Documentation

IIoT Simulator

Java controlled V-REP simulation

Serkan Aktas

Index

Index

List of Figures

List of Figures

Introduction

This Documentation is about an IIoT Simulator. The goal of this simulator is to demonstrate how a robotic arm can be controlled and moved and gather some sensor data through a *remote API*.

For the simulation the software “*V-REP PRO EDU”* is used. “*V-REP”* is a robot simulator developed by Coppelia Robotics. “*V-REP”* is available in three versions. “*V-REP PRO EDU”*, “*V-REP PRO”* and “*V-REP PLAYER”*. “*V-REP PRO EDU”* is for educational purposes and free, “V-REP PRO” is the commercial version and “*V-REP PLAYER"* is a free software with “limited editing capabilities” to play templates.[1]

To access the simulator Java is used. “*V-REP”* offers a “*remote API”* for some programming language. Language currently supported are “C/C++”, “Python”, “Java”, “Matlab”, “Octave”, “Urbi” and “Lua”. [2]

V-REP can be used on Windows 64 bit, Linux 64 bit and macOS. *V-REP* also offers source code to download.

Getting started

After downloading and installing the desired version, there is a package called *coppelia* containing “12 Java classes” which can be used on Java projects and a “*remoteAPI Java” library* to register. All this files can be found in “*V-REP's* installation directory”. The shared library is platform specific.[3]

Every *V-REP* scene has a main script which usually launches child scripts. Main script is default and initializes everything. Child script are recommended to modify instead of the main script.

V-REP offers models which can be drag and drop to the scene. The models have some predefined script and behavior. It is possible to build or import own model to *V-REP*.
V-REP supports file-formats like “*OBJ”*, “*DXF”*,”*3DS”*,”*STL”*, “*COLLADA”* and “*URDF*.”[4]

It is possible to create *child scripts* and assign them to *models (Objects)*.
Communication with *Java* takes place with *functions* defined in a *child script*.
To create a *child script* there is a bar on the left side with a symbol called *Scripts* marked red in fig. 1.

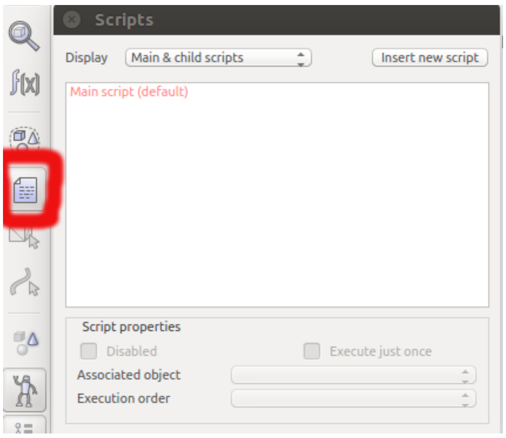

Abbildung 1: Scripts

This symbol will open the Scripts window where there are the option to *insert new script*, assign a script to an *object* and the option to change the *execution* to *just once*. The first step to enable *Remote API* is to create a new *non-threaded-script* .

```
if (sim_call_type==sim_childscriptcall_initialization) then

    simExtRemoteApiStart(19999)
end
```

Abbildung 2: SimExtRemoteAPIStart

In the new created *non-threaded-script* the *regular API function* `SimExtRemoteAPIStart(19999)` can be added. This function enables communication with a Java client on the `Port` number `19999`. In the Template this command is assigned to the *Dummy object* . This scrip is a *non-threaded-script*. Like all *child scripts* this *non-threaded-script* is launched by the *main script*.

An example how to connect to the server in a Java client is shown in fig.3

```
// Establish connection
clientId = api.simxStart(config.getIp(), config.getPort(), true, true, 5000, 5);
```

Abbildung 3: simxStart

In this example the `port` number and the `IP` number are defined in the class `Config`. The other values are `"waitUntilConnected"` , `"doNotReconnectOnceDisconnected"`, `"timeOutInMs"`, and `"commThreadCycleInMs"`. The values can be set to control the behavior of the connection.[5]

When this *method* is called from the *client side* and the *child script* in the *V-REP* scene is set up like in fig. 2 , than a connection should be possible. Important is to use the correct `port` number and the correct `IP` Number. `Port` number in this example it must be `19999`.

V-REP PhantomXPincher Template

The Template is very important. The *models* must be defined and a behavior must be implemented to theme to bring theme to move. This document shows the functions of the *Remote API* using the *PhantomXPincher* robot model.

The *template* which is used in this project has the robot model *PhantomXPincher* with added movement to control the robot with *sliders* created in *Java*. It also has a *proximity sensor* on the *left gripper* to detect a defined *cylinder* in the scene. *PhantomXPincher* is a *model* with several *links* and four *joints* that are put together to work as a unit. The model *PhantomXPincher* has a *child script* assign to it which describes its behavior. This script is a *threaded child script*. To call *threaded child scripts* through *remote API* the script must be running and don't be finished. It is important to configure and disable the option *Execute just once* in the *Scripts* menu.

To move robots *V-REP* uses "*Reflexxes Motion Library type II or IV"*. *RML* has vectors that must be set in the script. For the *PhantomXPincher* this can be like this:

```
jointHandles={-1,-1,-1,-1}
for i=1,4,1 do
    jointHandles[i]=simGetObjectHandle('PhantomXPincher_joint'..i)
end

modelBase=simGetObjectHandle('PhantomXPincher')
modelBaseName=simGetObjectName(modelBase)

vel=180
accel=40
jerk=80
currentVel={0,0,0,0}
currentAccel={0,0,0,0}
maxVel={vel*math.pi/180,vel*math.pi/180,vel*math.pi/180,vel*math.pi/180}
maxAccel={accel*math.pi/180,accel*math.pi/180,accel*math.pi/180,accel*math.pi/180}
maxJerk={jerk*math.pi/180,jerk*math.pi/180,jerk*math.pi/180,jerk*math.pi/180}
targetVel={0,0,0,0}
```

Abbildung 4: RML Vectors

"JointHandles" is handling the joints that are linked in the *model* .This robot has four *joints* to handle.
Vel stands for the *velocity of the joints*, *accel* stands for *acceleration* of the *joints* and *jerk* stands for *jerk* of the *joints*. This *vectors* also have *current values* and *max values* that can be defined like in fig. 4. These information are needed for the next step to move the robots.[6]

Moving the ***PhantomXPincher***

To move the *PhantomXPincher* there are functions created in *child script* of *PhantomXPincher*. This *functions* will be remotely called through *Java*.

```
Slider_function1=function(inInts,inFloats,inStrings,inBuffer)
--TargetPosition, inInts will be filled with slider results in java
targetPos1={inInts[1]*math.pi/180,inInts[2]*math.pi/180,inInts[3]*math.pi/180,inInts[4]*math.pi/180}
--function to move robot to desierd position
simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,maxAccel,maxJerk,targetPos1,targetVel)

        return {},{},{'message was displayed'},'' -- return a string
end
```

Abbildung 5: Slider_function1

The *Slider_function1* in the *child script* uses the function "simRMLMoveToJointPositions" which is part of the *regular API* and moves the robot to a *target position* which is defined in the *script*.[6]

In this example there is a defined *targetPos1* with four `inInts[1-4]`. This `inInts` are controlled through Java. The `inInts` get filled with *slider values* created in Java. There are four *functions* called `slider_function` every of them has a number 1 to 4. Every number represents a *slider*. Every *function* is filled with the same code. The *sliders* are defined in *Java*. The reason why there are four `slider_function` is that otherwise the other values would be lost. Goal is not to lose the current position and just control a *value* with one *slider*. The other *values* will stay untouched till the other *slider* gets moved. For example if *slider one* gets moved than it will only change the value of `inInts[1]` and when moving *slider two* the value of `inInts[2]` will be changed. `inInts[1]` will stay changed from the movement of *slider* one and not altered from *slider two*.

```
public boolean moveFirstSlider(int value) {
    sliderValues[0] = value;
    return api.simxCallScriptFunction(this, "Slider_function1", sliderValues);
}
```

Abbildung 6: moveSlider

Fig. 6 shows *methods* defined in the class `PhantomXPincher`. These are the *methods* that return the changed *values* from the *sliders* in Java to *V-REP*. The important point is that it calls the *Remote API* function `simxCallScriptFunction` in *Java* which allows to call a defined function in *V-REP* simulation. Like told earlier *threaded scripts* must still be running. The method `simxCallScriptFunction` is defined in the class `API` and is been used here with the two additional information, *name of the function* and the given input to the simulation to fill the `inInts` in `slider_function`. In this example `moveFirstSlider` in Java fills `inInts[1]` in function `slider_function1` with `sliderValues`. This value will be used to determinate the target position to move the robot. The function will call the `simRMLMoveToJointPositions` to move the robot to the desired position.[7]

Gripper functions

It is also possible to control the grippers of the robot. *PhantomXPincher* has the functionality to close and to open its *grippers*. This is controlled through *Java* with buttons.

```
public boolean openGripper() {
    return api.simxCallScriptFunction(this, "gripperopen");
}
```

Abbildung 7: Gripper Java

These methods do also use the *remote API* function `simxCallScriptFunction`. It does work similar to the move functions. The only difference is that there aren't inputs just a return value from the simulation. In this example it will just call the function `gripperopen`. `Gripperopen` is a defined function in the child script assigned to the model *PhantomXPincher*.

```
gripperclose=function(inInts,inFloats,inStrings,inBuffer)

simSetIntegerSignal(modelBaseName..'_gripperClose',1)
        return {},{},{'message was displayed'},'' -- return a string
    end
```

Abbildung 8: Gripper V-REP

The functions *gripperclose* and *gripperopen* use the *regular API* function “simSetIntegerSignal” [8]. To get the *modelBaseName* and the *ModelBase* the function “simGetObjectHandle”[9] and “*simGetObjectName”[10]* are called. The object is the *PhantomXPincher*. After getting the name it is possible to use the name and get to signal of *gripperClose* and set it to *1* to close the *gripper*. 0 would open the *gripper*. It is also possible to to create and manage signals through Java . The Class API has the function `simxSetIntegerSignal`.

```
if (sim_call_type==sim_childscriptcall_initialization) then
-- Handles initialization
    modelBase=simGetObjectHandle('PhantomXPincher')
    modelBaseName=simGetObjectName(modelBase)
    openCloseJoint=simGetObjectHandle('PhantomXPincher_gripperClose_joint')
end

if (sim_call_type==sim_childscriptcall_cleanup) then

end

if (sim_call_type==sim_childscriptcall_actuation) then

-- v is responsible for closing and opening the gripper.
-- Signal can be manipulated with the function simSetIntegerSignal.
    v=simGetIntegerSignal(modelBaseName..'_gripperClose')

-- If v value is not 0 than the gripper closes else it will be open.
    if (v==nil)or(v==0) then

-- Sets the desierd velocity to open gripper.
        simSetJointTargetVelocity(openCloseJoint,0.02)
    else

-- Sets the desierd velocity to close gripper.
        simSetJointTargetVelocity(openCloseJoint,-0.02)
    end
end
```

Abbildung 9: PhantomXPincher_gripperClose_joint

Fig. 9 shows the *child script* assign to the *gripper*. The function *gripperclose* in fig. 8 sets a *signal value* with `simSetIntegerSignal` , a *signal value* that will be received from the *child script* of *gripper*. To receive the signal `simGetIntegerSignal` will be used in the *gripper script*.

Read Proximity Sensor

`SimxReadProximitySensor` is a function of the *remote API*. This *function* reads the result of a *proximity sensor* and returns *detection state* and *detection point*. To use this function there must be a *sensor handle*. [11] To get the sensor handle the function `simxGetObjectHandle` must be called. This will return the *object handle* of the sensor.[12]

```java
    public boolean simxReadProximitySensor(Device device, ProximityResult result) {

        IntW sensor = new IntW(0);

        int returnCode = api.simxGetObjectHandle(clientId, device.getName()+"Proximity_sensor", sensor, remoteApi.simx_opmode_blocking);
        if (returnCode != remoteApi.simx_return_ok) {
            return false;
        }

        BoolW detState = new BoolW(false);
        FloatWA detectedPoint = new FloatWA(0);
        IntW detectedObjectHandle = new IntW(1);
        FloatWA SurfaceNormalVector = new FloatWA(1);

        returnCode = api.simxReadProximitySensor(clientId, sensor.getValue(), detState, detectedPoint,
                detectedObjectHandle, SurfaceNormalVector, remoteApi.simx_opmode_blocking);

        result.setObjectDetected(detState.getValue());
        result.setPoint(Arrays.toString(detectedPoint.getArray()));

        return returnCode == remoteApi.simx_return_ok;
    }
}
```

Abbildung 10: Proximity Sensor Java

In the example in fig. 10 the sensor is called *PhantomXPincherProximity_Sensor* .The return value will be saved in the *sensor* variable. After getting the *sensor handle* it is possible to use that result as *sensor handling* in the `simxReadProximitySensor` function of the *remote API*.

Own Model with Script

This part of the documentation will show how to build an own model in the *V-REP*. The best and realistic way to build a robot would be to create it with a CAD software. It is possible to create a robot with its constitution plan in a CAD software and import it to *V-REP*. Like told earlier *V-REP* supports file-formats like *OBJ*, *DXF*, *3DS*, *STL*, *COLLADA* and *URDF*. *V-REP* also offers *Primitive Shapes* that can be added into the *scene*. This *shapes* can be modified. To demonstrate how robots can be put together this documentation will put a robot together consisting of *primitive shapes* and *joints*.

The option *Add* in the menu bar allows the user to add a primitive shape. To add and modify a *cuboid* go to [*Menu bar --> Add --> Primitive shape --> Cuboid*] . This option will open the *Primitive cuboid* window in fig. 11. In this window you can modify the size of cuboid .This windows lets *X-size*,*Y-size* and *Z-size* change. In the right bottom of the main window you will see *X*,*Y* and *Z* as a help.

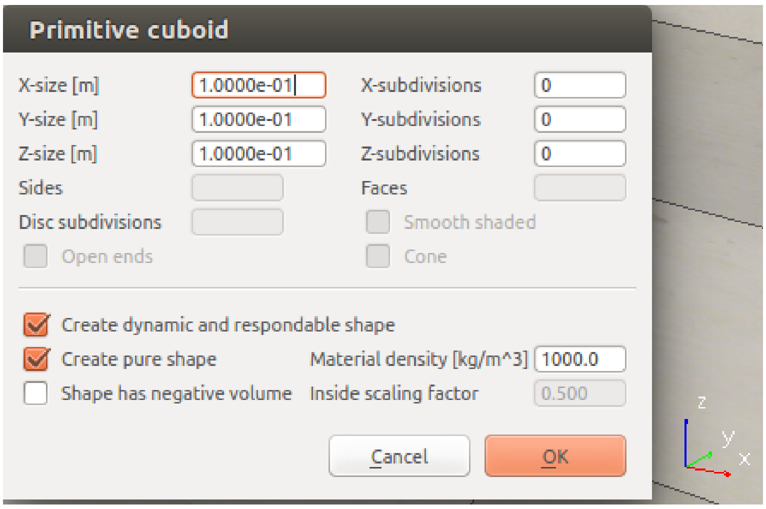

Abbildung 11: Primitive cuboid

After choosing a desired *size* and clicking OK, a *cuboid* should be displayed in the *scene*.
If there is the need to reconfigure the *size* it is possible to double click to the *cuboid* symbol in the *scene hierarchy* and open the *Scene Object Properties of the cuboid*.

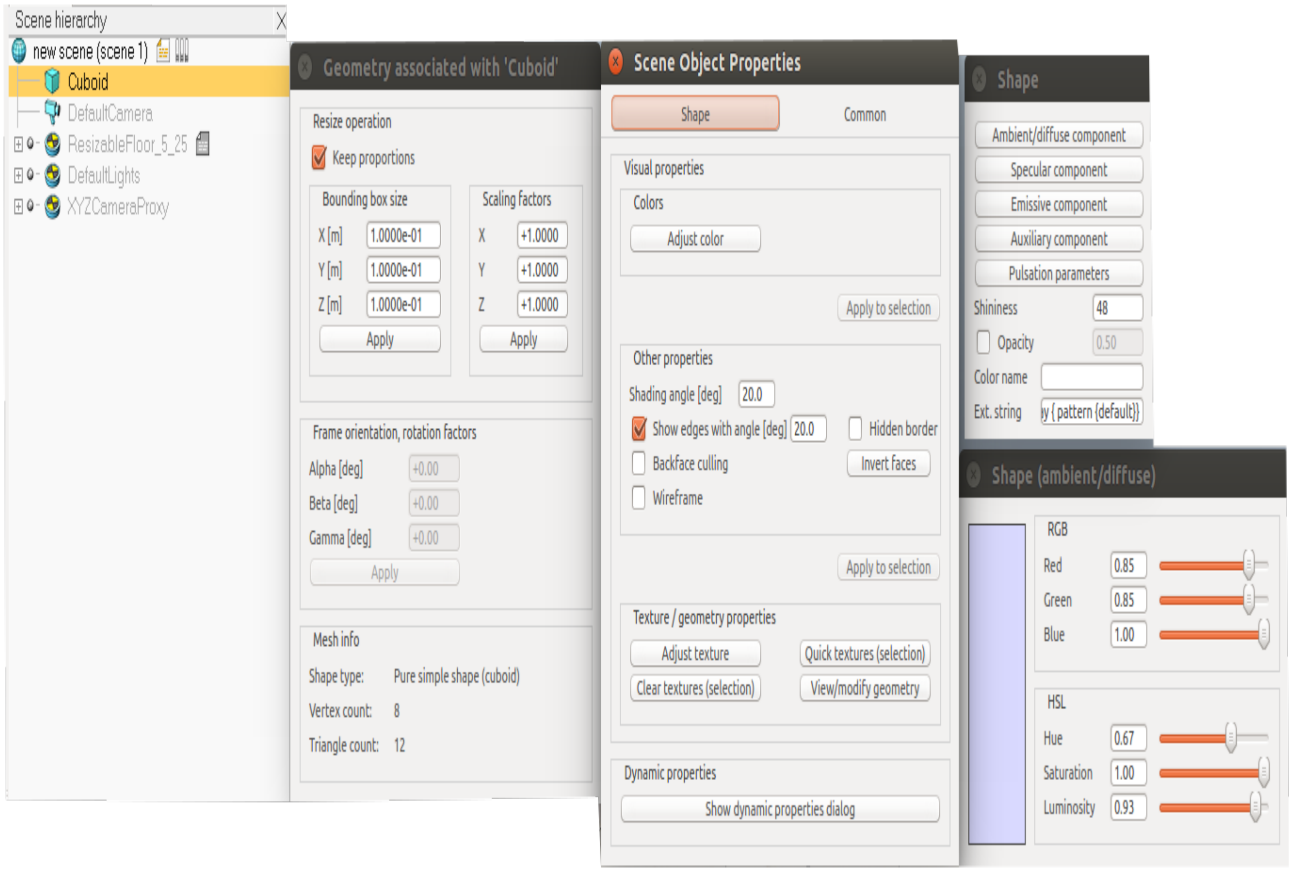

Abbildung 12: Scene Object Properties

In fig. 12 *view/modify geometry* button and *adjust color* button are in the *Scene Object Properties* window, *scene hierarchy* is in the upper left corner of the main window. Clicking the *view/modify geometry* button will open the *Geometry associated with 'Cuboid'* window. This window is managing the size of the object. Clicking *Adjust color* button will open the *Shape* window and clicking the *ambient/diffuse* component button will open the *Shape(ambient/diffuse)* window. This window can give the object the desired color.

With these windows it is possible to mange the size and the color of the cuboid.

The first step to build a robot is to build its *ground*. The size of the *ground cuboid* is x= 0.5, y= 0.3 and z = 0.05. It may be better to uncheck the *Keep Proportions* option in the Geometry associate with 'Cuboid' window to mange the size better. To rename the name of the *cuboid* double-click the name of the *cuboid* in the *scene hierarchy*. Write the new name and press enter to save the change. In the example the *ground cuboid* will be called *myRobot*.

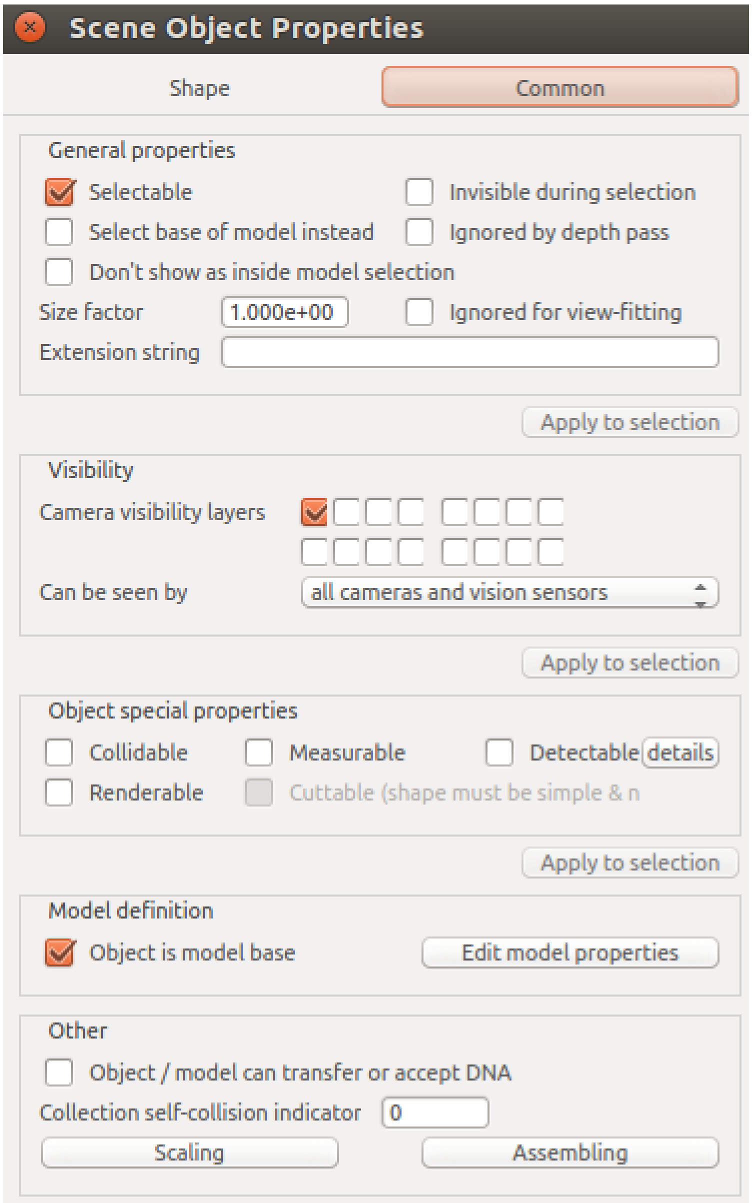

Abbildung 13: Scene Object Properties common

An import tab for models is the *common* tab in *Scene Object Properties* shown in fig. 13. There is the option to make an *object* the *model base*. It is also possible to configure if the *object* should be *Collidable*, *Measurable*, *Detectable* and *Renderable*. In this example these options are unchecked. The created *cuboid* called *myRobot* is the *model base*.

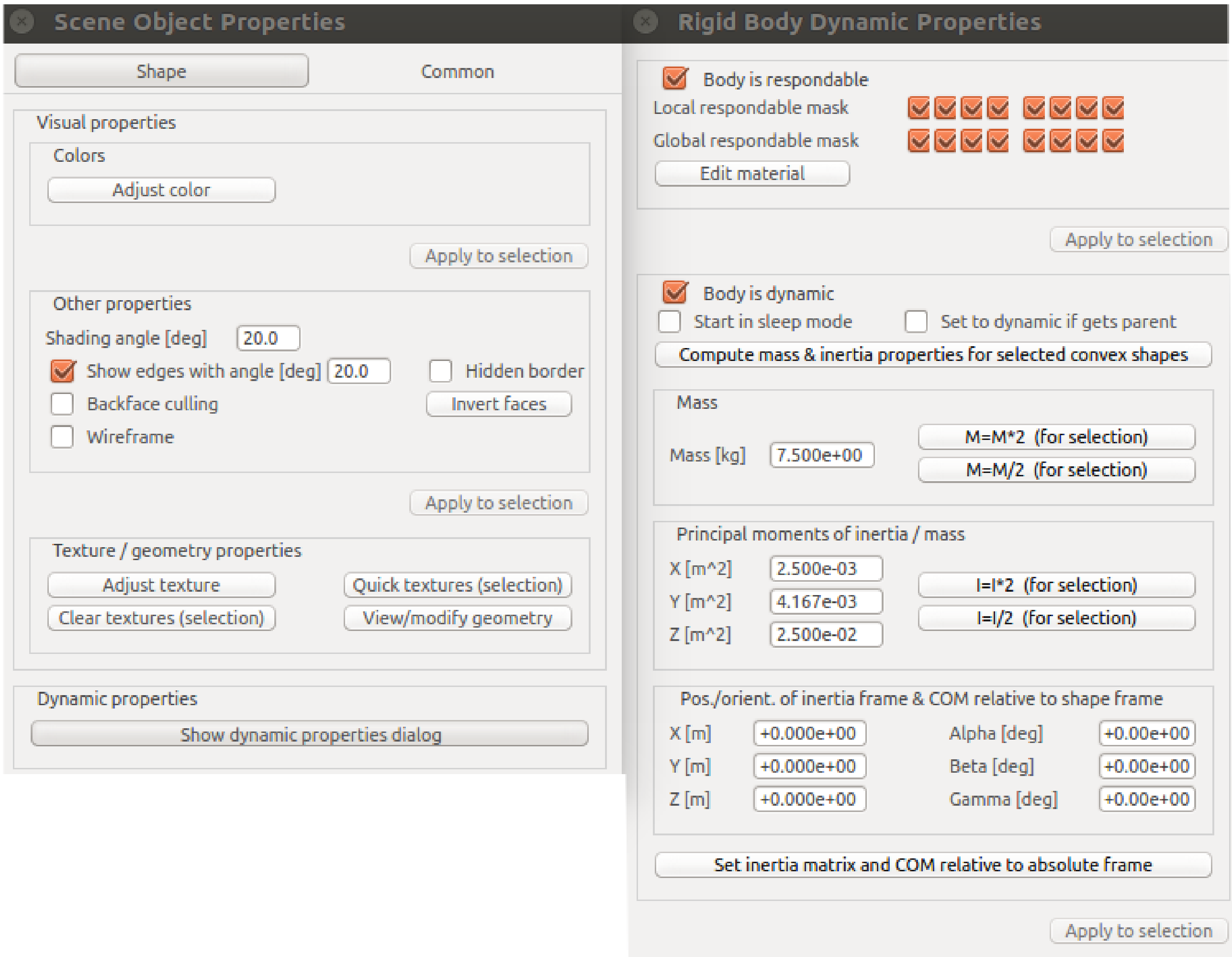

Abbildung 14: Body Dynamic Properties

Also, an important button in the *Scene Object Properties shape* tab is the *dynamic properties dialog* button. This will open the *Rigid Body Dynamic Properties*. In this example the body is *respondable* and is *dynamic*. The created *ground cuboid* called *myRobot* is the *base model*. As *base model*, this both options will be checked. Later there will be *link cuboids.* These *link cuboids* won't be *responsible* and *dynamic*.

The next step will be to create the first *link*. You could just repeat the task to add a *cuboid* or just copy the existing and modify its *size* and *properties*. Important is to rename it correctly this will help a lot later. In this example copy *myRobot* and rename the copied model to *myRobot_link1*. Open the *Geometry window* and set size *x* = 0.04, *y*= 0.06 and *z*=0.6. To move to *object* to desired place click the object/item shift button marked red in fig. 15.

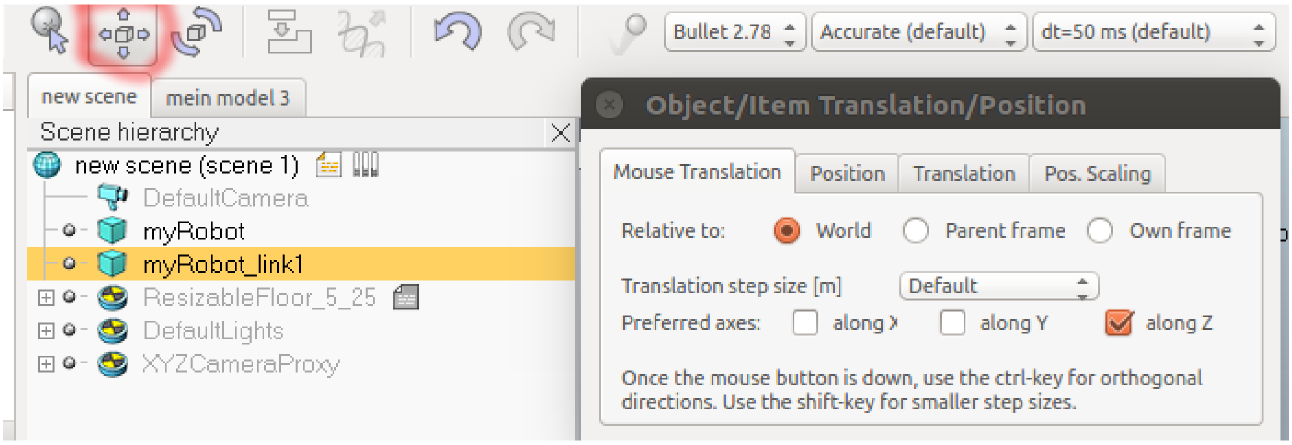

Abbildung 15: Object/Item Translation/Position

This opens the *object/item Translation/Position* window. This window allows to configure the preferred axes to move with the mouse. In fig. 15 it moves the *object along Z.*

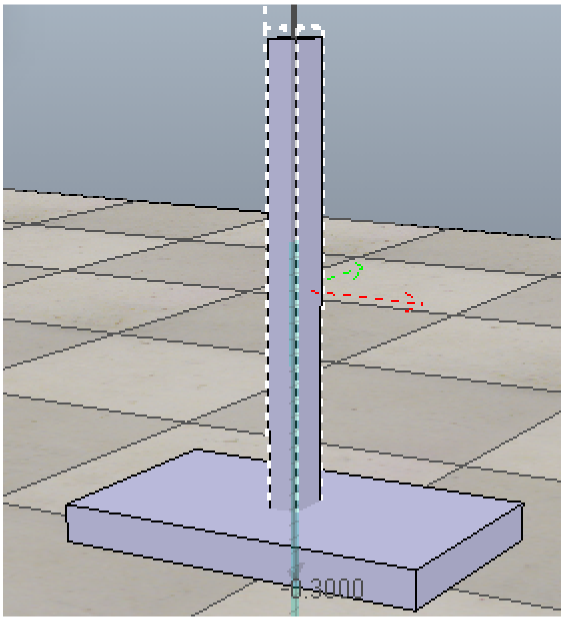

Abbildung 16: myRobot_link 1 movement

Click *myRobot_link1* to move and move it like fig. 16. The next step is to make the *object assemble* to get a true hierarchy. Ctrl-click *myRobot_link1* and *myRobot* then click the *assemble* button. Now *myRobot_link1* is part of *myRobot* both are linked. It is also possible to drag an *object* in the *scene hierarchy* to one other to *assemble* theme.

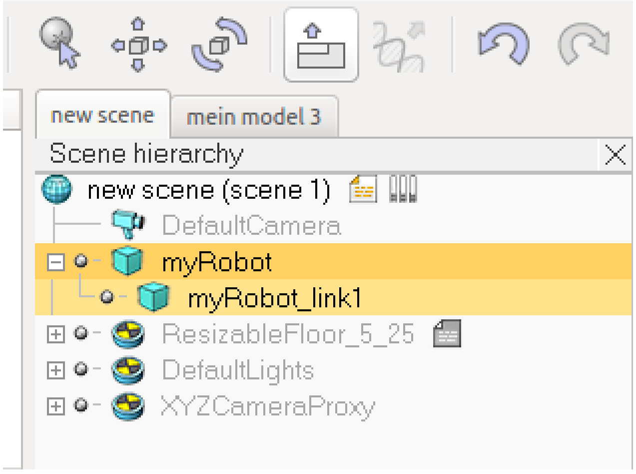

Abbildung 17: Assemble

Fig. 17 shows an object hierarchy. The a*ssemble* button is selected on top.

To make the link work, open the *Scene Object Properties* window and then open the *Body Dynamic Properties* window and uncheck *Body is respondable* and *Body is dynamic*. This must happen with all links. Copying this *cuboid* will pass all properties to the *copied cuboid*. So changing them now is most of time enough. It will just be copy paste only the *size* will be altered later.

The next step is to add the first *joint*. To add a *joint* go to [*Menu bar* --> *Add* --> *Joint* --> *Revolute*]. Add the *joint* and rename it to *myRobot_joint1*. Now move the *joint* to the top of the *myRobot_link1*. The *joint* needs to be *rotated*. The next step is to *rotate* the *joint*.

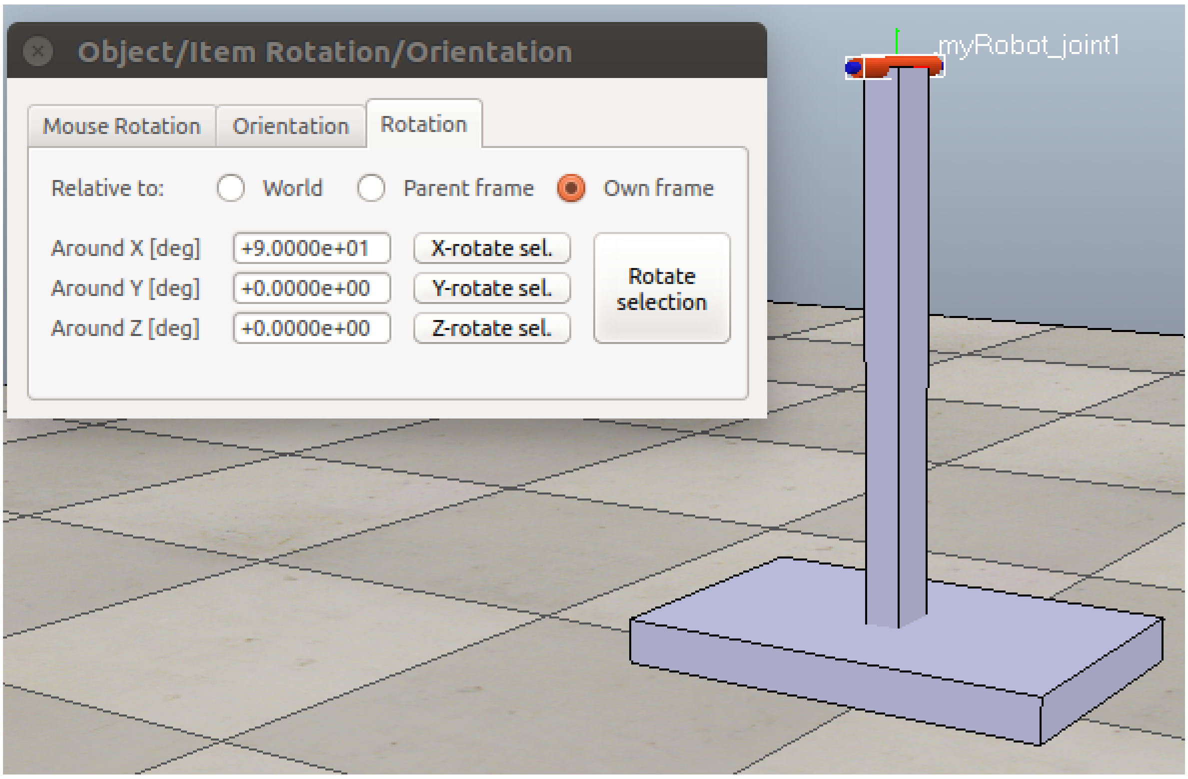

Abbildung 18: Rotation

Fig. 18 shows the rotation window. This window can be open by clicking the rotation button on top bar, it's the button right of the *Object/Item Translation/Position* button in fig. 15. It is similar to the *Object/Item Translation/Position window*. The difference is that it manages the rotation of the *object*. Rotation can be altered with the mouse or with the *rotation* tab. In the example *myRobot_joint1* is rotated around x with 90 one time in its *own frame*.

Joints do have their own S*cene Object Properties* window that can be opened with double clicking the symbol of the *joint* in *scene hierarchy*.

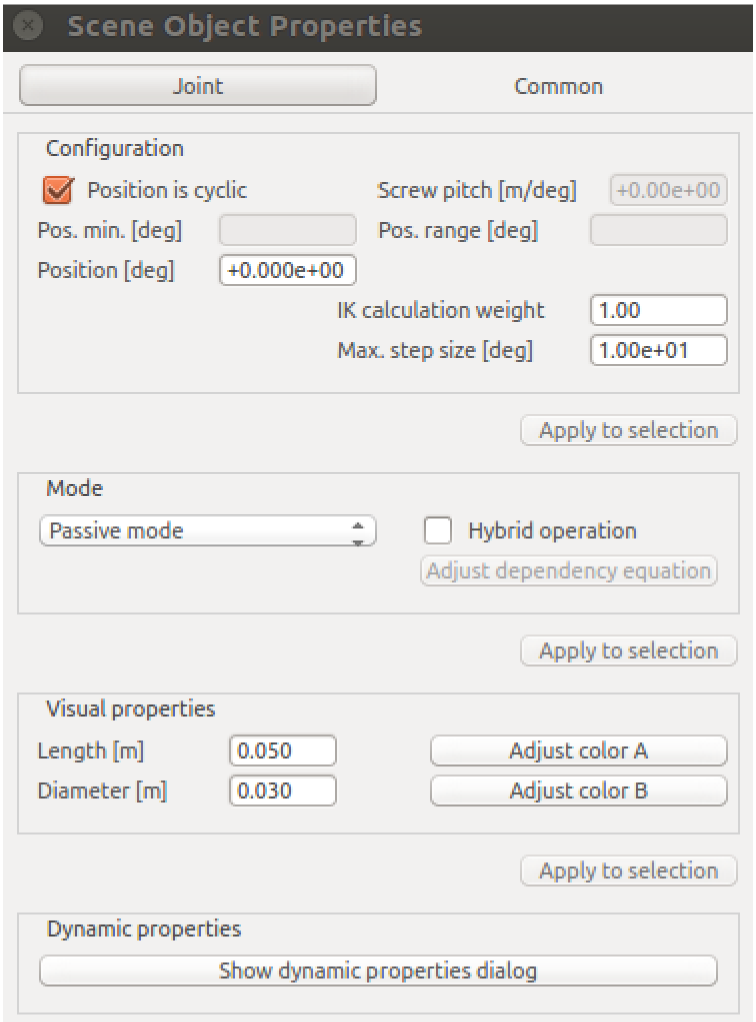

Abbildung 19: Scene Object Properties Joint

Fig. 19 shows the *Scene Object Properties* of *Joints*. This window allows to change the *length* and *diameter* of the *joint* also lets alter the *mode of joint*. In this example the *joints* are in *Passive mode* and the *length* is 0.05 and the *diameter* 0.03. Changing this once is enough. The *joints* will be copied and paste later on.

Now drag the *myRobot_joint1* to *myRobot_link1* to *assemble* it. Copy *myRobot_link1* and rotate it around y with 90. Change the size of *myRobot_link1* z=0.03 to fit it better. There should be now a *myRobot_link2* and a *myRobot_joint2*.

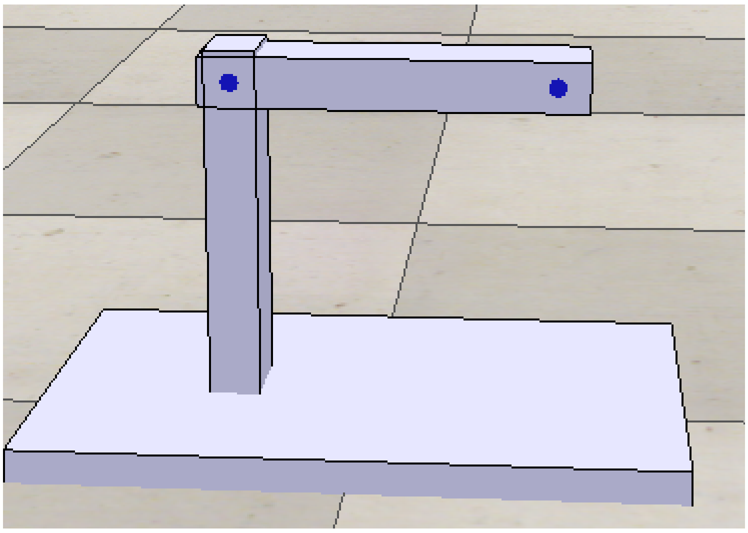

Abbildung 20: Robot with two links

Try to bring all the *links* and *joints* to the position as shown in fig. 20. Drag *myRobot_link2* to *myRobot_joint1* to *assemble* it. Copy *myRobot_link2* and paste it to create *myRobot_link3*. Remove *myRobot_joint3* it is not needed. Now change the size of the new created *link* to *x*= 0.04, *y*=0.06 and *z*= 0.09.

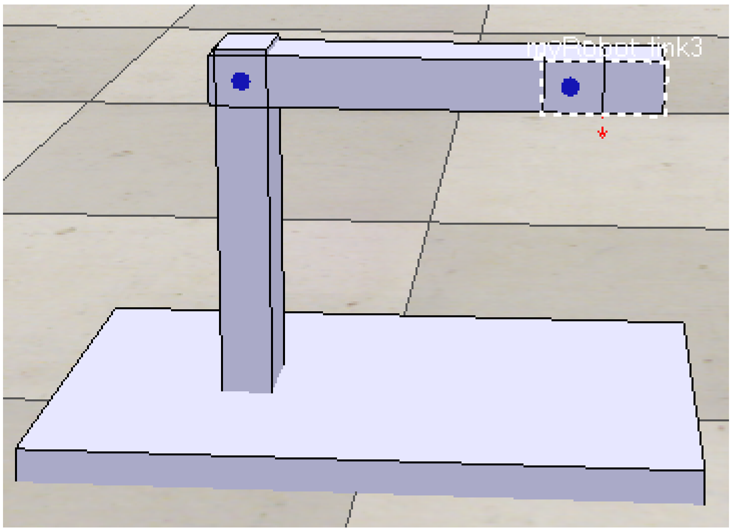

Abbildung 21: Robot with three links

Move *myRobot_link3* to the position as seen on fig. 21 and put it to *myRobot_joint2* to assemble it. Now it is possible to attach a *gripper* to the *model*. Go to *model browser* click *components* than go to *grippers*. Here it is possible to select *grippers*. Drag one into the *scene*. The *grippers* do have *attach-point objects*. Bring this *attach point object* to *myRobot_link3*. Ctrl-click the *attach point* object than Ctrl-click *myRobot_link3* . Open *Object/Item Translation/Position* window and go to position tab. Here is a button called *Apply to selection* by clicking this button the *object* will move to the selected *object*. Bring the *attach point* to the position where the *gripper* should be. Drag the *point object* to *myRobot_link3* to assemble it. Now repeat the process just select the *gripper model* first than the *attach point* and then click *Apply to Selection* . This should have brought the *gripper* to the *attach point*. Rotate the *gripper* to desired position. When , the position is correct, drag the *gripper object* in *scene hierarchy* to the attach point to assemble it.

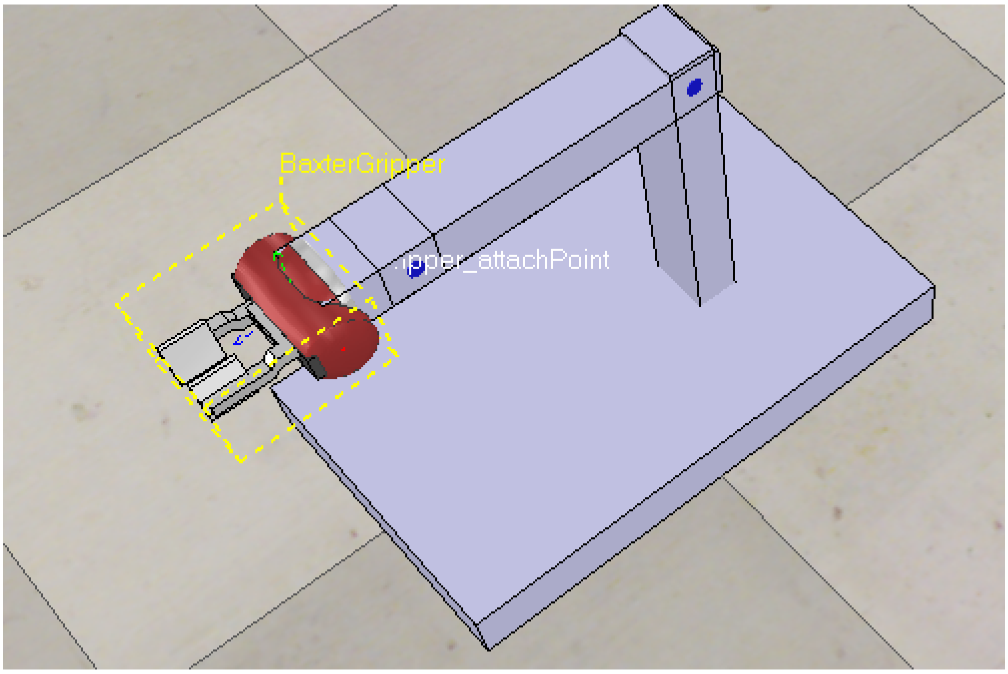

Abbildung 22: Robot with Gripper

Now the Robot *model* called *myRobot* which consist of three *cuboids* two *join*t and a *gripper* is finished .

The next step would be to give it a *child script* and enable it to use it on a *remote API Java* client. For example, you can create a *dummy object* and handle communication. Just create a *non-thread child script* and assign it to a added *Dummy Object*. Inside the code paste `SimExtRemoteAPIStart(19999).` This enables the communication on `port` number 19999.

Create a *new threaded child script*, uncheck the option *Execute just once* and assign it to the model base *myRobot*. It is needed to control the behavior of *myRobot*.

```
--Gets  Object handle and name of object to use them
--as variables
modelBase=simGetObjectHandle('myRobot')
modelBaseName=simGetObjectName(modelBase)

--Joint Handles for two joints
jointHandles={-1,-1}

for i=1,2,1 do
--Gets the handles of joints
    jointHandles[i]=simGetObjectHandle('myRobot_joint'..i)
end


--Functions
Slider_function1=function(inInts,inFloats,inStrings,inBuffer)
Slider_function2=function(inInts,inFloats,inStrings,inBuffer)
gripperclose=function(inInts,inFloats,inStrings,inBuffer)
gripperopen=function(inInts,inFloats,inStrings,inBuffer)
Pickcuboid=function(inInts,inFloats,inStrings,inBuffer)
-- Set-up some of the RML vectors:

vel=180
accel=40
jerk=80
currentVel={0,0}
currentAccel={0,0}
maxVel={vel*math.pi/180,vel*math.pi/180}
maxAccel={accel*math.pi/180,accel*math.pi/180}
maxJerk={jerk*math.pi/180,jerk*math.pi/180}
targetVel={0,0}

targetPos1={-14*math.pi/180,-80*math.pi/180}
simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,maxAccel,maxJerk,targetPos1,targetVel)
simSetIntegerSignal('BaxterGripper_close',1)
      --  simDisplayDialog(int1,inInts[1],sim_dlgstyle_ok,false)

--SimWait is important otherwise it will not move the robot properly
simWait(2)
```

Abbildung 23: Threaded child-script myRobot without functions

Fig. 23 shows how the *script* could be set to work properly. This *script* initializes everything. The *RML vectors* are set and *joint* and *object handles* are set up too. Important is that it has only two *joints*. Later there will be *two sliders* to control the *joints*. The variables have only two *joints* to control not four like *PhantomxPincher*.

Next step is to create the *function* for the movement. The *function* will be named like the ones of *PhantomXPincher*. They will use the function *simRMLMoveToJointPositions*. The function must be added after the *joint Handles* .

Fig. 5 is an example how it could be. The only difference is that the *targetpos1* has only two *InInts* instead of four . This model has only two *joints* .

Most of the *grippers* have their own *child script*. The *child script* of *grippers* are *non-threaded scripts*.

```
if (sim_call_type==sim_childscriptcall_actuation) then
    close=simGetIntegerSignal('BaxterGripper_close')

    if (close==1) then
        simSetJointTargetVelocity(motorHandle,0.005)
    else
        simSetJointTargetVelocity(motorHandle,-0.005)
    end
```
Abbildung 24: Gripper Script

As it can be seen in fig. 24 there is a variable *close.* This variable is the *result* of *function simGetIntegerSignal*(*BaxterGripper_Close*) The string *BaxterGripper_Close* is important and can be modified in the *myRobot* script.

If *close* has the value of *1* than it will use the function *simSetJointTargetVelocity* which just closes the *gripper* with a defined *velocity*. If *close* does not have the value *1* it will open the *gripper*.

```
gripperclose=function(inInts,inFloats,inStrings,inBuffer)
simSetIntegerSignal('BaxterGripper_close',1)
        return {},{},{'message was displayed'},'' -- return a string
    end
```

Abbildung 25: gripperclose myRobot

Fig. 25 shows *gripperclose* function of child script *myRobot*. Here in fig. 25 the function `simSetIntegerSignal` is used with the string from above. This function sets the signal of *BaxterGripper_close* to 1. This modifies the close variable of the *gripper* script from fig. 24 to 1. Value 1 lets the *gripper* close.

Its time to check the *Java* code. *PhantomXPincher* is a device. Create a similar code like class *PhantomXPincher*. Create a new class *myRobot* and extend class device . The only difference is that this robot has two *joints*. Change the `sliderValues` initialization to `sliderValues = new int[2]` instead of `int[4]`. Implement two `move slider methods` . It is almost the same code like that of class `PhantomXPincher`. It works the same way. `Main` class runs this `device` and creates the `sliders`.

It is possible to attach a *sensor* to the *robot* to detect *objects*. Go to [*Menu bar --> Add --> Proximity sensor --> Cone type*] . The *sensor* will be displayed in the scene . Change the name of the *sensor* to *myRobotProximity sensor* .Now click the *Proximity Senor* than Ctrl-click the sensor attach point of the gripper. *Open object item shift window* go to position tab and click *apply to selection*. No you can do that again with the *rotatio*n . Open the *rotation window* go to *orientation tab* and *click apply* to selection. Now the *sensor* is in the right place. Make the *sensor* part of the *gripper* by *assembling* it. By default the *sensor* is set to detect everything *detectable*. It is possible to change this and let it detect a specific *object* .

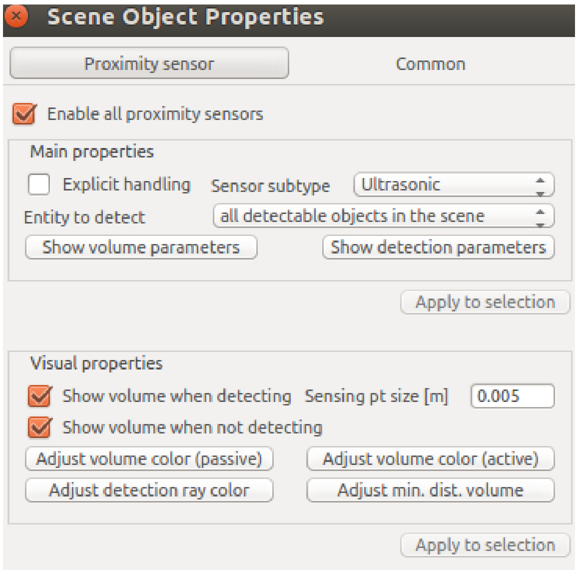

Abbildung 26: Scene Object Properties Proximity Sensor

Double click the *symbol* that represents the *sensor* in *scene hierarchy* to open the *Scene Object Properties* like in fig. 26. Here it is possible to change the *Entity to detect*.

Now it is possible to configure it like the sensor in `PhantomXPincher.` More details about setting up a `Proximity Sensor` are in the Read Proximity Sensor chapter .

This chapter showed how to create a model based on cuboid and control theme in *Java*. It is possible to create multiple *robots* and let theme operate something.

Multi-Robot Template with Conveyor Belt and own model

This *template* has the goal to show some functions which are done with *sensors* and *conveyor belts* interactions. In the industry there are always automated process with *conveyor belts* and *machines* doing different tasks. This template simulates a loop of bringing a *cuboid* from one *conveyor belt* to another with the help of *robots*. It is still possible to control the *robots* through Java like the `PhantomXPincher`. This *template* will show how to use *sensors* and their result to interact with *objects* and *robots*. This *template* is far from perfect. There are sometimes errors that let the *cuboid* drop or let the loop break. To get this better there must be more time spend. It is also important to calculate every size and position also every movement. This *template* can be seen as a possible way how something like a *conveyor belt* can be implemented.

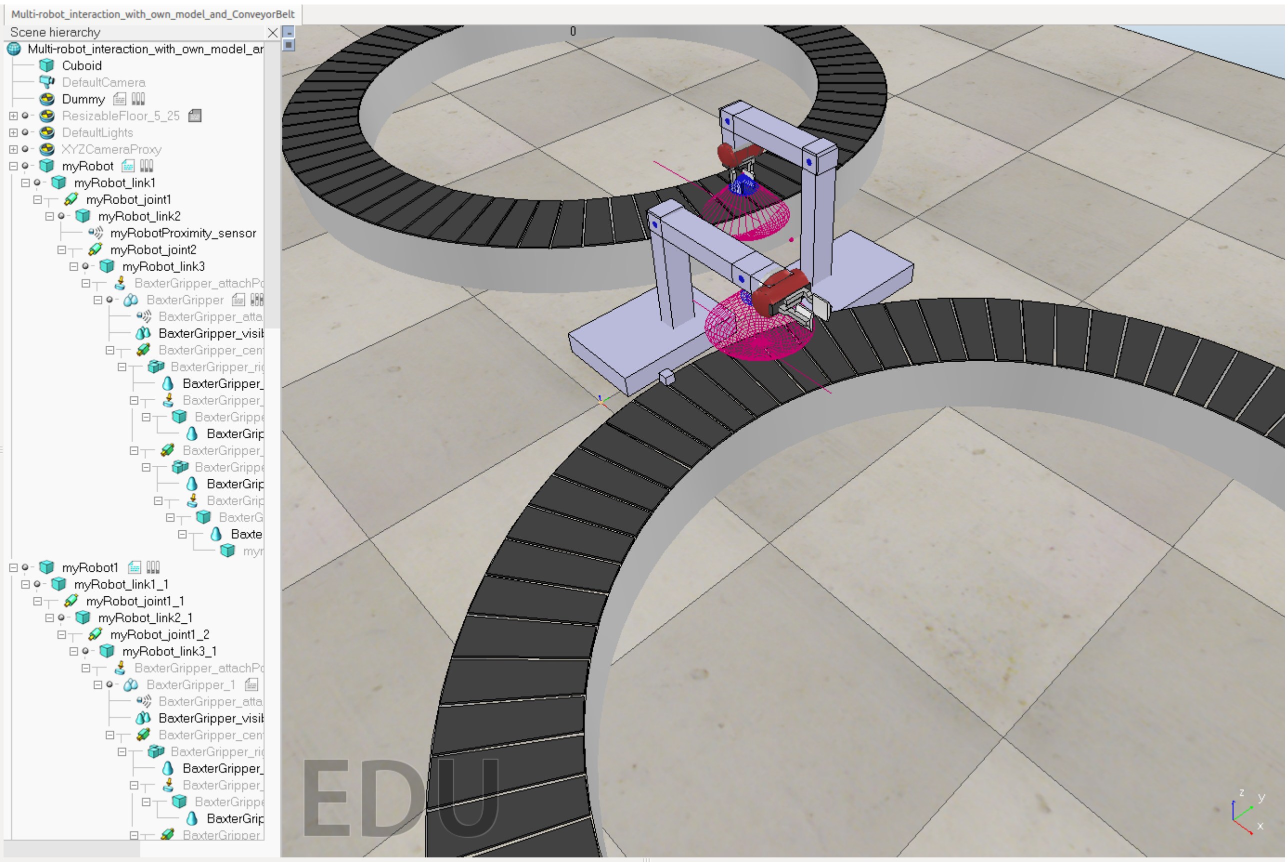

Abbildung 27: Template with Conveyor Belt

Fig. 27 shows a screenshot of the *template*. The connection to the *remote API* is established through the *script* of the *Dummy object*.

A *cuboid* can be seen on the *conveyor belt*. This cuboid will be brought to the first *robot* by the first *conveyor belt*. The first *robot* will transfer the *cuboid* to the second *conveyor belt*. The second *conveyor belt* will bring the *cuboid* to the second *robot*. This *robot* will then move it back to the first *conveyor belt*.

To establish this there are four *sensors*. Every *conveyor belt* and *robot* has its own *sensor*. This *sensors* are used to stop the *conveyor belts* and to bring the *robots* to move.

```
if (sim_call_type==sim_childscriptcall_initialization) then
    pathHandle=simGetObjectHandle("CircularConveyorPath1")
    simSetPathTargetNominalVelocity(pathHandle,0) -- for backward compatibility
    sensor=simGetObjectHandle('ConveyorBeltSensor1')
result,distance=simReadProximitySensor(sensor)


end

if (sim_call_type==sim_childscriptcall_actuation) then
    beltVelocity=-0.1


controll=simGetIntegerSignal('beltstop1')

if (simReadProximitySensor(sensor)>0 or controll == 1) then
        beltVelocity=0

end
```

Abbildung 28: Script of Conveyor Belt1

Fig. 28 shows the *script* of the first *conveyor belt*. This *conveyor belt* is a *model* which can be imported from the *model browser*. It can be found under *equipment → conveyor belts*.

The most important thing is to implement a *ray type proximity sensor*. This *sensor* can be placed like in fig. 27 somewhere over the *conveyor belt* and under the *robot*. This is the point where the *cuboid* must stop and be able to moved by the *robot*. The *ray type proximity sensor* will be part of the conveyor belt. The *robot* needs a *sensor* too. Here it is a *cone type sensor.* This *sensor* must be like in fig. 27. The *sensor* works as the "nose of the robot". It detects the *cuboid* that is in front of it. The *belt* stops when the *ray type sensor* gets touched and the *cone type sensor* is sensing the *object* that stopped in front of it. The *cone type sensor* lets the *robot* move while it is detecting something. The robots have "fake objects called *myrobotpart*" on the *grippers*. Those *objects* are not *dynamic*. The *sensors* can detect them. They can't be seen they don't collide. They have the purpose to simulate a lager area to the *ray sensors* of the *conveyor belt* to stop. Without the parts the process of grasping could be interrupted by the running *conveyor belt*. There were the case that while the *grippers* began grasping the *cuboid*, the *conveyor belt* began to move before the *grippers* could completely get the cuboid. These *objects* will eliminate this case.

In fig. 28 the script uses the *regular API function* `simreadProximitySensor` . The result will be saved in the *result* variable. When the *sensor* is detecting something it will return 1 else it will return 0 . There is a variable called *beltVelocity*. This variable has the value -0,1 . This will be changed with the following *if statement* in the script :

```
If  (simReadProximitySensor(sensor)>0) then
        beltVelocity=0
```

This *if statement* will bring the *conveyor beltVelocity* to 0 when a *object* is detected. The *conveyor belt* will stop.

The If statement can be extended . It is possible to control the *ConveyorBelt* through the *remote API*. The *Conveyor belts* are register as device in Java. They use the following function .

```
public boolean Stop1() {
            return api.simxSetIntegerSignal("beltstop1", 1);
      }
```

simxSetIntegerSignal is defined in the *API class*. This function creates a signal called *beltstop1* and gives it a 1 as value. This will be read from the object script in v-rep.

In fig 28. the script reads the *signal* and saves the value in a *variable* with the following line:

```
controll=simGetIntegerSignal('beltstop1')

if (simReadProximitySensor(sensor)>0 or controll == 1) then
        beltVelocity=0

end
```

With this line the *Conveyor belt* can be stopped though Java.

The second *robot* and *conveyor belt* work similar. The *conveyor belt* has a *ray sensor* that detects an *object* and stops the *conveyor belt*. The *robot* has a *cone type sensor* that senses the *cuboid* and begins with the movement.

Script Parameters can be set and used in *scripts*.

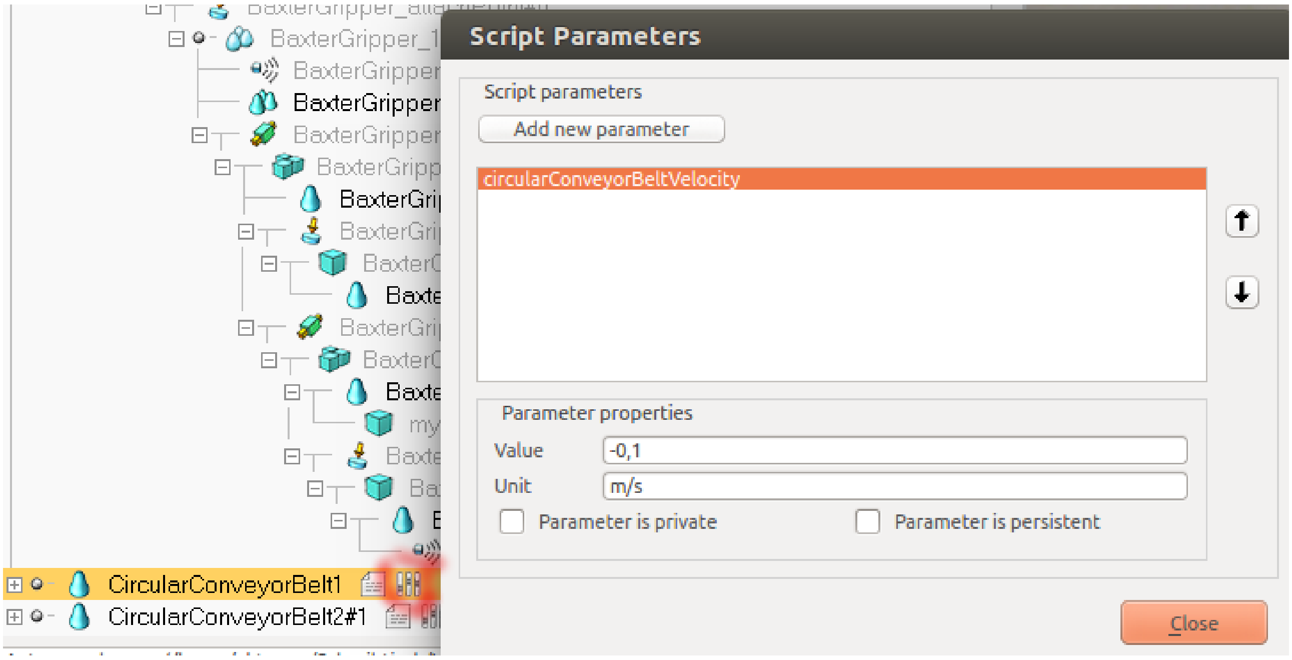

Abbildung 29: Script Parameter

Fig. 29 shows the *Script Parameters* window . Here it is possible to create a *variable* and give it a value. The red marked icon opens this window by clicking on it. Here in this fig. there is a *variable* called *circularConveyorBeltVelocity*. This *variable* has the value -0,1. This can be the value of the *beltVelocity*. The *conveyor belt* has this script parameter as a default value. The following lines demonstrate how to read the *script parameter*.

```
beltVelocity=simGetScriptSimulationParameter(sim_handle_self,"circ
ularConveyorBeltVelocity")
```

The *conveyor belt* in the *template* don't use *script parameters*. The value is given directly see fig 28.

To understand the movement of the *robots* it is important to look further more in the *script* of the *robots*.

```
--Gets  Object handle and name of object to use them
--as variables
modelBase=simGetObjectHandle('myRobot1')
modelBaseName=simGetObjectName(modelBase)
sensor=simGetObjectHandle('myRobot1Proximity_sensor')
result,distance=simReadProximitySensor(sensor)
```
Abbildung 30: myRobot 1 handles

Fig 30 shows the beginning of myRobot 1 script. Here the line `result,distance=simReadProximitySensor` lets return the result of the cone type sensor of the robot. This result will be saved as a variable called result .This will return either 1 or 0.

```
--if start == 1 then
-- Checks the sensor. If result = 1 than the object is detected else result will be 0
if result == 1 then
--Calling function Pickcuboid
res,err=xpcall(Pickcuboid,function(err) return debug.traceback(err) end)
if not res then
    simAddStatusbarMessage('Lua runtime error: '..err)
end

end
```

Abbildung 31: Calling Function Pickcuboid

Fig. 31 shows the *If statement* that checks the result of the *sensor*. If the *sensor* detects something it will call the *function pickcuboid* else it will do nothing.

```
Pickcuboid=function(inInts,inFloats,inStrings,inBuffer)
simWait(2)
targetPos1={-10*math.pi/180,-80*math.pi/180}
simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,maxAccel,maxJerk,targetPos1,targetVel)
simSetIntegerSignal('BaxterGripper_close_1',1)


simWait(7)

targetPos2={-10*math.pi/180,0*math.pi/180}
simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,maxAccel,maxJerk,targetPos2,targetVel)

simWait(2)

targetPos3={0*math.pi/180,0*math.pi/180}
simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,maxAccel,maxJerk,targetPos3,targetVel)

simWait(2)

targetPos4={180*math.pi/180,72*math.pi/180}
simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,maxAccel,maxJerk,targetPos4,targetVel)
simWait(2)

simSetIntegerSignal('BaxterGripper_close_1',0)

simWait(2)

targetPos5={60*math.pi/180,10*math.pi/180}
simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,maxAccel,maxJerk,targetPos5,targetVel)

targetPos6={0*math.pi/180,-90*math.pi/180}
simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,maxAccel,maxJerk,targetPos6,targetVel)
```

Abbildung 32: my Robot1 Function Pickcuboid

Fig. 32 shows the *function pickcuboid*. Here The *functions* `simRMLMoveToJointsPositions`, `simSetIntegerSignal` and `simWait` are used. The values and the moves must be calculated carefully. This *function* grasp the *cuboid* and moves it to the *conveyor belt*. `simWait` lets the simulation wait for a given time. `simWait` seems not to work through remote API function `simxCallScritpt` in Java . It does work when it is called from the script itself .The other *functions* were explained in the previous chapters.

Java architecture

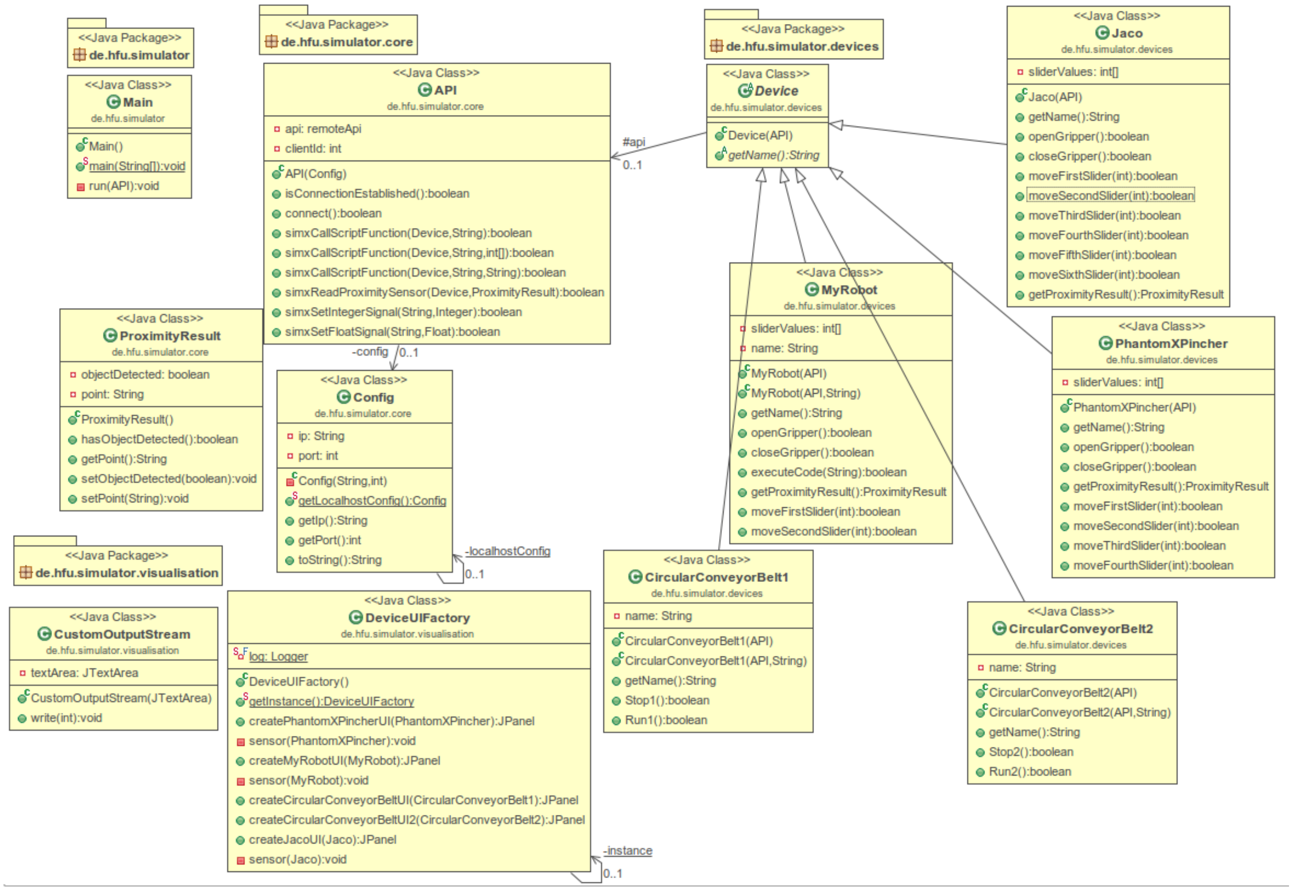

Abbildung 33: UML Diagram

Fig. 33 is a UML diagram of the Project. This UML diagram shows the Java architecture.

This Project has five *packages*. The one *package* that is not displayed in the diagram is the remote API *package* called coppelia. The other four packages are ***de.hfu.simulator***, ***de.hfu.simulator.core***, ***de.hfu.simulator.devices*** and ***de.hfu.simulator.visualisation***. These *packages* do have some *classes* that do several tasks. The ***de.hfu.simulator*** package does only have the *Main Class* that runs everything. The package ***de.hfu.simulator.core*** does have three classes. The *Class API* has all the used *remote API methods* defined in a class. It uses the given *Classes* of the coppelia *package*. The *Class Config* does have the *IP* and *Port* information to establish a connection. The *Class API* can use this information. The last Class in this *package* is called *ProximityResult*. It is a *class* that deals with the *proximity sensor results*. The package ***de.hfu.simulator.devices*** does have a Class called *device*. In this *package* several *devices* can be added. All added *devices* should inherit from the *device* class. Here are some classes like *MyRobot* , *PhantomXPincher* … this are all *devices* that inherit from the class *device*. These devices all use the API config defined in the *API Class*. The last package ***de.hfu.simulator.visualisation*** has classes that create the *UI* . This *package* has a *DeviceUIFactory Class* that creates UI *panels* for several *devices*. And a Class called *CustomOutputStream* that is used to display the *console output* in a GUI as a *text area*.

GUI

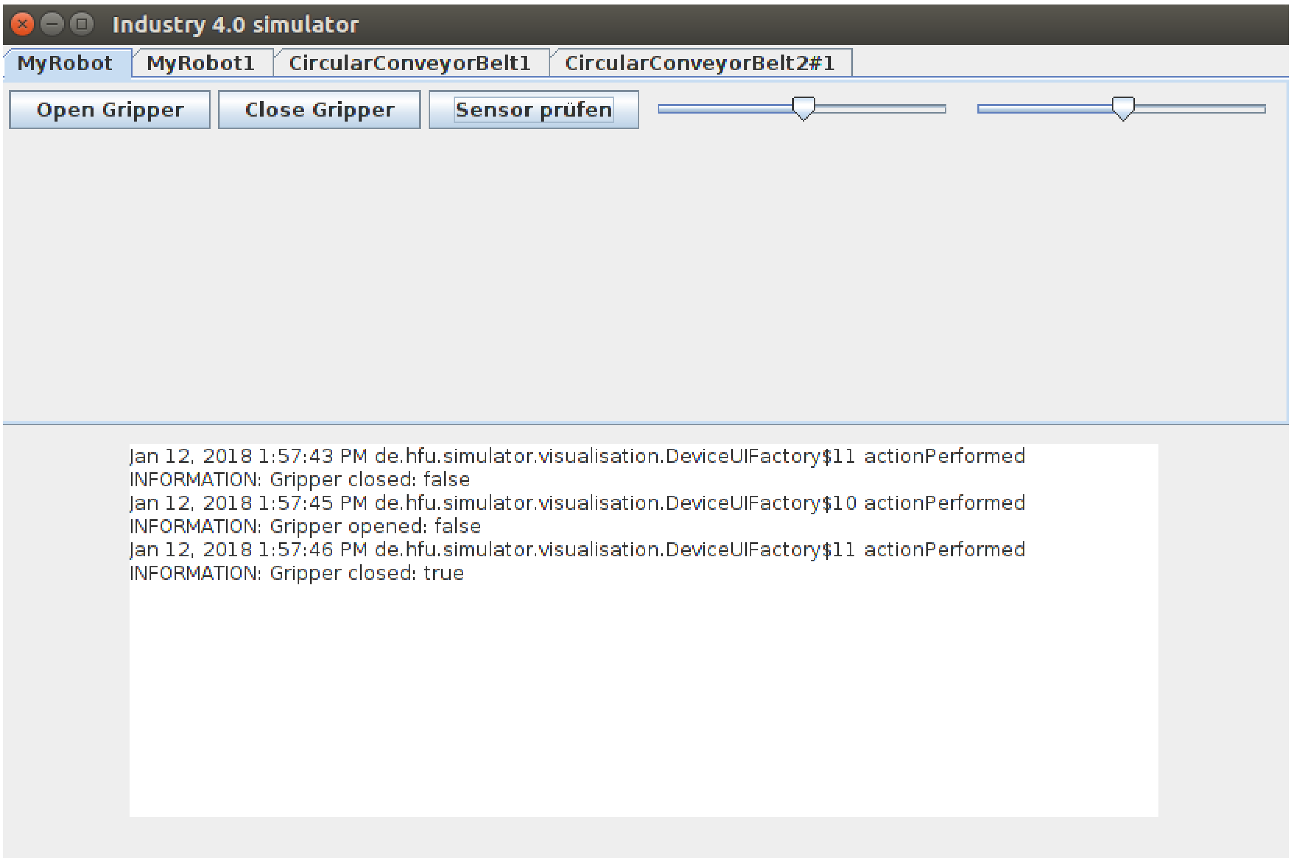

Abbildung 34: GUI of simulator

Fig 34 show the *GUI* of the simulator. It is *frame* with *tabbed panels*. All the *panels* are created at *DeviceUIFactory* Class. The *panels* have *device* specific buttons and sliders. The *CircularConveyorBelt* panels do have a *Stop button* and a *Run button*. The *sliders* will move the *robots*. Under the *panels* there is a *text area* displaying the *console outputs*.

Closing Remarks

This Simulator aims the goal to simulate a realistic environment of an Industry 4.0 working place. This project tried to show possible ways of how to create a simulation of realistic Industry 4.0 environment. This Simulator can be used to illustrate specific use cases in an industry 4.0 working place. V-REP is a very powerful software. There are more functions explained in the User Manual of V-REP. There are different supported files formatted that allow the user to import self created robot models. Alongside the given examples there are many more functions like path planning and milling. There are several other ways of creating and using joints. Joints do have different modes.

This Project just showed how to begin with V-REP and build first successful cases using some tools of V-REP. V-REP can do many other things. This Project can be seen as a starting example.

V-REP also has guides and beginner tutorials how to use the Remote API. Some guides and programs can be found in the downloaded files of V-REP.

V-REP does also have a forum where people can ask questions. It can be found under http://www.forum.coppeliarobotics.com/ .

List of References

[1] Coppelia Robotics , , *V-REP Downloads* [Online] Available: http://www.coppeliarobotics.com/downloads.html (Accessed: 17.11.2017)

[2] Coppelia Robotics , "V-REP API framework" ," Remote API", *V-REP User Manual* [Online] Available: http://www.coppeliarobotics.com/helpFiles/en/remoteApiOverview.htm (Accessed: 17.11.2017)

[3] Coppelia Robotics , "V-REP API framework" , "Remote AP"I, "Enabling the Remote API - client side", *V-REP User Manual* [Online] Available: http://www.coppeliarobotics.com/helpFiles/en/remoteApiClientSide.htm (Accessed: 17.11.2017)

[4] Coppelia Robotics , "Entities", "Shapes", "Import and export", *V-REP User Manual* [Online] Available: www.coppeliarobotics.com/helpFiles/en/importExport.htm (Accessed: 17.11.2017)

[5] Coppelia Robotics , "V-REP API framework" , "Remote API", "Remote API functions (Java)"," simxStart", *V-REP User Manual* [Online] Available: http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsJava.htm#simxStart (Accessed: 17.11.2017)

[6] Coppelia Robotics , "V-REP API framework" ," Regular API", " Regular API function" ,"simRMLMoveToJointPositions", *V-REP User Manual* [Online] Available: http://www.coppeliarobotics.com/helpFiles/en/regularApi/simRMLMoveToJointPositions.htm (Accessed: 17.11.2017)

[7] Coppelia Robotics , "V-REP API framework" , "Remote API" , "Remote API functions (Java)", " simxCallScriptFunction", *V-REP User Manual* [Online] Available: http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsJava.htm#simxCallScriptFunction (Accessed: 17.11.2017)

[8] Coppelia Robotics , "V-REP API framework" , "Regular API", "Regular API function", "simSetIntegerSignal", *V-REP User Manual* [Online] Available: http://www.coppeliarobotics.com/helpFiles/en/regularApi/simSetIntegerSignal.htm (Accessed: 17.11.2017)

[9] Coppelia Robotics , "V-REP API framework" , "Regular API", "Regular API function", "simGetObjectHandle", *V-REP User Manual* [Online] Available: http://www.coppeliarobotics.com/helpFiles/en/apiFunctions.htm#simGetObjectHandle (Accessed: 17.11.2017)

[10] Coppelia Robotics , "V-REP API framework" , "Regular API"," Regular API function"," simGetObjectName", *V-REP User Manual* [Online] Available: http://www.coppeliarobotics.com/helpFiles/en/regularApi/simGetObjectName.htm (Accessed: 17.11.2017)

[11] Coppelia Robotics , "V-REP API framework" , " Remote API" , "Remote API functions (Java)" ," simxReadProximitySensor", *V-REP User Manual* [Online] Available: http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsJava.htm#simxReadProximitySensor (Accessed: 17.11.2017)

[12] Coppelia Robotics , "V-REP API framework" ," Remote API", "Remote API functions (Java)", "simxGetObjectHandle", *V-REP User Manual* [Online] Available: http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsJava.htm#simxGetObjectHandle (Accessed: 17.11.2017)