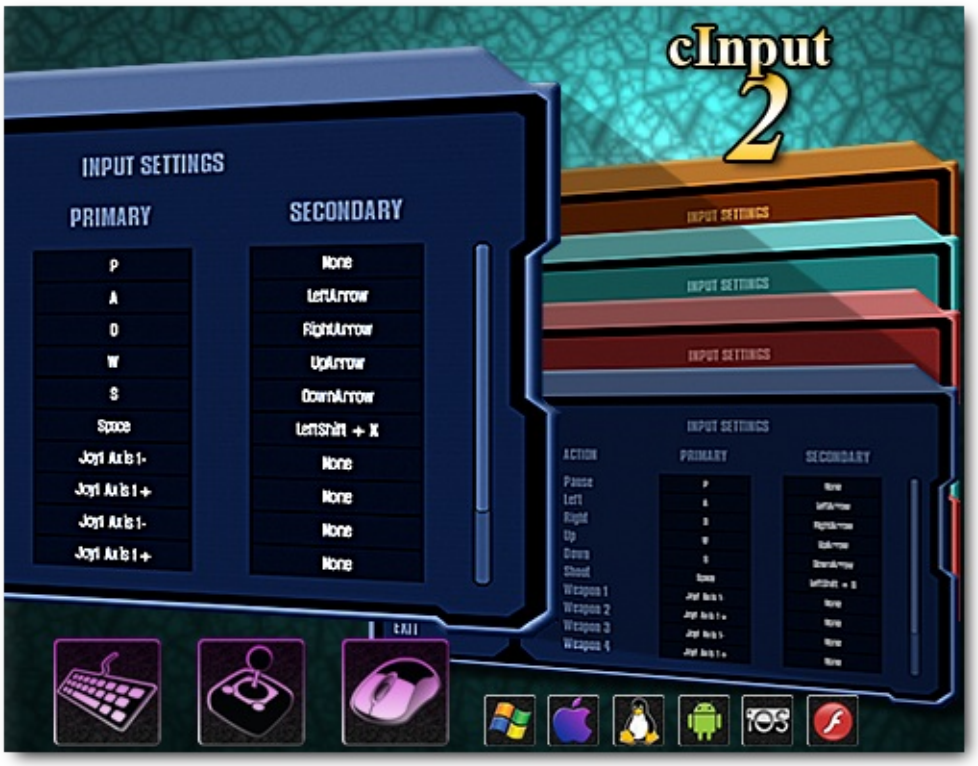
Table of Contents

cInput Reference Manual



Note

cInput 2 is a commercial release and is not free.

If you're reading this manual then most likely you have already purchased a license for cInput. If you don't already have a license, you can get cInput 2 from our website or from the Unity Asset Store.

We tried to keep the price low so everyone can still afford it. If for some reason you cannot afford the price you can use Custom InputManager 1.x which still works fine.

If you really need to use cInput 2 and can't afford the price then contact us to explain your needs and we'll see if we can work something out.

Using cInput

This section of the manual focuses on how to use cInput in your project. It covers the following topics:

Initial Setup

Most of the setup is done for you if you import the `.unitypackage` file or import cInput from the Asset Store. Nevertheless, cInput should only take--at most--the following number of steps to be ready for use in your project:

1. Use the `Edit -> Project Settings -> cInput -> Setup InputManager Asset` menu command in the Unity Editor to create an `InputManager.asset` file designed to work with cInput.
2. Place `cInput.cs` somewhere in your `Assets/Plugins` folder (or a subfolder of that directory) and it will automatically be accessible from your other scripts. *This step should happen automatically if you imported the `.unitypackage` file.*
3. cInput comes with some optional helper classes which can make it easier to do certain things, such as the `Keys` class which gives you autocomplete on key names. These should also go somewhere within the `Assets/Plugins` folder. *Again, this should happen automatically if you imported the `.unitypackage` file.*

That's it! You're now ready to start using cInput in your project.

A Brief Explanation of cInput

Before we get to the nuts and bolts of defining the default inputs, it may be useful to clarify the mindset behind cInput and how it's intended to function behind-the-scenes.

Action

cInput primarily works with `actions` which are generally named after what you intend to use an input *to do*. Some examples of action names could be `"Jump"`, `"Shoot"`, `"Move Left"`, or `"Move Right"`. That said, `actions` can be named anything you want.

For each action, you can assign up to two inputs which cInput will use to get the values of the action. For example, the `"Move Left"` action could have the `A` and `Left Arrow` keys assigned to it.

The word `key` (or `keys`) in this manual will most often refer to a key on the keyboard. However, because cInput attempts to mimic Unity's Input class as closely as possible, sometimes the word `key` is used when referring to `actions` (e.g., the `GetKey()` or `ChangeKey()` methods).

Additionally, the word `button` (or `buttons`) usually refers to a button on a gamepad. However, sometimes it is easier to use the word `button` to refer to any inputs assigned to the `action` itself, regardless of what types of inputs they may be. In this situation, the word `button` is usually preceded with the name of the action, which will be in quotes. For example, the "Jump" button.

Axis

An `axis` in cInput can be made up from one or two `actions`. If only one `action` is assigned to an `axis`, the `axis` will return a positive value between `0...1`. If two `actions` are assigned to an `axis` then the first `action` will give the negative values (`-1...0`) for the `axis`, while the second `action` will give the positive values (`0...1`) for the `axis`.

Because `axes` are made up from `actions`, you must first define the `actions` for each `axis` before you can define the `axis` itself.

If the inputs used for an axis are digital inputs--meaning they have only two states: on or off (`0` or `1`)--such as keyboard keys or most gamepad buttons, cInput will create a virtual (i.e. simulated) analog axis. In other words, instead of returning only integer values (`-1`,

`0`, or `1`), `GetAxis()` will return a float value in the range of `-1...1` inclusive. If this is undesirable and you want the values exactly as they actually are, use `GetAxisRaw()` instead.

Setting Up the Default Inputs

The first thing to do is to create a setup script which will run once *before* any of your other scripts try to get input from cInput. Note that this setup script only needs to run once, so if you put it in the first scene the player will see (e.g., a splash screen or loading screen) then you don't need to include it in any other scenes. cInput will persist when changing scenes.

It is considered best practice to call `Init()` in your `Awake()` function to allow cInput to instantiate a `GameObject` and load any customized settings. After that, you can define all of your default settings in the `Start()` (or other) function, starting with defining `actions` with `SetKey()` followed by any `axes` with `SetAxis()`, similar to what's shown below:

```
void Awake() {
  // initialize cInput
  cInput.Init();
}

void Start() {
  // this creates a new action called "Left" and binds A and Left Arrow to it
  cInput.SetKey("Left", "A", "LeftArrow");
  // this uses Keys.D and Keys.RightArrow to accomplish something similar
  cInput.SetKey("Right", Keys.D, Keys.RightArrow);
  // creates a new axis using "Left" and "Right" actions
  cInput.SetAxis("Horizontal Movement", "Left", "Right");
}
```

More details on how to use `SetKey()` or `SetAxis()` can be found in the Script Reference.

> **Remember: An `axis` cannot be defined until *after* its `actions` have been defined.**

For a full list of acceptable inputs (as strings) see the Valid Inputs section. Additionally, you can use the included `Keys` class to let autocomplete help you type in the correct keys. Many of the examples in this documentation take advantage of the `Keys` class.

The Keys Class

The `Keys` class was born from the frustration of not being able to remember the strings for various keyboard or gamepad inputs as well as an attempt to prevent errors due to typos. Its primary purpose is to enable autocomplete for the various cInput functions which require the names of inputs as strings.

> "What's the string for the `#` symbol? I can't remember if it's `NumberSign`, `Hash`, or `Pound`."

The `Keys` class has equivalents of all strings listed in Valid Inputs. It also attempts to make it easier to configure Xbox controllers for your game.

While this documentation makes heavy use of the Keys class, and recommends you do as well, its use is completely optional.

> Tip: The `Keys` class can be found in `cKeys.cs` if you have access to the source code (**cInput Pro only**).

A Note About Xbox Controllers

Xbox 360 controllers have the unfortunate "feature" of having both triggers being mapped to the same axis on Windows. This means that if you push in both triggers at the same time, they cancel each other out and Unity can't tell that you're pushing either trigger. Thankfully, the triggers are also (sometimes) mapped separately on different axes, though the driver may need to be installed on Windows 7 or older for this to work.

Even so, there are two further complications:

Multiple Xbox Controllers

If more than one Xbox 360 or Xbox One controller is connected, the triggers may or may not work properly, or may or may not be mapped to these separate axes as previously described. **This is an issue with Unity itself**, not a flaw in cInput. Recent versions of Unity have taken steps to correct this issue, but some problems may still exist. More information about this problem and how it has affected other custom input managers for Unity can be found here: Details on the Xbox 360 Controller Bug in Unity.

Different Gamepad Mappings Per OS

Another complication with using Xbox 360 or Xbox One controllers is that the buttons and axes are mapped differently depending on the OS. In some cases, the Xbox One controller has different mappings from the Xbox 360 controller *on the same OS*! This makes it difficult to set up a single default configuration which *just works* for various OSes and platforms.

The `Keys` class *attempts* to assist with this, by providing special entries specifically for the Xbox buttons and joysticks for up to four controllers. These entries follow a similar format as follows:

```
Xbox1A
Xbox1B
Xbox1X
Xbox1Y
Xbox1Back or Xbox1View
Xbox1Start or Xbox1Menu
Xbox1LStickButton
Xbox1RStickButton

Xbox1BumperLeft
Xbox1BumperRight
Xbox1TriggerLeft
Xbox1TriggerRight
Xbox1DPadLeft
Xbox1DPadRight
Xbox1DPadUp
Xbox1DPadDown

Xbox1LStickLeft
Xbox1LStickRight
Xbox1LStickUp
Xbox1LStickDown
Xbox1RStickLeft
Xbox1RStickRight
Xbox1RStickUp
Xbox1RStickDown
```

For the second, third, or fourth Xbox controllers, simply replace the `Xbox1` part at the beginning of the above entries with `Xbox2`, `Xbox3`, or `Xbox4` respectively.

Additionally, if you want to accept input from any Xbox controller, simply leave the number out entirely. For example, using `Keys.XboxA` will look for input from any Xbox controller's `A` button, without regard to the order in which the controllers are connected. This is very useful for single player games where you would like the player to be able to pick up any controller and play. But be warned that this can cause conflicts in multiplayer games where two players' inputs would interfere with each other.

Keep in mind that

Using Modifier Keys

cInput supports the use of modifier keys. You can designate a key to be used as a modifier with `AddModifier()`. Note that once a key has been designated as a modifier, it is forbidden from being used as an input all by itself. For example, if you designate `LeftShift` as a modifier, then it can be used in multiple key combinations such as `LeftShift+T` and `LeftShift+1`, but it cannot be used alone as just `LeftShift`. To define an action to use a modifier key by default, create it in your setup script with `SetKey()`, passing in the modifiers you want to use.

Modifier keys do not work with axes! If you use `SetKey()` to define an action that uses a modifier, and then use that action with `SetAxis()` to create an axis, the axis won't care if the modifier is pressed or not.

For example, say you have defined an action named `"Assign Group 1"` which uses `LeftControl+Alpha1` (Ctrl+1) as the keys to activate it. If you then create an axis which uses `"Assign Group 1"` as one of its inputs, then simply pressing `1` on your keyboard will activate the axis whether or not the `LeftControl` key is being pressed.

If you have added a key as a modifier and would like to remove it from being used as a modifier, use `RemoveModifier()`.

Using cInput to Control Your Game

Once your inputs are defined in your setup script, you'll probably want to use those inputs to control your game. This is very simple to do using cInput's `GetKey()`, `GetKeyDown()`, `GetKeyUp()`, and `GetAxis()` functions, which all work in the same way as Unity's Input class functions of the same name.

For example:

```
// this will only trigger once each time the "Jump" button is pressed
if (cInput.GetKeyDown("Jump")) {
  // jumping code here
}

// this will trigger repeatedly while the "Shoot" button is held down
if (cInput.GetKey("Shoot")) {
  // shooting code here
}

// this will only trigger once each time the "Grenade" button is released
if (cInput.GetKeyUp("Grenade")) {
  // throw grenade code here
}

// movement based on analog or virtual analog axis
float dx = cInput.GetAxis("Horizontal");
```

If the above code seems familiar, it should! One of the main tenets of cInput was to make its syntax as similar to Unity's `Input` class as possible.

Changing Inputs from the Defaults

One of the headlining features of cInput is that it enables you to change key bindings at runtime, a feature sorely missing from Unity

Changing Inputs via Script

For information on how to change key bindings from script, see the `ChangeKey()` documentation.

Using the Built-In GUI

cInput comes with a built-in GUI to make it simple to change which inputs are bound to which actions. This is great for rapid prototyping since it allows you to adjust the controls at runtime without having to invest a lot of time into a custom GUI.

Due to popular demand, we've separated this built-in GUI from the main `cInput` class into it's own class called `cGUI`. Simply call `cGUI.ToggleGUI()` to show or hide the built-in GUI menu, and assign a GUISkin to `cGUI.cSkin` to change the way it looks. cInput includes a GUISkin for your convenience. You can also optionally change the color/alpha of the GUI using `cGUI.bgColor`. For more specific information on how to use cGUI, please see the cGUI Reference Manual included in the `.unitypackage`.

Making a Custom GUI Menu

Admittedly, even though you can customize the look of the built-in GUI with `cGUI.cSkin`, it still won't fit the design or theme for every game. In situations like these where you're past the prototyping phase and getting closer to public release, it will likely be necessary (not to mention recommended) to replace the built-in GUI with something that fits the style of the rest of your game. With that in mind, cInput includes functions which will assist in the process of making your own custom GUI menu for changing controls.

A quick note: Explaining how to make a UI in Unity is beyond the scope of this document. To learn more about how to make GUIs in Unity, see the UI System or the Legacy GUI Scripting Guide sections in the online Unity Reference Manual. This section only explains the methods which cInput provides to assist you in making your own UI menu for displaying

and changing the key bindings in your game. Also note that if you make your own UI, you can prevent cInput from adding its own included GUI components by using `cInput.Init(false)` in your setup script.

There are a few methods and properties you will likely want to use when creating a UI to display the current key bindings and allow the players to change them or reset them back to defaults. They are the `GetText()`, `ChangeKey()`, `ResetInputs()` functions and the `length` property. You may also be interested in the `OnKeyChanged` event.

The basic idea is to use a `for` loop to iterate through all of the inputs and create buttons the player can click in order to change the key bindings while in game. Here is an example with Unity's legacy GUI:

```
// this variable keeps track of the vertical scroll position (if necessary)
private Vector2 _scrollPosition = new Vector2();
void OnGUI() {
  _scrollPosition = GUILayout.BeginScrollView(_scrollPosition);
  GUILayout.BeginHorizontal();

  // this sections lists the names of each action, such as "Move Left" or "Jump"
  GUILayout.BeginVertical();
  GUILayout.Label("Action");
  for (int n = 0; n < cInput.length; n++) {
    GUILayout.Label(cInput.GetText(n, 0));
  }
  GUILayout.EndVertical();

  // this sections lists the primary input for the actions, such as "A" or "Space"
  GUILayout.BeginVertical();
  GUILayout.Label("Primary");
  for (int n = 0; n < cInput.length; n++) {
    if (GUILayout.Button(cInput.GetText(n, 1)) && Input.GetMouseButtonUp(0)) {
      cInput.ChangeKey(n, 1);
    }
  }
  GUILayout.EndVertical();

  // this section lists the secondary input for the actions, such as "Left Arrow"
  GUILayout.BeginVertical();
  GUILayout.Label("Secondary");
  for (int n = 0; n < cInput.length; n++) {
    if (GUILayout.Button(cInput.GetText(n, 2)) && Input.GetMouseButtonUp(0)) {
      cInput.ChangeKey(n, 2);
    }
  }
  GUILayout.EndVertical();

  GUILayout.EndHorizontal();
  GUILayout.EndScrollView();

  // this gives the player a button they can click to reset the inputs to the defaults
.
  GUILayout.BeginHorizontal();
  if (GUILayout.Button("Reset to Defaults") && Input.GetMouseButtonUp(0)) {
    cInput.ResetInputs();
  }

  // this shows a button to close the menu
  if (GUILayout.Button("Close") && Input.GetMouseButtonUp(0)) {
    cInput.ShowMenu(false);
  }
  GUILayout.EndHorizontal();
}
```

The OnKeyChanged Event

cInput calls an event whenever either the `ChangeKey()` function which waits for player input, or the `ResetInputs()` function is called. Note that it will still be called in cases where the keys stay the same (such as from canceling a `ChangeKey()` action). An example for why you might want to use the OnKeyChanged event would be to run a function to update the text labels in a custom UI.

Here's some code showing one such example:

```
// subscribe to the OnKeyChanged event
void OnEnable() {
  cInput.OnKeyChanged += UpdateGUITexts;
}

// unsubscribe to the OnKeyChanged event so we don't cause errors
void OnDisable() {
  cInput.OnKeyChanged -= UpdateGUITexts;
}

void UpdateGUITexts() {
  // put your code here to update the GUI texts when the OnKeyChanged event is fired
}
```

What's an Event?

If events are completely new to you, you may find Unity's training video on Events helpful.

> **Embedded video: https://www.youtube.com/watch?v=6qyR73EO68w**

General Tips and Tricks

Use Clear() to Fix Misconfigurations

If you encounter odd behavior while developing with cInput, it may be helpful to make a call to `Clear()` to clear out any misconfigured cInput settings. **But be careful with this!** If you accidentally leave this in your release builds that make it to your players, it will erase any player-customized settings. They probably won't be too happy about that.

Script Execution Order

cInput should execute before other scripts to make sure that all inputs are updated for the current frame before your other scripts to to access them. By default, this should automatically be handled for you when you import the `.unitypackage` file, but it's something to be aware of if the script execution order is somehow lost or modified.

Script Reference

This section of the manual is a reference of the `cInput` class and its public properties and methods, as well as a reference to the valid strings you can use for inputs if you choose not to use the `Keys` class.

Variables and Properties

This section of the manual is an alphabetical list of all the public properties of the `cInput` class, along with a description of them and of how to use them.

allowDuplicates

bool **allowDuplicates**

Description

If set to true, clnput will allow the same inputs to be used for multiple actions. Default value is `false`.

anyKey

bool **anyKey**

Description

Returns `true` if any key, mouse button, or action defined with SetKey() is currently held down. **Read-only**.

anyKeyDown

bool **anyKeyDown**

Description

Returns `true` the first frame the user presses any key, mouse button, or action defined with SetKey(). **Read-only**.

deadzone

float **deadzone**

Description

Values less than this will register as `0` on the virtual axis. Default value is `0.001f`.

Can be overridden on a per-axis basis with `SetAxis()` or `SetAxisDeadzone()`.

externalInputs

string **externalInputs**

Description

A string containing all the cInput settings. **Read-only**.

Use this with `LoadExternal()` to save and load cInput settings somewhere other than PlayerPrefs. See also `usePlayerPrefs`.

gravity

float **gravity**

Description

How fast the virtual axis value will return to `0`. Default value is `3.0f`.

Can be overridden on a per-axis basis with `SetAxis()` or `SetAxisGravity()`.

length

int **length**

Description

How many `actions` have been defined using `SetKey()`. **Read-only**.

Useful for making a custom GUI menu.

scanning

bool **scanning**

Description

Whether or not cInput is currently scanning for a keypress/input to bind to an action.

Useful for making a custom GUI menu.

sensitivity

float **sensitivity**

Description

How fast the virtual axis value will reach `1`. Default value is `3.0f`.

Can be overridden on a per-axis basis with `SetAxis()` or `SetAxisSensitivity()`.

usePlayerPrefs

bool **usePlayerPrefs**

Description

Should cInput automatically save and load to/from PlayerPrefs? Default value is `true`.

Methods

This section of the manual is an alphabetical list of all the public methods of the `cInput` class, along with a description of them and of how to use them.

AddModifier

void **AddModifier**(*KeyCode* **modifierKey**)
void **AddModifier**(*string* **modifier**)

Parameters

modifierKey: The `KeyCode` of the key to be used as a modifier.
modifier: The string name of the key to be used as a modifier. Allows you to use the `Keys` class.

Description

Designates `modifier` or `modifierKey` to be used as a modifier. **Note that a modifier key cannot be used as a standalone input key**. See Using Modifier Keys for more details.

Example

```
// allows LeftShift to be used as a modifier key
cInput.AddModifier(KeyCode.LeftShift);
// allows LeftControl to be used as a modifier key
cInput.AddModifier("LeftControl");
```

AxisInverted

bool **AxisInverted**(*string* **description**[, *bool* **inversionStatus**])
bool **AxisInverted**(*int* **descriptionHash**[, *bool* **inversionStatus**])

Parameters

description: The name/description of the axis you want to invert.
descriptionHash: The hashcode of the name/description of the axis you want to invert.
inversionStatus: If true, axis will be inverted. If false, axis will not be inverted.

Returns

bool: Whether or not the axis of `description` or `descriptionHash` is inverted.

Description

If `inversionStatus` is not passed in, `AxisInverted()` will simply return the current inversion status of the axis. If `inversionStatus` is passed in, then the inversion status of this axis will be set to the value of `inversionStatus`.

Example

```
// this toggles the inversion status of "Horizontal"
cInput.AxisInverted("Horizontal", !cInput.AxisInverted("Horizontal"));
// this makes a toggle button in the GUI
cInput.AxisInverted("Horizontal", GUILayout.Toggle(cInput.AxisInverted("Horizontal"),
"Invert Axis"));
```

Calibrate

void **Calibrate()**

Description

Calibrates analog inputs to their default/neutral position.

ChangeKey

void **ChangeKey**(*string* **name**, [*int* **input**], [*bool* **mouseAx**], [*bool* **mouseBut**], [*bool* **joyAx**], [*bool* **joyBut**], [*bool* **keyb**])
void **ChangeKey**(*int* **index**, [*int* **input**], [*bool* **mouseAx**], [*bool* **mouseBut**], [*bool* **joyAx**], [*bool* **joyBut**], [*bool* **keyb**])
void **ChangeKey**(*string* **name**, [*string* **primary**], [*string* **secondary**], [*string* **primaryModifier**], [*string* **secondaryModifier**])

Parameters

name: The name/description of the `action` you wish to change.
index: The index number of the `action` you wish to change. Useful for custom GUIs.
input: Which input to change. 1 for primary. 2 for secondary. Defaults to `1`.
mouseAx: Allow mouse axes for input? Defaults to `false`.
mouseBut: Allow mouse buttons for input? Defaults to `true`.
joyAx: Allow gamepad axes (analog sticks/buttons) for input? Defaults to `true`.
joyBut: Allow gamepad buttons for input? Defaults to `true`.
keyb: Allow keyboard buttons for input? Defaults to `true`.

Description

Waits for player input, then assigns that input to trigger this `action`. All inputs are monitored by default except mouse axes. All arguments except `name` or `index` are optional and will use their default values if not explicitly passed in.

Note that you can also use `ChangeKey()` in the same way as `SetKey()` to change an action immediately via script instead of waiting for player input.

Example

```
// the next input pressed will be assigned as the primary input for "Pause"
cInput.ChangeKey("Pause");
cInput.ChangeKey("Pause", 1); // this does the same thing as the previous line
// the next input pressed will be assigned as the secondary input for "Pause"
cInput.ChangeKey("Pause", 2);
// only gamepad axes and buttons will be accepted for the primary "Accelerate" input
cInput.ChangeKey("Accelerate", false, false, true, true, false);
// only the keyboard can be used for the secondary "Jump" input
cInput.ChangeKey("Jump", 2, false, false, false, false, true);
// using ChangeKey like SetKey to change the primary and secondary inputs for "Up"
cInput.ChangeKey("Up", "W", Keys.UpArrow);
```

Clear

void **Clear**()

Description

Clears all data stored by cInput from PlayerPrefs. This can fix problems which may occur during development when changing the number or order of inputs used by cInput.

ForbidAxis

void **ForbidAxis**(*string* **axis**)

Parameters

axis: A string representing the axis to forbid. See Gamepad Inputs for valid parameters.

Description

Forbids the axis from being bound (or re-bound) by the player using cInput's UI.

Example

```
// forbid the player from binding Joy1 Axis 6- using the UI
cInput.ForbidAxis(Keys.Joy1Axis6Negative);
```

ForbidKey

void **ForbidKey**(*KeyCode* **key**)
void **ForbidKey**(*string* **keyString**)

Parameters

key: The `KeyCode` of the key to forbid.
keyString: A string representing the `KeyCode` of the key to forbid. See Valid Inputs for valid strings.

Description

Forbids the key from being bound (or re-bound) by the player using cInput's UI.

Example

```
// forbid the player from binding the number 1 using the UI
cInput.ForbidKey(KeyCode.Alpha1);
// forbid the player frmo binding "tab" and "space" using the UI
cInput.ForbidKey(Keys.Tab);
cInput.ForbidKey("Space");
```

GetAxis

float **GetAxis**(*string* **description**)
float **GetAxis**(*int* **descriptionHash**)

Parameters

description: The name of the axis, as defined in `SetAxis()`.
descriptionHash: The hashcode of the name of the axis, as defined in `SetAxis()`.

Returns

float: A value between `-1` and `1` inclusive.

Description

Returns the value of the axis.

Example

```
// move the transform horizontally with the "Horizontal Movement" axis
float horizMovement = cInput.GetAxis("Horizontal Movement");
float h = 60 * horizMovement * Time.deltaTime;
transform.Translate(h, 0, 0);
```

GetAxisDeadzone

float **GetAxisDeadzone**(*string* **description**)
float **GetAxisDeadzone**(*int* **descriptionHash**)

Parameters

description: The name of the axis, as defined in `SetAxis()`.
descriptionHash: The hashcode of the name of the axis, as defined in `SetAxis()`.

Returns

float: The `deadzone` value for the axis.

Description

Returns the `deadzone` value for the axis. You can change this value with `SetAxisDeadzone()`.

This is provided as a way to present the player with a UI to display and change the `deadzone` settings.

Example

```
// gets the deadzone value for a previously define axis
float hDeadzone = cInput.GetAxisDeadzone("Horizontal Movement");
```

GetAxisGravity

float **GetAxisGravity**(*string* **description**)
float **GetAxisGravity**(*int* **descriptionHash**)

Parameters

description: The name of the axis, as defined in `SetAxis()`.
descriptionHash: The hashcode of the name of the axis, as defined in `SetAxis()`.

Returns

float: The `gravity` value for the axis.

Description

Returns the `gravity` value for the axis. You can change this value with `SetAxisGravity()`.

This is provided as a way to present the player with a UI to display and change the `gravity` settings.

Example

```
// gets the gravity value for a previously define axis
float hGravity = cInput.GetAxisGravity("Horizontal Movement");
```

GetAxisRaw

float **GetAxisRaw**(*string* **description**)
float **GetAxisRaw**(*int* **descriptionHash**)

Parameters

description: The name of the axis, as defined in `SetAxis()`.
descriptionHash: The hashcode of the name of the axis, as defined in `SetAxis()`.

Returns

float: A value between `-1` and `1` inclusive.

Description

Returns the value of the virtual axis identified by `description` or `descriptionHash` with no smoothing applied. In other words, `sensitivity` and `gravity` have no effect on this value.

The value will be in the range `-1...1` for an axis made from analog (e.g., joysticks or gamepad triggers) inputs. Since input is not smoothed, an axis made from digital inputs (e.g., keyboard or gamepad buttons) will always be either `-1`, `0`, or `1`.

This is useful if you want to get the value of an axis before `sensitivity` or `gravity` smoothing is applied to it.

Example

```
// move the transform horizontally with the "Horizontal Movement" axis
transform.Translate(cInput.GetAxisRaw("Horizontal Movement"), 0, 0);
```

GetAxisSensitivity

float **GetAxisSensitivity**(*string* **description**)
float **GetAxisSensitivity**(*int* **descriptionHash**)

Parameters

description: The name of the axis, as defined in `SetAxis()`.
descriptionHash: The hashcode of the name of the axis, as defined in `SetAxis()`.

Returns

float: The `sensitivity` value for the axis.

Description

Returns the `sensitivity` value for the axis. You can change this value with `SetAxisSensitivity()`.

This is provided as a way to present the player with a UI to display and change the `sensitivity` settings.

Example

```
// gets the sensitivity value of a previously defined axis
float hSensitivity = cInput.GetAxisSensitivity("Horizontal Movement");
```

GetButton

Description

This works in exactly the same way as `GetKey()`.

GetButtonDown

Description

This works in exactly the same way as `GetKeyDown()`.

GetButtonUp

Description

This works in exactly the same way as `GetKeyUp()`.

GetKey

bool **GetKey**(*string* **description**)
bool **GetKey**(*int* **descriptionHash**)

Parameters

description: The name of the key, as defined in `SetKey()`.
descriptionHash: The hashcode of the name of the key, as defined in `SetKey()`.

Returns

bool: Returns `true` if the key is being held down.

Description

Use this to determine if a key is being held down. `GetKey()` returns `true` **repeatedly** while the user holds down the key, and returns `false` if the key is not being pressed.

The use of `GetKeyDown()` or `GetKeyUp()` is recommended if you want to trigger an event only once per keypress, e.g., for jumping.

Example

```
// prints the message repeatedly, as long the player keeps pressing the "Shoot" input

if (cInput.GetKey("Shoot")) {
  Debug.Log("The player is shooting.");
}
```

GetKeyDown

bool **GetKeyDown**(*string* **description**)
bool **GetKeyDown**(*int* **descriptionHash**)

Parameters

description: The name of the key, as defined in `SetKey()`.
descriptionHash: The hashcode of the name of the key, as defined in `SetKey()`.

Returns

bool: Returns `true` only once each time the key is first pressed down.

Description

Use this to determine if a key has been pressed during this frame. `GetKeyDown()` returns `true` *only once* when the key is first pressed down.

The use of `GetKey()` is recommended if you want to trigger an event repeatedly while the key is being held down, e.g., for continuous movement.

Example

```
// prints the message just once when the player starts pressing the key
if (cInput.GetKeyDown("Jump") {
  Debug.Log("You pressed the jump button!");
}
```

GetKeyUp

bool **GetKeyUp**(*string* **description**)
bool **GetKeyUp**(*int* **descriptionHash**)

Parameters

description: The name of the key, as defined in `SetKey()`.
descriptionHash: The hashcode of the name of the key, as defined in `SetKey()`.

Returns

bool: Returns `true` only once each time the key is released.

Description

Use this to determine if a key has been released during this frame. `GetKeyUp()` returns `true` *only once* when the key is first released.

The use of `GetKey()` is recommended if you want to trigger an event repeatedly while the key is being held down, e.g., for continuous movement.

Example

```
// prints the message just once when the player releases the key.
if (cInput.GetKeyUp("Jump") {
  Debug.Log("You released the jump button!");
}
```

GetText

string **GetText**(string **description**, [*int* **input**], [*bool* **returnBlank**])
string **GetText**(*int* **index**, [*int* **input**], [*bool* **returnBlank**])

Parameters

description: The name/description of the `action` you want to get the text for.
index: The index number of the `action` you want to get the text for.
input: Which input you want to get the text for.
returnBlank: Whether or not `"None"` should return an empty string. Defaults to `false`.

Description

Returns the text of the input used for the `action`. This is useful for making custom UIs.

Possible valid values for `input` are `0`, `1`, and `2`.

- Passing in `0` will return the `description` of the `action`.
- Passing in `1` will return the name of the primary input assigned to the `action`.
- Passing in `2` will return the name of the secondary input assigned to the `action`.

Note that `input` is optional, and if omitted will default to `0` if you use the `index` as the first parameter of `GetText()`. It will default to `1` if you use the `description` as the first parameter of `GetText()`.

If no input is assigned to the `action` (as may often be the case with secondary inputs), then `GetText()` will return `"None"` by default, or it will return an empty string if `returnBlank` is `true`.

See the Making a Custom GUI Menu section for more details on how and why you might want to use `GetText()`.

Example

```
cInput.SetKey("Shoot", Keys.LeftControl, Keys.RightControl); // index for this key is
0 in this example
Debug.Log(cInput.GetText("Shoot")); // displays "LeftControl"
Debug.Log(cInput.GetText("Shoot", 1); // also displays "LeftControl"
Debug.Log(cInput.GetText("Shoot", 2); // displays "RightControl"
Debug.Log(cInput.GetText("Shoot", 0)); // displays "Shoot"
Debug.Log(cInput.GetText(0)); // displays "Shoot"
Debug.Log(cInput.GetText(0, 0)); // also displays "Shoot"
Debug.Log(cInput.GetText(0, 1); // displays "LeftControl"
Debug.Log(cInput.GetText(0, 2); // displays "RightControl"
cInput.SetKey("Jump", Keys.Space); // Note that no secondary input is assigned to this
 key
Debug.Log(cInput.GetText("Jump", 1); // displays "Space"
Debug.Log(cInput.GetText("Jump", 2); // displays "None"
Debug.Log(cInput.GetText("Jump", 2, true); // displays "" (an empty string)
```

Init

void **Init**([*bool* **useGUI**])

Parameters

useGUI: Whether or not to use cInput's built-in GUI. Defaults to `true`.

Description

Use this to manually initialize the cInput object.

cInput will attempt to create the cInput object automatically upon first use, but there are some situations which might require you to manually call this method first. Thus it is considered best practice to call `Init()` at the beginning of your setup script.

Example

```
// no need to pass in a parameter
cInput.Init();
// if you're using the [UI system as of Unity 4.6+](http://docs.unity3d.com/Manual/UIS
ystem.html), you don't need to use cGUI.
cInput.Init(false);
```

IsAxisDefined

bool **IsAxisDefined**(*string* **axisName**)
bool **IsAxisDefined**(*int* **axisHash**)

Parameters

axisName: The name of the axis, as defined in `SetAxis()`.
axisHash: The hashcode of the name of the axis, as defined in `SetAxis()`.

Returns

bool: Returns `true` if `axisName` or `axisHash` exists.

Description

Use this to determine if an axis exists by the name of `axisName` or the hashcode of `axisHash`.

Note that you will probably never need to use this method unless you are a developer making a separate script/plugin and you want to make it compatible with cInput.

Example

```
// make sure user has defined this axis
if (cInput.IsAxisDefined("Horizontal")) {
  // do cInput stuff
} else {
  // fallback to non-cInput stuff?
}
```

IsKeyDefined

bool **IsKeyDefined**(*string* **keyName**)
bool **IsKeyDefined**(*int* **keyHash**)

Parameters

keyName: The name of the key (`action`), as defined in `SetKey()` .
keyHash: The hashcode of the name of the key (`action`), as defined in `SetKey()` .

Returns

bool: Returns `true` if `keyName` or `keyHash` exists.

Description

Use this to determine if a key exists by the name of `keyName` or the hashcode of `keyHash` .

Note that you will probably never need to use this method unless you are a developer making a separate script/plugin and you want to make it compatible with cInput.

Example

```
// make sure user has defined this key
if (cInput.IsKeyDefined("Left")) {
  // do cInput stuff
} else {
  // fallback to non-cInput stuff?
}
```

LoadExternal

void **LoadExternal**(*string* **externString**)

Parameters

externString: A string containing all of the cInput settings.

Description

Use this to load cInput settings from some source other than PlayerPrefs. This is used in conjunction with `externalInputs`. See also `usePlayerPrefs`.

Example

```
// loads the cInput settings from an external text file
string external = System.IO.File.ReadAllText(Application.dataPath + "/settings.cInput"
);
cInput.LoadExternal(external);
```

RemoveModifier

void **RemoveModifier**(*KeyCode* **modifierKey**)
void **RemoveModifier**(*string* **modifier**)

Parameters

modifierKey: The KeyCode of the key to stop using as a modifier.
modifier: The string name of the key to stop using as a modifier.

Description

Removes `modifier` or `modifierKey` from being used as a modifier.

Note that a modifier key cannot be used as a standalone input key. See Using Modifier Keys for more details. This function allows the key to be used again for normal inputs.

Example

```
// allows LeftShift to be used as a normal input key and prevents it from being used a
s a modifier key
cInput.RemoveModifier(KeyCode.LeftShift);
```

ResetInputs

void **ResetInputs**()

Description

Resets all controls back to the defaults as defined by `SetKey()` and `SetAxis()`.

For more information about defaults, see setting up the default inputs.

Example

```
// reset the inputs back to default
cInput.ResetInputs();
```

SetAxis

int **SetAxis**(*string* **description**, *string* **negative**, *string* **positive**, [*float* **sensitivity**], [*float* **gravity**], [*float* **deadzone**])
int **SetAxis**(*string* **description**, *string* **action**, [*float* **sensitivity**], [*float* **gravity**], [*float* **deadzone**])

Parameters

description: The description of what the axis is used for.
negative: The `action` which provides the negative value of the axis.
positive/action: The `action` which provides the positive value of the axis.
sensitivity: *Optional*. The sensitivity for the axis. Defaults to `3`.
gravity: *Optional*. The gravity for the axis. Defaults to `3`.
deadzone: *Optional*. Input values below this level are ignored. Defaults to `0.001`.

Returns

int: The hashcode of `description`.

Description

Creates an axis out of **one or two `actions`, which must have been set previously** with `SetKey()`. If two `actions` are passed in, the first (`negative`) will be used to provide the negative values while the second (`positive`) will provide the positive values for the axis. If only one `action` is passed in (`action`) then the axis will only return positive values (unless inverted).

You can optionally override the default sensitivity, gravity, or deadzone of the axis by passing in `sensitivity` , `gravity` , and `deadzone` respectively. Additionally, you can use `SetAxisSensitivity()` , `SetAxisGravity()` , and `SetAxisDeadzone()` if you want to change these values for an axis after it has already been created.

Also note that setting up axes is not required and you should only do this if you require analog-like controls instead of digital controls. An axis input can be analog (e.g., joystick) or digital (e.g., keyboard button) or even a combination of the two. A virtual analog axis will be created if necessary. More details can be found in A Brief Explanation of cInput.

Example

```
// uses "Left" and "Right" actions previously defined with SetKey to create a new axis

cInput.SetAxis("Horizontal Movement", "Left", "Right");
// or the same as above but with increased sensitivity
cInput.SetAxis("Horizontal Movement", "Left", "Right", 4.5f);
// uses only a single input as an axis (e.g., for a gas pedal in a driving game)
cInput.SetAxis("Acceleration", "Gas");
// or the same as above but with decreased sensitivity
cInput.SetAxis("Acceleration", "Gas", 1.5f);
```

SetAxisDeadzone

void **SetAxisDeadzone**(*string* **axisName**, *float* **deadzone**)
void **SetAxisDeadzone**(*int* **axisHash**, *float* **deadzone**)

Parameters

axisName: The name of the axis you want to change the deadzone value for.
axisHash: The hashcode of the name of the axis you want to change the deadzone value for.
deadzone: The value to assign as the deadzone.

Description

Sets the deadzone for the axis. This will override cInput's global `deadzone` value.

This value can also be set during the initial axis creation using `SetAxis()`.

Example

```
// sets the deadzone for a previously defined axis
cInput.SetAxisDeadzone("Horizontal Movement", 0.1f);
```

SetAxisGravity

void **SetAxisGravity**(*string* **axisName**, *float* **gravity**)
void **SetAxisGravity**(*int* **axisHash**, *float* **gravity**)

Parameters

axisName: The name of the axis you want to change the gravity value for.
axisHash: The hashcode of the name of the axis you want to change the gravity value for.
gravity: The value to assign as the gravity.

Description

Sets the gravity for the axis. This will override cInput's global `gravity` value.

This value can also be set during the initial axis creation using `SetAxis()`.

Example

```
// sets the gravity of a previously defined axis
cInput.SetAxisGravity("Horizontal Movement", 0.3f);
```

SetAxisSensitivity

void **SetAxisSensitivity**(*string* **axisName**, *float* **sensitivity**)
void **SetAxisSensitivity**(*int* **axisHash**, *float* **sensitivity**)

Parameters

axisName: The name of the axis you want to change the sensitivity value for.
axisHash: The hashcode of the name of the axis you want to change the sensitivity value for.
sensitivity: The value to assign as the sensitivity.

Description

Sets the sensitivity for the axis. This will override cInput's global `sensitivity` value.

This value can also be set during the initial axis creation using `SetAxis()`.

Example

```
// sets the sensitivity of a previously defined axis
cInput.SetAxisSensitivity("Horizontal Movement", 0.5f);
```

SetKey

void **SetKey**(*string* **description**, *string* **primary**, [*string* **secondary**], [*string* **primaryModifier**], [*string* **secondaryModifier**])

Parameters

description: The description of what the key (`action`) is used for.
primary: The primary input to be used for this `action` .
secondary: *Optional*. The secondary input to be used for this `action` . Defaults to `None` .
primaryModifier: *Optional*. The modifier key to be used for the primary input.
secondaryModifier: *Optional*. The modifier key to be used for the secondary input.

Returns

int: The hashcode of `description` .

Description

Defines the default input settings for the `action` .

Never use SetKey to change actions! Use `ChangeKey()` for that.

Example

```
// this creates a new action called "Left" and binds A and the Left Arrow to it
cInput.SetKey("Left", "A", "LeftArrow");
// this uses Keys.D and Keys.RightArrow to accomplish something similar
cInput.SetKey("Right", Keys.D, Keys.RightArrow);
// creates a new action called "Pause" and binds Escape to it
cInput.SetKey("Pause", Keys.Escape); // note that secondary input defaults to None
// creates a new action called "Target" with LeftControl as a modifier, making Ctrl-T
the "Target" key.
cInput.SetKey("Target", Keys.T, Keys.None, Keys.LeftControl, Keys.None);
```

Valid Inputs

Below is a list of all the acceptable strings you can use for `SetKey()` and similar functions that accept a string to represent some sort of input device.

We recommend that you use the included `Keys` class, which has equivalents of all strings listed below, and gives you the benefit of autocomplete to help you avoid typos and other mistakes. For example, instead of using the string `"KeypadPeriod"` you would use `Keys.KeypadPeriod` in its place.

Keyboard Inputs

```
Alpha0
Alpha1
Alpha2
Alpha3
Alpha4
Alpha5
Alpha6
Alpha7
Alpha8
Alpha9
AltGr
Ampersand
Asterisk
At
BackQuote
Backslash
Backspace
Break
CapsLock
Caret
Clear
Colon
Comma
Delete
Dollar
DoubleQuote
DownArrow
End
Equals
Escape
Exclaim
F1
F2
```

```
F3
F4
F5
F6
F7
F8
F9
F10
F11
F12
F13
F14
F15
Greater
Hash
Help
Home
Insert
Keypad0
Keypad1
Keypad2
Keypad3
Keypad4
Keypad5
Keypad6
Keypad7
Keypad8
Keypad9
KeypadDivide
KeypadEnter
KeypadEquals
KeypadMinus
KeypadMultiply
KeypadPeriod
KeypadPlus
LeftAlt
LeftApple
LeftArrow
LeftBracket
LeftControl
LeftParen
LeftShift
LeftWindows
Less
Menu
Minus
Numlock
PageDown
PageUp
Pause
Period
Plus
Print
```

```
Question
Quote
Return
RightAlt
RightApple
RightArrow
RightBracket
RightControl
RightParen
RightShift
RightWindows
ScrollLock
Semicolon
Slash
Space
SysReq
Tab
Underscore
UpArrow
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
```

Mouse Inputs

```
Mouse0
Mouse1
Mouse2
Mouse3
Mouse4
Mouse5
Mouse6
Mouse Wheel Down
Mouse Wheel Up
Mouse Down
Mouse Left
Mouse Right
Mouse Up
```

Gamepad Inputs

```
JoystickButton0
JoystickButton1
JoystickButton2
JoystickButton3
JoystickButton4
JoystickButton5
JoystickButton6
JoystickButton7
JoystickButton8
JoystickButton9
JoystickButton10
JoystickButton11
JoystickButton12
JoystickButton13
JoystickButton14
JoystickButton15
JoystickButton16
JoystickButton17
JoystickButton18
JoystickButton19

Joystick1Button0
Joystick1Button1
Joystick1Button2
Joystick1Button3
Joystick1Button4
Joystick1Button5
Joystick1Button6
Joystick1Button7
Joystick1Button8
Joystick1Button9
Joystick1Button10
Joystick1Button11
```

```
Joystick1Button12
Joystick1Button13
Joystick1Button14
Joystick1Button15
Joystick1Button16
Joystick1Button17
Joystick1Button18
Joystick1Button19

Joystick2Button0
Joystick2Button1
Joystick2Button2
Joystick2Button3
Joystick2Button4
Joystick2Button5
Joystick2Button6
Joystick2Button7
Joystick2Button8
Joystick2Button9
Joystick2Button10
Joystick2Button11
Joystick2Button12
Joystick2Button13
Joystick2Button14
Joystick2Button15
Joystick2Button16
Joystick2Button17
Joystick2Button18
Joystick2Button19

Joystick3Button0
Joystick3Button1
Joystick3Button2
Joystick3Button3
Joystick3Button4
Joystick3Button5
Joystick3Button6
Joystick3Button7
Joystick3Button8
Joystick3Button9
Joystick3Button10
Joystick3Button11
Joystick3Button12
Joystick3Button13
Joystick3Button14
Joystick3Button15
Joystick3Button16
Joystick3Button17
Joystick3Button18
Joystick3Button19

Joystick4Button0
Joystick4Button1
```

```
Joystick4Button2
Joystick4Button3
Joystick4Button4
Joystick4Button5
Joystick4Button6
Joystick4Button7
Joystick4Button8
Joystick4Button9
Joystick4Button10
Joystick4Button11
Joystick4Button12
Joystick4Button13
Joystick4Button14
Joystick4Button15
Joystick4Button16
Joystick4Button17
Joystick4Button18
Joystick4Button19

Joy Axis 1-
Joy Axis 1+
Joy Axis 2-
Joy Axis 2+
Joy Axis 3-
Joy Axis 3+
Joy Axis 4-
Joy Axis 4+
Joy Axis 5-
Joy Axis 5+
Joy Axis 6-
Joy Axis 6+
Joy Axis 7-
Joy Axis 7+
Joy Axis 8-
Joy Axis 8+
Joy Axis 9-
Joy Axis 9+
Joy Axis 10-
Joy Axis 10+

Joy1 Axis 1-
Joy1 Axis 1+
Joy1 Axis 2-
Joy1 Axis 2+
Joy1 Axis 3-
Joy1 Axis 3+
Joy1 Axis 4-
Joy1 Axis 4+
Joy1 Axis 5-
Joy1 Axis 5+
Joy1 Axis 6-
Joy1 Axis 6+
Joy1 Axis 7-
```

```
Joy1 Axis 7+
Joy1 Axis 8-
Joy1 Axis 8+
Joy1 Axis 9-
Joy1 Axis 9+
Joy1 Axis 10-
Joy1 Axis 10+

Joy2 Axis 1-
Joy2 Axis 1+
Joy2 Axis 2-
Joy2 Axis 2+
Joy2 Axis 3-
Joy2 Axis 3+
Joy2 Axis 4-
Joy2 Axis 4+
Joy2 Axis 5-
Joy2 Axis 5+
Joy2 Axis 6-
Joy2 Axis 6+
Joy2 Axis 7-
Joy2 Axis 7+
Joy2 Axis 8-
Joy2 Axis 8+
Joy2 Axis 9-
Joy2 Axis 9+
Joy2 Axis 10-
Joy2 Axis 10+

Joy3 Axis 1-
Joy3 Axis 1+
Joy3 Axis 2-
Joy3 Axis 2+
Joy3 Axis 3-
Joy3 Axis 3+
Joy3 Axis 4-
Joy3 Axis 4+
Joy3 Axis 5-
Joy3 Axis 5+
Joy3 Axis 6-
Joy3 Axis 6+
Joy3 Axis 7-
Joy3 Axis 7+
Joy3 Axis 8-
Joy3 Axis 8+
Joy3 Axis 9-
Joy3 Axis 9+
Joy3 Axis 10-
Joy3 Axis 10+

Joy4 Axis 1-
Joy4 Axis 1+
Joy4 Axis 2-
```

```
Joy4 Axis 2+
Joy4 Axis 3-
Joy4 Axis 3+
Joy4 Axis 4-
Joy4 Axis 4+
Joy4 Axis 5-
Joy4 Axis 5+
Joy4 Axis 6-
Joy4 Axis 6+
Joy4 Axis 7-
Joy4 Axis 7+
Joy4 Axis 8-
Joy4 Axis 8+
Joy4 Axis 9-
Joy4 Axis 9+
Joy4 Axis 10-
Joy4 Axis 10+
```

Changelog

v2.9.1

- Fixed saving individual axis sensitivity, gravity, and deadzone settings. (Thanks Kailan)
- Better handle edge cases when trying to use GetText for invalid inputs. (Thanks Guray)
- Fixed some console warnings in recent versions of Unity.
- No longer support versions of Unity older than 5.6.4.

v2.9.0

- Corrected/added Keys class entries for Xbox One's View and Menu buttons.
- Fixed error in Unity 5.4 about making an API call from the constructor. (Thanks Steven)

v2.8.9

- Fixed IndexOutOfRangeException introduced in 2.8.8. (Thanks Doghelmer)
- Xbox One gamepad triggers now try to map to axes 9 and 10 when scanning for inputs.

v2.8.8

- Keys class attempts to smartly handle Xbox 360 and Xbox One gamepad differences.

v2.8.7

- Loading no longer "stomps" axis sensitivity, gravity, and deadzone settings. (Thanks SinisterCycle)
- cInput only loads settings once on startup.
- Modifiers now do some sanity checking.
- Fixed input values not updating while scanning during key rebind. (Thanks Sascha)
- Other minor bugfixes.

v2.8.6

- Forbidden keys are now completely ignored when changing input.
- You can now forbid axes using ForbidAxis. (Thanks Dan)
- cInput setup no longer requires restarting the Unity Editor.
- Fixed more gravity/sensitivity/deadzone errors caused by changes in 2.8.4. (Thanks Doghelmer)
- cInput now warns when trying to change inputs for keys that haven't been defined. (Thanks Joshmond)

v2.8.5

- Fixed IndexOutOfRange exception introduced in 2.8.4. (Thanks paulordbm)
- Added optional parameter to cInput.Init to disable cGUI. (Thanks willtrax)
- No more garbage collection each frame when getting calibrated axis inputs.

v2.8.4

- cInput can now be set up to check for input from any gamepad axis.
- Fixed disabling PlayerPrefs causing problems. (Thanks jpthek9)
- Keys class now has proper mappings for Xbox 360 gamepad on WebGL player.

v2.8.3

- Fixed gravity/sensitivity/deadzone not working for positive axis inputs. (Thanks Doghelmer)

v2.8.2

- Fixed error when setting deadzone/gravity/sensitivity for axis with 1 input. (Thanks Doghelmer)
- cInput fires an event (OnKeyChanged) when keys are changed via GUI or ResetInputs. (Thanks CaptainChristian)

v2.8.1

- Added cInput.anyKey and cInput.anyKeyDown properties.
- Using PlayerPrefs is now optional with usePlayerPrefs bool (default is enabled).

v2.8.0

- Fixed Xbox 360 DPad up/down axis on Windows in Keys class. (Thanks yvesgrolet)
- Updates to documentation.
- Updates to demo scene for compatibility with Unity 5.

v2.7.9

- Added public setter for scanning variable. (Thanks Ed)
- Increased the limit on inputs from 99 to 250. (Thanks Chris)
- Fixed cGUI being transparent when cGUI.bgColor isn't set properly.

v2.7.8

- Expanded cross-platform Xbox 360 controller mappings in the Keys class to all buttons/axes. (Thanks Phi)
- ChangeKey now warns if you try to change an invalid input. (Thanks Foxxis)
- cGUI will no longer warn about cSkin if cGUI isn't being used. (Thanks Walter)

v2.7.7

- Fixed Dictionary lookup sometimes causing an exception. (Thanks Andre)
- Fixed possible out of range exception when loading from external string. (Thanks Nick)
- Setting Keys/Axes now returns the hashcode for easy future reference. (Thanks Foxxis)
- Added overloaded GetText functions to return empty string instead of None. (Thanks Kurt)

v2.7.6

- cInput now uses hash lookups for functions like GetKey, GetAxis, etc. (Thanks Foxxis)
- Fixed Xbox triggers returning strange values on OSX. (Thanks Warbands)
- Keys class now returns appropriate axes for Xbox triggers on OSX.
- Various performance improvements, less loops/iterations per frame.

v2.7.5

- Fixed Axis 10 always returning positive. (Thanks Erik)
- Fixed case where analog input wouldn't always give correct value. (Thanks Manolo)
- Fixed GetButtonDown and GetButtonUp not working properly with analog inputs. (Thanks hoeloe)
- Fixed external string loading working with recent axis inversion status changes.
- You can now retrieve sensitivity, deadzone, and gravity values. (Thanks defaxer)

v2.7.4

- Fixed misplaced #endif which caused build problems. (Thanks Credd)

v2.7.3

- GetKey and GetButton functions now work on cInput's first run. (Thanks Erhune)
- cInput now compares hashes instead of strings in many places. (Thanks Foxxis)
- Make sure inversion status was saved before trying to load it. (Thanks nagisaki)
- A debug warning is more clear and should show up less often.
- Fixed a couple cases of redundant lookups. (Thanks Foxxis)

v2.7.2

- Axes no longer return 0 if deltaTime is 0. (Thanks Mel)
- SetAxis no longer resets the axis inversion to false. (Thanks SpectralRook)
- Changed Script Execution Order for cInput so that it runs before other scripts.
- No more errors if cGUI files are missing. (Thanks Shyam)
- An important debug log warning is no longer commented out. (Thanks Shyam)

v2.7.1

- Fixed cInput sometimes not being initialized properly. (Thanks Warbands and Jesse)
- Added event to cGUI for when the GUI is opened/closed. (Thanks Kelly and Mike)
- Removed the last remnants of the Necromancer GUI. (Thanks Mike)
- Fixed axis gradually increasing more than it was being pressed. (Thanks Chris)

v2.7.0

- GetAxis returns uniformly smoothed results for both digital and analog inputs.

- GetAxisRaw actually returns raw values (no more clamping values). (Thanks Goblox)
- cInput now binds Xbox triggers to axis 9 or 10 for left or right trigger respectively. (Thanks Lexie)
- Added easier access to Xbox triggers in Keys class.
- Fixed ArrayOutOfBounds exception when setting axis Sensitivity, Gravity, Deadzone. (Thanks dbarbieri and morten_nost)
- cInput now restores the instantiated cInput object if it gets destroyed. (Thanks Sean)
- cInput no longer appears to support 5 gamepads since Unity only supports 4. (Thanks OhiraKyou)
- MonoDevelop should no longer complain about unsupported default parameters. (Thanks Daithi)
- Implemented a "scanningDeadzone" to prevent axes with deadzone issues from breaking input scanning. (Thanks megan_l_fox)

v2.6.1

- Corrected installation instructions in readme file.
- Fixed GetKeyUp and GetKeyDown always returning false for Mouse Wheel. (Thanks Krileon)
- Replaced cInput.dat with plaintext version.

v2.6.0

- Fixed immediately selecting an axis that gives non-zero value by default. (Thanks Goblox)
- Fixed case where GetAxisRaw always returned 0. (Thanks gumboots)
- Fixed axes sometimes returning max value (1) when halfway down. (Thanks Goblox)
- Fixed some issues with axis calibration.

v2.5.9

- Fixed SetAxis overload function not working properly. (Thanks Goblox)

v2.5.8

- Fixed an error in the console log. (Thanks Matt and Harabeck)

v2.5.7

- Fixed a bug in axis calibration code. (Thanks gumboots and Krileon)

v2.5.6

- Converted cInput scripts to UTF-8 which should prevent MonoDevelop errors. (Thanks Matumit)
- Renamed Demo Class files to prevent Class/Namspace conflicts. (Thanks Michael)
- Cleaned up a bunch of Debug Logs that accidentally got left in there somehow.

v2.5.5

- Added deadzone settings on a per-axis basis with SetAxisDeadzone().
- Fixed ChangeKey not setting modifiers properly (for reals this time). (Thanks ratking)
- Fixed mixing keyboard and joystick/mouse inputs on an axis being totally broken. (Thanks Laztor)
- Fixed individual sensitivity and gravity not working properly (for really reals this time).

v2.5.4

- Fixed GetAxisRaw almost always returning 0 when using gamepad and mouse axis. (Thanks V4nKw15h)
- Fixed ChangeKey not setting modifiers properly. (Thanks Krileon)
- Fixed secondary inputs not accepting modifiers. (Thanks hjupter)

v2.5.3

- Fixed cGUI not working when cSkin wasn't set. (Thanks KeithT)
- Fixed trying to load from PlayerPrefs when there was nothing saved.

v2.5.2

- Added Editor script to create the InputManager.asset file.
- Removed unused variable which caused Editor warnings. (Thanks Neurological)
- GUI color now works in both Unity 3.x and 4.x. (Thanks Neurological)

- Added functions to remove modifiers. (Thanks xadhoom)

v2.5.1

- Got individual axis gravity and sensitivity settings to work (for reals this time). (Thanks Kementh)
- Fixed bug where duplicate copies of same axis could be created.

v2.5.0

- Added the ability to set up and use modifier keys with AddModifier() and SetKey(). (Thanks Steve Tack and others)
- Added the ability to set gravity with SetAxisGravity() rather than having all axes use the same gravity. (Thanks Can Baycay)
- You can now also set the gravity for an axis when you create it with SetAxis(). (Thanks Can Baycay)
- Rewrote GUI code and separated it from cInput into its own script (see cGUI documentation).
- Separated Keys class from cInput into separate script.
- Added new GUISkins.

v2.4.5

- Fixed inability to use SetKey after cInput object was created. (Thanks Rob)

v2.4.4

- Fixed using mouse button as input triggering another call to ChangeKey. (Thanks Joseph)
- Fixed flickering Axis name in GUI when changing key.
- Deadzone for axes used as buttons now uses public "deadzone". (Thanks Lexie)
- Prevented _string2Key Dictionary from being created more than once.

v2.4.3

- Fixed out of range null-reference exception related to number of gamepads. (Thanks Julian)

v2.4.2

- cInput now caches joystick axis strings. (Thanks goodhustle)
- Escape doesn't close GUI menu in demo if scanning for new inputs. (Thanks MrG)

v2.4.1

- Added public Init() to manually create the cInput object. (Thanks goodustle)
- cInput now supports saving/loading inputs to/from sources other than PlayerPrefs. (Thanks gumboots)
- Made "scanning" property static. (Thanks goodhustle)
- Fixed a bug with allowDuplicates.
- ResetInputs() now re-saves defaults to PlayerPrefs. (Thanks goodhustle)
- Fixed ChangeKey() to properly (dis)allow certain inputs. (Thanks Adrian)
- Fixed certain debug messages showing when they shouldn't.

v2.4.0

- cInput is now compatible with iOS, Android, and Flash.
- Improved efficiency (less CPU overhead).
- Fixed key & axis mapped to same input causing KeyUp to fire instantly. (Thanks goodhustle)
- Added read-only public bool "scanning" as getter for the private bool "_scanning" (Thanks goodhustle)

v2.3.1

- Added the ability to set sensitivity with SetAxisSensitivity() rather than having all axes use the same sensitivity. (Thanks Jacob)
- You can now also set the sensitivity for an axis when you create it with SetAxis().
- You can now forbid keys from being used as inputs with ForbidKey(). (Thanks Kurt)

v2.3.0

- Added IsKeyDefined() and IsAxisDefined() methods to allow other developers to make cInput-compatible plugins. (Thanks David)
- Added GetAxisRaw() method. (Thanks Jay, patrickw)

- Added Clear() method to remove all PlayerPrefs keys stored by cInput
- Fixed changed/deleted/added keys not saving/loading properly
- Keys class properties are now const, as they should be. (Thanks Jay)
- SetAxis() now has some error checking to make sure you're using it correctly. (Thanks Jay)

v2.2.3

- Fixed bug with GetAxis() not creating cInput object properly. (Thanks Christopher)

v2.2.2

- Fixed bug with inverted axes not saving/loading properly. (Thanks Christopher)

v2.2.1

- Analog inputs are now compatible with GetKeyDown(), GetKeyUp(), GetButtonDown(), and GetButtonUp() functions. (Thanks goodhustle)

v2.2.0

- ShowMenu() now accepts additional arguments for easier customization of cInput's menu. (Thanks goodhustle)

v2.1.1

- InvertAxis() and IsAxisInverted() have been combined into a single function: AxisInverted().
- Minor bugfixes.

v2.1.0

- Length() method has been replaced by the read-only cInput.length property.
- Can now manually assign keys with ChangeKey() instead of waiting for input from player.
    - This can be used to pre-designate control "Profiles" which can be switched from a

menu.

- Added GetButton(), GetButtonDown(), and GetButtonUp(). (Thanks patrickw)
- Can now allow duplicate inputs for multiple actions using cInput.allowDuplicates (Thanks patrickw)
- Can now invert an axis using InvertAxis("axisName"). (Thanks Ryan)
- Can get axis inversion status with IsAxisInverted("axisName"). (Thanks Ryan)
- SetAxis() can now accept 1 action instead of requiring 2. (Thanks patrickw)
- Better support for sliders, gas pedals and other analog inputs with Calibrate(). (Thanks patrickw)

v2.0

- Initial release of cInput 2.