

OuyalInput.cs Documentation

INTRO

Purpose: This script is the core of an **input framework** providing **cross-platform controller** input for the Unity game engine in one simple to use, static access class. It does all that tiresome controller mapping and allows access to different controller types for multiple player. The setup should be very easy and as the provided access methods closely mirror the Unity.Input methods (which they wrap) you should feel at home and be ready to go in minutes. Although this framework was done with developing for the OUYA game console in mind (therefore the name „OuyalInput“), the scripts are completely independent from the OUYA SDK and work without it on many development platforms and testing devices. It is a general cross-platform tool which also works with the OUYA.

Audience: This script is most useful for developers working cross-platform with controllers on Windows, MacOSX and the Android consoles (OUYA, GameStick). It allows easy testing with standard controllers in the editor, will connect and unify many controllers in standalone builds and allows the connection of different controllers on the consoles. This is not working for the closed consoles PS3, XBOX360 or Wii - however it will allow using their controllers. So if you are an OUYA developer who wants to incorporate PS3 controller support in his game, tests with an XBOX360 controller on a Windows machine and plans to release a standalone for the MacOSX AppStore - than this is for you.

Credits & Improvements: This is a complete recoding of a framework done by Ouya's developer Tim Graupmann - he made this all possible. I wanted to do a new input wrapper building upon his work. It should be simple to understand, clean bare-bone (no confusing examples), easy to implement in any ongoing project and not dependent on other Ouya components. Another major goal was improving performance and avoiding garbage collection. This input wrapper doesn't contain slow string parsing, repeated string comparison or frame-wise string building (leading to garbage objects). Custom strings needed for axis access are cached, button keys are calculated instead of parsed and deadzone calculations can be adjusted for best performance and precision. This framework it is far from perfect and still misses support for many controllers on the market. Please help me to debug and expand this script.

Matthias Titze / [CLARK](#) / [GoldenTricycle](#) / goldentricycle@gmail.com

SETUP

Basic setup is as easy as dropping two files into your project: [OuyalInput.cs](#) and [InputManager.asset](#).

1. Copy the [InputManager.asset](#) to the ProjectSettings folder of your project. This adjusts the Unity Input Settings to provide generic controller axis channels for any controller. VERY IMPORTANT
2. Copy [OuyalInput.cs](#) to the Assets/Scripts folder (create one if necessary) of your project. This is the core input access server script which has to be used instead of Unity.Input. Its static, so no placement in the scene.

THAT'S IT - YOU ARE READY TO GO!

3. **Optional:** Copy [InputHandlerPattern.cs](#) to the Assets/Scripts folder of your project. This is a bare-bone controller client that you can attach to any GameObject in the scene. Expanding this script is a good start to develop your input handler. It's just a MonoBehaviour which shows how and where to call for input data - meant as a StarterKit.
4. **Optional:** Copy [OuyalInputTester.cs](#) to the Assets/Scripts folder of your project. This is a simple input handler client that you can attach to any GameObject in the scene. It will draw a GUI layer displaying input values for controller testing it also shows some events in the debug console. This is not an performance optimized script as it uses Unity's GUI framework for simplicity and produces a lot of strings. But we didn't want to add GUI tools - keep it simple. This MonoBehaviour is just useful for checking if your controller produces the correct values. Throw it away later.
5. **Optional:** Check all axis in the [Edit >> ProjectSettings >> Input](#) and set their „Dead“ value to 0. This allows to take full advantage of the advanced deadzone management in OuyalInput.cs (as we get raw axis data without deadzone). However this is not really necessary for easy setup and should only be done if precision adjustments are needed.

6. Optional: Install the [OUYA SDK](#) and the [Ouya Unity Plugin](#) as described in the documentation section of the OUYA developer webpage. However **DO NOT IMPORT OR USE the InputManager.asset or Input Settings** provided in these packages. You can do that and it should still work but the InputManager.asset provided in the OuyaInput framework is stripped down to the essential stuff and tested (I just killed some axis slots and cleaned up). Again, OuyaInput.cs also works with Ouya's InputManager.asset - but that's not checked (they might change stuff). You only have to use the Ouya Unity Plugin if you want to port to the OUYA. If you have nothing to do with the OUYA and just want to use cross platform controller support, than do not follow this step 6. OuyaInput.cs is completely independent of the OUYA SDK.

7. Optional: If you are using [OuyaGameObject](#) from the [Ouya SDK](#) make sure to **uncheck Legacy Input** in all OuyaGameObjects. Otherwise you will get Exceptions stating that some input axes are not set.

TESTSTED SUPPORT FOR

XBOX360 USB controller on Windows 7+8 64bit, MacOSX 10.8.1, OUYA

PS3 DS BLUETOOTH controller on Windows 7 64bit, MacOSX 10.8.1, OUYA

OUYA BLUETOOTH controller on Windows 7 + 8 64bit, OUYA

GAMESTICK BLUETOOTH controller on GameStick

MOGA PRO BLUETOOTH controller should be supported but is not tested

Some controller models will not be supported because we do not know their Unity name - tell us.

CONVENTIONS

As controller manufacturers seem to have difficulty to decide which joystick-axis directions are positive (axis invert issue) we are going to set this as a convention here. The goal of this framework is that all controllers behave the same and follow this convention. You want to have the same result for any controller you connect and not end up with an XBOX360 controller having inverted axis in comparison to the PS3 controller. This is set for all joysticks and d-pads:

Joystick or D-Pad **UP is +** positive / **DOWN is -** negative / **RIGHT is +** positive / **LEFT is -** negative

CLASS METHODS

SETUP Methods

void **SetContiniousScanning**(bool active)

void **SetPlugCheckInterval**(float seconds)

Update Methods

void **UpdateControllers**()

void **ResetInput**()

Controller Info Methods

bool **GetControllerConnection**()

string **GetControllerName**(OuyaPlayer player)

OuyaControllerType **GetControllerType**(OuyaPlayer player)

Input Access Methods

float **GetAxis**(OuyaAxis axis, OuyaPlayer player)

bool **GetButton**(OuyaButton button, OuyaPlayer player)

bool **GetButtonDown**(OuyaButton button, OuyaPlayer player)

bool **GetButtonUp**(OuyaButton button, OuyaPlayer player)

Advanced Input Access Methods

```
Vector2 GetJoystick(OuyaJoystick joystick, OuyaPlayer player)
float GetJoystickAngle(OuyaJoystick joystick, OuyaPlayer player)
float GetTrigger(OuyaTrigger trigger)
void SetDeadzone(DeadzoneType type, float radius)
void SetTriggerThreshold(float threshold)
```

Deadzone Helper Methods

```
Vector2 CheckDeadzoneAxial(float xAxis, float yAxis, float deadzone)
Vector2 CheckDeadzoneCircular(float xAxis, float yAxis, float deadRadius)
Vector2 CheckDeadzoneRescaled(float xAxis, float yAxis, float deadRadius)
```

Utility Helper Methods

```
float CalculateJoystickAngle(Vector2 joystickInput)
```

Setup Methods

OuyaInput.SetContiniousScanning

```
public static void SetContiniousScanning(bool active)
```

Generally speaking controllers show two different output types: pressure sensitive axis output (floats in a range of -1 to 1) and button event output (bools indicating the button states - down / pressed / up). Joystick always provide axis output whereas simple buttons provide button events (not real events - but specific states for each frame).

Unfortunately there are buttons that are somehow hybrids: the D-PAD and the TRIGGERS. Depending on the controller type and platform specific driver they might provide axis or button event output (or sometimes both). In case they are treated as axis they will show pressure sensitivity but can't natively provide the three button states for each frame.

That means we can't use `GetButtonUp()` or `GetButtonDown()` as these buttons don't refer to Unity's `Input.GetButton()` methods but to the `Input.GetAxis()` group.

In order to unify all controllers and make them work with all access methods, this framework includes a button states manager. The manager continuously scans these „dual mode“ buttons-axes and compares their values frame-wise to create `ButtonUp` and `ButtonDown` events for TRIGGERS and D-PADS that natively provide only axis output.

So, if you want to call `GetButtonUp()` or `GetButtonDown()` to retrieve the respective button event states (just showing in the one frame the button is pressed down or released) for TRIGGERS or D-PADS, you should

`SetContiniousScanning(true)` in the `Start()` of your input handler. This will ensure that you get these states with any supported controller. If you do not plan on using event states for these buttons you can `SetContiniousScanning(false)` to save some method calls reoccurring every frame. The **default is TRUE**.

Best called in: `Start()`

OuyaInput.SetPlugCheckInterval

```
public static void SetPlugCheckInterval(float seconds)
```

You can call this method to adjust the time between checks for new connected controllers. The smaller the interval is the quicker new controllers plugged in while running the game will be recognized. Setting a larger interval will save some tiny bit of performance and eventually garbage collection (for the one array that is renewed). As controller mappings for each known controller are cached plug in checking does in most cases not result in new controller

mappings. Therefore it does not have such a big impact on performance. The **default is 3 SECONDS**.

Best called in: `Start()`

Update Methods

OuyaInput.**UpdateControllers**

`public static void UpdateControllers()`

This has to be called in the `Start()` the `Update()` of your input handler every frame. It ensured that new joysticks are recognized, unplugging and reconnecting is checked and that the input is scanned continuously if activated via `SetContiniousScanning(true)`. Without calling this method you won't receive controller input at all. In some cases this method can also be called in `FixedUpdate()`. However we would not recommend it if you plan on using `SetContiniousScanning(true)`. It does not cost a lot of performance anyways as even the cycle for joystick recognition can be adjusted to check less often via `SetPlugCheckInterval(float seconds)`.

Best called in: `Update()`

OuyaInput.**ResetInput**

`public static void ResetInput()`

This sets all the controller axis to zero and cleans the cache for scanned button events. You can call this method at any game interruptions or level changes to make sure there is no deprecated input information stored. This is also useful when switching between different input handlers (e.g. in-game and menu controls).

Best called in: `Start()`, `OnDestroy()` ...

Controller Info Methods

OuyaInput.**GetControllerConnection**

`public static bool GetControllerConnection()`

Returns true if there is a known and supported controller connected to the system. This could be used to switch between keyboard and controller input depending on the connectivity of the controller. If you have an input handler for keyboard input you could switch to that as soon as all controllers are unplugged.

OuyaInput.**GetControllerName**

`public static string GetControllerName(OuyaPlayer player)`

Returns the name the Unity player received for a connected controller. These names are used to recognize a controller and sort it to a general controller type (which gets a preset mapping). You might be interested in the Unity controller name to find out what the player sees as the controller id. For example a PS3 controller running with the MotionInJoy driver is recognized as „MotionInJoy Virtual Game Controller“. This information comes in handy for debugging and expanding this framework to support more controllers or drivers.

OuyaInput.GetControllerType

public static **OuyaControllerType** GetControllerType(**OuyaPlayer** player)

Returns the type for a players controller (as OuyaControllerType enum). This is the general classification of the controller which is important for the mapping it receives. Controller types cover a collection of controller names. (e.g. the PS3 controller connected to MacOS via standard bluetooth gets the same type as the PS3 controller connected to a Windows machine via MotionInJoy driver.

Input Access Methods

OuyaInput.GetAxis

public static **float** GetAxis(**OuyaAxis** axis, **OuyaPlayer** player)

Returns the value of the virtual axis identified by axis and player. The value will be in the range -1 to 1 for the left joystick (OuyaAxis.LX / OuyaAxis.LY) and the right joystick (OuyaAxis.RX / OuyaAxis.RY) as well as the d-pad (OuyaAxis.DX / OuyaAxis.DY). The triggers (OuyaAxis.LT / OuyaAxis.RT) provide values in the range of 0 to 1.

Notes: Some controller have d-pads that do not really work as analog axes (e.g. Ouya). These d-pads have no sensitivity and give just -1 when pressed left or down and 1 when pressed right or up. It is better not to rely on the axis aspect of the d-pad when developing cross-platform. D-pad input can also be retrieved using the GetButton() method. Triggers on the other hand are mostly pressure sensitive axis. However some connections without proper drivers (e.g. PS3 on MacOSX via Bluetooth) don't show this sensitivity.

OuyaInput.GetButton

public static **bool** GetButton(**OuyaButton** button, **OuyaPlayer** player)

Returns true if the button identified by button and player is pressed. This is not usefull for press down or up events as this method just reveals if a button is currently held down or not. It is used to retrieve button states. This method could be called GetButtonHold() for better understanding. However we use the old name to mirror the wrapped Unity method and the general convention for mouse input events.

OuyaInput.GetButtonDown

public static **bool** GetButtonDown(**OuyaButton** button, **OuyaPlayer** player)

Returns true if the button identified by button and player was pressed down in THIS FRAME. This is usefull for press down events (i.e. you just want to retrieve a signal in the instant the button goes down).

Notes: The triggers and (for some cases) the d-pad are often native axis inputs. That means that they provide pressure sensitive values that can be retrieved via GetAxis(). The bools they provide in GetButton() are converted axis values. This also means that they do not natively answer to GetButtonDown() or GetButtonUp() for they do not map to Unity's button input methods. We need to enable the continous button state scanning to retrieve ButtonUp or ButtonDown „events“. See: [SetContiniousScanning\(bool active\)](#).

OuyaInput.GetButtonUp

public static **bool** GetButtonUp(**OuyaButton** button, **OuyaPlayer** player)

Returns true if the button identified by button and player was released in THIS FRAME. This is usefull for press up

events (i.e. you just want to retrieve a signal in the instant the button is released and goes up). This method therefore could be called `GetButtonRelease()` for better understanding. However we use the old name to mirror the wrapped Unity method and the general convention for mouse input events.

Notes: The triggers and (for some cases) the d-pad are often native axis inputs. That means that they provide pressure sensitive values that can be retrieved via `GetAxis()`. The bools they provide in `GetButton()` are converted axis values. This also means that they do not natively answer to `GetButtonDown()` or `GetButtonUp()` for they do not map to Unity's button input methods. We need to enable the continuous button state scanning to retrieve `ButtonUp` or `ButtonDown` „events“. See: `setContinuousScanning(bool active)`.

Advanced Input Access Methods

OuyaInput.GetJoystick

`public static Vector2 GetJoystick(OuyaJoystick joystick, OuyaPlayer player)`

Returns a `Vector2` (point) for the input position of the joystick (both joystick axis are checked). This can be more convenient than checking both axis separately. Moreover the input gets a refined deadzone clip or remapping which can be defined via `SetDeadzone(DeadzoneType type, float radius)`.

OuyaInput.GetJoystickAngle

`public static float GetJoystickAngle(OuyaJoystick joystick, OuyaPlayer player)`

This will directly return the angle (range 0° to 360°) of the joystick input. Internally this method gets the joystick input vector via `OuyaInput.GetJoystick()` and calculates the angle using the `CalculateJoystickAngle()` helper method. The input angle can be used for grid-based games that need to get an abstract direction from the analog joysticks. The returned angle starts at 0° on the positive joystick-x-axis and then develops counterclockwise (positive y-axis is 90° / negative x-axis is 180° / negative y-axis is 270°). This follows the conventions of polar coordinates.

OuyaInput.GetTrigger

`public static float GetTrigger(OuyaTrigger trigger)`

Returns a float for the input value of the trigger. This can be more convenient than checking the axis for the input gets a refined threshold clip or remapping which can be defined via `SetTriggerThreshold(float threshold)`. The threshold treatment depends on the `DeadzoneType` specified via `SetDeadzone(DeadzoneType type, float radius)`. Remapping will only be done for `DeadzoneType.CircularRemap`. Otherwise we just clip values below the threshold.

OuyaInput.SetDeadzone

`public static void SetDeadzone(DeadzoneType type, float radius)`

This method sets the global deadzone treatment that will be used in the advanced joystick access method. See `Deadzone enum` to get a better understanding of the different deadzone approaches. The default deadzone is `.CircularClip / 0.25f`. This is good for most games. However you may want to use the high precision `.CircularMap` approach for FPS-games or the `.AxialClip` for gridbased retro games. In order to take full advantage of the deadzone treatment implemented in `OuyaInput.cs` you will have to set all the „Dead“ values for all axis found in the `Input Manager Settings` to 0. We did not do that to allow very easy setup without prior knowledge of the deadzone problematic.

Best called in: `Start()`

OuyaInput.SetTriggerThreshold

public static void SetTriggerThreshold(float threshold)

This methods sets the global trigger threshold that will be used in the advanced trigger access method. In order to take full advantage of the advanced trigger remapping implemented in OuyaInput.cs you will have to set all the „Dead“ values for all axis found in the Input Manager Settings to 0. We did not do that to allow very easy setup without prior knowledge of the threshold (deadzone) problematic. The default threshold is 0.1f.

Best called in: Start()

Deadzone Helper Methods

OuyaInput.CheckDeadzoneAxial

public static Vector2 CheckDeadzoneAxial(float xAxis, float yAxis, float deadzone)

Converts the given axis values into an input vector which is returned. However each single axis value that is inside the deadzone gets clipped. This is a very simple approach to deadzones which leads to a cross shaped deadzone. For abstract grid based games this might be especially usefull and fast.

Parameters:

xAxis	The input value of the x-axis of a joystick or d-pad. Range: -1 to 1.
yAxis	The input value of the y-axis of a joystick or d-pad. Range: -1 to 1.
deadzone	The size of the deadzone. Range: -1 to 1. Recommended: 0.25f.

OuyaInput.CheckDeadzoneCircular

public static Vector2 CheckDeadzoneCircular(float xAxis, float yAxis, float deadRadius)

Converts the given axis values into an input vector which is returned. However vector points that are inside the deadzone radius are clipped. This approach is appropriate for most games - the deadzone is circular. However there is a sudden value jump on the border of the clipping zone. Small input values are just lost and most precise inputs therefore not possible. The input range is not complete.

Parameters:

xAxis	The input value of the x-axis of a joystick or d-pad. Range: -1 to 1.
yAxis	The input value of the y-axis of a joystick or d-pad. Range: -1 to 1.
deadRadius	The radius of the deadzone circle. Range: -1 to 1. Recommended: 0.25f.

OuyaInput.CheckDeadzoneRescaled

public static Vector2 CheckDeadzoneRescaled(float xAxis, float yAxis, float deadRadius)

Converts the given axis values into an input vector which is returned. However vector points that are inside the deadzone radius are clipped. The valid values are rescaled to map inside the full input value range -1 to 1. This approach is usefull for FPS games witch want to allow very precise movement with smallest input values. However this method is not as fast as the other deadzone approaches and should only used if really nescessary.

Parameters:

<code>xAxis</code>	The input value of the x-axis of a joystick or d-pad. Range: -1 to 1.
<code>yAxis</code>	The input value of the y-axis of a joystick or d-pad. Range: -1 to 1.
<code>deadRadius</code>	The radius of the deadzone circle. Range: -1 to 1. Recommended: 0.25f.

Utility Helper Methods

OuyaInput.CalculateJoystickAngle

public static float CalculateJoystickAngle(Vector2 joystickInput)

This convert the given joystick input vector and return the polar angle (range 0° to 360°) for that joystick input. The input angle can be used for grid-based games that need to get an abstract direction from the analog joysticks. The returned angle starts at 0° on the positive joystick-x-axis and then develops counterclockwise (positive y-axis is 90° / negative x-axis is 180° / negative y-axis is 270°). This follows the conventions of polar coordinates.

Global Enumerations

OuyaAxis Enumeration

Standard controller axes found on most controllers. This is not specific for the OUYA controller. We just use this controller as a reference. Joysticks on the same position can be called with the OuyaAxis equivalent.

- .LX** The horizontal axis of the left joystick - negative values for left and positive values for right (range -1 to 1).
- .LY** The vertical axis of the left joystick - negative values for down and positive values for up (range -1 to 1).
- .RX** The horizontal axis of the right joystick - negative values for left and positive values for right (range -1 to 1).
- .RY** The vertical axis of the right joystick - negative values for down and positive values for up (range -1 to 1).
- .DX** The horizontal axis of the d-pad cross - negative values for left and positive values for right (range -1 to 1).
- .DY** The vertical axis of the d-pad cross - negative values for down and positive values for up (range -1 to 1).
- .LT** The axis of the left trigger (range 0 to 1) - 1 being fully pressed.
- .RT** The axis of the right trigger (range 0 to 1) - 1 being fully pressed.

OuyaButton Enumeration

Standard controller buttons found on most controllers. This is not specific for the OUYA controller. We just use this controller as a reference. Buttons on the same position can be called with the OuyaButton equivalent (e.g. the buttons O, U, Y, A are equivalent to A, X, Y, B on the XBOX360 controller).

- .O** The O-button in O U Y A - maps to A on the XBOX360 (O U Y A == A X Y B) or the GameStick controller.
- .U** The U-button in O U Y A - maps to X on the XBOX360 (O U Y A == A X Y B) or the GameStick controller.
- .Y** The Y-button in O U Y A - maps to Y on the XBOX360 (O U Y A == A X Y B) or the GameStick controller.
- .A** The O-button in O U Y A - maps to B on the XBOX360 (O U Y A == A X Y B) or the GameStick controller.
- .L3** Click press button for the left joystick - the third joystick axis in a way (but not analog).

- .R3** Click press button for the right joystick - the third joystick axis in a way (but not analog).
- .LT** The left trigger treated as a button (no pressure sensitivity as in OuyaAxis.LT).
- .RT** The right trigger treated as a button (no pressure sensitivity as in OuyaAxis.RT).
- .DU** The up press on the d-pad treated as a button (this often makes more sense as OuyaAxis.DY).
- .DD** The down press on the d-pad treated as a button (this often makes more sense as OuyaAxis.DY).
- .DL** The left press on the d-pad treated as a button (this often makes more sense as OuyaAxis.DX).
- .DR** The right press on the d-pad treated as a button (this often makes more sense as OuyaAxis.DX).
- .START** The button on the start position (XBOX360). The OUYA controller doesn't have it.
- .SYSTEM** The button on the system position. This is not really working on most controllers.
- .SELECT** The button on the select or back position (GameStick / XBOX360). The OUYA doesn't have it.

OuyaJoystick Enumeration

The joysticks found on most controllers. This is not specific for the OUYA controller. We just use this controller as a reference. Joysticks on the same position can be called with the OuyaJoystick equivalent. This enum is used for advanced joystick access with preset, internal deadzone treatment.

- .LeftStick** The left joystick.
- .RightStick** The right joystick.
- .DPad** The Dpad.

OuyaTrigger Enumeration

The triggers found on most controllers. This is not specific for the OUYA controller. We just use this controller as a reference. This enum is used for advanced trigger access with preset, internal deadzone treatment.

- .Left** The left trigger.
- .Right** The right trigger.

OuyaPlayer Enumeration

As this framework supports multiple players with different controller types, each input inquiry needs to be combined with the player for which we want to get the input.

- .None** There is no player with a connected controller.
- .P01 - .P11** The players connected to the game (maximum .P11). Each player may use a different controller.

OuyaControllerType Enumeration

This is a list of all the controller types we try to support. Controllers not fitting in one of these types are not supported yet. Some types are not tested and not all drivers can be covered. Please contact the developers if you have issues with some controllers you use or can provide mapping information for uncommon controllers. We have not

all controllers and therefore can't get the Unity-IDs (names) for them. Names can differ depending on driver or connection - so if you have a name you miss, contact us.

.Broadcom	Controller connected via the Broadcom Bluetooth driver - not tested yet - Android
.GameStick	The GameStick Bluetooth controller - tested on GameStick DevKit - GameStick Android
.MogaPro	The Moga Pro Bluetooth controller - not tested yet - Android
.Ouya	The Ouya Bluetooth controller - tested on Windows 8, Ouya Ouya Android via standard Bluetooth connection Windows 7 64bit via standard Bluetooth connection Windows 8 64bit via standard Bluetooth connection
.PS3	The PS3 Dual Shock Bluetooth controller - tested on MacOSX, Windows 7, Ouya, Unity Editor MacOSX 10.8.1 via standard Bluetooth connection (no special driver) or USB Windows 7 64bit via MotionInJoy Driver and USB Ouya Android via standard Bluetooth connection or USB
.XBox360	The XBOX360 USB controller for Windows - tested on Windows 7 + 8, Ouya, Unity Editor Windows 7 64bit via Official Microsoft Driver Windows 8 64bit via standard Preinstalled Driver Ouya Android via simple USB connection
.TattieBogle	The XBOX360 USB controller for Windows - tested on MacOSX 10.8.1, Ouya, Unity Editor MacOSX 10.8.1 via TattieBogle Driver and USB
.Unknown	We do not know the name for that controller, driver or connection type.
.None	No controller is connected to the game as far as we can see.

ButtonAction Enumeration

This enum describes the different state a button can have.

.Pressed	The button is held down. This is a real button state.
.UpFrame	The button is released. This state will only appear during one action frame - like an event.
.DownFrame	The button goes down. This state will only appear during one action frame - like an event.

DeadzoneType Enumeration

This enum describes different deadzone treatment approaches. These approaches convert the raw input values to cope with worn out joysticks. They lead to different results and have different impacts on performance. For convenience there is already a deadzone set in the Input Manager Settings. If you want to have full control over deadzone you should set the „Dead“ value in every axis you will find in the Input Settings to 0.

.AxialClip	Each axis is clipped individually when inside the deadzone - cross shaped deadzone. This approach is good for grid based games and very fast.
.CircularClip	Input is only clipped when the input point (both axes vector) is inside the deadzone radius. This approach is good for most games and still quite fast.
.CircularMap	Same as .CircularClip but values outside the deadzone will be rescaled to allow full range input. This approach is needed for high precision input (FPS) but costs more performance.

Known Issues

As this framework is far from perfect there are always some details that are not working correctly. We try to fix all bugs or issues that were discovered by testers and communicated in the forums or via mail. However there are some things which can't be fixed easily or need extensive research. Most of these issues depend on the Unity engine and not on the scripts of this framework. As Unity is not open source we do not have access to the lower level classes this framework wraps. So a bug report sent to the Unity guys is often our only chance to adress these problems. That's why we document them here and explain the background:

Hot-Plugging on Windows Standalone

On standalone builds for Windows controllers won't be recognized if they where not connected before the start of the game. This makes swapping controllers during the game nearly impossible. This issue does not appear on MacOSX or in the Unity Editor. It is due to a lower level bug in the Unity Windows Standalone Player and connected to the setup dialog for resolution and input that shows up at the start of the game. Some testers reported that the problem does not occur when this dialog is prevented via the [Player Settings >> Standalone >> Display Resolution Dialog >> Disabled](#). However this makes custom resolution setup at start difficult and is not a suitable fix.

Summary: The Input Setup Dialog on Windows Standalone Builds seems to prevent future controller polling. Therefore hot-plugging is not working correctly.

Connection & Drivers

This part is a collection of tips I found on the internet while trying to connect the different controllers to specific platforms. They are especially usefull if there is no native driver a certain controller on a testing platform or OS.

Connect OUYA to Windows via Bluetooth

An OUYA controller can easily be connected to Windows 8 via standard Bluetooth:

On Windows 8: Go to SETTINGS (sidebar) >> Change PC Settings (lower right corner) - a settings screen will pop up. In the list to the left scroll down to DEVICES. Make sure your Bluetooth adapter is switched on. Now hold down the OUYA-center-button on your controller until the four flashing lights switch to two. In your devices settings pane hit ADD DEVICE. The system is now searching for new devices. It might be that you have to hold the OUYA-center-button again. After a moment the OUYA Game Controller should appear in your list - click on it. A bar below the device name indicates that some stuff is downloaded, installed or set up. However you should be ready to go as soon as you have added the device. I noticed that sometimes the controller stops working. Removing and reinstalling the device solved that issue.

On Windows 7: Connecting the OUYA Game Controller via standard Bluetooth is pretty similiar to Windows 8. Of course you will have to search for the DEVICES setting elsewhere. You can find the in your SYSTEM SETTINGS under DEVICES & PRINTERS. There is an option in the menu for adding new devices. From there you can follow the steps described under Windows 8.

Reference: <http://www.youtube.com/watch?v=Cs7EgWI0DcQ>

Connect XBOX360 to OUYA via USB

An USB XBOX360 controller can easily plugged into the USB port of the OUYA:

Using OUYA's USB port will unfortunately shut down Bluetooth. This prevents the use of multiple controllers. XBOX360 Wireless controllers need a speiel receiver dongle (as far as I know). The setup should be very similiar to the PS3 setup via bluetooth.

Connect XBOX360 to Windows or MacOSX via USB

XBOX360 has driver support on Windows and MacOSX:

On Windows you will of course find Microsoft's official XBOX360 controller driver best suited. Unfortunately this support doesn't exist on MacOSX. There is however the TattiBogle driver project:

<http://tattiebogle.net/index.php/ProjectRoot/Xbox360Controller/OsxDriver>

This driver works well as long as you boot your Mac with the controller already plugged-in. On MacOSX 10.8.1 I experienced immediate reboots of the whole machine whenever I plugged or unplugged the controller while the system was running. So be careful and save your project while testing!

Connect PS3 to OUYA via Bluetooth

Here's how to pair your PS3 controller with the OUYA via Bluetooth:

This guide is intended to guide you through the steps needed to connect your Playstation 3 Dual Shock Bluetooth Wireless controller to your OUYA. It should be noted that not all OUYA games support the PS3 controller (well, that's what this Unity framework is for). However, it should allow you at a minimum to navigate the OUYA menu.

1. Turn your OUYA on and make sure you have at least one OUYA controller paired.
2. Plug your PS3 Controller into your OUYA using a mini-USB to standard USB cable.
3. Hold down the home button on your PS3 controller. This should take you back to the OUYA home screen. Make sure the PS3 controller is working by using it to navigate the menus. You will also note that the OUYA controller has been disconnected and won't pair. This is because plugging in the PS3 controller on USB turns off bluetooth.
4. To solve this, first completely power down your OUYA. You can do this by either holding down the power button on the top of the console for 5 seconds (recommended) or unplugging the power cable.
5. Unplug the PS3 controller from the mini-USB wire, and unplug the same wire from the OUYA.
6. Turn the OUYA back on. When it finishes booting up, you should see the usual pairing screen appear.
7. Hold down the home button on the PS3 controller as you would with a normal OUYA controller to pair it. Other OUYA controllers should now also pair.

Reference: <http://ouyaforum.com/showthread.php?2752-How-to-Pair-a-PS3-Controller-with-OUYA>

Connect PS3 to MacOSX via Bluetooth

Here's how to set up your PS3 controller via Bluetooth for some wireless gaming joy with your Mac:

First, launch System Preferences and click to open the Bluetooth preference panel. Click the checkbox next to On, and Discoverable. Then plug the mini USB charging cable into an available USB port on your Mac. Next, plug the mini USB end of that same cable into your PS3 controller. The red LED lights will blink slowly, as your Mac is now charging the PS3 controller.

Now, hold the PS button down on the controller, for about 3 seconds. Then release it, and pull out the USB mini cable from the controller. Leave it plugged into the Mac; you'll need it in a moment. The red LED lights will start to flash faster, and your Mac will ask for a passcode. Make one up, or use the standard '0000' to make the passcode dialog box disappear.

It'll come back up again. Before you type in the passcode again, and again, and again, navigate to the preferences window again and click on the little gear there at the bottom. Add the PS3 controller to your Favorites list, then click the checkbox next to the ON label again to turn Bluetooth OFF. This is counter-intuitive, but do it anyway.

Now, strangely enough, reconnect the mini USB cable to your PS3 controller (since you left it in your Mac as directed above, right?), press the PS button for 3 seconds again, then pull the cable out. Your PS3 controller will continue to flash like it's trying to pair. Ignore that and turn Bluetooth on again on your Mac. Your PS3 controller should now show up in the device list, letting you use it with any applications that have support for the controller. The standard Mac

connection however will not support all features of this controller. Pressure sensitivity for triggers and d-pad as well as gyro input is lost without a designated driver. The core functionality is there at least (tested on OS 10.8.1).

Reference: <http://www.cultofmac.com/198793/use-a-playstation-3-controller-on-your-mac-with-bluetooth-os-x-tips>

Connect PS3 to Windows via USB

Here's how to connect your PS3 controller to your Windows machine via USB (or Bluetooth):

The most common procedure is to use a driver tool called MotionInJoy:

Win 7-8 32bit http://www.xinputer.com/download/MotioninJoy_060001_x86_signed.zip

Win 7-8 64bit http://www.xinputer.com/download/MotioninJoy_060001_amd64_signed.zip

These drivers and setup tools have a lot of options, can emulate XBOX360 controllers and be customized. They are however also known for being a bit risky and unstable. (We did not discover any issues, but...). These drivers are compatible with Windows 7 and Windows 8 only.

After downloading the drivers, connect your PS3 controller to your PC. As you connect the controller, Windows will detect the device and appear to install some drivers. However what happens here isn't enough to allow you to use the controller and further drivers - downloaded previously - are required. Open the ZIP file and run the .EXE file to install the MotionInJoy software (DS3 Tool), agreeing to any notifications Windows displays. With your PC connected to the web, wait while the USB driver for the PlayStation 3 controller is downloaded and installed and then follow the on-screen instructions. When ready, click Load to complete this stage of the process and your controller should appear ready to use.

However we need to do some custom settings as the default do not really make sense with Unity. To proceed you will need to exit the DS3 Tool software and then reload it - this allows the software to detect the controller and forces it to display a different set of options. In „Profiles“ you will find a option to create new settings. Create a new custom button setting with the values shown in the screenshot. Save the new settings under UnityInput (or whatever). Go back to the „Profiles“-screen and switch the first roll-down to UnityInput (fourth checkbox). The driver should now send usefull values in the correct channel to the Unity player. This is a bit complicated but the only clean way we found. Using the default settings MotionInJoy send the gyro sensor data on the same channel as the right stick axis. That's why we do the custom setup.

MotioninJoy Gamepad tool

Home Local ? Help

Profiles Driver Manager BluetoothPair Language Options About

Dxinput game options

Reload this p

DS3/Sixaxis	Dxinput	Auto	Turbo	Macro
Triangle	Button 1	<input type="checkbox"/>	<input type="checkbox"/>	
Circle	Button 2	<input type="checkbox"/>	<input type="checkbox"/>	
Cross	Button 3	<input type="checkbox"/>	<input type="checkbox"/>	
Square	Button 4	<input type="checkbox"/>	<input type="checkbox"/>	
L1	Button 5	<input type="checkbox"/>	<input type="checkbox"/>	
R1	Button 6	<input type="checkbox"/>	<input type="checkbox"/>	
L2	Button 7	<input type="checkbox"/>	<input type="checkbox"/>	
R2	Button 8	<input type="checkbox"/>	<input type="checkbox"/>	
SELECT	Button 11	<input type="checkbox"/>	<input type="checkbox"/>	
L3	Button 9	<input type="checkbox"/>	<input type="checkbox"/>	
R3	Button 10	<input type="checkbox"/>	<input type="checkbox"/>	
START	Button 12	<input type="checkbox"/>	<input type="checkbox"/>	
PS	Button 13	<input type="checkbox"/>	<input type="checkbox"/>	
D-pad up	Button 14	<input type="checkbox"/>	<input type="checkbox"/>	
D-pad right	Button 15	<input type="checkbox"/>	<input type="checkbox"/>	
D-pad down	Button 16	<input type="checkbox"/>	<input type="checkbox"/>	
D-pad left	Button 17	<input type="checkbox"/>	<input type="checkbox"/>	
Triangle	None			
Circle	None			
Cross	None			
Square	None			
L2	Z+			
R2	Rz+			
L1	None			
R1	None			
Dpad Up	None			
Dpad Right	None			
Dpad Down	None			
Dpad Left	None			
Left stick x+	X+			
Left stick x-	X-			
Left stick y+	Y+			
Left stick y-	Y-			
Right stick x+	Rx+			
Right stick x-	Rx-			
Right stick y+	Ry+			
Right stick y-	Ry-			
Front-tilt	None			
Back-tilt	None			
Left-tilt	None			
Right-tilt	None			

Red arrows indicate the values differing from the default settings.

Don't forget to change the input configuration in the "Profiles" tab.

Config Macro Options

new Config(unname) + New

- DX-Default
- UnityInput

Profiles Driver Manager BluetoothPair Language Options

Quick config for your game(s)

Donate Recommend

Connected game controller(s): 1. Dualshock 3 sixaxis (USB) Disconnect 100%

Select one mode:

- Playstation 1 (Dpad, without joystick)
- Playstation 2 (POV, joysticks)
- Playstation 2' (for PCSX2 with pressure sensitive)
- UnityInput** Playstation 3=>Options
- Xinput-Defau Xbox 360 Controller Emulator=>Options
- Custom-Defa Custom=>Create

LED

Battery information Custom: 1 2 3 4

Left Motor: 109% Right Motor: 99%

Enable Vibration Testing Game Controller Panel

Save Changes Default Game Controller Panel