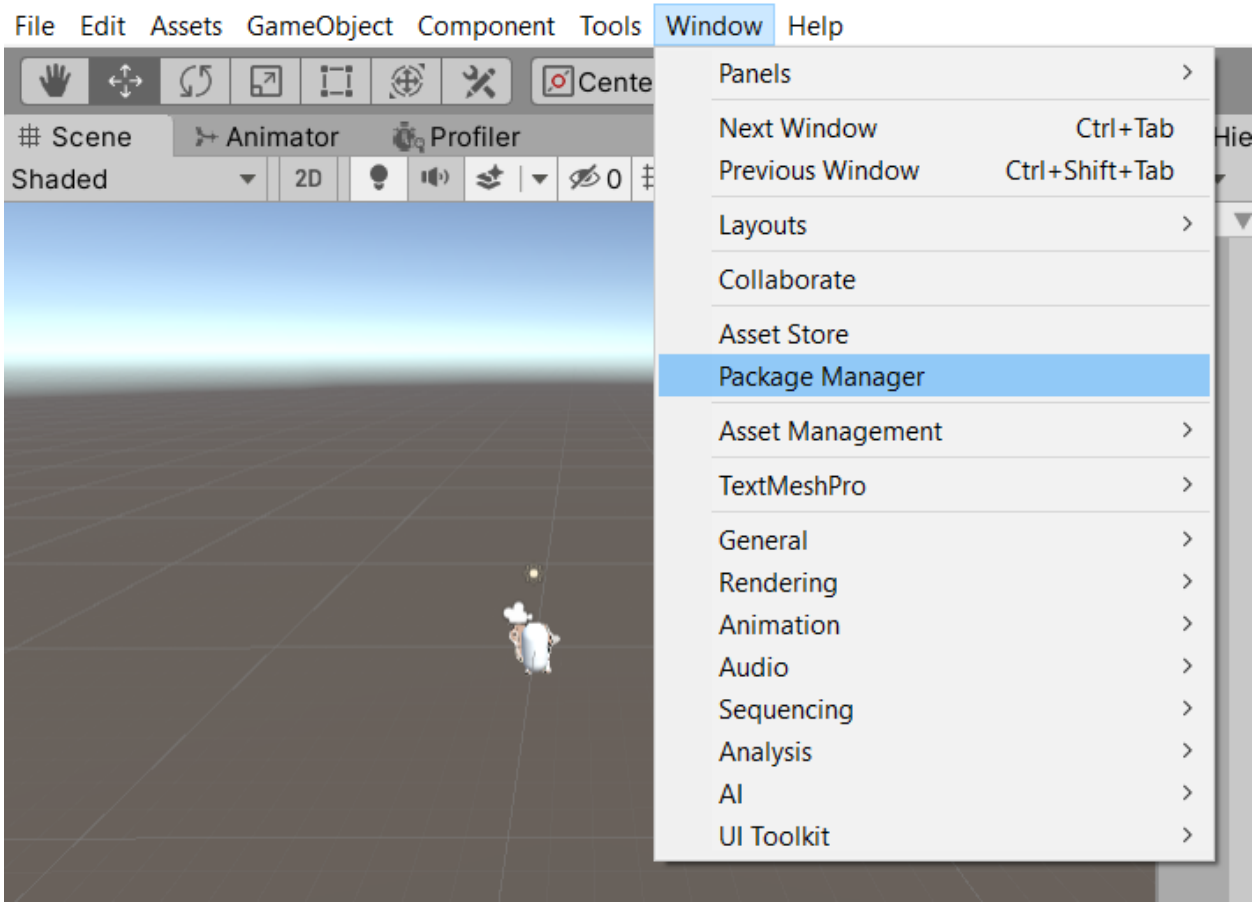
New Unity Input System Documentation

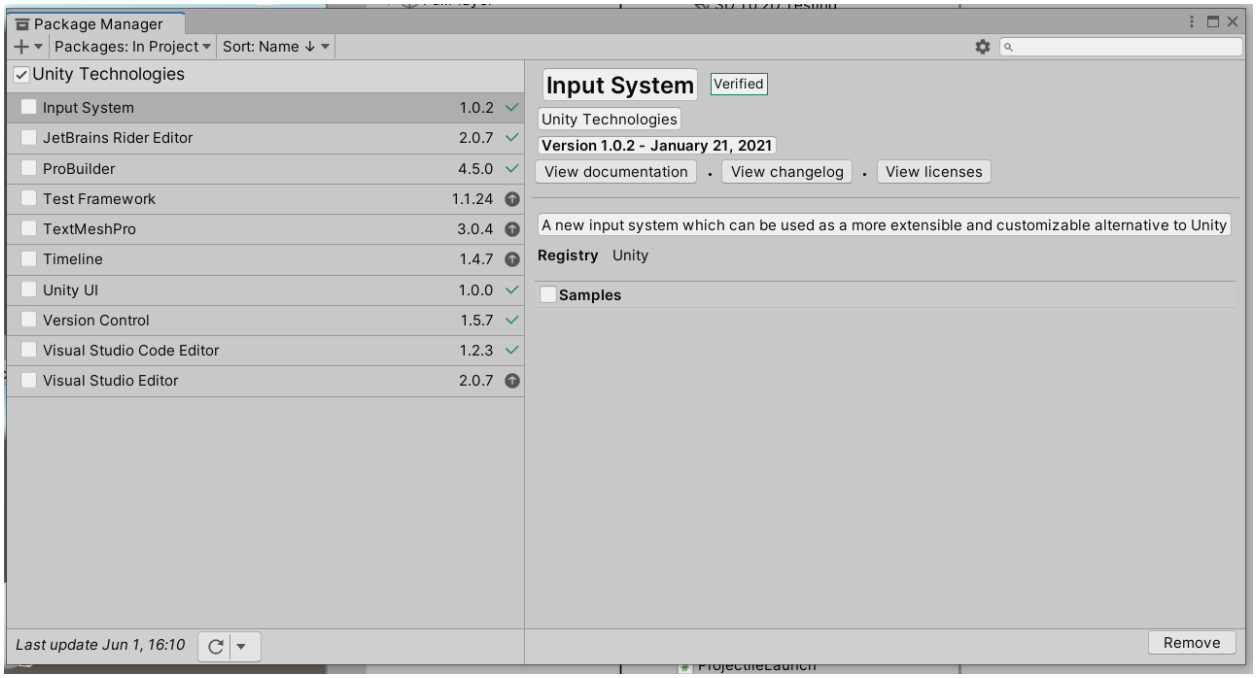
Table of Contents:

1. Installation:

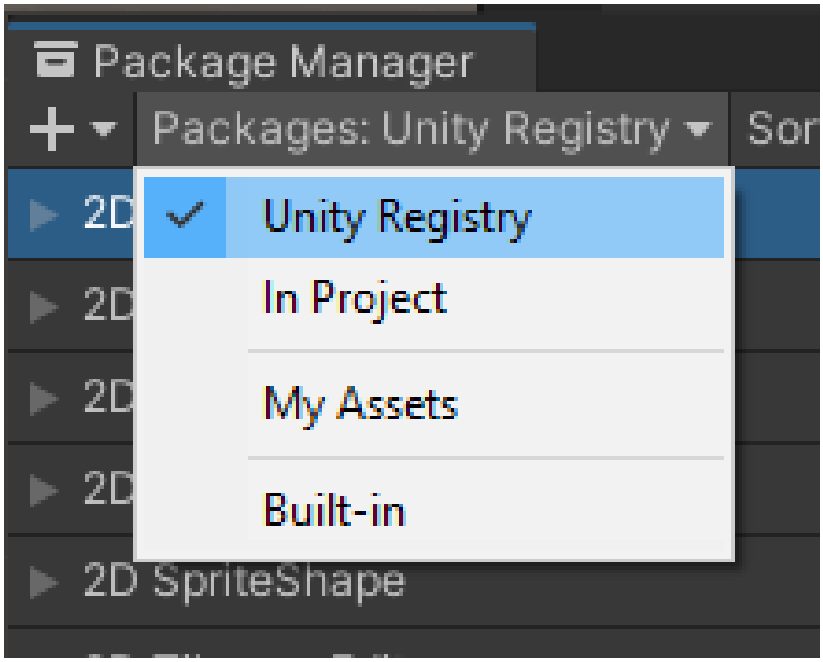
1. In Unity, go to **Window** -> **Package Manager**:



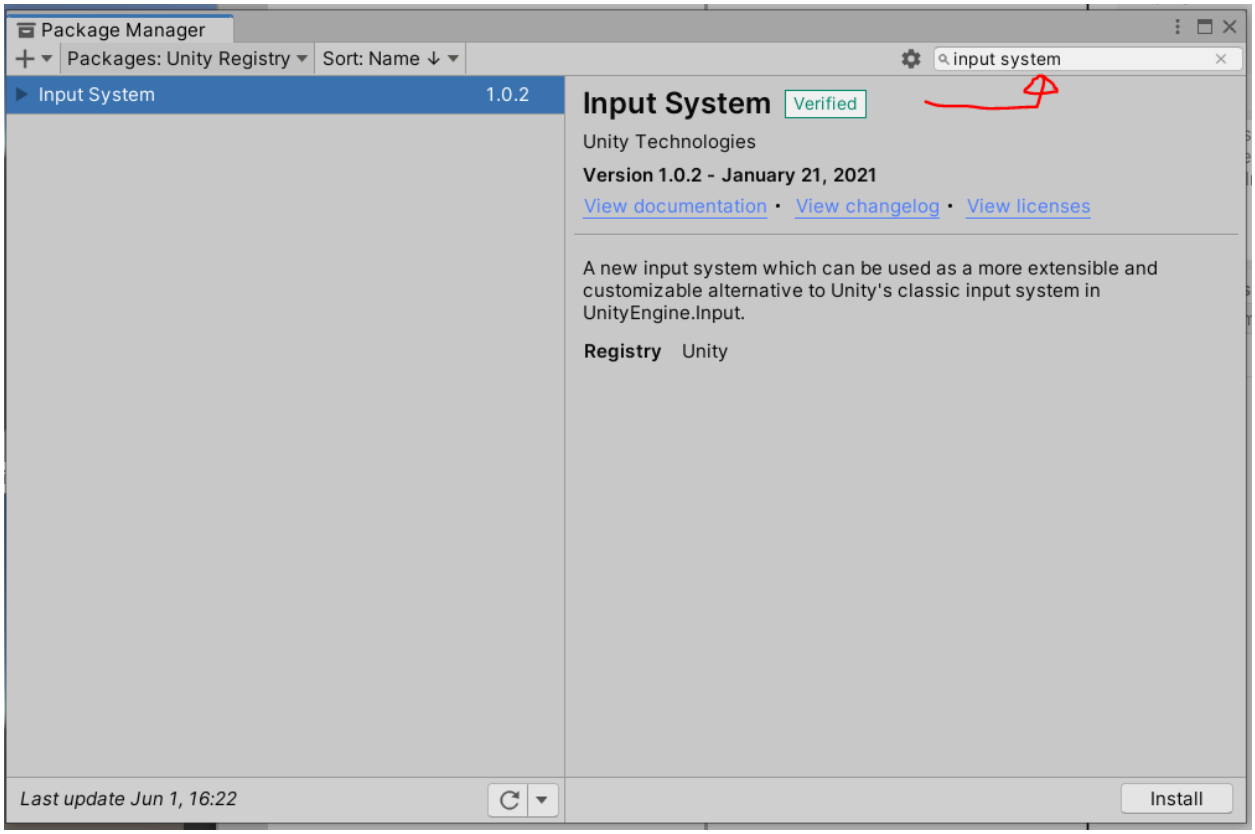
2. You will see a screen similar to this:



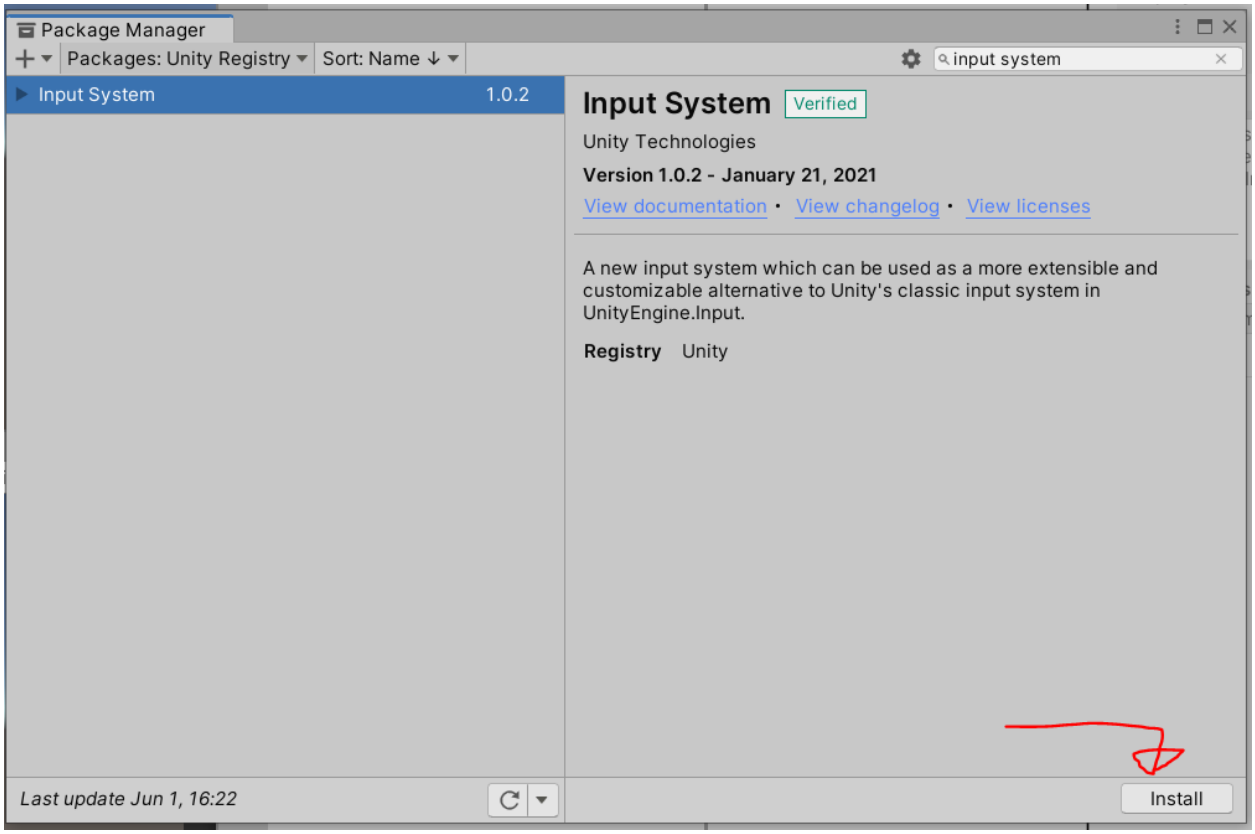
3. Switch the packages to Unity Registry (**Packages:...** -> **Unity Registry**):



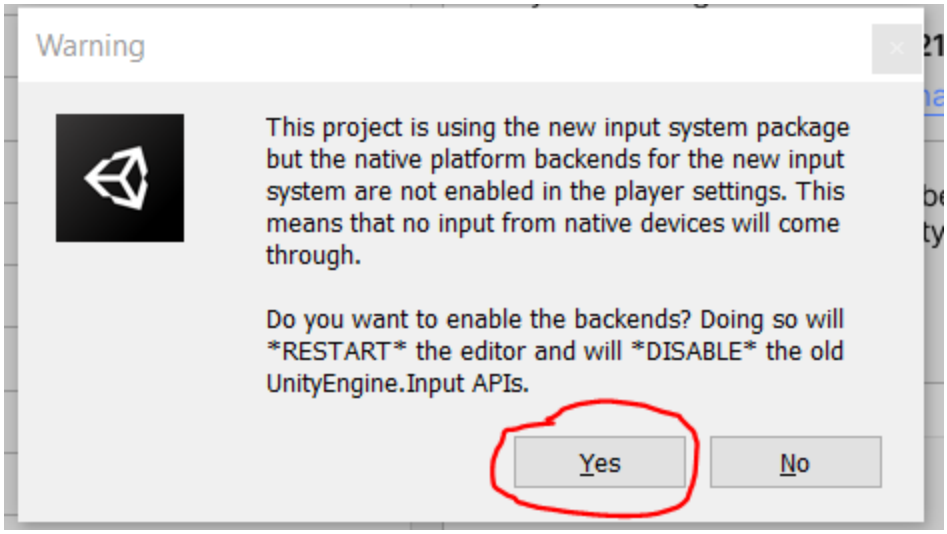
4. Search for **input system** through the **search bar** in the top right of the package manager window:



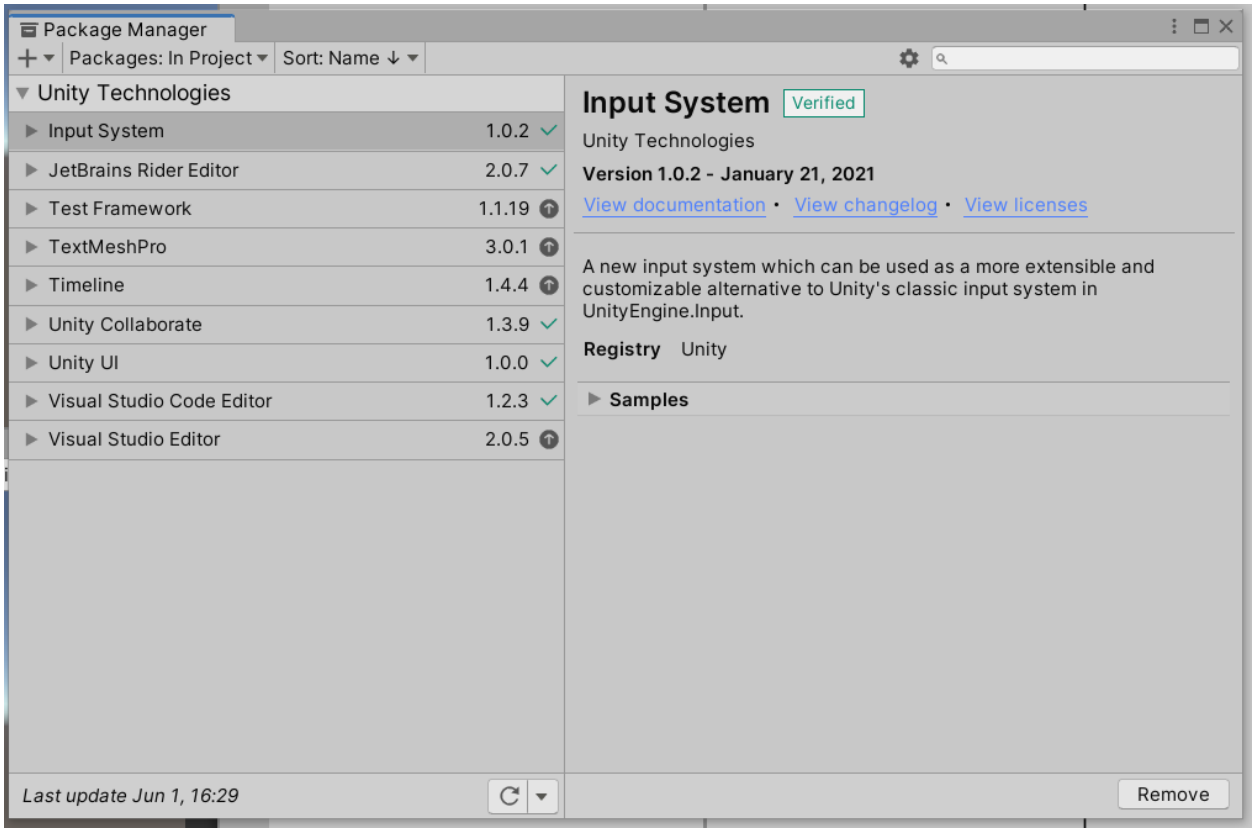
5. Click **install** in the bottom right of the package manager window:



6. It will take some time for the installation to finish and you will receive a warning about switching to the new Input system, enabling the needed backends, and disabling the old input code. Select **Yes** to allow the editor to switch to the new input system:

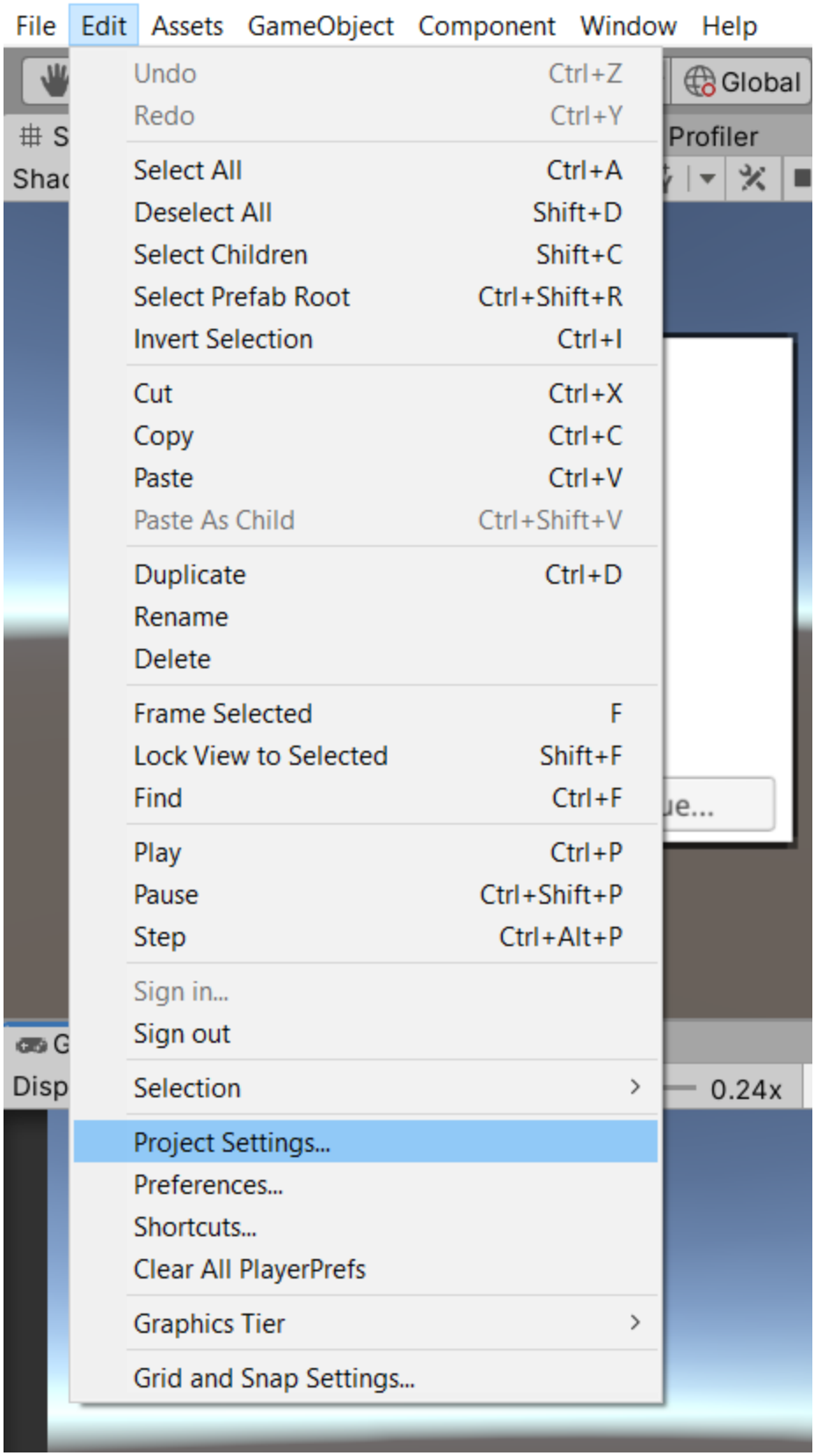


7. After Selecting **Yes**, unity will restart itself. This can take some time to finish.
8. Once the restart has finished you can verify that the new input system is installed by going to the Package Manager Window (**Window** -> **Package Manager**) and selecting **Packages: In Project** from the **Packages:** drop down:

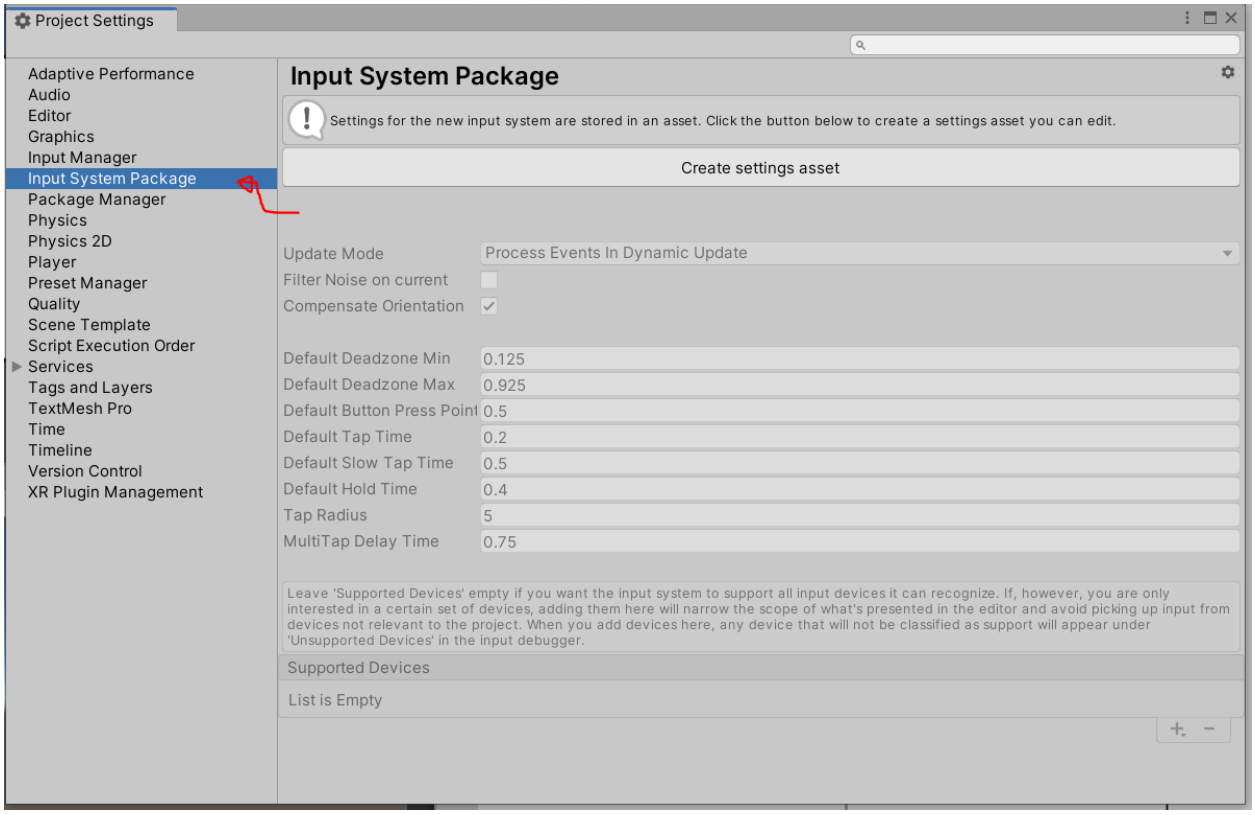


If you see the input system listed among the packages in your project it has been installed and needs just a little more setup to be used.

9. Go to **Edit** -> **Project Settings**

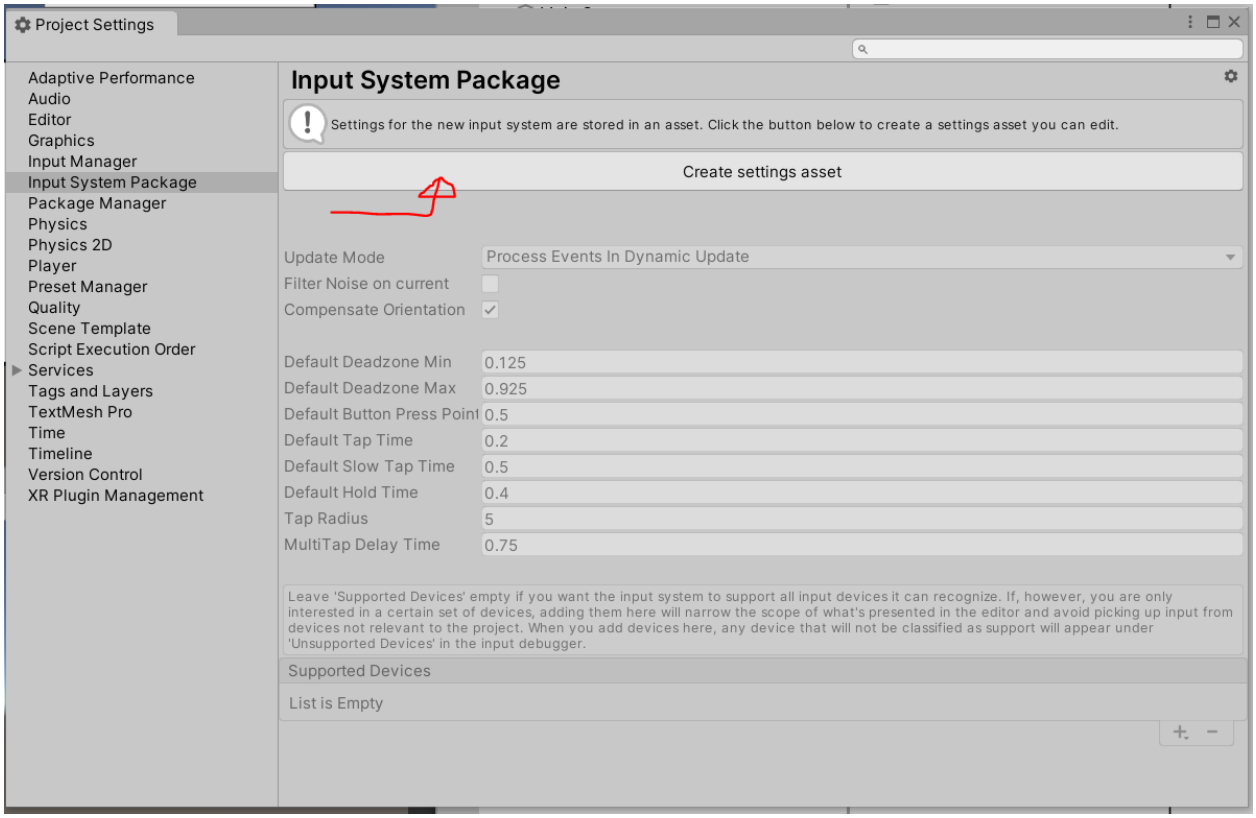


10. The Project Settings window should now be open. From here, select the **Input System Package** from the left side of the Project Settings window:



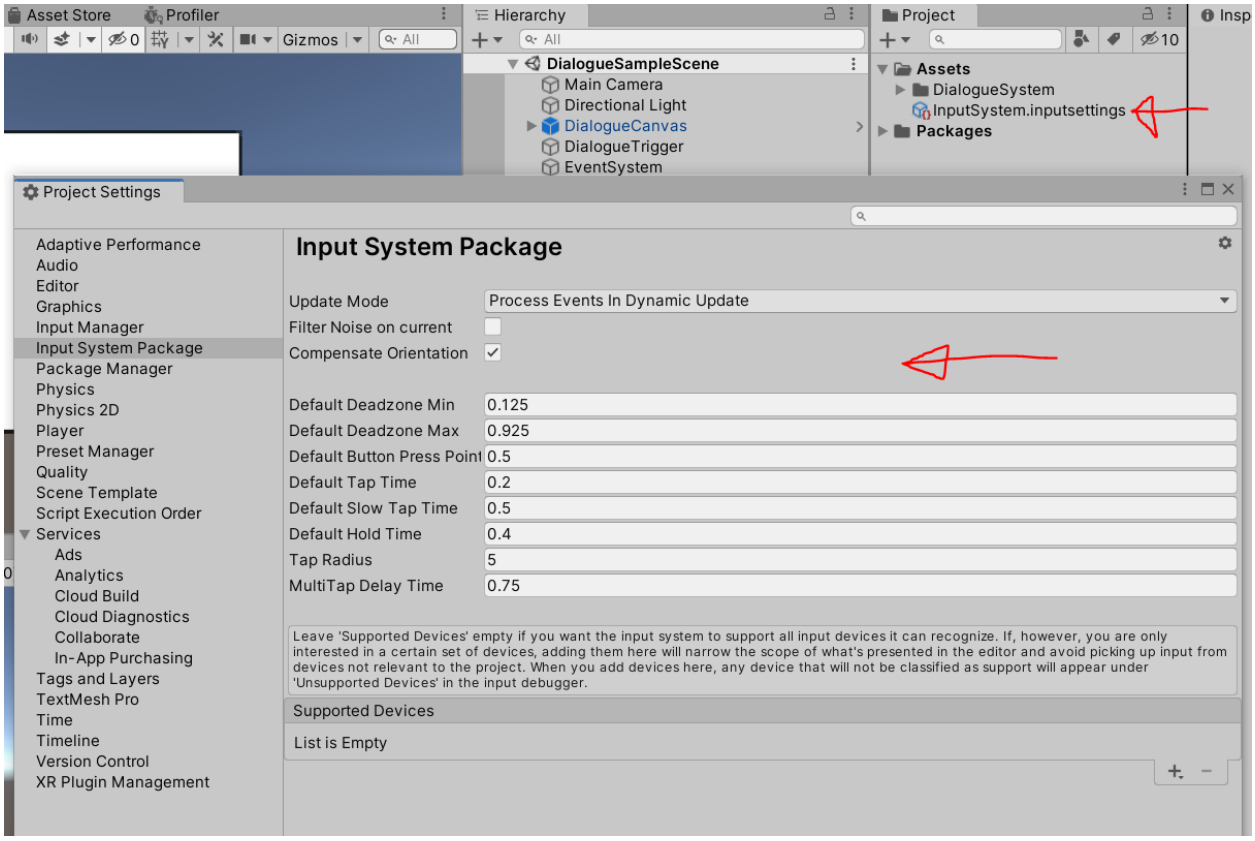
11. The input system requires a **Settings Asset** to allow you to modify its settings for things like default dead zones. On the right of the project settings window you should see the

Create settings asset button.



12. Click the **Create settings asset** button
13. A settings asset for the input system called "**InputSystem.inputsettings**" should now be in your assets folder and you should now be able to change the settings in the **Project**

Settings window:

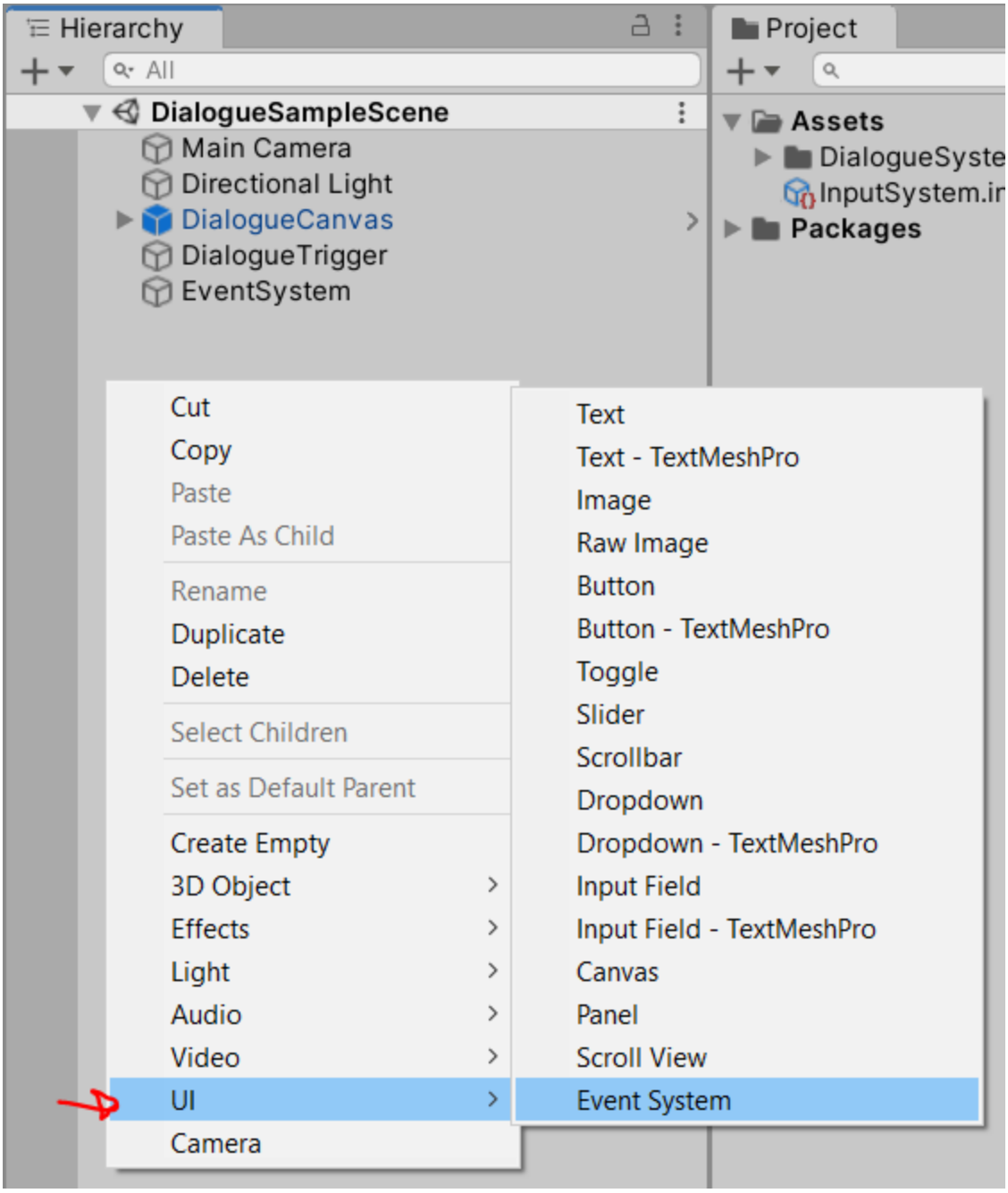


14. You are now ready to use the new input system in your project through several methods outlined later in this document.
    a. If working through Unity Collab, everyone must restart Unity after syncing to the update that contains the new input system in order for it to work for them locally.

2. Unity UI, the EventSystem, and the new Input System:

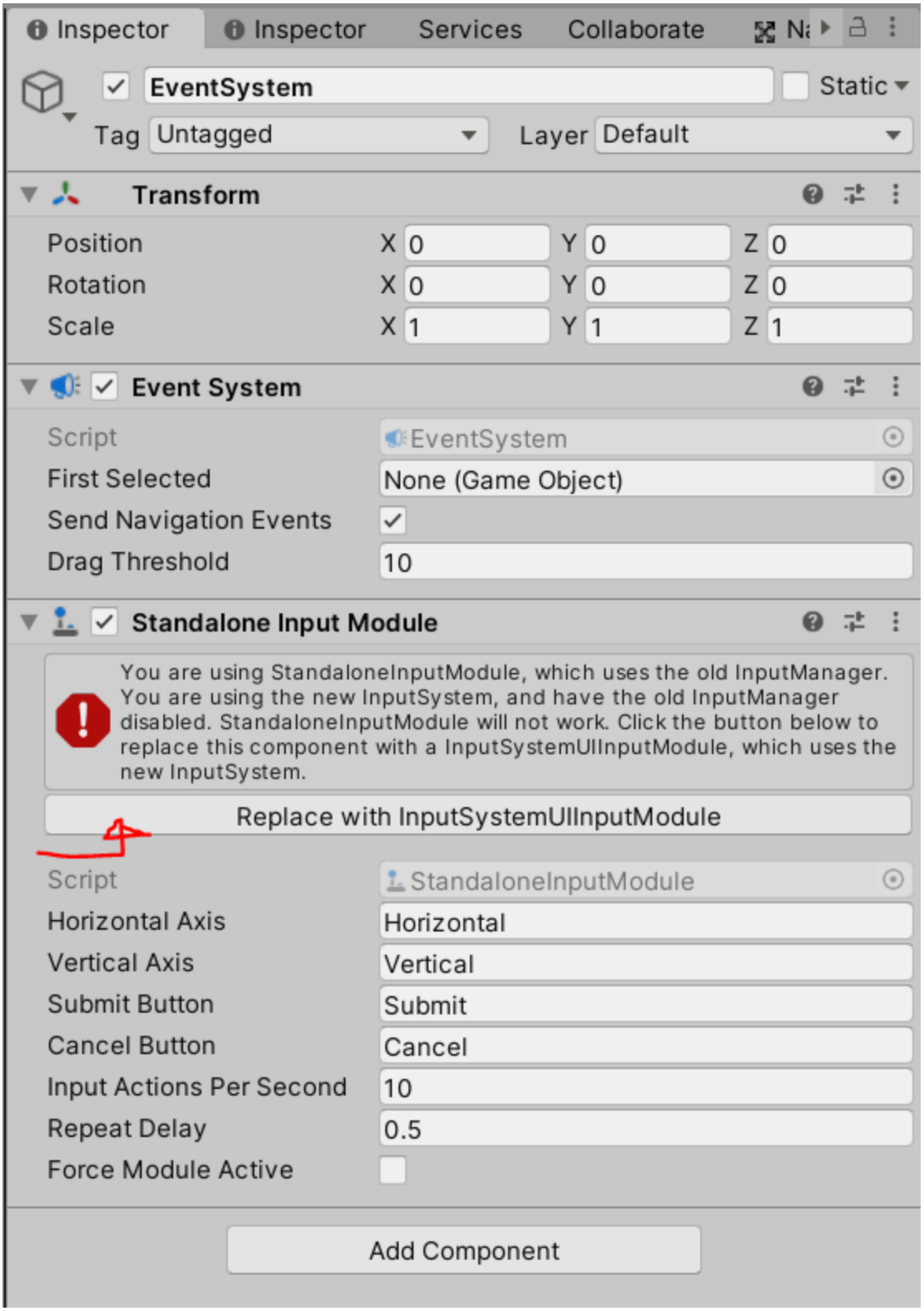
Any time you want to interact with Unity UI elements you need an **EventSystem**. This is true for the old Input System and the new Input System. With the new Input System installed, any time you create an **EventSystem** in a scene you will need to replace its **Standalone Input Module component** with the **Input System UI Input Module component**:

1. If you do not have an EventSystem in the scene you want UI to be intractable in, create one by **right clicking in the scene Hierarchy -> UI -> EventSystem**:

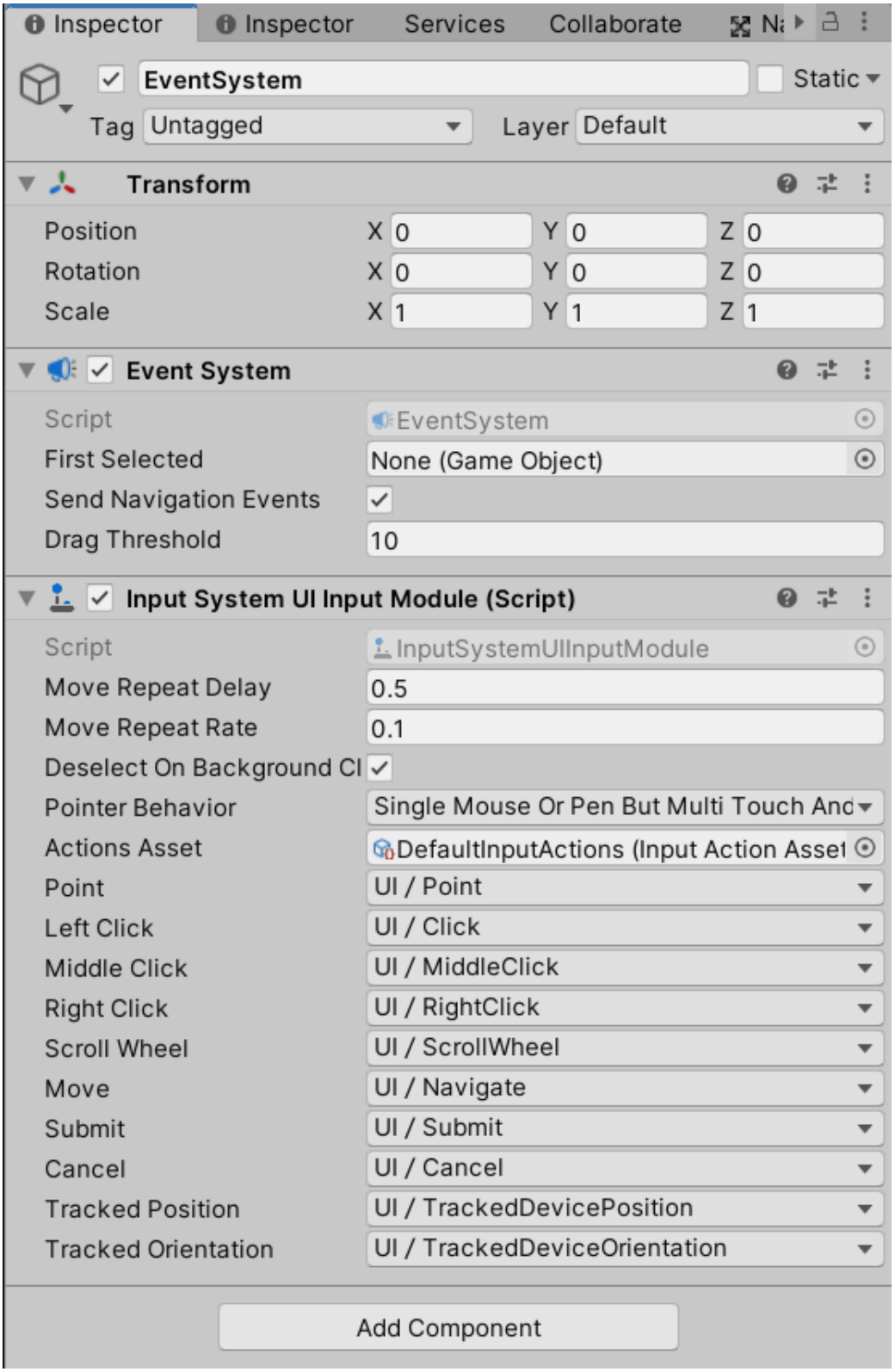


2. Once there is an **EventSystem** in your scene, **left click** on the **EventSystem** game object and select the **Replace with InputSystemUIInputModule button**

under the **Standalone Input Module component** in the inspector window:



3. Your **EventSystem** should now look like this in the Inspector window:



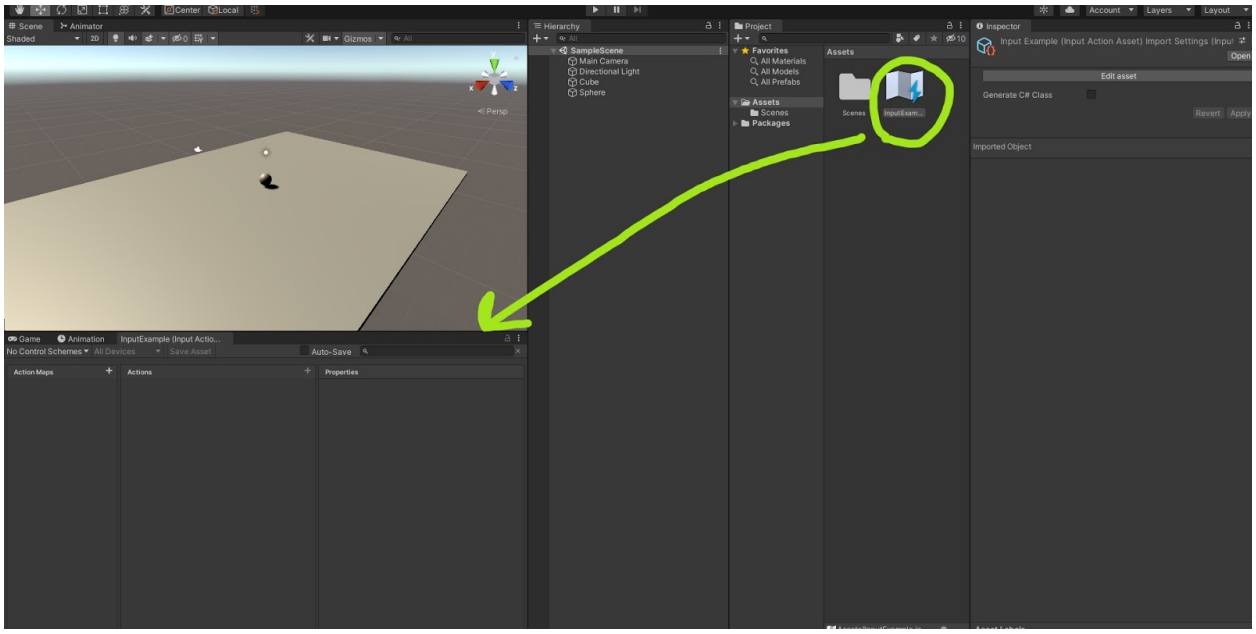
   a. Now your **EventSystem** will use the new Input System's default UI navigation controls.

b. You can change the defaults using your own input actions outlined later in this document, but the defaults will work for all mouse based UI navigation.

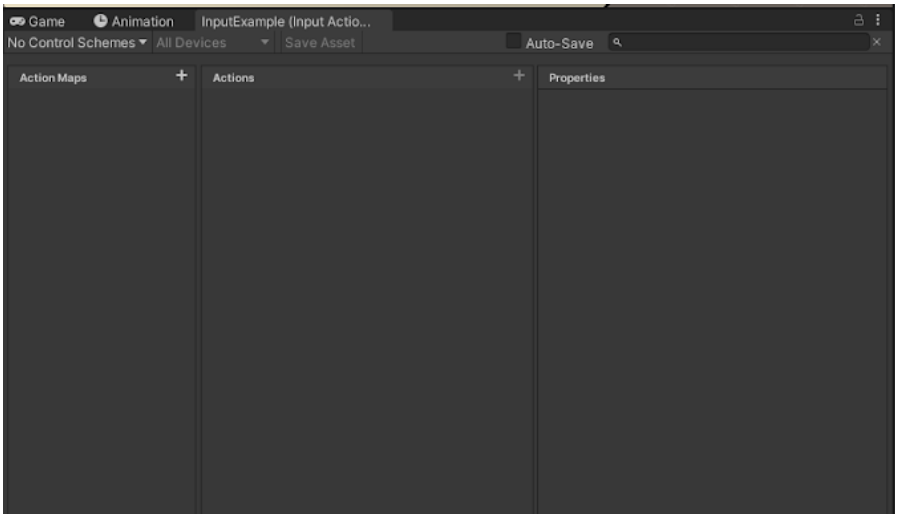
3. Using The Input System

3.1. The Input Actions Prefab:

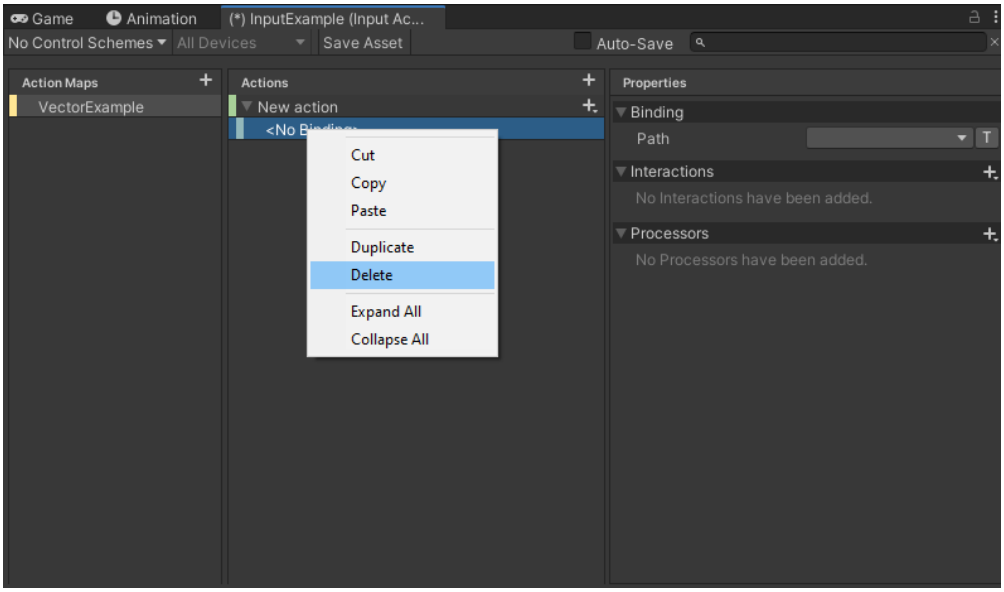
To create an **Input Action**, right-click in the Project window and select "**Input Actions**" (at the bottom of the options). Double-click on your new input action to open the Input window:



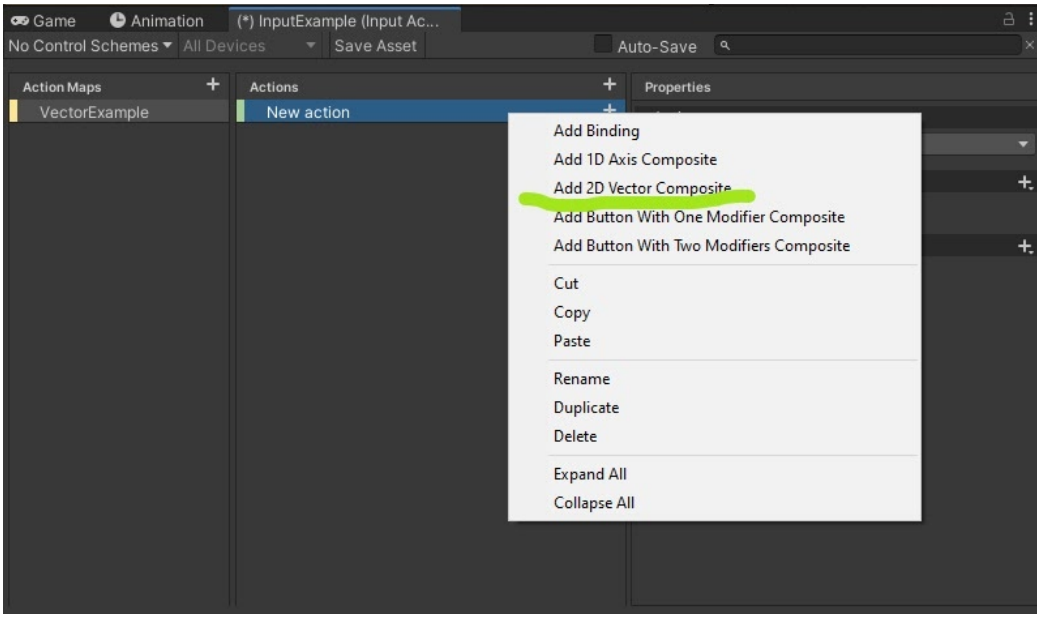
There are two main ways to use input actions: through the Input window and directly through scripts. To use the window, you will first need to **create a new map** by pressing the "+" next to **Action Maps**:



In the Action panel within the input window, select the “+” next to “**New Action**” to create a new binding. In this example, we will be creating a 2D vector composite as well as a single-button binding in order to show how to get input from both. To create a **2D vector composite**, delete the binding underneath the action (right click on it to delete it):

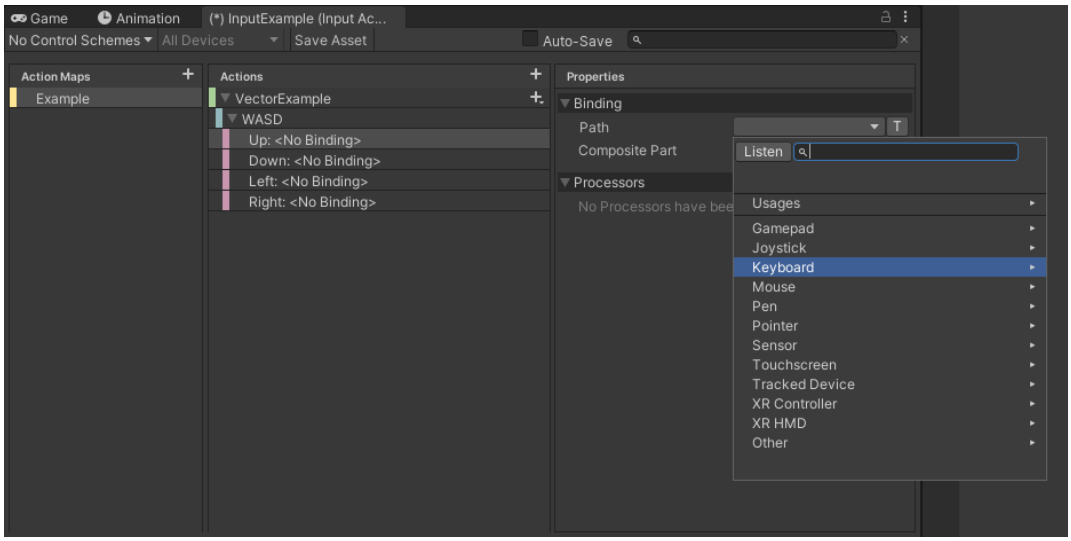


Then **right-click the action** to create the vector composite (click “Add 2D Vector Composite”):

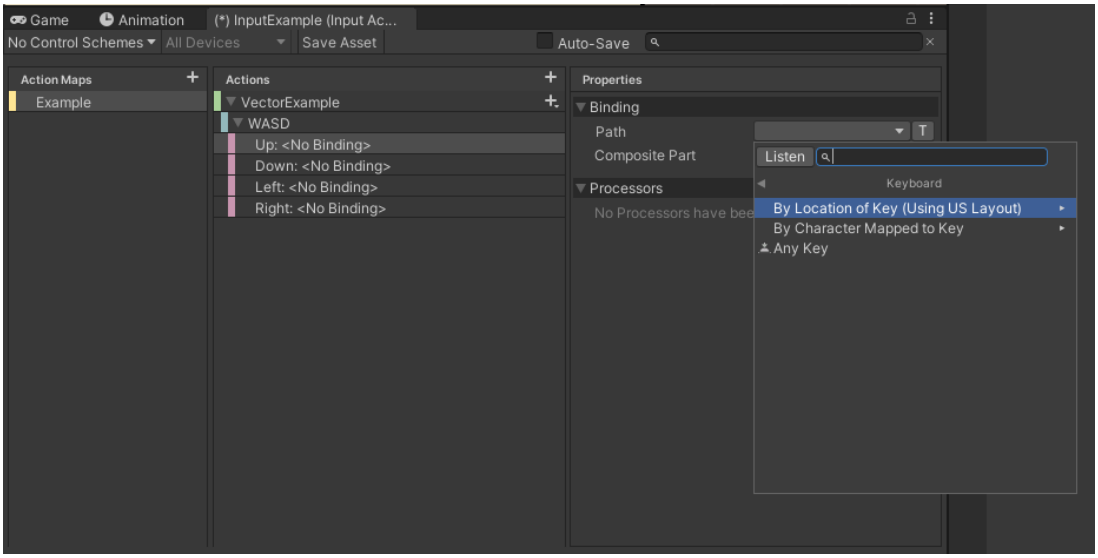


Now, you need to bind each direction to a key. We are going to use WASD for this example. To bind a key to a direction, select the direction, click the **drop-down** for **Path** (in the **Properties** panel of the input window), select **Keyboard**, and find the key that would correspond to the direction (i.e. W is up):

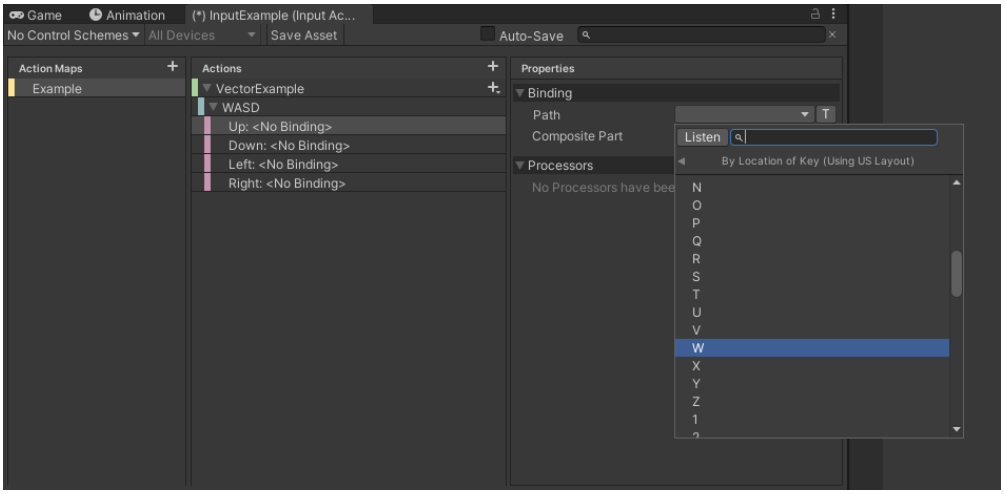
For this example, we know we want to use the Keyboard:



We also know we want a specific key, so we can find it through its location:

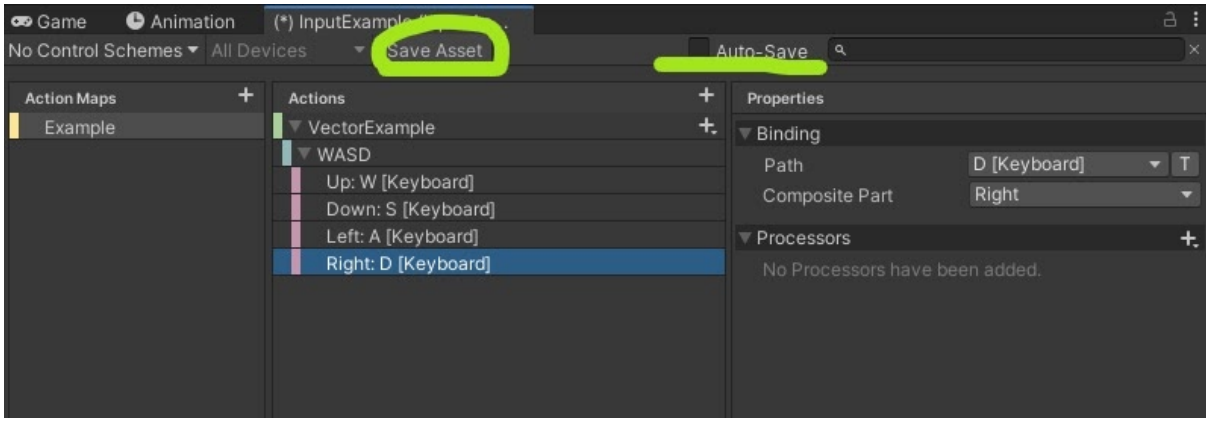


Now we can see a list of keys, so we can select whichever one we need (in this example, W is being selected):

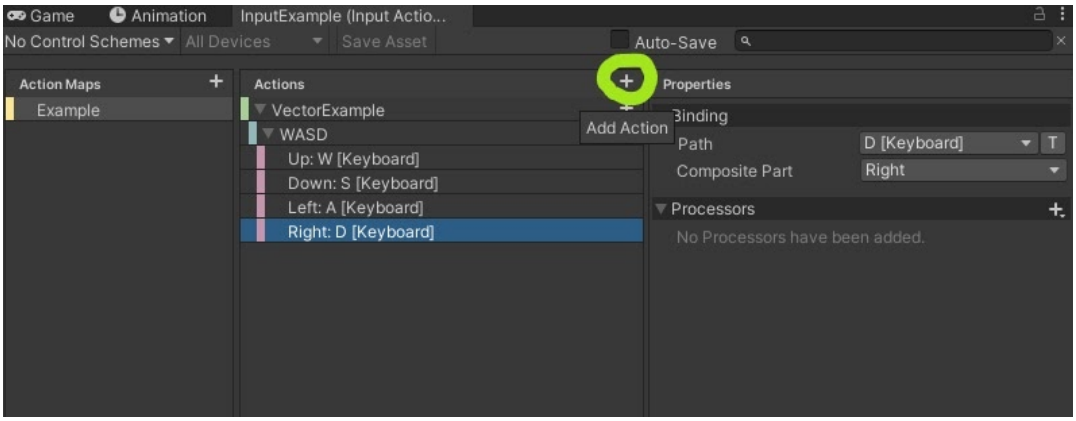


For more information on how to set bindings, go to Ways to Add Bindings.

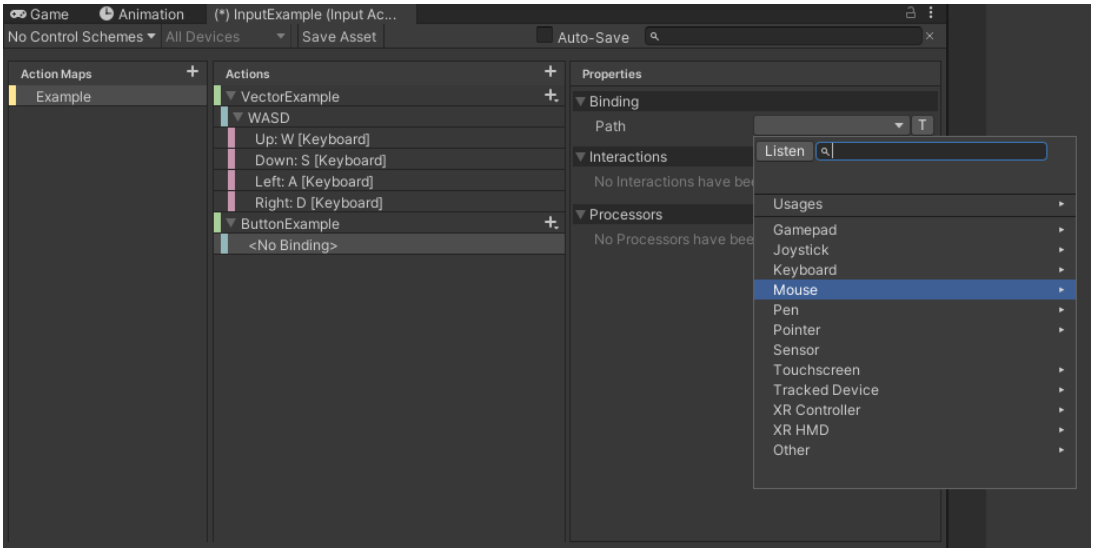
NOTE: Remember to click “Save Asset” in the Input window while you work! There is also an auto-save option, but it tends to work slowly, so saving the asset while you work is a more secure way of ensuring your work will be applied and won’t be lost!



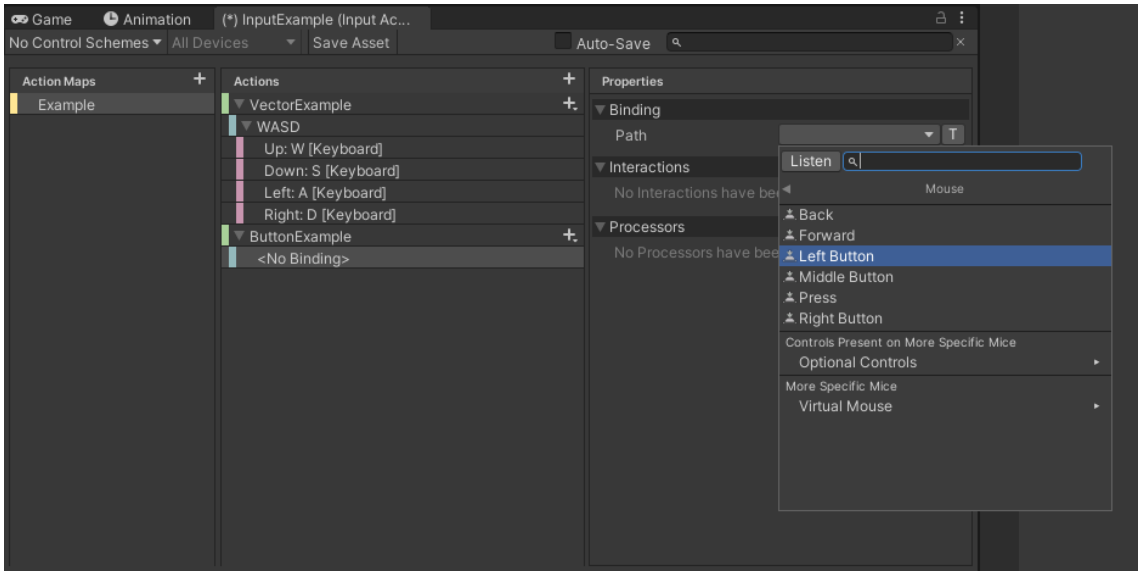
We are going to do an example using mouse input as well, so add another action to the action map. To do this, press the “+” next to **Actions**:



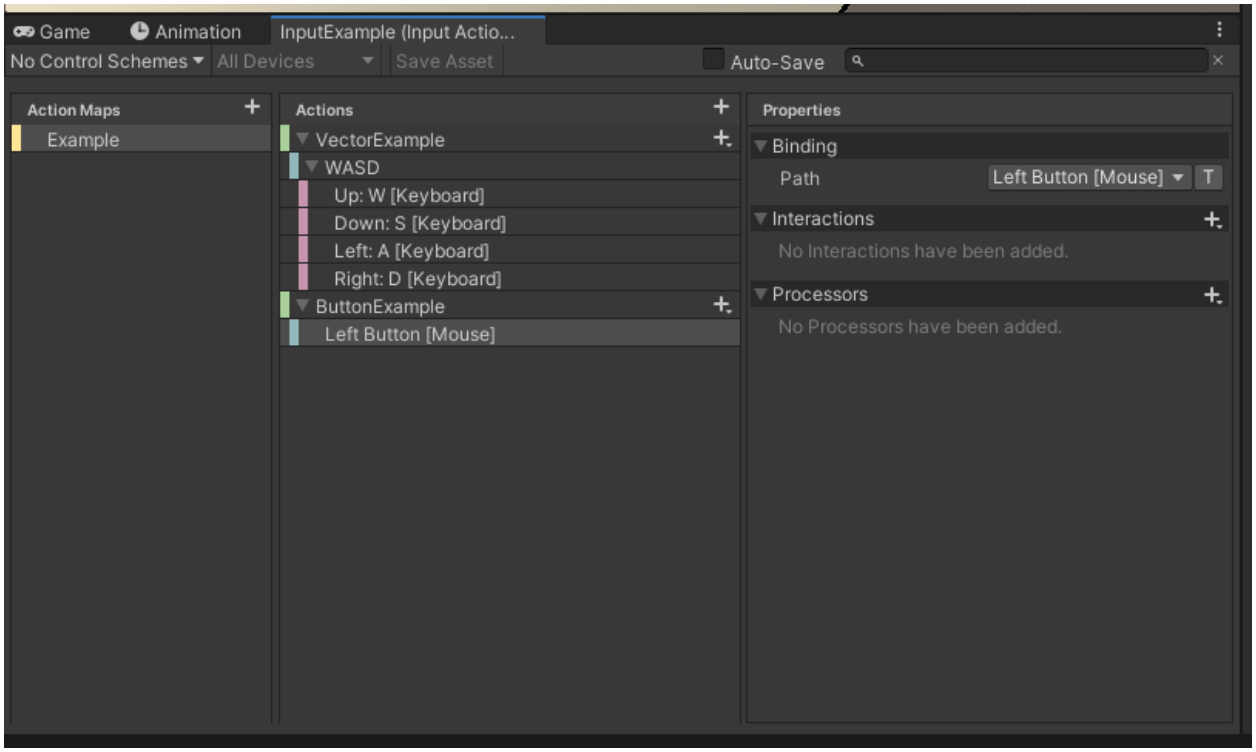
Rather than a 2D Vector Composite, just add a binding by pressing the “+” next to the action (or use the binding that’s already there, if it appeared automatically). The binding process will be similar: select the **path** (**Mouse**, in this case):



...and the **button** (**Left Button**, in this case):



The final result will look like this:



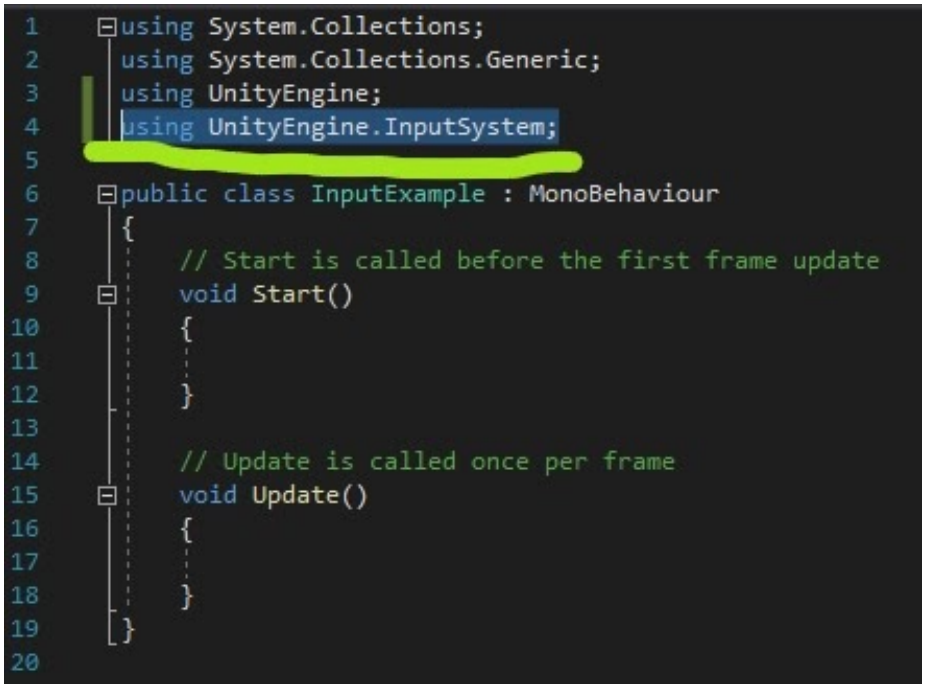
3.1.1 Ways to Add Bindings:

There are multiple ways to add bindings to an Input Action. Each is useful and quick, so which one you choose to utilize is up to you. The options are:

1. Search for the button you want
    a. After clicking on "**Path**", you can type the key, button, or other input you would like to use into the **search bar**. This will pull up any type of input from any device, so make sure your search entry is **as specific as possible**.
2. Click through the different categories of button
    a. After clicking on "**Path**", it will provide you with a list of input devices, such as **Keyboard**, **Mouse**, and **Gamepad**. You can select one of these input devices, and from there scroll through the possible inputs they have (i.e. keys on a keyboard).
3. "Listen" for button input
    a. After clicking on "**Path**", if you select the **Listen** button (next to the **search bar**), you can simply press the button you would like to read input from. The Input Action then **listens** for your input and brings up the option for whatever you pressed!

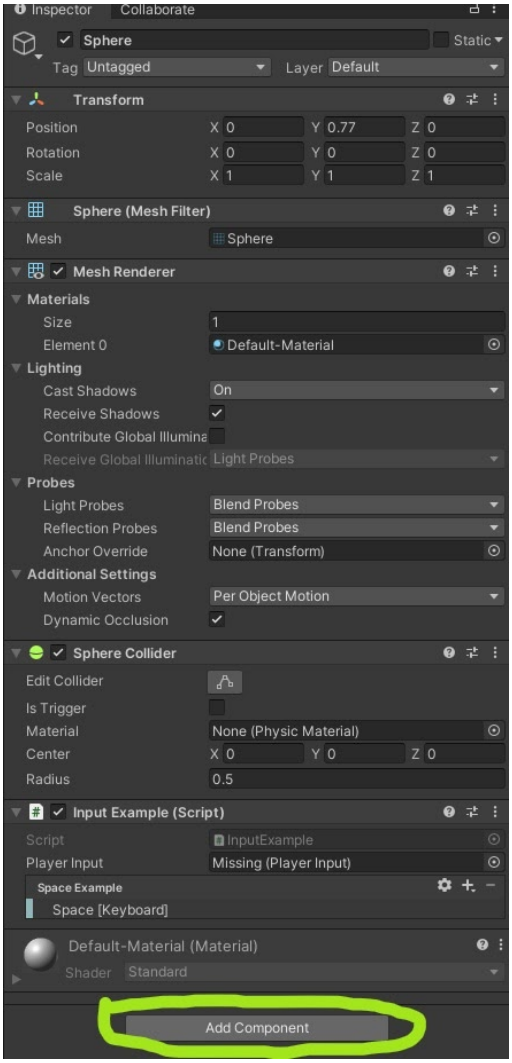
3.2 Connecting The Prefab to the Script:

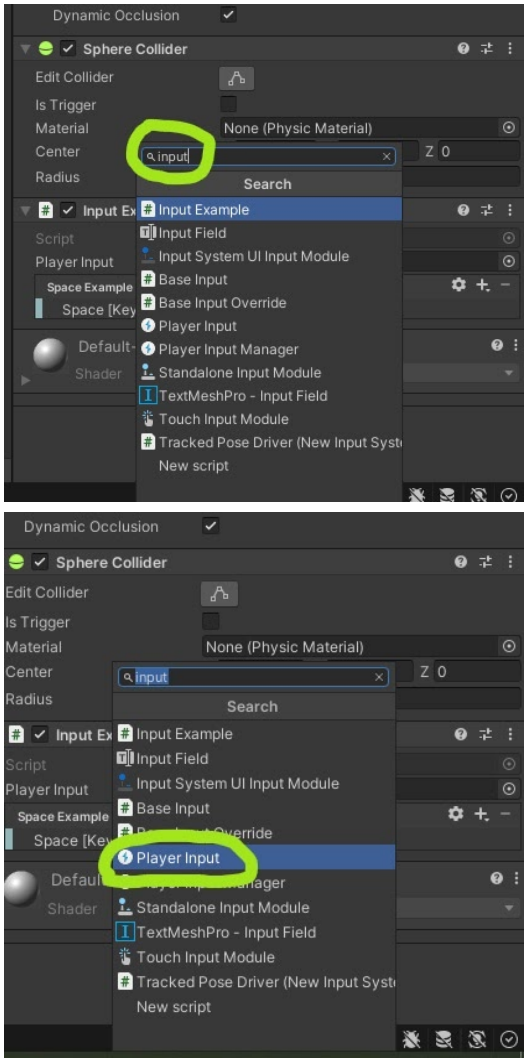
To connect the **Input Actions** to a script, you will first need to make sure you have access to the input system. In your script, add this line to the list of "using" statements:



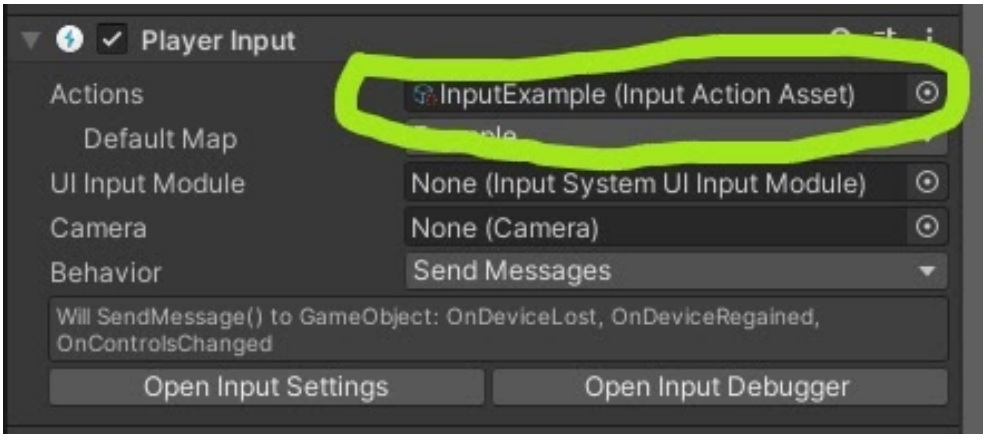
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

public class InputExample : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

You will also need to make sure the game object that will be using the input has a **Player Input component** attached:





Drag your Input Actions object into the Actions variable:



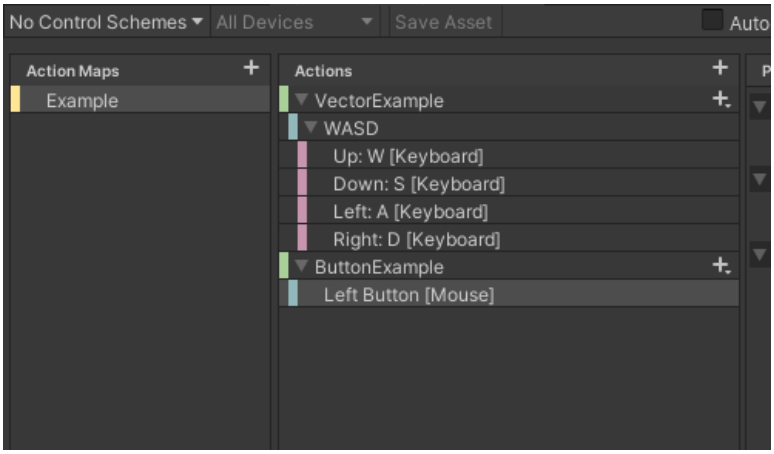
In your script, you can then reference this component by using **GetComponent**:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

public class InputExample : MonoBehaviour
{
    public PlayerInput playerInput;

    // Start is called before the first frame update
    void Start()
    {
        playerInput = GetComponent<PlayerInput>();
    }

```

This allows you to access the actions you created! To work with them, simply create functions that call them. **The naming structure is "On[ActionName]"** (for example, "OnJump" or "OnVectorExample"):

```
    // Update is called once per frame
    void Update()
    {

    }

    public void OnVectorExample()
    {
        Debug.Log("Vector keys connected to code");
    }

    public void OnButtonExample()
    {
        Debug.Log("Left mouse button connected to code");
    }
}

```

The **name of the action** and the **name of the function** need to match in order for them to be connected (here you can see the names of the actions):

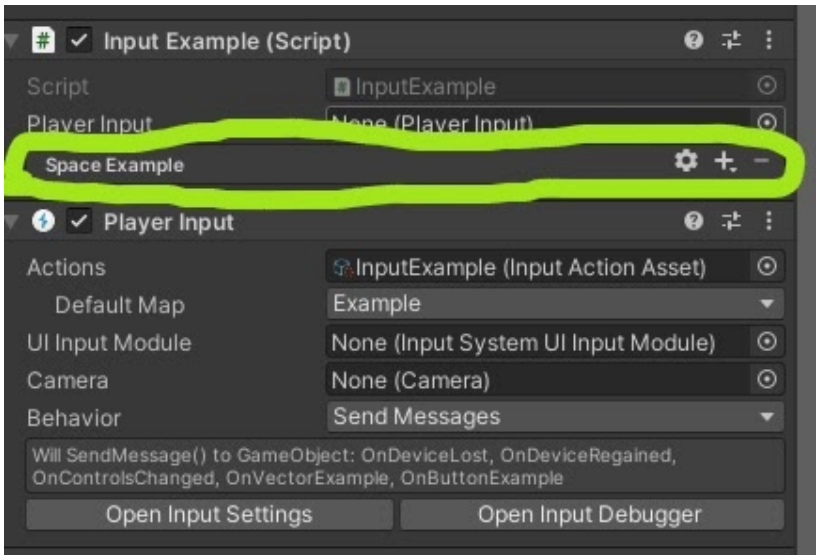


3.3 Using Input Actions Directly in Scripts:

You can also create **Input Actions** within your script, rather than through the **Player Input** component. To do this, create a public variable of type **InputAction** (the example below has an input action called “spaceExample”):

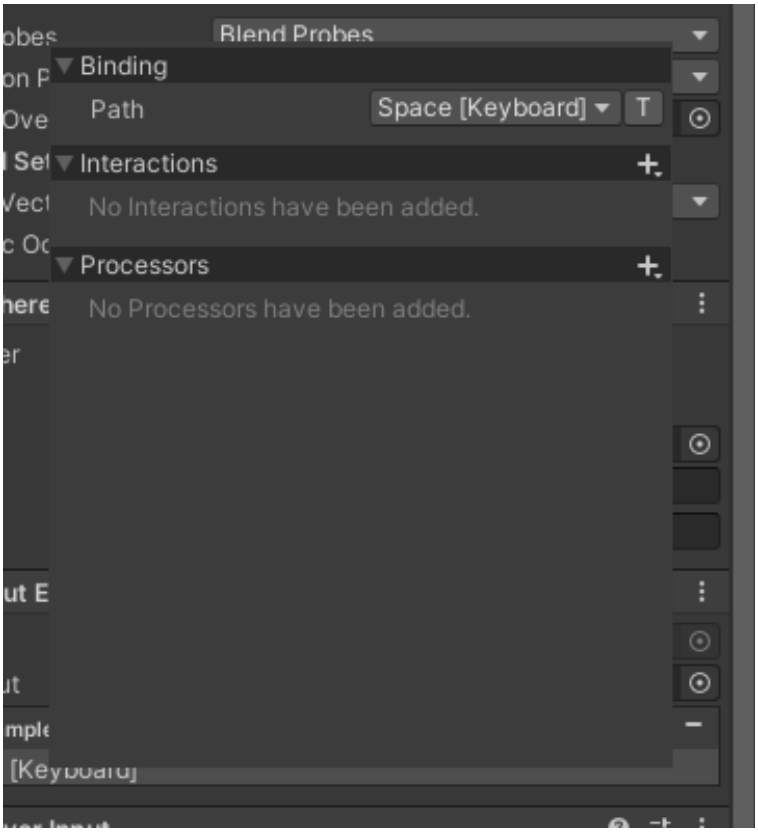
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

public class InputExample : MonoBehaviour
{
    public PlayerInput playerInput;

    public InputAction spaceExample;
```

In the inspector, you should see it come up:

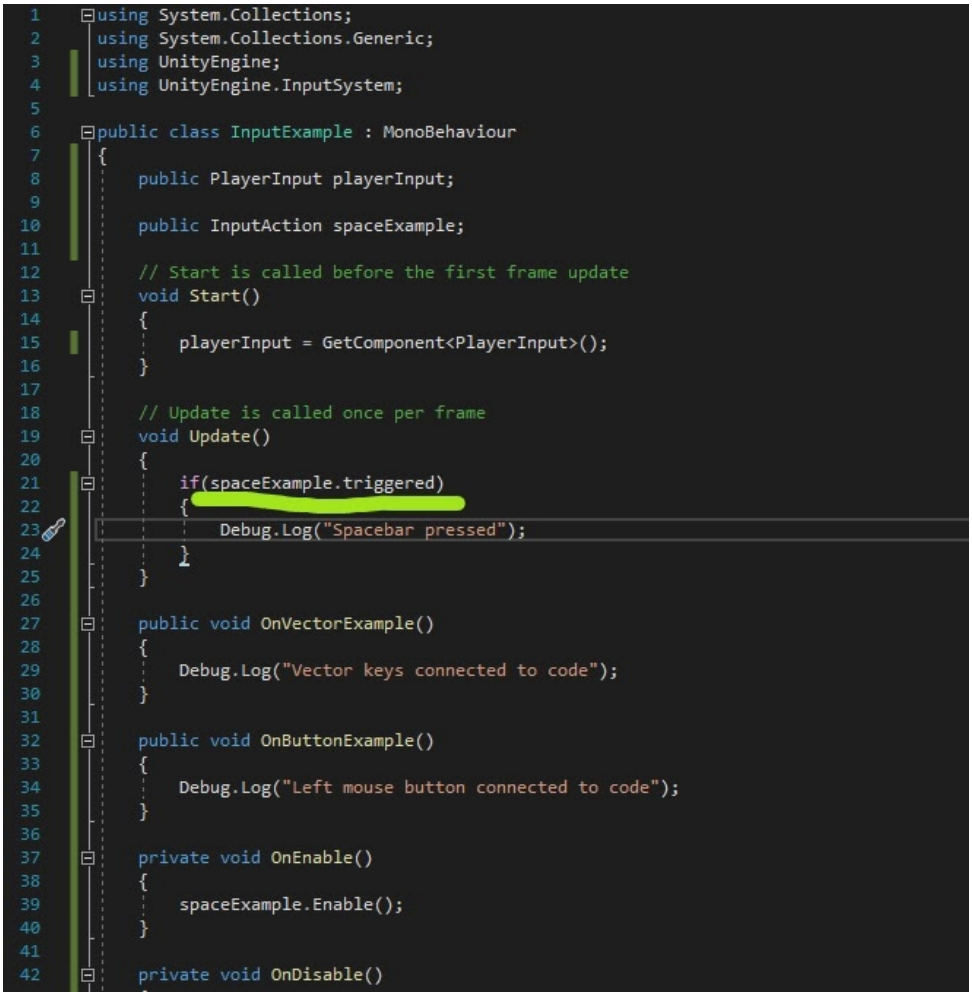


To add a binding, press the “+” on the bar where it says “Space Example” (or whatever the name of your variable is) and select “**Add Binding**”:

In this example, we will be setting the binding to the spacebar by selecting Keyboard and finding the Space option (for more detail, see Ways to Add Bindings):



To use this input, first make sure your **Input Action** is **enabled** (see Enabled/Disabled). Then you can simply check if the button was pressed by using "**.triggered**":



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

public class InputExample : MonoBehaviour
{
    public PlayerInput playerInput;

    public InputAction spaceExample;

    // Start is called before the first frame update
    void Start()
    {
        playerInput = GetComponent<PlayerInput>();
    }

    // Update is called once per frame
    void Update()
    {
        if(spaceExample.triggered)
        {
            Debug.Log("Spacebar pressed");
        }
    }

    public void OnVectorExample()
    {
        Debug.Log("Vector keys connected to code");
    }

    public void OnButtonExample()
    {
        Debug.Log("Left mouse button connected to code");
    }

    private void OnEnable()
    {
        spaceExample.Enable();
    }

    private void OnDisable()
```

This checks if the button has been pressed, thus triggering the action.

NOTE: Reading values from input this way can vary between types of input. For 2D Vector Composites, for example, you would do **[Vector Variable].ReadValue<Vector2D>()** and set it equal to a 2D Vector variable. You can then use these values to create movement vectors, apply the values you read in, etc. For more detail on ReadValue, reference the **Input Action Class** section on ReadValue.

3.4 Getting Input Directly in Scripts (Without Input Actions):

3.4.1 Keyboard.current:

Keyboard.current is used to read the current state/input from the keyboard. In other words, it checks what is happening with the keyboard the moment it is called.

There are multiple ways to use Keyboard.current, and there are many things that can be done with it. A very simple example would be to check if any **button has been pressed this frame**:

```
if (Keyboard.current.anyKey.wasPressedThisFrame)
{
    Debug.Log("A key was pressed");
}
```

This is a helpful way to check for any kind of input, rather than a specific input like with **Input Actions**.

3.4.2 Gamepad.current:

Gamepad.current is similar to **Keyboard.current**, but it checks for input from a gamepad rather than from a keyboard. Unity is able to read input from a variety of devices, from touchscreen to keyboard to gamepad.

Gamepad.current can be used very similarly to Keyboard.current. Here is that same simple example from before, but using **Gamepad** rather than Keyboard:

```
if (Keyboard.current.anyKey.wasPressedThisFrame)
{
    Debug.Log("A key was pressed");
}

if (Gamepad.current.aButton.wasPressedThisFrame)
{
    Debug.Log("A button was pressed");
}
```

4. Input Action Class

4.1 ReadValue:

ReadValue is used exactly how the name implies: to read the value of an input. It is called on an **Input Action**. Calling it looks like this:

```
    readVec = readValExample.ReadValue<Vector2>();
}
```

It is frequently used with **Vector Composite Input Actions**, though that is not the only way to use it. It can be used with the **mouse**, for example, to find the direction for the player to look:

```
// Update is called once per frame
void Update()
{
    var look = new Vector2();
    var mouse = Mouse.current;

    if (mouse != null)
        look = mouse.delta.ReadValue();
```

4.2 Get:

Get is similar to **ReadValue** in its functionality, but it differs in a few ways. While ReadValue is used on an InputAction, **Get** is used on an **InputValue** (often the parameter of a function):

```
public void OnVectorExample(InputValue value)
{
    Debug.Log("Vector keys connected to code");
    inVec = value.Get<Vector2>();
    Debug.Log("x: " + inVec.x + " y: " + inVec.y);
}
```

Similar to ReadValue, Get is used to obtain the value of an input, often to be used in a new way within the code.

4.3 Triggered:

Triggered is used to check whether an **Action** has been performed this frame. This is used in the Space Example in the **Using Input Actions Directly** section.

4.4 AddBinding:

Something that may prove very useful to you is the ability to add **bindings** to **Input Actions** within code. So, rather than setting the binding in the inspector, you can set it directly in your script:

```
    var action = new InputAction();
    action.AddBinding("<Keyboard>/Q");
}
```

This also gives you the ability to add multiple bindings to the same action. For example, you could have movement controlled by both WASD and the arrow keys!

4.5 Action Map:

You can access the **Action Map** to which an **Input Action** belongs directly within your code. This gives you multiple capabilities, such as adding a new Action to an Action Map:

```
    var actionMap = new InputActionMap();
    var action2 = actionMap.AddAction("action");
}
```

For example, if you wanted to add a Crouch action to your player, you could simply add the action within your script.

4.6 Enabled/Disabled:

Input Actions have attributes to check whether or not they have been **enabled**. In other words, these attributes check to see if the Action currently responds to input. To set these, you would create functions for **OnEnable** and **OnDisable**:

```
62
63      private void OnEnable()
64      {
65          spaceExample.Enable();
66      }
67
68      private void OnDisable()
69      {
70          spaceExample.Disable();
71      }
72
73  }
74
```

4.7 Events:

Input Actions utilizes events that are called depending on the phase of the input. The Different phases are as follows.

- Started
- Performed
- Canceled

The order these events are called are standard throughout each press of a button. They tend to follow the Started-Performed-Canceled model.
When a button is **pressed**, its phase is **Started.** The next phase, **Performed**, follows immediately afterwards. Then when the button is **released** its phase is now **Canceled.**

Lets see how each of these events work.

```
9     public InputAction myInputAction;
10
      Unity Message | 0 references
11    private void Start()
12    {
13        myInputAction.started += context => MyStartedEvent(context);
14        myInputAction.performed += context => MyPerformedEvent(context);
15        myInputAction.canceled += context => MyCanceledEvent(context);
16    }
17
      1 reference
18    private void MyCanceledEvent(InputAction.CallbackContext context)
19    {
20        //Input Action is Canceled
21    }
22
      1 reference
23    private void MyPerformedEvent(InputAction.CallbackContext context)
24    {
25        //Input Action is Performed
26    }
27
      1 reference
28    private void MyStartedEvent(InputAction.CallbackContext context)
29    {
30        //Input Action is Started
31    }
32
```

Each of these events have the same setup process.
First you start by assigning function(s) to the started/performed/canceled calls

```
13        myInputAction.started += context => MyStartedEvent(context);
```

To add a function you use the += operator and the => (lambda) operator.
The **context** in this case is of the type **InputAction.CallbackContext.** This is the context or the reference of the InputAction that triggered the event.
The **MyStartedEvent()** is the function that is subscribed to the event.
You can add as many functions as you want with the += operator.

Now everytime the Input Action is in one of these phases the subscribed function will be called. In other words. If an action is pressed it will call the function subscribed to **started** and **performed**. If an action is released it will call the function subscribed to **canceled.**

4.8 Other Functionality:

Input Actions have many capabilities. The core elements of Input Actions have been included in this documentation, but there are many, many ways to utilize input within your game.