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

ROS tutorial

This tutorial will try to explain in a simple way how you can manage to have CoppeliaSim *ROS enabled*, based on ROS Melodic and Catkin build.

First of all you should make sure that you have gone through the official ROS tutorials, at least the beginner section, and that you have installed the Catkin tools. Then, we assume that you have the latest Ubuntu running, that ROS is installed, and that the workspace folders are set. Here also refer to the official documentation regarding the ROS installation.

The general ROS functionality in CoppeliaSim is supported via the ROS Interface (*libsimROS.so*). The Linux distribution should include that file already compiled in *CoppeliaSim/compiledROSPlugins*, but it first needs to be copied to *CoppeliaSim/*, otherwise it won't be loaded. You might however experience plugin load problems, depending on your system specificities: make sure to always inspect the terminal window of CoppeliaSim for details about plugin load operations. The ROS plugin is loaded with:

Python | Lua

```
#python

simROS = require('simROS')
```

The load operation will only be successful if roscore was running at that time. Also make sure to source the ROS environment prior to running CoppeliaSim.

If the plugin cannot be loaded, then you should recompile it by yourself. It is open source and can be modified as much as needed in order to support a specific feature or to extend its functionality. If specific messages/services/etc. need to be supported, make sure to edit files located in *simROS/meta/*, prior to recompilation. There are 2 packages:

- simROS: this package is the ROS Interface that will be compiled to a ".so" file, and that is used by CoppeliaSim.
- ros_bubble_rob: this is the package of a very simple robot controller that connects to CoppeliaSim via the ROS Interface. This node will be in charge of controlling the red robot in the demo scene *controlTypeExamples/controlledViaRos.ttt*

Above packages should be copied to your *catkin_ws/src* folder. Make sure that ROS is aware of those packages, i.e. that you can switch to above package folders with:

```
$ roscd sim_ros_interface
$ roscd ros_bubble_rob
```

In order to build the packages, navigate to the *catkin_ws* folder and type:

```
$ export COPPELIASIM_ROOT_DIR=~/path/to/coppeliaSim/folder
$ catkin build --cmake-args -DCMAKE_BUILD_TYPE=Release
```

Note for Ubuntu 20.04: due to a bug in python-catkin-tools you need to use catkin_make instead of catkin build.

That's it! The packages should have been generated and compiled to a library. Copy the devel/lib/libsimROS.so file to the CoppeliaSim installation folder. The plugin is now ready to be used!

Now open a terminal and start the ROS master with:

```
$ roscore
```

Open another terminal, move to the CoppeliaSim installation folder, start CoppeliaSim and load the ROS plugin with above's Lua code. Upon succesful ROS Interface load, checking the available nodes gives this:

```
$ rosnode list
/rosout
/sim_ros_interface
```

In an empty CoppeliaSim scene, select an object, then attach a non-threaded simulation script to it with [Add > Script > simulation script > Non threaded > Lua/Python]. Open the script editor for that script and replace the content with following:

Python | Lua

```
#python

def subscriber_callback(msg):
    # This is the subscriber callback function
    sim.addLog(sim.verbosity_scriptinfos, f'subscriber receiver following Float32: {str(msg["data"])}')

def get_transform_stamped(obj_handle, name, rel_to, rel_to_name):
    # This function retrieves the stamped transform for a specific object
    t = sim.getSystemTime()
    p = sim.getObjectPosition(obj_handle, rel_to)
    o = sim.getObjectQuaternion(obj_handle, rel_to)

    return {
        'header': {
            'stamp': t,
            'frame_id': rel_to_name
        },
        'child_frame_id': name,
        'transform': {
            'translation': {'x': p[0], 'y': p[1], 'z': p[2]},
            'rotation': {'x': o[0], 'y': o[1], 'z': o[2], 'w': o[3]}
        }
    }

def sysCall_init():
    sim = require('sim')
    simROS = require('simROS')

    # The simulation script initialization
    self.object_handle = sim.getObject('..')
    self.object_alias = sim.getObjectAlias(object_handle, 3)

    # Prepare the float32 publisher and subscriber (we subscribe to the topic we advertise):
    self.publisher = simROS.advertise('/simulationTime', 'std_msgs/Float32')
    self.subscriber = simROS.subscribe('/simulationTime', 'std_msgs/Float32', subscriber_callback)

def sysCall_actuation():
    # Send an updated simulation time message and send the transform of the object attached to this script
    simROS.publish(self.publisher, {'data': sim.getSimulationTime()})
    simROS.sendTransform(get_transform_stamped(self.object_handle, self.object_alias, -1, 'world'))
```

```
    # To send several transforms at once, use simROS.sendTransforms instead (if such a method exists in Python's simROS)

def sysCall_cleanup():
    # Following not really needed in a simulation script (i.e. automatically shut down at simulation end):
    simROS.shutdownPublisher(self.publisher)
    simROS.shutdownSubscriber(self.subscriber)
```

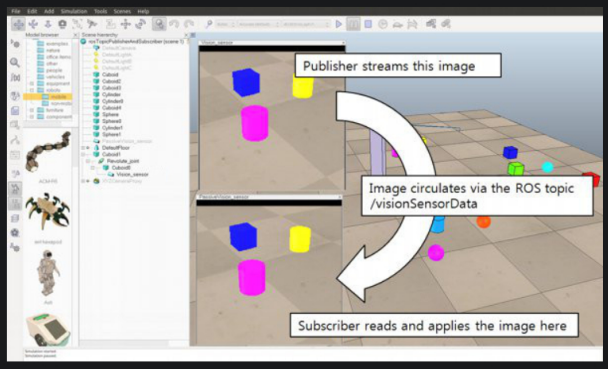
Above script will publish the simulation time, and subscribe to it at the same time. It will also publish the transform of the object the script is attached to. You should be able to see the simulation time topic with:

```
$ rostopic list
```

To see the message content, you can type:

```
$ rostopic echo /simulationTime
```

Now load the demo scene *messaging/rosInterfaceTopicPublisherAndSubscriber.ttt*, and run the simulation. The code in the simulation script attached to *Vision_sensor* will enable a publisher to stream the vision sensor's image, and also enable a subscriber to listen to that same stream. The subscriber applies the read data to the passive vision sensor, that is only used as a data container. So CoppeliaSim is streaming data, while listening to the same data! This is what is happening:



[Image publisher and image subscriber demo]

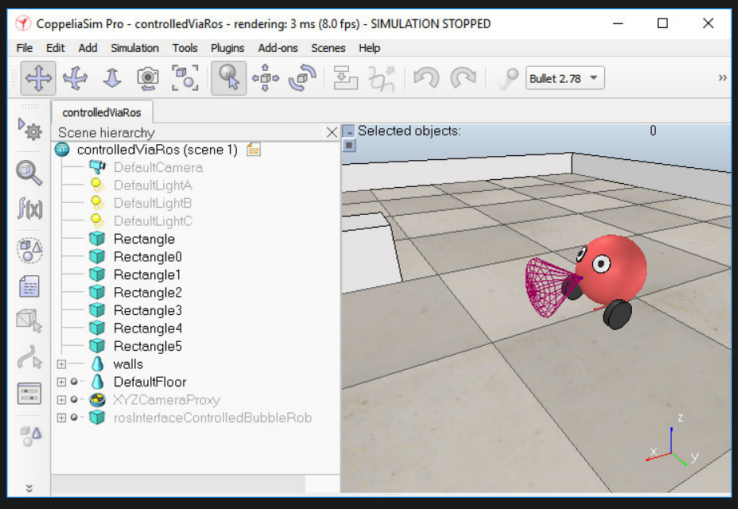
Try experimenting a little bit with the code. You can also visualize the image that CoppeliaSim streams with following command:

```
$ rosrun image_view image_view image:=/image
```

Had you been streaming simpler data, then you could also have visualized it with:

```
$ rostopic echo /image
```

Now stop the simulation and load the demo scene *controlTypeExamples/controlledViaRos.ttt*, and run the simulation. The robot is simplistic, and also behaving in a simplistic way for simplification purposes. It is controlled via the ROS Interface:



[External client application controlling the robot via ROS]

The simulation script attached to the robot, and running in a non-threaded fashion, is in charge of following:

- determine some object handles (e.g. motor joint handles and proximity sensor handle)
- Launch motor speed subscribers
- Launch a sensor publisher and a simulation time publisher
- and finally launch a client application. The application is called with some topic names as arguments, so that it will know which topics to listen to and to subscribe. The client application (*rosBubbleRob*) is then taking over the control of the robot via ROS.

While simulation is running, copy and paste a few times the robot. Notice that every copy is directly operational and independent. This is one of the many strengths of CoppeliaSim.

Now stop the simulation and open a new scene, then drag following model into it: *Models/tools/rosInterface helper tool.ttm*.

This model is constituted by a single customization script that offers following topic publishers and subscribers:

- *startSimulation* topic: can be used to start a simulation by publishing on this topic a *std_msgs::Bool* message.
- *pauseSimulation* topic: can be used to pause a simulation by publishing on this topic a *std_msgs::Bool* message.
- *stopSimulation* topic: can be used to stop a simulation by publishing on this topic a *std_msgs::Bool* message.
- *enableSyncMode* topic: by publishing a *std_msgs::Bool* message on this topic, you can enable/disable the stepping mode.
- *triggerNextStep* topic: by publishing a *std_msgs::Bool* message on this topic, you can trigger the next simulation step, while in the stepping mode.
- *simulationStepDone* topic: a message of type *std_msgs::Bool* will be published at the end of each simulation pass.
- *simulationState* topic: messages of type *std_msgs::Int32* will be published on a regular basis. *0* indicates that the simulation is stopped, *1* that it is running and *2* that it is paused.
- *simulationTime* topic: messages of type *std_msgs::Float32* will be published on a regular basis, indicating the current simulation time.

Have a look at the content of the customization script, that can be fully customized for various purposes. Try generating topic messages from the command line, for instance:

```
$ rostopic pub /startSimulation std_msgs/Bool true --once
$ rostopic pub /pauseSimulation std_msgs/Bool true --once
$ rostopic pub /stopSimulation std_msgs/Bool true --once
$ rostopic pub /enableSyncMode std_msgs/Bool true --once
$ rostopic pub /startSimulation std_msgs/Bool true --once
$ rostopic pub /triggerNextStep std_msgs/Bool true --once
$ rostopic pub /triggerNextStep std_msgs/Bool true --once
$ rostopic pub /triggerNextStep std_msgs/Bool true --once
$ rostopic pub /stopSimulation std_msgs/Bool true --once
```

In order to display the current simulation time, you could type:

```
$ rostopic echo /simulationTime
```

Finally, make sure to have a look at the remote API functionality in CoppeliaSim: it allows for remote function execution, fast data streaming back and forth, is quite simple to use, lightweight and cross-platform. The remote API functionality is available for 7 different languages and can be an interesting alternative to ROS in some cases.