1 Notes

First off, say that this document is a slightly modified version of VREPs Bubble Rob Tutorial.

In V-REP there are **Pure shapes** which have dynamics and physics (a sphere), and **Regular shapes** which are just a mesh (the skeleton of a shape).

When building a new model, first, we handle only the visual aspect of it (we want it to look right). The dynamic aspect: its underlying model of how it works, joints, sensors, etc. will be handled at a later stage.

2 Body

- We add a primitive sphere of diameter 0.2 to the scene with [Menu bar → 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[1], and be dynamic and respondable (since we kept the item *Create dynamic and respondable shape* enabled).

- To see the dynamic properties, first select the object properties (click on the sphere and then click on the magnifying glass on the icon bar to the left). Then, click on the "Show Dynamic Properties Dialogue" and check: items "Body is respondable" and "Body is dynamic" are enabled.

2.1 Test the Spheres

- Copy and paste the sphere. Now you have two solid and dynamic spheres in one spot (*sphere* and *sphere0*. Once you run the simulation (press "play") the two spheres will rapidly separate from each other[2].
- Now, select the sphere on the Scene hierarchy and press delete to remove it.
- We also want the BubbleRob's body to by usable by the other calculation modules (e.g. the minimum distance calculation module). For that reason, we enable Collidable, Measurable, Renderable and Detectable in the **object common properties** for that shape, if not already enabled. For this, click on the magnifying glass on the icon bar to the left. Then click on the tab that says "common." If we wanted, we could now also change the visual appearance of our sphere in the "shape" properties in the same dialog.

[1] There are many layers in which to build

[2] two physical objects cannot occupy the same space, so the shape's physics tries to correct this, resulting in a rapid separation

3 Position it

- Open the position and translation dialogue (icon with a cube and 4 arrows on the top icon bar)
- Select the sphere (the robot's body)
- Select "Translation" and enter 0.02 for Along Z. We make sure that the Relative to-item is set to World.
- Then we click Z-Translate selection. This translates all selected objects by 2 cm along the absolute Z-axis, and effectively lifted our sphere a little bit.

4 Name it

In the **scene hierarchy**, we double-click the sphere's name (*sphere*), so that we can edit its name. We enter "bubbleRob" and press enter.

5 Proximity Sensor

- select [Menu bar $\rightarrow$ Add $\rightarrow$ Proximity sensor $\rightarrow$ Cone type]. This will add a proximity sensor at $(0, 0, 0)$ pointing in the Z direction. We need to point it right and put it on the robot.
- Select the orientation and rotation dialogue (cube with two round arrows around it.)
- In the orientation and rotation dialog, under the tab "Rotation", we enter 90 for Around Y and for Around Z, then click Rotate selection
- Now we translate the sensor. In the position and translation dialog, in the section "Position", 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[3].

- We double-click the proximity sensor's **icon** in the scene hierarchy to open its properties dialog. If you click on its name, the property dialogue won't open.[4]
- click "Show volume parameters" to open the proximity sensor volume dialog. We adjust items Offset to 0.005, Angle to 30 and Range to 0.15. Here, we are adjusting our sensor reach, as well as offsetting a bit away from the sphere, like a sensor that "sticks out" as opposed to being flushed with the sphere.
- in the proximity sensor properties, we click "Show detection parameters."

[3] Think of why the sensor ends up positioned right where it is... do the math

[4] This is an alternative method to opening the properties dialogue. The other one is to click on the object once and then on the magnifying glass on the left menu bar

- In the proximity sensor detection parameter dialog. We uncheck item “Don’t allow detections if distance smaller than” then close that dialog again.

Notice how the colors and shape of the sensor are changing.

5.1 Name

In the scene hierarchy, we double-click the proximity sensor’s name, so that we can edit its name. We enter bubbleRob_sensingNose and press enter.

6 Link/Attach the sensor and the robot’s body

We select bubbleRob_sensingNose, then control+select[5] bubbleRob, then click [Menu bar → Edit → Make last selected object parent]. We could also have dragged bubbleRob_sensingNose onto bubbleRob in the scene hierarchy.

7 Wheels

It is often very convenient to work across several scenes, in order to visualize and work only on specific elements per scene. For example, one scene contains the body of the robot. Another contains the wheels and motors, etc. We will create a new scene for the wheels.

- create a new scene with [Menu bar → File → New scene]. The previous scene won’t be shown.
- Add a pure primitive cylinder with dimensions (0.08,0.08,0.02).
- Just like for the body of BubbleRob, we enable Collidable, Measurable, Renderable and Detectable in the object **common properties** for that cylinder (the magnifying glass), if not already enabled.
- In the Object/Item position dialog (cube w/ four arrows) set the cylinder’s absolute position to (0.05,0.1,0.04)
- Set its absolute orientation (cube with round arrows → orientation tab) to (-90,0,0)
- We change the name to bubbleRob_leftWheel.
- copy and paste the wheel, and set the absolute Y coordinate of the copy to -0.1.
- rename the copy to bubbleRob_rightWheel.
- We select the two wheels, copy them, then switch back to scene 1, then paste the wheels. They should be under the sphere and slightly forward.

[5] depending on your computer it can also be Shift+Select

8 Adding Motors (Joints)

- click [Menu bar → Add → Joint → Revolute] to add a revolute joint to the scene.
- Keep the joint selected, then control+select[6] bubbleRob_leftWheel.
- In the position and translation dialog, in section "position," we click the Apply to selection button. This positioned the joint at the center of the left wheel.
- Then, in the orientation and rotation dialog, in section Object/item orientation, we do the same: this oriented the joint in the same way as the left wheel.
- rename the joint to bubbleRob_leftMotor
- double-click the joint's icon in the scene hierarchy to open the joint properties dialog. Then we click Show dynamic parameters to open the joint dynamics properties dialog.
- Enable the motor, and check item "Lock motor when target velocity is zero."
- we attach the left wheel to the left motor. (Select the wheel first, together with the motor and Edit → Make last selected object parent).

Then repeat the steps by attaching a motor to the "bubbleRob_rightWheel" and renaming that motor to "bubbleRob_rightMotor." Lastly, attach the two motors (with their wheels) to bubbleRob.

It is important that it is the **motors** the ones attached to the robot, and then the wheels attached to the motors. Do not attach the wheels to the robot.

9 Adding a third contact point (slider/caster)

We run the simulation and notice that the robot is falling backwards. We are still missing a third contact point to the floor.

- In a new scene we and add a pure primitive sphere with diameter 0.05 (X-size).
- In the common properties, make the sphere Collidable, Measurable, Renderable and Detectable (if not already enabled)
- rename it to bubbleRob_slider
- set the Material to noFrictionMaterial in the "shape dynamics properties" under Edit Material→ Apply predifined settings. (this step is optional).
- To rigidly link the slider with the rest of the robot, we add a force sensor object with [Menu bar → Add → Force sensor]
- rename it to bubbleRob_connection

[6] or shift+select

- Shift it up by 0.05 by changing the Z-coord in the Object/Item position/orientation dialog.
- attach the slider to the force sensor (bubbleRob_connection)
- Copy both objects, switch back to scene 1 and paste them.
- Shift the force sensor by -0.07 along the absolute X-axis,
- 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. bubbleRob_slider and bubbleRob) are colliding with each other. To avoid strange effects during dynamics simulation, we have to inform V-REP that both objects do not mutually collide

- in the shape dynamics properties, for bubbleRob_slider we set the local respondable mask to 00001111 (That is, the first four check boxes are unchecked (0) and the last four are checked (1))
- for bubbleRob, we set the local respondable mask to 11110000 (This is the exact opposite of the previous mask).

If we run the simulation again, we can notice that both objects do not interfere anymore. However, we 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 inertias of the involved non-static shapes.

This has to do with the objects masses: Keep masses similar and not too light. When linking two shapes with a dynamically enabled joint or a dynamically enabled force sensor, make sure the two shape's masses are not too different ($m_1 < 10 * m_2$ and $m_2 < 10 * m_1$), otherwise the joint or force sensor might be very soft and wobbly and present large positional/orientational errors (this effect can however also be used as a natural damping sometimes).

Additionally, very low mass shapes should be avoided since they won't be able to exert very large forces onto other shapes (even if propelled by high force actuators!).

Lastly, the intertia has a role to play: Keep principal moments of inertia* relatively large. Try keeping the principal moments of inertia / mass (*refer to the shape dynamics properties dialog) relatively large, otherwise mechanical chains might be difficult to control and/or might behave in a strange way.

Therefore, we need to modify the massses of the objects. We will multiply all involved object's masses by 8:

- select the two wheels and the slider and click on Tools $\rightarrow$ scene object properties or click the magnifying glass on the left.
- Then in the shape dynamics dialog we click three times M=M*2 (for selection).

And multiply the inertias by 8.

- select the two wheels and the slider and click on Tools → scene object properties, or click the magnifying glass on the left.
- Then in the shape dynamics dialog we click three times I=I*2 (for selection).

10 Make the robot move

For a first test: We need to spin the motors with some velocity. Select one motor and bring up its properties. In the "joint" tab, under "Sow dynamic properties" dialog, we set the Target velocity to 50 deg/s. Do this for the other motor as well. Then, run the simulation.

BubbleRob now moves forward and eventually falls off the floor. We reset the Target velocity item to zero for both motors. If BubbleRob does not move forward, but it goes backwards, then your motors are spinning in the wrong direction. You can rotate them if you need to. If BubbleRob starts spinning, then the velocities in the motors are not the same.

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. In the mean time, we want to define a collection of objects that represent BubbleRob.

- To define a collection object:
- Click [Menu bar → Tools → Collections] to open the collection dialog. (the dotted lasso with geometrical shapes on the left menu bar, does the same)
- In the collection dialog, we click Add new collection
- A new collection object appears in the list just below. It is empty.
- While the new collection item is selected in the list, select bubbleRob in the scene hierarchy
- click Add in the collection dialog. You should see a message saying "+From object (incl.) up to [bubbleRob] "
- edit the collection name, we double-click it, and rename it to bubbleRob_collection.
- Our collection is now defined as containing all objects of the hierarchy tree starting at the bubbleRob object (the collection's composition is displayed when you click on "Visualize Collection")
- We close the collection dialog.

11 Knowing where the robot is

At this stage we want to be able to track the minimum distance between BubbleRob and any other object.

- For that, we open the distance dialog with [Menu bar → Tools → Calculation module properties] or with the $f(x)$ button.
- In the distance calc tab, we click Add new distance object.
- select a distance pair: [collection] bubbleRob_collection - all other measurable objects in the scene.
- rename the distance object to bubbleRob_distance with a double click on its name.
- close the distance dialog.

This just added a distance object that will measure the smallest distance between collection bubbleRob_collection (i.e. any measurable object in that collection) and any other measurable object in the scene.

When we now run the simulation, we won't see any difference, since the distance object will try to measure (and display) the smallest distance segment between BubbleRob and any other measurable object in the scene.

The problem is that at this stage there is no other measurable object in the scene (the shape defining the floor has its measurable property turned off by default). Just for kicks, add a cuboid to the scene, change its position to -2 on the X axis. Also, make it collidable, measurable, renderable and detectable. Then run the simulation. You should see a line from bubbleRob to the cuboid. You can safely delete the cuboid at this point.

12 Tracking The Robot

We will create a graph to check the robot trajectory as well as the distance to the closest object over time.

- click Menu bar → Add → Graph
- rename it to bubbleRob_graph
- attach the graph to bubbleRob (move it under bubbleRob)
- set the graph's absolute coordinates to (0,0,0.005)
- open the graph properties dialog by double-clicking its icon in the scene hierarchy
- uncheck "Display XYZ-planes"
- click "Add new data stream to record"
- select "Object: absolute x-position" for the Data stream type and bubbleRob_graph for the Object / item to record.

An item has appeared in the Data stream recording list. That item is a data stream of bubbleRob_graph's absolute x-position (i.e. the bubbleRobGraph's object absolute x position will be recorded). Now we also want to record the y and z positions. To do that, we add those data streams in a similar way as above. We now have 3 data streams that represent BubbleRob's x-, y- and z-trajectories. We are going to add one more data stream so that we are able to track the minimum distance between our robot and its environment:

- in the the graph properties dialog, click "Add new data stream to record" and select "Distance: segment length" for the Data stream type.
- select bubbleRob_distance for the Object / item to record
- In the Data stream recording list, we now rename Data to bubbleRob_x_pos, Data0 to bubbleRob_y_pos, Data1 to bubbleRob_z_pos, and Data2 to bubbleRob_obstacle_dist.

Now, we set up what is visible in the graph. Only the bubbleRob_obstacle_dist data stream will be visible in a time graph:

- select bubbleRob_x_pos in the Data Stream recording list and in the Time graph properties section, uncheck Visible
- We do the same for bubbleRob_y_pos and bubbleRob_z_pos.

Why did we need the x,y and z positions then? Well, we will set-up a 3D curve that displays BubbleRob's trajectory and the x,y and z positions are important there.

- click Edit 3D curves to open the XY graph and 3D curve dialog
- click Add new curve.
- we select bubbleRob_x_pos for the X-value item, bubbleRob_y_pos for the Y-value item and bubbleRob_z_pos for the Z-value item.
- rename the newly added curve from Curve to bubbleRob_path
- check the Relative to world item and set Curve width to 4:
- We close all dialogs related to graphs.

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.

Next, we will add a visualization window for the minimum distance data stream. Now, if you are on Windows, running the latest V-REP, you do not need to do this. It is done automatically for you. If you are running on a Mac, the following steps may or may not work.

- Right click inside the scene (popup menu) $\rightarrow$ Add $\rightarrow$ Floating view
- select bubbleRob_graph

- in the floating view, right-click [Popup menu → View → Associate view with selected graph].

Running the simulation will not yet display anything in the graph window, since there are still no objects (i.e. obstacles) to measure against. Let's now add some obstacles!

13 Adding Obstacles

We add a pure primitive cylinder. 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.

- add a pure primitive cylinder with following dimensions: (0.1, 0.1, 0.2).
- Now, while the cylinder is still selected, we click the object translation toolbar button and 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 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.
- disable Body is dynamic in the shape dynamics properties
- We also want our cylinder to be Collidable, Measurable, Renderable and Detectable. We do this in the object common properties
- 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).

Tip: 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).

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.

14 Add a Vision Sensor

Next we will add a vision sensor, at the same position and orientation as BubbleRob's proximity sensor. We will set up stuff such as how far it can "see" and the resolution of the camera. We will also set "what" it can see (these are algorithms that the vision sensor may work with, such as edge detection).

- click [Menu bar → Add → Vision sensor → Perspective type]
- then attach the vision sensor to the proximity sensor (by dragging it under bubbleRob_sensingNose)

- set the position of the vision sensor to (0,0,0). (It may turn your numbers into very small numbers of order of 10^{-7} or smaller (e.g. 1.2e-8) but that's close enough)
- set the orientation of the vision sensor to (0,90,90)
- Now this one may be challenging, but you can do it: make sure the vision sensor is not not visible,and that it is ignored by the model's bounding box, and that if clicked, the model will be selected instead (hint: object common properties, change its layer/or camera visibility, and set two other properties).
- open the vision sensor's properties dialog.
- Set the Far clipping plane item to 1m
- Set the Resolution x and Resolution y items to 256 and 256
- Select the vision sensor and right click on it. Then Add → Child Script → Non-Threaded. A little script icon will be present next to the sensor.
- Now, we will make it filter the image from the vision sensor, and only return the edges. This is called "edge detection." To do this, double click on the script icon, and add the following code and then close the scripting environment:

```
function sysCall_vision(inData)
    simVision.sensorImgToWorkImg(inData.handle)
    simVision.edgeDetectionOnWorkImg(inData.handle,0.2) --here,
    the threshold for vision is 0.2 (20cm).
    simVision.workImgToSensorImg(inData.handle)
end
```

  This is a "callback" function. So, when the vision sensor is active, it will be calling all of its functions. When it calls the `sysCall\_vision` function, your code will be run. Also, the language you are using here is called LUA script.

- add a floating view to the scene (popup menu → Add → Floating View)
- With the vision sensor selected, over the newly added floating view, right-click [Popup menu → View → Associate view with selected vision sensor]

To be able to see the vision sensor's image, we start the simulation, then stop it again. Try to comment the `edgeDetectionOnWorking` call and insert this right below it: `simVision.intensityScaleOnWorkImg(inData.handle,0.1,1.0,true)` This will render the image in grayscale. Change the last parameter to `false` and it will display it in color. Play with these functions. You may want to enable them both too. There are many filter functions like this documented in The coppelia's Vision Plugin API

15 Save as a Model

We now need to finish BubbleRob as a model definition.

- select the model base (i.e. object bubbleRob)
- Open the object common properties and then check items “Object is model base” and “Object/model can transfer or accept DNA”.
- there is now a bounding box that encompasses all objects in the model hierarchy.
- With the properties window open, select the two joints, the force sensor and the graph
- Enable item “Ignore by model bounding box”
- click “Apply to selection”
- select the two joints (motors) and disable camera visibility layer 2 (the second checkbox from the top left)
- 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. Click on “Apply to selection”
- we select the vision sensor, the two wheels, the slider, and the graph, then enable item “Select base of model instead”. Don’t forget to “Apply to selection.”

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.

Try running the simulation again. Play with it and change parameters, like motor speeds, etc. Try to have one motor spin with a certain speed, another at a different one. Perhaps with the exact same speed, but negative.

16 Controlling the Robot with a UI and a Child Script

Next, we want to be able to modulate BubbleRob’s velocity with an OpenGl-based custom UI. To do this we will create a script that instantiates a little user interface (UI) from which to control the robot.

16.1 Child Script

Finally, we will add some code to our components so that the speed of the robot is controlled by a graphical user interface (GUI or simply, UI). The UI will have a sliding bar to control the speed. For this we need to access the UIs API and read the values of said sliding bar. Then adjust the speed of the motors accordingly.

- select bubbleRob and click [Menu bar → Add → Associated child script → Non threaded]
- This just added a non-threaded child script to the scene, and associated it with bubbleRob. We can also add, remove or modify scripts via the script dialog which can be opened with [Menu bar → Tools → Scripts] or through the appropriate toolbar button:
- We double-click the little script icon that appeared next to bubbleRob's name in the scene hierarchy: this opens the child script that we just added.
- We copy and paste following code into the script editor, then close it:
- Check the comments on the script to see what it is doing. The language used here is LuaScript.

```
-- this modifies the speed making sure to never go over the max speed,
    nor under the min speed. minMaxSpeed is an array with the minimum
    and maximum speed of the motors.
function speedChange_callback(ui,id,newVal)
    speed=minMaxSpeed[1]+(minMaxSpeed[2]-minMaxSpeed[1])*newVal/100
end

function sysCall_init()
    -- This is executed exactly once, the first time this script is
    executed
    bubbleRobBase=sim.getObjectAssociatedWithScript(sim.handle_self) --
     this is bubbleRob's handle so we can manipulate the robot and its
    components.
    leftMotor=sim.getObjectHandle("bubbleRob_leftMotor") -- Handle of
    the left motor
    rightMotor=sim.getObjectHandle("bubbleRob_rightMotor") -- Handle of
     the right motor
    noseSensor=sim.getObjectHandle("bubbleRob_sensingNose") -- Handle
    of the proximity sensor
    minMaxSpeed={50*math.pi/180,300*math.pi/180} -- Min and max speeds
    for each motor
    backUntilTime=-1 -- Tells whether bubbleRob is in forward or
    backward mode
    -- Create the custom UI in xml:
        xml = '<ui title="'..sim.getObjectName(bubbleRobBase)..' speed"
     closeable="false" resizeable="false" activate="false">'..[[
        <hslider minimum="0" maximum="100" onchange="
    speedChange_callback" id="1"/>
        <label text="" style="* {margin-left: 300px;}"/>
        </ui>
```

```
        ]] -- check how we concatenate objects, and the function used
    when speed changes
    ui=simUI.create(xml) -- effectively creates a UI window with a
    slider.
    speed=(minMaxSpeed[1]+minMaxSpeed[2])*0.5 -- initial speed is at
    the middle.
    simUI.setSliderValue(ui,1,100*(speed-minMaxSpeed[1])/(minMaxSpeed
    [2]-minMaxSpeed[1])) -- set the slider value at the middle in
    percentage. (so it is within 0 and 100).
end

function sysCall_actuation() --always called to check actuators.
    result=sim.readProximitySensor(noseSensor) -- Read the proximity
    sensor
    -- If we detected something, we set the backward mode:
    if (result>0) then backUntilTime=sim.getSimulationTime()+4 end

    if (backUntilTime<sim.getSimulationTime()) then
        -- When in forward mode, we simply move forward at the desired
    speed
        sim.setJointTargetVelocity(leftMotor,speed)
        sim.setJointTargetVelocity(rightMotor,speed)
    else
        -- When in backward mode, we simply backup in a curve at
    reduced speed
        sim.setJointTargetVelocity(leftMotor,-speed/2)
        sim.setJointTargetVelocity(rightMotor,-speed/8)
    end
end

function sysCall_cleanup()
  simUI.destroy(ui)
end
```

Check these functions, and in particular, check `sysCall_actuation`. See if you can modify it to make the collision detection "smarter" For example, turn before hitting an object.