Tab 1

Playful Plugins Dev Guide
by. Furimanejo

This is a guide on developing plugins using PP Script, the event detection library from Playful Plugins. The library source code can be found here

Summary Of Detection Methods

Computer Vision

Consists of analysing a live screen capture of the user's screen to detect or measure relevant visual elements, like icons, health bars or text. A good approach for games with anti-cheating software because it does not require modifying game files or touching the game's process in any way

Process Memory Reading

consists of analysing the in-ram memory of a computer process to acquire relevant data. Not to be used in games with anti-cheating software. Also not recommended to be used in games that are frequently updated because changes in the game's code very often require redesigning the detection system

HTTP

Consist of sending or receiving HTTP requests to acquire relevant data, for example, getting game events from League of Legends through its live client data API

Creating A Plugin

A plugin is a package (a folder or a .zip file) that contains at least a metadata.yaml file and a python script file. The package may also contain other files or folders if required by specific detection methods (a “templates” folder with image files is a common case).

I highly recommend having a plugin package by your side while reading this doc so you can observe how things look like in an actual plugin. Example: Peggle Plugin

OBS: names marked with this color represent important fields, files or API functions. They are case-sensitive, using the wrong name will break things

Metadata

A YAML file, named `metadata.yaml`, that contains:

- `name` (str): The name of the plugin
- `author` (str, optional): The name of plugin’s author
- `version` (str, optional): The version number of the plugin
- `script` (str): The path of the python script with the plugin’s logic, relative to the package folder. Ex: “plugin.py”
- `req_lib_version`: (int, optional) The minimum library version required to run the plugin. Defaults to 0, a version from before requirement checks existed, in PP v1.99.2

Python Script

A python script (.py file), referenced in the metadata.yaml file, that contains the plugin's logic. The 2 main parts of a plugin script are:

- a call to the init function
- the definition of the update function

The init function should be called once on the global scope of the plugin script taking as parameter a dictionary with relevant data to initialize the plugin. For example, on a computer vision plugin this dictionary would have a cv key with a dictionary value defining regions, templates, etc. Fields required by specific detection methods will be documented on their respective sections. Common fields on an init call are:

- description (str, optional): A description of the plugin, shows up in the plugin's tab. Ex: "This is a plugin for the game Powerwash Simulator"
- config_file_name (str, optional): The name of the file, without the extension, the plugin's configs will be saved to. Ex: "powerwash_sim"

- target_window (str, optional): A regex to match the title of the plugin's target window and get its rect and focus. Ex: r"^PowerWash Simulator$"

- events (dict, optional): A dictionary defining what events the plugin can raise. Keys are the event names (str). Values are dictionaries with the fields:
  - description (str, optional):
  - scale_amount (function, optional): WIP

The update method will be called internally once per detection frame. This is where we use the detection methods and raise events

If editing on an IDE, like VSCode, you can “import” the stub file on the first line of the script file to enable autocompletion for the PP methods (“import” in quotes cause the import don’t actually happen in the usual python sense, the line is ignored in runtime and the methods are injected into the script’s globals post compilation)

Debug Folder

The execution of plugins might create a debug folder to save relevant files (for development or debug purposes), it can be found at App Data Folder > Debug > Name Of The Plugin

Executing a plugin deletes everything on its respective debug folder, be careful to not lose data that way.

Raising Events

To raise an event call the method `raise_event` from inside the update function passing as argument a dictionary with the following fields:

- `type` (str, optional): The name of the event you are raising, as defined in the `init` data. Most output modules will ignore events without a type, the exception being the overlay module that will still show a region and label of an event without type, good for debug purposes
- `id` (str, optional): Giving an event an id makes it override or extend previous events with that same id, good for "ongoing" events. Events with different ids stack. Defaults to a random UUID4
- `amount` (int or float, optional): Defaults to 1. Some events have an amount associated with them that can be used to scale the effect of the event, like the amplitude of patterns. Example:
- `region` (str, optional):
- `label` (str, optional):

Computer Vision

Computer vision is a set of functions to acquire information from images or videos. In the case of Playful Plugins, the video is a live capture of an application’s window or one of the user’s monitors

To utilize computer vision methods you need to add the `cv` field to the `init` data, `cv` being a dictionary with the following fields:

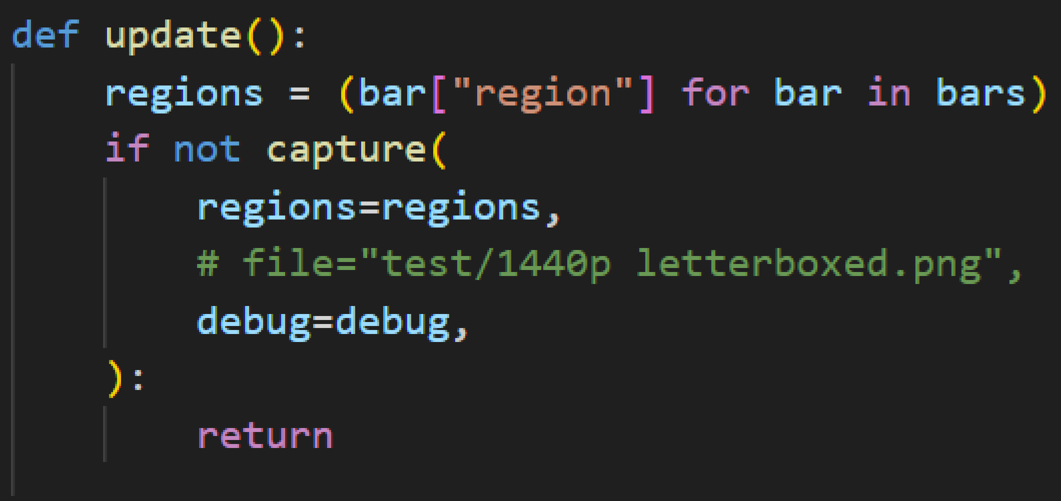
- `regions` (dict): A dictionary that defines the regions of interest. Keys are the names of the regions (str). Values are dictionaries with the following fields:
    - `rect` (dict): The position and size of the region within the capture area. Require x and y values, size can be given with w and h (width and height), or with l and b (left and bottom). Example: {"x": 163, "y": 41, "w": 600, "h": 11}
    - `label_position` (str, optional): Used by the overlay to position a label on the region. Can be “top”, “left”, “bottom”, or “right”. When None the label is positioned centered inside the region
- `templates` (dict, optional): A dictionary of templates to be used on template matching operations. Keys are the names of the templates (str), values are dicts with:
    - `file` (str): The path to the image file
    - `threshold` (float): Value between 0.0 and 1.0
- `scaling_method` (function or tuple, optional): A method that defines how regions and templates should be scaled based on the capture’s dimensions. It takes the arguments (x, y, w, h, rw, rh), where x,y,w,h are the rect of the region or template, and rw,rh are the width and height of the detected capture (usually the resolution of a window)...

Capture

Before utilizing any of the CV detection methods you must call the capture method to acquire an image to operate on:

- regions (tuple[str]): A tuple of the names of the regions you want to capture. If groups of regions are too far apart (for example: some regions on the bottom left and some regions on the top right) it might be more performant to execute 2 different captures, one for each group, than a single big capture that bounds all regions
- file (str, optional): The path of a file you want to load instead of capturing a window or monitor (for development or debug purposes). Must be inside the plugin package or folder
- debug (bool): A flag that causes the captured image to be saved to a debug folder when set to True. See Debug Folder

The capture method is not guaranteed to work (for example, if the target app was not found) so you must also check the return of the capture method before calling any detection methods

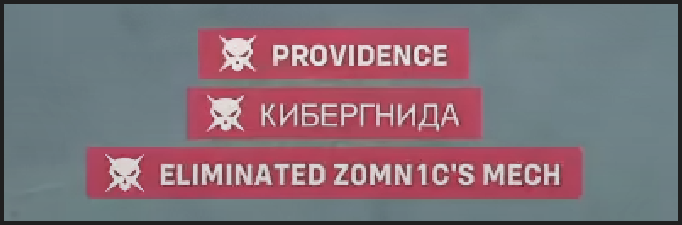


```python
def update():
    regions = (bar["region"] for bar in bars)
    if not capture(
        regions=regions,
        # file="test/1440p letterboxed.png",
        debug=debug,
    ):
        return
```

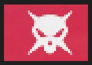
Template Matching

Template matching is a technique that consists of “scanning” a target region of an image comparing it to a template image. See this for more info on the theory behind the technique. Useful to detect events with clear on-screen indicators, for example, getting eliminations in Overwatch 2

Elimination popups in Overwatch 2:



Elimination template:



The main result of this operation is a number, between 0 and 1, that represents the best match between the template and the region, 1 being a perfect match. That result is then compared to a threshold value associated with the template, if the result is equal or bigger than the threshold the match is said to be positive, that is, the template image was found within the region. In real applications of this technique the results are rarely equal to 1 (a perfect match), so it’s the developers job to find a good enough threshold value to avoid false positive and or false negative detections

It’s also good to note that this operation might be performance intensive if the region is too big compared to the template

To perform a template matching operation call the function `match_template` with the following arguments:

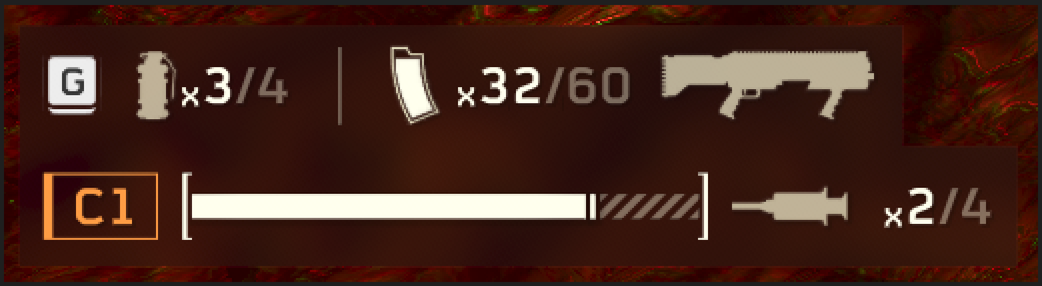
- `template` (str): The name of the template
- `region` (str): The name of the region

- filter (function, optional): An image processing function to be applied to the template and the region before the template matching happens
- div (tuple, optional): A tuple of 4 float values between 0 and 1 that allows the operation to happen in a subdivision of the specified region, that represent, in percentages, the starting X, ending X, starting Y and end Y values. Defaults to (0,1,0,1) which equates to the whole region
- debug (bool, optional): Cause images relevant to the operation, like the template and region, to be saved to the respective debug folder

Region Fill Ratio

Consists in counting how many pixels within a region fall within a specified range of colors. Requires a color filter that results in a binary image. Useful to, for example, measure how full a health bar is

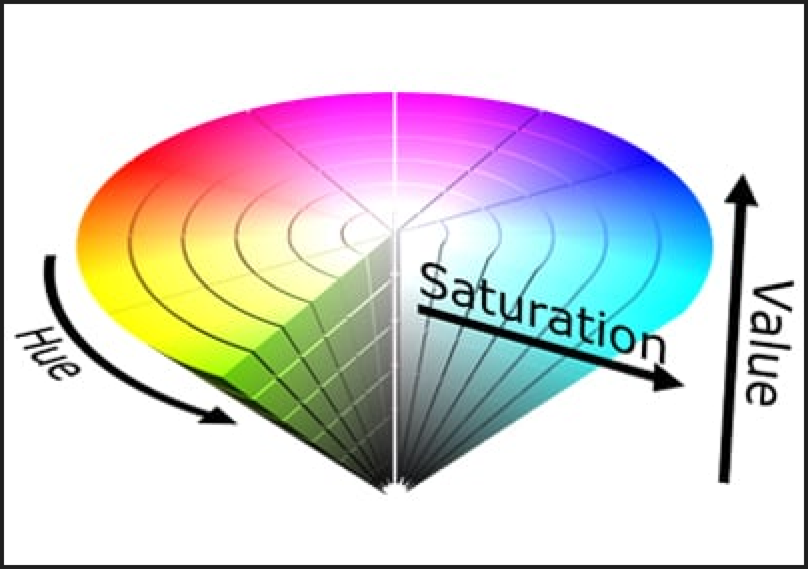
Health bar UI in Helldivers 2:



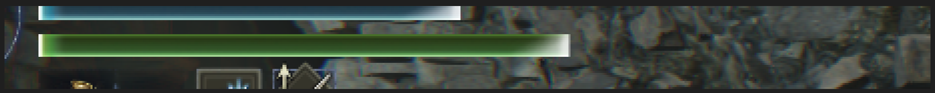
Filters

An image is composed of pixels, each pixel containing 1 or more values representing its color. The most common encoding for that data is the RGB color space, but depending on the application other color spaces might be useful, for example, grayscale or HSV.

In the HSV color space colors are represented by 3 values: Hue, Saturation and Value.



This is useful, for example, to filter colors of the same Hue, regardless of their saturation. As an example we can look at filtering the blue color from the following bar (stamina bar in Elden Ring Nightreign):



Here we call the get_region_fill_ratio method with the following filter function and debug = True, then we separate the channels of the resulting file to obtain the individual Hue, Saturation and Value channels:

```python
def green_filter(img):
    img = cv_to_hsv(img)
    return img
```

Then, with the color picker tool and focusing on the relevant part of the region we want to filter for, we find that the Value ranges from 50 to 255, Saturation ranges from 0 to 180 and finally, the most important channel, Hue ranges from 50 to 100, representing the green hue of the bar. By passing the hsv image and these values to the in_range function we can now filter for the bar inside the region. Tip: when creating a filter be sure to give the ranges a little wiggle room to account for changes in lightning and what not, for example, if analyzing a channel you find its lowest value to be 58, consider round it down to 50 when creating the filter.

```python
def green_filter(img):
    img = cv_to_hsv(img)
    return cv_in_range(
        img,
        (50, 0, 50),
        (100, 180, 255),
    )
```

Process Memory Reading

To read a process' memory we first need the name of the process, for example, VampireSurvivors.exe, then we need to find the relevant static pointers that point to the variables we want to read

A static pointer consists of a module name, like GameAssembly.dll, a list of offsets, like [0x03F88358, 0x138, 0x40, 0x58, 0x68] and also the type of variable we expect to read from that pointer (bool/int/float)

Finding these values can be done using a program called Cheat Engine. Here's an example of what the process looks like. You might also find static pointers in "cheat tables" posted online

DISCLAIMER: Playful Plugins only reads values (contrary to cheat programs that would change values to give players advantages). That said merely touching a game's process memory can trigger hacking accusations or detections, even if we're not changing values, so never use this approach with games that have anti-cheat software

HTTP Requests

Plugins can detect events through HTTP requests, by sending GET requests to a server or receiving POST requests as a server, all in the localhost address. To do so add the field `http` to the `init` data, a dictionary with the following fields:

- `port` (int): The port on the localhost through which the requests will happen
- `handle_content` (function, optional): Reference to a function that receives 1 argument (content). When this field is present the module will open a server, listen to incoming POST requests and pass the content of the requests to the handle_content function as they happen. An usual way to use this function is to store the received content in a global variable to be used later in the update function