CarbonInput Library for Unity

Henrik Heine
InfectedBytes

October 1, 2017

Contents

1 Introduction

CarbonInput is a simple extension for better gamepad management in Unity. Unity itself already provides a simple input facility for gamepads. It is based on defining the buttons and axes beforehand and uses them at runtime by their name.

```
float axis = Input.GetAxis("Horizontal");
bool fire = Input.GetButton("Fire1");
```

The input system works, but there are at least two problems with it:

1. String handling is error prone. It might happen that one accidentally writes `"Fire 1"` instead of `"Fire1"`.

2. Gamepads are not standardized. The right thumbstick of a PS3 controller uses axis 3 and 5, while a XBox360 Controller uses axis 3 and 4.

CarbonInput defines a new Input System, which will try to recognize the used controller in order to use the correct mapping. You don't have to define any axis, CarbonInput is doing that for you and you can access the correct buttons directly:

```
// X axis of right thumbstick, automatically mapped to the correct Unity axis
float axis = GamePad.GetAxis(CAxis.RX);
// A-Button on XBox360 controller, Cross on PS3 controller
bool jump = GamePad.GetButton(CButton.A);
```

2 Getting started

This section will cover the basic steps to get started with CarbonInput.

2.1 Getting CarbonInput

If not yet done, you should download CarbonInput from the Unity Asset Store and import it to your project. In the import window, make sure that all files are selected.

The `Demo` folder is not necessary, but it might be a good idea to run the test scene to get familiar with CarbonInput.

2.2 Initialize CarbonInput

CarbonInput is a wrapper around the Unity Input manager. Therefor it needs to setup some axis, but don't worry, you don't have to define any axis manually. It will initialize itself by simply clicking a menu item:

```
Edit > Project Settings > Carbon Input > Create Carbon Input Axis
```

Now the system will add many axes to your Input manager and you can start using CarbonInput.

2.3 First steps

If you like you could start the `Demo` scene inside the `CarbonInput/Demo` folder. The scene only consists of simple GUI, showing all buttons and axes of all players. Additionally it adds some onscreen controls. If you have less than four controllers connected, CarbonInput will use a fallback keyboard mapping for the missing gamepads. So if you only have one XBox360 controller connected, you will see one column matching any input, one column matching the XBox360 controller (player one), one column for the onscreen controls (player two) and two columns using the fallback keyboard input (player three and four). You can modify the scene as you like and play a bit around. The following section might be useful.

2.4 Basics

The easiest way to get started is by replacing your calls to the Unity input system with calls to CarbonInput.

```
bool button = GamePad.GetButton(CButton.A); // A button
float value = GamePad.GetAxis(CAxis.LX); // left x axis
```

You could also get both axes of a thumbstick or the dpad directly:

```
// using CStick enumeration:
Vector2 left = GamePad.GetStick(CStick.Left);
Vector2 right = GamePad.GetStick(CStick.Right);
Vector2 dpad = GamePad.GetStick(CStick.DPad);
// direct functions:
Vector2 left = GamePad.GetLeft();
Vector2 right = GamePad.GetRight();
Vector2 dpad = GamePad.GetDPad();
```

2.5 EventSystem

If you want to use CarbonInput with Unity's UI System, you have to make the EventSystem aware of it. By default the `Standalone Input Module` script is attached to it, which is not aware of the correct gamepad mappings. This is of course not the desired behavior, therefore you have to remove the `Standalone Input Module` and add the `Carbon Input Module` instead. Internally they are basically the same, but instead of `Input.GetAxis(...)` it uses `GamePad.GetAxis(...)`.

3 GamePad Details

The CarbonInput API is designed similar to the XNA framework. The central access point is the `GamePad` class. On startup, it will load up some mapping files from a resources directory called `Mappings`. Each of those files contains information about how that specific gamepad will use each axis. After loading those files, it will check which controllers are currently connected and it will try to find the best match.

⚠ All mapping files have to be inside a folder called `Mappings`, which must be inside a `Resources` folder. So the easiest way would be to keep them in their default directory, which is: `Assets/CarbonInput/Resources/Mappings`

3.1 Enumerations

One typical problem with Unity is the excessive usage of strings. It might happen that you run into problems, just because you misspelled something. Maybe you wrote `"Fire 1"` instead of `"Fire1"` or you forgot which one was correct and then you have to look it up. Therefor CarbonInput uses enumerations instead of strings. So your compiler would tell you if you misspelled something and the intellisense will show you all possible values.

The following enumerations will be very important for your project:

```
// needed if you want to check the gamepad of a specific player
public enum PlayerIndex {
    Any, One, Two, Three, Four, Five, Six, Seven, Eight
}
```

```
// All available buttons
// using the XBox naming scheme
public enum CButton {
    A, B, X, Y,
    Back, Start,
    LB, RB,
    LS, RS,
}
```

ⓘ If you don't like the XBox naming scheme, you could also use the Playstation naming scheme, given by the `PSButton` enumeration.

```
// All available axes
public enum CAxis {
    LX, LY, // X/Y of Left thumbstick
    RX, RY, // X/Y of Right thumbstick
    LT, RT, // Left and Right trigger
    DX, DY  // X/Y of DPad
}
```

```
// Used to query two axes at once
public enum CStick {
    Left, Right, DPad
}
```

The following methods are defined statically in the `GamePad` class:

```
bool GetButton(CButton btn)
float GetAxis(CAxis axis)
Vector2 GetStick(CStick stick)
Vector2 GetLeftStick()
Vector2 GetRightStick()
float GetLeftTrigger()
float GetRightTrigger()
Vector2 GetDPad()
GamePadState GetState()
```

Each method can take an optional parameter of type `PlayerIndex`. If you dont specify an index, `PlayerIndex.Any` will be used. Therefor the following two lines will do the same:

```
bool button = GamePad.GetButton(CButton.A, PlayerIndex.Any);
bool button = GamePad.GetButton(CButton.A);
```

ⓘ You can change the behaviour of `PlayerIndex.Any` in the settings, see 5.1

3.2 GetState

Sometimes it might be useful to get a complete state of a gamepad. This can be done with this:

```
GamePadState state = GamePad.GetState(); // you can again add a PlayerIndex
```

The `GamePadState` represents a snapshot of the current frame for a specific gamepad. You can now query all buttons more directly:

```
if(state.A) { ... }
```

The state not only consists of all buttons, but also of all axes. The x and y axes of the thumbsticks and the dpad are combined into `Vector2` instances.

```
Vector2 leftStick = state.Left; // or Right or DPad
float value = state.LT; // LeftTrigger
```

ⓘ A lot of mobile gamepads don't have real triggers, but only buttons. For those gamepads CarbonInput will fake the triggers by returning 0 if the button is not pressed and 1 if it is.

One additional feature of the `GamePadState` is the ability to check if a button was pressed or released during this frame. This comes in handy, if you want to trigger something only once and not for for the whole time a button is hold down.

```
if(state.Pressed(CButton.A)) { ... }
if(state.Released(CButton.A)) { ... }
```

3.3 Aliases

Sometimes it might be useful to use aliases like `Jump` instead of `CButton.A`. To do so, you can create a simple `CButton` variable.

```
CButton Jump = CButton.A;
```

Now you can use this alias instead of the `CButton`:

```
if(GamePad.GetButton(Jump)) { ... }
```

If you want to access your aliases from different scripts, it might be a good idea to put them in one place.

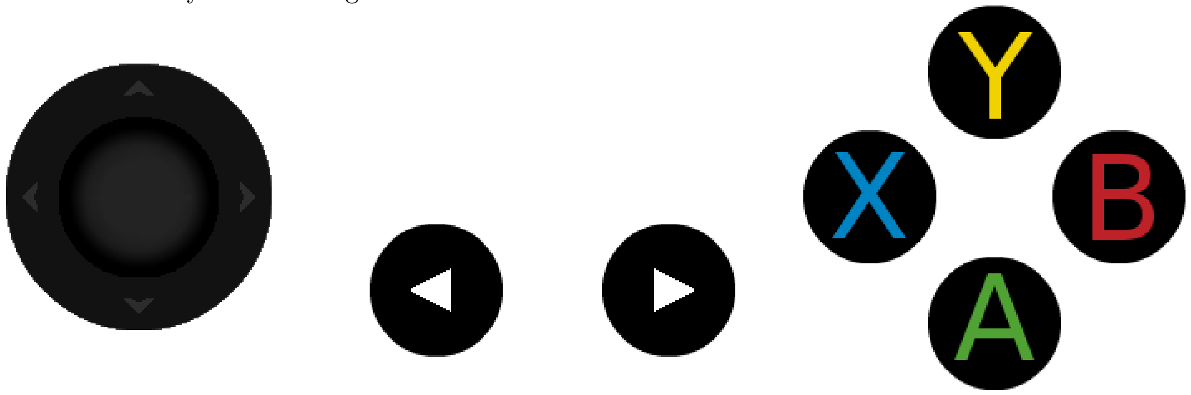
```
public static class Aliases {
        public static CButton Jump = CButton.A;
        public static CButton Fire = CButton.X;
        // ...
}
```

`CButton` is only a simple enumeration, therefor it is serializable. If you like, you could store them also in the `PlayerPrefs`, because an enumeration is internally nothing more than a simple integer. Therefor you can always cast between `CButton` and `int`:

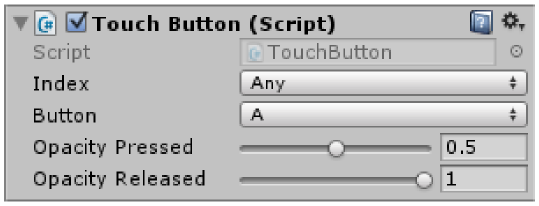
```
PlayerPrefs.SetInt("Jump", (int)Jump);
Jump = (CButton)PlayerPrefs.GetInt("Jump");
```

4 TouchInput

CarbonInput comes with an own touch input system. On systems like mobile devices, you often don't have a gamepad connected, therefor you have to use touch controls. You can use thumbsticks as well as normal buttons. Both are available in different styles, like a dark and a light style. Instead of using a generic button without a label, you could also use the predefined A, B, X and Y buttons. Of course you could also use your own images.



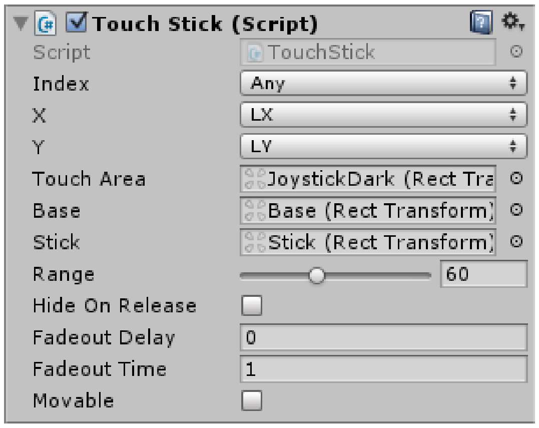
Touch controls are no real gamepads, therefor they are handled a bit different internaly. On each control you directly define which `PlayerIndex` it belongs to and which `CButton` or `CAxis` it is mapped to.



Touch Button (Script)

Script	TouchButton
Index	Any
Button	A
Opacity Pressed	0.5
Opacity Released	1

Figure 1: Inspector of a `TouchButton`



Touch Stick (Script)

Script	TouchStick
Index	Any
X	LX
Y	LY
Touch Area	JoystickDark (Rect Tra
Base	Base (Rect Transform)
Stick	Stick (Rect Transform)
Range	60
Hide On Release	☐
Fadeout Delay	0
Fadeout Time	1
Movable	☐

Figure 2: Inspector of a `TouchStick`

4.1 TouchButton

As you can see in the inspector, there are two additional settings. They are used to modify the opacity of the button, depending on its state. By default a released button is rendered opaque and a pressed button is rendered with transparency.

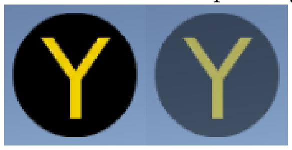


Figure 3: Released and pressed state

4.2 TouchStick

The inspector of a `TouchStick` is a bit more complex:

- The `Range` defines how far away the center of the stick can be from the center of the base
- If `Hide On Release` is set to true, the Joystick will disappear after releasing it and it will reappear if you touch anywhere inside the `TouchArea`.
  The following settings are only needed if it is set to true:
  - `Fadeout Delay`: Defines after how many seconds the joystick will start to fade out.
  - `Fadeout Time`: Specifies how long it takes to fade out.
- If `Movable` is set to true and the user moves out of `Range`, the joystick will follow the movement. The joystick will never leave the `Touch Area`.

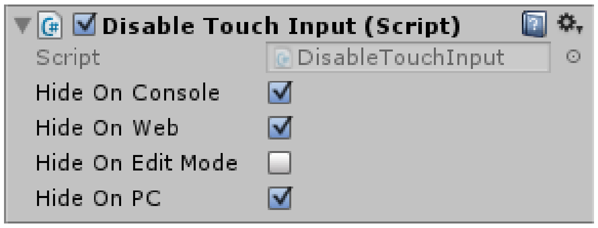
4.3 Graphics

You can replace the provided graphics with your own. To do so, just click on any touch control and drag and drop a new image onto the `Image Component` of any control. A joystick consists of two images, one for the `Base` and one for the `Stick`.

4.4 Auto Disable

If there is a real gamepad connected, you probably don't need the touch controls. For this cases you could add the `DisableTouchInput` component to the `Canvas`. On startup it will check if there is any gamepad and if so, it will deactivate all touch controls. On some platforms it doesn't make much sense to have touch controls at all. For example on a normal pc build, because there you have a keyboard.

If you want to disable touch controls always on some platforms, you can set some checkboxes on the `DisableTouchInput` component:



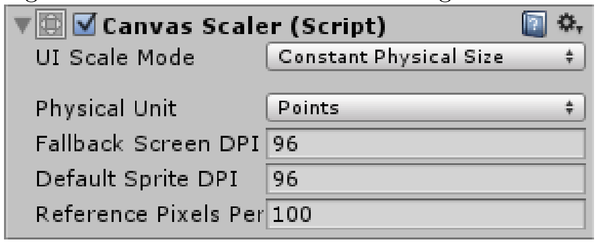
Disable Touch Input (Script)	
Script	DisableTouchInput
Hide On Console	✓
Hide On Web	✓
Hide On Edit Mode	☐
Hide On PC	✓

Figure 4: Inspector of `DisableTouchInput` component

The settings shown in the image will always hide the touch controls in the webplayer, on console and on standalone platforms. In edit mode and on mobile platforms, like Android, the touch controls will only disappear if there is a gamepad connected.

4.5 Best Practice

The root gameobject of any touch control must be a `Canvas`. Otherwise it won't be part of the UI. Your game will probably run on different screen sizes and resolutions. Therefor it might happen that the touch controls are really small on high DPI screens, like on mobile phones. To overcome this issue, you can adjust the settings of the canvas. It has a `CanvasScaler` component attached, which can be used to scale the UI up. One good way to go is setting the scale mode to `Constant Physical Size`. With this setting your UI will always have the same physical size on any screen. If you do so, you have to change the size of all your touch controls to your preferred size. For joysticks you should change the scale of the `Base` and for buttons it might be the best to change the scale of the button itself or of the parent transform. A scaling factor of about 0.3 seems to be a good choice.



Canvas Scaler (Script)	
UI Scale Mode	Constant Physical Size
Physical Unit	Points
Fallback Screen DPI	96
Default Sprite DPI	96
Reference Pixels Per	100

Figure 5: `CanvasScaler`

If you want to get started as quick as possible, you could try the `ScalableTouchControls` prefab. It is a fully configured `Canvas` with one thumbstick, `Back` and `Start` buttons and an `ABXY` button group. The scaling mode of the canvas is already set to `Constant Physical Size` and all controls are scaled to a nice physical size.

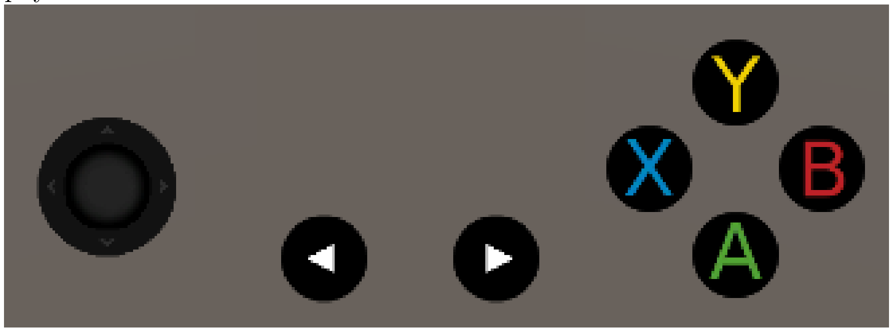


Figure 6: Scalable Touch Controls

5 CarbonInput Settings

You can find the CarbonInput settings file inside the `Assets/CarbonInput/Resources` directory.

5.1 Behaviour of `PlayerIndex.Any`

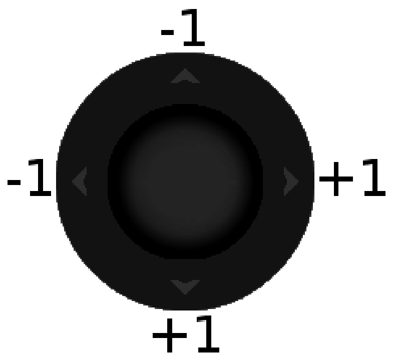
The `AnyBehaviour` enumeration defines the following three behaviours:

- `UseMappingOne`:
  Whenever `PlayerIndex.Any` is used, the system will get the motion of all connected gamepads, using the mapping of `PlayerIndex.One`.
  If all connected gamepads are using the same layout, this is the best solution. But if you have two controller with different layouts, like one XBox360 and one PS3 controller, this might cause problems, because they use different Unity Axes for their controls. For those cases, it would be better to use `CheckAll`.
- `UseControllerOne`:
  The system will always use `PlayerIndex.One` instead of `PlayerIndex.Any`.
- `CheckAll`:
  Whenever `PlayerIndex.Any` is used, it will get the motion of all four gamepads and it will return the first match.

By default the behaviour is set to `CheckAll`.

5.2 Inverted Axes

By default the X axis of any axis will go from -1 (left) and +1 (right). Every Y axis will go from -1 (up) to +1 (down). So if you don't like that, you can just check the checkbox for the specific axis and the system will then flip the axis for you.



6 Define new mappings

CarbonInput tries to support many mappings, but for the case that your controller is not fully supported, you can define a new mapping for it.

6.1 Using AutoMapper scene

Just run the `Assets/CarbonInput/Demo/AutoMapper/AutoMapper` scene and press your buttons in the shown order. The scene consists of a basic controller image and a blinking sprite (see figure 8). This sprite shows you which button you should press. After every button is pressed, you have to move the thumbsticks in the indicated direction. If your gamepad doesn't have a specific button or axis, just press the skip button above the image. After all steps, the scene will create a new mapping in the mappings directory of CarbonInput. If that directory does not exist, it will be stored in the root directory of your assets and you have to move it to any `Resources/Mappings` directory.

The mapping is not complete, you have to specify the allowed platforms and the regex by yourself. To do so, just click on the mapping and modify the values in the inspector. Now you should run the `Demo` scene and test your controller. If something is not working properly, you should check the mapping and modify the values by hand.

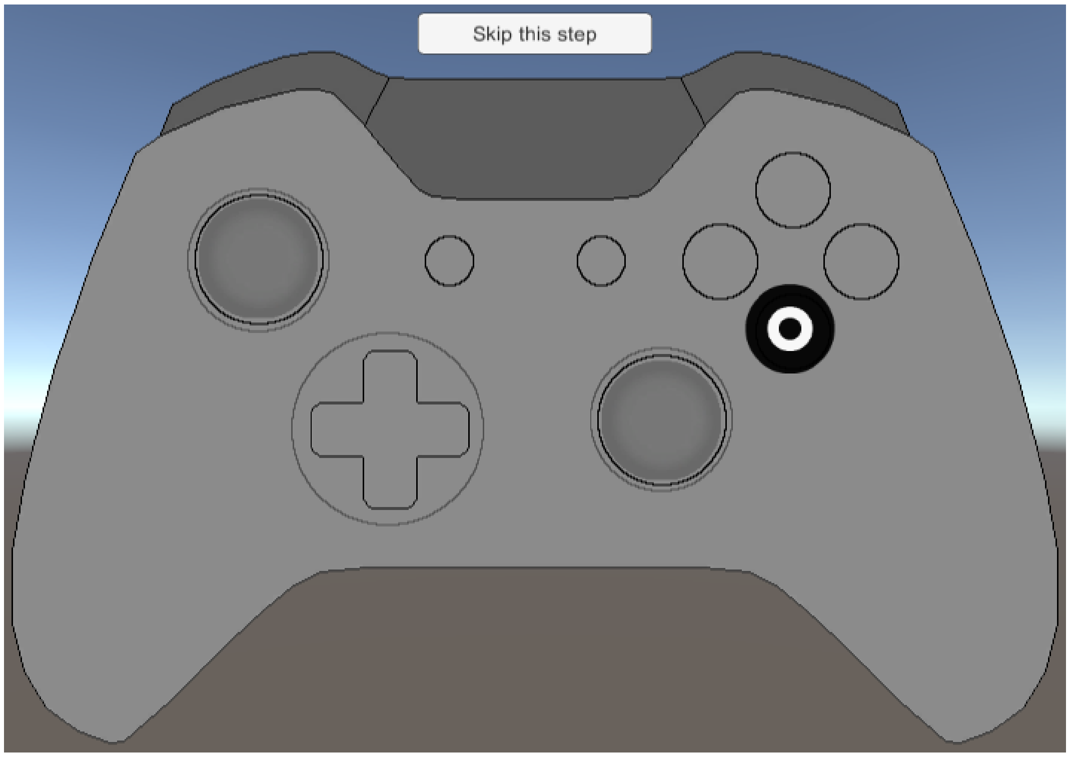
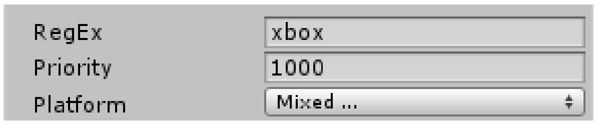


Figure 7: AutoMapper



RegEx	xbox
Priority	1000
Platform	Mixed ...

Figure 8: AutoMapper

6.2 By Hand

To create a new mapping by hand, navigate to a `Resources/Mappings` directory and click on `Assets > Create > Carbon Input > GamePad Mapping`. This will create a new mappings file for you. You can now modify the mapping in the inspector.

The `RegEx` is used to check if a connected gamepad can be used with this mapping. The regex `xbox` would match any controller name that contains the word `xbox`, ignoring case. Mappings with lower `Priority` will be tested earlier. The last setting is the `Platform`. Click on the drop down and select all platforms this mapping can be used on. The concrete mappings are now a bit more complex.

6.2.1 Buttons

The dropdown entry will determine how the button will behave. If it is set to `Default` the following number is the index of the joystick button (0-19). If it is set to `Wrapper` then the next entry is a `KeyCode`. If this button is not supported, you could set its keycode to `None`.

6.2.2 Axis

There are different types of mappings possible:

- `Default`: The mapping will use a real gamepad axis, which must be in the range 0-11.

- `Clamped`: The mapping will also use a real gamepad axis, but its value will be clamped to the range `Min - Max`. Some gamepads are using a single axis for the left and right trigger. For those cases you should clamp the values.
- `Key Wrapper`: Instead of using any gamepad control, this axis will be emulated by two keyboard keys. The axis will return -1 if the `negative` key is pressed, +1 if the `positive` key is pressed and 0 if none or both are pressed.
- `Button Wrapper`: The gamepad only has a single button for this axis. Often used when the gamepad does not have trigger.
- `Button Wrapper 2`: This axis is emulated by two gamepad buttons. The axis will return -1 if the `negative` key is pressed, +1 if the `positive` key is pressed and 0 if none or both are pressed. This is useful if your gamepad only has buttons for the dpad.
- `TriggerLimiter`: The gamepad axis goes from -1 to 1, but because it is a trigger, it should go from 0 to 1. Gamepads like the PS4 gamepad are using a full range axis for each trigger. For those cases you should use this setting in order to shift the axis value to the correct range.