Description

InControl is an input manager for Unity3D (version 4+) that tames the cross-platform controller beast.

Features

- Standardizes input mappings across various platforms.
- Support for 10 connected devices with up to 20 analogs and 20 buttons each.
- Trivial to support new devices and platforms.
- Events for attached and detached devices.
- Events for active device switches.

Supported Controllers

- Xbox 360 controller support for Windows, Mac and OUYA.
- Playstation 3 controller support for Windows, Mac and OUYA.
- Playstation 4 controller support for Windows, Mac and Linux.
- OUYA controller support on OUYA and Windows.
- GameStick support.
- Keyboard and Mouse support on Windows, Mac and Linux.

Note: New device profiles are simple to create. Please feel free to submit profiles for any controller/platform not currently in the list, but do ensure it correctly supports all the standardized inputs (see below).

Standardized Inputs

Device profiles map supported controllers on various platforms to a strict set of named inputs that can be relied upon to be present. Physical positions (particularly for action buttons) will match across devices for uniformity.

- `LeftStickX`, `LeftStickY`, `LeftStickButton`
- `RightStickX`, `RightStickY`, `RightStickButton`
- `DPadUp`, `DPadDown`, `DPadLeft`, `DPadRight`
- `Action1`, `Action2`, `Action3`, `Action4`
- `LeftTrigger`, `RightTrigger`
- `LeftBumper`, `RightBumper`

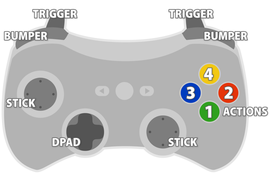

Illustration: Standardized Inputs

Note: the API makes little distinction between analog and button controls, so both a `float` value and `bool` state can be queried for any input.

Unsupported devices can be used, however their default mappings are utterly unpredictable. From the API, inputs for unsupported devices will appear as `Button0` thru `Button19` and `Analog0` thru `Analog9`. Do with them what you will.

Getting Started

First, generate a new `ProjectSettings/InputManager.asset` through the editor menu: `Edit > Project Settings > InControl > Generate InputManager Asset`

Next, create an empty GameObject and the script below attached to it.

The project is namespaced under `InControl`. The entry point is the `InputManager` class. You'll need to call `InputManager.Setup()` once and `InputManager.Update()` every tick (or whenever you wish to poll for new input state).

```
using UnityEngine;
using InControl;

public class UpdateInputManager : MonoBehaviour
{
	void Start()
	{
		InputManager.Setup();
	}

	void Update()
	{
		InputManager.Update();
	}
}
```

Note: It is a good idea to alter the execution order of the script responsible for calling `InputManager.Update()` so that every other object which queries the input state gets a consistent value for the duration of the frame, otherwise the update may be called mid-frame and some objects will get the input state from the previous frame while others get the state for the current frame.

By default, InControl reports the Y-axis as positive pointing up to match Unity. You can invert this behavior if you wish:

```
InputManager.InvertYAxis = true;
InputManager.Setup();
```

Now that you have everything set up, you can query for devices and controls. The active device is the device that last received input.

```
InputDevice device = InputManager.ActiveDevice;
InputControl control = device.GetControl( InputControlType.Action1 )
```

Query an indexed device when multiple devices are present like so:

```
var player1 = InputManager.Devices[0];
```

Given a control, there are several properties to query:

```
control.IsPressed;   // bool, is currently pressed
control.WasPressed;  // bool, pressed since previous tick
control.WasReleased; // bool, released since previous tick
control.HasChanged;  // bool, has changed since previous tick
control.State;       // bool, is currently pressed (same as IsPressed)
control.Value;       // float, in range -1..1 for axes, 0..1 for buttons / tr
control.LastState;   // bool, previous tick state
control.LastValue;   // float, previous tick value
```

Controls also implement implicit conversion operators for `bool` and `float` which allows for slightly simpler syntax:

```
if (InputManager.ActiveDevice.GetControl( InputControlType.Action3 ))
{
	player.Boost();
}
```

The `InputDevice` class provides handy shortcut properties to the standardized inputs:

```
if (InputManager.ActiveDevice.Action1.WasPressed)
{
	player.Jump();
}
```

It also provides four properties that each return a directional `Vector2`:

```
Vector2 lsv = device.LeftStickVector;
Vector2 rsv = device.RightStickVector;
Vector2 dpv = device.DPadVector;
Vector2 dir = device.Direction;
```

The fourth, `Direction`, is a combination of the D-Pad and Left Stick, where the D-Pad takes precedence. That is, if there is any input on the D-Pad, the Left Stick will be ignored.

Finally, you can subscribe to events to be notified when the active device changes, or devices are attached/detached:

```
InputManager.OnDeviceAttached += inputDevice => Debug.Log( "Attached: " + inp
InputManager.OnDeviceDetached += inputDevice => Debug.Log( "Detached: " + inp
InputManager.OnActiveDeviceChanged += inputDevice => Debug.Log( "Switched: "
```

To-do List

- Support Apple MFi controllers on Mac and iOS.
- Support Android controllers like the Moga Pro.
- Support more controllers on Linux.

Known Issues

- Not all platforms trigger the `DeviceAttached` event correctly. If Unity's `Input.GetJoystickNames()` is updated by the platform while the app is running, it will work. Every platform does, however, report all newly connected devices once the app is relaunched.
- Some controller specific buttons (like Start, Select, Back, OUYA, Xbox Guide, PS3, etc.) are not part of the standardized set of supported inputs simply because they do not work on every platform. You should not be using these buttons in a generalized cross-platform capacity.

Meta

Handcrafted by Patrick Hogan [twitter • website]