1. Overview

Joystick Gremlin is a program that allows the configuration of joystick like devices, similar to what CH Control Manager and Thrustmaster's T.A.R.G.E.T. do for their respectively supported joysticks. However, Joystick Gremlin works with any device be it from different manufacturers or custom devices that appear as a joystick to Windows. Joystick Gremlin uses the virtual joysticks provided by [vJoy](https://github.com/shauleiz/vJoy/) to map physical to virtual inputs and apply various other transformations such as response curves to analogue axes. In addition to managing joysticks, Joystick Gremlin also provides keyboard macros, a flexible mode system, scripting using Python, and many other features.

The main features are:

* Works with arbitrary joystick like devices
* User interface for common configuration tasks
* Merging of multiple physical devices into a single virtual device
* Axis response curve and dead zone configuration
* Arbitrary number of modes with inheritance and customizable mode switching
* Macros with joystick, keybouard and mouse inputs
* Python scripting support

Joystick Gremlin provides a graphical user interface, described in [Section 3](https://whitemagic.github.io/JoystickGremlin/interface), which allows commonly performed tasks, such as input remapping, axis response curve setups, and macro recording to be performed easily. Functionality that is not accessible via the UI can be implemented through custom modules, explained in detail in [Section 4](https://whitemagic.github.io/JoystickGremlin/custom_modules).

1.1 Installation

Joystick Gremlin has one major dependency, vJoy which provides virtual joysticks which Joystick Gremlin feeds with data. Download links to the programs needed are listed below:

* [Joystick Gremlin](https://whitemagic.github.io/JoystickGremlin/download)
* [vJoy](https://github.com/jshafer817/vJoy/releases/tag/v2.1.9.1)

vJoy creates virtual joysticks which show up as a device in Windows and Joystick Gremlin uses these to forward inputs to them. The VC2010 package is required by Python but is likely already installed. The same goes for the VC2013 package which is required by vJoy but is most likely already installed on the machine. If one or both of these are missing they can be obtained from:

* [VC Redistributable 2010 (x86)](http://www.microsoft.com/en-us/download/details.aspx?id=5555)
* [VC Redistributable 2013 (x86)](http://www.microsoft.com/en-us/download/details.aspx?id=40784)

It's important that the VC Redistributables are the x86 ones, even if you're running a x64 system, as vJoy and Python require the 32bit libraries.

vJoy Configuration

In order to properly use Joystick Gremlin vJoy has to be configured first. This is done via the *Configure vJoy* program. This program allows setting the properties of all existing vJoy devices. Typically a single vJoy device is enough. In order to use 8-way POV hats with Joystick Gremlin the hats have the be configured as continuous in vJoy. The image below shows what a properly configured vJoy device looks like. Once everything is set as desired clicking *Apply* configures the vJoy device and the window can be closed.

A screenshot of a computer

AI-generated content may be incorrect.

*VJoy configuration dialog with settings required for proper Joystick Gremlin operation.*

1.2 Concepts

The following section introduces the terminology used by Joystick Gremlin. Next, the different concepts will be exapleined starting with individual actions and ending with a complete profile.

Input

An input is an axis, button, hat, or keyboard key on a physical device.

Action

An action is something Joystick Gremlin executes in response to the user activating a physical input. Examples include running a macro, sending button presses to vJoy, or changing to a different mode. Each action has a condition attached to it which can further dictate when an action is to be executed.

Container

A container holds one or more actions and uses them in the manner specific to the particular container. In the simplest case it simply executes the action(s). More complex cases are where the container decides which action(s) to execute based on the duration a button is held down.

Device

A device is a physical controller connected to the computer such as a joystick, pedal, or throttle. Each device has multiple inputs with their own associated actions.

Mode

A mode is a collection of actions associated with the inputs for each connected device. Each mode can inherit from one other mode, his parent. If a mode and its parent both define actions for the same input only the actions of the mode and not its parent are used. If the mode defines no actions for an input but the parent does the parent's actions are used. This allows having a common set of base actions that more specialised modes add to or change.

Profile

A profile is a folder which contains a XML configuration file together with any custom modules used. The profile contains the settings made via the user interface for each of the connected peripherals.

Example

The following is a simple example showing how the above introduced concepts work together. The profile contains two modes, *Default* and *Landing* and two physical devices *Thrustmaster Warthog Stick* and *CH Pro Pedals*. Both devices have varying actions executed on the exemplary inputs used which change between the two modes. The image on the right shows the conceptual nesting of the concepts as shown in the explicit example.

* **Profile:** star\_citizen.xml
  + **Device:**Thrustmaster Warthog Stick
    - **Mode:**Default
      * **Input:**Axis X
        + **Container:**Basic

**Action:**Map to vJoy Axis X

* + - * **Input:**Button 1
        + **Container:**Basic

**Action:**Map to vJoy Button 1

* + - **Mode:**Landing
      * **Input:**Axis X
        + **Container:**Basic

**Action:**Map to vJoy Axis X

* + - * + **Container:**Basic

**Action:**Apply response curve with deadzone

* + - * **Input:**Button 1
        + **Container:**Tempo

**Action:**Map to vJoy Button 10

**Action:**Map to vJoy Button 12

* + **Device:**CH Pro Throttle
    - **Mode:**Default
      * **Input:**Axis X
        + **Container:**Basic

**Action:**Map to vJoy Axis Z

* + - * + **Container:**Basic

**Action:**Apply response curve

* + - * **Input:**Button 1
        + **Action:**Run chaff & flare macro
    - **Mode:**Landing
      * **Input:**Axis X
        + **Container:**Basic

**Action:**Map to vJoy Axis Z

* + - * + **Container:**Basic

**Action:**Apply response curve

* + - * **Input:**Button 1
        + **Container:**Basic

**Action:**Map to vJoy Button 21

A screenshot of a computer program

AI-generated content may be incorrect. *The image shows how the various concepts explained below relate to each other.*

Last update: June 23, 2021

[](https://whitemagic.github.io/JoystickGremlin)

2. Quick Start

2.1 Installation & Setup

On most sytems installing Joystick Gremlin only requires the installation of [vJoy](https://github.com/jshafer817/vJoy/releases/tag/v2.1.9.1) and [Joystick Gremlin](https://whitemagic.github.io/JoystickGremlin/download) itself. Once vJoy is installed it is advisable to configure the vJoy device(s), using vJoyConf, to contain the desired number of axes, buttons, and hats.

A screenshot of a computer

AI-generated content may be incorrect.

*VJoy configuration dialog with settings required for proper Joystick Gremlin operation.*

When configuring the hat instances make sure to select **continuous** for their type.

2.2 Simple Profiles

The following is intended as a simple guide that introduces the concepts used in Joystick Gremlin by way of examples of how common tasks are achieved. One can think of Gremlin as a system that reacts to the user's input. This means that when the user does something Gremlin will react in a way specified by the user. In a sense the following summarizes the way Gremlin operates:

When I press this button on my joystick, I want the following to happen.

This is also reflected by the design of the UI which is divided into two halves. The left side lists all the physical inputs of a particular joystick, such as axes, buttons, and hats. The right side lists the actions associated with a particular physical input. Going back to the above idea the left hand side contains the “*When I do*” part while the right hand side reflects the “*Gremlin does*” part.

A screenshot of a computer

AI-generated content may be incorrect. *Joystick Gremlin UI*

Basic Mappings

The simplest type of mappings are remapping a physical input to a virtual one with the **Remap** action. The other basic action is to map a physical input to a keyboard key or combination of keys using the **Map to Keyboard** action. Both of these actions will press and hold their virtual or simulated input for as long as the physical input is pressed.

These actions allow merging of multiple physical devices into a single one which can be useful for games that only recognize a single joystick. Mapping joystick buttons to keyboard keys allows using the default bindings of a game without having to rebind everything or if a game doesn't support joystick input.

Macros

Sometimes it's not enough to simply press and hold a button, for example when a sequence of keys needs to be pressed or an axis ramped up in a particular manner. This is where the **Macro** action comes into play. This action allows the recording and customizing of events that should be played back when the associated physical input is pressed.

Besides simply executing the macro once upon activation it is also possible to have a macro repeat in a variety of ways:

* Repeat a fixed number of times
* Repeat until the physical input is pressed again
* Repeat as long as the physical input is pressed

Dealing with Axes

Axes often require additional configuration by adding deadzones to prevent accidental inputs or a response curves that modify the response to a physical input. This allows increasing or reducing the sensitivity of particular areas of physical input to tailor the axis to a user's preference. The horizontal axis of the curve editor reflects the physical deflection of the axis from minimum (left) to maximum (right) while the vertical axis reflects to output generated for the corresponding physical input.

2.3 Complex Profiles

While simple profiles as described above are already capable of supporting a large variety of setups, sometimes a bit more complexity is required to create a particular setup. This is where **containers**, **conditions**, and **modes** come into play.

Container

A container holds one or more actions and is capable of executing additional logic specific to the container. The simplest container type **Basic** simply executes all contained actions. This is the type which we've used implicitly when creating a simple profile. There are currently two more advanced containers, **Tempo** and **Chain**. The tempo container executes different sets of actions depending on how long the corresponding physical input is pressed. This allows using a single button for two different actions where, for example, one is used more often then the other. The chain container contains an arbitrary number of action sets and with each press of the associated physical input the next set of ations in the chain are executed. When the last action set was processed the first one will be run next. This can be used when a single physical button should be used to switch through different actions.

Condition

Sometimes it can be helpful to control the condition under which an action or container is executed. The execution can thus be conditioned on the state of other input devices such as joysticks or keyboard, as well as the state of the input associated with the action or container. This can be used to implement a shift state which normally executes one action but if a certain button is pressed a different action is executed.

Mode

In certain situations it can be beneficial to split a profile into several parts, called modes. Each mode can have their own individual set of physical input to action mappings. For example in flight simulators this can be used to have one mode for air combat, one for ground attack, and one for communications. One powerful feature of the mode system is the **inheritance** capability. A mode can inherit from a single parent which results in the child mode executing whatever action their parent defined for a given input if the child has no actions of their own specified. This allows creating basic modes and then fine tuning them in a child mode for particular usage scenarios without having to duplicate common functionalities.

Last update: June 23, 2021

3. User Interface

In the following the various components of the user interface are introduced and their usage described. The UI should be sufficient for most common use cases. However, if functionality is missing, user created modules can provide these. Examples of how to achieve certain tasks using the tools provided by Joystick Gremlin are shown in [5. Examples](https://whitemagic.github.io/JoystickGremlin/examples/).

3.1 Overview

The following is a short overview of the different components that make up the main user interface, shown in the following image.

A screenshot of a computer

AI-generated content may be incorrect. *Joystick Gremlin UI*

1. Overview of all the inputs available for a given physical device. The small icons on the far right of each input indicate the type of actions associated with the input. The icons and their meaning are summarised in the table below.
2. The right hand portion of the UI shows the list of containers and actions associated with the currently selected input. This panel allows configuring the actions to execute when the physical input is used.
3. This drop down lists all available actions for this type of input. Pressing the "Add" button will embed the selected action in a basic container.
4. This drop down contains the available containers for the currently selected input. Pressing the "Add" button inserts an empty container of the desired type.
5. Each container can contain up to three tabs that handle various configurations. The *Action* tab allows the basic configuration of the actions. The *Condition* tab allows fine tuning under which conditions the action or container should be executed. Finally the *Virtual Button* allows configuring an axis or hat to be used like a button.
6. The mode section allows changing the mode currently being configured.
7. Each tab represents an individual device that is currently connected to the computer. The "Settings" tab allows configuring properties for the entire profile.
8. Tool bar which holds the most commonly used actions, from left to right:
   * Open an existing profile.
   * Activate Joystick Gremlin, when active the button is pressed and a green icon is shown. This also changes the status bar display. Pressing the button while Joystick Gremlin is running will disabled it again.
9. The status bar shows whether or not the program is currently running a profile. If the program is running the currently active mode is shown as well if code execution is paused. If the input repeater is active it's status is also shown in the far right.

The following table shows the different icons and the actions they represent.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Cycle modes |  | Description |
|  | Macro |  | Map to keyboard |
|  | Map to Mouse |  | No-Op |
|  | Pause |  | Play sound |
|  | Remap axis |  | Remap button |
|  | Remap hat |  | Response curve |
|  | Resume |  | Split axis |
|  | Switch mode |  | Switch to previous mode |
|  | Temporary mode switch |  | Text-to-speech |
|  | Toggle pause & resume |  |  |
|  |  |  |  |

3.2 Virtual Buttons

Each input has a natural behaviour or type of states it can be in. Certain actions, however, require a binary state reflecting that of buttons, i.e. *pressed* and *released*. In order for axis and hat inputs to support this they require to specify the condition under which they should be considered *pressed* and *released*, effectively turning them into a virtual button.

This is often needed, for example when using individual hat directions as buttons when a game doesn't support mapping individual hat directions. A common use with axis is to enable afterburners when the throttle reaches 100% thrust. This type of situations are where a virtual button is used. The details of these activation conditions is covered next.

Button and Keyboard Key

As these inputs are already binary there is no virtual button configuration needed.

Axis

A white rectangular object with black text

AI-generated content may be incorrect.

*Turning an axis into a virtual button by specifiying the range in which the virtual button is considered pressed.*

Actions associated with axis inputs have a range based setup which simulates a button press when the axis enters the specified range and releases the virtual button when it leaves the range. In addition to being able to specify the range for the button you can also specify a direction in which said range has to be entered if desired. This allows triggering two different actions for the same range depending on the direction it is traversed.

Hat



*Turning a hat into a button by selecting the directions which when pressed are considered to be pressing a virtual button.*

The eight directions of POV hats can be treated as states in which the virtual button is considered pressed. It is possible to select multiple directions at once and any of the selected directions will be considered as part of a single virtual button.

3.3 Activation Conditions

Sometimes executing actions when a physical input is pressed is doesn't provide enough control over the execution of actions. To provide additional control over the execution of actions Gremlin allows the user to specify conditions under which action can or cannot be executed.

Typical usage scenarios for this would be the use of a modifier or shift key to change what a particular physical input does in certain situations. Another common case is to trigger different actions when a button is pressed or released, for example on throttles with switches rather then buttons.

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

*Conditions can be placed on either each action individually or on the container as a whole.*

As mentioned in [3.1. Overview](https://whitemagic.github.io/JoystickGremlin/interface/3_1) the conditions are configured in the condition tab present in every container widget. The tab allows placing conditions on either every action individually or on the container as a whole. These conditions will dictate whether or not a particular action or container is executed when the corresponding physical input is used.

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

*Activation conditions can check for the state of the action's physical input, a physical joystick input, or a keyboard input.*

There are three types of conditions available:

**Action**

The *Action* condition checks whether or not the physical or virtual button triggering the action or container is pressed or released. This enables for example to run a macro when the physical button is released rather then when the button is pressed.

**Joystick**

The *Joystick* condition allows to use any physical joystick input as a condition. For buttons this is whether or not the button is pressed, for hats this is whether a particular direction is pressed, and for axis whether or not it currently is inside or outside a specified range.

**Keyboard**

The *Keyboard* conditions allows checking whether or not a particular keyboard key is pressed or not. This would typically be used to implement a modifier or shift key. To trigger different actions depending on the state of a key.

Multiple of these conditions can also be used jointly by specifying whether or not an action / container should be executed if *any* or *all* of the conditions are satisfied.

3.4 Containers

As the name implies containers are designed to hold one or more actions. In addition to grouping actions together a container can also apply further logic to groups of actions. Currently the following three types of containers exist.

Basic

A screenshot of a computer

AI-generated content may be incorrect.

*Basic container widget.*

This container simply holds a group of actions without any further specific functionality. This is the default container and suitable for most situations.

Chain

A screenshot of a computer program

AI-generated content may be incorrect.

*Chain container with two groups.*

This container holds several independent groups of actions. Each time the physical action with which this container is associated the next action group is executed. When the last group in the chain is reached the first group is executed again. Additionally, a timeout can be set after which the chain is reset to the first entry.

Double Tap

A screenshot of a computer

AI-generated content may be incorrect.

*Double tap container.*

This container holds two groups of actions, the first group is executed on a single button press while the second one is executed on a double press. The *Double tap delay* allows the modify the double tap speed, while *Single / Double tap* configures whether the single tap action is active together with the doble tap action or not. In the exclusive mode, the single tap action will be executed with a delay corresponding to the double tap delay.

Hat Buttons

A screenshot of a computer

AI-generated content may be incorrect.

*Mapping of a hat to 4 action groups with two being populated.*

This container allows a hat to easily be mapped onto four or eight individual action groups. Each action group can contain any combination of actions that are valid for buttons and will be executed when the corresponding hat direction is triggered.

Smart Toggle

A screenshot of a computer

AI-generated content may be incorrect.

*Smart Toggle container with the action specified as a remap.*

The smart toggle container allows for a single group of actions that are have on and off states, such as remap and map to keyboard to be used in two manners. If the input is held down the action will perform as a typical remap action would, i.e. staying active as long as the input is pressed. However, when a short button press is detected, specified by the *Toggle time* then the first such press toggles the down state, i.e. holding the action down, and the second short press releases the action again.

Tempo

A screenshot of a computer

AI-generated content may be incorrect.

*Tempo container with a remap action in both the short and long press action groups.*

This container holds two groups of actions that get executed under different conditions. The first one is run when the input is pressed for a duration shorter then the specified delay. The second group is run when the input is pressed for longer then the specified delay. Whether the first group is executed with the input being pressed or only when it is released can be controlled by the activation setting.

3.5 Actions

As described in [Section 1.2](https://whitemagic.github.io/JoystickGremlin/overview/#1_2), each input, such as axis, button, hat, or keyboard key, can have multiple actions associated with it. However, not all inputs have access to the same types of actions as they are not applicable to them. In the following all available actions are described.

Each action has an activation condition associated with it, see [Section 3.2](https://whitemagic.github.io/JoystickGremlin/interface/#3_2), which can be used for fine grained control over them. Additionally, each input can also be configured to be always executed by checking the *Always Execute* box, see item 2 in the above image. Enabling this option results in actions associated with that input to be performed even if execution of callbacks is paused. This is mainly useful for resuming when execution has been paused.

Remapping

A grey and white striped background

AI-generated content may be incorrect.

*Axis remapping dialogue*

A grey and white striped background

AI-generated content may be incorrect.

*Button remapping dialogue*

A grey and white striped background

AI-generated content may be incorrect.

*Hat remapping dialogue*

The remapping actions enable the user to propagate inputs from a physical joystick input to an equivalent vJoy input, i.e. axis to axis, button to button, and hat to hat. This allows the merging of multiple physical devices into a single device.

In the case of axis remapping the condition can be used to press a vJoy button while the axis in a certain range and release it once the range is left. This could for example be used to engage afterburners when a throttle is set to greater then 95%.

A remap actions for an axis can be configured in two ways: absolute, the physical input dicates the virtual output value directly, and relative, the physical input dicates the rate of change of the virtual output.

In most cases button remap actions should always trigger, i.e. on button press and release, as otherwise the forwarding to vJoy is incomplete.

Response Curve

A screen shot of a graph

AI-generated content may be incorrect.

*Response curve action dialogue which allows the customisation of the joystick response as well as dead zones.*

The response curve dialogue allows the customisation of the response produced by the joystick using the curve editor. The shape of the curve is controlled using a set of control points.

A new control point is added by a double left click in an empty area of the curve editor. Removing an existing control point is achieved by selecting the desired point and pressing the Del key on the keyboard. A single left click on a control point will mark the point as active. An active point can be dragged in the window to modify its position. Alternatively, the text fields below the curve editor allow for precise numerical control of the position.

Finally, the dead zones for the axis can be defined using the sliders and input fields at the bottom of the dialogue. The fields and sliders control the full deflection dead zone (1st and 4th field from the left) as well as the centre deflection dead zones (2nd and 3rd field).

Currently there are two types of response curve types available which are selected from the drop down menu at the top of the widget:

**Cubic Spline**

A simple spline with only control points dictating locations the curve has to pass through. No control over the shape is provided.

**Cubic Bézier Spline**

A more complex spline with control points and "handles" that can modify the shape of the overall curve. Importantly this allows to control how the curve approaches the end points of the curve.

A response curve mapping action always needs to be paired with a remap action, as otherwise the transformation will not have any effect.

In order for response curves to work properly the game has to be configured to use a linear 1:1 curve, as otherwise the two curve settings will interfere with each other producing undesirable results. This also applies when using vJoy response curves, if they are not linear the output produced by this action will be further modified.

Macro

A screenshot of a computer

AI-generated content may be incorrect.

*Macro action dialogue allowing the recording of event sequences.*

The macro dialogue allows the creation of macros capable of emitting keyoard, mouse, and joystick events with timing information between individual events.

Adding new keystrokes to a macro is done by pressing the *Record* button, after which all keystrokes will be recorded sequentially. To stop the recording simply press the *Record* button again. A pause can be added by pressing the *Add Pause* button, which will insert a pause after the currently selected entry. The length of the pause can be modified by double clicking the entry. Selecting an entry allows it to be moved up and down using the *Up* and *Down* buttons.

The stopwatch icon  allows recording the input delays when recording a macro. The buttons with labels Axis, Button, Hat, Keyboard, and Mouse allow enabling or disabling those specific input events when recording a macro.

Clicking on any entry in the action list allows modifying that entry as appropriate for the input. Alternatively the entire action can also be changed through the dialog on the right hand side of the action. In the bottom right corner, the *Macro Settings* section allows changing the behaviour of the macro. *Exclusive* ensures that no other macro is running when this one is executing. The dropdown allows selecting a repeat mode for the macro.

**None**

The macro is executed exactly once.

**Count**

The macro is executed as often as specified with the specified delay between subsequent executions.

**Hold**

The macro is executed as long as the (virtual) button is held down with a configurable delay between subsequent executions of the macro.

**Toggle**

On the first press of the (virtual) button the macro is started and will repeatedly execute with the specified delay between executions until the (virtual) button is pressed again.

Each macro is executed independently, which means that multiple long running macros can have their key presses interfere with each other.

Map to Keyboard

A screen shot of a computer

AI-generated content may be incorrect.

*Map to Keyboard dialogue which allows mapping of a physical key to an arbitrary keyboard key combination.*

The map to keyboard action allows an arbitrary keyboard key sequence to be linked to a single physical input, such as a button. The specified keyboard sequence is pressed when the physical button is pressed and released when it is released.

Map to Mouse

A screenshot of a computer

AI-generated content may be incorrect.

*Mapping an axis to a mouse direction, X or Y with speed range.*

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

*Mapping a button to a mouse button or accelerating mouse motion along a specified direction.*

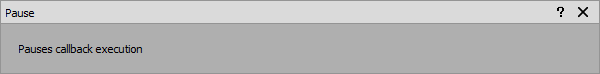
A screenshot of a computer

AI-generated content may be incorrect.

*Mapping a hat to mouse directional control with speed range.*

This action allows using joystick axes, button, or a hat to control the relative motion of the mouse cursor as well as press mouse buttons. The speed values are in pixels per step while the acceleration on the hat regulates how long it takes to go from the slowest to the fastest movement speed.

Pause, Resume & Toggle



*Pause dialogue which allows to temporarily stop the execution of callbacks.*

A grey rectangular object with a white stripe

AI-generated content may be incorrect.

*Resume dialogue which allows Joystick Gremlin to execute callbacks again after enabling the pause state.*

A grey rectangular object with a white stripe

AI-generated content may be incorrect.

*Toggle between pause and resume dialogue, which toggles between the two states.*

The pause, resume, and toggle actions control whether or not Joystick Gremlin executes callbacks when inputs are used. When the application is paused only inputs that were configured to *always execute* will be executed.

In order for the *resume* and *toggle* actions to be useful they have to be defined as *always execute*, as otherwise it will be impossible to resume from the paused state.

Change Mode

A grey and white striped background

AI-generated content may be incorrect.

*Change mode action dialogue which allows to select the mode to switch to when the action is triggered.*

This action will change the currently active mode of Joystick Gremlin to the one selected from the drop down list.

Temporary Mode Switch

A grey and white striped background

AI-generated content may be incorrect.

*Temporary mode switch dialogue which allows specifying a mode to switch to as long as the action is triggered, i.e. the "button" is held down.*

This action will change the current mode to the one specified when the button or button-like input is pressed. When the input is released the mode will automatically be changed back to the previous one.

Switch to Previous Mode

A grey rectangular object with a white stripe

AI-generated content may be incorrect.

*Dialogue of the action that switches to the previously active mode.*

This action changes the current mode to the previously active one.

Cycle Modes

A screenshot of a computer

AI-generated content may be incorrect.

*The dialogue of the cycle modes action allows adding various action which will be cycled through in order on each subsequent activation.*

This action allows the configuration of a list of modes that are cycled through consecutively with each activation. In case that the currently active mode is not part of the list the first mode in the list will be activated.

New modes can be added by selecting the desired mode in the drop down list and pressing the *Add* button. The currently selected mode can be deleted by pressing the *Delete* button and moved up or down with the *Up* and *Down* button respectively.

Split Axis

A screenshot of a computer

AI-generated content may be incorrect.

*Split axis dialog which allows splitting a single axis into two virtual ones.*

This action allows splitting a single physical axis into two virtual ones. The value at which the physical axis is split can be adjusted by the user.

Play Sound

A white line on a gray surface

AI-generated content may be incorrect.

*Plays the sound file specified at the given volume when the action is triggered.*

This action allows the specification of a sound to be played with the desired volume when the action is triggered. Only one sound can be played at a time and new sounds will interrupt currently running ones.

Text to Speech

A screenshot of a computer

AI-generated content may be incorrect.

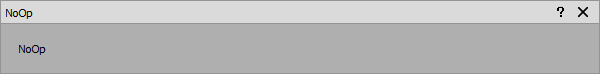
*Text to speech action dialogue which allows entering text to be spoken by Joystick Gremlin.*

This action allows arbitrary text to be spoken by Joystick Gremlin. The system knows how to replace some values with current information. The currently known replacements are listed below.

|  |  |
| --- | --- |
| **Expression** | **Replacement** |
| ${current\_mode} | The name of the currently active mode |

The text segments are spoken in the order they are requested and as such queueing multiple long segments can lead to significant delays between them being triggered and played.

No Operation



*An action that performs no function other then to serve as a place holder.*

Sometimes one wants an empty or no operation action that can be used as a place holder. For example to use a tempo container with no short press action. This action allows having an action inside a container without it doing anything.

3.6 Profile Settings

The *Settings* tab allows configuring settings that apply to the entire profile. This includes:

* Defining how the initial mode is selected
* Default delay between macro actions
* Defining which vJoy devices are to be treated as physical inputs rather then outputs
* Configuring the initial value of vJoy output axis on activation

A screenshot of a computer

AI-generated content may be incorrect.

*Profile settings tab showing three vJoy devices and their configuration*

Startup Mode

By default Gremlin uses a heuristic to figure out which mode to select upon activation of a profile. The mode selected is either the one that was active upon deactivation of Gremlin or the first mode without a parent in alphabetical order.

If this heuristic is not suitable the drop down allows the selection of any of the modes present in the profile to be selected as the initial mode.

Default Macro Action Delay

The macro system will inject a fixed delay between individual non-pause actions. Certain games may require a longer delay between actions which can be set using this option.

vJoy as Input

In certain scenarios it can be desirable to have Gremlin treat a vJoy device as an input, e.g. when another program feeds a vJoy device but doesn't offer some features that Gremlin does. By checking the box next to the vJoy devices it will be treated as an input rather then output device.

When a vJoy device is declared as an input device Gremlin will no longer show it as a target in the Remap action and other similar actions. However, a tab for that vJoy device will appear which allows adding actions to all the axes, buttons, and hats of the vJoy device remapping to other vJoy (output) devices.

This features was newly introduced in Release 11 and as such may still contain bugs and unforeseen interactions with other parts of Gremlin.

Initial vJoy Axes Values

By default Gremlin sets all vJoy axes to a value of 0.0, i.e. centered. In some situations this is not ideal, for example a throttle, and as such this section allows setting the initial value to use for each axis.

3.7 Modules

The *Modules* tab allows adding and configuring user plugins that can provide behaviour that is complicated or impossible to achieve through the UI otherwise.

A screenshot of a computer

AI-generated content may be incorrect.

*The modules tab allows adding and configuring of modules or user plugins.*

User plugins are the next development of the initial custom modules. Custom modules allowed extending Gremlin with arbitrary code. However, they were not very user friendly as they required code to be modified for simple adaptations to a user's setup. The new user plugins can be configured directly in the UI and can be used multiple times if the same functionality with slightly different settings is needed. An introduction into how to write such plugins can be found in [4. User Plugins](https://whitemagic.github.io/JoystickGremlin/user_plugins/).

Adding a new plugin is achieved by clicking on the *Add Module* button and selecting the appropriate file. Once a plugin has been added additional instances can be created via the plus sign. Existing instances can be renamed via the pen icon, and removed by clicking on the minus sign. To configure an instance, press the cog icon.

The configuration of a plugin instance allows setting all the exposed variables of the plugin. Each of the variables presents a UI appropriate to the particular type of input needed.

3.8 Tools

In the following section the various tools provided by Joystick Gremlin are described. They can be accessed via the *Tools* and *Actions* menus.

Mode Manager

A screenshot of a computer

AI-generated content may be incorrect.

*The mode manager allows the creation, deletion, and renaming of modes as well as defining their relationships.*

The mode manager, accessed via *Tools > Manage Modes*, allows users to create new modes, delete existing ones and define which mode inherits configurations from others. Pressing "Add Mode" adds a new mode, pressing the button to the right of each mode with the **x** on it deletes it, and the button with a pen icon allows renaming a mode. Finally, using the drop down menu each mode can be configured to inherit from one other mode by selecting its name or not inheriting from any mode by selecting the "None" entry.

A mode that inhertits from another is typicalled called a *child* and will perform the same actions as its *parent* when no actions are defined for a particular input.

Merge Axis

A screenshot of a computer

AI-generated content may be incorrect.

*This dialog allows the specification of two physical axis that should be combined into a single virtual one.*

This dialog allows the selection of any arbitrary two physical axis and the virtual axis their combined output should be assigned to.

Input Repeater

When enabling the *Input Repeater* option, while Joystick Gremlin is active, via *Tools > Input Repeater*, Joystick Gremlin will repeat the last remapped input after a short wait period. This allows button presses and axes movements to be mapped in-game when the game detects the physical device's inputs instead of the virtual ones. The status bar provides information about the current action of the input repeater when activated.

Device Information

A screenshot of a computer

AI-generated content may be incorrect.

*The device information dialogue provides an overview of the inputs and identification of the various joystick devices present in the system.*

Information about connected devices can be accessed via *Tools > Device Information*. This window provides an overview of the devices known to Joystick Gremlin, listing the name as well as some basic information about each device. The *Vendor ID* and *Product ID* represent the USB information about this device. The *GUID* is an identifier which is unique to each physical device on a particular Windows installation and is used by Gremlin to identify the different devices.

Calibration

A screenshot of a computer

AI-generated content may be incorrect.

*The calibration tool allows the calibration of all connected devices independent of Windows.*

The axes calibration screen, accessed via *Tools > Calibration*, allows the calibration of connected devices independent of Windows. The window shows the current, minimum, centre, and maximum value for each analogue axis of the selected device. To calibrate the axes simply move them through their full range of motions and press the *Centred* button when all axes are centred. Once done press *Save* to save the calibration of the currently selected device to the configuration file.

Input Viewer

A screenshot of a computer

AI-generated content may be incorrect.

*The input viewer allows visualizing the values of axes, buttons, and hats of all devices recognized by Gremlin.*

The input viewer allows visualizing the input from the physical as well as virtual devices. The window shows recognized devices and data visualizations for their inputs on the left hand side while the visualizations themselves will be shown in the right hand side of the window.

Cheatsheet Generation

To help remember complex setups Joystick Gremlin has the ability to create PDF and HTML cheatsheets of the current profile. The output lists for each mode the actions of the available devices together with the user provided description.

Create 1:1 Mapping

In order to make the initial setup quicker Joystick Gremlin can create a 1:1 mapping for the user. To do this select the entry from the *Actions* menu. This will result in each input being assigned the next free corresponding action, i.e. buttons are mapped to buttons and axes to axes.

Profile Creator

In order to enable easier sharing of configurations, even if different devices are used, the Profile Creator allows creating a new profile from an existing one. When started the tool asks for a profile to use as a template and then presents all the bound actions within to the user for binding to their particular devices.

A screenshot of a computer

AI-generated content may be incorrect.

The interface shows:

* Description of the action
* Type of actions used
* What input the action is currently bound to

Clicking on the button allows the user to press / move the input they wish to associate with the particular action. Once this is done the button will reflect this binding. When all desired actions are bound the configuration can be saved as a new profile by clicking on the save button.

This tool is only capable of binding actions defined via the UI and if custom modules are used in a profile those will not be automatically adapted to the new devices used.

Log viewer

A screenshot of a computer program

AI-generated content may be incorrect.

*The log viewer allows the user to easily check the content of the log file as well as clear it.*

The log viewer can be accessed via *Tools > Options* and allows easy viewing of both the system and user log files. The display is updated whenever the content of the log changes. Clearing the entire log is alos possible from this screen.

Swap Devices

While Gremlin should be able to always differentiate between physical devices there may be occasions when moving configurations associated with one device to another. This tool permits moving actions, macros, and conditions associated with one device to be moved to another connected one.

A screenshot of a computer

AI-generated content may be incorrect.

*This dialog allows swapping the actions assigned to identical devices between each other.*

3.9 Options

A screenshot of a computer

AI-generated content may be incorrect.  
A screenshot of a computer

AI-generated content may be incorrect.  
A screenshot of a computer

AI-generated content may be incorrect.

*The options dialogue which is used to set some default settings and configure the profiles associated with programs.*

There are several user controllable options in Joystick Gremlin which can be controlled via the options dialogue, accessible through *Tools > Options*.

General

The following options are accessible via the *General* tab:

**Highlight currently used input**

This option causes Joystick Gremlin to automatically select the input configuration matching the physical input's being used.

**Highlight swaps device tab**

With this option enabled, activating any physical input will swap to that device's tab if it is not the current one

**Closing minimizes to system tray**

Instead of quitting the application when it is closed, Joystick Gremlin is simply minimized to the system tray.

**Activate profile on launch**

This option causes Gremlin to activate the profile as soon as the program has started.

**Start Joystick Gremlin minimized**

Launches Joystick Gremlin minimized to the system tray instead of opening the main window.

**Show message when changing mode**

Displays a windows notification when the mode is changed.

**Default action**

Sets the action to be pre-selected in action selection drop down menus

**Macro axis polling rate**

This sets the rate at which axis values are recorded in the macro editor.

**Macro axis minimum change value**

This determines the minimum axis value change required to record a new entry in the macro editor.

Profiles

The *Profiles* tab exposes the following options:

**Automatically load profile based on current application**

This option makes Joystick Gremlin to automatically load and activate the profile assigned to the currently active application.

The drop down menu and input fields allow selecting the executable and the profile associated with it. When an executable specified in this list is active the profile associated with it is automatically loaded and activated. Once the profile stops being the active one Joystick Gremlin is automatically stopped.

The edit icon allows changing the path of the selected executable or to enter a regular expression that will be used to match against the foreground process. Joystick Gremln will first attempt to find a direct match for an existing file and once this fails attempt to use the remaining entries as regular expressions, picking the first successful match.

HidGuardian

HidGuardian is a tool that solves the problem of a game seeing both the physical device and the virtual device at the same time and processing these inputs in unwanted ways. HidGuardian is capable of hiding those physical devices from all non whitelisted programs, thus effectively allowing games to be fooled into believing only the virtual devices exist.

If Gremlin is started with admin priviliedges and [HidGuardian](https://github.com/ViGEm/HidGuardian) (ideally installed through [this tool](https://autohotkey.com/boards/viewtopic.php?t=34890)) is present this tab allows hiding and unhiding devices. Once a device's status was changed it has to be unplugged and replugged for HidGuardian to be able to use it.

This feature as well as HidGuardian are still in development. In fact HidGuardian is undergoing rework and as such newer versions will not work with Gremlin and this feature will be updated once HidGuardian has stabilized.

An alternative to HidGuardian, which is no longer in active development, is [HidHide](https://github.com/ViGEm/HidHide) from the same author. HiHide is in active development and appears to work well even without additional convenience functionality provided by Gremlin.

3.10 Command Line Options

Joystick Gremlin allows controlling the following options via command line parameters:

* Loading a specific profile
* Enabling the program on launch
* Starting Gremlin minimized

Profile Loading

By passing the --profile parameter followed by the path to the profile to load Joystick Gremlin will load the specified profile.

Enabling on Launch

Passing the --enable flag will enable Joystick Gremlin once the program has been fully loaded.

Start Minimized

Passing the --start-minimized option to Joystick Gremlin will cause it to launch minimized.

Last update: June 23, 2021

4. User Plugins

While common configuration tasks can be performed directly via the UI, more advanced and specialised configurations may either only be possible or much simpler by using a user plugin. Each plugin is a simple Python script which defines the variables a user can configure via the UI and the functions (callbacks) triggered in reaction to (configured) inputs being used. Since the functions are written in Python there is no limit as to what can be expressed. The following section assumes some basic familiarity with Python.

We start with a general overview of the layout of a user plugin in [Section 4.1](https://whitemagic.github.io/JoystickGremlin/user_plugins/#4_1) which is followed by an quick overview of the API in [Section 4.2](https://whitemagic.github.io/JoystickGremlin/user_plugins/#4_2) followed by the description of the decorator based callback system in [Section 4.3](https://whitemagic.github.io/JoystickGremlin/user_plugins/#4_3) with periodic function callbacks described in [Section 4.4](https://whitemagic.github.io/JoystickGremlin/user_plugins/#4_4). Some words on how to debug user plugins is provided in [Section 4.5](https://whitemagic.github.io/JoystickGremlin/user_plugins/#4_5). Finally, [Section 4.6](https://whitemagic.github.io/JoystickGremlin/user_plugins/#4_6) provides a few practical examples.

4.1 Principles & Layout of User Plugin

Joystick Gremlin uses callbacks, i.e. functions that are executed in reaction to user inputs such as key presses or axis motion. These callbacks have access to some convenience functions which allow accessing and controlling commonly used parts of the system, such as setting the value of vJoy devices or retrieving keyboard and joystick states. Combining these readily available functions with custom code allows the implementation of varied functionality.

In order to make user plugins reusable and convenient to use a set of classes exist which allow setting them via the UI, thus allowing a user to customize the plugins directly from the UI. These variable classes allow configuration of commonly used types such as modes, inputs, as well as numerical values.

The general structure of a callback is as follows:

@decorator\_function(<input name>)

def callback\_function(event, <optional parameter list>):

<callback implementation>

The event parameter contains information about the event that triggered the execution of the function. Each event is of type gremlin.event\_handler.Event and contains the following data:

Event

**event\_type**

The type of the event this represents.

**identifier**

The identifier of the event source.

**device\_guid**

Unique device ID associated with the device that created the event.

**is\_pressed**

If the event represents a button or key the value is True for pressed and False for released state.

**value**

Value of an axis or hat. In case of an axis the value is in the range [−1,1] and in the case of a hat a tuple (x direction, y direction) is used. This field's value is only valid for joystick axes and hats.

**raw\_value**

The raw axis value, this field is only valid for joystick axes.

From this list the only values that are typically of interest are the is\_pressed and value entries depending on the input type.

4.2 Device Access API

The following describes the API of the optional variables exposed via the decorator plugin framework. The plugins provide access to commonly used information by simply adding a properly named parameter to the callback function.

These parameters must be listed after the event parameter in the case of user input callbacks.

vJoy

Any decorated function that has a parameter named vjoy in its parameter list will have access to all vJoy devices. Accessing a specific VJoy instance is done by indexing the vjoy object. This object then allows setting the state of inputs by indexing the member variables axis, button, and hat. Indices of buttons and hats start at 1. For axes the indices correspond to the axis index as defined by the device. The following demonstrates typical usage:

# Access the first vJoy device and press the third button

vjoy[1].button(3).is\_pressed = True

# Access the second vJoy device and move the Y axis to -0.25

vjoy[2].axis(AxisName.Y).value = -0.25

# or equivalently

vjoy[2].axis(2).value = -0.25

# Access the first vJoy device and move the first hat to

# the top right position

vjoy[1].hat(1).direction = (1, 1)

Joystick State

Any decorated function that has a parameter named joy in its parameter list will have access to all joystick devices via that variable.

**Accessing a specific joystick**  
In order to access a specific joystick its system id needs to be known. Using the device's system id as index the joystick can be accessed by:

joystick\_device = joy[device\_guid]

**Reading axis value**  
To read the current value of a joystick axis both the index of the axis as well as the system id of the joystick, starting with 1, are needed, with these the axis value is obtained as:

axis\_value = joy[device\_guid].axis(axis\_index).value

**Reading button state**  
To read the current state of a button both the joystick's system id as well as index of the button, starting at 1, are needed. The following then reads the button state:

state = joy[device\_guid].button(button\_id).is\_pressed

**Reading hat position**  
To read the current position of a hat both the joystick's system id and hat index, starting at 1, are needed. The position of the hat is reported as a (x,y) tuple x,y∈{−1,0,1}. A x value of 1 is right and -1 left while a value of 1 for y means up and -1 down. A value of 0 represents a centred position. The value is read as follows:

position = joy[device\_guid].hat(hat\_id).direction

Keyboard State

Any decorated function that has a parameter named keyboard in its parameter list will have access to the state of all keyboard keys.

**Reading key state**  
To read the key state the string representation of the key or the gremlin.macro.Key instance corresponding to the key is needed. Both can be found in the gremlin.macro module. Reading the state is then done as follows:

is\_pressed = keyboard.is\_pressed(key)

4.3 User Input Callback Generation

Callbacks reacting to user inputs are created by decorating functions using specific decorators. Here are two useful links if you're not familiar with decorators, [official PEP](https://www.python.org/dev/peps/pep-0318/) and an [exhaustive StackOverflow answer](https://stackoverflow.com/questions/739654/how-to-make-a-chain-of-function-decorators/1594484#1594484) There are two types of decorators, one for joysticks and one for the keyboard. Joystick decorators are created for specific devices using the gremlin.input\_devices.JoystickDecorator class as follows:

joystick\_decorator = gremlin.input\_devices.JoystickDecorator(

"<device name>",

"{<device guid>}",

"<mode>"

)

The value of device\_guid is the unique dentifier of the device. An object created in this way has three decorators customised for the given joystick and mode, which can be used as follows:

@joystick\_decorator.axis(1)

def axis\_callback(event):

pass

@joystick\_decorator.button(4)

def button\_callback(event):

pass

@joytick\_decorator.hat(2)

def hat\_callback(event):

pass

The keyboard decorator can be used directly as follows:

@gremlin.input\_devices.keyboard(<key name>, <mode>)

def keyboard\_callback(event):

pass

Where key name can be either a string representation of the key's name as or an instance of gremlin.marco.Key which are both defined in the gremlin.macro module.

The event parameter of the decorated function is always required and contains the state of the input that triggered the callback, the contents of the variable are described in [Section 4.1](https://whitemagic.github.io/JoystickGremlin/user_plugins/#4_1).

4.4 Periodic Function Callbacks

In some situations a function needs to be executed at regular intervals. This is facilitated by a decorator that ensures that the function is run at a specified interval while Joystick Gremlin is active.

The decorator takes a single argument that indicates the interval, i.e. the duration, between executions of the function in seconds. The callback function can use the same plugin system as the user input callbacks to gain access to device information, e.g. vjoy, joy, and keyboard. A generic example of periodic function callback is shown below.

@gremlin.input\_devices.periodic(<seconds>)

def periodic\_function():

pass

4.5 Configurable Variables

In order to allow user plugins to be configured by users via the UI several types of variable classes exist, that are automatically extracted from the plugin and presented to the user for customization.

The following variable types exist and will be explained in more detail below.

**IntegerVariable**

Holds a single integer value.

**FloatVariable**

Holds a single float value.

**BoolVariable**

Holds a single boolean value.

**StringVariable**

Holds an arbitrary length string.

**ModeVariable**

Holds the value of one of the modes existing in the profile.

**VirtualInputVariable**

Holds one vJoy selection.

**PhysicalInputVariable**

Holds one physical device input.

IntegerVariable

This variable can hold any single integer value and presents the user with a field which allows entering of values. The variable also allows the specification of limits for valid values.

gremlin.user\_plugin.IntegerVariable.\_\_init\_\_

**label**

mandatory

Label shown for the UI element.

**description**

mandatory

Text describing the purpose of the variable.

**initial\_value**

optional

Default value to use for this variable.

**min\_value**

optional

Minimum value this variable can take on.

**max\_value**

optional

Maximum value this variable can take on.

FloatVariable

This variable can hold any single floating point value and presents the user with a field which allows entering of values. The variable also allows the specification of limits for valid values.

gremlin.user\_plugin.FloatVariable.\_\_init\_\_

**label**

mandatory

Label shown for the UI element.

**description**

mandatory

Text describing the purpose of the variable.

**initial\_value**

optional

Default value to use for this variable.

**min\_value**

optional

Minimum value this variable can take on.

**max\_value**

optional

Maximum value this variable can take on.

BoolVariable

This variable can hold any single boolean value and presents the user with a checkbox. This can be used to turn features of a plugin on and off.

gremlin.user\_plugin.BoolVariable.\_\_init\_\_

**label**

mandatory

Label shown for the UI element.

**description**

mandatory

Text describing the purpose of the variable.

**initial\_value**

optional

Default value to use for this variable.

StringVariable

This variable can hold any string and presents the user with a text input field.

gremlin.user\_plugin.StringVariable.\_\_init\_\_

**label**

mandatory

Label shown for the UI element.

**description**

mandatory

Text describing the purpose of the variable.

**initial\_value**

optional

Default value to use for this variable.

ModeVariable

This variable holds the name of one mode present in this profile. The user is presented with a dropdown list containing all modes that exist.

gremlin.user\_plugin.ModeVariable.\_\_init\_\_

**label**

mandatory

Label shown for the UI element.

**description**

mandatory

Text describing the purpose of the variable.

VirtualInputariable

This variable holds one specific vJoy input selection. The user is presented with the typical vJoy dropdown boxes.

gremlin.user\_plugin.VirtualInputVariable.\_\_init\_\_

**label**

mandatory

Label shown for the UI element.

**description**

mandatory

Text describing the purpose of the variable.

**valid\_types**

optional

List of valied gremlin.common.InputType values

PhysicalInputariable

This variable holds one specific physical input device selection. The user can press a button at which point Gremlin will record the physical input being activated next.

gremlin.user\_plugin.PhysicalInputVariable.\_\_init\_\_

**label**

mandatory

Label shown for the UI element.

**description**

mandatory

Text describing the purpose of the variable.

**valid\_types**

optional

List of valied gremlin.common.InputType values

Example

This example plugin lets a user specify four physical joystick buttons and map them to a single virtual hat output.

import gremlin

from gremlin.user\_plugin import \*

mode = ModeVariable(

"Mode",

"The mode to use for this mapping"

)

vjoy\_hat = VirtualInputVariable(

"Output Hat",

"vJoy hat to use as the output",

[gremlin.common.InputType.JoystickHat]

)

btn\_1 = PhysicalInputVariable(

"Button Up",

"Button which will be mapped to the up direction of the hat.",

[gremlin.common.InputType.JoystickButton]

)

btn\_2 = PhysicalInputVariable(

"Button Right",

"Button which will be mapped to the right direction of the hat.",

[gremlin.common.InputType.JoystickButton]

)

btn\_3 = PhysicalInputVariable(

"Button Down",

"Button which will be mapped to the down direction of the hat.",

[gremlin.common.InputType.JoystickButton]

)

btn\_4 = PhysicalInputVariable(

"Button Left",

"Button which will be mapped to the left direction of the hat.",

[gremlin.common.InputType.JoystickButton]

)

state = [0, 0]

decorator\_1 = btn\_1.create\_decorator(mode.value)

decorator\_2 = btn\_2.create\_decorator(mode.value)

decorator\_3 = btn\_3.create\_decorator(mode.value)

decorator\_4 = btn\_4.create\_decorator(mode.value)

def set\_state(vjoy):

device = vjoy[vjoy\_hat.value["device\_id"]]

device.hat(vjoy\_hat.value["input\_id"]).direction = tuple(state)

@decorator\_1.button(btn\_1.input\_id)

def button\_1(event, vjoy):

global state

state[1] = 1 if event.is\_pressed else 0

set\_state(vjoy)

@decorator\_2.button(btn\_2.input\_id)

def button\_2(event, vjoy):

global state

state[0] = 1 if event.is\_pressed else 0

set\_state(vjoy)

@decorator\_3.button(btn\_3.input\_id)

def button\_3(event, vjoy):

global state

state[1] = -1 if event.is\_pressed else 0

set\_state(vjoy)

@decorator\_4.button(btn\_4.input\_id)

def button\_4(event, vjoy):

global state

state[0] = -1 if event.is\_pressed else 0

set\_state(vjoy)

4.6 Debugging

To facilitate the debugging of custom modules without setting up the source code of Joystick Gremlin in an IDE the logging function gremlin.util.log() can be used. This stores the provided text to the user log file which can be viewed directly in Joystick Gremlin via the *Tools -> Log display* option.

For more detailed debugging Joystick Gremlin needs to be run from within an IDE by getting the development environment setup. While this provides the best debugging experience it also involves the most work, thus for simple tasks the logging approach may be preferable.

4.7 Examples

In the following a few examples of custom modules are shown. They provide an illustration of some of the things that can be achieved thanks to the combination of Joystick Gremlin provided functions and custom Python code.

Keyboard Controlled Throttle

This script allows the user to control an analogue throttle in 1/3rd increments using the 1, 2, 3, and 4 number keys.

import gremlin

from vjoy.vjoy import AxisName

def set\_throttle(vjoy, value):

vjoy[1].axis(AxisName.Z).value = value

@gremlin.input\_devices.keyboard("1", "Global")

def throttle\_0(event, vjoy):

if event.is\_pressed:

set\_throttle(vjoy, -1.0)

@gremlin.input\_devices.keyboard("2", "Global")

def throttle\_33(event, vjoy):

if event.is\_pressed:

set\_throttle(vjoy, -0.33)

@gremlin.input\_devices.keyboard("3", "Global")

def throttle\_66(event, vjoy):

if event.is\_pressed:

set\_throttle(vjoy, 0.33)

@gremlin.input\_devices.keyboard("4", "Global")

def throttle\_100(event, vjoy):

if event.is\_pressed:

set\_throttle(vjoy, 1.0)

Joystick Response Curve

This script configures a response curve which provides more control around the centre position and uses it for the X and Y axis of the joystick.

import gremlin

from gremlin.spline import CubicSpline

from vjoy.vjoy import AxisName

chfs = gremlin.input\_devices.JoystickDecorator(

"CH Fighterstick USB",

2382820288,

"Global"

)

curve = CubicSpline([

(-1.0, -1.0),

(-0.5, -0.25),

( 0.0, 0.0),

( 0.5, 0.25),

( 1.0, 1.0)

])

@chfs.axis(1)

def pitch(event, vjoy):

vjoy[1].axis(AxisName.X).value = curve(event.value)

@chfs.axis(2)

def yaw(event, vjoy):

vjoy[1].axis(AxisName.Y).value = curve(event.value)

Mode Switching

This script presents a few different ways of using mode switching functionalities. The first callback switches to the *Radio* mode while the button is being held down and switches back to the previous mode once the button is released. The next callback cycles through the *Global*, *Radio*, and *Landing* modes with each button press. The last callback switches directly to the *Global* mode when the button is pressed.

import gremlin

chfs = gremlin.input\_devices.JoystickDecorator(

"CH Fighterstick USB",

2382820288,

"Global"

)

mode\_list = gremlin.control\_action.ModeList(

["Global", "Radio", "Landing"]

)

@chfs.button(10)

def temporary\_mode\_switch(event):

if event.is\_pressed:

gremlin.control\_action.switch\_mode("Radio")

else:

gremlin.control\_action.switch\_to\_previous\_mode()

@chfs.button(11)

def cycle\_modes(event):

if event.is\_pressed:

gremlin.control\_action.cycle\_modes(mode\_list)

@chfs.button(12)

def switch\_to\_global(event):

if event.is\_pressed:

gremlin.control\_action.switch\_mode("Global")

Precision Mode

This script switches to a lower sensitivity curve when any of the weapon groups are being fired and switches back to the default profile once no weapon is being fired any more. This is similar to the "sniper mode" that some gaming mice have, which drops the DPI setting at the press of a button. In this instance pressing the trigger automatically enables and disables this by switching the used response curve to one which halves the maximum response provided by the joystick at maximum deflection.

import gremlin

from gremlin.spline import CubicSpline

from vjoy.vjoy import AxisName

tm16000 = gremlin.input\_devices.JoystickDecorator(

"Thrustmaster T.16000M", 1325664945, "Global"

)

default\_curve = CubicSpline(

[(-1.0, -1.0), (0.0, 0.0), (1.0, 1.0)]

)

precision\_curve = CubicSpline(

[(-1.0, -0.5), (0.0, 0.0), (1.0, 0.5)]

)

active\_weapon\_groups = {}

active\_curve = default\_curve

def set\_weapon\_group(gid, is\_pressed):

global active\_curve

global active\_weapon\_groups

if is\_pressed:

active\_curve = precision\_curve

active\_weapon\_groups[gid] = True

else:

active\_weapon\_groups[gid] = False

if sum(active\_weapon\_groups.values()) == 0:

active\_curve = default\_curve

@tm16000.button(1)

def weapon\_group\_1(event, vjoy):

set\_weapon\_group(1, event.is\_pressed)

vjoy[1].button(1).is\_pressed = event.is\_pressed

@tm16000.button(2)

def weapon\_group\_2(event, vjoy):

set\_weapon\_group(2, event.is\_pressed)

vjoy[1].button(2).is\_pressed = event.is\_pressed

@tm16000.button(3)

def weapon\_group\_3(event, vjoy):

set\_weapon\_group(3, event.is\_pressed)

vjoy[1].button(3).is\_pressed = event.is\_pressed

@tm16000.axis(1)

def pitch(event, vjoy):

vjoy[1].axis(AxisName.X).value = active\_curve(event.value)

@tm16000.axis(2)

def yaw(event, vjoy):

vjoy[1].axis(AxisName.Y).value = active\_curve(event.value)

Last update: June 23, 2021

5. Technical Design

This section contains information related to implementation details of Joystick Gremlin. This is mainly intended for those that want to get a deeper understanding of the internals or extend Joystick Gremlin in some way. The following provides an overview of the various components that are part of Joystick Gremlin. The components can roughly be split into two categories: functional and user interface.

5.1 Input Handling

User inputs are captured using [SDL2](https://www.libsdl.org/) for joystick devices while keyboard and mouse inputs are obtained by hooking from Windows' event queue. Both types of events are processed by gremlin.event\_handler.EventListener which converts the input events into Qt signals which other interested parts of Gremlin can be connected to. This makes use of the [Signal & Slots](http://doc.qt.io/qt-5/signalsandslots.html) mechanism from Qt.

Event

**event\_type**

The type of the event this represents.

**identifier**

The identifier of the event source.

**hardware\_id**

Hardware ID assigned to the deivce that created the event.

**windows\_id**

Index assigned by Windows to the device that created the event.

**is\_pressed**

If the event represents a button or key the value is True for pressed and False for released state.

**value**

If the event represents an axis or hat the value contains the state.

5.2 Macro System

Each macro consists of a sequence of actions which are executed one after the other. Each macro is run in a separate thread in order to prevent Joystick Gremlin to become unresponsive.

In order to be provide some control over the execution of macros there exists a macro manager which handles the queueing and dispatching of macros. This permits some macros to be run in an exclusive mode, i.e. only the particular macro can be active and no other macro can run during the life time of the exclusive macro.

The various modes in which a macro can be executed, i.e. repeat, hold, and toggle, are implemented inside the manager which takes care of repeating the macro and terminating it when required.

5.3 Action and Container System

Actions and containers work in a similar fashion under the hood and as such share quite a few design traits. For each action or container there are three different classes, each with their own purpose:

* A data storage class derived from gremlin.base\_classes.AbstractAction or gremlin.base\_classes.AbstractCtonainer. This class defines the values stored by the action or container and loads them from an XML profile and stores them to it as well.
* A UI widget class derived from gremlin.ui.input\_item.AbstractActionWidget or gremlin.ui.input\_item.AbstractContainerWidget. This class presents the data of the instance to the user and allows modifying the contents, storing changes back into the data storage class.
* A functor class derived from gremlin.base\_classes.Functor which implements the execution logic of the action or container based on the contents of the data class.

Interface Definitions

The following presents the interfaces that are inherited from when creating a new action or container in order to provide the functionality of the three classes outlined above.

AbstractAction

**from\_xml(node)**

Populates the instance based on the XML node's content.

**to\_xml()**

Returns an XML node representing the instance's content.

**icon()**

Returns the path to the icon indicating this action.

**requires\_virtual\_button()**

Returns True if this instance requires a virtual button to be used.

AbstractContainer

**\_parse\_xml(node)**

Populates the instance based on the XML node's content.

**\_generate\_xml()**

Returns an XML node representing the instance's content.

**\_is\_container\_valid()**

Returns whether or not the contents of the container result in a valid setup.

AbstractActionWidget

**\_create\_ui()**

Creates all the necessary UI elements to show action elements.

**\_populate\_ui()**

Populates the UI elements Creates all the necessary UI elements for condition elements.

AbstractContainerWidget

**\_create\_action\_ui()**

Creates all the necessary UI elements to show action elements.

**\_create\_condition\_ui()**

Creates all the necessary UI elements for condition elements.

**\_get\_window\_title()**

Returns the string to use for the window title.

AbstractFunctor

**process\_event(event, value)**

Executes the action's or container's logic for the provided event and value inputs.

5.4 XML Profile Storage

Each profile is stored in a single XML file. The contents of the XML file are used to populate the UI as well as generate Python code representing the profile. The file is parsed by the gremlin.profile.Profile class. Parsing of action or container content is performed by the corresponding data storage classes.

6. Examples

In the following simple usage examples of Joystick Gremlin will be shown. These should give you an idea of what is possible and illustrate the use of the UI.

Shift Button

Sometimes it is desirable to have a single physical button to have different functionality based on whether or not another button is pressed. There are two ways to achieve this with different advantages and disadvantages. These two approaches differ in how the shift functinoality is achieved:

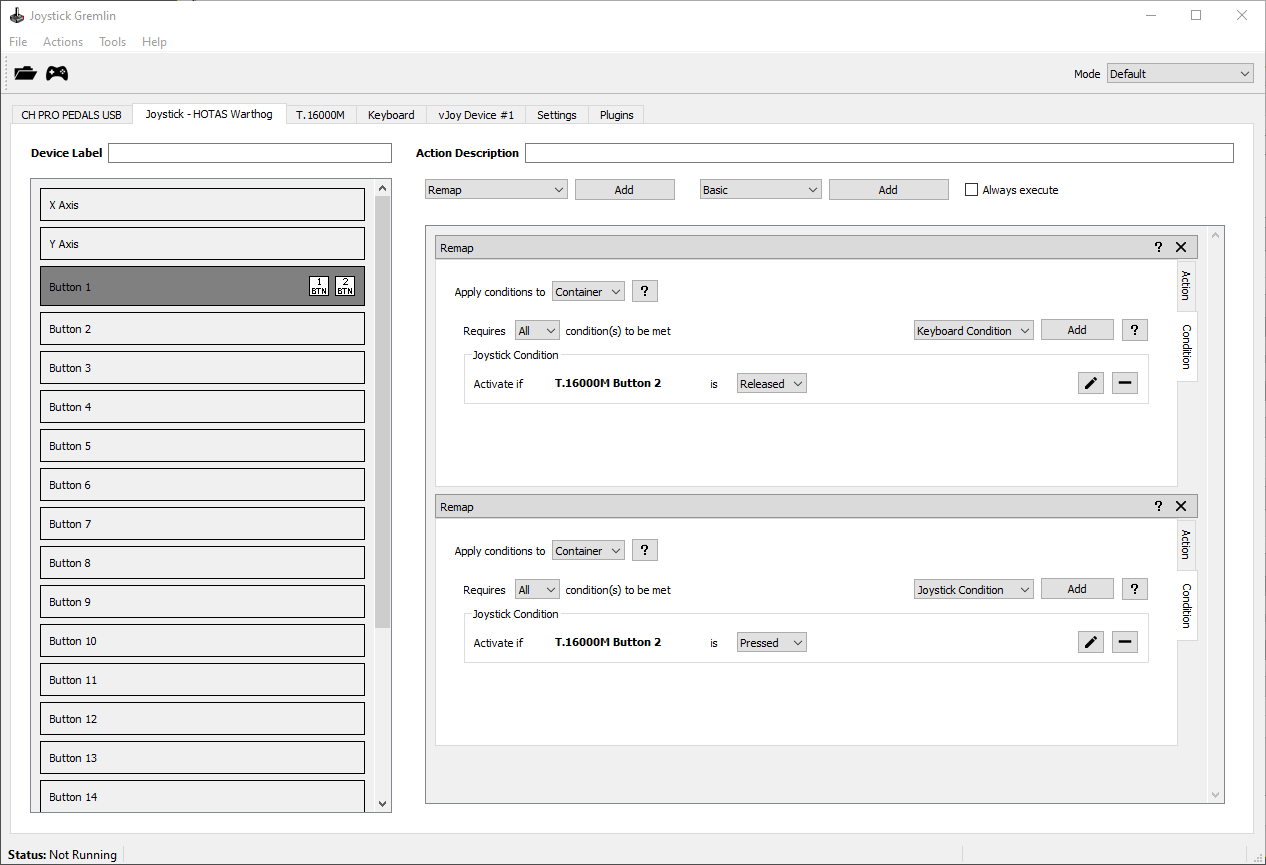
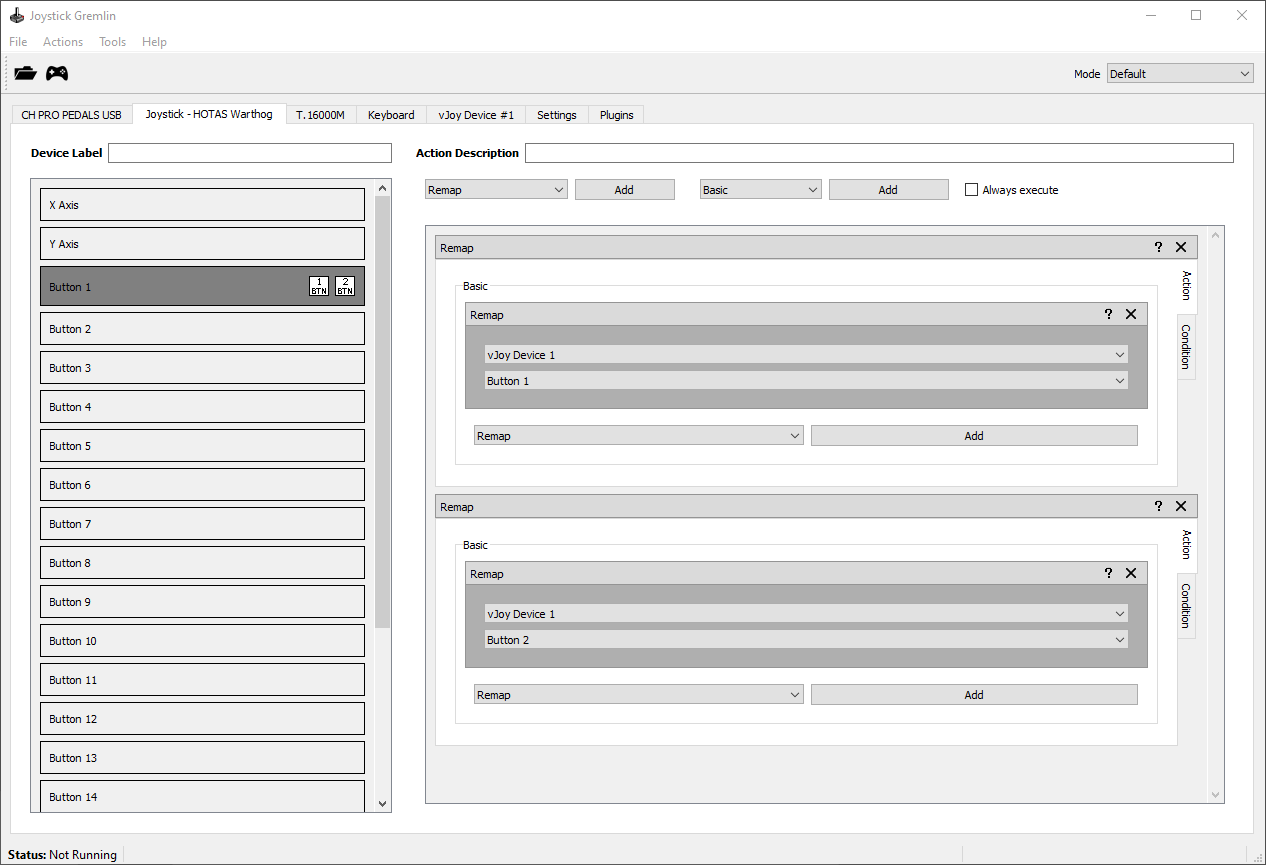
1. Condition
2. Mode

In the following, these two approaches are explained. In the example, *Warthog, Button 1* is the input with a shifted functionality, and *T16000.M Button 2* is the shift button. The default, i.e. unshifted, action is to a remap to *vJoy Button 1* while the shifted action is a remap to *vJoy Button 2*.

Condition-Based Shift Button

In this setup the shift functionality is achieved by having Gremlin do different things using conditions to check if the shift button is pressed or not. The two remap actions are added inside their own containers, to allow a container condition to be added to each of them. In both cases the condition checks the state of *T16000.M Button 2*. If that button is not pressed the first remap action is executed, and if not the second one. It is paramount that both the shifted as well as the default action have a condition associated with them, otherwise the action lacking a condition will be executed independent on the state of the shift button.

The advantage of this setup is that you can directly see what both the default and shifted actions do. The downside is that this can get a bit tedious when having many shifted actions. As a result, this approach is usually preferred when only one or two inputs need to be shifted via a particular shift button.

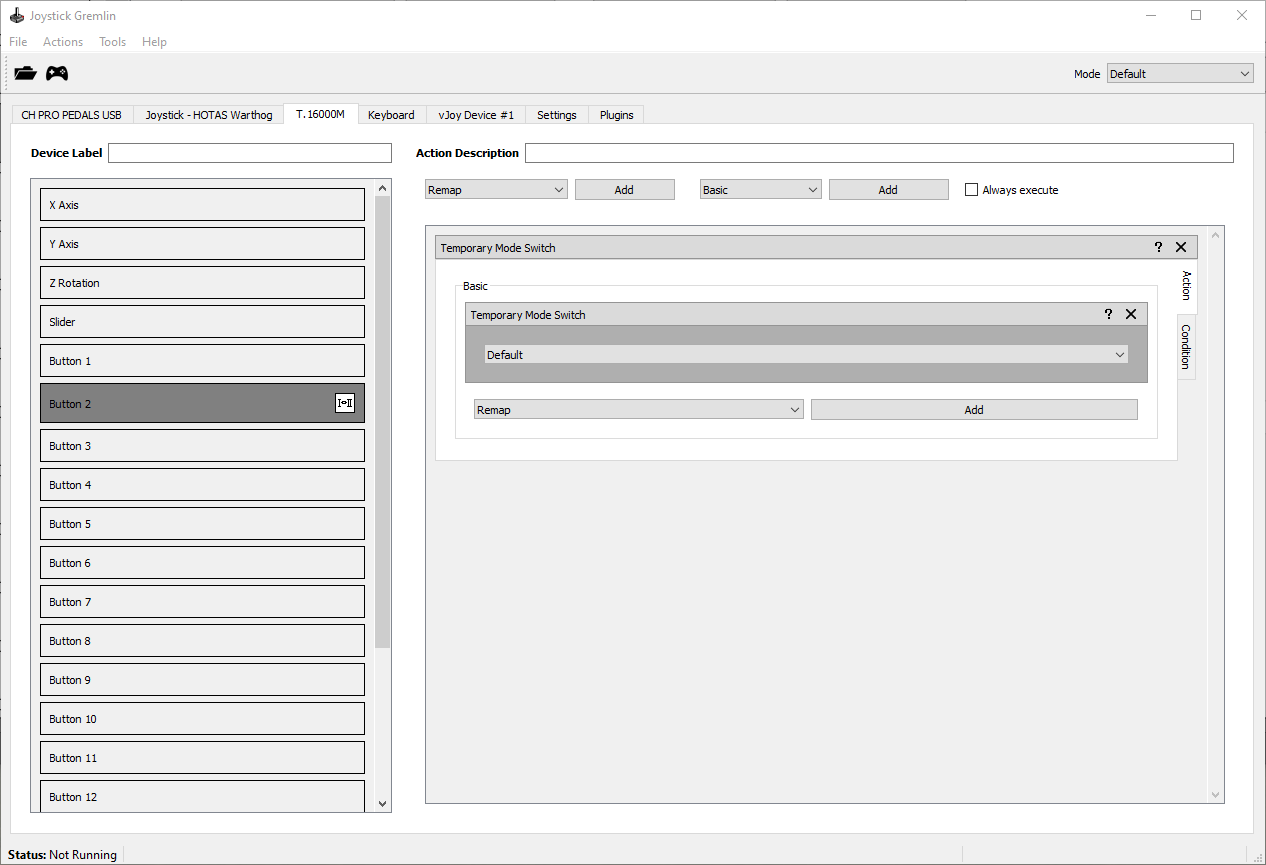
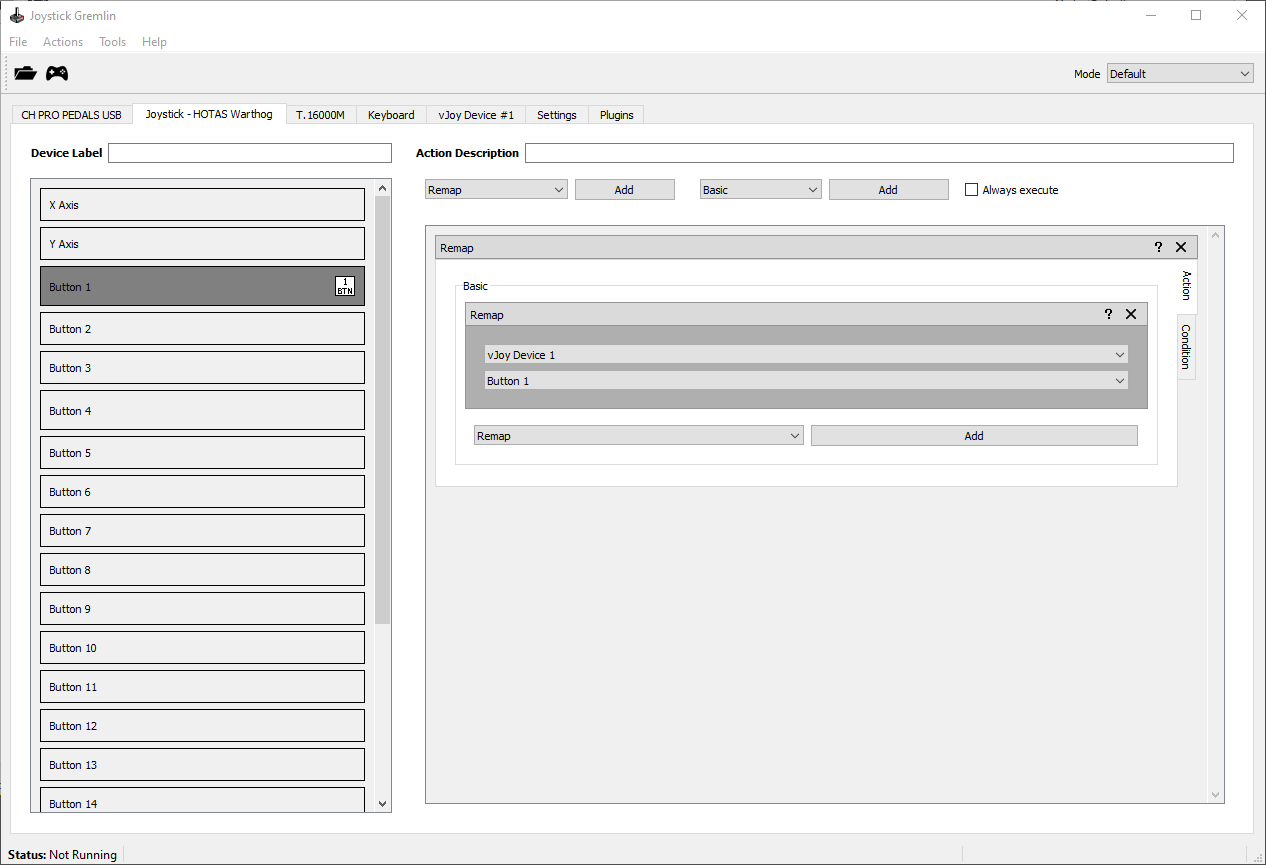


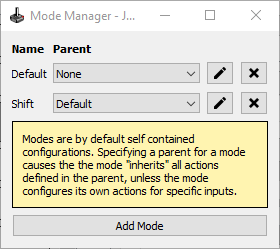
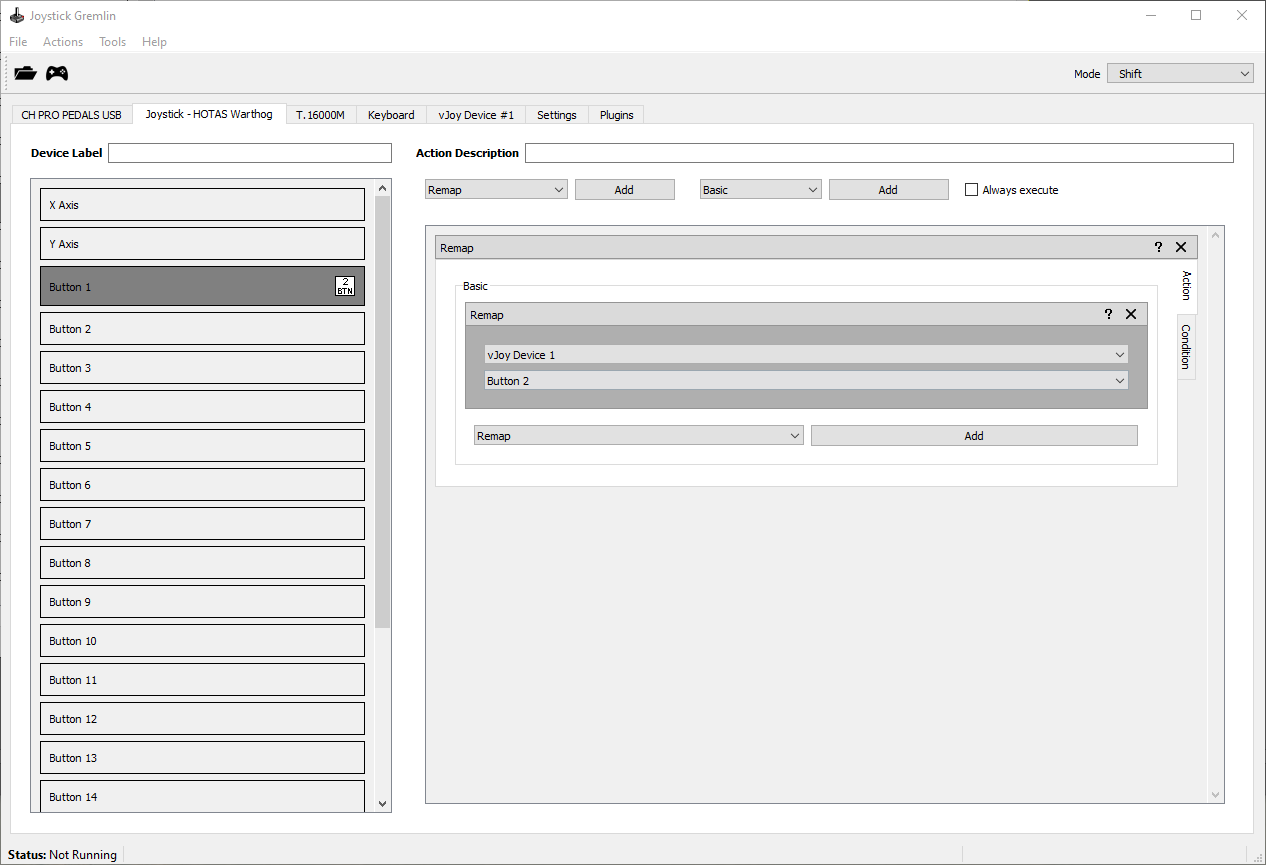
*The above image shows how the example shift setup looks like when using conditions.*

Temporary Mode Shift Button

The other approach uses the mode system of Gremlin to use an entire mode to hold the shifted functionality. To this end a new mode, *Shift* in this example, is added. Crucially, this new mode is a parent of the *Default* mode, which ensures that all *Default* mode actions are still executed while in the *Shift* mode, with the exception of shifted buttons. The shift button *T16000.M Button 2* is assigned a *Temporary Mode Switch* action which will activate the *Shift* mode for as long as that button is pressed. The *Default* mode simply contains the default actions while the *Shift* mode holds the shift behaviour.

The advantage of this method is that adding shifted actions is straight forward and one can easily see which inputs have a shift action associated with them. The disadvantage is that this is only really suitable for profiles with a single shift button, as having different buttons act as different shift modes can quickly make for a hard to understand setup. As such this is mainly useful when one wants to use a single button to implement an entire shift mode.





*The above image shows how the example shift setup looks like when using a dedicated mode.*

Last update: August 18, 2021