INTERNATIONAL HELLENIC UNIVERSITY
SCHOOL OF ENGINEERING
DEPARTMENT OF INFORMATICS, COMPUTER AND TELECOMMUNICATIONS ENGINEERING

Robotic Arm

Color sorting objects

Guide with steps in the CoppeliaSim simulator

Project team:

Anastasiades Alkinoos (20003)

Zina Eleni (20046)

Supervisor:

Koureas Argyrios

SERRES, 2024

Contents

Chapter 1

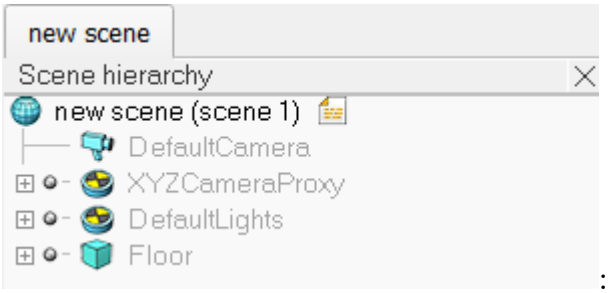
Introduction

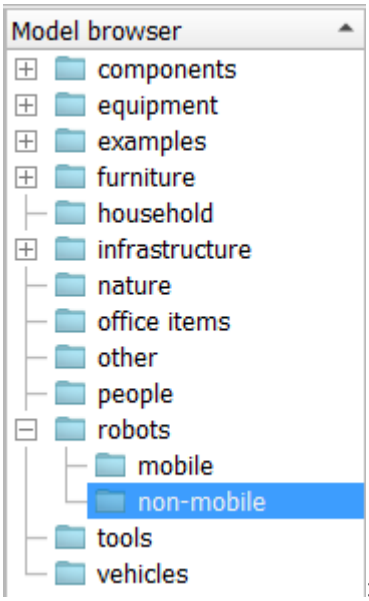
The robotic arm is an important invention in the field of robotics. With the ability to mimic the movements of the human hand, it can perform a variety of tasks accurately and efficiently. One of the examples of its use is sorting objects by color. Using CoppeliaSim software (v4.6.0 rev .18), we can simulate and control the robotic arm to automatically sort objects by color. In this project, we will learn how this process works and how it can be implemented in a simple and understandable way.

User manual for CoppeliaSim

Menu:

Description of menu options:

-  : camera movement to the surface (camera pan)
-  : camera rotation (camera rotate)
-  : zoom in on selected object (fit to view)
-  : movement of the selected object (object/item shift)
-  : selected object rotation (object/item rotate)
-  : start/resume – suspend – stop simulation
- 

: scene hierarchy – this is where all added components will be displayed
- 

: list of components (model browser)

Chapter 2

Adding components

To add the components we will need, we take them from the model browser and drag them into the simulation area. The model browser has many subcategories of things we can add, such as robots (with/without movement), tables, helicopters, etc. Specifically, the simulation will require the following:

ABB IRB 140 robotic arm:

This robotic arm is "made" specifically for the purpose of this project, so we can easily and quickly manage its movements with a few commands. Obviously, the robotic arm has the main role in the simulation to sort the boxes into the bins to be added. We find it through the model browser, by navigating to **Model browser>robots>non-mobile** and by pressing it, we drag it into the simulation area.

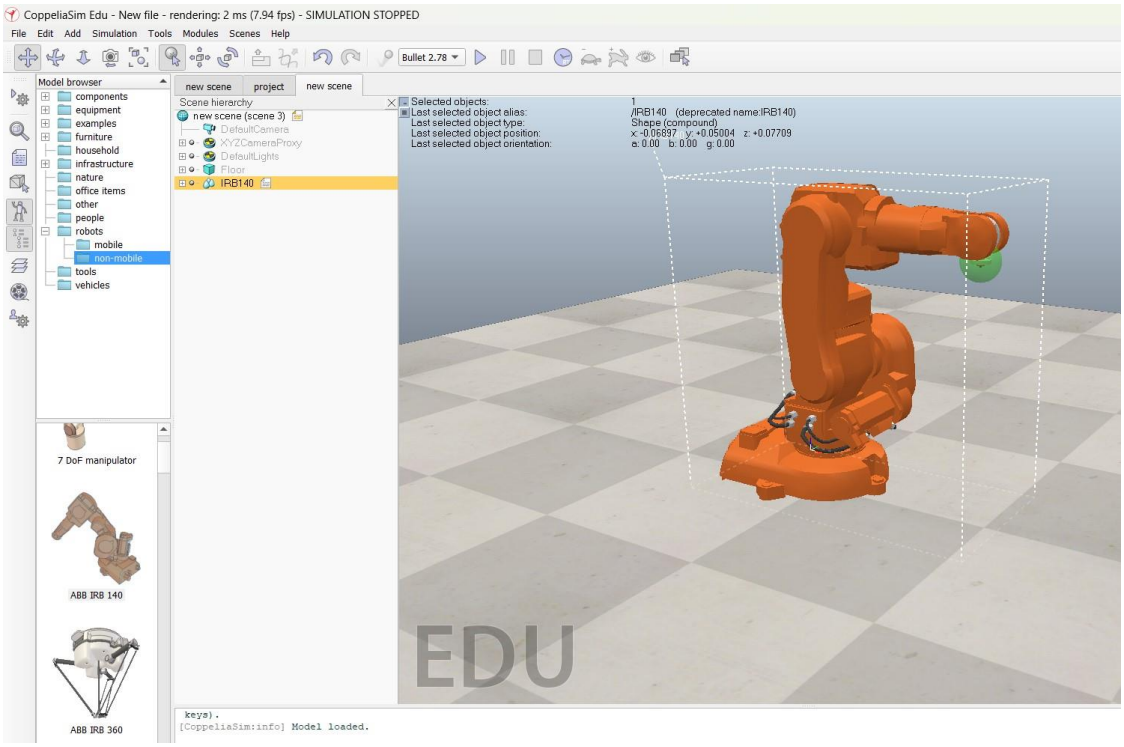


Figure 1 Robotic arm placement

The images shown are indicative for guidance of the project in terms of their coordinates (x, y, z), because this will matter later. The coordinates of the robotic arm are **(x=-0.34397, y=+0.77504, z=+0.07709)**. These values can be changed easily with the "object/item shift" button in the "Position" tab with "World" selected.

Conveyor belt:

The conveyor belt is the one that will bring the boxes to the robotic arm in a suitable place to pick them up. The way the belt is positioned is important, as the robotic arm must have the freedom of movement to grip the box effectively. We find it through the model browser, navigate to **Model browser>equipment>conveyors** and select "**generic conveyor (belt)**». After attaching the conveyor belt, rotate it 180 degrees, having it selected and pressing the "fit to view" button mentioned in Chapter 1. This rotation is necessary for the boxes to come from the opposite direction to the robotic arm.

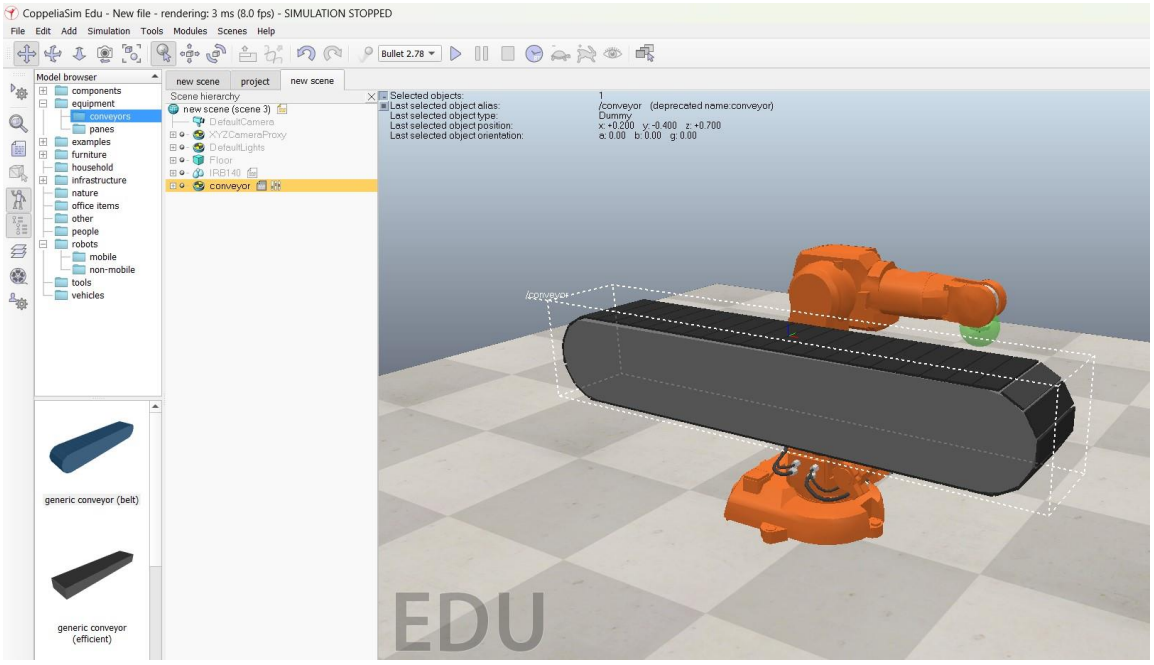


Figure 2 Conveyor belt placement

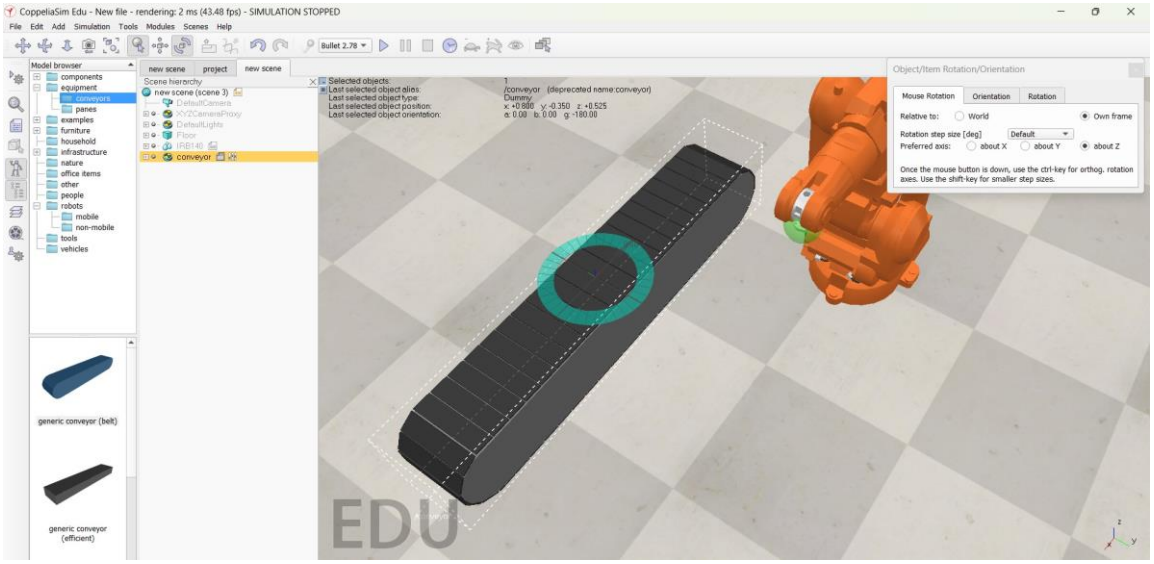


Figure 3 Conveyor belt rotation

The conveyor belt coordinates are **(x=+0.600, y=+0.425, z=+0.525)**. Another way to rotate it is with the "object/item rotate" button in the "Orientation" tab with "World" selected in "Gamma [deg]" equal to +180.00 and the remaining 0.00.

Proximity sensor – color detection sensor:

The proximity sensor is the one that will detect the color from each box and inform the robotic arm in which bin to place it. How the color detection sensor is positioned is important, as it must be directly above the conveyor belt and cover its entire length to have full detection accuracy. The sensor cannot be added from the model browser and we add it by pressing right click on the simulation area and select **Add>Proximity Sensor>Ray type**.

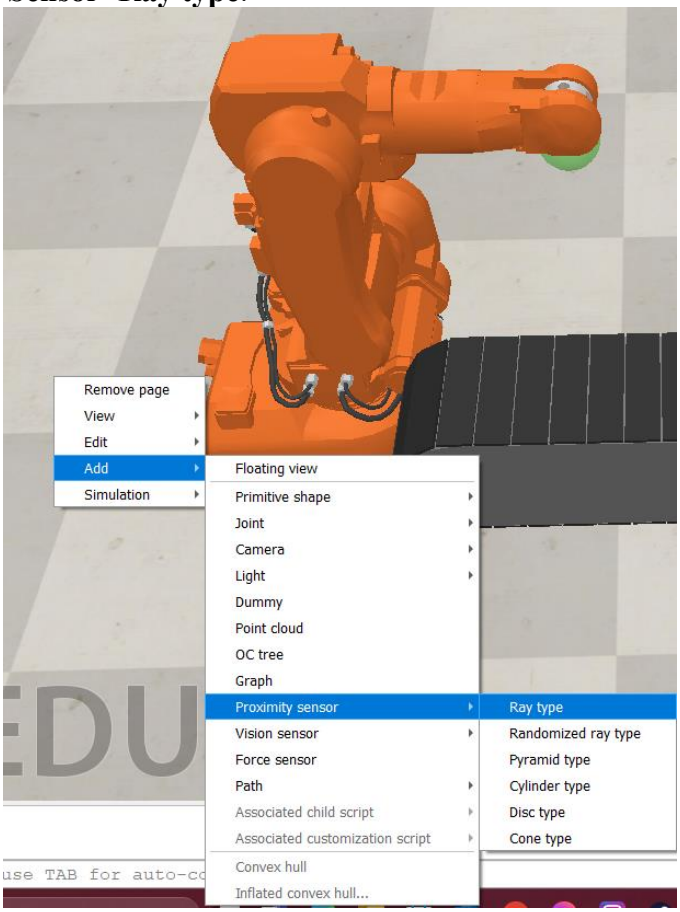


Figure 4 Adding a proximity sensor

Once the sensor is created, we will select it from the "Scene hierarchy" and move it with the "object/item shift" button first under the belt to a point where the box can be stopped in time for the robotic arm to access.

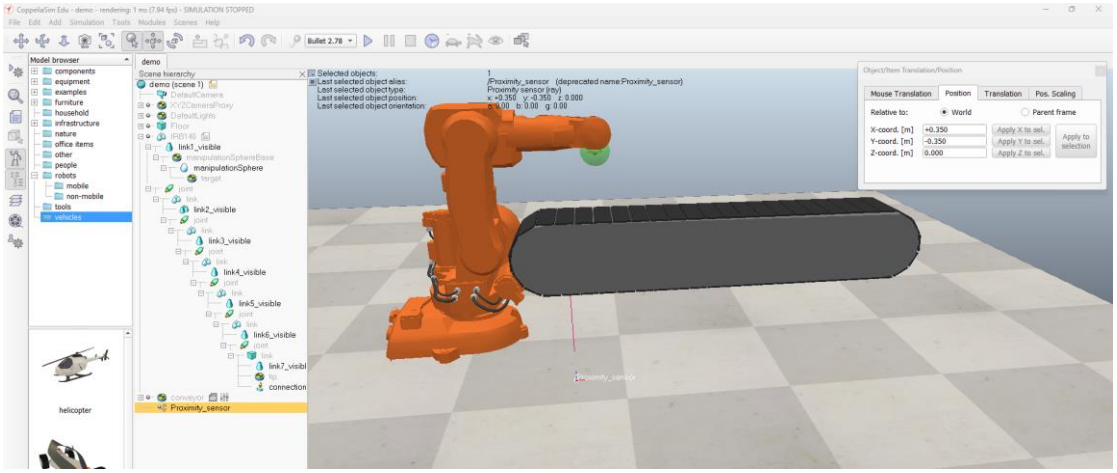


Figure 5 Placement of the proximity sensor under the conveyor belt

Then we move it over the belt with the same button, by pressing the "Mouse Translation" tab and having only "Own frame" and "along Z" selected.

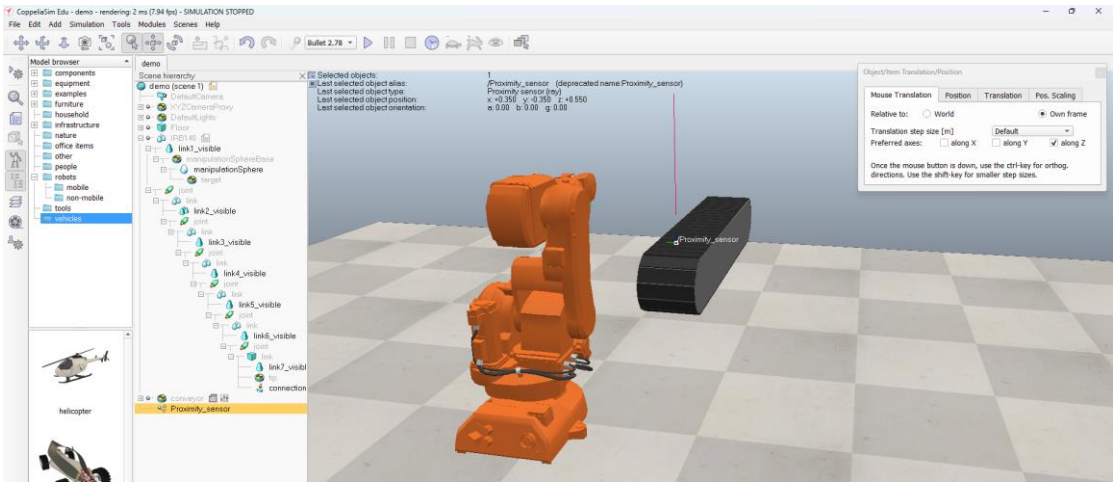


Figure 6 Placement of the proximity sensor on the conveyor belt

It is noticed that the sensor consists of a circular object, from which we grasp the sensor, and a straight line, which is essentially the sensor itself, that is, from which that color detection is made. For this reason, we adjust the position of the sensor (only selected "along X", "along Y"), so that the circular object is outside the belt and inside the straight line covering the surface of the belt.

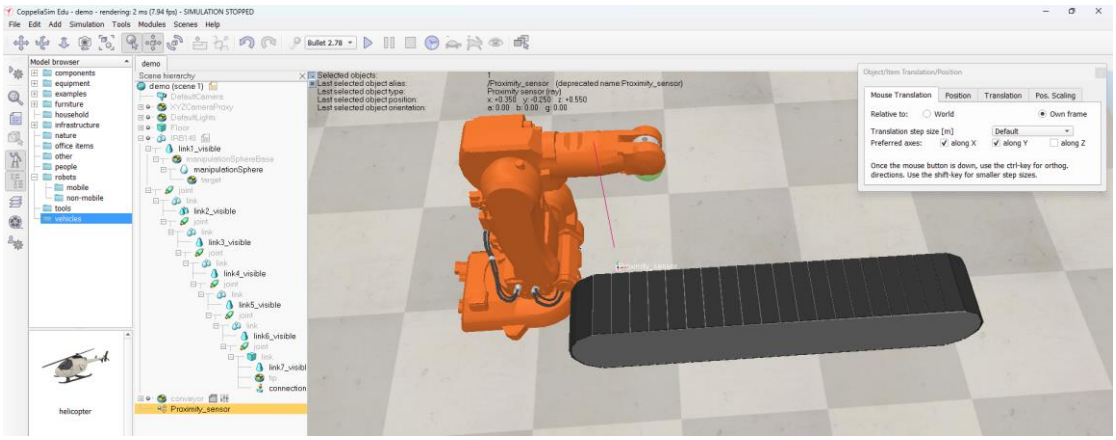


Figure 7 Adjusting proximity sensor position

The way the sensor is positioned it cannot detect anything, because it "looks" at the air and not at the belt. To achieve this, we must rotate it with the "object/item rotate" button, pressing the "Orientation" tab with "World" selected in "Alpha [deg]" equal to +90.00 and the remaining 0.00.

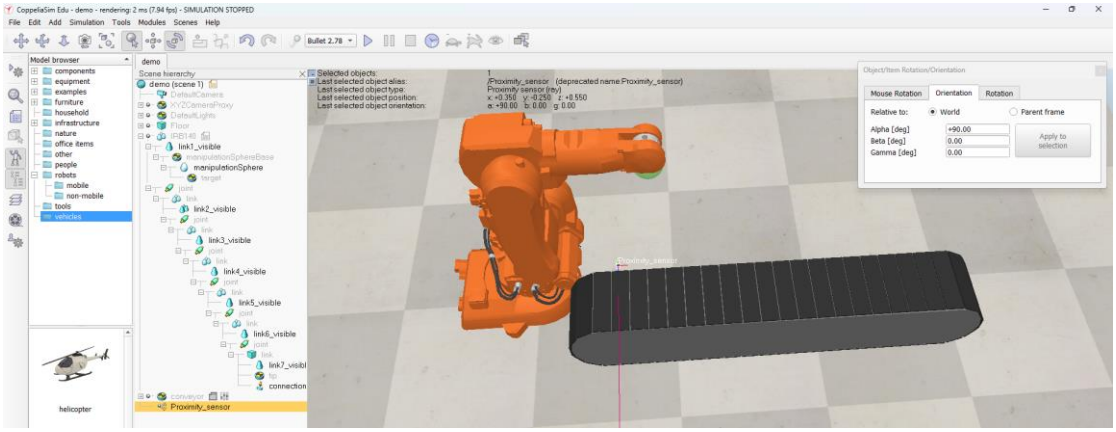


Figure 8 Adjusting proximity sensor direction

Finally, the straight line of the sensor must cover the entire length of the belt and for this reason we double click on the "Proximity_sensor" icon in the "Scene hierarchy" to see its properties. Press "Volume parameters" and set "Offset" equal to 0 and "Range" equal to 0.20 which is ideal.

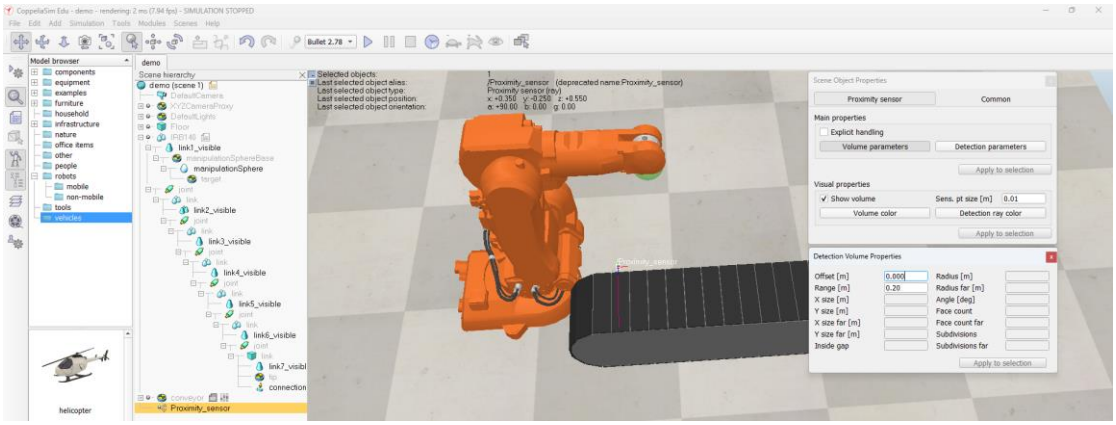


Figure 9 Adjusting proximity sensor range

The coordinates of the proximity sensor are **(x=+0.175, y=+0.525, z=+0.550)**.

Another step that needs to be taken for a purpose that will be analyzed in a later chapter, is to add a virtual "Dummy" object to put the sensor inside. This will help us write code in Lua programming language, to inform the robotic arm about the color of the box. We add "Dummy" in a similar way, as we did with the sensor, i.e. right click and **Add>Dummy**.

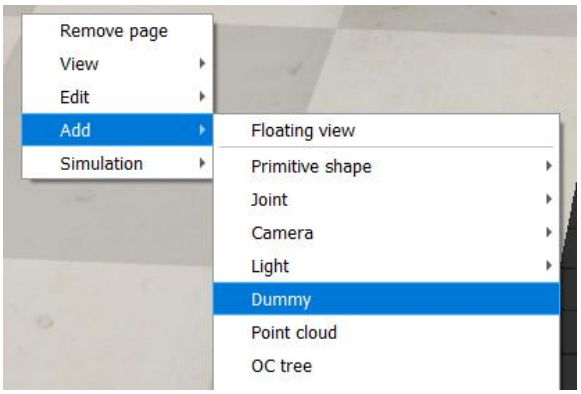


Figure 10 Adding a virtual object

Then, from the "Scene hierarchy", drag the "Proximity_sensor" into "Dummy" and right-click on "Dummy" and press **Add>Associated child script>Threaded>Lua** to write the code later. By double-clicking on "Dummy" as a word, we change its name to "conveyor_sensor".

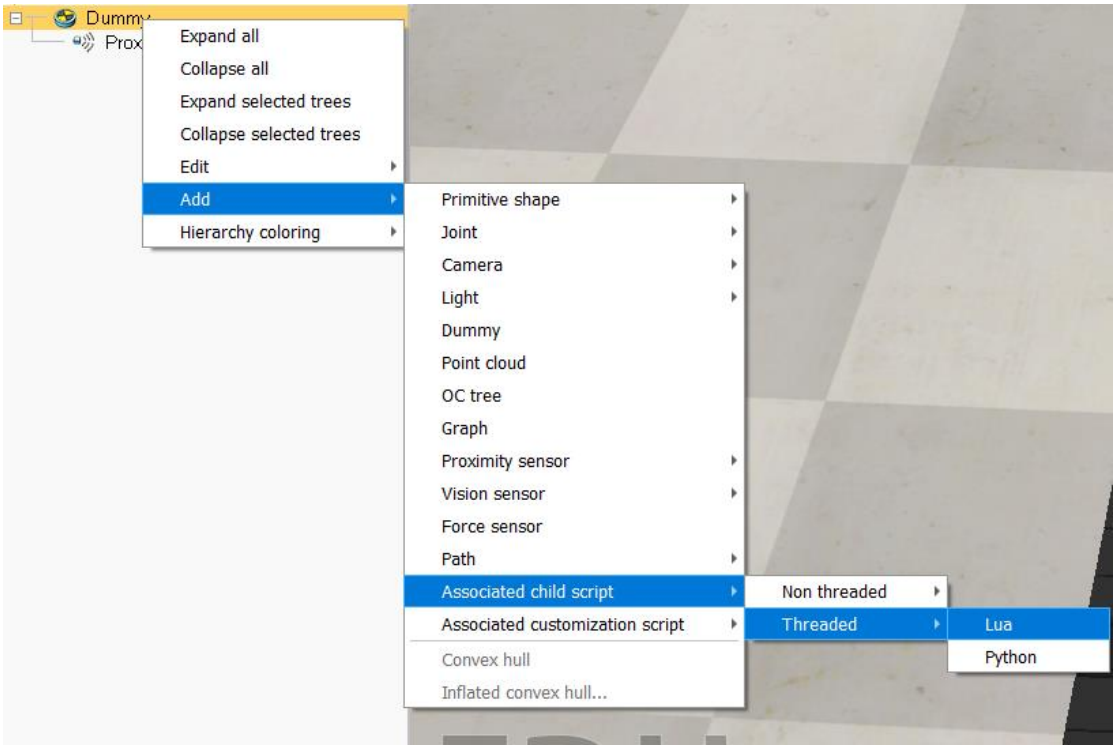


Figure 11 Customizing virtual object

Colored bins:

The colored bins are the ones in which the boxes will be stored depending on their color. The bins used are not included in the model browser, as they have been created manually and have a .stl file format that must be downloaded from this link: [2]. To import this file, we press on the File **>Import>Mesh menu...** , select it and then "Import".

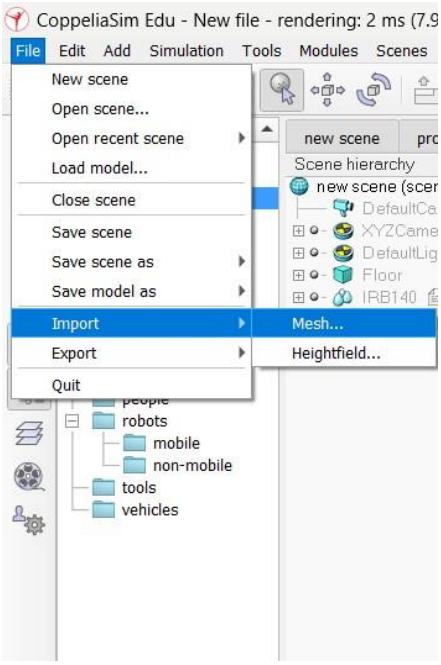


Figure 12 Importing bin file (1)

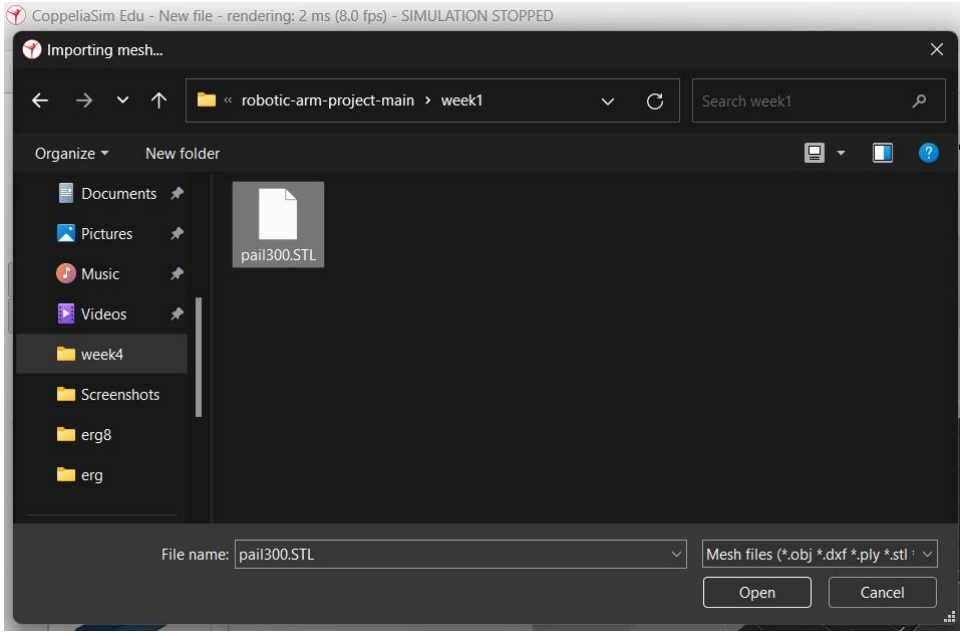


Figure 13 Importing bin file (2)

Once imported into the program, it should appear, as in *Figure 14*. As mentioned, the bins must be in front of the robotic arm, so we move respectively with the "object/item shift" button and rotate with the "object/item rotate" button, so that they are vertical.

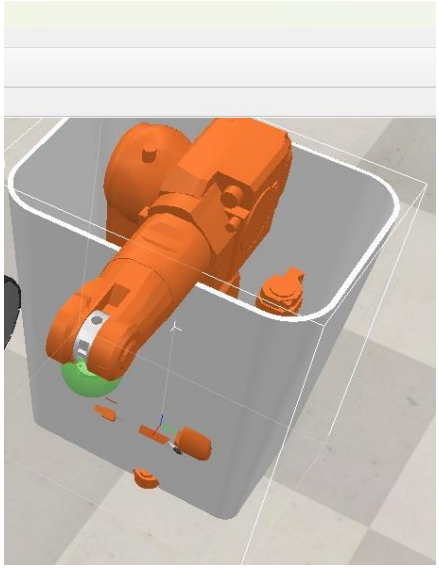


Figure 14 Displaying bin

The coordinates of the bin are **(x=+0.35, y=+0.725, z=0.00)**. Another way to rotate it is with the "object/item rotate" button in the "Orientation" tab with "World" selected in "Gamma [deg]" equal to -90.00 and the remaining 0.00.

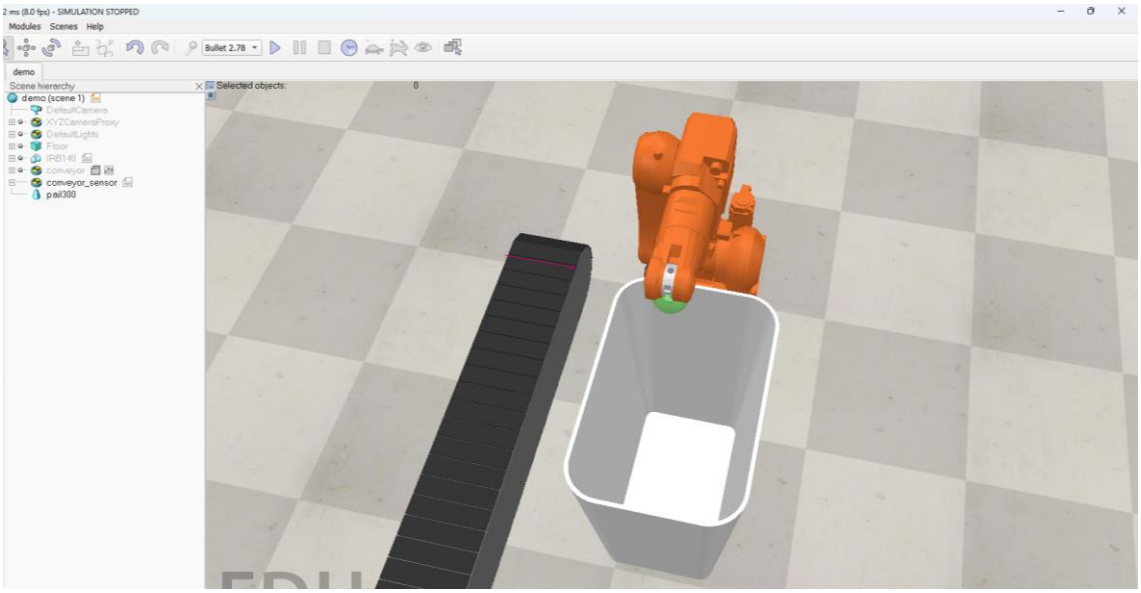


Figure 15 Moving bin

A total of 3 bins will be placed, red, green and blue. At this point, we need to set particular settings before adding color, such as size. By double-clicking on the bin icon from "Scene hierarchy" (pail300), its properties are shown. Thus, we define two settings:

1. **Dynamic properties dialog>Body is respondable (this checked only)**
2. **Geometry>X [m]=0.31165, Y [m]=0.45897, Z [m]=0.50>Apply (values are set based on your needs)**

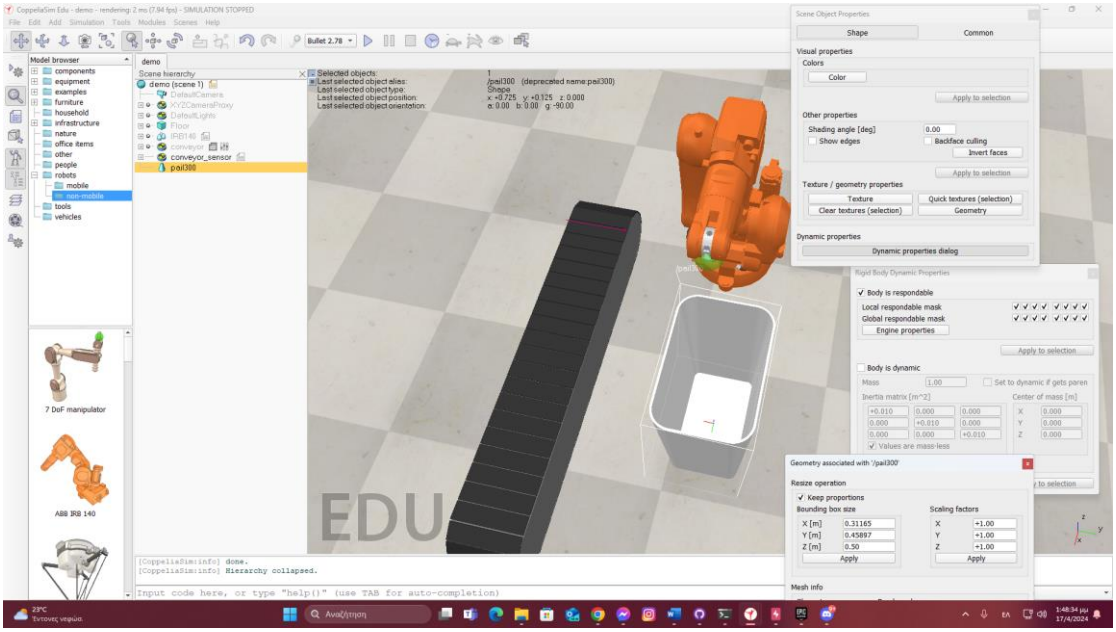
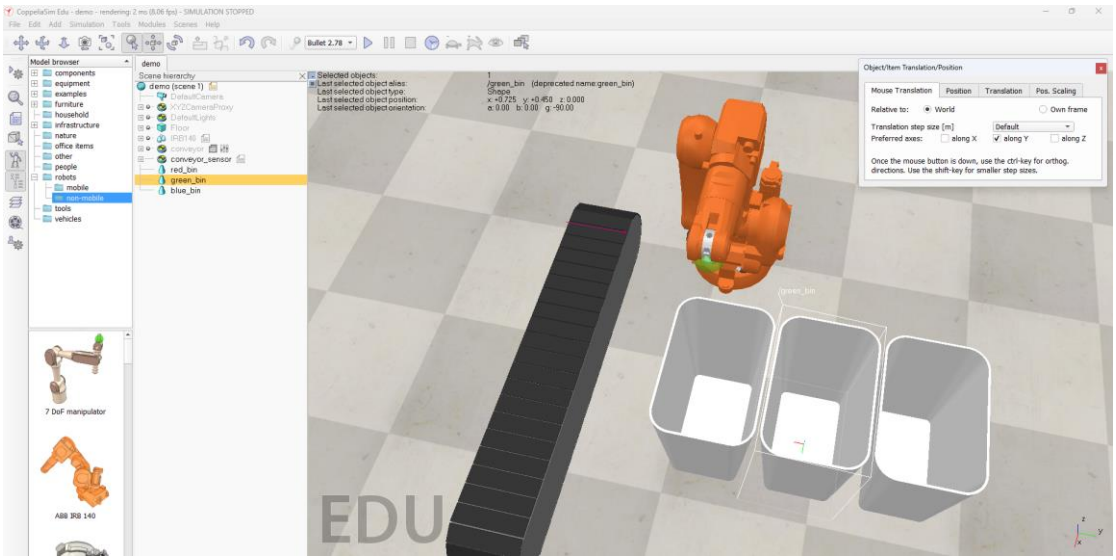


Figure 16 Bin settings

Once the above settings have been applied, we can copy the same bin with these settings to create the other two. We do Ctrl+C and Ctrl+V the pail300 inside the "Scene hierarchy" and at the same time we rename each bin with the first bin we added being the red one.

Figure 17 Bin names

We move each bin except the red bin (red_bin) by step 0.325 only relative to Y. For example, if the coordinates of the red bin are **(x=+0.35, y=+0.725, z=0.00)**, then the green bin will be **(x=+0.35, y=+1.050, z=0.00)** and so on.



Εικόνα 18 Moving bins

Finally, each bin must have the color depending on its name. To set the first bin to red, double-click on the "red_bin" icon and **Color>Ambient diffuse component>Red=1.00, Green=0.00, Blue=0.00**, do so respectively for the other bins.

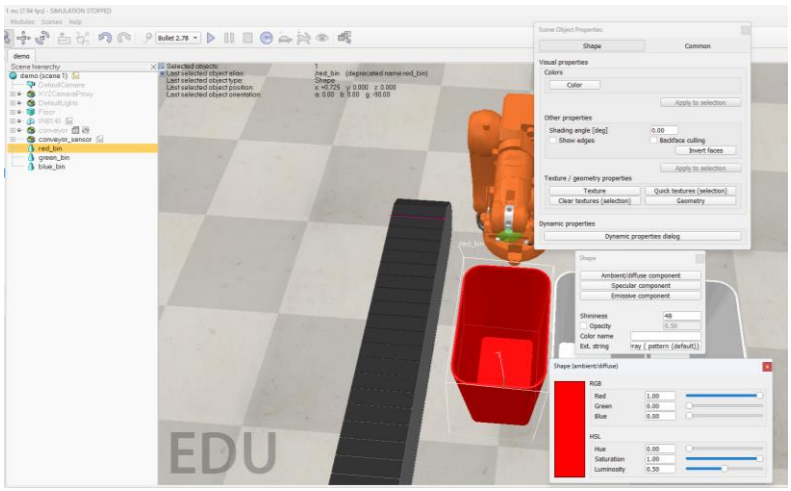


Figure 19 Bin coloring

Another thing we need to do to have the bins gathered in the "Scene hierarchy" is to add a simple "Dummy", by clicking **Add>Dummy**, call it "bins" and drag the bins into it.

Figure 20 Bin collection

Baxter Vacuum Cup - suction cup:

The suction cup we will use is to supposedly "stick" the boxes when the robotic arm takes them. In a simulation environment, we don't care, as in reality, if it catches the boxes correctly from above or from the side, etc. Add the suction cup through the model browser, navigate to **the Model browser>components>grippers** and select the "**Baxter vacuum cup**".

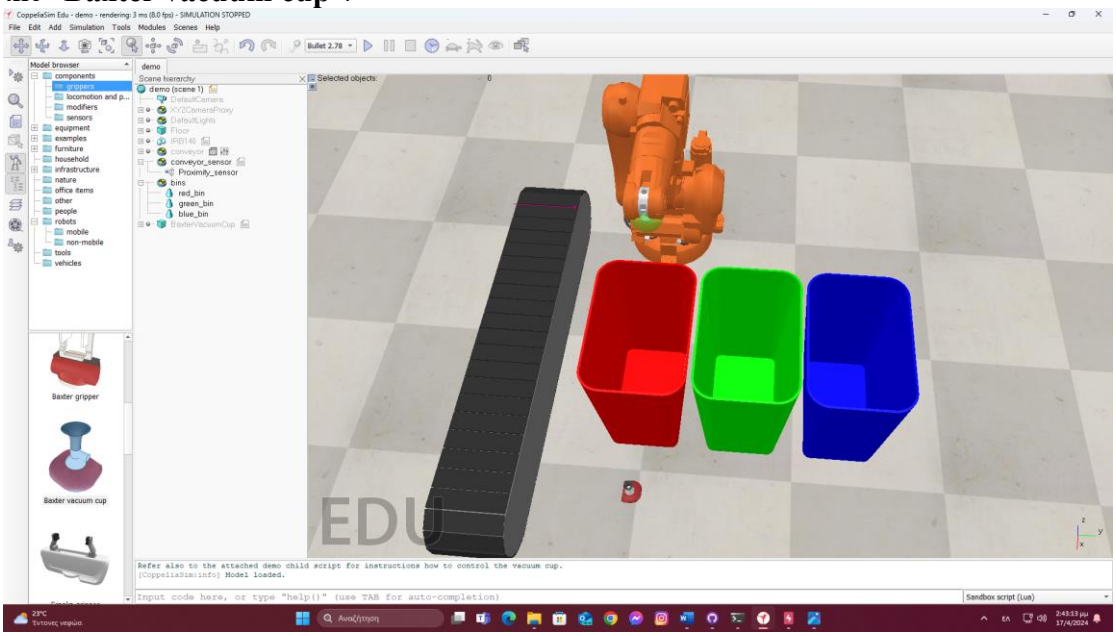
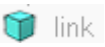
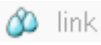


Figure 21 Adding the suction cup

Initially, the suction cup must be incorporated into the tip from which the robotic arm will "grasp". **This is a very crucial point and it is recommended to first do Ctrl+S to save your changes.** Steps:

1. **Moving the target to manipulationSphereBase and deleting manipulationSphere.**
2. **Deleting link7_visible and connection which contains**  **.**
3. **Moving the BaxterVacuumCup to link6_visible and moving link6_visible to the last**  **.**

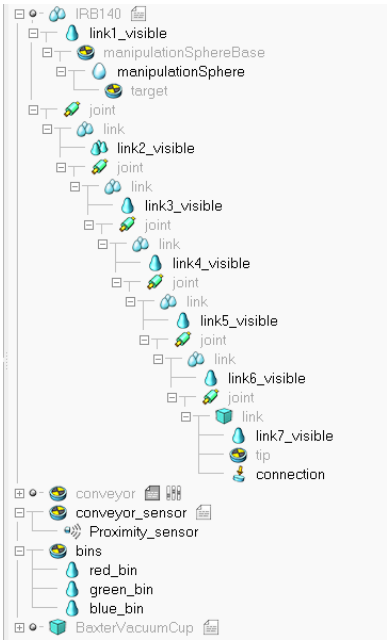


Figure 22 Before settings

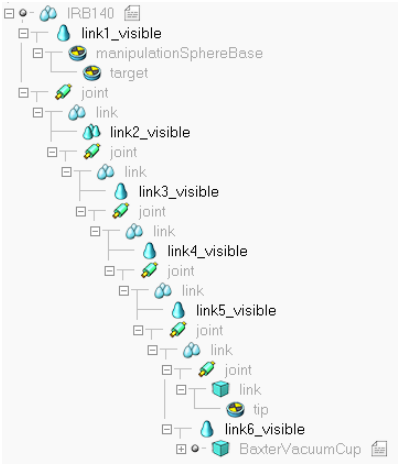


Figure 23 After settings

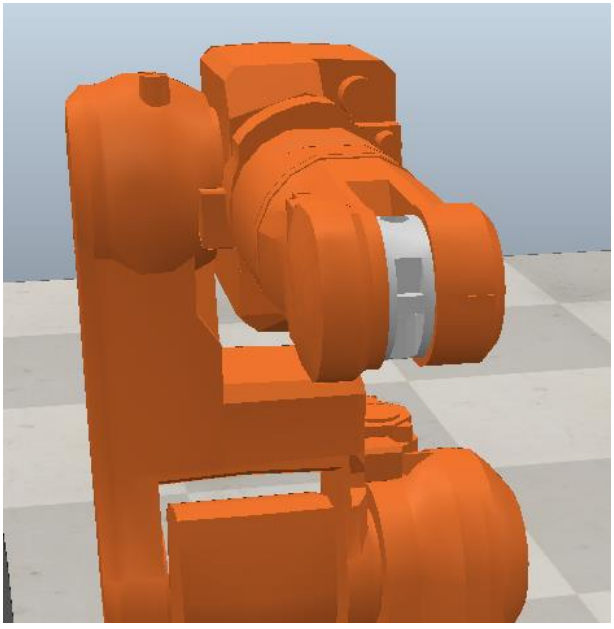


Figure 24 Settings result

Finally, we must place the suction cup on the end of the robotic arm. The suction cup must be moved over the middle point where there is a gap in the previous image with the black part of the suction cup facing outwards. In order to direct it to go to the "link6_visible" that we implemented it, it must be all 0.00 with the "object/item shift" button in the Position tab with selected "Parent frame".

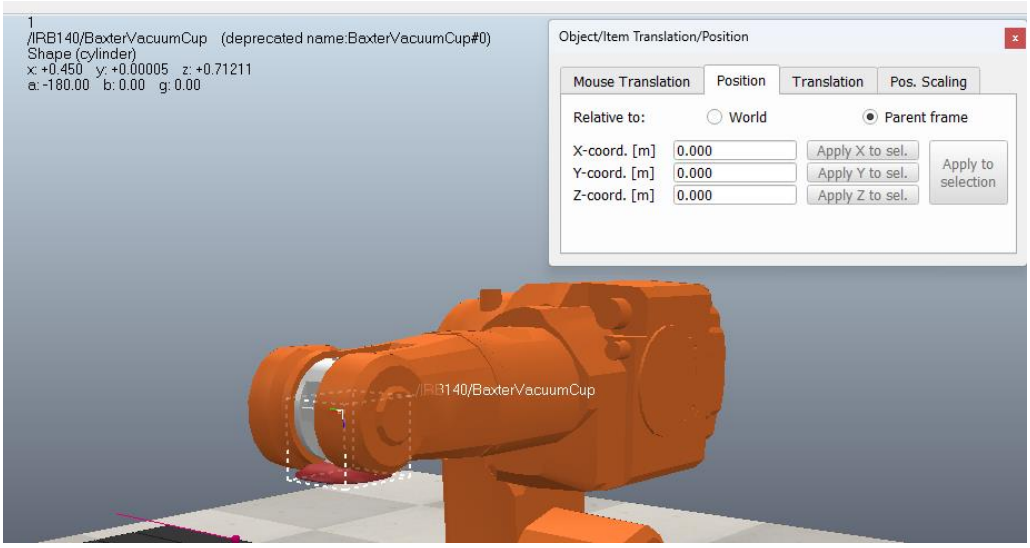


Figure 25 Moving the suction cup

The suction cup coordinates are **(x=+0.275, y=+0.77505, z=+0.71211)** with "World" selected and rotated ("object/item rotate" button) in the "Orientation" tab with "World" selected, "Alpha [deg]" equal to 0.00, "Beta [deg]" equal to -90.00 and "Gamma [deg]" equal to +180.00. Move as much as necessary with the X-axis to pull the edge of the suction cup out and look like this. The final step is to uncheck "Body is dynamic" from the properties of the suction cup so that it does not fall.

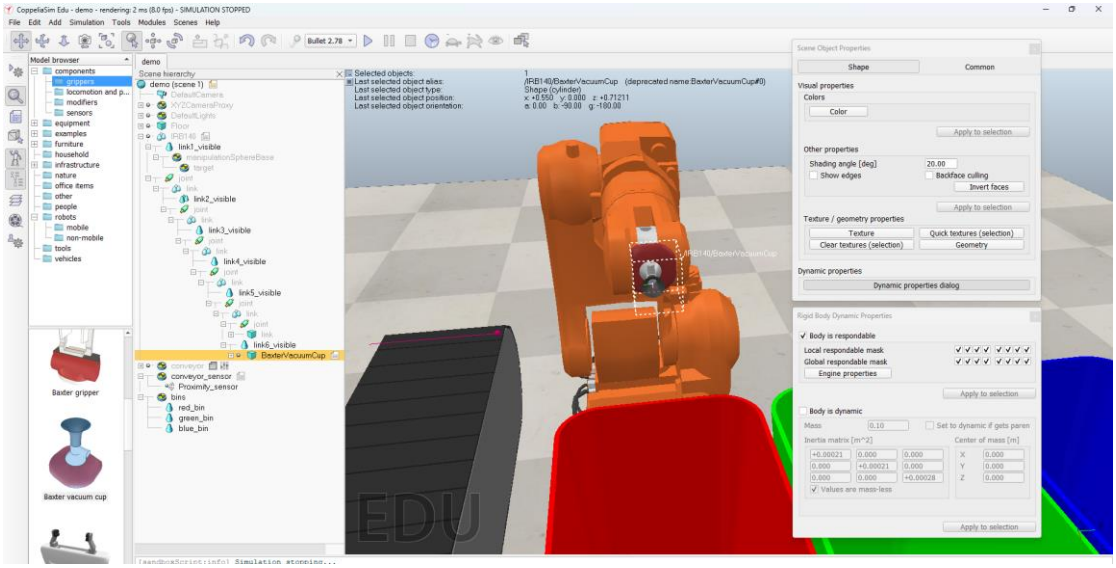


Figure 26 Adjusting the suction cup position

Chapter 3

Code for automatically generating objects with random color

The way the boxes will be created in color will be automatic through code with the Lua programming language. Once again, we create a simple "Dummy" and right-click on "Dummy" and press **Add>Associated child script>Threaded>Lua**. We rename "Dummy" as "create_boxes" and click right on the white box created to write code.

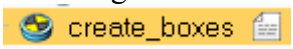


Figure 27 Display of virtual "create_boxes" object

Before writing the code, we need to right-click on "conveyor" and press "Expand selected trees" to see its entire structure.

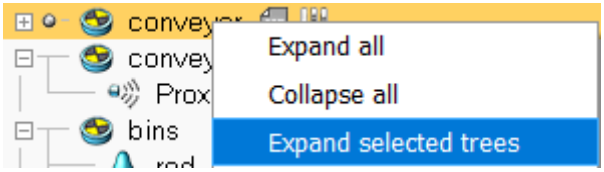


Figure 28 Conveyor belt structure revealed

What we need to do now is select the last "_ctrlPt" to find its coordinates (we mark them for the code – the red arrow) and rename it to "ctrlPt". In the picture, we change the name of "_ctrlPt[17]". The reason we want to get these coordinates is to determine where the boxes will fall automatically.

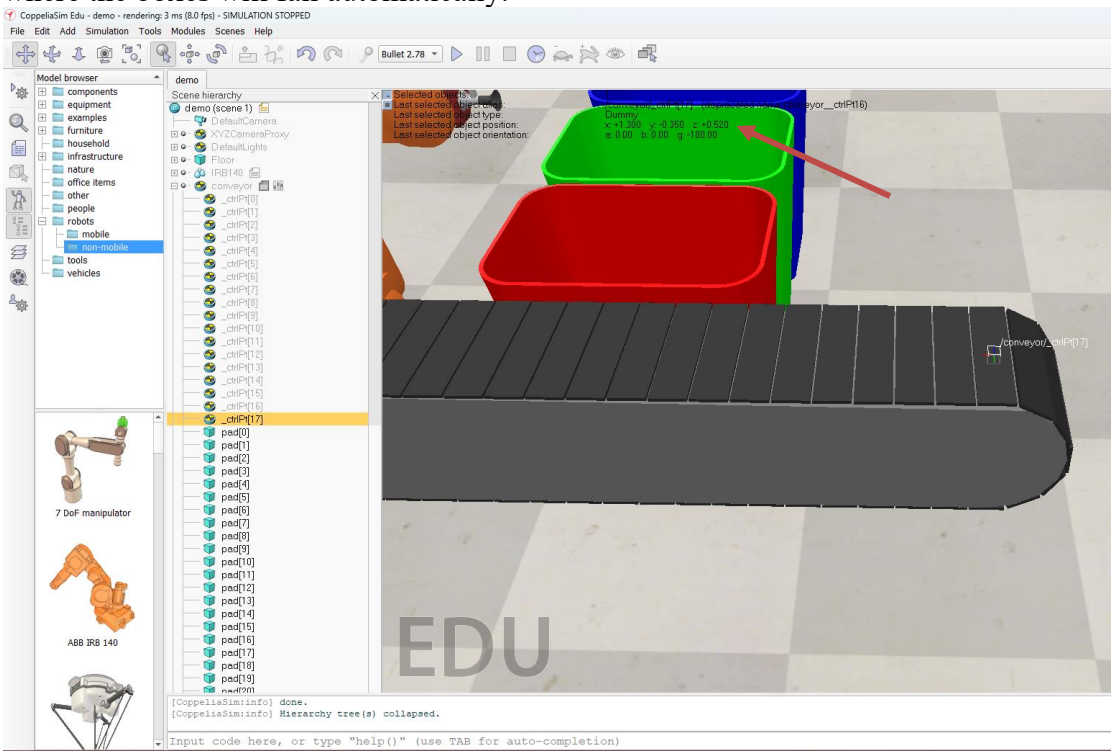


Figure 29 Capturing box creation point

```lua
-- lua

function sysCall_init()
    sim = require('sim')

    -- Put some initialization code here
    ctrlPt = sim.getObjectHandle('ctrlPt')
    -- sim.setStepping(true) -- enabling stepping mode
end

function sysCall_thread()
    -- Put your main code here, e.g.:



    ---------------
    -- orismos ton xromaton
    local colors = {
        {1.0, 0.0, 0.0, 1.0}, -- kokkino
        {0.0, 1.0, 0.0, 1.0}, -- prasino
        {0.0, 0.0, 1.0, 1.0}  -- mple
    }


    while sim.getSimulationState() ~= sim.simulation_advancing_abouttostop do

        local colorIndex = math.random(1, #colors) -- Generate random index
        local color = colors[colorIndex] -- Access random color from the array
        local box = sim.createPureShape(0, 16, {0.1, 0.1, 0.1}, 0.01, color)

        sim.setObjectPosition(box, -1, {1.1, 0.425, 0.59})
        sim.setShapeColor(box,nil,sim.colorcomponent_ambient,color)
        sim.setObjectInt32Parameter(box,sim.shapeintparam_static,0)
        sim.setObjectInt32Parameter(box,3004,1)
        sim.setObjectSpecialProperty(box,sim.objectspecialproperty_collidable+sim.objectspecialproperty_measurable+sim.objectspecialproperty_detectable_all)
        sim.wait(30)

    end
end
```

Figure 30 create_boxes code

In short, we have defined the table of colors and with an endless loop, the boxes are created with the color chosen randomly (with dimensions we have defined – 0.1, 0.1, 0.1) in the coordinates **(x=1.1, y=0.425, z=0.59)** where the belt's starting point is every 30 seconds. The value that z has is the one we captured +0.07 to drop the can from above. Last but not least, we define specific properties for each object/box, i.e. that the specified object is collidable, measurable and can be detected by any other object in the simulation environment [1].

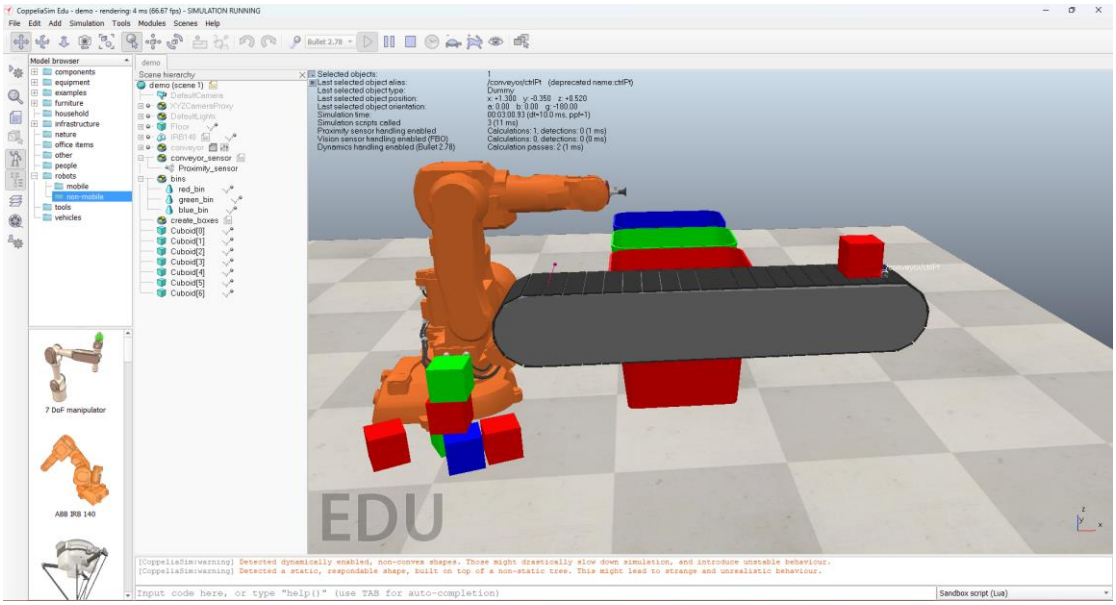


Figure 31 Example of simulation execution

Chapter 4

Color detection sensor management code on conveyor belt

The way in which the robotic arm will be informed about the color of the object is based on the code below. By clicking on the icon  of the dummy "conveyor_sensor" from the "Scene hierarchy", we can manage the code.

Child script "/conveyor_sensor"

```lua
-- lua

function sysCall_init()
    sim = require('sim')

    -- Put some initialization code here
    Proximity_sensor= sim.getObjectHandle('Proximity_sensor')
    conveyor= sim.getObjectHandle('conveyor')

    sim.setInt32Signal('boxColor',1)
    sim.setInt32Signal('detectedBox',0)
    -- sim.setStepping(true) -- enabling stepping mode
end

function sysCall_thread()
    -- Put your main code here, e.g.:
    while true do
        result,sensor_dist,detectedPoint,detectedObjectHandle=sim.readProximitySensor(Proximity_sensor)
        if result>0 then -- if sensor detect the box
                -- stop the conveyor
                --sim.writeCustomDataBlock(conveyor,'CONVMOV',sim.packTable({vel=0}))
                sim.writeCustomTableData(conveyor,'__ctrl__',{vel=0.0})
                sim.setInt32Signal('detectedBox',1)
                sim.setInt32Signal('detectedObjectHandle',detectedObjectHandle)

                -- get the detected box color and impliment it to the 'boxcolor' signal
                r,rgbData=sim.getShapeColor(detectedObjectHandle,nil,sim.colorcomponent_ambient_diffuse)
                if rgbData[1] == 1 then sim.setInt32Signal('boxColor',1)
                elseif rgbData[2] == 1 then sim.setInt32Signal('boxColor',2)
                else sim.setInt32Signal('boxColor',3)
                end
        else
            --restart the conveyor
            --sim.writeCustomDataBlock(conveyor,'CONVMOV',sim.packTable({vel=0.1}))
            sim.writeCustomTableData(conveyor,'__ctrl__',{vel=0.05})
            sim.setInt32Signal('detectedBox',0)
            sim.setInt32Signal('detectedObjectHandle',-1)

        end
    end
end
```

Figure 32 Conveyor belt code

In a few words, it checks if the sensor has detected something. If so, it stops the conveyor, determines the color of the box (rgbData[1]=red, .. [2]= blue, .. [3]=green) and updates some signals. If it does not detect anything, it restarts the conveyor belt and resets the signals. It is worth noting that the initial speed set in the belt was 0.1 m/s which did not allow enough time for the sensor to detect the box, causing it to drop. For this reason, the speed is now 0.05m/s with a waiting time of 30 seconds between the boxes.

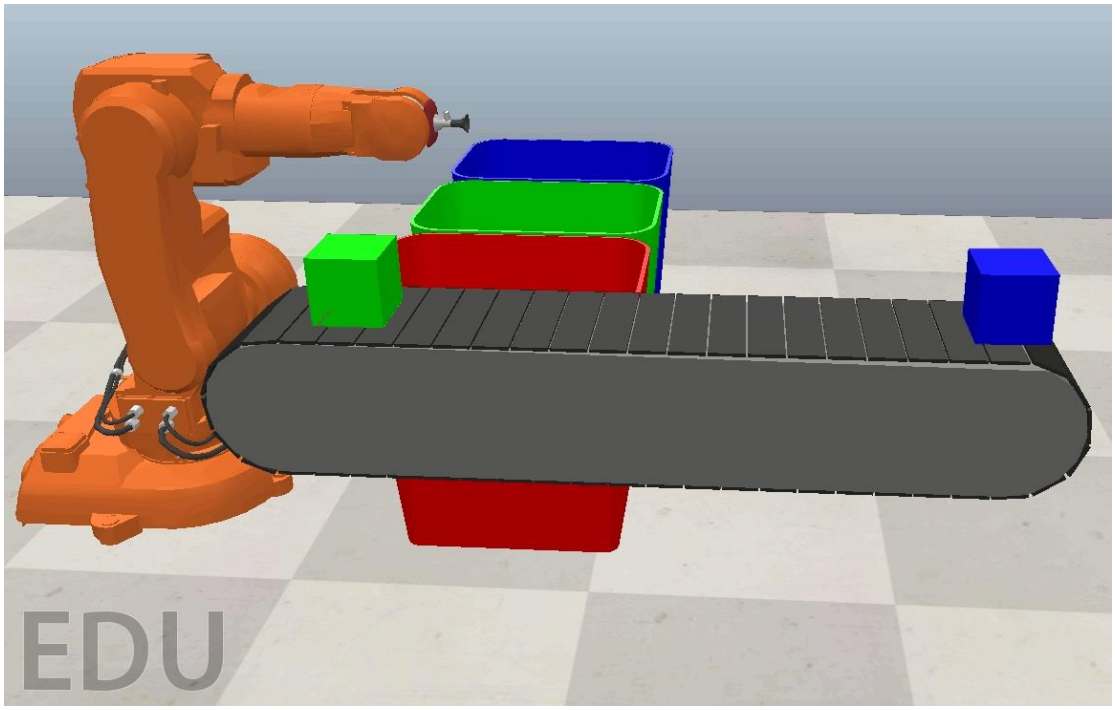


Figure 33 Example of simulation execution

We notice that the green box has been detected and is waiting until the robotic arm picks it up (in the next chapter). Every 30 seconds, a new box is created (like the blue one pictured) and waits until the color sensor detects nothing, so that the belt starts working again and moving.

Chapter 5

Motion management of the robotic arm

The difficulty of the whole project lies in the way the robotic arm moves, in order to sort the colored objects in the appropriate bin. With the help of the "display_position" tool (link to download [3]) coordinates (x,y,z) are determined with great precision, in order to determine the phases of sorting. Essentially, this tool gives the coordinates of its position (quaternion and position) to define in the code in which position we want the robot to move. To import it, we press from the File **>Load model menu...** and we choose it.

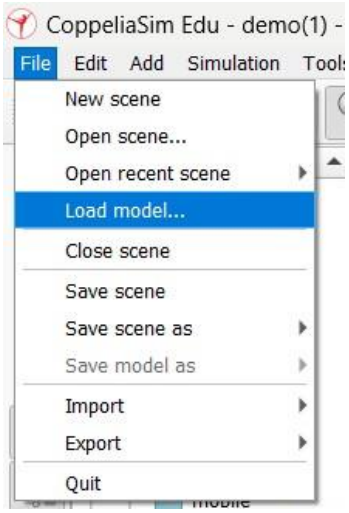


Figure 34 Importing display_position (1)

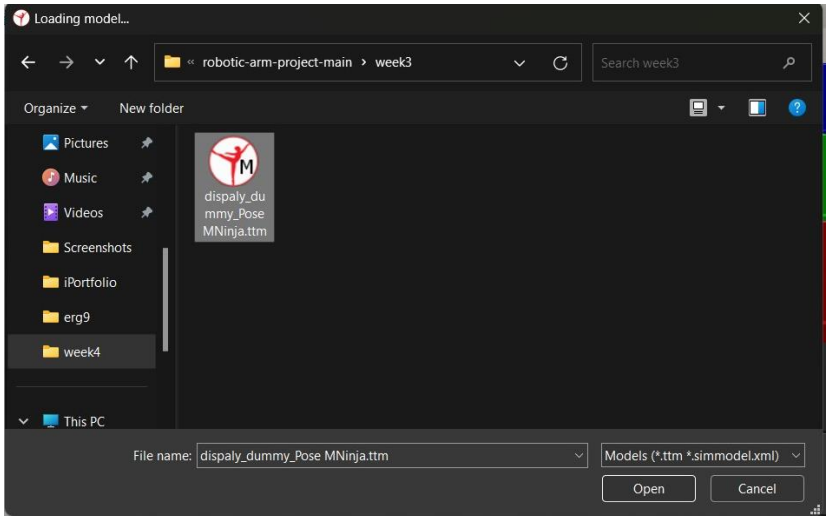


Figure 35 Importing display_position (2)

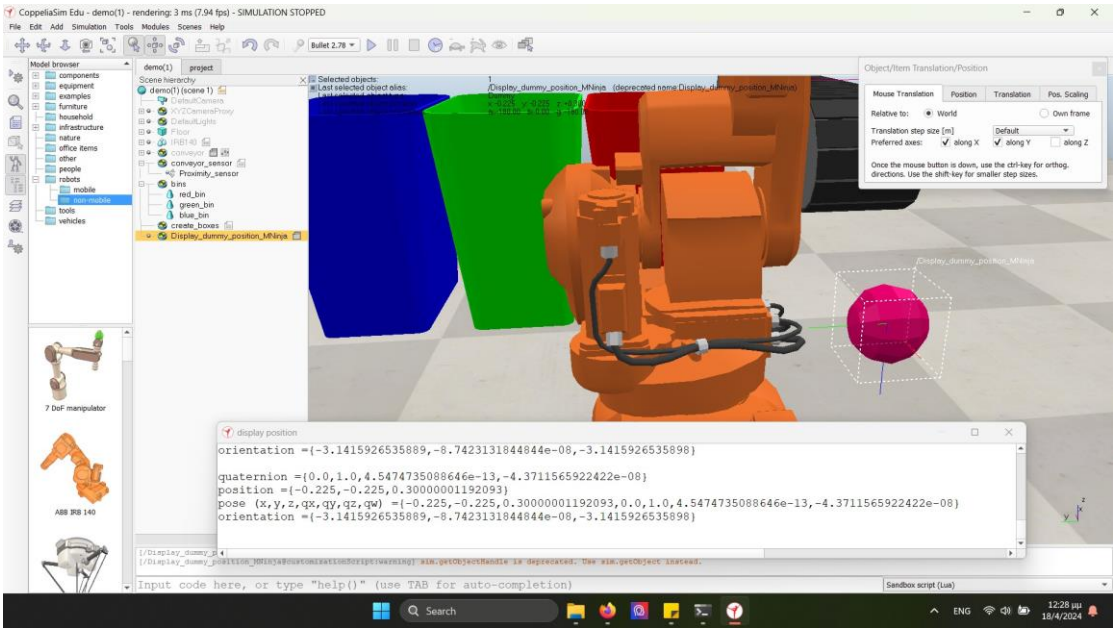


Figure 36 Diplaying display_position (Display_dummy_position_MNinja)

Sorting phases:

1. The robot moves over the position where an object should be picked up.
2. An object is expected to be detected on the conveyor belt.
3. The robotic arm picks up the detected object.
4. The detected object is picked up by the robot.
5. The robotic arm returns to the starting position.
6. The detected object is moved to the correct bin.
7. The robotic arm releases the object.
8. If the bin is full, it is emptied.

In phases 1, 3 and 6 we use the tool for each case:

a. Above the box (starting position).

b. To the box – pickup.

c. To the bin – release.

Details for each step of sorting and the code used to do all the above with the sense of automation in an effective way are described below.

Capturing robotic arm positions:

The "display_position" is a "Dummy" and can be moved, as can all the components we added in Chapter 1 with the "object/item shift" button. In order for the coordinates to appear when we move it, we must press the gray box to the right of the "Display_dummy_position_MNinja", to see its code and press the icon aka restart script.

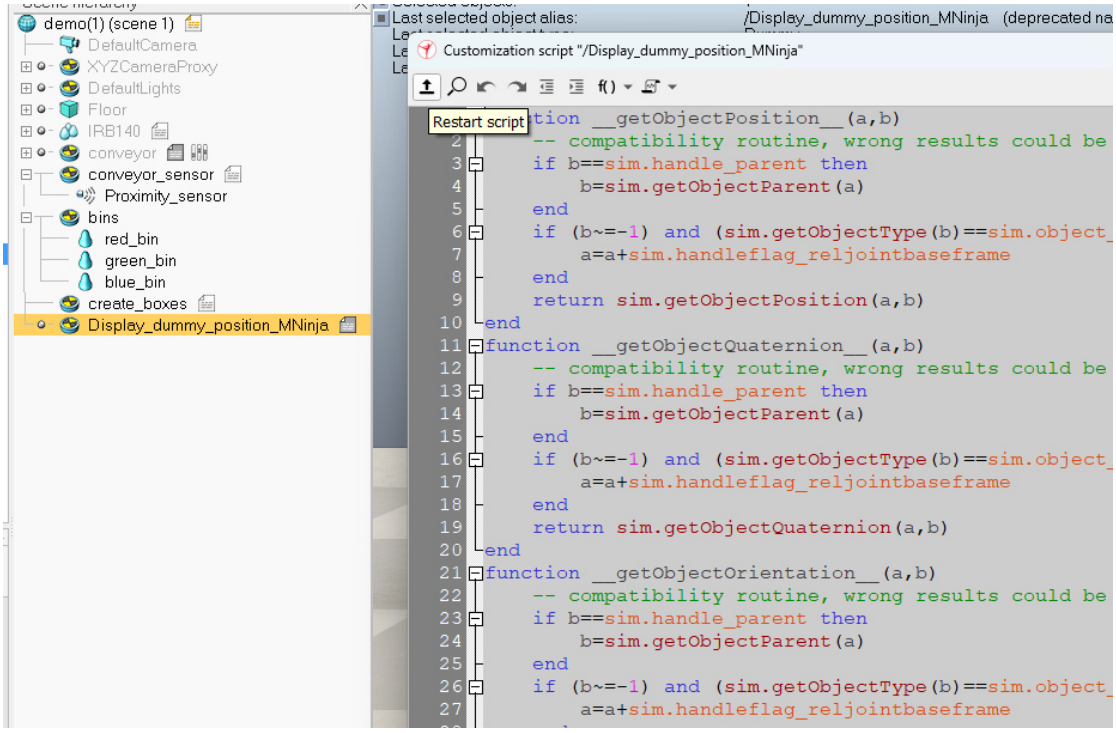


Figure 37 How to display coordinates

a. Above the box (starting position).

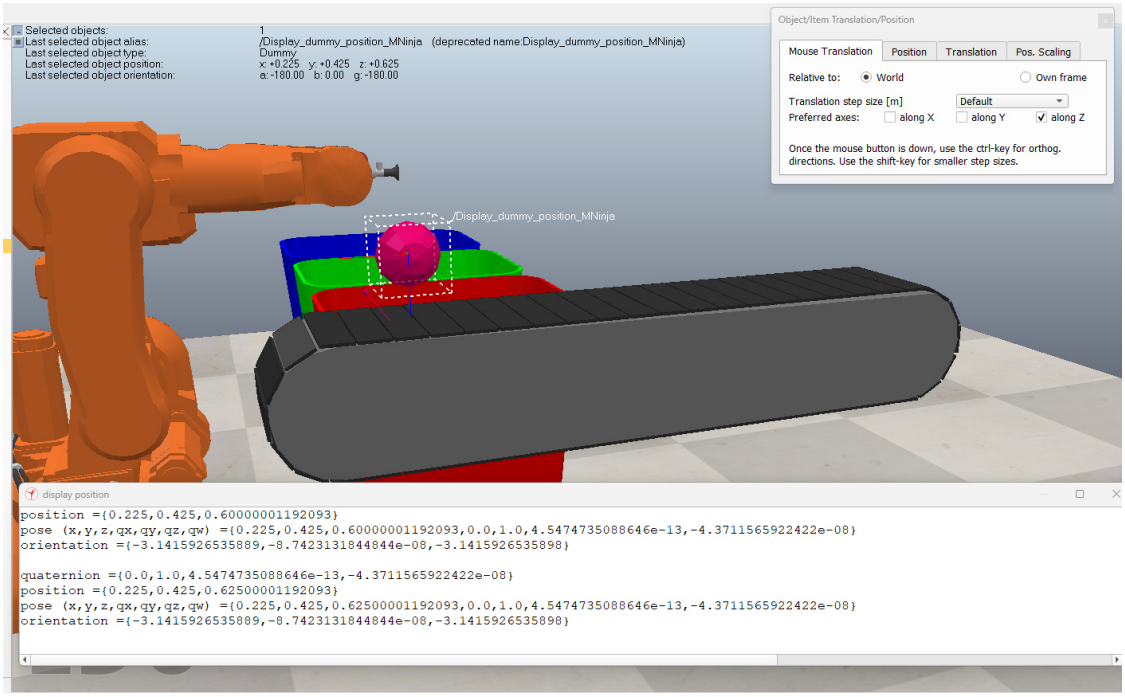


Figure 38 Starting position - Direction of robotic arm towards the box

b. To the box – pickup.

The only difference is that Z is -0.05 from the starting position to the position.

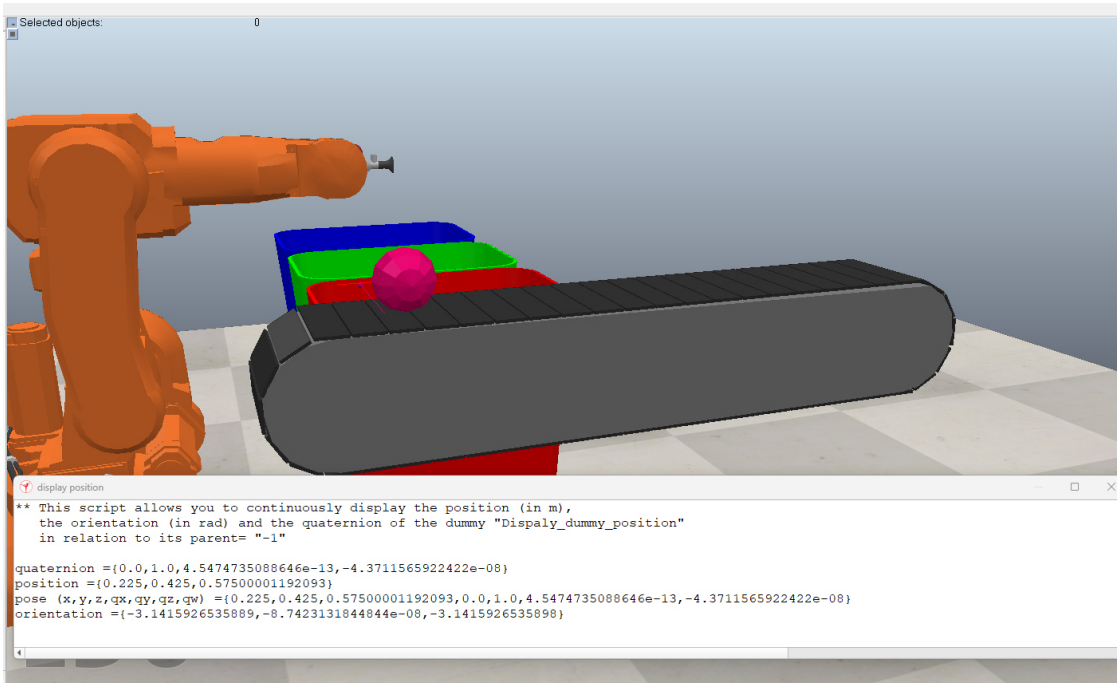


Figure 39 Item pickup position

c. To the bin – release.

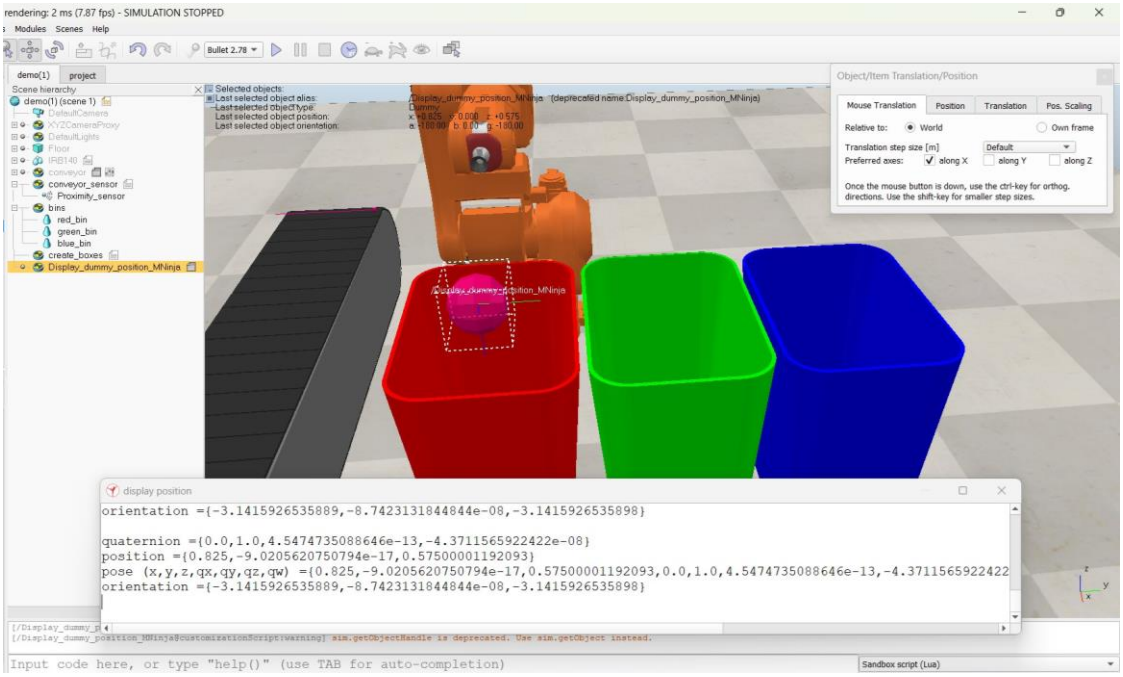


Figure 40 Object release position in the centre of the bin

We do not need to capture this position for each bin, as we can use the distance difference between them against the Y axis ("step"), as the other coordinates are the same. This will be needed for the code. Once we finish with the "display_position", we move it aside and look at the "step" between the red and the green bucket and see that it is 0.325 (**object item/shift>Position**).

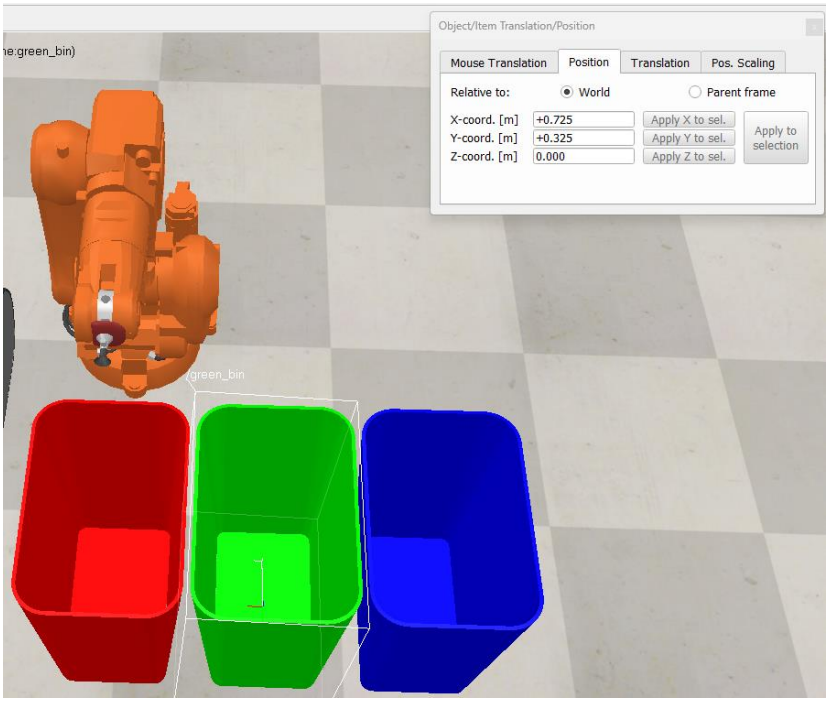


Figure 41 Determination of the distance between the bins

With the help of the "step", we will inform the robotic arm how far to its left it needs to move to find the appropriate bin, given the color of the object, i.e. in action it will become red=0 (0*0.325+position of case c), green=1 (1*0.325+position of case c) and blue=2 (2*0.325+position of case c). This can be best seen in the code below, since we have created a new "Dummy" named "main" (**Add>Associated child script>Threaded>Lua**).

Child script "/main"

```lua
-- lua

function sysCall_init()
    sim = require('sim')

    -- Put some initialization code here



    --object handles

    target = sim.getObjectHandle('/IRB140/target')
    IRB140 = sim.getObjectHandle('/IRB140')
    tip = sim.getObjectHandle('/IRB140/tip')


    -- set up parameters :
    vel = 0.1
    accel = 70
    jerk = 70
    currentVel={0,0,0,0}
    currentAccel={0,0,0,0}
    maxVel={vel,vel,vel,vel}
    maxAccel={accel,accel,accel,accel}
    maxJerk={jerk,jerk,jerk,jerk}
    targetVel={0,0,0,0}


    -- sim.setStepping(true) -- enabling stepping mode
end

function sysCall_thread()
    -- Put your main code here, e.g.:
    local bins = {
    [1] = {handle = sim.getObjectHandle('/red_bin'), count = 0, name="red"},
    [2] = {handle = sim.getObjectHandle('/green_bin'), count = 0, name="green"},
    [3] = {handle = sim.getObjectHandle('/blue_bin'), count = 0, name="blue"}
        }
    local max=2

    --main code
```

Figure 42 Robotic arm motion management code with bin cleaning (1)

Child script "/main"

```lua
    -- sim.setStepping(true) -- enabling stepping mode
end

function sysCall_thread()
    -- Put your main code here, e.g.:
    local bins = {
    [1] = {handle = sim.getObjectHandle('/red_bin'), count = 0, name="red"},
    [2] = {handle = sim.getObjectHandle('/green_bin'), count = 0, name="green"},
    [3] = {handle = sim.getObjectHandle('/blue_bin'), count = 0, name="blue"}
        }
    local max=2

    --main code
    while true do

        --1-- move the robot above the pickup position
        quaternion1 ={0.0,1.0,4.5474735088646e-13,-4.3711565922422e-08}
        position1 ={0.41600000113249,-0.35,0.55500004142523}

        targetPos1={0.39100000113249,-0.35,0.55500004142523,0.0,1.0,4.5474735088646e-13,-4.3711565922422e-08}

         --.moveToPose rmlMoveToPosition an den douleuei to apo kato
        sim.rmlMoveToPosition(target,IRB140,-1,currentVel,currentAccel,maxVel,maxAccel,maxJerk,position1,quaternion1,targetVel)


        --2-- waiting for the sensor to detect the imported object on the conveyor
        while sim.getInt32Signal('detectedBox')==1 do
            detectedObjectHandle=sim.getInt32Signal('detectedObjectHandle')

            --3-- pickup the detected object
            quaternion2 ={0.0,1.0,4.5474735088646e-13,-4.3711565922422e-08}
            position2 ={0.41600000113249,-0.35,0.50500004142523}

            targetPos2={0.39100000113249,-0.35,0.50500004142523,0.0,1.0,4.5474735088646e-13,-4.3711565922422e-08}

            sim.rmlMoveToPosition(target,IRB140,-1,currentVel,currentAccel,maxVel,maxAccel,maxJerk,position2,quaternion2,targetVel)

            --4-- fake the gripping by making the detected object a child to the robot flange
            sim.setObjectInt32Param(detectedObjectHandle,3003,1) --static parameter
            sim.resetDynamicObject(detectedObjectHandle)
```

Figure 43 Robotic arm motion management code with bin cleaning (2)

Figure 44 Robotic arm motion management code with bin cleaning (3)

Figure 45 Robotic arm motion management code with bin cleaning (4)

Briefly, the robotic arm first moves to the starting position (**case a - quaternion1 and position1**) which is directly above the belt. Also, an indicative threshold for emptying a full bin (local max=2) and a "bins" table containing objects for each color with elements such as the handle of the bin, the number of items in each bin (count) and its name to display appropriate messages, has been defined, when one of them is filled.

Then, the detection of an object is checked and if it is true, then the robotic arm enters the pickup phase (**case b - quaternion2 and position2**) bending to take the object and "takes" it in a hypothetical way with the suction cup, implementing it as a child with tip as its parent. This means that it will be stuck to the tip (the tip of the suction cup), even if it's further away from it. Once the color of the object has been detected, at this point the corresponding number of objects must be increased and sorted in the appropriate bin (**case c - quaternion3 and position3**). An important detail that contributed to the emptying of the bins is the definition of their objects as childs in them (setObjectParent() [1]), as in this way we were able to simply delete the "branches" (removeObject()) of the "tree" (getObjectsInTree()).
In order for the code to work with our own coordinates and "step", we replace the values quaternion1,2,3 and position1,2,3 and the "step" (code point: sim.getInt32Signal('boxColor')-1)***0.325**) respectively. Also, we can set our own threshold for emptying the bins to see how many boxes it will take to fill it all the way up, e.g. local max=50.

Example of simulation execution:

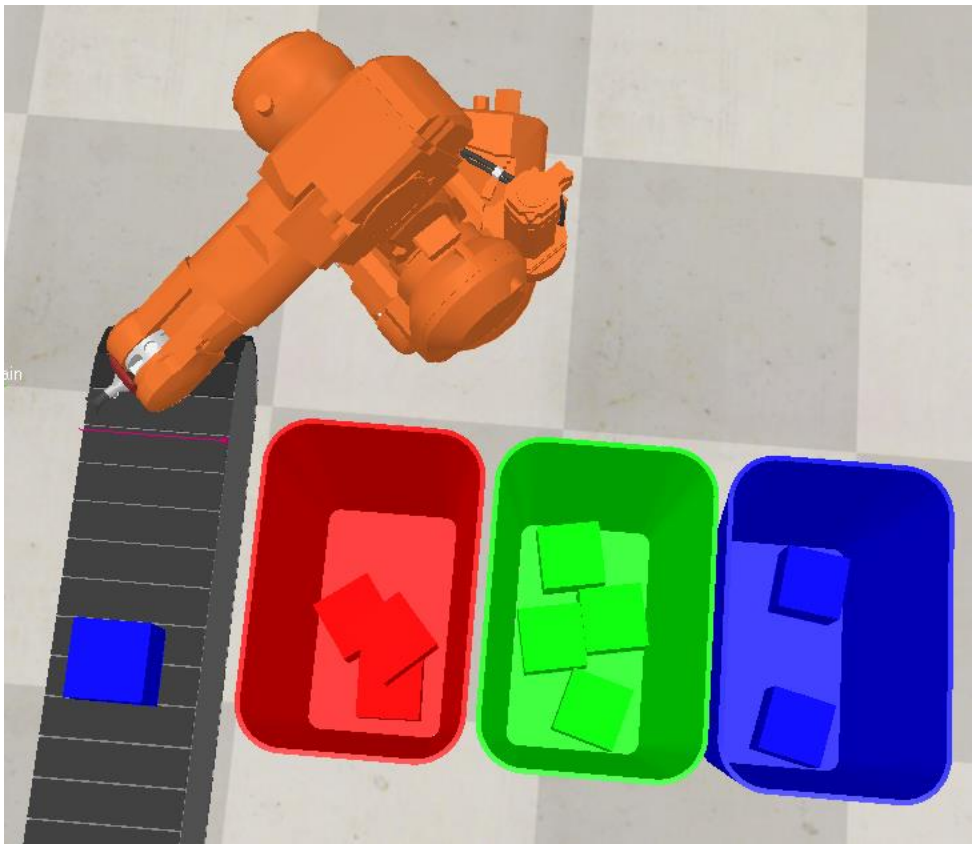


Figure 45 Arm in position

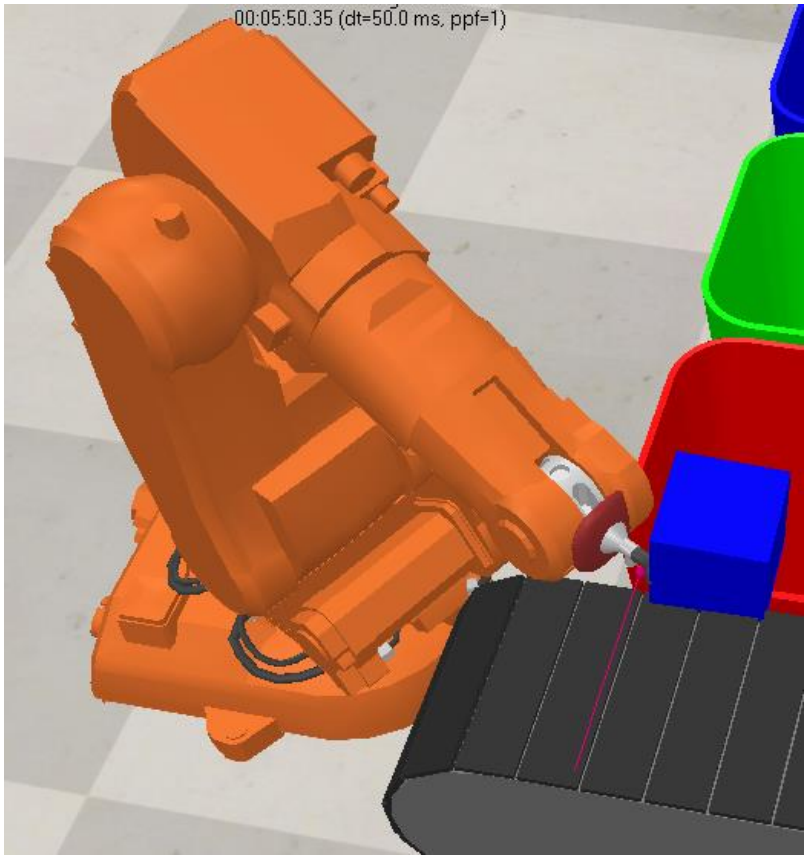


Figure 46 Box pickup with the suction cup of the arm

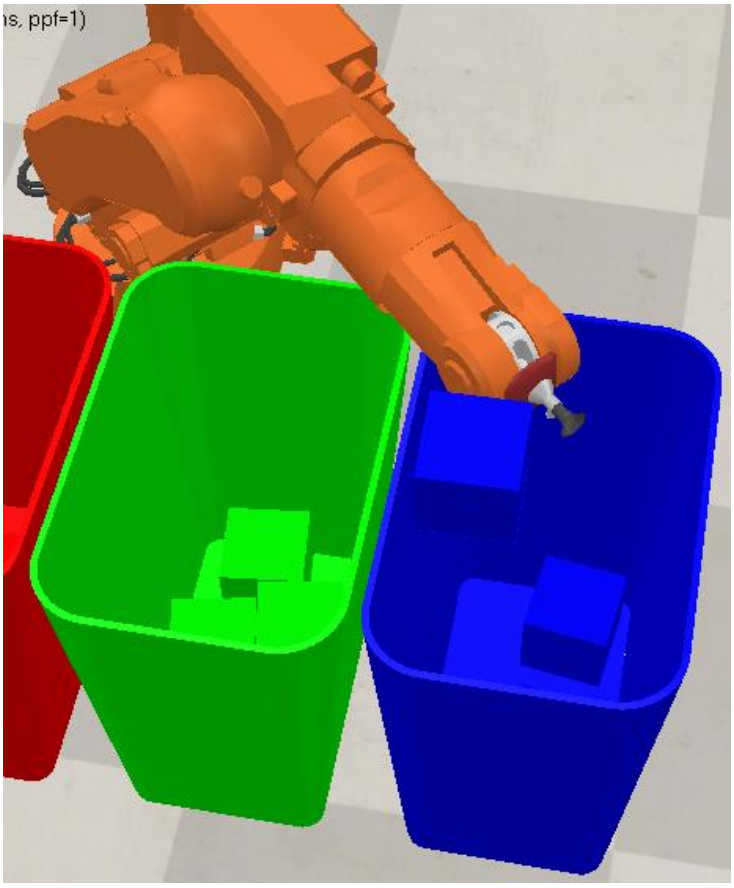


Figure 47 Isolating the box from the suction cup of the arm – Placing the box in the bin of its color

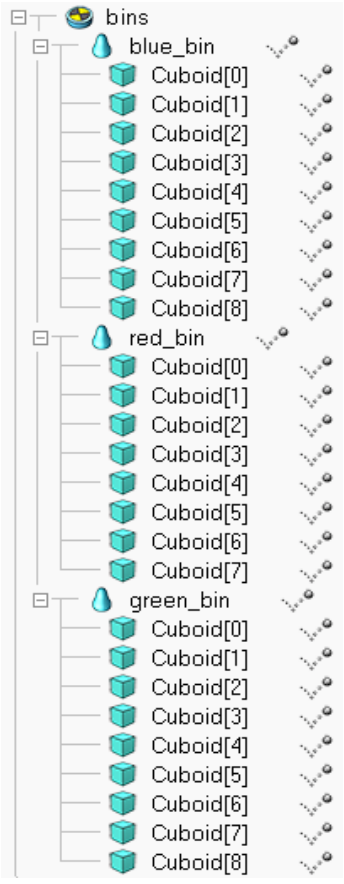


Figure 48 Way of displaying the placing of the boxes in each bin

```
Emptying blue bin because it's full.
```

Figure 49 Informative message when a full bin is emptied

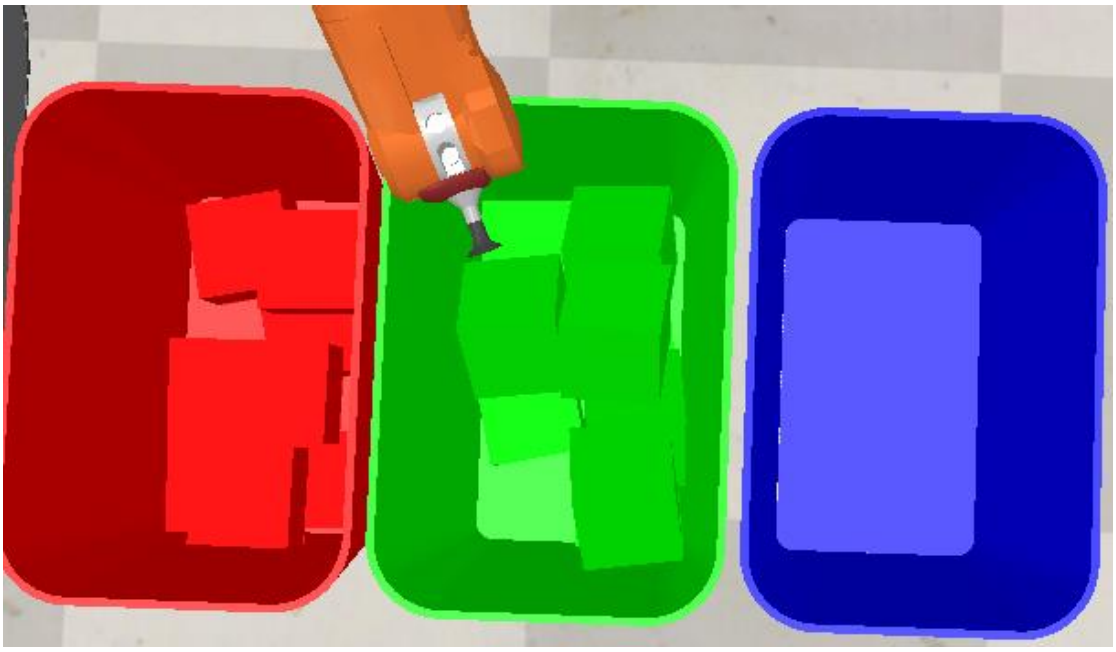


Figure 50 Emptying the bin that was filled

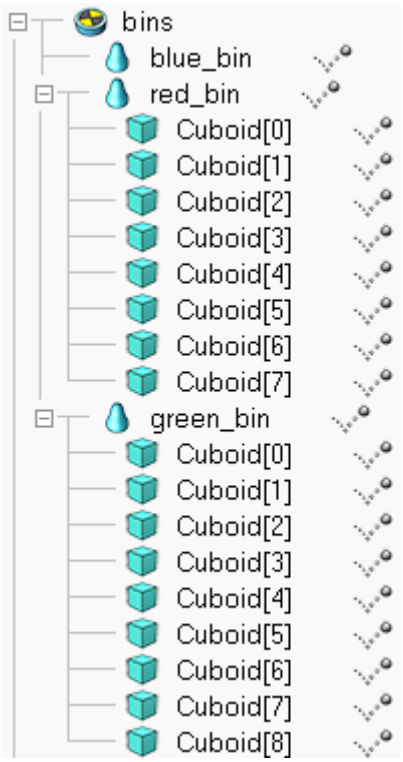


Figure 51 Way of displaying the placing of the boxes in each bin after emptying

Bibliography

[1] CoppeliaSim - Regular API reference
https://manual.coppeliarobotics.com/en/apiFunctions.htm

[2] 3D bin file
https://drive.google.com/file/d/1zD0vsH7t15jEGhuzms557tJR-9FaqMjb/view

[3] Display position tool
https://drive.google.com/file/d/1xjCQ6Hkgimtvn9xdH7vlqhxknn_ty3wN/view