Lab 1: Script

Modules:

Introduction:

Welcome to Robot Academy. In this lab we will be building a simulation based on a scenario in which products coming down an assembly line will be sorted into categories by a robotic arm. The applications of this simulation include industrial tasks such as defective product inspection and removal, or passing products being built from one part of assembly to another.

This lab will consist of 10 sections. We will start by importing models and creating our mock factory to produce a variety of objects. These objects will populate a conveyor belt that constantly pushes items towards a robotic arm. A vision sensor will read the type of product coming down the line. The ABB IRB 140 robotic arm model will be configured with inverse kinematics to take the products to the correct location.

By the end of the lab, you will be able to drive a robotic arm consisting of multiple joints to simulate the automation of industrial tasks.

Build the Environment:

Build the Robotic Arm:

We will begin this project by creating a new scene and importing some models that are included with CoppeliaSim. If the pane is not already open, you can click on the robot icon ( ) on the left icon bar to open it. The centerpiece of our scene will be the robotic arm, so navigate to **robots -> non-mobile** in the model browser and drag the **ABB IRB 140.tmm** somewhere near the center of our scene.

Next, we can grab the gripper for our robotic arm. Travel to **components -> grippers** in the model browser, select the **ROBOTIQ 85** model and drag it anywhere onto the scene. We can attach the gripper by first selecting the **ROBOTIQ 85** object, holding shift, and second, selecting the **IRB_connection** object. With both objects selected in that order, click on the **Object/item shift** icon ( ) and click the **Position** tab in the window that opens. With the position set relative to the **World** and click **Apply to selection** to translate the gripper's position to the designated connection on our robotic arm.

We can do the same but with just **ROBOTIQ 85** with the orientation by selecting the **Object/item rotate** icon ( ) and the **Rotation** tab. Relative to **Own frame** enter 180 into the field for **Around Y [deg]** to correct the gripper's orientation. Return to the **Object/item shift** mode to pull the gripper down a bit. Go to the **Translate** tab and enter **-0.02** into the **Along Z [m]** field and click **Z-translate sel.** once.

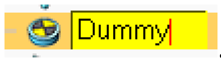
Lastly, select **ROBOTIQ 85** in the **Scene hierarchy** and drag it over **IRB140_connection** to make it the child of that object. Play the scene with . Try opening moving the robotic arm and closing the gripper with the provided UI by clicking on **ROBOTIQ 85** or **IRB140** (either in the scene or the scene hierarchy).

Build the Assembly Line:

The next step in this lab is to build our assembly line. This will consist of importing a **Dummy** object that will act as a marker for the position to programmatically produce cuboids of various colors. We will also be importing a **conveyor belt (efficient)** which can be used to move the assembly line and detect product types (cuboid colors).

In the model browser, click on **equipment -> conveyor belts** and drag the **conveyor belt (efficient)** model out onto the scene, one end of which should be in reach of the robotic arm.

Next, Click **Add** in the menu bar (alternatively, right click on the game scene -> **Add**) and add **Dummy** object to the scene. Move the dummy somewhere near the end of the conveyor belt opposite to the robotic arm. This dummy's position will be located by a script that will also produce our cuboids. For now, we can shrink the dummy and place it such that, when the scene is playing, the produced shapes will be generated somewhere over that end of the conveyor belt.

Double click the icon to the left of the **Dummy** object in the **Scene hierarchy**: . Under the **Dummy** enter **0.05** in the **Object size[m]** field**.** Move the **Dummy** to the location you want your cuboids to be produced using the mouse and **Object/item shift** (holding **ctrl** allows you to adjust the Z position). Double-click on the name of the **Dummy** object in the scene hierarchy to rename it to '**Factory**' 
Dummy
.

Build the Product Detector:

Click **Add** and under **Vision sensor** choose **Orthographic type**. We will use this to measure a single pixel of a block's color as it nears the robotic arm's end of the conveyor. Use **Object/item rotate** to flip the **Vision_sensor** by 90 degrees by entering **90** into the **Around X** field to make the vision sensor parallel to the floor.

Use **Object/item shift** and mouse controls to align the sensor perpendicular to the conveyor's movement.

Double click on the next to the newly added vision sensor named **Vision_sensor**. Under the **Vision sensor** tab, change the far clipping plane to **0.3** in the **Near / far clipping plane [m]** second field. Also change the **Resolution X / Y** to **1 / 1**. Exit this window and use the **Object/item shift** tool to adjust the **Vision_sensor**.

Our last objects to add are the tables. In the model browser, navigate to **furniture -> tables** and drag a **customizable table** onto the scene within reach of the robotic arm. In the slider the opens up, turn **Height** all the way down to **0.41**. Adjust the **Length** and **Width** so that two more tables can fit within reach of the robotic arm. Repeat this process for two more tables surrounding the robotic arm.

Select the **ConveyorBelt** and then, while holding shift, select one of the tables. In the **Object/item shift** mode, go to the **Position** tab and relative to **World,** click **Apply Z to sel.** Our tables and conveyor belt should now all be at the same height.

The last step is to color-code our tables for each category of product. Expand one of the **customizableTable**s using the ⊞ button. Double click the icon( ) next to the **customizableTable_tableTop** object. Click on **Adjust color** under the **Shape** tab and in the window that opens click **Ambient/diffuse component**. You can choose any three colors for the three tables, but this tutorial will choose a shade of red, green, and blue. Repeat this process for the other two tables.

Configure the Factory:

Generate Colored Cuboids:

In this section we will be writing the code to produce an endless number of products for our assembly line.

In the menu, click on **Tools** and then **Scripts**. In the window that opens ensure that **Simulation scripts** in the top dropdown is selected and click **Insert new script**. This will be a **Child script (threaded)** and click **OK**.

In the list view for the **Scripts** window, click on **Main script (default)** and then click on **Non-threaded child script(unassociated)** to select our newly created script. Under **Script properties** change the value for **Associated object** to the name of the dummy at the end of the conveyor belt (probably **Factory**) and exit the window. A script icon should appear next to that object:

Double click it.

We can start by erasing all of the green text commented out with **'—'** which leaves us with two functions: **sysCall_threadmain()** and **sysCall_cleanup()**. We will add the first lines of **sysCall_threadmain()** with a call to get the handle for our factory dummy to get it's position:

- **factory=sim.getObjectHandle("Factory")**
- **factoryPosition=sim.getObjectPosition(factory,-1)**
  (-1 specifies to get factory position relative to the world)

Next we can add lines to define the cuboid's color:

- **colors={ {0.9,0.5,0.5},{0.5,0.9,0.5},{0.5,0.5,0.9} }**
  (R, G, and B: Three sets of RGB values between 0.0 and 1.0)
- **currentColor=colors[1]**
  (Lua sequential data structures are 1-indexed, here our first cuboid will be red)

Create a function **changeColor()** to handle randomizing colors:

- **function changeColor()**
    - **randomNumber=sim.getRandom()**
    - **if (randomNumber < 0.33) then**
        - **return colors[1]**
    - **elseif (randomNumber >=0.33 and randomNumber <0.67) then**
        - **return colors[2]**
    - **else**
        - **return colors[3]**
    - **end**
- **end**

Create a while loop in **sysCall_threadmain** to forever produce cubes in 10 second intervals:

- **while (true) do**
    - **currentColor=changeColor()**
      Assign a new color to our current color by calling our color randomizing function
    - **cuboid=sim.createPureShape(0,15,{0.05,0.05,0.05},0.2,nil)**
      Create a new cuboid (**0**), with these properties.
      Reference:
      https://www.coppeliarobotics.com/helpFiles/en/regularApi/simCreatePureShape.htm
    - **sim.setObjectInt32Parameter(cuboid, 3003, 0);**
      Make the object dynamic.
    - **sim.setObjectInt32Parameter(cuboid, 3004, 1);**
      Make the object respondable.
      Reference:
      https://www.coppeliarobotics.com/helpFiles/en/objectParameterIDs.htm
    - **sim.setObjectSpecialProperty(cuboid, sim.objectspecialproperty_renderable)**
      Make the object renderable
      Reference:
      https://www.coppeliarobotics.com/helpFiles/en/apiConstants.htm#sceneObjectSpecialProperties
    - **sim.setShapeColor(cuboid, nil,sim.colorcomponent_ambient, currentColor)**
      Set the shape's color
    - **sim.setObjectParent(cuboid,-1,true)**
      Make the object have no parent
    - **sim.setObjectPosition(cuboid,-1,factoryPosition)**
      Place the object exactly where our **Factory** dummy is.
    - **sim.wait(10)**
      Wait 10 seconds before producing another cuboid

- **end**

We can now save the script and return to our scene and press play. Cubes of 3 different colors should be coming out over our **Factory** dummy. If the conveyor belt is going in the wrong direction or too fast/slow, double-click this icon: next to it in the **Scene hierarchy**. Change the direction by selecting the **converyorBeltVelocity** parameter and flipping the sign of the number in the **Value** field. Play around with this parameter to get the cuboids to steadily flow towards the robotic arm.

Detect the Cuboids:

At the other end of the conveyor belt, where the vision sensor is, we want to stop the conveyor belt when a cuboid is available to the robotic arm and read the color. We can start by opening the child script of **ConveyorBelt** by double-clicking the icon next to it in the **Scene hierarchy.** This script contains code that handles the movement of the conveyor belt. The local variable **beltVelocity** on line 8 will be what we manipulate to stop and start it programmatically.

Grab the handle for our vision sensor by adding this line to the end of **sysCall_init()**:

- **visionSensor=sim.getObjectHandle("Vision_sensor")**

At the start of **sysCall_actuation()** we can begin inserting our own code to read from the vision sensor.

- **imageBuffer = sim.getVisionSensorCharImage(visionSensor,0,0,1,1)**
  Returns a single pixel buffer.
- **r = tonumber(string.byte(imageBuffer, 1))**
- **g = tonumber(string.byte(imageBuffer, 2))**
- **b = tonumber(string.byte(imageBuffer, 3))**
  r, g, and b are values between 0 and 1 representing intensity of red, green, or blue

By default, if the vision sensor cannot read anything, the color will be black or 0, 0, and 0 for red, green, and blue. Therefore, as long as the **r, g,** and **b** values are all not 0, the belt should be moving. Otherwise, a cube is detected and it should stop. We will take the line defining **beltVelocity** and set it based on these conditions:

- **local beltVelocity=nil**
- **if (r~=0 and g~=0 and b ~=0) then**
  - **beltVelocity=0**
- **else**
  - **beltVelocity=sim.getScriptSimulationParameter(sim.handle_self,"conveyorBeltVelocity")**
- **end**

We will utilize this same concept in another script that will control the robotic arm. In the menu bar go to **Tools -> Scripts** and select the **Non-threaded child script** associated with **IRB140**. Make the **Associated object** the value **none**. Create a new **Simulation script** of type **Threaded child script** and make the **Associated object** of this newly created script **IRB140.** In the **Scene hierarchy** double click the script icon next to **IRB140** . Add the code already written to get the color of cuboids:

- **visionSensor=sim.getObjectHandle("Vision_sensor")**

- **while (true) do**
  - **imageBuffer = sim.getVisionSensorCharImage(visionSensor,0,0,1,1)**
  - **r = tonumber(string.byte(imageBuffer, 1))**
  - **g = tonumber(string.byte(imageBuffer, 2))**
  - **b = tonumber(string.byte(imageBuffer, 3))**

  - **print("R: "..r.." G: "..g.." B: "..b)**
  - **sim.wait(5)**
    The above 2 lines are optional and can be deleted after verifying correct behavior when pressing play.
- **end**

Our factory is now all set up and the last steps are to configure our robotic arm to carry them away to their correct table.

Configure the Robotic Arm:

Turn on Inverse Kinematics:

Inverse kinematics can be used to automatically compute the positions of the joints in our robotic arm such that we only need to worry about the positions and path that the center of the gripper follows. This point will be designated as the **tip** and is already included in the **IRB140** model with the name **IRB140_tip**. Similarly, we have a dummy near the base of the **IRB140** hierarchy named **IRB140_target**. This is the object we will use to tell the tip where to be in terms of position and orientation.

We can start by selecting the **IRB140_tip** and using the **Object/item shift** mode to bring the dummy down to the center of the gripper where items will be grabbed. This can be achieved by clicking the **Translation** tab and translating **Along Z:** -0.1 and **Along Y:** -0.015.

Next, open the **Calculation Modules** window by clicking the f(x) icon on the left bar. Click on the **Kinematics** tab and notice the two inverse kinematics groups already added for us: **IRB140_undamped** and **IRB140_damped.** To allow the main script to handle inverse kinematic functionality, click on the **IRB140_undamped** IK group and uncheck **Explicit handling**. Choose the **IRB140_damped** IK group and uncheck **IK group is active**. With damping enabled, the movement of the robotic arm is more accurate but convergence on the locations we define for it to move will be slower. This is unnecessary for this lab.

Close this window and play the scene. Notice that the robotic arm jumped up to reach the position of the **IRB140_target.** Select the **IRB140_target** and use mouse controls to move the target around the game scene. Th robotic arm should track the position using **Object/item shift** and the rotation with **Object/item rotate**. Also notice that the gripper may become damaged if hitting other objects in the scene. Our next step will be to set positions and paths for the robotic arm to follow to avoid this.

Setup Dummy Points:

Like our **Factory** dummy, we will be creating dummy points to specify locations for the robotic arm to move to in certain situations. This will include our grab position where the cuboid stops on the conveyor at the vision sensor, release positions for each table, an idle position, and points to help the tip find the right path to the tables.

We can start by playing the scene until a cube stops at the vision sensor. Select that **Cuboid** object and go to **Object/item shift** under the **Position** tab. Copy and paste the three values for **X-coord, Y-coord,** and **Z-coord** somewhere. Stop the scene and **Add** a new dummy named **grabPosition**. In **Object/item shift** mode, copy and paste the three values into the **Position** relative to **World.** The dummy designating our target to grab should now be exactly where the cuboids end up on the conveyor. Move this dummy up in the **Z** direction by **0.03** meters to account for the size of the gripper.

We will do the same for a new dummy named **idlePosition.** A good idea is to make this dummy somewhere above the grab position we just made. A good way to do this is to select **idlePosition** and then click **grabPosition** while holding shift. Open **Object/item shift** mode and **Apply to selection** relative to **World** under the **Position** tab. Then with only **idlePosition** selected, go to the **Mouse Translation** tab and select **along X** and **along Z** for the **Preferred axes.** Move the **idlePosition** dummy manually to somewhere above the grab position.

To align three dummy objects for each table, we will want to bring the gripper to just above the tabletop. Start by creating three dummy object named **redPosition, greenPosition,** and **bluePosition** which will be the position for the robotic arm to release blocks. Expand the hierarchy for one of the tables to see the **customizableTable_tableTop(**#X). Select one of the color dummies in the **Scene hierarchy** and while holding shift, select the color's corresponding tabletop. In **Object/item shift** mode, **Apply to selection** the position of the dummy to sit on the center of the table. Select the **Mouse Translation** button and check the **Along X** or **Along Y** to move the dummy to the edge of the table nearest the robotic arm.

Since we defined the 3 dimensions of our cuboid to be **0.05** in the **Factory** script, we should move the dummies on the tables above the table by **0.05** in the **Z** direction. This will allow for a small drop but accounts for the size of the gripper. In **Object/item shift** mode, go to the **Translation** tab and relative to **World** and the value **0.05**, click **Z-translate sel.** once. Do this for each color's release position dummy.

Lastly, we will make three more dummies named **redPathPosition, greenPathPosition, and bluePathPosition**. These dummies will not only allow us to move to the correct positions without knocking into already placed cuboids or tables. They can also be used to define a path for the robotic arm to turn around and drop on the table behind it.

Make these match the position for the release dummies for their color with position **Apply to selection.** Translate the position for these three dummies **0.2** meters above their corresponding color's release dummies.

Write Code to Automate the Robotic Arm:

We can start by writing the code to move the gripper to the correct position at the right time. Open the threaded script for our **IRB140** base object. This should already be reading the color of cuboids in a while loop. We will start by grabbing the object handles for each of our dummies and getting the positions and orientations of them. Under where we grab the handle for the **visionSensor** in **sysCall_threadmain(),** add the following lines:

- **target=sim.getObjectHandle("IRB140_target")**
  Grab the target, we will be changing the position and orientation of this object which the robotic arm follows with inverse kinematics
- **grabDummy=sim.getObjectHandle("grabPosition")**
- **idleDummy=sim.getObjectHandle("idlePosition")**
- **redDummy=sim.getObjectHandle("redPosition")**
- **greenDummy=sim.getObjectHandle("greenPosition")**
- **blueDummy=sim.getObjectHandle("bluePosition")**
- **redPath=sim.getObjectHandle("redPathPosition")**
- **greenPath=sim.getObjectHandle("greenPathPosition")**
- **bluePath=sim.getObjectHandle("bluePathPosition")**

Grab the positions and orientations. Although it is not necessary, it may help to group the ones relating to colors:

- **grabPosition=sim.getObjectPosition(grabDummy, -1)**
- **grabOrientation=sim.getObjectOrientation(grabDummy, -1)**
- **idlePosition=sim.getObjectPosition(idleDummy, -1)**
- **idleOrientation=sim.getObjectOrientation(idleDummy, -1)**
- **colorReleasePositions={ sim.getObjectPosition(redDummy, -1),**
  sim.getObjectPosition(greenDummy, -1),
  sim.getObjectPosition(blueDummy, -1)}
- **colorReleaseOrientations={ sim.getObjectOrientation(redDummy, -1),**
  sim.getObjectOrientation(greenDummy, -1),
  sim.getObjectOrientation(blueDummy, -1)}
- **colorPathPositions={ sim.getObjectPosition(redPath, -1),**
  sim.getObjectPosition(greenPath, -1),
  sim.getObjectPosition(bluePath, -1)}
- **colorPathOrientations={ sim.getObjectOrientation(redPath, -1),**

sim.getObjectOrientation(greenPath, -1),
sim.getObjectOrientation(bluePath, -1)}

- **changeTarget(idlePosition, idleOrientation)**

The last line added is a call to a function we will create called **changeTarget** which will start our robotic arm in the idle position. Position and orientation are passed into this function which manipulates our target:

- **function changeTarget(position, orientation)**
  - **sim.setObjectPosition(target, -1, position)**
  - **sim.setObjectOrientation(target, -1, orientation)**
- **end**

Now, in the while loop after where we define **r, g,** and **b**, we will check whether any color is detected and if so, which one, which is used to define our target:

if (r ~= 0 and g ~= 0 and b ~= 0) then

- **targetPosition = nil**
- **targetOrientation = nil**
- **targetPathPosition = nil**
- **targetPathOrientation = nil**

- **if (r > g and r > b) then**
  - **targetPosition = { colorReleasePositions[1][1], colorReleasePositions[1][2], colorReleasePositions[1][3]}**
  - **targetOrientation = { colorReleaseOrientations[1][1], colorReleaseOrientations[1][2], colorReleaseOrientations[1][3]}**
  - **targetPathPosition = { colorPathPositions[1][1], colorPathPositions[1][2], colorPathPositions[1][3]}**
  - **targetPathOrientation = {colorPathOrientations[1][1], colorPathOrientations[1][2], colorPathOrientations[1][3]}**

  Define the two dummies to navigate to the red table.

- **elseif (g > r and g > b) then**
  - **targetPosition = { colorReleasePositions[2][1], colorReleasePositions[2][2], colorReleasePositions[2][3]}**
  - **targetOrientation = { colorReleaseOrientations[2][1],**

colorReleaseOrientations[2][2],
colorReleaseOrientations[2][3]}

- **targetPathPosition = { colorPathPositions[2][1], colorPathPositions[2][2], colorPathPositions[2][3]}**
- **targetPathOrientation = {colorPathOrientations[2][1], colorPathOrientations[2][2], colorPathOrientations[2][3]}**

Define the two dummies to navigate to the green table.

- **elseif (b > r and b > g) then**
    - **targetPosition = { colorReleasePositions[3][1], colorReleasePositions[3][2], colorReleasePositions[3][3]}**
    - **targetOrientation = { colorReleaseOrientations[3][1], colorReleaseOrientations[3][2], colorReleaseOrientations[3][3]}**
    - **targetPathPosition = { colorPathPositions[3][1], colorPathPositions[3][2], colorPathPositions[3][3]}**
    - **targetPathOrientation = {colorPathOrientations[3][1], colorPathOrientations[3][2], colorPathOrientations[3][3]}**

  Define the two dummies to navigate to the blue table.

- **End**

The next bit is the heart of our program that will define all of the movement our robotic arm will make. It consists of multiple calls to **changeTarget()** and **sim.wait(n)** which waits for **n** seconds so the arm can have time to move:

- **changeTarget(grabPosition, grabOrientation)**
- **sim.wait(1)**
- **changeTarget(idlePosition, idleOrientation)**
- **sim.wait(1)**
- **changeTarget(targetPathPosition, targetPathOrientation)**
- **sim.wait(2)**
- **changeTarget(targetPosition, targetOrientation)**
- **sim.wait(2)**
- **changeTarget(targetPathPosition, targetPathOrientation)**
- **sim.wait(1)**
- **changeTarget(idlePosition, idleOrientation)**
- **sim.wait(1)**

If you press play now, the robotic arm should be moving automatically when a cube reaches the vision sensor. However, if the robotic arm must turn around to the table behind it, it may bend backwards instead of turning completely around. This can be amended by adding a case for that color, which in my case is **green**, to stop at one of the path points for the other tables to define an intermediate point. This will force the robotic arm to turn around. The above code can be fixed to look like this:

- **changeTarget(grabPosition, grabOrientation)**
- **sim.wait(1)**
- **changeTarget(idlePosition, idleOrientation)**
- **sim.wait(1)**
- **if (g > r and g > b) then**
  - **changeTarget(colorPathPositions[3], colorPathOrientations[3])**
    Here the robotic arm travels to the blue table before the green table to drop the cuboid.
  - **sim.wait(1)**
- **end**
- **changeTarget(targetPathPosition, targetPathOrientation)**
- **sim.wait(2)**
- **changeTarget(targetPosition, targetOrientation)**
- **sim.wait(2)**
- **changeTarget(targetPathPosition, targetPathOrientation)**
- **sim.wait(1)**
- **if (g > r and g > b) then**
  - **changeTarget(colorPathPositions[3], colorPathOrientations[3])**
    Stop at the same point on the way back around.
  - **sim.wait(1)**
- **end**
- **changeTarget(idlePosition, idleOrientation)**
- **sim.wait(1)**

Lastly, we need the gripper to actually pick up and drop the cuboid. The author of the **ROBOTIQ 85** gripper model we are using suggests two methods to achieve this: by closing the gripper or simulating a pickup with object attachment. We will be using the second method as it requires less configuration and tweaking of parameters.

We will be working in the same script for **IRB140**. Start by defining some new variables in **sysCall_threadmain():**

- **gripperSensor=sim.getObjectHandle('ROBOTIQ_85_attachProxSensor')**
  This will detect if an object is in reach of the gripper
- **connector=sim.getObjectHandle('ROBOTIQ_85_attachPoint')**
  The object to bind our cuboid to as it is grabbed.
- **attachedObject=nil**
  A reference to our attached object for when we detach (drop the Cuboid)

We will now create a function called **grabCuboid()**:

- **function grabCuboid()**
  - **index=0**
  - **while (true) do**
    - **objectInScene=sim.getObjects(index,sim.object_shape_type)**
      Loop through all the objects in the scene.
    - **if (objectInScene==-1) then break end**
      If the cuboid is not found, break out. This should not be a normal case.
    - **objectName = sim.getObjectName(objectInScene)**
    - **isCuboid = "Cuboid" == string.sub(objectName,1,6)**
      Check if the object is one of our generated cuboids
    - **if ((isCuboid) and (sim.getObjectInt32Parameter(objectInScene,sim.shapeintparam_respondable)~=0) and (sim.checkProximitySensor(gripperSensor,objectInScene)==1)) then**
      If the object is in range of the gripper and has the respondable property set, attach it to the connector and break out.
      - **attachedObject = objectInScene**
      - **sim.setObjectParent(objectInScene,connector,true)**
      - **break**
  - **end**
  - **index=index+1**
  - **end**
- **end**

Our function to drop the cuboid is then as simple as setting the parent to the world instead of the connector which will simulate a drop:

- **function dropCuboid()**
  - **sim.setObjectParent(attachedObject,-1,true)**
- **end**

Lastly, we can add to the while loop in **sysCall_threadmain()** to grab and drop the cuboid when in the correct position:

- **changeTarget(grabPosition, grabOrientation)**
- **sim.wait(1)**
- **grabCuboid()**
  Grab the cuboid here and add time to wait for the grab operation.
- **sim.wait(1)**
- **changeTarget(idlePosition, idleOrientation)**
- **sim.wait(1)**
- **if (g > r and g > b) then**
  - **changeTarget(colorPathPositions[3], colorPathOrientations[3])**
  - **sim.wait(1)**
- **end**

- **changeTarget(targetPathPosition, targetPathOrientation)**
- **sim.wait(2)**
- **changeTarget(targetPosition, targetOrientation)**
- **sim.wait(2)**
- **dropCuboid()**
- **sim.wait(1)**
- **changeTarget(targetPathPosition, targetPathOrientation)**
- **sim.wait(1)**
- **if (g > r and g > b) then**
    - **changeTarget(colorPathPositions[3], colorPathOrientations[3])**
    - **sim.wait(1)**
- **end**
- **changeTarget(idlePosition, idleOrientation)**
- **sim.wait(1)**

We should now be all set to play our scene and have a fully automated robotic arm sorting our assembly line. However, there is one last step. Notice how the robotic arm places cuboids in the same place, often knocking others off the table. In the next section we will program the robotic arm to adjust the drop position on the table.

Organizing the Cuboids:

To adjust the drop position of our cuboids on the table, we should start by moving the **path** and **drop** dummies to one edge of the tables so that we have room to place more blocks in a line.

Open up the threaded script associated with **IRB140** and add the following lines at the top of the **sysCall_threadmain()** function:

- **redProducts={}**
- **greenProducts={}**
- **blueProducts={}**

These will hold the handles of the cuboids that have been placed on each table. In the same function in the while loop where we handle assigning target paths for each color add the following code under the assignments for **targetPosition, targetOrientatation, targetPathPosition,** and **targetPathOrientation:**

Red: **if (r > g and r > b) then …**

- **targetPosition[1] = targetPosition[1] - (0.1*getLength(redProducts))**
- **targetPathPosition[1] = targetPathPosition[1] - (0.1*getLength(redProducts))**
- **cuboidColor="red"**

Green: **elseif (g > r and g > b) then …**

- **targetPosition[2] = targetPosition[2] - (0.1*getLength(greenProducts))**

    - **targetPathPosition[2] = targetPathPosition[2] - (0.1*getLength(greenProducts))**
    - **cuboidColor="green"**

Blue: **elseif (b > r and b > g) then …**

    - **targetPosition[1] = targetPosition[1] - (0.1*getLength(blueProducts))**
    - **targetPathPosition[1] = targetPathPosition[1] - (0.1*getLength(blueProducts))**
    - **cuboidColor="blue"**

* **getLength()** is not a built-in function and we will define it later

Playing the scene now should result in cubes placed evenly across the table until eventually being placed out of range of the robotic arm. To fix this and have a continuously looping scene, we can delete the cuboids after a certain amount have been placed to simulate products being carried to the next part of an assembly line.

To begin we can pass the **cuboidColor** variable we defined in **sysCall_threadmain()** as an argument to our call to **grabCuboid()** (-> **grabCuboid(cuboidColor)**). In **grabCuboid()**, we can make the following changes:

- **function grabCuboid(color)**
    - **index=0**
    - **while (true) do**
        - **objectInScene=sim.getObjects(index,sim.object_shape_type)**
        - **if (objectInScene==-1) then**
        - **break**
        - **end**
        - **objectName = sim.getObjectName(objectInScene)**
        - **isCuboid = "Cuboid" == string.sub(objectName,1,6)**
        - **if ((isCuboid) and**
            - **(sim.getObjectInt32Parameter(objectInScene,sim.shapeintparam_respondable)~=0) and**
            - **(sim.checkProximitySensor(gripperSensor,objectInScene)==1)) then**
            - **attachedObject = objectInScene**
            - **sim.setObjectParent(objectInScene,connector,true)**
            - **if (color == "red") then**
                - **table.insert(redProducts, objectInScene)**
            - **elseif (color == "green") then**
                - **table.insert(greenProducts, objectInScene)**
            - **else**
                - **table.insert(blueProducts, objectInScene)**

            - **end**
            - **break**
        - **end**
    - **index=index+1**
    - **end**
- **end**

Next, we can create our **getLength()** function to help us determine how many cuboids have been placed on each table:

- **function getLength(arr)**
    - **count=0**

    - **for index,value in pairs(arr) do**
        - **count = count + 1**
    - **end**
    - **return count**
- **end**

Lastly we will define a function called **clearTables()** that will be called after each time a block is handled, at the end of the while loop in **sysCall_threadmain()**:

- **function clearTables()**
    - **redCount=getLength(redProducts)**
    - **greenCount=getLength(greenProducts)**
    - **blueCount=getLength(blueProducts)**
    - **if (redCount >= 3) then**
        - **for i,v in pairs(redProducts) do**
            - **sim.removeObject(redProducts[i])**
        - **end**
        - **redProducts={}**
    - **end**
    - **if (greenCount >= 3) then**
        - **for i,v in pairs(greenProducts) do**
            - **sim.removeObject(greenProducts[i])**
        - **end**
        - **greenProducts={}**
    - **end**
    - **if (blueCount >= 3) then**
        - **for i,v in pairs(blueProducts) do**
            - **sim.removeObject(blueProducts[i])**

        - **end**
        - **blueProducts={}**
    - **end**
- **end**

The above function first checks how many red, blue, and green objects have been placed on each table (they are inserted in our changes to **grabCuboid(color)**). Then if that number is over a certain amount the table is cleared of all the cuboids. For this lab I have kept the limit to 3, but many more can be added if desired.

Concluding Remarks:

In this lab we have learned how to utilize the principles of inverse kinematics to automate the jointed movements of our robotic arm. We have created a fully functional scene that can endlessly loop to keep the mock factory running. I encourage you to take this project further by adding more robots to the assembly line, creating dynamic obstacles for the robotic arm, or even creating a custom robotic arm and configuring it for a specialized purpose. Stay tuned for more labs from Robot Academy!