```
"""
File handles Robot movements and maths about it.
"""

import sys
import time
from math import sin, cos
from threading import Thread

from robotState import RobotState
import Trig
from path import Path
from Postman import postSpeed, getLaser
import draw
import os.path

from math import pi

class RobotMove:

    def __init__(self, pathName, padding, ifDraw):
        self.pathHandler = Path(pathName)
        self.state = RobotState(padding)
        self.threashHold = self.defineGoalTreshHold()
        if ifDraw:
            self.drawLog()

    def drawLog(self):
        t = Thread(target=draw.main, args=(self.state, self.pathHandler))
        t.start()

    def turnDirection(self, goalAngle):
        """returns what direction the robot should turn,
        returns -1 for clockwise, 1 for counter-clockwise"""
        """
            Returns what way is the closest to turn
            1 for counter-clockwise
            -1 for clockwise
        """
        currentAngle = self.state.getDirection()

        currentAngle %= 360
        goalAngle %= 360
        if currentAngle < goalAngle:
            if currentAngle + 180 > goalAngle:
                return 1
            else:
                return -1
        else:
            if currentAngle - 180 < goalAngle:
                return -1
            else:
                return 1

    def robotCanSee(self, goalx, goaly):
        """
            returns true if the robot can see this position from where it is
        """
        x,y = self.state.getPosition()
        dist = Trig.distanceToPoint(x, y, goalx, goaly)

        if dist < self.state.getActualSize() / 2: return True

        angle = Trig.angleToPoint(x, y, goalx, goaly)

        laserLength = self.state.getLaserLength(angle)

        return laserLength > dist
    def robotCanGo(self, gx, gy):
        """
            returns if the robot can go to a point in a straight line
            used to take shortcuts
        """
        x,y = self.state.getPosition()
        goalAngle = Trig.angleToPoint(x, y, gx, gy)
        dist = Trig.distanceToPoint(x, y, gx, gy)

        maxCount = round(dist / self.state.getActualSize())
        canBeCount = maxCount

        while canBeCount > 0:
            nowDist = dist * float(canBeCount) / float(maxCount)
            gx = cos(goalAngle) * nowDist
            gy = sin(goalAngle) * nowDist
            if not self.robotCanBe(gx, gy, goalAngle): return False

        return True

    def robotCanBe(self, gx, gy, goalAngle):
        """
            returns if we can be at a certain position, checks the whole front of the robot
            to make sure that it fits
        """
        x,y = self.state.getPosition()
        lx, ly = self.state.getCorners(gx, gy, goalAngle, 0)
        rx, ry = self.state.getCorners(gx, gy, goalAngle, 3)

        leftAngle = Trig.angleToPoint(x, y, lx, ly)
        rightAngle = Trig.angleToPoint(x, y, rx, ry)

        robotDirection = self.state.getDirection()
        leftIndex = Trig.radToLaser(leftAngle, robotDirection)
        rightIndex = Trig.radToLaser(rightAngle, robotDirection)

        distance = min(Trig.distanceToPoint(x, y, lx, ly), Trig.distanceToPoint(x, y, rx, r
y))

        for i in range(min(leftIndex, rightIndex), max(leftIndex, rightIndex) + 1):
            if self.state.getLaserLengthFromIndex(i) < distance: return False

        return True

    def purePursuit(self, goalx, goaly):
        """
            decides how to alter the angularspeed (and sometimes the linear speed)
            to get to a specifed goalPoint
            :param: linearSpeed, what default we use for the linear speed
        """
        x,y = self.state.getPosition()
        linearSpeed = self.state.getMaxSpeed()

        goalAngle = Trig.angleToPoint(x, y, goalx, goaly)

        dist = Trig.distanceToPoint(x, y, goalx, goaly)
        vb = goalAngle - self.state.getDirection()

        # xprim = cos(vb) * dist
        yprim = sin(vb) * dist
        # constant
        gammay = (2 * yprim) / (dist ** 2)

        angularSpeed = gammay * linearSpeed

        if abs(angularSpeed) > 2:
            # 2 times sign of angularspeed
            angularSpeed = 2 * Trig.sign(angularSpeed)
            linearSpeed = angularSpeed / gammay

        return angularSpeed, linearSpeed

    def safeTravel(self):
            """
                Returns what index the robot should aim for by checking if it can go there
by using pure pursuit
            """

            loop = True
            x, y = self.state.getPosition()

            currentIndex = self.pathHandler.getCurrentIndex()
            angle = self.state.getDirection()

            # while we can  to a point, move forward
            while loop and currentIndex < self.pathHandler.length() - 1:
                goalx, goaly = self.pathHandler.position(currentIndex)
                loop = self.robotCanSee(goalx, goaly)
                if loop: currentIndex += 1

            loop = True

            if currentIndex == self.pathHandler.length(): currentIndex -= 1

            # while we cant go to a point, try a point closer
            while (loop and currentIndex > 0):
                goalx, goaly = self.pathHandler.position(currentIndex)
                dist = Trig.distanceToPoint(x, y, goalx, goaly)
                vb = Trig.angleToPoint(x, y, goalx, goaly) - angle
                # print "x %.3f, y %.3f, goalx %.3f, goaly %.3f. Angle %.3f, Dist %.3f" % (
x,y,goalx,goaly,Trig.radToDeg(Trig.angleToPoint(x, y, goalx, goaly)),dist)

                radius = dist / (2 * (sin(vb)))

                loop = self.collisionAlongPath(goalx, goaly, radius)
                if loop: currentIndex -= 1

            self.pathHandler.setCurrentIndex(currentIndex)
            return currentIndex

    def defineGoalTreshHold(self):
        """
            Define what index we must have seen to start checking if we are close enough to
 the goal
            So we do not say we are in goal until we gone through the path
        """
        goalIndex = self.pathHandler.length() - 1
        index = goalIndex - 1

        gx, gy = self.pathHandler.position(goalIndex)
        ix, iy = self.pathHandler.position(index)

        while (Trig.distanceToPoint(gx, gy, ix, iy) <= 1):
            index -= 1
            ix, iy = self.pathHandler.position(index)

        return index + 1

    def inGoal(self):
        """
            returns if the robot is in goal
        """
        goalx, goaly = self.pathHandler.getLast()
        x, y = self.state.getPosition()
        currentIndex = self.pathHandler.getCurrentIndex()
        return Trig.distanceToPoint(x, y, goalx, goaly) < 1 and currentIndex > self.threash
Hold

    def collisionAlongPath(self, goalx, goaly, r):
        """
            Checks if the robot can see a pure-puruit-based path to the point specified
        """

        #Center of circle of turn
        robotAngle = self.state.getDirection()
        x,y = self.state.getPosition()

        cx, cy = Trig.getCenterOfTurn(r, robotAngle, x, y)
        #Angle from center to robot
        centerToRobot = Trig.angleToPoint(cx, cy, x, y)
        #Angle from center to goalPoint
        centerToGoal = Trig.angleToPoint(cx, cy, goalx, goaly)

        #The way we are going to turn
        turnDir = Trig.sign(r)

        #The difference in angle between centerToRobot and centerToGoal
        angleDiff = Trig.angleDifferenceDirection(centerToRobot, centerToGoal, turnDir)

        maxCount = round (r * angleDiff / self.state.getActualSize())
        canBeCount = maxCount

        while (canBeCount > 0):

            checkAngle = centerToRobot + angleDiff*(float(canBeCount)/float(maxCount))

            checkRobotAngle = checkAngle + (pi / 2) * turnDir
            checkX = cx + cos(checkAngle) * abs(r)
            checkY = cy + sin(checkAngle) * abs(r)

            if not self.robotCanBe(checkX, checkY, checkRobotAngle): return True

            canBeCount -=1

        return False

    def mainPure(self):
        """
            :param linearPreference:
            :param pathHandler: the handler for the path
            :param laserHandler: communicates with the draw-object about the laser
            finishes when the robot has reached the goal
        """

        start = time.time()
        sleepy = 0.1
        # so we wont sleep the first go-around
        end = start + sleepy

        self.state.update()
        okToGo = True
        startOfSimulation = time.time()

        while not self.inGoal():
            time.sleep(max(0, sleepy - (end - start)))

            self.state.update()

            # can we see
            if okToGo:
                self.safeTravel()

                goalx, goaly = self.pathHandler.position(self.pathHandler.getCurrentIndex()
)
                x, y = self.state.getPosition()
                goalAngle = Trig.angleToPoint(x, y, goalx, goaly)

            angleDifference = Trig.angleDifference(self.state.getDirection(), goalAngle)

            if angleDifference < (pi / 2) and okToGo:
                ang, lin = self.purePursuit(goalx, goaly)

                if angleDifference>(pi/3) and self.robotCanGo(goalx,goaly):
                    ang = Trig.sign(ang) * 2

            else:
                turnDir = self.turnDirection(goalAngle)
                if angleDifference>Trig.degToRad(30): ang = turnDir * 2
                else: ang =  turnDir * 2 * angleDifference / (pi / 2)

                lin = 0
                okToGo = angleDifference < Trig.degToRad(10)

            postSpeed(ang, lin)
            end = time.time()

        endOfSimulation = time.time()
        postSpeed(0,0)

        timeTook = endOfSimulation - startOfSimulation

        print "The robot took %.3f seconds to finish the course" % (timeTook)

if __name__ == '__main__':

    length = len(sys.argv) - 1
    helpString = "For more information about arguments, run robotMove with help as only arg
ument."

    if length<2:
        if length==0 or sys.argv[1]!="help":
            print "Too few arguments for robotMove\n" + helpString
            exit(1)
        else:
            print "--HELP-- (what arguments go where)\n\n" \
                  "Argument 1, pathName: the fileName of the path, must be of type .json\n\
n" \
                  "Argument 2, padding:  how far from walls the robot shall stay away,\n" \
                  "must be between 0 and 0.5, recommended padding: 0.4\n\n" \
                  "Argument 3, ifDraw:  if the robot should run with a graphical log of the
 robots desiscions\n" \
                  "and information about the path. true to draw, false to not draw\n" \
                  "(you can also skip the third argument to not draw))\n\n" \
                  "Example: src\\robotMove.py Path-file.json 0.4 True\n" \
                  "to run the simulation and draw it."
            exit(0)


    pathName = sys.argv[1]

    if os.path.exists(pathName)==False:
        print "Could not find file " + pathName + "\n" + helpString
        exit(1)

    padding = float(sys.argv[2])

    if padding<0 or padding>0.5:
        print "argument 2, Padding must be between 0 and 0.5\n" + helpString
        exit(1)

    ifDraw = False
    if length==3:
        if sys.argv[3]=="True" or sys.argv[3]=="true":
            ifDraw = True
        elif sys.argv[3]=="False" or sys.argv[3]=="false": ifDraw = False
        else:
            print "Incorrect third argument ifDraw\n" + helpString
            exit(1)

    elif length>3:
        print "Too many arguments for robotMove\n" + helpString
        exit(1)

    #Actual simulation starts
    ai = RobotMove(pathName, padding, ifDraw)
    ai.mainPure()





# This test i think is a good indicator that the laser works now,
# some times there is a diff when reading the angle upward cause the laser is not perfect a
nd sometimes measures the
# corridor and sometimes the inner room but that is to be expected
```