MORSE

Modular OpenRobots Simulation Engine

Developers Guide

v.0.1.1

blender LAAS-CNRS ONERA THE FRENCH AEROSPACE LAB STAE TOULOUSE

v.0.1.1

The Modular OpenRobots Simulation Engine - Developers guide

LAAS-CNRS - ONERA

July 28, 2010

v.0.1.1

Contents

Uncomment these line to display the "generate PDF[1] " button that allow to fetch the TEX source

Figure 1: Logo MORSE

[1] Portable Document Format

Chapter 1

Developers documentation

1.1 Get involved!

You can write to the list of developers of the project, writing to this address: mailto:morse-dev@laas.fr.
MORSE developement can be tracked online with CGIT: MORSE on CGIT
You can as well fetch the latest version of the code with GIT:

```
$ git clone http://trac.laas.fr/git/robots/morse.git
```

Easy installation instructions will be added after the official release of MORSE 0.1.

1.2 Overview

An overview of code organization in MORSE
Overview for MORSE organization

1.3 Workflow

Principle of interaction with Blender and the game engine
Overview for MORSE workflow for developers

1.4 Adding a new component

Create a sensor. Explain the logic
Tutorial: Adding a new component

1.5 With an example of a modifier

EZ if in python, a bit more complex if we call an external library (cf impact on the dependencies)
Tutorial: Creating a modifier

Chapter 2

A MORSE organization overview

Files hierarchy

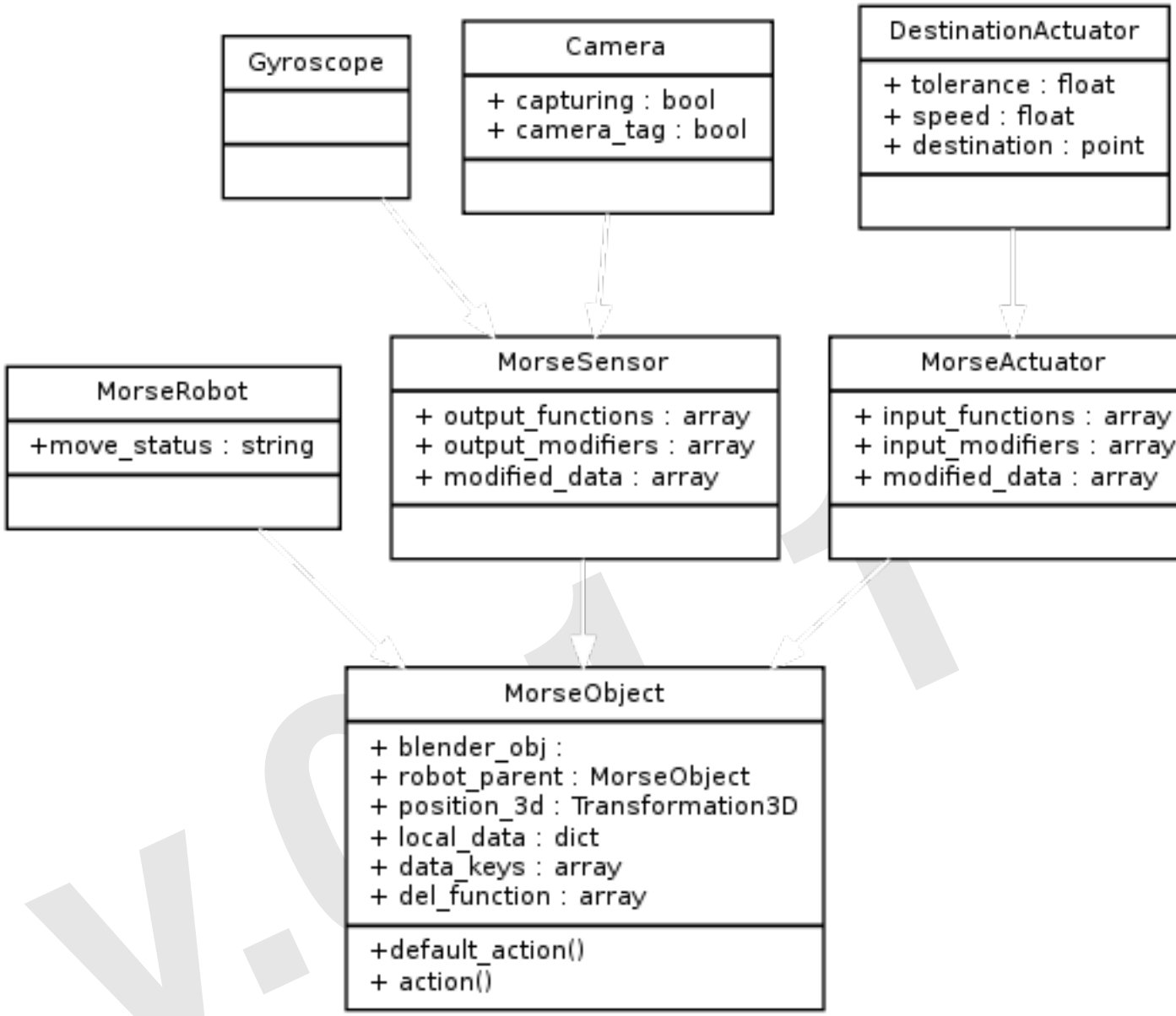
The source code of Morse is organised in the following way :

- bin : it contains the main entry point of the morse simulation
- config : it contains stuff for CMake
- data : it contains the blender model of sensors / actuators
- doc : the documentation (in DokuWiki format)
- examples : it contains examples about :
  - how to control the simulator (in clients)
  - more or less elaborate scenarii (in scenarii)
- src : it contains all the python scripts used by the simulator : it is the core of the simulator
- morse/actuators : it contains implementation for various robot controllers
- morse/blender : it contains some scripts needed at the initialization of the game engine
- morse/helpers : it contains various helpers (math transformation) and base classes
- morse/middleware : it contains the code for linking with different middlewares
  - yarp
  - pocolibs
  - socket
  - text (for logging)
- modifiers : it contains implementation for various modifier to basic components
- robots : it contains instantiation of different robot classes
- sensors : it contains implementation for various robot sensors

Code organization

The following diagram shows the class hierarchy currently used in MORSE.

The main entry point for Blender for each component is the method `action`. Yet, it is not supposed to be overridden by leaf-classes. To modify the behaviour of a component, you need to modify the method `default_action`. The action of `action` depends if it is a sensor or an actuator (robot doesn't do anything by themself).



Behaviour of a sensor When Blender calls the method `action` for a sensor, the following things happen :

1. update of the position of the sensor
2. call `default_action`
3. apply in order each function of `output_modifiers` (modify the content of the sensor)
4. apply in order each function of `output_functions` (output the content of the sensor to different clients)

Behaviour of an actuator When Blender calls the method `action` for an actuator, the following things happen :

1. apply in order each function of `input_functions` (receive input from different clients)
2. apply in order each function of `input_modifiers` (if needed)
3. call `default_action`

Another interesting point is the `local_data` dictionary : it contains a representation of the external state of the sensor (or the actuator) (for example, it will contains the position for a GPS)

Chapter 3

The MORSE workflow for developers

The internal functioning of MORSE is based on Blender's Game Engine events. These are defined in the Logic buttons window, using the graphical interface called "Logic Bricks". These are a set of predefined **Sensor**, **Controller** and **Actuator** events which can be linked together. **Actuators** in particular are important because they are the places where Python scripts are called.

For Blender to find the Python scripts refered to in the Actuators, the Python files must be directly in a directory listed in `PYTHONPATH`. The standard location for these files in MORSE is the directory: `ORS_ROOT/src/morse/blender`, which should be included in `PYTHONPATH` as described in the MORSE installation documentation.

Each object in Blender can have its own set of Logic Bricks. In every MORSE simulatino scene, there must be one `Scene_Script_Holder` object, which holds the predefined Logic Bricks necessary to initialize and control the simulation. When a simulation is started (when launching the Game Engine) it will call the initialization scripts of MORSE, contained in the file `$ORS_ROOT/src/morse/blender/main.py`.

The script `main.py` is charged of multiple tasks:

- Upon lanching the simulation, it will initialize all components:
  - Create a dictionary of robots
- Create a dictionary of components, and the robot their are associated with
- Create the dictionary of modifiers
- Create the dictionary of middlewares
- Link the modifiers and middlewares to components, as specified in the file `component_config.py`
- When the simulation ends, it will destroy the objects created, and call the methods to cleanup ports, files, connections, etc.

The initialization of all components, including middlewares and modifiers, is done by instantiating an object of the Python class specific to every component. This is done dynamically, using the **Class** and **Path** properties that should be present in the Blender file of every component.

3.1 Events during a simulation

Blender's Game Engine is set to work at a predefined `Tickrate`, which is the number of times the Logic Brick events are executed per second. In the default settings, it is equal to 60, so that during one real second there will be 60 "ticks".

While the simulation is running, the Logic Bricks of each component will make regular calls to their `default_action` method. At this point the component will perform its task and update its internal data.

Chapter 4

Adding a new component

Each component is basically represented by two files, which must be created when adding a new sensor or actuator:

- A Blender file: has the representation of the sensor on blender, and associate the script with the blender object
- A Python script: contains the logic of the sensor

4.0.1 The python part

You need to implement a sub-class of `morse.helpers.sensor.MorseSensorClass` (respectively of `morse.helpers.actuator.Mo`
Important things to do :

- in the constructor of the object (`__init__`) :
- initialize each variable you want to expose to the world into `local_data`
- initialize correctly `data_keys` with the set of variable exposed in `local_data` : the order is important here, because this order will be used for the automatic serialization
- initialize correctly `modified_data`, with a copy of `local_data` (the easiest way is to copy-paste the following code)

```
for variable in self.data_keys:
        self.modified_data.append(self.local_data[variable])
```

- override `default_action` : it must contains the logic of our component. Avoid to do some big computation here : the function is called often, and it will slow down the whole processing of the Game Engine

4.0.2 The blender part

1. First, create a nice modelling of your object, and save it in `$MORSE_ROOT/data/morse/components/sensors/`
2. Press **N** to display the properties of the object. Change its name.
3. Press **F4** to enter in the logic mode
   (a) Add the two following properties:
      - **Class** of type string, which contains the name of the associated python class <Sensor>Class
      - **Path** of type string, which contains the path to the associated python script `morse/sensors/<Sensor>`
   (b) You can add more properties if needed for your components. Additional properties can be used to configure the behaviour of the component, and can be integrated into a GUI[1] in future versions of MORSE.

[1] Graphical User Interface

The names specified in the **Path** and **Class** properties must match exactly the location of the Python file and the name of the defined class, respectively.The information in these variables will be used to dinamically load the module and class during initialisation of the simulation.

4.1 Getting data or exporting data

A component is not really useful if it doesn't get any input (for an actuator) or if you can't use the output of a sensor. You can use different middleware to import / export data.

In the simplest case, you can use automatic serialization, which will try to convert the data in `modified_data` array into the appropriate format to send throug the middleware. This works only for the basic data types of integer, float or string. If you want more specific behaviour for other data types, you need to add a method to the middleware provider of your choice (for example, if you want to export a new sensor through YARP, you need to add a method to MorseYarpClass, in `$MORSE_ROOT/src/morse/middleware/yarp_mw.py`). The method must have the following prototype :

```
def your_method(self, component_instance)
```

In this method, you can access / store component information through its array `modified_data`. In case of a sensor, it is not expected that you change the content of the sensor, but only read information in this array.

After that, you need to register your new function into the middleware abstraction. For that, you need to modify the method `register_component`. It is basically a switch case with the different possible functions. This method is called when parsing the configuration file for the scene, so it is the right place to initialize stuff (opening Yarp ports, sockets, files ...)

4.1.1 Middleware specific information

YARP In MorseYarpClass, the different port_name are stored in a dictionary `_component_ports`, indexed by the name of the component (`component.blender_obj.name`). You can retrieve the associated port with the method `getPort(port_name)`

Example:

```
port_name = self._component_ports[component_instance.blender_obj.name]

try:
        yarp_port = self.getPort(port_name)
except KeyError as detail:
        print ("ERROR: Specified port does not exist: ", detail)
        return
```

Pocolibs In MorsePocolibsClass, the different poster_id are stored in a dictionary `_poster_dict`, indexed by the name of the component (`component.blender_obj.name`)

Text In TextOutClass, the different files are stored in a dictionary `_file_list`, indexed by the name of the component (`component.blender_obj.name`)

Chapter 5

Creating a modifier

Creating a modifier is more or less the same than creating a sensor. There are still two parts, the blender part and the python logic. For the blender part, it is the same thing than for a sensor / actuator.

Python part

There is no strict class hierarchy for modifiers, we rely on python duck typing. We only expect than a modifier exposes a method `register_component`, similarly to middleware, with the following prototype :

```
def register_component(self, component_name, component_instance, mod_data)
```

The method is responsible to add the right modifier specified in `mod_data` in the list of modifier of `component_instance` (input_modifiers or output_modifiers).

Modifier functions must have the prototype

```
modifier(self, component_instance)
```

In your modifier function, you must only access to the array `modified_data` of the component.

GPS\textunderscorenoise.py shows a simple example for a modifier.