Multi-Input

Version 0.12.2b

INTRODUCTION

This is the user manual for the Multi-Input library, which allows game developers using Unity to handle multiple input devices of the same type (keyboards, mice, gamepads) without losing the distinction between them.

Multi-Input is currently in beta, offering core multi-device API on Windows, Linux and OSX/macOS (see prerequisites section below for more details on platform support).

FEATURES

The bulk of the library interaction will be through two things: a device object, and its virtual axes. We're focused on providing a solid core design first, and bells and whistles second.

All the necessary platform-specific code to get the input from various devices is handled by the library. The exact way in which you use this information, including all UI considerations, is left to you. This enables you to create games that easily support local multiplayer using a combination of keyboards, mice and gamepads, and guarantees that the library won't be getting in your way (as there are no high-level design decisions you might need to work around or adapt your game to).

Over time we'll extend this library with more helpers, and perhaps a higher-level API that takes care of more things for you, but this simple core will always be there for developers who feel our high-level choices are not good fit for their games.

Currently the library lets you:

- Query input devices connected to the user's system.
- React to changes in devices' state.
- Query axes and buttons on any device through a simple, uniform interface.
- Support Xbox-compatible controllers (up to 4 on Windows, more on other platforms).
- Support multiple keyboards and mice.

FUTURE PLANS

These are our plans for future development of Multi-Input, listed in no particular order. Feel free to contact us if you have any wishes, or are interested in seeing one of these features appear sooner.

- More platforms: perhaps mobile and/or consoles.
- More input sources and extended device support: DirectInput, absolute pointing devices, non-Xbox controllers etc.
- Unity GUI integration. We're also working on our own UI system which will support multiple focus and so will be a great add-on for local multiplayer games using this library.
- Robust higher-level helper APIs (e.g. serializing bindings, implementing device assignment, automatic reconnecting).
- Extension API (writing custom sources and device objects).
- Visual input scheme configuration.

WHAT'S NEW IN 0.12.2

0.12.2 release (2018-08-23) is a maintenance release. There are no new features.

Changes in this version:

- Integration: updated the asset for Unity 2018.2.0.

WHAT'S NEW IN 0.12.1

0.12.1 release (2017-05-18) is a bugfix release. There are no new features or changes to the API.

Changes in this version:

- Bug: fixed an issue preventing non-HID devices (e.g. PS/2 keyboards and mice) from being enumerated properly on Windows.

WHAT'S NEW IN 0.12

0.12 release (2017-03-20) is our fourth feature release, focused on device identification. There are no backwards-incompatible changes to the API.

Changes in this version:

- Feature: added new device properties: `IDevice.Location`, `IDevice.InternalName`, `IDevice.ProductId`, `IDevice.VendorId`, `IDevice.Revision`, and `IDevice.Serial`. These should allow developers to more accurately identify the device, as `IDevice.Id` is only meant to distinguish devices within one session. In future, the new properties will be the basis of an automatic reconnection logic.
- Bug: fixed the "failed to get the size for UTF-8 buffer" exception being thrown in the native code on Windows.

UPGRADE NOTES (0.11 TO 0.12)

No action beyond updating the asset is necessary.

WHAT'S NEW IN 0.11

0.11 release (2016-11-15) is our third feature release, focused on OSX/macOS support. There are no backwards-incompatible changes to the API.

Changes in this version:

- Introduction of OSX support.

UPGRADE NOTES (0.10 TO 0.11)

No action beyond updating the asset is necessary.

WHAT'S NEW IN 0.10

0.10 release (2016-06-08) was our second release, focused on internal cleanup and Linux support. **There were breaking changes to the API and library structure, see upgrade notes below.** The API should be much more stable now, but we can't promise that expanding platform support won't require further breaking changes.

Changes in this version:

- Introduction of Linux support.
- Fixes to Windows 32-bit support.
- Introduction of new vibration API with duration support.
- Increased reliability of the native component.
- Reduction of the amount of state that resides in the Unity bridge. The native component is now responsible for vast majority of the state. As such, abstract device classes `BaseDevice`, `BaseKeyboardDevice`, `BaseMouseDevice` and `BaseGamepadDevice` and `IDevice.PostCommit` and `PreCommit` methods have been removed.
- Reduction of the assemblies used by the library. All platform-specific code now resides in the native component, and so only one managed assembly is needed
- `InputState` lifetime management to help make it outlive the device objects.
- Editor support for wiring `InputState` events and setting log level.
- Removal of unreliable device properties `IDevice.IsAttached` and `IDevice.Type`.
- Removal of `DeviceId` type.
- Move to `IVirtualAxis` interface in lieu of using concrete `VirtualAxis` class. `IVirtualAxis.Code` is also no longer nullable.
- Unification of project branding and namespaces. Any mentions of the internal “libinput” codename should now be replaced by “Multi-Input”. The primary managed namespace for the library is now `RavingBots.MultiInput`.

UPGRADE NOTES (0.9 TO 0.10)

Remove all `RavingBots.InputLib.*` files from your `Assets/Plugins/` directory. They are no longer included or used by the library. You will need to restart your editor because Unity cannot unload native libraries.

After importing the new library, any `InputState` components on the scene and in prefabs will need to be replaced by the new one. Simply remove and re-add it after taking care of old library files.

Devices no longer have `IsAttached` property, which was both unreliable and redundant. Detached devices will become unusable and then subsequently removed from memory by the library. Where devices actually are reported by the system as reattached (currently only controllers on Windows), they will simply become usable again. Simply replace any checks of this property with checks for `IsUsable`.

Devices no longer have `Type` property (the related `DeviceType` enumeration has been removed as well). Even when we can get the device type, it's extremely unreliable and shouldn't be used to make any decisions. Instead you can use `InputCodeExt.IsMouse`, `InputCodeExt.IsKeyboard` and `InputCodeExt.IsGamepad` extension methods to classify the input as you get it. Look at the included example projects to see how it can be done.

`IDevice` indexer and `FindFirst` methods now return and take `IVirtualAxis` instead of `VirtualAxis` (internally `VirtualAxis` is no longer instantiated by the library: a separate hidden type is used instead). You should update any axis variables, fields and parameters to be of `IVirtualAxis` type. You can continue to use `VirtualAxis` instances for your own axes.

`IVirtualAxis.Code` (and by extension `VirtualAxis.Code`) is no longer nullable. Replace `InputCode?` with just `InputCode`, and replace null values with `InputCode.None`.

`BaseDevice` and related classes have been removed. If you previously relied on the fact that `IDevice` objects can be cast to them, you will need to modify that code to use the interface (and the axis indexer instead of fields that were exposed by these classes) instead.

All types now live in `RavingBots.MultiInput` namespace. You can remove all the other imports.

Public `Logger` class has been removed. You should set `InputState.LogLevel` property instead of calling `Logger` methods.

`IDevice.Vibrate` method got a new duration parameter. You will need to update your usages.

`InputState.DeviceStateChanged` is now an `UnityEvent` instead of C# event. You will need to use its `AddListener` method instead of `+=` operator to register new listeners (and `RemoveListener` instead of `-=` to unregister them). Additionally, it now receives two arguments: device affected and the type of the change. You might also need to register for `InputState.DevicesEnumerated` event to know when you can read the `InputState.Devices` property for the first time.

`DeviceId` type has been removed and `IDevice.Id` property is now `long`. You should update the types in your usages. If you've used the `DeviceId.Value` property, replace it with the ID itself (e.g. `device.Id.Value` becomes just `device.Id`).

USER GUIDE

This section is a narrative user guide to the library. You can find full API reference online at https://files.ravingbots.com/docs/multi-input. The documentation should also be available in your IDE, as all types, fields and methods have documentation comments.

PREREQUISITES

Multi-Input currently supports:

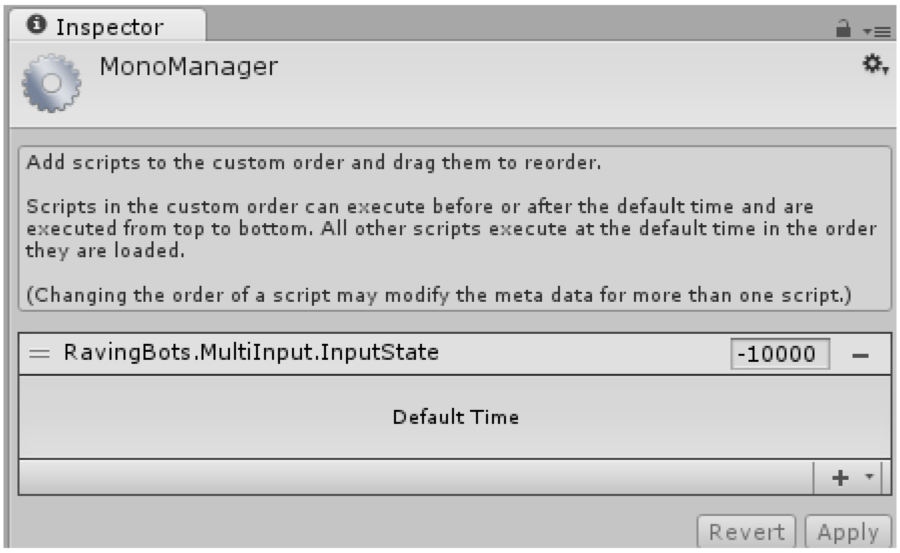
- Windows Vista and newer, both 32- and 64-bit.
- Linux with X11R7.7 (Xorg 1.12 or newer with Xinput 2.2 extension) and libevdev 1.4 (or newer) installed, both 32- and 64-bit. Currently tested on Ubuntu 15.10 and 16.04 LTS.
- OSX 10.9 Mavericks and newer, 64-bit only. Currently tested on OSX 10.11 El Capitan. Third party driver (https://github.com/360Controller/360Controller) is necessary for the controller support.

When deploying, make sure you install or instruct the user to install the required dependencies. On Ubuntu, libevdev lives in `libevdev` and `libevdev:i386` (to run 32-bit builds on a 64-bit OS) packages.

Only Xbox-compatible controllers are supported at this time. Vibration support on OSX is limited.

INPUTSTATE COMPONENT

As a first step after importing the asset package into your project, you should set the execution priority of the `InputState` component on the Script Execution Order screen (reachable from Edit > Project Settings menu). The exact value you set doesn't matter, but `InputState` must be in front of everything else.



Inspector

MonoManager

Add scripts to the custom order and drag them to reorder.

Scripts in the custom order can execute before or after the default time and are executed from top to bottom. All other scripts execute at the default time in the order they are loaded.

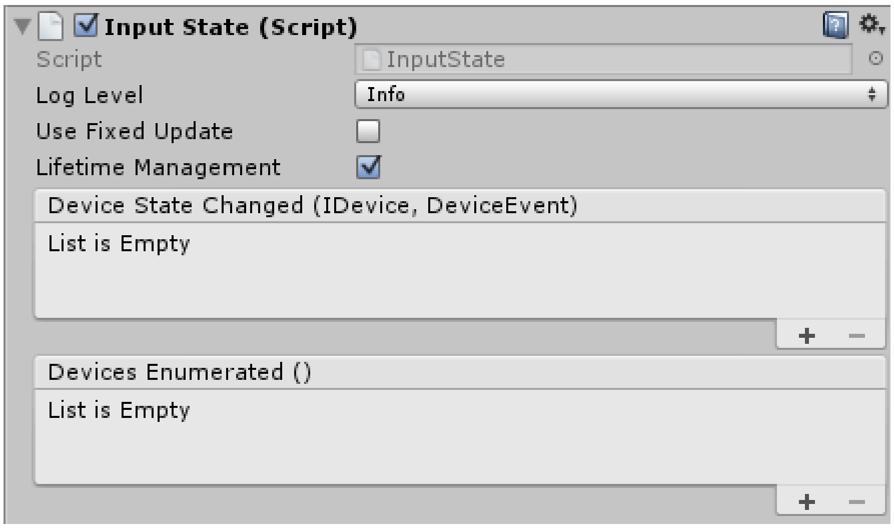
(Changing the order of a script may modify the meta data for more than one script.)

RavingBots.MultiInput.InputState -10000

Default Time

Revert Apply

After doing that, you will need to add the component to your scene. By default, it will call `DontDestroyOnLoad` on itself (because it needs to outlive the device objects you get out of it). You should not have more than one active instance.



Input State (Script)

Script: InputState

Log Level: Info

Use Fixed Update: ☐

Lifetime Management: ✓

Device State Changed (IDevice, DeviceEvent)

List is Empty

Devices Enumerated ()

List is Empty

With the component added you're ready to start using the library. In the code that uses the component you will need a reference to it. In this example, we'll use `FindObjectsOfType` method of `MonoBehaviour` to do that:

```
InputState inputState = FindObjectOfType<InputState>();
```

You can adjust the verbosity of library's logging by changing the `LogLevel` property of `InputState`. Setting it in the editor ensures that it will have the desired effect from the very start of the library lifetime.

EVENTS

Attach a listener to the `DevicesEnumerated` event so you know when the device list is ready. After you receive that event you can access `InputState.Devices` property to get a list of active devices.

To keep yourself up-to-date with device changes, you should attach a listener to the `DeviceStateChanged` event. Whenever something changes you will receive a reference to the affected device object and the type of the event (created, removed, became unusable, became usable).

You can keep device objects that are unusable, but you should discard objects that have been removed, as they will never be updated by the library again.

DEVICE OBJECTS

Device objects represent devices as reported by the system. They don't necessarily have one-to-one correspondence with actual physical devices. All device objects implement the `IDevice` interface and can be found in the `InputState.Devices` collection.

Every device object has several bits of information that describes it (note that most of these are currently returned exactly as reported by the OS, and so guarantees about them will vary by platform):

- `Id` property is the unique identifier for the device that the library uses internally. Note that the identifiers are only unique within given library instance and during its lifetime. This property never changes and no value is ever used more than once, and can be safely used to check whether two device objects are identical. This value is not persistent across library resets and so is only valid within one session.
- `IsUsable` property describes whether the device should be used by the code. Unusable devices will not report any meaningful input and should be ignored until they become usable (that includes using any of the following properties).
- `Name` property is the friendly label for the device. It's not guaranteed to be usable or meaningful. It might be a generic name like "HID-compatible mouse". On some platforms, this can be set by the user in system settings.
- `InternalName` property is the device's internal identifier. This might not persist after device is disconnected.
- `Location` property is the physical location of the device. This might be shared by multiple devices.
- `VendorId` property is the fixed numeric identifier of the device's vendor, assigned by the appropriate registry. This value will be shared by all devices made by the same vendor.
- `ProductId` property is the fixed numeric identifier of the device, assigned by the vendor. This value will be shared by identical devices (along with `VendorId`).
- `Revision` property is the version number reported by the device's driver. This value might change after driver or firmware updates. It can sometimes be used to distinguish between otherwise identical devices, but there's no guarantee this will be possible.
- `Serial` property is the device's serial number. If this value is present, it should be globally unique (i.e. no two devices should share it). Very often this value is not reported by input devices at all, though.
- `CanVibrate` property describes whether the device accepts force feedback commands. It doesn't necessarily mean that the device's motors will definitely work, but it does mean you can call `Vibrate` method on this device.

- SupportedAxes property is a list of all axes that this device claims to support. From library use standpoint, it means that the device's indexer will not return null for that axis. Devices are allowed to claim support for axes but never report any input on them, so you should be careful about using this information to classify devices. The values reported by this enumerable may change from frame to frame.

To use the force feedback motors that the device might have, call the Vibrate method. It will return true if the command was successfully uploaded to the device (but note that the devices don't report whether they actually did anything, so it's mostly useful to check whether your parameters were correct):

```
if (device.CanVibrate) {
    // vibrate at 25% force for 2 seconds
    device.Vibrate(2000, 0.25f, 0.25f);
}
```

QUERYING INPUT

All digital buttons and analøg axes the device might have are represented by unified interface: a virtual axis. To query a virtual axis, you can use the device object's indexer:

```
// axis representing a physical key
IVirtualAxis leftArrow = device[InputCode.KeyLeftArrow];

// axis representing a physical analog axis
IVirtualAxis leftStickX = device[InputCode.PadLeftStickX];

// axis representing a dervied analog axis
IVirtualAxis mouseYUp = device[InputCode.MouseYUp];
```

The most important part of the IVirtualAxis type is the Value property, representing value reported on that axis in the current frame. Non-zero Value means that the button is pressed, the stick is away from center, the mouse has moved, etc. Zero Value means that the axis is currently at rest. Additionally, you can query the value for previous frame, and the yet-uncommitted value that will be reported in the next frame, via PreviousValue and NextValue properties respectively.

Aside from float values, the axis object exposes several shortcut properties for common checks:

- HasChanged describes whether PreviousValue is different from Value.
- IsHeld describes whether the axis is not at rest, i.e. has non-zero Value.
- IsUp describes whether the axis has returned to rest state in current frame, i.e. has non-zero PreviousValue and zero Value.
- IsDown describes whether the axis has left rest state in current frame, i.e. has zero PreviousValue and non-zero Value.

For convenience `InputState` also has `FindFirst` method and related overloads, which let you query multiple axes across all devices at once, when you don't yet have a specific device in mind (typically needed in device assignment or key binding):

```
IDevice device;
IVirtualAxis axis;

// find first device where any axis has just
// returned to rest state
device = inputState.FindFirstUp();
if (device != null) {
    // act on the device
}

// ... and also get the axis in question
if (inputState.FindFirstUp(out device, out axis)) {
    // act on the device and axis
}
```

CREATING NEW VIRTUAL AXES

It's also useful to create custom axes, for example if you're implementing keyboard-based movement and would like two pair of keys to represent two ends of X and Y axis. To do that, simply create a new instance of `VirtualAxis` class (it implements `IVirtualAxis` interface just like the internal library axis objects), set the value and commit it.

The `Commit` call is necessary due to how some axes work: sometimes you have to accumulate the changes to them between frames. Explicit committing lets us do that by not updating `Value` until next frame begins.

```
// create an X and Y axes based on arrow keys
// all of these are either 0 or 1
IVirtualAxis xNegative = device[InputCode.KeyLeftArrow];
IVirtualAxis xPositive = device[InputCode.KeyRightArrow];
IVirtualAxis yNegative = device[InputCode.KeyDownArrow];
IVirtualAxis yPositive = device[InputCode.KeyUpArrow];

// note that you can treat custom axes
// exactly the same as the library ones in your code
IVirtualAxis x = new VirtualAxis();
IVirtualAxis y = new VirtualAxis();

// set to -1 if negative button is held
// set to +1 is positive button is held
// both set to 0 if neither button is held
x.Set(xNegative.IsHeld ? -xNegative.Value : xPositive.Value);
y.Set(yNegative.IsHeld ? -yNegative.Value : yPositive.Value);

// commit the axes
// until you call Commit, the new value will only be accessible
// through NextValue, which should not typically be used directly
x.Commit();
y.Commit();

// use the value
Vector2 move = new Vector2(x.Value, y.Value);
```

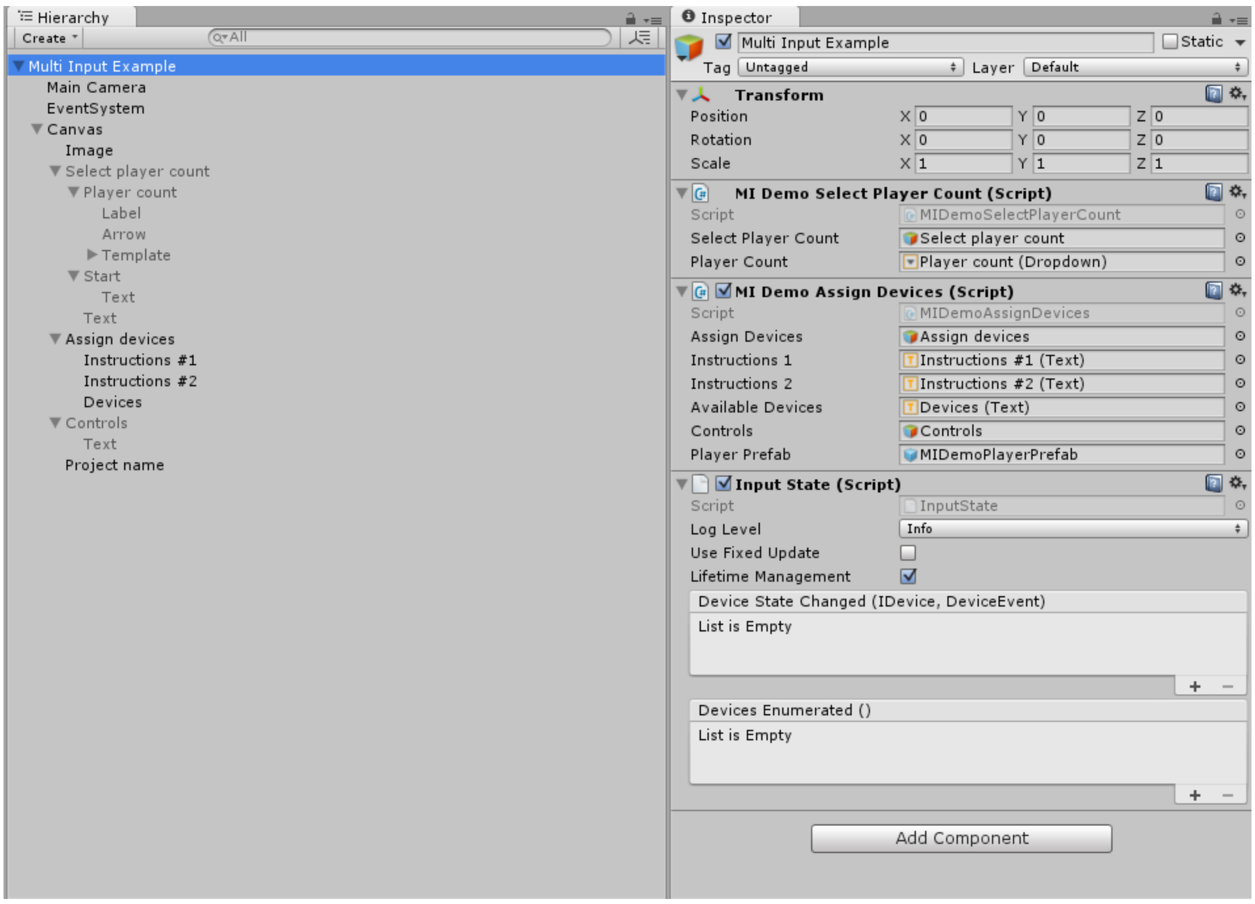
The example project contains more complex version of this code that handles multiple devices and alternate bindings.

EXAMPLE PROJECT: MOVING 2D SPRITES AROUND

This section will walk you through the included 2D example.

PROJECT STRUCTURE

Everything is set up on a single scene, `Example.unity`. The scene contains standard Unity GUI components (note that Multi-Input is not used to drive the UI here) and some of ours: `MIDemoSelectPlayerCount`, `MIDemoAssignDevices` and `InputState`.



```
Hierarchy
Create ▾                                  Q▾All
▼ Multi Input Example
    Main Camera
    EventSystem
  ▼ Canvas
      Image
    ▼ Select player count
      ▼ Player count
          Label
          Arrow
        ▶ Template
      ▼ Start
          Text
        Text
    ▼ Assign devices
        Instructions #1
        Instructions #2
        Devices
    ▼ Controls
        Text
      Project name
```

```
Inspector
☑ Multi Input Example                              ☐ Static ▾
Tag  Untagged           Layer  Default

▼ Transform
Position        X 0     Y 0     Z 0
Rotation        X 0     Y 0     Z 0
Scale           X 1     Y 1     Z 1

▼ MI Demo Select Player Count (Script)
Script                  MIDemoSelectPlayerCount
Select Player Count     Select player count
Player Count            Player count (Dropdown)

▼ ☑ MI Demo Assign Devices (Script)
Script                  MIDemoAssignDevices
Assign Devices          Assign devices
Instructions 1          Instructions #1 (Text)
Instructions 2          Instructions #2 (Text)
Available Devices       Devices (Text)
Controls                Controls
Player Prefab           MIDemoPlayerPrefab

▼ ☑ Input State (Script)
Script                  InputState
Log Level               Info
Use Fixed Update        ☐
Lifetime Management     ☑
Device State Changed (IDevice, DeviceEvent)
List is Empty
                                                      + −
Devices Enumerated ()
List is Empty
                                                      + −

                    Add Component
```

The UI is split into 3 panels:

- "Select players" contains a player count select box and a button to start the device assignment.
- "Assign devices" contains labels that show currently reported devices, and device assignment instructions for the player.
- "Controls" contains a label that explains player controls.

Additionally, there are two prefabs used:

- `MIDemoPlayerPrefab` contains a `MIDemoPlayer` component and a reference to the `MIDemoPlayerPawnPrefab`. This prefab represents the main player object (see `MIDemoPlayer` component for details). Instances of this prefab are created by `MIDemoAssignDevices` component.
- `MIDemoPlayerPawnPrefab` contains a `MIDemoPlayer` component and a reference to the `MIDemoPlayerPawnPrefab`. This prefab represents a single pawn (the game entity that's moved by player input). Each player has two: left (keyboard or a left gamepad stick) and right (mouse or a right gamepad stick). Icons and colors are configured through the values stored in this prefab. Instances of this prefab are created by `MIDemoPlayer` component for the given player.

CONTROL FLOW

When the example starts, `MIDemoSelectPlayerCount` component activates "select player" panel. The "start" button in the UI is wired to `MIDemoSelectPlayerCount`.

After "start" is pressed, `MIDemoAssignDevices.SetPlayerCount` is called and in turn creates an array of `MIDemoPlayer` objects (without pawns) by instantiating a prefab, and switches UI panel to "assign devices".

While players are not ready (to be ready, player needs to have either a keyboard and a mouse, or a gamepad assigned), `MIDemoAssignDevices.Update` will continually query all attached devices for input. Here we ignore devices that are already assigned, and if player selected a keyboard or a mouse already, we make sure that only the other one from the set will be considered.

When all players are ready, `MIDemoAssignDevices.StartGame` is called, pawns are created by calling `MIDemoPlayer.CreatePawns` for each player and UI switches to "controls" panel.

Each pawn contains a `MIDemoPlayerPawnController` component which in `MIDemoPlayerPawnController.Update` continuously queries the input, but this time only for a specific device that's stored in `MIDemoPlayerPawnController.Device` field.

Aside from main flow, two components register callbacks to `DeviceStateChanged` event:

- `MIDemoPlayer.OnDeviceStateChanged` checks the new list of devices and deassigns unplugged devices from the player, if needed. The example doesn't reassign devices if they get plugged back in.
- `MIDemoAssignDevices.UpdateDeviceList` simply updates on-screen device list, assigning a color based on the state to each one.