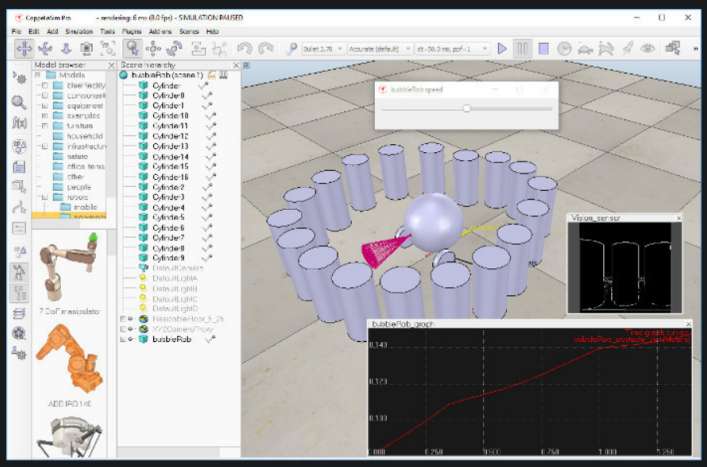
BubbleRob tutorial

This tutorial introduces various functionality while designing the simple mobile robot *BubbleRob*. The CoppeliaSim scene file related to this tutorial is located in *scenes/tutorials/BubbleRob*. Following figure illustrates the simulation scene that we will design:



Since this tutorial will fly over many different aspects, make sure to also have a look at the other tutorials, mainly the tutorial about building a simulation model. First of all, freshly start CoppeliaSim. The simulator displays a default scene. We will start with the body of *BubbleRob*.

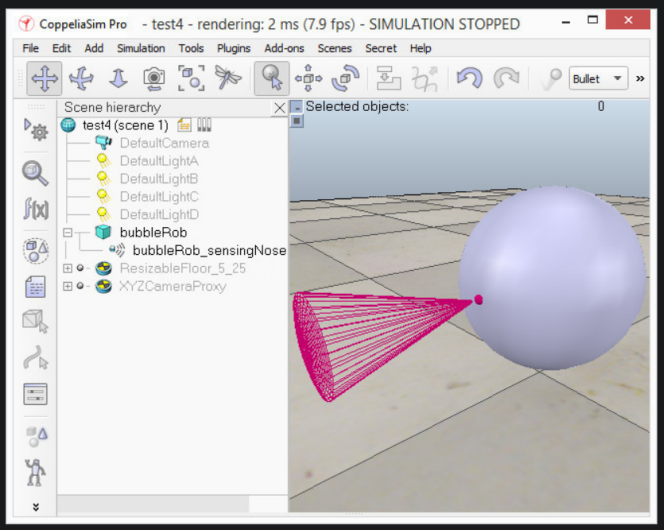
We add a primitive sphere of diameter 0.2 to the scene with [Add > Primitive shape > Sphere]. We adjust the **X-size** item to 0.2, then click **OK**. The created sphere will appear in the visibility layer 1 by default, and be dynamic and respondable (since we kept the item **Create dynamic and respondable shape** enabled). This means that *BubbleRob's* body will be falling and able to react to collisions with other respondable shapes (i.e. simulated by the physics engine). We can see this is the shape dynamics dialog: items **Body is respondable** and **Body is dynamic** are enabled. We start the simulation (via the toolbar button, or by pressing <control-space> in the scene window), and copy-and-paste the created sphere (with [Edit > Copy object(s)] then [Edit -> Paste buffer], or with <control-c> then <control-v>): the two spheres will react to collision and roll away. We stop the simulation: the duplicated sphere will automatically be removed. This default behaviour can be modified in the simulation dialog.

We also want the *BubbleRob's* body to by usable by the other calculation modules (e.g. distance calculation). For that reason, we enable Collidable, Measurable and Detectable in the general scene object properties dialog for that shape, if not already enabled. If we wanted, we could now also change the visual appearance of our sphere in the shape dialog.

Now we open the position dialog on the **translation** tab, select the sphere representing *BubbleRob's* body, and enter 0.02 for **Along Z**. We make sure that the **Relative to**-item is set to **World**. Then we click **Translate selection**. This translates all selected objects by 2 cm along the absolute Z-axis, and effectively lifted our sphere a little bit. In the scene hierarchy, we double-click the sphere's alias, so that we can edit it. We enter *bubbleRob* and press enter.

Next we will add a proximity sensor so that *BubbleRob* knows when it is approaching obstacles: we select [Add > Proximity sensor > Cone type]. In the orientation dialog on the **orientation** tab, we enter 90 for **Around Y** and for **Around Z**, then click **Rotate selection**. In the position dialog, on the **position** tab, we enter 0.1 for **X-coord.** and 0.12 for **Z-coord.** The proximity sensor is now correctly positioned relative to *BubbleRob's* body. We double-click the proximity sensor's icon in the scene hierarchy to open its properties dialog. We click **Show volume parameter** to open the proximity sensor volume dialog. We adjust items **Offset** to 0.005, **Angle** to 30 and **Range** to 0.15. Then, in the proximity sensor dialog, we click **Show detection parameters**. This opens the proximity sensor detection parameter dialog. We uncheck item **Don't allow detections if distance smaller than** then close that dialog again. In the scene hierarchy, we double-click the proximity sensor's alias in order to edit it. We enter *sensingNose* and press enter.

We select *sensingNose*, then control-select *bubbleRob*, then click [Edit > Set parent, keep pose(s)]. This attaches the sensor to the body of the robot. We could also have dragged *sensingNose* onto *bubbleRob* in the scene hierarchy. This is what we now have:

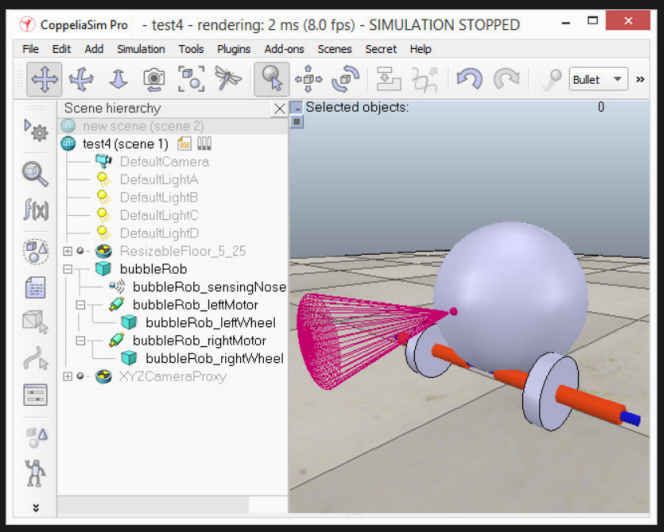


[Proximity sensor attached to *bubbleRob's* body]

Next we will take care of *BubbleRob's* wheels. We create a new scene with [File > New scene]. It is often very convenient to work across several scenes, in order to visualize and work only on specific elements. We add a primitive cylinder with dimensions (0.08,0.08,0.02). As for the body of *BubbleRob*, we enable Collidable, Measurable and Detectable in the general scene object properties dialog for that cylinder, if not

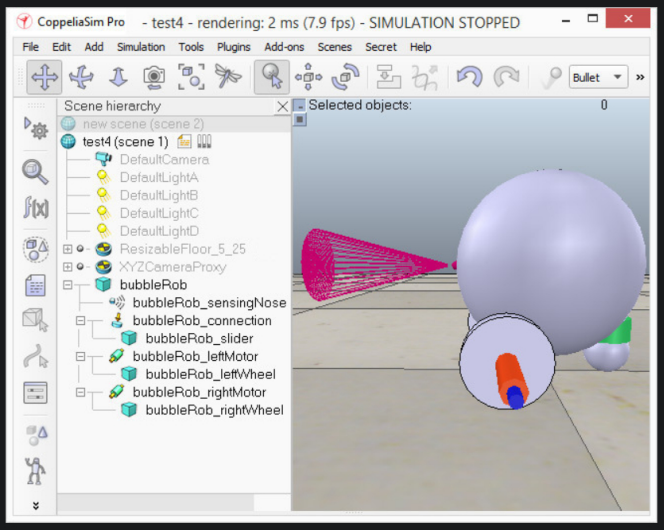
already enabled. Then we set the cylinder's absolute position to (0.05,0.1,0.04) and its absolute orientation to (-90,0,0). We change the alias to *leftWheel*. We copy and paste the wheel, and set the absolute Y coordinate of the copy to -0.1. We rename the copy to *rightWheel*. We select the two wheels, copy them, then switch back to scene 1, then paste the wheels.

We now need to add joints (or motors) for the wheels. We click [Add > Joint > Revolute] to add a revolute joint to the scene. Most of the time, when adding a new object to the scene, the object will appear at the origin of the world. We Keep the joint selected, then control-select *leftWheel*. In the position dialog, on the **position** tab, we click the **Apply to selection** button: this positioned the joint at the center of the left wheel. Then, in the orientation dialog, on the **orientation** tab, we do the same: this oriented the joint in the same way as the left wheel. We rename the joint to *leftMotor*. We now double-click the joint's icon in the scene hierarchy to open the joint dialog. Then we click **Show dynamic parameters** to open the joint dynamics dialog. We select the *velocity* **control mode**. We now repeat the same procedure for the right motor and rename it to *rightMotor*. Now we attach the left wheel to the left motor, the right wheel to the right motor, then attach the two motors to *bubbleRob*. This is what we have:



[Proximity sensor, motors and wheels]

We run the simulation and notice that the robot is falling backwards. We are still missing a third contact point to the floor. We now add a small slider (or caster). In a new scene we and add a primitive sphere with diameter 0.05 and make the sphere Collidable, Measurable and Detectable (if not already enabled), then rename it to *slider*. We set the **Material** to *noFrictionMaterial* in the shape dynamics dialog. To rigidly link the slider with the rest of the robot, we add a force sensor object with [Add > Force sensor]. We rename it to *connection* and shift it up by 0.05. We attach the slider to the force sensor, then copy both objects, switch back to scene 1 and paste them. We then shift the force sensor by -0.07 along the absolute X-axis, then attach it to the robot body. If we run the simulation now, we can notice that the slider is slightly moving in relation to the robot body: this is because both objects (i.e. *slider* and *bubbleRob*) are colliding with each other. To avoid strange effects during dynamics simulation, we have to inform CoppeliaSim that both objects do not mutually collide, and we do this in following way: in the shape dynamics dialog, for *slider* we set the **local respondable mask** to 00001111, and for *bubbleRob*, we set the **local respondable mask** to 11110000. If we run the simulation again, we can notice that both objects do not interfere anymore. This is what we now have:



[Proximity sensor, motors, wheels and slider]

We run the simulation again and notice that *BubbleRob* slightly moves, even with locked motor. We also try to run the simulation with different physics engines: the result will be different. Stability of dynamic simulations is tightly linked to masses and inertia matrices of the involved non-static shapes. For an explanation of this effect, make sure to carefully read this and that sections. We now try to correct for that undesired effect. We select the two wheels and the slider, and increase the masses by a factor 8 via [Edit > Shape mass and inertia > scale mass...]. We do the same with the inertia matrices of the 3 selected shapes ([Edit > Shape mass and inertia > scale inertia...]), then run the simulation again: stability has improved. In the joint dynamics dialog, we set the **Target velocity** to 50 for both motors. We run the simulation: *BubbleRob* now moves forward and eventually falls off the floor. We reset the **Target velocity** item to zero for both motors.

The object *bubbleRob* is at the base of all objects that will later form the *BubbleRob* model. We will define the model a little bit later. Next we are going to add a graph object to *BubbleRob* in order to display its clearance distance. We click [Add > Graph] and rename it to *graph*. We attach the graph to *bubbleRob*, and set the graph's absolute coordinates to (0,0,0.005).

Now we set one motor **target velocity** to 50, run the simulation, and will see *BubbleRob's* trajectory displayed in the scene. We then stop the simulation and reset the motor target velocity to zero.

We add a primitive cylinder with following dimensions: (0.1, 0.1, 0.2). We want this cylinder to be static (i.e. not influenced by gravity or collisions) but still exerting some collision responses on non-static respondable shapes. For this, we disable **Body is dynamic** in the shape dynamics dialog. We also want our cylinder to be Collidable, Measurable and Detectable. We do this in the general scene object properties dialog. Now, while the cylinder is still selected, we click the object translation toolbar button:

Now we can drag any point in the scene: the cylinder will follow the movement while always being constrained to keep the same Z-coordinate. We copy and paste the cylinder a few times, and move them to positions around *BubbleRob* (it is most convenient to perform that while looking at the scene from the top). During object shifting, holding down the shift key allows to perform smaller shift steps. Holding down the ctrl key allows to move in an orthogonal direction to the *regular* direction(s). When done, select the camera pan toolbar button again:

We set a **target velocity** of 50 for the left motor and run the simulation: the graph view now displays the distance to the closest obstacle and the distance segment is visible in the scene too. We stop the simulation and reset the target velocity to zero.

We now need to finish **BubbleRob** as a model definition. We select the model base (i.e. object *bubbleRob*) then check **Object is model base** in the general scene object properties dialog: there is now a stippled bounding box that encompasses all objects in the model hierarchy. We select the two joints, the proximity sensor and the graph, then enable item **Ignored by model bounding box** and click **Apply to selection**, in the same dialog: the model bounding box now ignores the two joints and the proximity sensor. Still in the same dialog, we disable **camera visibility layer** 2, and enable **camera visibility layer** 10 for the two joints and the force sensor: this effectively hides the two joints and the force sensor, since layers 9-16 are disabled by default. At any time we can modify the visibility layers for the whole scene. To finish the model definition, we select the vision sensor, the two wheels, the slider, and the graph, then enable item **Select base of model instead**: if we now try to select an object in our model in the scene, the whole model will be selected instead, which is a convenient way to handle and manipulate the whole model as a single object. Additionally, this protects the model against inadvertant modification. Individual objects in the model can still be selected in the scene by click-selecting them with control-shift, or normally selecting them in the scene hierarchy. We finally collapse the model tree in the scene hierarchy.

Next we will add a vision sensor, at the same pose as *BubbleRob's* proximity sensor. We open the model hierarchy again, then click [Add > Vision sensor > Perspective type], then attach the vision sensor to the proximity sensor, and set the local pose of the vision sensor to (0,0,0). We also make sure the vision sensor is not not visible, not part of the model bounding box, and that if clicked, the model will be selected instead. In order to customize the vision sensor, we open its properties dialog. We set the **Far clipping plane** item to 1, and the **Resolution x** and **Resolution y** items to 256 and 256. We add a floating view to the scene, and over the newly added floating view, right-click [Popup menu > View > Associate view with selected vision sensor] (we make sure the vision sensor is selected during that process).

We attach a simulation script to the vision sensor by clicking [Add > Script > simulation script > Non threaded > Lua]. We double-click the new script icon in the scene hierarchy: this opens the simulation script that we just added. We copy and paste following code into the script editor, then close it:

Python Lua

```python
#python

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

def sysCall_vision(inData):
    simVision.sensorImgToWorkImg(inData['handle']) # copy the vision sensor image to the work image
    simVision.edgeDetectionOnWorkImg(inData['handle'], 0.2) # perform edge detection on the work image
    simVision.workImgToSensorImg(inData['handle']) # copy the work image to the vision sensor image buffer
```

To be able to see the vision sensor's image, we start the simulation, then stop it again.

The last thing that we need for our scene is a simulation script that controls *BubbleRob's* behavior. We select *bubbleRob* and click [Add > Script > simulation script > Non threaded > Lua]. We double-click the new script icon in the scene hierarchy and copy and paste following code into the script editor, then close it:

Python Lua

```python
#python

import math

def sysCall_init():
    # This is executed exactly once, the first time this script is executed
    sim = require('sim')
    simUI = require('simUI')
    self.bubbleRobBase = sim.getObject('..') # this is bubbleRob's handle
    self.leftMotor = sim.getObject("../leftMotor") # Handle of the left motor
    self.rightMotor = sim.getObject("../rightMotor") # Handle of the right motor
    self.noseSensor = sim.getObject("../sensingNose") # Handle of the proximity sensor
    self.minMaxSpeed = [50*math.pi/180, 300*math.pi/180] # Min and max speeds for each motor
    self.backUntilTime = -1 # Tells whether bubbleRob is in forward or backward mode
    self.robotCollection = sim.createCollection(0)
    sim.addItemToCollection(self.robotCollection, sim.handle_tree, self.bubbleRobBase, 0)
    self.distanceSegment = sim.addDrawingObject(sim.drawing_lines, 4, 0, -1, 1, [0, 1, 0])
    self.robotTrace = sim.addDrawingObject(sim.drawing_linestrip + sim.drawing_cyclic, 2, 0, -1, 200, [1, 1, 0], None, None, [1, 1, 0])
    self.graph = sim.getObject('../graph')
    # sim.destroyGraphCurve(self.graph, -1)
    self.distStream = sim.addGraphStream(self.graph, 'bubbleRob clearance', 'm', 0, [1, 0, 0])
    # Create the custom UI:
    xml = '<ui title="' + sim.getObjectAlias(self.bubbleRobBase, 1) + ' speed" closeable="false" resizable="false" activate="false">'
    xml = xml + '<hslider minimum="0" maximum="100" on-change="speedChange_callback" id="1"/>'
    xml = xml + '<label text="" style="* {margin-left: 300px;}"></label></ui>'
    self.ui = simUI.create(xml)
    self.speed = (self.minMaxSpeed[0] + self.minMaxSpeed[1]) * 0.5
    simUI.setSliderValue(self.ui, 1, 100 * (self.speed - self.minMaxSpeed[0]) / (self.minMaxSpeed[1] - self.minMaxSpeed[0]))

def sysCall_sensing():
    result, distData, *_ = sim.checkDistance(self.robotCollection, sim.handle_all)
    if result > 0:
        sim.addDrawingObjectItem(self.distanceSegment, None)
        sim.addDrawingObjectItem(self.distanceSegment, distData)
        sim.setGraphStreamValue(self.graph,self.distStream, distData[6])
    p = sim.getObjectPosition(self.bubbleRobBase)
    sim.addDrawingObjectItem(self.robotTrace, p)

def speedChange_callback(ui, id, newVal):
    self.speed = self.minMaxSpeed[0] + (self.minMaxSpeed[1] - self.minMaxSpeed[0]) * newVal / 100

def sysCall_actuation():
    result, *_ = sim.readProximitySensor(self.noseSensor) # Read the proximity sensor
    # If we detected something, we set the backward mode:
    if result > 0:
        self.backUntilTime = sim.getSimulationTime() + 4
    if self.backUntilTime < sim.getSimulationTime():
        # When in forward mode, we simply move forward at the desired speed
        sim.setJointTargetVelocity(self.leftMotor, self.speed)
```

```
        sim.setJointTargetVelocity(self.rightMotor, self.speed)
    else:
        # When in backward mode, we simply backup in a curve at reduced speed
        sim.setJointTargetVelocity(self.leftMotor, -self.speed / 2)
        sim.setJointTargetVelocity(self.rightMotor, -self.speed / 8)

def sysCall_cleanup():
    simUI.destroy(self.ui)
```

We run the simulation. *BubbleRob* now moves forward while trying to avoid obstacles (in a very basic fashion). While the simulation is still running, change *BubbleRob's* velocity, and copy/paste it a few times. Be aware that the minimum distance calculation functionality might be heavily slowing down the simulation, depending on the environment.

Using a script to control a robot or model is only one way of doing. CoppeliaSim offers many different ways (also combined), have a look at the external controller tutorial.