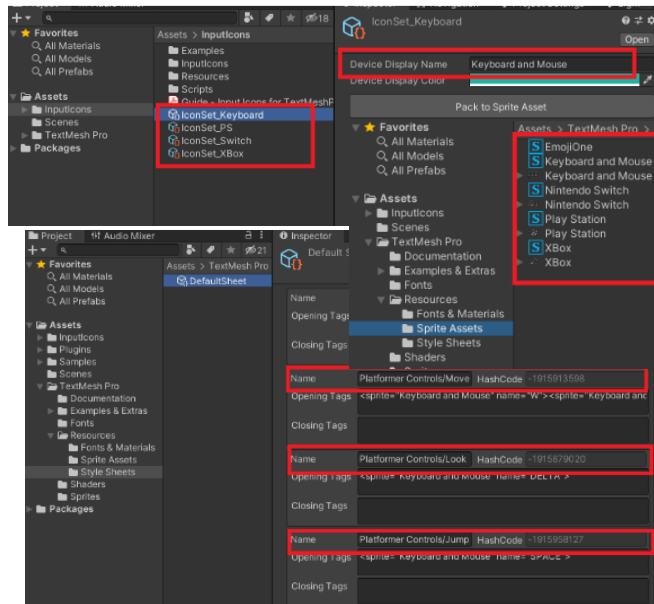
The icons change correctly in the editor but not in the build

- Make sure you added the `II_InputIconsActivator` prefab to the first scene (this ensures the Input Icons Manager will be active and listening for device changes)
- In some strange cases you might have to recreate the Input Action Asset. I had users reporting that they created their Input Action Asset with Input System 1.3.0, then updated to 1.4.4 and had to recreate their Input Action Asset. Otherwise somehow in the final build the 1.3.0 version was still present.

7.1 Troubleshooting TextMeshPro style tags

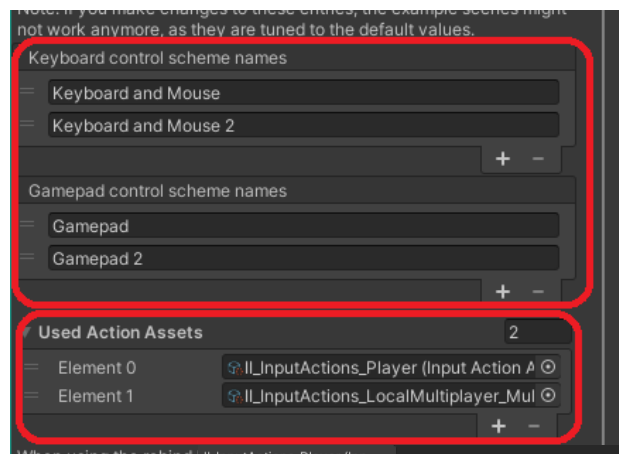
Have our Sprite Assets been created correctly and does the TMP Pro Default Style Sheet contain our actions?

- **Sprite Assets:** In “Assets/TextMesh Pro/Resources/Sprite Assets/” we should find assets containing the sprites of the keys we want to display. It is important that these assets have the same names as the Icon Sets in “Assets/InputIcons/”.
- **TMP Pro Default Style Sheet:** The Default Style Sheet should somewhere contain the actions defined in our Input Action Assets. Check the style sheet in “Assets/TextMesh Pro/Resources/Style Sheets/”
Special case: The styles were added correctly, but the styles disappear once we restart Unity ... the Default Style Sheet sometimes won't save the changes. A workaround is to make a small change manually in any of the fields of the style sheet and then undo the change. This seems to work and the changes will be saved.



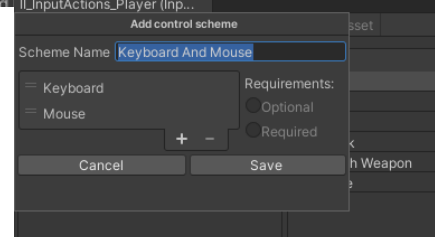
If we don't see any input icons make sure that:

- The “Used Input Action Assets” list contains all Input Action Assets we use and we have updated the Default Style Sheet with all changes we made to the Input Action Asset(s).
- All our Input Action Assets have **control schemes for keyboard/gamepad** and their names are **equal to** the ones set in the Input Icons Setup or in the **Input Icons Manager** respectively.
- The **control schemes** of our Input Action Assets have **devices** added to them (see chapter “Adding Devices To Control Schemes”)



Icons are shown, but don't get updated when a different device is used:

- Enable Style Sheet Autoupdates in the setup window under “Setup for TextMeshPro style tag”
- Check section “Keyboard And Gamepad Support”



If you have gone through this guide and still experience problems, it is probably an unknown bug. Please contact me at tobias.froihofer@gmx.at

There is also a [Forum](#) page about this asset. Join the discussion.