Plugin tutorial

This tutorial describes how to write a plugin for CoppeliaSim. The CoppeliaSim scene file related to this tutorial is located in *scenes/tutorials/BubbleRobExt*. The plugin project files of this tutorial can be found here.

CoppeliaSim plugins are loaded on demand via the loadPlugin function. Most of the time the load procedure is wrapped inside of a simple Lua module tha can additionally provide pure Lua functions, under the same namespace.

The plugin was written for BubbleRob from the BubbleRob tutorial. The BubbleRob plugin adds 4 new script functions:

simBubble.create

Description	Creates an instance of a BubbleRob controller in the plugin.
Lua synopsis	int bubbleRobHandle=simBubble.create(table[2] motorJointHandles,int sensorHandle,table[2] backRelativeVelocities)
Lua parameters	**motorJointHandles**: a table containing the handles of the left and right motor joints of the BubbleRob you wish to control. **sensorHandle**: the handle of the proximity sensor or the BubbleRob you wish to control **backRelativeVelocities**: when BubbleRob detects an obstacle, it will move backwards for some time. relativeBackVelocities[1] is the relative velocity of the left wheel when moving back. relativeBackVelocities[2] is the relative velocity of the right wheel when moving back
Lua return values	result: -1 in case of an error, otherwise the handle of the plugin's BubbleRob controller.
Python synopsis	int bubbleRobHandle=simBubble.create(list motorJointHandles,int sensorHandle,list backRelativeVelocities)

simBubble.destroy

Description	Destroys an instance of a BubbleRob controller previously created with simBubble.create.
Lua synopsis	bool result=simBubble.destroy(int bubbleRobHandle)
Lua parameters	**bubbleRobHandle**: the handle of a BubbleRob instance previously returned from simBubble.create.
Lua return values	result: false in case of an error
Python synopsis	bool result=simBubble.destroy(int bubbleRobHandle)

simBubble.start

Description	Sets a BubbleRob into an automatic movement mode
Lua synopsis	bool result=simBubble.start(int bubbleRobHandle)
Lua parameters	**bubbleRobHandle**: the handle of a BubbleRob instance previously returned from simBubble.create.
Lua return values	**result**: false in case of an error
Python synopsis	bool result=simBubble.start(int bubbleRobHandle)

simBubble.stop

Description	Stops the automatic movement of a BubbleRob
Lua synopsis	bool result=simBubble.stop(int bubbleRobHandle)
Lua parameters	**bubbleRobHandle**: the handle of a BubbleRob instance previously returned from simBubble.create.
Lua return values	result: false in case of an error
Python synopsis	bool result=simBubble.stop(int bubbleRobHandle)

Now open the threaded simulation script attached to the BubbleRob model in the scene and inspect the code:

Python | Lua

```python
#python

def sysCall_thread():
    sim = require('sim')
    simBubble = require('simBubble')
    jointHandles = [sim.getObject('../leftMotor'), sim.getObject('../rightMotor')]
    sensorHandle = sim.getObject('../sensingNose')
    robHandle = simBubble.create(jointHandles, sensorHandle, [0.5, 0.25])
    if robHandle >= 0:
        simBubble.start(robHandle) # start the robot
        local st = sim.getSimulationTime()
        sim.wait(20) # run for 20 seconds
        simBubble.stop(robHandle)
        simBubble.destroy(robHandle)
```

After loading the required module, joint and sensor handles are retrieved and given to the custom script function that creates a controller instance of our BubbleRob in the plugin. If the call was successfull, then we can call *simBubble.start*. The function instructs the plugin to move the BubbleRob model while avoiding obstacles. Run the simulation: BubbleRob moves for 20 seconds then stops, as expected.

Let's have a look at how the plugin registers and handles the above 4 custom Lua functions. Open the simBubble plugin project, and have a look at file *simBubble.cpp*:

Notice the 3 plugin entry points: *simInit*, *simCleanup*, and *simMsg*: *simInit* is called once when the plugin is loaded (initialization), *simCleanup* is called once when the plugin is unloaded (clean-up), and *simMsg* is called on a regular basis with several type of messages.

During the initialization phase, the plugin loads the CoppeliaSim library (in order to have access to all CoppeliaSim's API functions), then registers the 4 custom script functions. A custom script function is registered

by specifying a function name (without prefix/namespace), and a callback address.

When a script calls the specified function name, then CoppeliaSim calls the callback address. The most difficult task inside of a callback function is to correctly read the input arguments, and correctly write the output values. This happens via a stack. The user have two options to interact with the stack:

- the easiest is to use two helper classes (*CScriptFunctionData* and *CScriptFunctionDataItem*, located in *programming/include/simLib*). This however only allows to exchange simple data types (i.e. Booleans, integers, floating point numbers, strings and arrays of any of those)
- the more flexible way, allowing to exchange any type of data, is to use stack functions (located in *programming/include/simStack*): the plugin simSkeleton illustrates this.

In general, callback routines should execute as fast as possible, and control should then be given back to CoppeliaSim, otherwise the whole simulator will halt.