Course ID: C951
Course title: Introduction to AI
Project Title: Task II
Student name: Lawrence Sanzogni
Student ID: 012143472

Introduction

This submission will be a demonstration of how a search-and-rescue robot can aid and facilitate recovery efforts of the victims involved in a natural disaster.

Task A

The disaster recovery environment will be the ground floor of a house affected by a tornado. There will be two rescue subjects involved. One will be walled off in a room by debris, preventing escape. The other will be trapped under a support beam.

Task B

The robot will improve disaster recovery efforts by scanning the environment for disaster victims and reporting their locations, and the total number of victims to a remote human rescue team. This will provide a preliminary situational report to a human rescue team, allowing them to better develop rescue strategies based on the victims' location (e.g., choosing entry and exit routes, prioritizing victims, etc.). Another benefit of deploying a robot before a human rescue team is that it removes the risk of potential injury to the rescue team during reconnaissance, improving the safety of the team and, consequently, improving the odds of the rescue.

Task C

The bubbleRob tutorial program was used as a program scaffolding for the modifications that I've made. Therefore, I won't be listing any boilerplate code that was already present in the default tutorial (e.g., the vision sensor). I'll begin by listing off the CoppeliaSim shapes that I've added and why they were necessary additions:

Left and right sensors

These are used to detect walls adjacent to the robot. The robot's navigation controls are based on a labyrinthian principle (i.e., if at a wall, keep moving east or west, depending on the labyrinth, and you'll eventually either traverse the entire area or find the exit). These sensors aid in the recovery efforts for the robot as they allow the robot to either turn or move forward when needed. When one of the sensors makes contact with a wall, the robot will trace the wall until it reaches a corner, where it will proceed to turn. This provides a sense of logic to the robot's search algorithm and has a potentially lower time complexity when compared to randomly moving forward, colliding with an object, rotating to a preset angle, and then moving forward again until another object collides with the robot.

A proximity sensor

This sensor performs 360-degree revolutions, essentially scanning the nearby area of the robot. The sensor is calibrated so that it can only detect people, and it can't detect the same person twice. Once the person is detected, a global state object is updated, adding the person's name and {x, y, z} coordinates (meant to represent geographic coordinates. This sensor sits on top of the robot and is meant to pick up people at a sitting or standing eye level.

A proximity sensor dummy

This is used as the proximity sensor's anchor to the bubbleRob robot and allows the sensor to rotate clockwise 360 degrees.

A floor sensor

It acts identically to the proximity sensor, with the only difference being its placement. The sensor is angled down to the floor. This sensor was added to detect people who are injured and lying on the floor flat (which the regular proximity sensor can't pick up).

A floor dummy

This is used as the floor sensor's anchor to the bubbleRob robot and allows the sensor to rotate counterclockwise 360 degrees.

Task D

The robot keeps an internal representation of its environment in two ways:

A global state script

A script file was added to contain all global state, acting similarly to a state machine. The following two environment variables are monitored and reported: the number of people that the robot's *proximity* and *floor sensors* have detected, and the name and location of all people that have been detected. Once the simulation ends, the robot prints out this state date (i.e., the console logs the total number of people found and a printed list, showing the name and location of every person found).

This was achieved using CoppeliaSim API methods for passing data across multiple scripts using *Custom Data* and *Signals* methods.

code fig.:

```
function sysCall_init()
    sim = require('sim')
    -- 'STATE':

    -- 'count state':
    sim.setIntProperty(sim.handle_scene, "signal.count", 0) -- keeps count

    -- 'People' table state
    People = {}
    sim.setBufferProperty(sim.getObjectHandle('bubbleRob'),
'customData.people', sim.packTable(People)) -- caches a list of people
found

End

function sysCall_cleanup()
    printState()
end

function printState()
    count = sim.getIntProperty(sim.handle_scene, "signal.count",
        {noError = true})
    people = sim.getBufferProperty(sim.getObjectHandle('bubbleRob'),
        'customData.people', {noError = true})

    if people then
        people = sim.unpackTable(people)
    end

    print('TOTAL FOUND:', count, 'people:', people) -- prints the total
number of people found and a list of all people objects, including their
names and location
end
```

The signal data is non-persistent and keeps track of data at runtime, while the custom data persists in memory even if the scene is terminated.

Navigation flags

The robot keeps track of its immediate environment through the use of three sensors (front, left, and right) that activate when they come in contact with a wall, and script flags (grouped together as ‘*cases*’) that are either turned on or off, depending on which sensor is activated. The flags are essentially prompts that, depending on which flag is on/off (or true/false), will guide the robot to either turn left or right, or continue straight. The state of the flags can be viewed as a representation of the environment’s current state in relation to the robot. This allows the robot to navigate through the area and give logic to its navigation choices. It can essentially be seen as a 2-D modeling of the environment.

code fig.

```
sensors = {
    rightSensorDetecting = sim.readProximitySensor(rightSensor) >= 1,
    leftSensorDetecting = sim.readProximitySensor(leftSensor) >= 1,
    frontSensorDetecting = sim.readProximitySensor(frontSensor) >= 1
}

cases = {
    atRightCorner = sensors.rightSensorDetecting == true and
        sensors.frontSensorDetecting == true and
        sensors.leftSensorDetecting == false,
    tracingRightWall = sensors.rightSensorDetecting == true and
        sensors.frontSensorDetecting == false and
        sensors.leftSensorDetecting == false,
    tracingLeftWall = sensors.rightSensorDetecting == false and
        sensors.frontSensorDetecting == false and
        sensors.leftSensorDetecting == true
    }
```

Coordinate constants

The robot also keeps track of its current Z-coordinate orientation and a targetZ coordinate that represents the target coordinate that the robot needs to reach to turn an entire 90 degrees either left or right.

code fig.

```
function turnLeft()
    local function resetCoord()
        turning = false
        targetZCoordinate = 0
        calls = 0
        threads.leftThread = false
        threads.forwardThread = true
    end

    turning = false
    targetZCoordinate = bubbleRob:getOrientation()[3] + math.rad(90)
    absoluteCurrCoord = math.abs(currentOrientation)
    absoluteZtarget = math.abs(zWrapNormalization(targetZCoordinate))
    count = 0

    if absoluteCurrCoord > absoluteZtarget then
        while absoluteCurrCoord - absoluteZtarget > 0.01 do
            count = count + 1
            bubbleRob:setOrientation(math.rad(1))
            currentOrientation = bubbleRob:getOrientation()[3]
            absoluteCurrCoord = math.abs(currentOrientation)
            sim.step()
        end
        resetCoord()
    elseif absoluteZtarget > absoluteCurrCoord then
        while absoluteZtarget - absoluteCurrCoord > 0.01 do
            count = count + 1
            bubbleRob:setOrientation(math.rad(1))
            currentOrientation = bubbleRob:getOrientation()[3]
            absoluteCurrCoord = math.abs(currentOrientation)
            sim.step()
        end
        resetCoord()
    else
        print('COMPLETE_______________')
        resetCoord()
    end
end
```

Task E

The robot implements the following concepts to achieve its recovery goal:

- Knowledge: As described in previous sections, the robot collects person data and saves it to a makeshift 'state' machine. This allows the user to keep track of how many people there are and where they are.
- Reasoning: Also described in previous sections, the robot uses flags to navigate the environment and, depending on which flag is on/off (i.e., true/false), the robot will either turn or continue forward.
- Uncertainty: The robot's reasoning in uncertain environments is limited, but effective in a 2-D space. The robot can really only have two states: when at least one sensor is activated, and when no sensors are activated. The robot's default state is to turn right (labyrinthian principle) since it's assumed that the robot is always in contact with a wall. When at least one sensor is activated, the robot will have a group of cases to choose from in regards to navigation. When no sensors are activated (i.e.,the robot is in an open space), the robot has a failsafe. When it has completed two full 360 clockwise revolutions, the robot will move forward until the front sensor hits an object. These prompt trees allow the robot to perpetually move forward and fully explore its environment, eventually leading it to the end.

  code fig.

```
 -- failsafe flags
   failSafe = {
       calls = 0,
       inOpenSpace = function (this) return this.calls >= 145 end
   }

-- actuation code
   failSafe.calls = failSafe.calls + 1
   if failSafe:inOpenSpace() then
       goForward(rightMotor, leftMotor)
   else
       ...
```

- Intelligence: The robot exhibits independent intelligence through the use of a *marker* object and its state. To summarize, a marker is placed at the entrance of the area. This marker is essentially a scanner, checking if the robot has found all of the disaster victims in the area. When the robot completes its path to exit the area, its floor sensor will detect the marker. If the robot successfully detects all individuals in the area (verified by checking the *count* state), the marker will turn green and print a message signaling that the mission to detect all individuals has been completed.

code fig.

```
        if object_name == 'marker' then
            count = getCount()
            if count == 2 then print('MISSION COMPLETE')
            sim.setShapeColor(marker, nil,
sim.colorcomponent_ambient_diffuse,{0,1,0})
            end
        end
```

Conversely, if the robot's floor sensor detects the marker and its count is below the expected amount, the marker won't react to being detected. This demonstrates that the robot can recognize whether it has completed its objective or not.

Task F

Further capabilities have been considered for this program and abandoned due to time requirements. An improvement for this robot's performance, which was the original navigation idea behind was to program a navigation algorithm that addresses objects that are in an open space (i.e., incontiguous from the inner border (i.e., the walls) of an area. In summary:

The robot would trace the entire inner border of an area. When the robot completed the path, it would check its state to see if all people had been accounted for. If not, the robot would shift inwards by an area roughly the same size as the robot itself and cycle through the path again. This pattern would repeat until either all subjects have been located or the epicentre of the area has been reached. This can be thought of as "filling up" the empty space of an empty, enclosed area.

This concept is essentially a uniformed search algorithm, or more specifically, a breadth-first search algorithm. The benefit is that the robot can exhaustively search an area without backtracking.

Reinforcement learning could have improved the robot's performance by adding time and distance incentives. Additional state, such as 'time passed' or 'distance traveled', for example, could prompt the robot to seek better paths when searching for disaster victims.

References

- Coppelia Robotics. (n.d.). *Properties*. *CoppeliaSim User Manual*. https://manual.coppeliarobotics.com/en/properties.htm