User manual search (offline, online), forum search
- Threaded and non-threaded script code
- Callback functions
  - Dynamics callback functions
  - Joint callback functions
  - Contact callback function
  - Vision callback functions
  - Trigger callback functions
  - User config callback functions
- Script execution order
- Buffers
- Lua vs Python scripts
- Lua crash course
- Plugins
- CoppeliaSim's library
- Accessing scene objects programmatically
- Explicit and non-explicit calls
- CoppeliaSim API framework
  - Regular API reference
  - Regular API constants
  - Properties
    - Properties reference
- Simulation
  - Simulation dialog
- Tutorials
  - BubbleRob tutorial
  - Building a clean model tutorial
  - Line following BubbleRob tutorial
  - Inverse kinematics tutorial
  - External controller tutorial
  - Plugin tutorial
  - Robot language interpreter integration tutorial
- ROS tutorials
  - ROS tutorial
  - ROS 2 tutorial
- Compiling CoppeliaSim

CoppeliaSim's library

CoppeliaSim offers a wide range of functionality via its graphical user interface. For more customized scenarios where fine-grained control of the simulation loop is required (e.g. reinforcement learning), it also offers the flexibility to be used as a library within custom programs.

Using CoppeliaSim as a library consists in loading the CoppeliaSim's library (regular or headless library):

- Windows: `LoadLibrary("coppeliaSim.dll")` / `LoadLibrary("coppeliaSimHeadless.dll")`
- Linux: `dlopen("libcoppeliaSim.so")` / `dlopen("libcoppeliaSimHeadless.so")`
- macOS: `dlopen("libcoppeliaSim.dylib")` / `dlopen("libcoppeliaSimHeadless.dylib")`

then creating a secondary thread (the simulation thread), and:

- calling `simInitialize` to initialize.
- calling any other API functions, and/or calling `simLoop` until `simGetExitRequest` returns false
- calling `simDeinitialize` before termination.

Finally, in the main thread (the UI thread), calling `simRunGui`. Argument to `simRunGui` can be used to toggle on/off individual part of the interface (use `sim_gui_all` to enable all), or to run in *GUI suppression mode* (`sim_gui_headless`).

When using the headless library, there is no need to create a secondary thread, or call `simRunGui`

C++ client

The default main client application that comes with the installation package is a C++ application: *coppeliaSimClient*, but also available in *<CoppeliaSim folder>/programming/*. It compiles to *coppeliaSim.exe* (Windows), or *coppeliaSim* (macOSX and Linux).

Python client

A complete working example of loading the CoppeliaSim library in Python is given in coppeliaSimClientPython, but also available in *<CoppeliaSim folder>*. From your command line:

```
$ python3 ./coppeliaSim.py --help     (display run options)
$ python3 ./coppeliaSim.py            (run CoppeliaSim with GUI and local coppeliaSim lib)
$ python3 ./coppeliaSim.py -O=0       (run CoppeliaSim with minimalistic GUI and local coppeliaSim lib)
$ python3 ./coppeliaSim.py -h         (run CoppeliaSim with emulated headless mode and local coppeliaSim lib)
$ python3 ./coppeliaSim.py -H         (run CoppeliaSim with true headless mode and local coppeliaSimHeadless lib)
$ python3 ./coppeliaSim.py -L <path/to/CoppeliaSim/library>   (run CoppeliaSim with specific lib)
```

In this section, relevant parts are explained.

Initialization

Module `coppeliasim.lib` handles the load and setup of core functions (`simInitialize`, `simDeinitialize`, etc...) via `ctypes`.

```
import builtins
import threading

# set the path to the coppeliaSim's library:
builtins.coppeliasim_library = "/path/to/libcoppeliaSim.so"

# import the coppeliaSim's library functions:
from coppeliasim.lib import *

# start the sim thread (see in the next section)
if trueHeadless:
    simThreadFunc()
else:
    t = threading.Thread(target=simThreadFunc)
    t.start()
    simRunGui(sim_gui_all) # use sim_gui_headless for headless mode
    t.join()
```

SIM thread loop

The basic implementation of simThreadFunc simply executes the application until quit is requested:

```
def simThreadFunc():
    simInitialize(appDir().encode('utf-8'), 0)
    while not simGetExitRequest():
        simLoop(None, 0)
    simDeinitialize()
```

Another possible scenario would be to manually control the operations used to setup a simulation environment (e.g. `sim.loadScene`) and to execute a simulation, e.g. by manually stepping for a predetermined amount of steps:

```
def simThreadFunc():
    simInitialize(appDir().encode('utf-8'), 0)

    # script bridge, see next section
    import coppeliasim.bridge
    coppeliasim.bridge.load()

    global sim
    sim = coppeliasim.bridge.require('sim')

    sim.loadScene('path/to/scene.ttt')
    simStart()
    for i in range(1000):
        t = sim.getSimulationTime()
        print(f'Simulation time: {t:.2f} [s] (running in stepped mode)')
        simStep()
    simStop()

    simDeinitialize()
```

```
def simStart():
    if sim.getSimulationState() == sim.simulation_stopped:
        sim.startSimulation()

def simStep():
    if sim.getSimulationState() != sim.simulation_stopped:
        t = sim.getSimulationTime()
        while t == sim.getSimulationTime():
            simLoop(None, 0)

def simStop():
    while sim.getSimulationState() != sim.simulation_stopped:
        sim.stopSimulation()
        simLoop(None, 0)
```

coppeliasim.bridge

The `coppeliasim.bridge` module provides seamless access to the regular API used by scripts.

Instead of `moduleName = require('moduleName')`, use `moduleName = coppeliasim.bridge.require('moduleName')`.

All other API functions can be used normally.

Example:

```
import coppeliasim.bridge

# load the bridge component:
coppeliasim.bridge.load()

# fetch API objects:
sim = coppeliasim.bridge.require('sim')

# call some API function:
program_version = sim.getIntProperty(sim.handle_app, 'productVersionNb')
```

Callbacks

Some regular API functions use callbacks, i.e. they take a function parameter, and call it one or more time during the execution of the function or at a later time. Examples are sim.moveToConfig, simIK.findConfigs, simOMPL.setStateValidationCallback, etc...

To use Python functions as callbacks, those need to be exposed as C functions, and called via CoppeliaSim's C function wrapper.

A C callback function is called with an `int` argument being the handle of the stack containing input/output arguments, and has to return an `int`, nonzero in case of success, e.g.:

```
def myCallback(stackHandle):
    # use CoppeliaSim stack API to read input args, e.g.:
    doubleValue = ctypes.c_double()
    if simGetStackDoubleValue(stackHandle, ctypes.byref(doubleValue)) != 1:
        print('error in reading double from stack')
        # return 0 for error:
        return 0
    doubleValue = doubleValue.value
    simPopStackItem(stackHandle, 1)
    # and so on...

    # use CoppeliaSim stack API to write return values
    simPushInt32OntoStack(stackHandle, 42)
    stringRetVal = 'xyz'
    simPushTextOntoStack(stackHandle, stringRetVal)

    # return 1 for success:
    return 1
```

for convenience, a `coppeliasim.stack.callback` decorator can be used, which automatically performs reading input arguments from stack and writing return values to stack for most common types, also handling of exceptions:

```
import coppeliasim.stack

@coppeliasim.stack.callback
def myCallback(arg1, arg2, arg3):
    print('myCallback called with args:', arg1, arg2, arg3)
    ret1, ret2 = 42, 'xyz'
    print('myCallback returning:', ret1, ret2)
    return ret1, ret2
```

In both cases, the callback has to be registered with `simRegCallback`:

```
from ctypes import CFUNCTYPE, c_int

# signature is always int(int), i.e. the undecorated callback:
myCallback_c = CFUNCTYPE(c_int, c_int)(myCallback)

# note: maintain a reference to above variable for the whole lifetime
#       of the application, otherwise it will be garbage collected and
#       it will crash when called by C

# callback with index 0 will be available under the name 'ccallback0':
simRegCallback(0, myCallback_c)
```

The callback can be referenced by the name 'ccallback0' (or 'ccallback1' and so on...) and called by e.g. `sim.callScriptFunction('ccallback0', sim.scripttype_sandbox, 0.25, 400, 'str')` or passed to API functions and plugins that accept (Lua) callbacks.

Refer to coppeliaSimClientPython/coppeliaSim.py for the complete code.