17th March 2018

Document issue : 1

Universal Joystick Programming Software: UJPS C++

V1.2.0 MANUAL

CONTENTS

1 Introduction

Raw joysticks

By "raw joysticks" we mean joysticks used the simplest way: you plug your joystick, the system detects it and recognize its buttons and axes. Then you go in the control options of your software / game and map the buttons and axes to "in-game actions / controls".

Programmable or programmed joysticks?

Beyond that there is also what we could call the "programmed joysticks". Many manufacturers called them "programmable joysticks", but I don't like this word as their joysticks are no more programmable as any other ones: they are programmable only because the manufacturer provides a software especially to program them, but there is nothing intrinsically related to the joystick. And with UJPS C++ you will be able to program any joystick recognize by the system as a game controller.

Raw joysticks vs programmed joysticks

With raw joysticks you are limited to bind a button or axis to an "action" provided in the game control options menu. You cannot for example use one button for different actions without interference. Moreover the list of actions provided is not always suited for all types of controls, and with programmed joysticks you can do ~~almost~~ what you want between the physical joysticks and what sees the game. You can also make different joysticks interact with each other. This way you can create more ergonomic controls than just with raw joysticks. You can even sometimes fill gaps in the game controls by "programming" your joysticks.

The state of art

The manufacturers that do the most for "programmable" joysticks are CH and Thrustmaster (among general public joysticks). They both provide 2 different ways to program joysticks:

- One first and easier way using graphical interfaces (e.g. with CH Control Manager Programming Software, or Thrustmaster TARGET GUI)
- Another more advanced and less easy way using real programming or scripting (with CH scripts, or Thrustmaster scripts that are written in C programming language using a Thrustmaster provided functions library)

On the other side some free software exist to do so with joysticks of any brand but only using graphical interfaces:

- UCR (Universal Control Remapper) : https://github.com/evilC/UCR
- Joystick Gremlin : https://whitemagic.github.io/JoystickGremlin/

UJPS C++

The present software allows to program joysticks of any brand, on Windows only, by programming in C++. Even if some knowledge of C++ is needed to get the best out of it, only basic C++ knowledge should be necessary to obtain good results. Anyway this software is quite similar to Thrustmaster scripts, is not harder than it, is a little more flexible and mostly allow to program any joystick.

2 The main principles

As all other softwares of this type, UJPS relies on virtual joysticks. These virtual joysticks are seen by the system just as real joysticks. The principle of UJPS is that the inputs of your joysticks are read by the software and the profile you programmed transform these inputs into virtual joysticks actions or keyboard keystrokes. The profile is interchangeable and is chosen in the main window.

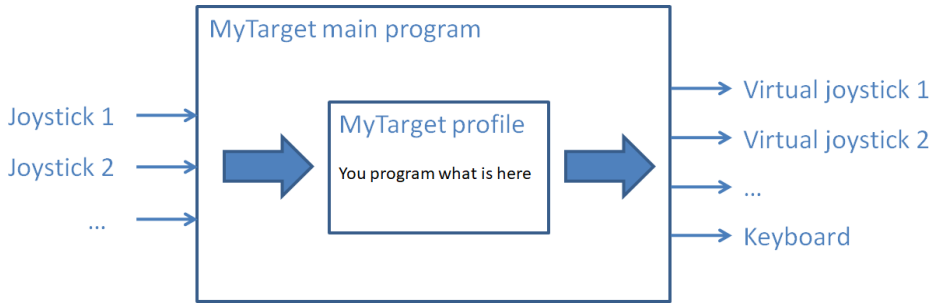

Figure 1: UJPS main principle

As a comparison, the Thrustmaster scripts allow only one virtual joystick and also supports mouse inputs. In its current state the software has also some limitations (see §12 for more details) and interesting ones for now are:

- the tool is available only for Windows, because vJoy works only on Windows
- real joysticks are still seen by the system and that is quite annoying for settings controls in the control options menu of game (see §12 for a possible solution)

UJPS profiles are in fact plugins (dll that can be load dynamically at runtime) that must be compiled before running them. Creating a profile is described in §5.

UJPS software uses external tools:

- Qt 5, a general C++ library
- vJoy, for virtual joysticks: http://vjoystick.sourceforge.net/site/
- QtGameController, a non-official Qt module: https://github.com/openforeveryone/QGameController

And I recommend you to use "devorder" from Brian Kendall to hide real joysticks to the game (which is really easier to use than VIGEm). The necessary files are provided in the download of this tool, but here is the link for more information:

https://github.com/briankendall/devreorder

Read §3 before installing anything.

I would like to thank "Shaul" for vJoy that allowed this software and many others (among them UCR and joystick Gremlin) to exist.

To create UJPS profiles you will need to install Qt 5 and vJoy (see §3 for more details). Only vJoy is needed if you want to run profiles created by other users.

3 Installation

3.1 External tools installation

You will need to install Qt 5 and vJoy. For Qt 5 I will even recommend you to use Qt 5.7 with the mingw C++ compiler as this is exactly the one used to code this software. And download the last version of vJoy. This software has been developed with the 2.1.8 release from November 2016, which still was the last release in beginning of 2018. If a new version is released, updating the "vJoy218SDK-291116" directory may be necessary and anyway advised. I would provide an update in this case.

https://download.qt.io/archive/qt/5.7/5.7.1/

http://vjoystick.sourceforge.net/site/index.php/download-a-install

3.2 Build the application

Once these 2 steps are done, do the following:

- Verify if Qt paths are included the "PATH" environment variable. To do so you can open a Windows console (cmd.exe) and type "set PATH".
    - if yes, open the "build_ UJPS.bat" file as text (don't double-click on it) and remove the lines 3 to 6
    - if not, modify the line 5 with the 2 Qt paths matching your own Qt setup. You should find a "qtenv2.bat" file in the Qt directories that contains them
- Close the " build_ UJPS.bat" and run it (double-click on it). It will
    - compile and deploy the "CleanQtProjects" utility
    - compile the static libraries
    - compile and deploy the main program "UJPS"
    - compile and deploy the additional programs "Monitoring" and "AxesCurves"

If all went correctly, your installation is complete. If you want, for practical reasons, you can create shortcuts to the 3 following executables:

- PROGRAM/release/UJPS.exe
- MONITORING/release/Monitoring.exe
- AXES_CURVES/release/AxesCurves.exe

The "cleanAll.bat" allows you to clean (restore as the beginning by removing the binaries, the makefiles, ... but keeping source code, resources, .pro file, ...) the directories "STATIC_LIBS", "PROFILES", "PROGRAM" and "MONITORING".

The other files "build_xxxx.bat" will build the provided test profile that is in the "PROFILES" directory. But before running it, do the same operation on the first lines as you did for "build_UJPS.bat".

4 The main window

The main window pops up when running " UJPS.exe". It is quite basic and provides the following functionalities:

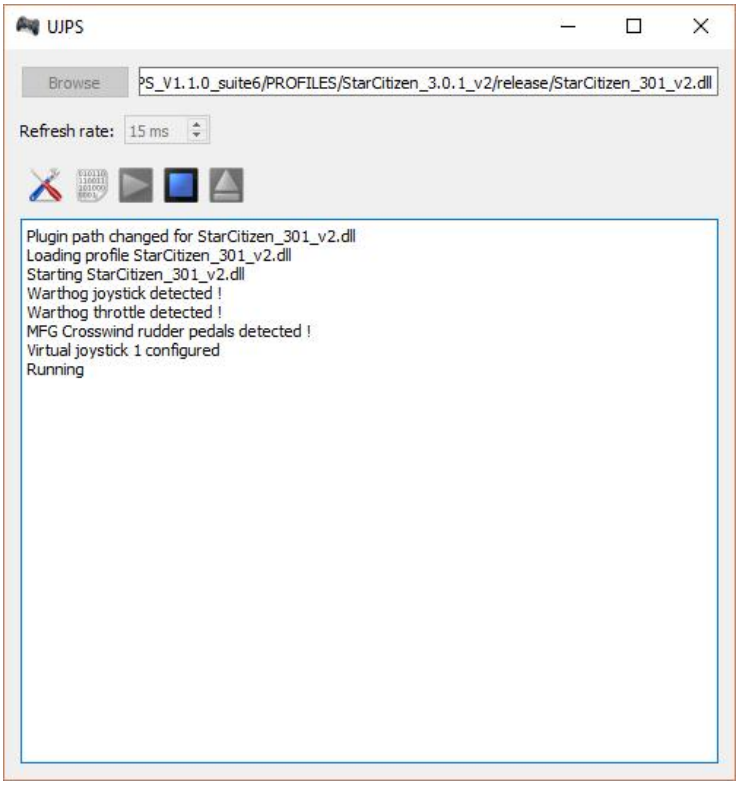

Figure 2: UJPS main window

- browse to select the profile to run. The profile is not loaded right away
- select a time step. The default one is 15 ms. At each time step the data of real joysticks is read, transformed by the profile and the result is sent to virtual joysticks and / or keyboard
- show the settings menu. You especially needs to specify the path to the "vJoyConfig" executable so that the setup of the virtual joysticks succeeds
- compilation: if you installed Qt with the MinGw compiler, you can use directly this feature to compile your profile without using an IDE or compile with the command line. But UJPS does not provide a text editor to modify the profile source code
- play button: the profile / plugin is loaded when it is played if it is not already loaded. If another one is already loaded, it is unloaded before loading the new one
- stop button: pause the profile
- unload button: unload the profile. It is not needed before selecting a new profile. Its purpose is only to unload the profile in order to be able to recompile it after a code modification. Otherwise the profile compilation will fail because the dll cannot be rewritten. It is very useful when editing the profile
- a text zone to send messages and errors

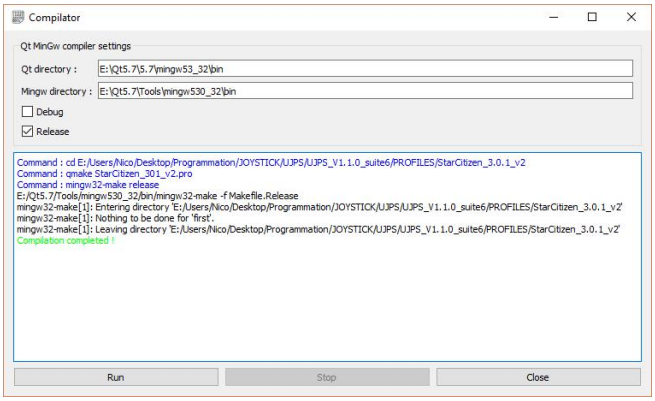

Figure 3: compiler widget

Here is a list of the highly probable errors you could encounter:

- an error when loading the profile. It can be due to several reasons:
  - the file does not exist anymore or you don't have read access on it
  - the dll file specified is not a UJPS profile

- an error due to the initialization of the profile: it may be due to the fact that you did not run the program as an administrator. Indeed currently the configuration of virtual joysticks is done in the program (even in the plugins in fact), not outside of it. To do so the main program "UJPS.exe" calls "vJoyConfig.exe" and this one needs administrator privileges to interact with the vJoy driver

- errors due to programming issues in the profile that passed the compilation

5 Create a profile (plugin)

5.1 Profile creation (by manual copy of the provided example)

This step is not yet automated by the main window, so for now you will need to do it manually:

- create a new directory next to the provided example. If you create it somewhere else, you will need to modify some paths in the .pro file
- copy the .pro file for the example. Rename it as you want. Open it and replace the line 2 (the one with "TARGET = ..."), by "TARGET = theNameYouWantForTheProfileDll"
- also copy the CODE directory of the example

5.2 Profile coding

The easier way to do it is to use one of the provided examples as a starting base. You have to create a class derived from AbstractProfile and implement the pure virtual functions "play" and "stop". Programmatically speaking, that is the only requirement.

function play

This function is executed every time the play button is pressed in the main window. It can be executed several times on the same object (but you have to press stop between 2 presses on play). Your implementation of this function should do the following operations:

- 1. Setup the real joysticks:
    - a. Allocate dynamically using the "new" operator (not on the stack, because of a current limitation in QtGameController) a RealJoysticksManager, if not already done a previous time.

```
RealJoysticksManager *rjm;                  // in a private section in Profile.h
rjm = nullptr;                              // in Profile constructor
if (!rjm) {rjm = new RealJoysticksManager{};}   // at the beginning of play()
```

    - b. Check if the real joysticks you want to include in your profile are connected by testing the results of the "joystick". You can send a message in the main window to keep the user aware of what is happenning.

```
// in play
bool btmwj = rjm->joystick("Joystick - HOTAS Warthog");
bool btmwt = rjm->joystick("Throttle - HOTAS Warthog");

if (btmwj) {emit message("Warthog joystick detected !",Qt::black);}
else {emit message("Warthog joystick not detected !",Qt::red);}

if (btmwt) {emit message("Warthog throttle detected !",Qt::black);}
else {emit message("Warthog throttle not detected !",Qt::red);}

if (!btmwj || !btmwt) {return false;}
```

    - c. Create one EnhancedJoystick object per joystick you want to include in your profile and "register" them so that they will monitored at each time step. The names are

exactly than the ones seen by the system. You can go to the game controller panel to have the exact name to use.

```
// in a private section of Profile.h
EnhancedJoystick *tmwj; // for Thrustmaster Warthog Joystick
EnhancedJoystick *tmwt; // for Thrustmaster Warthog Throttle

// in play
tmwj = new EnhancedJoystick(rjm->joystick("Joystick - HOTAS Warthog"),false);
tmwt = new EnhancedJoystick(rjm->joystick("Throttle - HOTAS Warthog"),false);
this->registerRealJoystick(tmwj);
this->registerRealJoystick(tmwt);
```

- 2. Setup the virtual joysticks:
    - a. Create the virtual joysticks. If the configuration fails, the constructor of VirtualJoystick throws an exception that is caught by the MainWindow.

      ```
      vj1 = new VirtualJoystick(1,128);
      ```

    - b. Do the connection necessary for the virtual joysticks to send messages to the main window. I recommend you to send a message to the main window using "emit message" to inform the configuration is complete. At last, register the virtual joystick so that at each time step the virtual joystick object check if it is necessary (or not) to send an HID report. Otherwise the virtual joystick will not be refreshed.

      ```
      QObject::connect(vj1,&VirtualJoystick::message,this,&Profile::message);
      emit message("Virtual joystick 1 configured",Qt::black);
      this->registerVirtualJoystick(vj1);
      ```

    - c. The steps a and b must be both done for virtual joystick 1 before doing it for the second one, and so on...

- 3. Register shift buttons for layers. For details about layers see §6.2.

  ```
  this->registerLayerDim1(Layers::In, tmwt, TMWT::MSD);
  this->registerLayerDim2(Layers::Up, tmwt, TMWT::BSF);
  this->registerLayerDim2(Layers::Down, tmwt, TMWT::BSB);
  ```

- 4. It is advised to initialize the virtual joysticks buttons and axes taking into account the position of real joysticks buttons and axes so that they are in sync at the start. Otherwise maybe a movement will be needed to get them in sync. Here is a simple example:

  ```
  vj1->resetReport();
  vj1->setButton(VJOY::DX1,tmwj->buttonPressed(TMWJ::TG1));
  vj1->setAxisValue(VJOY::Y,tmwt->axisValue(TMWT::THR_LEFT));
  vj1->sendReport();
  ```

- 5. Create the initial mappings using the functions Map, MapButton, MapAxis, MapAxis1, MapAxis2 or MapTempo that are explained further.

function stop

The only thing to do in the stop function is to delete the real joysticks (EnhancedJoystick object) and the virtual joystick (you may not for this second one, but in this case you must know what you are doing).

```
// in stop
if (tmwj) {delete tmwj; tmwj = nullptr;}
if (tmwt) {delete tmwt; tmwt = nullptr;}
if (vj1)  {delete vj1;  vj1  = nullptr;}
return true;
```

5.3 Profile compilation

Programmers may not need advices on this point, especially if you use an IDE and that you integrated Qt into it. For others, if you use Qt 5 with a mingw compiler (only slight adjustments should be needed for other compilers) you can:

- use the compilation widget of the main window (§4)
- compile your profile with a bat file (examples are provided)

5.4 Hiding the real joysticks

If you run a profile, run your game and go to its control menu to map the controls, you should have noticed that the system and the game still see the real joysticks. It can generate duplicate inputs that are annoying when trying to define the controls. It is possible to remedy to it using:

- "devorder" from Bryan Kendall
- "ViGEm" from Benjamin Höglinger a.k.a. nefarius

I suggest you to use the first one that is really simple to use. The necessary files are already provided in the directory "devorder". To prevent a program / game to see one or several of your joysticks, you have to:

- copy the file "dinput8.dll" next to the game executable. Use the x64 version if the game executable is 64 bits compatible, otherwise use the x86 version
- copy the "devorder.ini" file next to it to and edit it by placing the names of the joysticks you want to hide just after "[hidden]"

6 How it works

This part 0 is essential for a full comprehension of the tool but it may be difficult to understand at this stage... and you may want to go quickly to description of the features. In this case you can skip to §7 to §11 and go back here later.

6.1 What is under the hood

UJPS relies on what we call "mappings". A mapping is a sort of connection between a real joystick event and a virtual joystick event. A profile includes a whole set of mappings and each mapping is tested at each time step. The different pre-defined mappings are explained in §9.

At each time step, the runOneLoop function of AbstractProfile is run. It performs exactly the following steps:

- A. Read the events coming from every registered real joystick
- B. Compute the current layer given the shift buttons positions (see §6.2)
- C. For each real joystick event, look which mappings it triggers (only among the mappings that can be triggered by events), and for each triggered mapping, perform the associated action (only post events in a queue)
- D. Look for all triggered mappings (only among the mappings that can be triggered by "states", as contrary to "events") and for each triggered mapping, perform the associated action (only post events in a queue)
- E. Perform all the keyboard events (available only for Windows for now), and call setAxis and setButton functions of VirtualJoystick
- F. Call sendReport function of each registered virtual Joystick to send the new buttons and axes positions to the vJoy driver
- G. Process pending events. If a callback, from an ActionCallback (see §8.1.5), is called at step C or D, and if the callback contains calls to some Map or Unmap functions, the mappings are not deleted or added during C or D step, but now

Additional custom mappings, triggers and actions classes can be added to add new features to the software. To do so you have to sub-class respectively AbstractMapping, AbstractTrigger or AbstractAction. Before doing so, take a look at the existing one to understand how it works.

6.2 Layers and LayersCombos

6.2.1 LayersCombos as mapping filters

Each mapping is given only for a set of layers, called a "layers combo". It means that if the current layer is included in the layers combo of the mapping, the mapping can be triggered, otherwise it is not possible. The layer combo of the mapping acts as a filter.

The definition of a layer is the same of in Thrustmaster TARGET, but the way we use it is a bit different. In TARGET, when you map a button (for example), you have to specify the action for each layer and puts 0 for the layers where you want to do nothing. The following example should be clearer than a long speech:

In TARGET: MapKeyIOUMD(&Joystick, TG1, DX1, DX1, DX1, DX2, DX2, 0);

In UJPS:

```
MapButton(tmwj, TMWJ::TG1, {"i"}, vj1, VJOY::DX1);
MapButton(tmwj, TMWJ::TG1, {"od|om"}, vj1, VJOY::DX2);
```

6.2.2 Definition of a layer

The ones that are not used to Thrustmaster concept of layers should not have understand anything of the previous lines, this is perfectly normal. Now is time for more explanations. The layers in UJPS is implemented just as in TARGET and is not customizable profile by profile in the current version V1.0.0.

Layers can be seen as a way to give different functions to a same button / axis. It can be seen different ways: as shifting (as the shift or control keys of a keyboard) or as different "modes" (e.g. air/air or air/ground modes for military aircraft simulations). The implementation of the layers is 2-dimensional: one dimension with 2 possibilities "In" and "Out", another with 3 possibilities "Down", "Middle" and "Up". Don't focus on the names they are not really meaningful!

Layers		Dim 1	
		Out	In
Dim 2	Down	OD	ID
	Middle	**OM**	IM
	Up	OU	IU

Figure 4: built-in fixed layers as in Thrustmaster TARGET

The default layer is Out / Middle. It is generally used as the layer when none of the "shift buttons" are pressed.

6.2.3 Registering shift buttons

To register a shift button, you have specify the dimension, the sub-layer, a pointer the real joystick and the number of the button. Usually you do so in the play function of the profile (Profile.cpp):

```
this->registerLayerDim1(Layers::In, tmwt, TMWT::MSD);
this->registerLayerDim2(Layers::Down, tmwt, TMWT::BSB);
this->registerLayerDim2(Layers::Up, tmwt, TMWT::BSF);
```

It is also perfectly possible to define buttons for Out and Middle sub-layers, but to understand what can be the interest of it, look at the next part.

6.2.4 How is computed the current layer

The current layer is computed using the registered shift buttons, their associated sub-layer and the current state of these buttons. The exact algorithm, in pseudo-code, is the following:

```
// initialize with default
subLayer1 = Out;
subLayer2 = Middle;

for each d1 in dim1ShiftButtons // one item per call of registerLayerDim1
        if d1.button.isPressed
                subLayer1 = d1.subLayer1; break;

for each d2 in dim2ShiftButtons // one item per call of registerLayerDim2
        if d2.button.isPressed
                subLayer2 = d2.subLayer2; break;

currentLayer = (subLayer1,subLayer2);
```

So the priority is given to the shift button registered first. And if none of the registered buttons is pressed, the default sub-layer is used. If you understand exactly how the algorithm works, you can try to play more with "registerLayerDim1 and 2".

6.2.5 Constructing LayersCombo

To specify a LayersCombo to a mapping, you have to construct one. Simple first: if you want that your mapping is activated for all layers, you can just provide "AllLayers". It is a global variable that covers all layers. To do so you can also provide a default-constructed LayersCombo. The 3 next lines are equivalent:

```
MapButton(tmwj, TMWJ::TG1, AllLayers, vj1, VJOY::DX1);
MapButton (tmwj, TMWJ::TG1, LayersCombo{}, vj1, VJOY::DX1);
MapButton (tmwj, TMWJ::TG1, {}, vj1, VJOY::DX1);
```

If you don't want to specify all layers in your LayersCombo, you have to initialize it with a string. The accepted characters are "i", "o", "d", "m", "u", "|" and you can spaces wherever you want. Please refer to table 1 when reading the following lines:

- "i" means ID, IM and IU (the "In" column)
- "o" means OD, OM and OU (the "Out" column)
- "io" means AllLayers
- "u" means OU and IU (the "Up" line)
- "m" means OM and IM (the "Middle" line)
- "d" means OD and ID (the "Down" line)
- "md" means OD, ID, OM and IM (the "Middle" and "Down" lines)
- "umd" means AllLayers
- ...

And you can also combined several ones with a "|":

- "od | im" means OD and IM
- "od | id" is equivalent to "d"

- "md" is equivalent to "m|d"
- "im" means IM and is not equivalent to "i|m" that means ID, IM, IU and OM

6.3 Real joysticks classes

The base class for real joystick is "AbstractRealJoystick". "RealJoystick" is the standard class for real joysticks and contains a pointer to a QGameController object from the QtGameController module. "EnhancedJoystick" is a decorator class and add several features to an existing real joystick:

- axes curves and trims (see §11.1)
- buttons and axes locking (see §11.2)

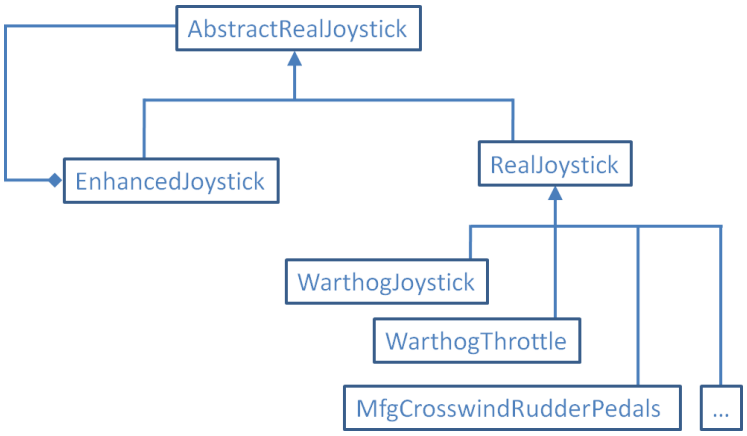

Figure 5: real joysticks classes

The classes concerning specific joysticks (WarthogJoystick, ...) are here for 3 reasons:

- QtGameController has the drawback not to correctly handle POV. Discrete POVs are interpreted as 2 axes that can only take the values -1, 0 and 1, and there is no way to know in advance which axes are real axes and what axes are in fact a POV. Consequently I decided to begin simple and not to support POV in the RealJoystick class. Then in order to get around this issue, I implemented classes for specific joysticks: for a given joystick there is no problem to know what axis is a POV or not... but this is not generalizable. It is just a temporary work-around. For example, the WarthogJoystick class implement the POV as 4 buttons named H1U, H1R, H1D and H1L (whose number are 20 to 23) just as defined by Thrustmaster in TARGET GUI and scripts. The same is done for the Warthog throttle.

  To resolve the issue, modifications in the QtGameController module are needed (see §12). In the meantime, you can also define your own specific classes to resolve the issue and send them to me to share them with everyone in the next version.

  If you can't do it, you can also use the provided program "Monitoring.exe" available with UJPS to see what axes are POV. Then you can send this data to me with also the names you would want for each axis and buttons. For example, in this below figure axes 0 and 1 are the POV, axis 2 is JOYY and axis 3 is JOYX. Buttons 1 to 19 are unchanged. Buttons 20 to 23 are the POV.

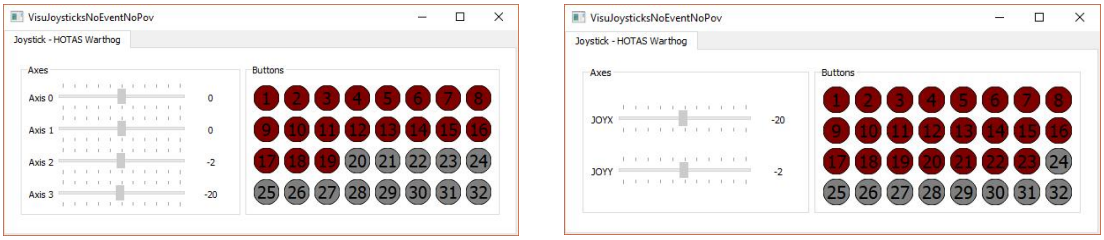

Figure 6: Warthog Joystick without (left) and with (right) the specific class

- Defining additional buttons even if we are not dealing with a POV. The Warthog throttle does a wide usage of this. Thrustmaster calls that "virtual buttons"... virtual buttons of real joysticks, we are not dealing with virtual joysticks here. Let's take the autopilot switch of the Warthog throttle as an example (it has 3 stable positions). The 2 up and down positions are buttons 27 (named APPAT for "autopilot path") and 28 (named APALT for "autopilot alt"), but the middle position is not mapped to any button, if used as a raw joystick. But with Thrustmaster TARGET or the WarthogThrottle class of MyTarget, the middle position is assigned to the button 52 (named APAH for "autopilot alt/hdg").

- Defining names for buttons and axes numbers. The following examples are self-speaking:
  - "static const uint THR_LEFT = 0;" in WarthogThrottle.h to define the axis number for the left thrust lever axis
  - "static const uint H4R = 15;" in WarthogJoystick.h to define the button number of the right button of hat 4

7 Very basic axis and button mapping

MapButton and MapAxis

To perform the simplest bindings, i.e. reproduce the state / value of a button / axis from a real joystick to a virtual one, you can simply call the "MapButton" and "MapAxis" functions:

```
MapButton(tmwj, TMWJ::H2L, AllLayers, vj1, VJOY::DX32);
MapAxis(tmwj, TMWJ::JOYX, AllLayers, vj1, VJOY::X, AxisDirection::Reversed);
```

Where tmwj is a pointer to the EnhancedJoystick embedding the Warthog joystick and vj1 a pointer to the virtual joystick 1. The first line links the H2L button of the Warthog joystick with the button 32 of the virtual joystick 1. And the second one links the JOYX axis to the X axis but reversed. Not to reverse, just specify "Normal" instead of "Reversed".

Be aware that if your profile only uses these 2 functions, there is strictly no interest to use MyTarget... or any software for joystick programming.

Unmapping

You can also delete bindings by calling the following functions (from class AbstractProfile):

```
UnmapAll();
UnmapButton(tmwj,TMWJ::TG1);
UnmapAxis(tmwj,TMWJ::JOYX);
```

8 Pre-defined actions

All action classes inherit from the AbstractAction class.

8.1.1 Actions about buttons

ActionButtonSetChange

The action ActionButtonSetChange reacts only to changes (on real joysticks) and uses the new real joystick button state to reproduce it on the button of the virtual joystick. It is useful to do a "simple button binding". To construct one you have to specify a pointer to the virtual joystick and the number of the button.

```
ActionButtonSetChange(VirtualJoystick*,uint)
```

To perform a "simple button binding", you can use one of the 2 following lines, they are exactly equivalent:

```
MapButton (tmwj, TMWJ::H2L, AllLayers, vj1, VJOY::DX128);
Map(tmwj, ControlType::Button, TMWJ::H2L, AllLayers,
        new TriggerButtonChange{},
        new ActionButtonSetChange{vj1,VJOY::DX128}
);
```

Any of these 2 instructions links directly the button H2L of the Warthog Joystick (pointed by tmwj in this example) to the button 128 of the first virtual joystick (pointed by vj1 in this example) for all layers.

ActionButtonSetChangeOpposite

The action ActionButtonSetChangeOpposite do the same but the value of the button of the virtual joystick will be the opposite of the value of the real joystick button.

ActionButtonToggle

The action ActionButtonToggle is constructed using the same arguments but when activated this action inverts a virtual joystick button. This action is not meant to be activated so often (i.e. not at each time step for example) as otherwise the virtual joystick button will be toggled at each time step.

ActionButtonPulse

The action ActionButtonPulse is constructed using an additional argument, the number of cycles of the pulse. You can easily transform a time interval in milliseconds into a number of cycles using the ms2cycles function. This way, whatever the time step is (it is the time step specified in the main window) the pulse will last the same duration.

```
Map(tmwj, ControlType::Button, TMWJ::H2L, AllLayers,
        new TriggerButtonPress{},
        new ActionButtonPulse{vj1,VJOY::DX128,ms2cycles(500)}
);
```

With the above line, when the H2L button is pressed (passes from up to down position, not is in the down position), a pulse of 500 ms on the 128th button of virtual joystick 1 is performed.

ActionButtonPress and **ActionButtonRelease**

These 2 actions are used to press or release a button of a virtual joystick when activated. With this you can for example press a virtual button when one real button is pressed and release it when another real button is pressed:

```
Map(tmwj, ControlType::Button, TMWJ::H2L, AllLayers,
        new TriggerButtonPress{},
        new ActionButtonPress{vj1,VJOY::DX128}
);
Map(tmwj, ControlType::Button, TMWJ::H2R, AllLayers,
        new TriggerButtonPress{},
        new ActionButtonRelease{vj1,VJOY::DX128 }
);
```

8.1.2 Actions about axes

ActionAxisSetChange

The action ActionAxisSetChange reacts only to changes (on real joysticks) and uses the new real joystick axis state to reproduce it on the axis of the virtual joystick. It is useful to do a "simple axis binding". Curves and trims are defined separately (see §11.1) but with this action you can reverse the direction. To construct one you have to specify a pointer to the virtual joystick, the number of the axis and the direction ("Normal" or "Reversed").

```
ActionAxisSetChange(VirtualJoystick*,uint,AxisDirection)
```

To perform a "simple axis binding", you can use one of the 2 following lines, they are exactly equivalent:

```
MapAxis(tmwj, TMWJ::JOYX, AllLayers, vj1, VJOY::X, Reversed);
Map(tmwj, ControlType::Axis, TMWJ::JOYX, AllLayers,
        new TriggerAxisChange{},
        new ActionAxisSetChange{vj1,VJOY::X,Reversed}
);
```

Any of these 2 instructions links directly the axis JOYX of the Warthog Joystick (pointed by tmwj in this example) to the axis X of the first virtual joystick (pointed by vj1 in this example) for all layers.

ActionAxisIncrement

The action ActionAxisIncrement is useful to create an axis from buttons.

```
Map(tmwj, ControlType::Button, TMWJ::H1L, AllLayers, new TriggerButtonPress{},      new ActionAxisIncrement(vj1,VJOY::Z, 0.05f));
Map(tmwj, ControlType::Button, TMWJ::H1L, AllLayers, new TriggerButtonState{true}, new ActionAxisIncrement(vj1,VJOY::Z, 0.01f));
Map(tmwj, ControlType::Button, TMWJ::H1R, AllLayers, new TriggerButtonPress{},      new ActionAxisIncrement(vj1,VJOY::Z,-0.05f));
Map(tmwj, ControlType::Button, TMWJ::H1R, AllLayers, new TriggerButtonState{true}, new ActionAxisIncrement(vj1,VJOY::Z,-0.01f));
```

With this example, the buttons H1L and H1R are used to control the axis Z of the virtual joystick 1. When H1L is pressed the value of the axis increases of 0.06, then it increases of 0.01 until the Z axis reaches the maximal value of 1 or until the H1L button is released.

8.1.3 Actions about keyboard

As much as possible, you should prefer to use virtual joystick instead of keyboard actions. Indeed, if you have many keyboard actions, you will need to use modifiers, and some issues can arise. For example if you generates LCtrl+A and LShift+B, the game could also see more inputs: LCtrl+B and/or LShift+A.

ActionKeyPress, **ActionKeyRelease** and **ActionKeyPulse**

Keyboard actions are only supported on Windows for now. There are 3 pre-defined keyboard actions available: ActionKeyPress, ActionKeyRelease and ActionKeyPulse. Modifiers can be used. There is no modifier by default. The prototypes of the constructors are the following:

```
ActionKeyPress(uint key, uint modifier = 0)
ActionKeyRelease(uint key, uint modifier = 0)
ActionKeyPulse(uint key, uint modifier = 0, uint cycles = 4)
```

In the file WindowsKeys.h, also available in the Annex (§15.4) you can find all the available keys names.

```
Map(tmwj, ControlType::Button, TMWJ::H3U, AllLayers,
        new TriggerButtonPress{},
        new ActionKeyRelease(Key_Tab,Key_LShift)
);
Map(tmwj, ControlType::Button, TMWJ::H3R, AllLayers,
        new TriggerButtonPress{},
        new ActionKeyPulse(Key_M,0,ms2cycles(60))
);
```

With this example, when the H3U button is pressed, the "Left Shift + Tab" keystroke is release. And when the H3R button is pressed, a pulse of 60 ms on the "M" key is performed.

8.1.4 Actions Chain, Delay and Sequence

The users of Thrustmaster TARGET scripts should recognize these actions. The definition is exactly the same as in Thrustmaster.

ActionChain

A chain action is useful to perform several actions at the same time. The actions are not done at the same only if you use a delay in the chain:

```
Map(tmwj, ControlType::Button, TMWJ::H1U, AllLayers, new TriggerButtonPress{}, new ActionChain({
        new ActionKeyPulse(Key_A,0,ms2cycles(60)),
        new Delay{ms2cycles(150)},
        new ActionKeyPulse(Key_B,0,ms2cycles(60))}));
```

The above code performs a pulse on the "A" key and 150 ms later another one on the "B" key. If the time of the delay is smaller than the time of the first pulse, the "B" key gets pressed before the "A" key gets released.

ActionSequence

An sequence is used to generate different actions each time it is activated. Once all sub-actions have been performed, it starts again with the first one.

```
Map(tmwj, ControlType::Button, TMWJ::H1D, AllLayers, new TriggerButtonPress{}, new ActionSequence{
        new ActionKeyPulse(Key_C,0,ms2cycles(60)),
        new ActionKeyPulse(Key_D,0,ms2cycles(60))});
```

When you press several times the H1D button, it will generate pulses on "C" key the first time, "D" key the second time, "C" key the third time, ...

And you can combine chains with sequences at will...

8.1.5 Executing a custom function

ActionCallback

The ActionCallback is used to execute a custom function when it is activated. It is the equivalent of the "EXEC" feature of Thrustmaster scripts. To construct one, you must provide a std::function object (or something that can be converted to, such as a pointer to a function that takes no argument and has no return, or a compatible lambda).

```
ActionCallback(std::function<void()>)
```

Here is an example of use with both std::bind and a lambda for a Profile member function. Of course the same can be done with non-members functions be it has less interest:

```
void Profile::myMemberFunction()
{
        // just a few random instructions for the example
        DoAction(new ActionButtonPulse(vj1,VJOY::DX13));
        UnmapButton (tmwj, TMWJ::S4);
        MapButton (tmwj, TMWJ::S4, vj1, VJOY::DX1);
        vj1->setAxis(VJOY::Z,0f);
}

// with std::bind
Map(tmwj, ControlType::Button, TMWJ::TG1, AllLayers,
        new TriggerButtonPress{},
        new ActionCallback(std::bind(&Profile::myMemberFunction,this))
);

// with a lambda
Map(tmwj, ControlType::Button, TMWJ::TG1, AllLayers,
        new TriggerButtonPress{},
        new ActionCallback([this](){this->myMemberFunction();})
);
```

When the TG1 button gets pressed the member function "myMemberFunction" is executed. Here is another example with a lambda:

```
auto trimRandomY = [this](){tmwj->setAxisTrim(TMWJ::JOYY,((float)rand()/(float)RAND_MAX)*2.0f-1.0f);};
Map(tmwj, ControlType::Button, TMWJ::TG2, AllLayers, new TriggerButtonPress{}, new ActionCallback(trimRandomY));
```

In the above example, when the TG2 button gets pressed, the axis JOYY of the Warthog joystick is trimmed with a random value.

9 Pre-defined mappings

9.1 MappingStandard

A "standard mapping" uses one trigger and one action. When the current layer is in the mapping's layers combo and that the "trigger is triggered", the mapping performs its action. This is a very versatile mapping that can cover most of our needs, as they are already many different pre-defined triggers and actions and as you can easily create new ones. To use standard mapping, you have to use the "Map" function of the AbstractProfile class:

```
void Map(AbstractRealJoystick *rj, ControlType type, uint rnum, LayersCombo, AbstractTrigger*, AbstractAction*);
```

The "ControlType" argument must be "Button" or "Axis" and "rnum" is the number of the button or axis.

9.2 MappingTempo

But we cannot do everything with a standard mapping. The "tempo" mapping can be only linked to real buttons, not axes. The principle is that if the button is pressed less than a given Δt the action1 is performed, otherwise once the button has been pressed during Δt the action2 is performed. To create one and append it to existing mappings, you have to use the "MapButtonTempo" function of AbstractProfile class:

```
void MapButtonTempo(AbstractRealJoystick *rj, uint rButton, LayersCombo lc, uint cycles, AbstractAction *act1, AbstractAction *act2);
```

9.3 MappingAxis1

The "MappingAxis1" is equivalent to the "AXMAP1" of Thrustmaster TARGET. It is used to map one real axis to 2 virtual buttons. The axis is splitted in a given number of areas and:

- each time the axis changes of area for the one above, actionPlus is performed
- each time the axis changes of area for the one below, actionMoins is performed

To define the areas you can provide:

- a number: the areas will be distributed uniformly
- breakpoints defining the limits between the areas: N points means N+1 areas

```
MapAxis1(tmwj, TMWJ::JOYY, AllLayers, 5, action1, action2);
MapAxis1(tmwj, TMWJ::JOYY, AllLayers, {-0.9,-0.3,0.3,0.9}, action1, action2);
```

The following figure shows what is obtained when you ask for 5 points. You obtain the same thing if you provide the values {-0.8, -0.4, 0, 0.4, 0.8}.

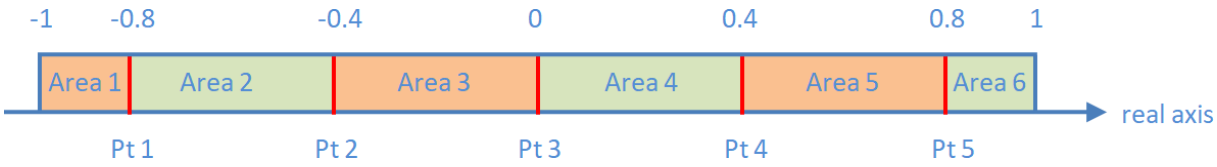

Figure 7: definition of areas

9.4 MappingAxis2

The "MappingAxis2" uses the same splitting into areas than the "MappingAxis1" but there is no just 2 actions function of the direction of the movement, but one action per area: when the axis enters this area, the action of the area is performed. As for "MappingAxis1" you can provide the number of points or a list of breakpoints. If the number of points is different than the number of actions plus 1, an exception is thrown. It will be caught by the main window and everything that should have been done after is skipped.

```
MapAxis2(tmwj, TMWJ::JOYY, AllLayers, {-0.9,-0.3,0.3,0.9},{
        new ActionKeyPulse(Key_Numpad1,0,ms2cycles(30)),
        new ActionKeyPulse(Key_Numpad2,0,ms2cycles(30)),
        new ActionKeyPulse(Key_Numpad3,0,ms2cycles(30)),
        new ActionKeyPulse(Key_Numpad4,0,ms2cycles(30)),
        new ActionKeyPulse(Key_Numpad5,0,ms2cycles(30)),
});
```

9.5 MappingMergeAxes

This mapping merge 2 axes into one, using weightings. In the following example, when at least one of the 2 brake pedals of the MFG crosswind moved, the Y axis of the first virtual joystick is set with the difference between the axes values of the 2 brake pedals (BRK_LEFT and BRK_RIGHT). A 0.5 is applied because the output must be in the range -1;1.

```
MapMergeAxes(mfgx, MFGX::BRK_LEFT, 0.5f, mfgx, MFGX::BRK_RIGHT, -0.5f, AllLayers, vj1, VJOY::Y);
```

10 Pre-defined triggers for "standard mapping"

All trigger classes inherit from AbstractTrigger. The triggers does not know what button or axis they are watching, only the mapping has that information and give the trigger that info when calling the isTriggered function of the triggers. That is why we don't specify the real joystick and the axis / button number when creating a trigger.

10.1 Triggers about axes

TriggerAxisChange

Such objects are activated every time a physical axis moves. A trigger of this type is created when the "simple axis mapping" function "MapAxis" is called. The constructor takes no argument.

TriggerAxisChangeComparison

Such objects are activated every time a physical axis moves and that the comparison test is passed. The constructor is the following:

```
TriggerAxisChangeComparison(bool bGreaterThan, float threshold)
```

If "bGreaterThan" is true the comparison test is "> threshold", otherwise it is "< threshold".

10.2 Triggers about buttons

TriggerButtonChange

Such objects are triggered when the button is pressed or released. The constructor takes no argument. It reacts only

to changes.

TriggerButtonPress and **TriggerButtonRelease**

Such objects are triggered when the button is released. The constructor takes no argument. It reacts only to changes.

TriggerButtonState

Such objects are triggered when the button is down (if true is provided to the constructor) or up (if false is provided to the constructor). The condition is evaluated at each time step. The constructor takes one argument, a Boolean.

11 Other features

11.1 Axis trim and curves

The provided utility program "AxesCurves.exe" allows you to tune your curves and trims.

The function EnhancedJoystick::axisValue, which implements the pure virtual function AbstractRealJoystick::axisValue, returns the position of the axis taking the curve and the trim into account. If you need to access to the raw position of the axis, you can use the function EnhancedJoystick::rawAxisValue.

Axis trimming

The curve is applied to the EnhancedJoystick objects. Their axes can also be trimmed but the virtual joysticks axes too. To better understand how it works, the final virtual joystick axis value is:

$$v2 = k * f(v1 + trim1) + trim2$$

Where:

- v1 is the raw (no curve no trim) position of the real axis
- trim1 is the trim of the real axis
- trim2 the trim of the virtual axis
- f the curve
- k is 1 or -1 function of the axes mapping direction (1 if Normal, -1 if Reversed).

So real axis trimming and virtual axis trimming are not the same thing. If you visualize the curve in a plot with the raw value v1 in abscissa and v2 in ordinate:

- real axis trimming moves the curve left/right
- virtual axis trimming moves the curve up/down

Real axis trimming is for example useful to compensate for a stick non perfectly centered. The function takes 3 arguments:

```
tmwj->setAxisTrim(TMWJ::JOYX,0.01f,AbsoluteOrRelative::Absolute); // set the trim at 0.01
vj1->setAxisTrim(VJOY::X,-0.2f, AbsoluteOrRelative::Relative); // subtract 0.2 to the trim
```

For example, to compensate for Warthog joystick centering inaccuracy you can use the 2 following code lines in a callback (see §8.1.5). This way, the current position of the joystick becomes the new zero position:

```
tmwj->setAxisTrim(TMWJ::JOYX,-tmwj->axisRawValue(TMWJ::JOYX));
tmwj->setAxisTrim(TMWJ::JOYY,-tmwj->axisRawValue(TMWJ::JOYY));
```

Curves

The curves are applied to the real joystick objects axes (whose class is EnhancedJoystick). You can add your own curves by sub-classing "AbstractCurve". To set a curve to the axis, you just have to:

```
tmwj->setCurve(TMWJ::JOYX,pointerToTheCurve);
```

And you can remove the curve by doing one of the 2 following:

```
tmwj->removeCurve();
tmwj->setCurve(nullptr);
```

If you set a curve, the previous one (if any) will be deleted.

Pre-defined curves

3 curves are pre-defined. They are the same as in Thrustmaster TARGET:

- Custom curves defined by breakpoints
- J curves with one "zoom" parameter (float type)
- S curves with parameters (all are float types):
  - lower, center and upper dead zone
  - "curve" parameter
  - zoom

You can use the provided program "AxesCurves.exe" to tune the parameters of your curves. You can also add new curves to the program (C++ and Qt coding).

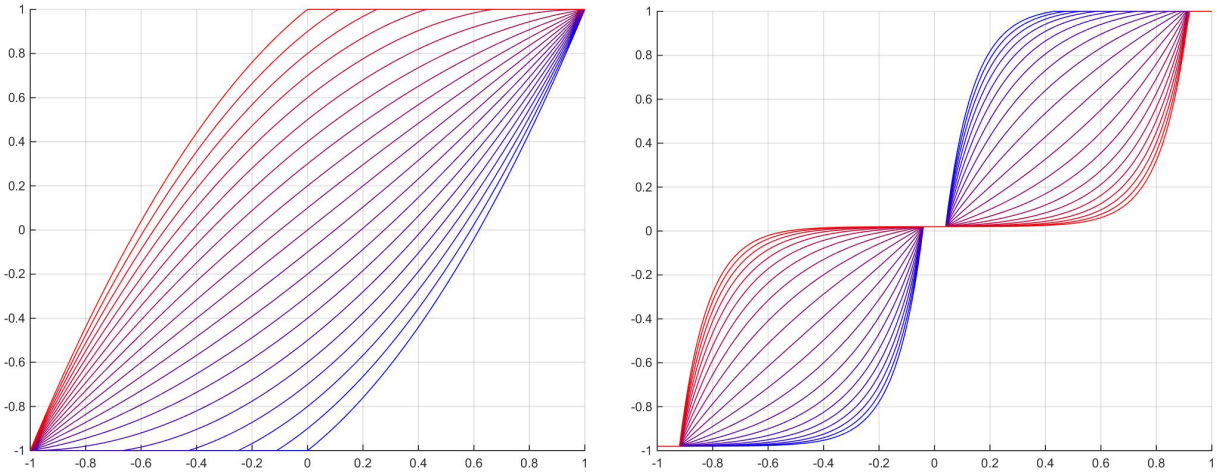

Figure 8: J curves with different zoom (left) and S curves with different "curve" (right)

For practical reasons, 3 functions are defined to set custom curves. They just create the new curve of the correct type with the given parameters and call setCurve:

```
tmwj->setSCurve(TMWJ::JOYX, lowerDZ, centerDZ, upperDZ, curve, zoom);
tmwj->setJCurve(TMWJ::JOYX, zoom);
tmwj->setCustomCurve({-1,-1,  0,0.2,  1,1});
```

Also for practical reasons, to create the custom curve, you have to specify a std::vector of floats. If the number of values is odd, an exception is thrown (it will be caught by the main window that will display an error message and of course no curve will be applied).

The "AxesCurves" additional program

This program allows you to finely tune the parameters of your curves using a GUI. Using the current position of you joystick (the purple point) and the "real joystick trim", you can also re-center your controllers that need it. Only the J and S Thrustmaster curves are implemented for now.

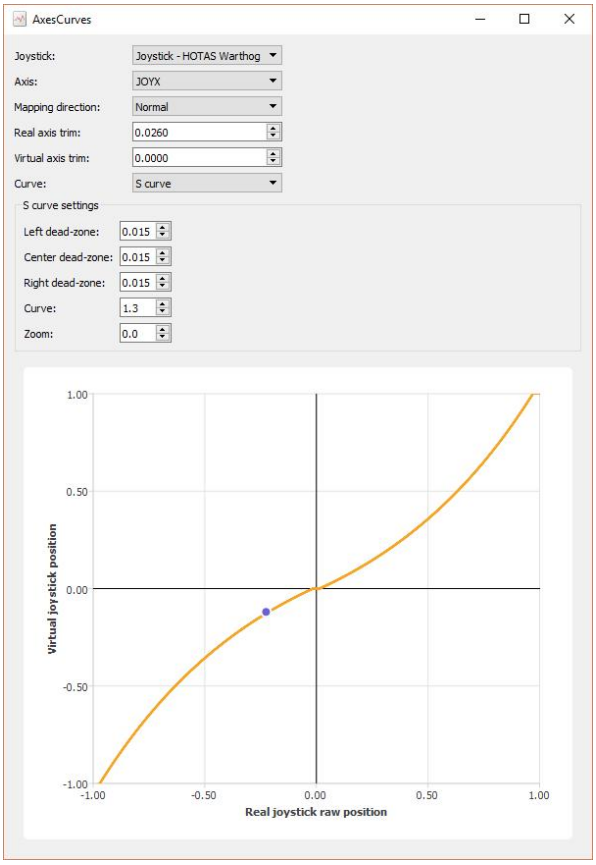

Figure 9: AxesCurves main window

11.2 Button and axis locking

Both real and virtual joysticks can have their buttons and axes locked. In many cases it will not make any difference to lock the real or virtual button / axis, but in other cases it can be really useful. So not to limit the possibilities of the software, both are possible.

When you lock a real joystick button or axis, the software will not see any modification of the button state or the axis value even if the real one has changed and if you call the function "buttonPressed" or "axisValue", you will get the last value before locking.

```
tmwj->setButtonLocked(TMWJ::TG1,true);
tmwj->setAxisLocked(TMWJ::JOYX,true);
```

If you lock a button or an axis of a virtual joystick, it will be frozen. Once unlock, if nothing is sent to this button / axis, its state remains the same.

```
vj1->setButtonLocked(VJOY::DX1,true);
vj1->setAxisLocked(VJOY::X,true);
```

11.3 DoAction

The function "DoAction" of the class AbstractProfile is useful to perform an action, from a callback from example. It is quite equivalent to the "ActKey" function of Thrustmaster TARGET but really more powerful as it can perform any action. But for some actions, there is no need to use DoAction. Here is an example:

```
DoAction(new ActionButtonPress{vj1, VJOY::DX1});
vj1->setButton(VJOY::DX1, true);
```

By default the action is deleted after performed. If you don't want to delete it, provide false as a second argument.

Using a chain action and a delay, it can replace the "DeferCall" function of Thrustmaster TARGET. In the following example, the function "myCallback" will be executed 1 second after this call to DoAction.

```
DoAction(new ActionChain{
        Delay{ms2cycles(1000)},
        ActionCallback{&myCallback}
});
```

12 Current limitations and possible improvements

- Tool only works on Windows, mostly because vJoy is only available on Windows. The only other blocking parts are related to keyboard events generation and deployment:
  - Keyboard keystrokes that I coded only for Windows. The concerned files are:
    - WindowsKeys.h
    - VirtualEventsQueue.h and .cpp
  - The deployment bat files

- Add new settings in the settings menu to ease the use of the main program.

- Add new features, as e.g. a new mapping "split axis" to split one real axis into 2 virtual axes... but don't forget that you can define your own ones!

- QtGameController does not support POV very well. For now the design is to replace the POV by additional buttons (additional compared to the buttons seen by the system)... but only in custom sub-classes, not in the RealJoystick class that just ignore the POV. To change that in the next version, some changes will be necessary in QtGameController or QtGameController must be replaced by another library.

- For now the creation of new profiles is done manually. It would be possible to automate it more in future versions.

- In the current version, the possible layers are fixed (2 dimensions with I/O and U/M/D). Even if I see no need to add give the choice for more dimensions, it may be possible to need more than 3 possibilities for a given sub-layer... even if too much complexity on this ground should harm ergonomics of the profile.

- Provide LED support for Thrustmaster Warthog throttle.

13 Changes notes

13.1 V1.1.0

- renaming of the tool for "UJPS C++"
- addition of the "AxesCurves" program to tune axes curves
- change of icons to use only free icons
- S curve computation correction
- addition of the "unload profile" feature in the main program
- few minor code changes

13.2 V1.2.0

- addition of "devorder" release files (tool from Brian Kendall to hide real joysticks)
- addition of a compilation tool in the main program (only if Qt MinGw compiler installed)
- now you have to provide the path to the .pro file when opening a profile, instead of to the dll file. This way you can open a profile before it has been compiled for the first time
- addition of a settings menu in the main program
- specifying the path to vJoyConfig.exe in the main program (instead of in profiles)
- solved a bug where the profile did not want to load again after unloading
- addition of the "Merge Axes" mapping
- a few minor code changes

14 Quick programming reference for users

This part lists only the functions that may be useful for simple users or for users that would want to have a quick look at some classes before creating their own mapping / trigger / action. So this list is not exhaustive. So before creating new classes, the best is to look directly in the code of UJPS.

14.1 AbstractProfile

- void play(): §5.2
- void stop(): §5.2

- uint ms2cycles(uint ms): convert milliseconds to cycles. Output is the ceiling of the number of milliseconds divided by the time step

- void MapButton(AbstractRealJoystick*, uint, LayersCombo, VirtualJoystick*, uint): §7
- void MapButtonTempo(AbstractRealJoystick *rj, uint rButton, LayersCombo lc, uint cycles, AbstractAction *action1, AbstractAction *action2): §9.2

- void MapAxis(AbstractRealJoystick*, uint, LayersCombo, VirtualJoystick*, uint, AxisDirection = Normal): §7
- void MapAxis1(AbstractRealJoystick*, uint, LayersCombo, const std::vector<float>& points, AbstractAction *action1, AbstractAction *action2) and void MapAxis1(AbstractRealJoystick*, uint, LayersCombo, uint nbPoints, AbstractAction *action1, AbstractAction *action2): §9.3
- void MapAxis2(AbstractRealJoystick*, uint, LayersCombo, const std::vector<float> &points, const std::vector<AbstractAction*>) and void MapAxis2(AbstractRealJoystick*, uint, LayersCombo, uint nbPoints, const std::vector<AbstractAction*>): §9.4

- void UnmapAll(): delete all mappings (§7)
- void UnmapButton(AbstractRealJoystick*, uint): delete all mappings concerning this button (§7)
- void UnmapAxis(AbstractRealJoystick*, uint): delete all mappings concerning this axis (§7)

- void DoAction(AbstractAction *action, bool bDeleteWhenDone = true): §11.3

- void registerRealJoystick(AbstractRealJoystick*): §5.2
- void registerVirtualJoystick(VirtualJoystick*): §5.2
- void registerLayerDim1(Layers::LayerDim1 layer1, AbstractRealJoystick *rj = nullptr, uint rButton = 0): §6.2.3
- void registerLayerDim2(Layers::LayerDim2 layer2, AbstractRealJoystick *rj = nullptr, uint rButton = 0): §6.2.3

14.2 AbstractMapping

Pure virtual functions:

- bool reactsToChanges() const
- bool reactsToStates() const
- bool isTriggered()
- bool isTriggered(const JoystickChange &)
- bool isMappingButton(AbstractRealJoystick *rj, uint rButton) const
- bool isMappingAxis(AbstractRealJoystick *rj, uint rAxis) const
- void performAction()
- void performAction(const JoystickChange &)
- void activateByLayerChange()
- void deactivateByLayerChange()

Other functions:

- void setLayer(Layer)

14.3 AbstractTrigger

Pure virtual functions:

- bool reactsToChanges() const
- bool reactsToStates() const
- bool isTriggered(AbstractRealJoystick *rj, ControlType t, uint rnum) const
- bool isTriggered(AbstractRealJoystick *rj, ControlType t, uint rnum, const JoystickChange &) const

14.4 AbstractAction

Pure virtual functions:

- QVector<VirtualEvent> generateEvents()
- QVector<VirtualEvent> generateEvents(const JoystickChange &)

- uint additionalDelay() const
- QVector<VirtualEvent> activateByLayerChange(AbstractRealJoystick *rj, ControlType t, uint rnum)
- QVector<VirtualEvent> deactivateByLayerChange()

14.5 VirtualJoystick

- bool setButton(uint button, bool bPressed)
- bool toggleButton(uint button)
- bool getButton(uint button) const
- bool setAxis(uint axis, float value)
- float getAxis(uint axis) const

- void setButtonLocked(uint button, bool locked): §11.2
- void setAxisLocked(uint axis, bool locked): §11.2
- void setAxisTrim(uint axis, LONG trim, AbsoluteOrRelative aor = Absolute): §11.2

- void resetReport()
- bool sendReport(bool bEvenIfNoChange = false)

- bool enableVJoyIfNot()
- void disableVJoy()
- bool isVJoyDeviceFree(uint numDevice)
- static void setVJoyConfigExePath(const QString &dirPath, const QString &fileName)

- signal: void message(const QString &str, QColor color)

- axes numbers: X, Y, Z, ROTX, ROTY, ROTZ, SLIDER0 and SLIDER1
- buttons numbers: DX1, DX2, ... to DX128

14.6 AbstractRealJoystick

- uint id()
- QString description()
- QString hardwareId(): the result is valid only for Windows for now

- uint buttonCount() const
- bool buttonPressed(uint button) const
- QString buttonName(uint button) const
- QStringList buttonsNames() const

- uint axisCount() const
- float axisValue(uint axis) const
- QString axisName(uint axis) const
- QStringList axesNames() const

14.7 EnhancedJoystick

Only additional functions compared to AbstractRealJoystick:

- void setButtonLocked(uint button, bool locked): §11.2
- void setAxisLocked(uint axis, bool locked): §11.2
- void setAxisTrim(uint axis, float trim, AbsoluteOrRelative aor = Absolute): §11.1
- float axisRawValue(uint axis) const: §11.1

- void setSCurve(uint axis, float lowerDZ, float centerDZ, float upperDZ, float curve, float zoom): §11.1
- void setJCurve(uint axis, float zoom): §11.1
- void setCustomCurve(uint axis, const std::vector<float> &points): §11.1
- void setCurve(uint axis, AbstractAxisCurve *curve): §11.1
- void removeCurve(uint axis): §11.1

15 Annexes

15.1 List of pre-defined triggers

- TriggerAxisChange
- TriggerAxisChangeComparison
- TriggerButtonChange
- TriggerButtonPress
- TriggerButtonRelease
- TriggerButtonState

15.2 List of pre-defined actions

- ActionAxisIncrement
- ActionAxisSetChange
- ActionButtonPulse
- ActionButtonPress
- ActionButtonRelease
- ActionButtonSetChange
- ActionButtonSetChangeOpposite
- ActionButtonToggle
- ActionCallback
- ActionChain
- ActionSequence
- ActionKeyPress
- ActionKeyRelease
- ActionKeyPulse
- Delay

15.3 List of pre-defined mappings

- MappingStandard
- MappingTempo
- MappingAxis1
- MappingAxis2
- MappingMergeAxes

15.4 Keyboards keys names (Windows)

This list has been established using the values available here:

https://msdn.microsoft.com/fr-fr/library/windows/desktop/dd375731(v=vs.85).aspx

```
namespace Keys
{
        // modifiers
        const uint Key_LShift = 0xA0;
        const uint Key_RShift = 0xA1;
        const uint Key_LCtrl  = 0xA2;
        const uint Key_RCtrl  = 0xA3;
        const uint Key_LAlt         = 0xA4;
        const uint Key_RAlt         = 0xA5;

        // arrows
        const uint Key_ArrowLeft  = 0x25;
        const uint Key_ArrowUp            = 0x26;
        const uint Key_ArrowRight = 0x27;
        const uint Key_ArrowDown  = 0x28;

        // special keys
        const uint Key_Backspace = 0x08;
        const uint Key_Tab      = 0x09;
        const uint Key_Return    = 0x0D;
        const uint Key_CapsLock  = 0x14;
        const uint Key_Escape    = 0x1B;
        const uint Key_Space     = 0x20;
        const uint Key_PageUp    = 0x21;
        const uint Key_PageDown  = 0x22;
        const uint Key_Insert    = 0x2D;
        const uint Key_Delete    = 0x2E;

        // digits (the ones that are above letters)
        const uint Key_0 = 0x30;
        const uint Key_1 = 0x31;
        const uint Key_2 = 0x32;
        const uint Key_3 = 0x33;
        const uint Key_4 = 0x34;
        const uint Key_5 = 0x35;
```

```
const uint Key_6 = 0x36;
const uint Key_7 = 0x37;
const uint Key_8 = 0x38;
const uint Key_9 = 0x39;

// letters
const uint Key_A = 0x41;
const uint Key_B = 0x42;
const uint Key_C = 0x43;
const uint Key_D = 0x44;
const uint Key_E = 0x45;
const uint Key_F = 0x46;
const uint Key_G = 0x47;
const uint Key_H = 0x48;
const uint Key_I = 0x49;
const uint Key_J = 0x4A;
const uint Key_K = 0x4B;
const uint Key_L = 0x4C;
const uint Key_M = 0x4D;
const uint Key_N = 0x4E;
const uint Key_O = 0x4F;
const uint Key_P = 0x50;
const uint Key_Q = 0x51;
const uint Key_R = 0x52;
const uint Key_S = 0x53;
const uint Key_T = 0x54;
const uint Key_U = 0x55;
const uint Key_V = 0x56;
const uint Key_W = 0x57;
const uint Key_X = 0x58;
const uint Key_Y = 0x59;
const uint Key_Z = 0x5A;

// numpad
const uint Key_Numpad0 = 0x60;
const uint Key_Numpad1 = 0x61;
const uint Key_Numpad2 = 0x62;
const uint Key_Numpad3 = 0x63;
const uint Key_Numpad4 = 0x64;
const uint Key_Numpad5 = 0x65;
```

```
        const uint Key_Numpad6 = 0x66;
        const uint Key_Numpad7 = 0x67;
        const uint Key_Numpad8 = 0x68;
        const uint Key_Numpad9 = 0x69;
        const uint Key_NumpadMultiply  = 0x6A;
        const uint Key_NumpadAdd       = 0x6B;
        const uint Key_NumpadSubstract = 0x6D;
        const uint Key_NumpadDecimal   = 0x6E;
        const uint Key_NumpadDivide    = 0x6F;
        const uint Key_NumpadNumlock   = 0x90;

        // F keys
        const uint Key_F1  = 0x70;
        const uint Key_F2  = 0x71;
        const uint Key_F3  = 0x72;
        const uint Key_F4  = 0x73;
        const uint Key_F5  = 0x74;
        const uint Key_F6  = 0x75;
        const uint Key_F7  = 0x76;
        const uint Key_F8  = 0x77;
        const uint Key_F9  = 0x78;
        const uint Key_F10 = 0x79;
        const uint Key_F11 = 0x7A;
        const uint Key_F12 = 0x7B;
        const uint Key_F13 = 0x7C;
        const uint Key_F14 = 0x7D;
        const uint Key_F15 = 0x7E;
        const uint Key_F16 = 0x7F;
        const uint Key_F17 = 0x80;
        const uint Key_F18 = 0x81;
        const uint Key_F19 = 0x82;
        const uint Key_F20 = 0x83;
        const uint Key_F21 = 0x84;
        const uint Key_F22 = 0x85;
        const uint Key_F23 = 0x86;
        const uint Key_F24 = 0x87;
}
```