Processor file

We want to reduce the impact of blenderVR on the blender file (`.blend`). For instance, all the interactions issued from the plugins (VRPN, OSC ...) don't have to be defined inside the `.blend` file, since they do not exists outside blenderVR development frame. Moreover, elements to synchronize interaction from master to slaves cannot be defined inside `.blend` file.

blenderVR thus introduces the notion of processor file. It is a Python file associated with the `.blend` that contains all the interactions required to use the `.blend` file within blenderVR. By default (and you should not change it) this file is in the same folder than the `.blend` file and its name is the name of the blender file minus `.blend`, but post-fixed by `.processor.py`. For instance, the processor file of `simple.blend` is `simple.processor.py`.

Minimal processor file

The minimal processor file contains:

minimal.processor.py

```
import blendervr

if blendervr.is_virtual_environment():
    class Processor(blendervr.processor.getProcessor()):
        def __init__(self, parent):
            super(Processor, self).__init__(parent)
else: # not VR screen => Console
    class Processor(blendervr.processor.getProcessor()):
        def __init__(self, console):
            super(Processor, self).__init__(console)
```

Debugging processor through log messages

As you have probably seen in Debug window per screen, the output of blenderplayer is not displayed by default in the console during blenderVR runs.

Thus, you cannot use basic `print` python commands to help you while debugging.

You should instead use the blenderVR standard logger usable inside any blenderVR object (due to inheritance):

```
self.logger.debug(blah blah ...)
```

blah blah ... is whatever you want, comma separated, as long as there is a “stringification” method (`__str__`) for each element. The `logger` object inherits from python `loggin` module. Thus, you can

replace debug by `info`, `warning`, `error`, `critical`. Depending on the log window level selection (see the screen window of the Run tab of the console), you will see your message or not.

You can also use `self.logger.log_traceback(False)` to display the traceback of your program. `True` in parenthesis means an error, then blenderVR will stop running in “Virtual Environment”. This traceback is available inside as well as outside an exception.

There is also `self.logger.log_position()` that simply displays the position of the calling method in debug level.

Keyboard and mouse

You can get access to keyboard and mouse information of the master node by defining the `keyboardAndMouse` method. The `info` provided has the same format than any provided through the VRPN plugin.

You can use a logger to see what is contained inside the `info` argument. You can also have a look at the `simple.processor.py` file inside `sample` folder to get an example of how to use this method.

Choose objects to synchronize

By default, blenderVR doesn't synchronize scene objects (blacklisting for efficiency issues). You must specify the elements you want to synchronize by explicitly flagging the objects to synchronize by the master node:

```
# synchronizer.objects.getItem(enable, recursive = True)
# synchronizer.objects.item_base.Base.activate(enable, recursive = True)
if self.blenderVR.isMaster():
   self.blenderVR.getSceneSynchronizer().getItem(bge.logic).activate(True,
True)
```

This method will synchronize (first `True` as `activate` parameter) all elements recursively (second `True` as `activate` parameter) from the `bge.logic` (that is the root of the `.blend` file). In other words, it will activate all the objects of the scene. You can also synchronize only a few objects by applying this call to each item (the objects as parameter of `getItem`).

Processor inheritance

We commonly use the same interactions on different scenes. For instance, the Head Control Navigation system is useful on most scenes. blenderVR allows the developer to have a “generic” processor that all other processors will be able to use by inheritance. You can add an intermediate processor by adding a line at the beginning of your processor:

```
blendervr.processor.appendProcessor(os.path.join(blenderVR_root, 'samples',
'processors.py'))
```

This line specifically adds the `processors.py` (from folder `samples` of blenderVR) processor to each processor in the sample folder.

This processor proposes:

- Inside the “Virtual Environment”:
  - **head control navigation system** to navigate through the world just with your head as joystick (see mountain sample),
  - **laser** interaction, useful when you want to select objects from your scene (see chess sample) ;
  - **viewpoint manipulation** in the same way than blender uses in its graphic window (see simple sample and press 'v' to use it) ;
  - **'q' to quit** clean quit the “Virtual Environment”.
- Inside the console:
  - **user interface** that can include buttons for Head control navigation system.

We suggest you to have a look at the processor files inside the sample folder before you write your owns.

Master-slaves communication

Inside the processors, you can send data from the master to the slaves.

There is no solution to send data from any slave to the master nor any other slave !

There is two mechanisms to send data to the slaves: stream and one-shot.

Stream: processor as synchronized object

You can register your processor as a synchronized object. As such, at each frame, the synchronizer will ask the master's processor (through `getSynchronizerBuffer()` method) the buffer to send to the slaves. Then, if the buffer is not empty (getSynchronizerBuffer() doesn't return `None`), each slave, in the same frame, will receive it through its `processSynchronizerBuffer()` method.

To register your processor, you must call from the constructor of your “virtual environment” processor:

```
self.blenderVR.addObjectToSynchronize(self, 'main processor')
```

The argument in single quote is the name of the processor used by the synchronizer to disambiguate between all synchronized objects. You can use anything else than `main processor`, but this is a good default choice.

As an example, you can have a look at the `simple.processor.py` in sample folder, where `try_use_stream_between_master_and_slave` is set to `True`.

One-shot: specifically send a data

When you don't need to send data through a stream (ie.: each frame), you can send one data sometime with `sendToSlaves`/`receivedFromMaster` methods. The first argument is a string describing your data whereas the second argument is the data.

Beware that this processing use encapsulation and JSON to encode and decode the data. That is heavier than the stream mechanisms and must be applied to data with a low update rate only.

As an example, you can have a look at the `simple.processor.py` in sample folder, when you press 's' (see method `keyboardAndMouse`) on the master.

Run() method

The `run` method will be called at each frame on the master node. Thus, if you need to process something (register a data, update a value, etc.), you can add whatever you want here. To process something on the slaves, you should unlock it with previous mechanisms to send data from the master to the slaves.

Console-"virtual environment" communication

You can send data from the master “virtual environment” node to the console (`sendToConsole`/`receivedFromVirtualEnvironment`). You can also do the opposite, from console to “virtual environment” (`sendToVirtualEnvironment`/`receivedFromConsole`).

As usual, the `simple.processor.py` file shows the use of this mechanism. If you set `try_wait_user_name` to `True`, then the “virtual environment” is paused. To unlock it, you must type a name in the processor window from the console and you click on `Set user name`. Then, the name will be sent to the master node that will display it and answer the console.