User manual search (offline, online), forum search

Lua vs Python scripts

The way Lua and Python scripts are handled in CoppeliaSim (next to differences related to the language itself) displays small differences, which are discussed here:

- Prior to CoppeliaSim V4.7, Python scripts required a *#python* header tag as the very first instruction/comment. This is not the case anymore: language is decided at script creation time, or deduced from file extension
- Global variables are inconvenient and not recommended in Python:

Python Lua

```
#python

def sysCall_init():
    self.myVariable = 21

def sysCall_sensing():
    print(self.myVariable)
```

- CoppeliaSim selects the default Python interpreter present on your system. If this does not work, or if another interpreter is desired, then one can edit the *usrset.txt* file entry named *defaultPython*. The location of the *usrset.txt* file can be queried via the settingsPath property. In addition to that, each script can specify a custom Python executable if required, via the *luaExec* directive (see further down)
- Lua code always executes in the same thread as CoppeliaSim and is thus quite fast. Python code on the other hand is launched in a new process, which connects to CoppeliaSim via socket communication. This implies that Python code will usually start and run slower than Lua code. However, when Python is operating in non-stepping mode, and doesn't access CoppeliaSim API, it truely runs parallel to CoppeliaSim.
- Python scripts are handled via Lua wrapper code, which handles the launch and communication of each Python script. Auxiliary settings, Lua code or Lua functions can be passed via the *luaExec* directive, within a Python comment section. That code must come immediately at the beginning of the script:

```
#python

#luaExec wrapper = 'myWrapperCode' -- looks for myWrapperCode.lua in Lua's search path
#luaExec pythonExecutable = 'c:/python38/python.exe' -- runs a specific Python executable
#luaExec additionalPaths = {'c:/path1', 'c:/path2'} -- adds additional Python module search paths
#luaExec additionalIncludePaths = {'c:/Python38', 'c:/path2'} -- adds additional paths to search for the include file
'''luaExec
function myLuaFunction()
    print('hello Paul!')
    sim.callScriptFunction('myPythonFunction', sim.handle_self)
end
'''

def myPythonFunction():
    print('hello Jeanine!')

def sysCall_init():
    sim = require('sim')
    sim.callScriptFunction('myLuaFunction', sim.handle_self)
    print("hello Marc!")
```

- The import directive does not work when importing the primary Python file, and following should be used instead:

Python Lua

```
#python

include myExternalFile

# myExternalFile is the pythonScript name or path (absolute or relative), without quotes nor the ending '.py'
# searched paths include:
# <CoppeliaSim executable path>/
# <CoppeliaSim executable path>/python
# <current scene path>/
# <additional path>/ (see system/usrset.txt and value 'additionalPythonPath')
# additional include paths passed via #luaExec additionalIncludePaths={'c:/Python38'}
```

- Python threaded scripts will always execute their initialization section (if present). The clean-up section (and other system callbacks) however are only executed in specific situations or in an asynchronous manner
- Python threaded scripts will need to call a message pump (via sim.handleExtCalls) in order to have user callback functions to be serviced, depending on the situation:

Python Lua

```
#python

def sysCall_thread():
    xml = '<ui title="Custom UI"> <button text="Click me!" on-click="click_callback"/> </ui>'
    ui = simUI.create(xml)
    while not sim.getThreadExistRequest():
        sim.handleExtCalls()
    simUI.destroy(ui)

def click_callback(ui,button):
    print("button was clicked!")
```

- When some Python code section needs to be executed without interruption, then one can use following construct:

```
#python

def sysCall_thread():
    sim.acquireLock()

    # perform some task that involves CoppeliaSim

    sim.releaseLock()
```

- A few callback functions need to be enabled explicitly, in order to avoid constant slowdowns:

```
#python

#luaExec contactCallback = true    -- enable contact callback
#luaExec dynCallback = true        -- enable dyn callback
#luaExec eventCallback = true      -- enable event callback

def sysCall_contact(inArg):
    print(inArg)
```

```
def sysCall_dyn(inArg):
    print(inArg)

def sysCall_event(inArg):
    print(inArg)
```

- Some rare API functions will only be availabe in Lua, while other only in Python.
- The word *table* in the documentation refers to array-like or map-like items in Lua, while in Python, it refers to either *lists* or *dicts*.
- Finally remember that Python lists have a zero-based numbering, while Lua array-like tables have a 1-based numbering.