crawler.py (original)

```
# crawler.py
# ----------
# Licensing Information: Please do not distribute or publish solutions to this
# project. You are free to use and extend these projects for educational
# purposes. The Pacman AI projects were developed at UC Berkeley, primarily by
# John DeNero (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu).
# For more info, see http://inst.eecs.berkeley.edu/~cs188/sp09/pacman.html

#!/usr/bin/python
import math
from math import pi as PI
import time
import environment
import random

class CrawlingRobotEnvironment(environment.Environment):

   def __init__(self, crawlingRobot):

       self.crawlingRobot = crawlingRobot

       # The state is of the form (armAngle, handAngle)
       # where the angles are bucket numbers, not actual
       # degree measurements
       self.state = None
       self.nArmStates = 9
       self.nHandStates = 13

       # create a list of arm buckets and hand buckets to
       # discretize the state space
       minArmAngle,maxArmAngle = self.crawlingRobot.getMinAndMaxArmAngles()
       minHandAngle,maxHandAngle = self.crawlingRobot.getMinAndMaxHandAngles()
       armIncrement = (maxArmAngle - minArmAngle) / (self.nArmStates-1)
       handIncrement = (maxHandAngle - minHandAngle) / (self.nHandStates-1)
       self.armBuckets = [minArmAngle+(armIncrement*i) \
          for i in range(self.nArmStates)]
       self.handBuckets = [minHandAngle+(handIncrement*i) \
        for i in range(self.nHandStates)]

       # Reset
       self.reset()

   def getCurrentState(self):
       """
         Return the current state
         of the crawling robot
       """
       return self.state

   def getPossibleActions(self, state):
       """
         Returns possible actions
         for the states in the
         current state
       """

       actions = list()

       currArmBucket,currHandBucket = state
       if currArmBucket > 0: actions.append('arm-down')
       if currArmBucket < self.nArmStates-1: actions.append('arm-up')
       if currHandBucket > 0: actions.append('hand-down')
       if currHandBucket < self.nHandStates-1: actions.append('hand-up')
```

```
        return actions

    def doAction(self, action):
        """
          Perform the action and update
          the current state of the Environment
          and return the reward for the
          current state, the next state
          and the taken action.

          Returns:
            nextState, reward
        """
        nextState, reward =  None, None

        oldX,oldY = self.crawlingRobot.getRobotPosition()

        armBucket,handBucket = self.state
        armAngle,handAngle = self.crawlingRobot.getAngles()
        if action == 'arm-up':
          newArmAngle = self.armBuckets[armBucket+1]
          self.crawlingRobot.moveArm(newArmAngle)
          nextState = (armBucket+1,handBucket)
        if action == 'arm-down':
          newArmAngle = self.armBuckets[armBucket-1]
          self.crawlingRobot.moveArm(newArmAngle)
          nextState = (armBucket-1,handBucket)
        if action == 'hand-up':
          newHandAngle = self.handBuckets[handBucket+1]
          self.crawlingRobot.moveHand(newHandAngle)
          nextState = (armBucket,handBucket+1)
        if action == 'hand-down':
          newHandAngle = self.handBuckets[handBucket-1]
          self.crawlingRobot.moveHand(newHandAngle)
          nextState = (armBucket,handBucket-1)

        newX,newY = self.crawlingRobot.getRobotPosition()

        # a simple reward function
        reward = newX - oldX

        self.state = nextState
        return nextState, reward


    def reset(self):
        """
         Resets the Environment to the initial state
        """
        ## Initialize the state to be the middle
        ## value for each parameter e.g. if there are 13 and 19
        ## buckets for the arm and hand parameters, then the intial
        ## state should be (6,9)
        ##
        ## Also call self.crawlingRobot.setAngles()
        ## to the initial arm and hand angle

        armState = self.nArmStates/2
        handState = self.nHandStates/2
        self.state = armState,handState

self.crawlingRobot.setAngles(self.armBuckets[armState],self.handBuckets[handState])
        self.crawlingRobot.positions = [20,self.crawlingRobot.getRobotPosition()[0]]


class CrawlingRobot:

    def setAngles(self, armAngle, handAngle):
```

```
        """
            set the robot's arm and hand angles
            to the passed in values
        """
        self.armAngle = armAngle
        self.handAngle = handAngle

    def getAngles(self):
        """
            returns the pair of (armAngle, handAngle)
        """
        return self.armAngle, self.handAngle

    def getRobotPosition(self):
        """
            returns the (x,y) coordinates
            of the lower-left point of the
            robot
        """
        return self.robotPos

    def moveArm(self, newArmAngle):
        """
            move the robot arm to 'newArmAngle'
        """
        oldArmAngle = self.armAngle
        if newArmAngle > self.maxArmAngle:
            raise 'Crawling Robot: Arm Raised too high. Careful!'
        if newArmAngle < self.minArmAngle:
            raise 'Crawling Robot: Arm Raised too low. Careful!'
        disp = self.displacement(self.armAngle, self.handAngle,
                                 newArmAngle, self.handAngle)
        curXPos = self.robotPos[0]
        self.robotPos = (curXPos+disp, self.robotPos[1])
        self.armAngle = newArmAngle

        # Position and Velocity Sign Post
        self.positions.append(self.getRobotPosition()[0])
#         self.angleSums.append(abs(math.degrees(oldArmAngle)-
math.degrees(newArmAngle)))
        if len(self.positions) > 100:
            self.positions.pop(0)
 #           self.angleSums.pop(0)

    def moveHand(self, newHandAngle):
        """
            move the robot hand to 'newArmAngle'
        """
        oldHandAngle = self.handAngle

        if newHandAngle > self.maxHandAngle:
            raise 'Crawling Robot: Hand Raised too high. Careful!'
        if newHandAngle < self.minHandAngle:
            raise 'Crawling Robot: Hand Raised too low. Careful!'
        disp = self.displacement(self.armAngle, self.handAngle, self.armAngle,
newHandAngle)
        curXPos = self.robotPos[0]
        self.robotPos = (curXPos+disp, self.robotPos[1])
        self.handAngle = newHandAngle

        # Position and Velocity Sign Post
        self.positions.append(self.getRobotPosition()[0])
 #       self.angleSums.append(abs(math.degrees(oldHandAngle)-
math.degrees(newHandAngle)))
        if len(self.positions) > 100:
            self.positions.pop(0)
 #           self.angleSums.pop(0)

    def getMinAndMaxArmAngles(self):
```

```
        """
            get the lower- and upper- bound
            for the arm angles returns (min,max) pair
        """
        return self.minArmAngle, self.maxArmAngle

    def getMinAndMaxHandAngles(self):
        """
            get the lower- and upper- bound
            for the hand angles returns (min,max) pair
        """
        return self.minHandAngle, self.maxHandAngle

    def getRotationAngle(self):
        """
            get the current angle the
            robot body is rotated off the ground
        """
        armCos, armSin = self.__getCosAndSin(self.armAngle)
        handCos, handSin = self.__getCosAndSin(self.handAngle)
        x = self.armLength * armCos + self.handLength * handCos + self.robotWidth
        y = self.armLength * armSin + self.handLength * handSin + self.robotHeight
        if y < 0:
            return math.atan(-y/x)
        return 0.0


    ## You shouldn't need methods below here

    def __getCosAndSin(self, angle):
        return math.cos(angle), math.sin(angle)

    def displacement(self, oldArmDegree, oldHandDegree, armDegree, handDegree):

        oldArmCos, oldArmSin = self.__getCosAndSin(oldArmDegree)
        armCos, armSin = self.__getCosAndSin(armDegree)
        oldHandCos, oldHandSin = self.__getCosAndSin(oldHandDegree)
        handCos, handSin = self.__getCosAndSin(handDegree)

        xOld = self.armLength * oldArmCos + self.handLength * oldHandCos +
self.robotWidth
        yOld = self.armLength * oldArmSin + self.handLength * oldHandSin +
self.robotHeight

        x = self.armLength * armCos + self.handLength * handCos + self.robotWidth
        y = self.armLength * armSin + self.handLength * handSin + self.robotHeight

        if y < 0:
            if yOld <= 0:
                return math.sqrt(xOld*xOld + yOld*yOld) - math.sqrt(x*x + y*y)
            return (xOld - yOld*(x-xOld) / (y - yOld)) - math.sqrt(x*x + y*y)
        else:
            if yOld  >= 0:
                return 0.0
            return -(x - y * (xOld-x)/(yOld-y)) + math.sqrt(xOld*xOld + yOld*yOld)

        raise 'Never Should See This!'

    def draw(self, stepCount, stepDelay):
        x1, y1 = self.getRobotPosition()
        x1 = x1 % self.totWidth

        ## Check Lower Still on the ground
        if y1 != self.groundY:
            raise 'Flying Robot!!'

        rotationAngle = self.getRotationAngle()
        cosRot, sinRot = self.__getCosAndSin(rotationAngle)
```

```
        x2 = x1 + self.robotWidth * cosRot
        y2 = y1 - self.robotWidth * sinRot

        x3 = x1 - self.robotHeight * sinRot
        y3 = y1 - self.robotHeight * cosRot

        x4 = x3 + cosRot*self.robotWidth
        y4 = y3 - sinRot*self.robotWidth

        self.canvas.coords(self.robotBody,x1,y1,x2,y2,x4,y4,x3,y3)

        armCos, armSin = self.__getCosAndSin(rotationAngle+self.armAngle)
        xArm = x4 + self.armLength * armCos
        yArm = y4 - self.armLength * armSin

        self.canvas.coords(self.robotArm,x4,y4,xArm,yArm)

        handCos, handSin = self.__getCosAndSin(self.handAngle+rotationAngle)
        xHand = xArm + self.handLength * handCos
        yHand = yArm - self.handLength * handSin

        self.canvas.coords(self.robotHand,xArm,yArm,xHand,yHand)


        # Position and Velocity Sign Post
#        time = len(self.positions) + 0.5 * sum(self.angleSums)
#        velocity = (self.positions[-1]-self.positions[0]) / time
#        if len(self.positions) == 1: return
        steps = stepCount - self.lastStep
        if steps==0: return
 #        pos = self.positions[-1]
#        velocity = (pos - self.lastPos) / steps
  #       g = .9 ** (10 * stepDelay)
#        g = .99 ** steps
#        self.velAvg = g * self.velAvg + (1 - g) * velocity
 #        g = .999 ** steps
 #        self.velAvg2 = g * self.velAvg2 + (1 - g) * velocity
        pos = self.positions[-1]
        velocity = pos - self.positions[-2]
        vel2 = (pos - self.positions[0]) / len(self.positions)
        self.velAvg = .9 * self.velAvg + .1 * vel2
        velMsg = '100-step Avg Velocity: %.2f' % self.velAvg
#        velMsg2 = '1000-step Avg Velocity: %.2f' % self.velAvg2
        velocityMsg = 'Velocity: %.2f' % velocity
        positionMsg = 'Position: %2.f' % pos
        stepMsg = 'Step: %d' % stepCount
        if 'vel_msg' in dir(self):
            self.canvas.delete(self.vel_msg)
            self.canvas.delete(self.pos_msg)
            self.canvas.delete(self.step_msg)
            self.canvas.delete(self.velavg_msg)
 #            self.canvas.delete(self.velavg2_msg)
 #        self.velavg2_msg = self.canvas.create_text(850,190,text=velMsg2)
        self.velavg_msg = self.canvas.create_text(650,190,text=velMsg)
        self.vel_msg = self.canvas.create_text(450,190,text=velocityMsg)
        self.pos_msg = self.canvas.create_text(250,190,text=positionMsg)
        self.step_msg = self.canvas.create_text(50,190,text=stepMsg)
#        self.lastPos = pos
        self.lastStep = stepCount
#        self.lastVel = velocity

    def __init__(self, canvas):

        ## Canvas ##
        self.canvas = canvas
        self.velAvg = 0
#        self.velAvg2 = 0
#        self.lastPos = 0
```

```
        self.lastStep = 0
#         self.lastVel = 0

        ## Arm and Hand Degrees ##
        self.armAngle = self.oldArmDegree = 0.0
        self.handAngle = self.oldHandDegree = -PI/6

        self.maxArmAngle = PI/6
        self.minArmAngle = -PI/6

        self.maxHandAngle = 0
        self.minHandAngle = -(5.0/6.0) * PI

        ## Draw Ground ##
        self.totWidth = canvas.winfo_reqwidth()
        self.totHeight = canvas.winfo_reqheight()
        self.groundHeight = 40
        self.groundY = self.totHeight - self.groundHeight

        self.ground = canvas.create_rectangle(0,
            self.groundY,self.totWidth,self.totHeight, fill='blue')

        ## Robot Body ##
        self.robotWidth = 80
        self.robotHeight = 40
        self.robotPos = (20, self.groundY)
        self.robotBody = canvas.create_polygon(0,0,0,0,0,0,0,0, fill='green')

        ## Robot Arm ##
        self.armLength = 100
        self.robotArm = canvas.create_line(0,0,0,0,fill='orange',width=5)

        ## Robot Hand ##
        self.handLength = 40
        self.robotHand = canvas.create_line(0,0,0,0,fill='red',width=3)

        self.positions = [0,0]
#         self.angleSums = [0,0]


if __name__ == '__main__':
  from graphicsCrawlerDisplay import *
  run()
```