The main script

By default, each scene in CoppeliaSim has one main script. It contains the basic code that allows a simulation to run. Without main script, a running simulation won't do anything significant.

The main script contains a collection of callbacks functions that are appropriately called by CoppeliaSim. The default main script is typically segmented in 4 callback functions:

- the initialization function: **sysCall_init**. It is called one time, just at the beginning of a simulation
- the actuation function: **sysCall_actuation**. It is called in each simulation pass. The code is in charge of handling all the actuation functionality of the simulation. One command is of particular interest: sim.handleSimulationScripts, which calls simulation scripts' *sysCall_actuation* callback functions. Without that commands, simulation scripts won't execute, or won't execute their actuation function, and specific model functionality or behavior won't operate as intended.
- the sensing function: **sysCall_sensing**. It is called in each simulation pass. The code is in charge of handling all the sensing functionality of the simulation (proximity sensors, etc.) in a generic way. One command is of particular interest: sim.handleSimulationScripts, which calls simulation scripts' *sysCall_sensing* callback functions. Without that commands, simulation scripts won't execute their sensing function, and specific model functionality or behavior won't operate as intended.
- the restoration function: **sysCall_cleanup**. It is executed one time just before a simulation ends. The code is in charge of restoring object's initial poses, clearing sensor states, etc.

Following is the typical main script, slightly simplified:

Python Lua

```python
#python

def sysCall_init():
    sim = require('sim')
    # Initialization part

def sysCall_actuation():
    # Actuation part:
    sim.handleEmbeddedScripts(sim.syscb_actuation)
    sim.handleAddOnScripts(sim.syscb_actuation)
    sim.handleSandboxScript(sim.syscb_actuation)
    sim.handleDynamics(sim.getSimulationTimeStep())

def sysCall_sensing():
    # Sensing part:
    sim.handleProximitySensor(sim.handle_all_except_explicit)
    sim.handleVisionSensor(sim.handle_all_except_explicit)
    sim.handleEmbeddedScripts(sim.syscb_sensing)
    sim.handleAddOnScripts(sim.syscb_sensing)
    sim.handleSandboxScript(sim.syscb_sensing)

def sysCall_cleanup():
    # Clean-up part:
    sim.handleSimulationScripts(sim.syscb_cleanup)
    sim.resetProximitySensor(sim.handle_all_except_explicit)
    sim.resetVisionSensor(sim.handle_all_except_explicit)

def sysCall_suspend():
    sim.handleEmbeddedScripts(sim.syscb_suspend)
    sim.handleAddOnScripts(sim.syscb_suspend)
    sim.handleSandboxScript(sim.syscb_suspend)

def sysCall_suspended():
    sim.handleEmbeddedScripts(sim.syscb_suspended)
    sim.handleAddOnScripts(sim.syscb_suspended)
    sim.handleSandboxScript(sim.syscb_suspended)

def sysCall_resume():
    sim.handleEmbeddedScripts(sim.syscb_resume)
    sim.handleAddOnScripts(sim.syscb_resume)
    sim.handleSandboxScript(sim.syscb_resume)
```

There are however many more system callback functions the main script can use to react to various events.

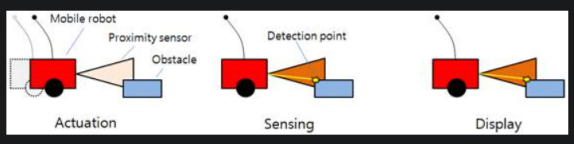
The main script is not supposed to be modified. The reason for this is following: one of CoppeliaSim's strength is that any model (robot, actuator, sensor, etc) can be copied into a scene and is immediately operational. When modifying the main script, you run the risk that models won't perform as expected anymore (e.g. if your main script lacks the command sim.handleSimulationScripts then all models copied into the scene won't operate at all). Another reason is that keeping a default main script allows old scenes to easily adjust for new functionality.

If however, for a reason or another you really need to modify the main script of a scene, you can do this by double-clicking the light-red script icon next to the world icon at the top of the scene hierarchy:

[Main script icon]

Most of simulation script's system callback functions are called from the main script, via the sim.handleSimulationScripts function, which operates in a cascading fashion upon the scene hierarchy and the simulation scripts.

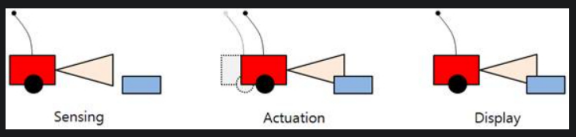
If you look at the default main script, you can notice that the actuation function allows actuating or modifying the scene content (e.g. sim.handleDynamics, etc.), while the sensing function allows sensing and probing the scene content (e.g. sim.handleProximitySensor, etc.). Following illustrates what happens in the default main script when a mobile robot equipped with a proximity sensor is simulated:



[Default actuation - sensing - display sequence]

With above's sequence in mind, simulation scripts will always read (with sim.readProximitySensor) the proximity sensor's state from previous sensing (which happened at the end of previous simulation pass, inside of the main script, with sim.handleProximitySensor), then react to obstacles.

If you need to explicitly handle a sensor, then make sure to always do it while in the sensing section, otherwise you might end-up with situations where the display appears wrong as illustrated in following figure:



[Sensing - actuation - display sequence]

Just as the main script has an actuation-sensing sequence, so do the simulation scripts.