Lab 2 Script: Maze Traversal

Modules:

Introduction:

Welcome to Robot Academy. In this lab we will be building a simulation of a maze pathfinding mobile robot. The 4-wheeled vehicle will be equipped with multiple sensors to traverse the maze and recognize when it has completed. This project uses the recursive backtracking algorithm to generate the maze which guarantees a singly connected maze [1]. This will allow us to start our robot at any position and define any other position in the maze to be the end. In this lab, you will learn:

- How to programmatically generate our maze environment
- Three types of sensors: Ray Proximity, Cone Proximity, and Orthographic Vision
- How to control mobile robot's movement with revolute joints
- How to use sensors to detect the mobile robot's environment
- The wall-following algorithm for maze traversal

[1] - http://people.cs.ksu.edu/~ashley78/wiki.ashleycoleman.me/index.php/Recursive_Backtracker.html

In the next section we will start our project by importing one of the included mazes and configuring it for our robot's environment.

Build the Maze:

Recursive Backtracking Algorithm for Procedurally Generated Maze:

Included with this lab is a file, **mazes.txt,** which contains some two-dimensional arrays consisting of 0's and 1's. This is the format that will be used when writing in Lua to represent a wall (1) and empty space (0). Additionally, a file that contains the algorithm for generating these mazes is included. This lab will be using the first array in **mazes.txt.**

We will start by creating a new scene. Select **ResizableFloor_5_25** to open the **Floor Customizer**. Change both **X** and **Y** sliders to **15.00**.

Now we will generate the maze into the scene, and then copy the model to use it statically. In the menu bar, select **Tools -> Scripts**. Confirm that **Simulation script** is selected and click **Insert new script**. Select **Child script (non-threaded)** and click **OK.** Change the **Associated object** to the object **ResizableFloor_5_25**.

In the **Scene hierarchy**, double-click the script icon now next to the **ResizableFloor_5_25** object: . You may erase all of the green commented text. Grab the first 2D array from the included resource, **mazes.txt** called **Maze 1**. Copy and paste this array into the function **sysCall_init()** under which we will define some constants:

- **width = 15**

    - **height = 15**
    - **size = {1, 1, 1}**

The width and height of the maze will be 15 blocks, and the dimensions of each block is 1 on all sides. We will now write loops to generate the maze:

- **for i=1,width,1 do**
    - **for j=1,height,1 do**
        - **if (maze[i][j] == 1) then**

If this position is a 1, create a cuboid.

            - **h = sim.createPureShape(0, 8, size, 10, nil)**

Create a cuboid (**0**) that is respondable (**8**) of size, **size**, and a mass of **10**.

            - **sim.setObjectPosition(h, -1, {i-8.0, j-8.0, 0.5})**

Move cuboid to its position in the maze offset by the floor width and height divided by two plus half the width of the cuboid.

            - **sim.setObjectSpecialProperty(h, sim.objectspecialproperty_detectable_all)**

Allow the cuboid to be detected by the robot's proximity sensors.

        - **end**
    - **end**
- **end**

After saving and playing the scene, you should now see the maze centered on the floor with two cutouts for the start and end positions. With the scene playing, select all the cuboids by holding shift and dragging over all of them in the **Scene hierarchy**. Press **ctrl+c** to copy all the cuboids, stop the scene, and press **ctrl+v**. The maze should now be statically placed on the map. Select all the cuboids again with the scene stopped and right click them. Go to **Edit -> Grouping / Merging -> Group selected shapes.**

We can now either delete the script for generating the maze or disable it. Go to **Tools -> Scripts** in the menu and select the script associated with **ResizableFloor_5_25** and check the box for **Disabled**.

The last step in setting up our environment is to add an indicator for the end of the maze. This can easily be done by going to **Add -> Primitive shape -> Disc**. In the window that opens, change **X-size [m]** to **1** and click **OK**. Open up the **Object/item shift** mode () and go to the **Position** tab. Relative to **World** change the coordinates to the following:

- **X-coord. [m] → -3**
- **Y-coord. [m] → -6**
- **Z-coord. [m] → 2.0e-3**

The disc should now be at one of the two openings in the maze. This will be the end and we will color it **red**. Double click on the name of the new disc in the **Scene hierarchy** and name it **end**. Then double click the same object on the icon: . Under the **Shape** tab, select **Adjust color** and then in the window that opens click **Ambient//diffuse component.** In the window that opens, under **RGB,** turn the **Red** slider all the way up to **1.00** and the **Green** and **Blue** sliders all the way down to **0.00**.

You should now have a completed maze with an end indicator. The other opening will be where we put the vehicle in the next module. Stay tuned!

TODO: Show student how to use maze generator

Build the Robot:

Configuring the Model:

This lab will be using a four-wheeled vehicle model included with CoppeliaSim. Click on the **Model browser** on the left: Travel to **robots -> mobile** and scroll down near the bottom to find the model, **Robotnik_Summit_XL** and drag it onto the scene in the opening opposite the red disc. We can orient it towards the maze by selecting the robot in the **Scene hierarchy** and selecting **Object/item rotate** mode: . Under the **Rotation** tab, select relative to **Own frame** and enter **90** into the field for **Around Z [deg]**. Click **Z-rotate sel.** once. We can zoom into our model with **Fit to view** () and start adding the sensors.

In the menu, click **Add -> Proximity_sensor -> Cone type**. Rename it **frontProximity**. Click and drag the newly created sensor in the **Scene hierarchy** over the **Robotnik_Summit_XL** base object to make it an immediate child of it. In **Object/item shift** mode, select the sensor and in the **Position** tab, relative to **Parent frame**, and enter the following values:

- **X-coord. [m] → +3.0456e-01**
- **Y-coord. [m] → +1.6570e-04**
- **Z-coord. [m] → +4.7439e-02**

Using the **Object/item rotate** mode, select the sensor and in the **Rotation** tab enter **90** degrees in the **Around X [deg]** field and click **X-rotate sel.** until the wide part of the cone is facing away from the front of the robot.

Next, the two side proximity sensors will be added. These will tell us whether there is a wall next to the robot and how far away it is. For this we will **Add** two of the **Proximity sensor -> Ray type** to the scene. Name one of them **leftProximity** and the other **rightProximity**. Select one of the side sensors and open the **Object/item shift** mode and **Position** tab. Relative to the **Parent frame**, enter the following numbers for both side sensors:

Left:

- **+2.6085e-01**

- **+1.5107e-01**
- **-2.2450e-04**

Right:

- **+2.6116e-01**
- **-1.4912e-01**
- **+2.1708e-04**

Next, we can configure the sensors for this robot. The ray sensors for the sides of the robot should be able to sense the walls it is between no matter the position between them. However, the next square over should be out of reach and cause no detection which will tell use which ways we can turn. The default length of the ray sensors provides this effect.

The front proximity sensor should be able to detect a wall in front of it with enough room to turn around. It should also be able to detect whether there is a square one block ahead of its position. This is so that before making a turn, the robot can decide whether that is the right choice depending on if the front is blocked. This has to do with the algorithm being used which will be discussed further in the coding modules to come.

For now, select the sensor icon next to **frontProximity** in the **Scene hierarchy**. Click on **Show volume parameters** and in the window that opens, change the **Range** to **1.2000** and the **Angle** to **20.00**.

Lastly, a one-pixel vision sensor will be added to the robot, facing the floor. This will allow us to recognize the red disc indicating the maze is complete.

Select **Add -> Vision sensor -> Orthographic type** to add rename it **visionDown**. Drag this object over **Robotnik_Summit_XL** in the **Scene hierarchy** to make it the child. Next, with **visionDown** selected, hold shift and click **Robotnik_Summit_XL.** Open **Object/item shift** mode and in the **Position** tab click **Apply to selection**. Zoom in on the object and with **Object/item shift** mode still open, go to the **Mouse Translation** tab and selected relative to the **World** and **along Z**. Click and hold the mouse on the scene to lower the sensor until the blue rectangle is just poking out from the bottom of the robot. It may be necessary to change the **Translation step size [m]** to a lower value.

Change the length of the sensor by clicking the object icon in the **Scene hierarchy** ( ) and changing the **far clipping plane [m]** to **0.5**. (Second value of **Near / far clipping plane**). Also change the **Resolution X / Y** to **1 / 1** as we only need to read one red pixel to know that we have reached the end.

The robot is now completely built and ready to be programmed to traverse the maze. In the next module we will be learning how to move the robot and stay between two walls.

Vehicle Controls:

In this module we will create a very simple API that will allow us to abstract away some of the details having to do with the robot's movement and focus on getting through the maze.

We will start by opening the threaded script provided with the **Robotnik_Summit_XL** by double clicking the icon: . Delete everything in the single **sysCall_threadmain()** function except for the lines defining **motorHandles**.

Add a line defining **speed** above the joint handle definitions: **speed = 3**. Next, under the motor handles create a while loop such that it will continuously run. This will be the main driver of this project:

- **while true do**
  - **drive(0)**
- **end**

Next we can define the function **drive()**. This function will take in a single float as an argument defining the direction to travel at **speed** which was already defined. For example, **-1** will be turning in place left, **-1** for right, and 0 will be forward. Numbers in between such as **0.5** would be interpreted as turn right some while also moving forward some. This function is as follows:

- **function drive(direction)**
  - **offset=direction*speed**
    Change speed based on global variable
  - **if (direction > 1 or direction < -1) then return end**
  - **if (direction >= 0) then**
    - **offset=-offset*2**
    - **fl=speed**
    - **fr=speed+offset**
    - **br=speed+offset**
    - **bl=speed**
  - **else**
    - **offset=offset*2**
    - **fl=speed+offset**
    - **fr=speed**
    - **br=speed**
    - **bl=speed+offset**
  - **end**
  - **fr = -fr**
  - **br = -br**
  - **sim.setJointTargetVelocity(motorHandles[1],fl)**
  - **sim.setJointTargetVelocity(motorHandles[2],fr)**
  - **sim.setJointTargetVelocity(motorHandles[3],br)**
  - **sim.setJointTargetVelocity(motorHandles[4],bl)**
- **end**

The scene can now be played and with **drive(0)** in the **sysCall_threadmain()** function, the robot should move forward. Lastly we can define the function to stop the robot. The **drive** function will consistently move the robot based on **speed** and **direction**. We will define a function **stop()** to halt the wheel motors.

- **function stop()**
  - **for i=1,4,1 do**
    Loop through all four motors, setting their velocities to 0
    - **sim.setJointTargetVelocity(motorHandles[i],0)**

    - **end**
- **end**

The robot should onlyj need to stop when blocked in the front, left, and right, or if the end has been reached.

Automate the Robot:

Stay Between Walls:

The next step is to configure the robot to stay between two walls. According to the algorithm we are using, the robot should follow a single wall until either that side is unblocked, or the front is blocked. We can begin by adding a call to **sim.checkProximitySensor()** to check whether or not the side sensor detect anything, and if so, how far away it is. This will go right under the while loop in **sysCall_threadmain()**.

- **leftBlocked,leftDist=sim.checkProximitySensor(sensorHandles[1],sim.handle_all)**
- **rightBlocked,rightDist=sim.checkProximitySensor(sensorHandles[2],sim.handle_all)**

At the top of **sysCall_threadmain()**, we can add the retrieval of the sensor handles. Add the following lines:

- **sensorHandles={-1,-1,-1,-1}**
- **sensorHandles[1]=sim.getObjectHandle('frontProximity')**
- **sensorHandles[2]=sim.getObjectHandle('leftProximity')**
- **sensorHandles[3]=sim.getObjectHandle('rightProximity')**
- **sensorHandles[4]=sim.getObjectHandle('visionDown')**

We can also add a line under the call to **drive(0)** to handle following a wall:

- **followWall(wall)**

At the top of **sysCall_threadmain()**, add the lines under the declaration for **speed.** This will define the wall the algorithm is following.

- **wall=1**
- **direction=0**
- **wallDist={0.350, 0.355}**
- **turningRadius=0.25**

Change the call to **drive(0)** to **drive(direction)**. We will be modifying the **direction** variable which will define the vehicles current movement. The variables **wallDist** and **turningRadius** are the tolerance for distance from the wall and the amount of turning necessary when making corrections to this distance.

Lastly, we can add the **followWall(wall)** method to maintain distance from that wall while also traveling next to it:

- **function followWall(wall)**
    - **if (wall == 1) then**
        - **if (rightBlocked == 1) then**
            - **if (rightDist < wallDist[1]) then**

```
                direction = -turningRadius
            elseif (rightDist > wallDist[2]) then
                direction = turningRadius
            else
                direction = 0
            end
        end
    end
    if (wall == -1) then
        if (leftBlocked == 1) then
            if (leftDist < wallDist[1]) then
                direction = turningRadius
            elseif (leftDist > wallDist[2]) then
                direction = -turningRadius
            else
                direction = 0
            end
        end
    end
end
```

The robot should now be able to detect walls and travel between them. The above method simply checks which **wall** is being followed and corrects **direction** by the amount of **turningRadius** to stay between the bounds of **wallDist[1]** and **wallDist[2]**.

Try playing the scene to see this mechanism in action. Try rotating the robots start position slightly such that it must correct. However, once reaching a turn, the robot will ram straight into it. This will be handled in the next module.

Curving and Turning Around:

We have configured the side sensors such that as soon as an opening to the left or right is detected, the ray-type proximity sensors will be out of reach and detect nothing. These are the variables we have already defined in the while loop of **sysCall_threadmain()** in the **Robotnik_Summit_XL** child script called **leftBlocked** and **rightBlocked**. We can start by replacing the call to **followWall(wall)** with the following code:

```
if (leftBlocked == 0 or rightBlocked==0) then follow90()
else followWall(wall) end
```

The function **follow90()** will determine the turn to take and follow a curve along the 90 degree turn to avoid a collision.

- drive(0)
- simulationWait(2/speed)
- stop()
- leftBlocked,leftDist=sim.checkProximitySensor(sensorHandles[3],sim.handle_all)
- rightBlocked,rightDist=sim.checkProximitySensor(sensorHandles[4],sim.handle_all)
- if (leftBlocked == 1) then d=1 elseif (rightBlocked == 1) then d=-1 else d=1 end

- drive(d*0.5)
- simulationWait(12/speed)
- drive(0)
- simulationWait(1/speed)
-
- **function follow90()**
    - **drive(0)**
    - **sim.wait(1.6/speed)**
      Drive into the curve straight a little before turning.
    - **leftBlocked,leftDist=sim.checkProximitySensor(sensorHandles[1],sim.handle_all)**
    - **rightBlocked,rightDist=sim.checkProximitySensor(sensorHandles[2],sim.handle_all)**
      Read sensors again to verify which sides are available to turn into, default to **wall** direction.
    - **if (leftBlocked == 1) then dir=1 elseif (rightBlocked == 1) then dir=-1 else dir=wall end**
    - **drive(d*0.6)**
    - **sim.wait(13/speed)**
      Make the turn
    - **drive(0)**
    - **sim.wait(1/speed)**
      Travel straight for a small amount of time, don't try to correct position immediately.
- **end**

The robot should now be fully capable of making the turns in the first half of the maze, however it cannot handle dead ends or making the right choice on turns yet.

We can add a function **turn180()** to handle the case for dead ends which is as simple as this:

- **function turn180()**
-   **stop()**
-   **sim.wait(0.5)**
-   **drive(1)**
-   **sim.wait(13)**
- **end**

First the vehicle stops and then turns in place right with **drive(1).** The robot then turns until the sensor is clear and a little bit more to straighten itself with the path. Double-click the object icon next to **Robotnik_Summit_XL** : . In the **Shape** tab, click on **Show dynamic properties dialogue** and in the field for **Mass [kg]** enter **50**. This increased mass will give the tires better grip and prevent sliding into walls when performing 180 degree turns.

Now, the while loop in **sysCall_threadmain()** must be modified to check when a wall is in front of it and no left or right turn to make. We can start by reading the front proximity sensor near where the side ones are:

- **frontBlocked,frontDist=sim.checkProximitySensor(sensorHandles[3],sim.handle_all)**

Under this in the same while loop we can replace the if-else statement with this:

- **if (frontBlocked==1 and frontDist <= 0.3) then turn180()**

  This line will short circuit if the front sensor reads nothing and turns around when it is close enough.

- **elseif (leftBlocked == 0 or rightBlocked==0) then follow90()**
- **else followWall(wall) end**

Playing the scene now should result in a robot that autonomously navigates the maze. Try moving the robot to a position in the maze where there is a dead end to test this new feature.

Detecting the Finish Line:

The vision sensor has already been added to the vehicle hierarchy but has yet to be configured. A tool to help us visualize this sensor can be added by right clicking anywhere on the scene and going to **Add -> Floating view**. Select the **visionDown** object in the **Scene hierarchy** and then right click on the newly created floating view. Go to **View -> Associate view with selected vision sensor.** Playing the scene now will allow us to see the floor as the robot does.

Next we can open up the script associated with **Robotnik_Summit_XL**. We already define the sensor's handle and assign it to **sensorHandles[4]**. In the while loop of the **sysCall_threadmain()** function, we can add some code to read the sensor and detect the color red:

- **sim.readVisionSensor(visionSensor)**
  Read an image into the sensor's buffer
- **im = sim.getVisionSensorCharImage(visionSensor,0,0,1,1)**
  Read the character image from the buffer
- **r = tonumber(string.byte(im, 1))**
  r is a value between 0 and 2^8-1 representing the intensity of red
- **if (r >250) then stop() print("Finished!") return end**
  A red value over 250 is not possible with the default floor but will be triggered by the red disc

If a red color is detected, the robot will stop and the **sysCall_threadmain()** function will be returned from, breaking out of the while loop and ending the scenario.

You should now be able to start the scene with the robot in the start position and observe it traverse the entire maze and stop at the finish line. However, our robot can still be improved. In the next module we will discuss the algorithm used in this lab and how to do a full search of any maze.

Improving the Algorithm:

The front proximity sensor is much larger than we are currently using. This is because we will want to detect an entire cuboid's length ahead of the robot before making a turn.

The way the wall-following algorithm works is that it must always take the right-most direction when there is a choice. Consider the scenario in which the robot starts at the end line and tries to travel to the start. As soon as the robot arrives at the first left-hand turn (the one leading to the dead-end), it will turn left into it although it could have gone straight. When the robot returns from the dead end, since we are using the <u>right wall-following</u> algorithm, we want it to take the forward path (the right-most one) instead of turning left. This will ensure that a maze is always completely traversed before returning to the start point which signifies no end can be found.

We can achieve this result by reading the front proximity sensor whenever a turn is detected and decide the correct path to take. Start by swapping the vehicle **Robotnik_Summit_XL** and the finish line **end**.

At the first left-hand turn the robot should know to go straight. We can add this feature with the following code modified in the while loops of **sysCall_threadmain()**.

- **if (frontBlocked==1 and frontDist <= 0.4) then turn180()**
- **elseif (leftBlocked==0 or rightBlocked==0) then**
    - **if (frontBlocked==0) and**
      If a side is open to turn down but the front is also open
    - **(wall==1 and leftBlocked==0) or**
      Wall following right (**wall**=1) and left side open
    - **(wall==-1 and rightBlocked==0)) then**
      Wall following left (**wall**=-1) and right side open
        - **followWall(wall)**
          Then go straight
    - **else**
        - **follow90()**
          Otherwise, turn down that path
    - **end**
- **else followWall(wall) end**

In this lab we have learned how to programmatically produce a maze of cuboids and how to build a mobile robot with sensors to detect its surroundings. The wall following algorithm used is simple but powerful way to configure a robot to fully traverse a maze until reaching this finish line.

I encourage you to continue building this scenario by possibly remembering the shortest path for the next time through. We can count which turns the vehicle takes and compare the effect with the next time through. You can also try turning the speed variable in the child script of Robotnik_summit_xl up and tweaking parameters such as the vehicles weight and tire materials to finish the maze as fast as possible. Stay tuned for more labs from Robot Academy!