graphicsDisplay.py (original)

```
# graphicsDisplay.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

from graphicsUtils import *
import math, time
from game import Directions

###########################
#  GRAPHICS DISPLAY CODE  #
###########################

# Most code by Dan Klein and John Denero written or rewritten for cs188, UC
Berkeley.
# Some code from a Pacman implementation by LiveWires, and used / modified with
permission.

DEFAULT_GRID_SIZE = 30.0
INFO_PANE_HEIGHT = 35
BACKGROUND_COLOR = formatColor(0,0,0)
WALL_COLOR = formatColor(0.0/255.0, 51.0/255.0, 255.0/255.0)
INFO_PANE_COLOR = formatColor(.4,.4,0)
SCORE_COLOR = formatColor(.9, .9, .9)
PACMAN_OUTLINE_WIDTH = 2
PACMAN_CAPTURE_OUTLINE_WIDTH = 4

GHOST_COLORS = []
GHOST_COLORS.append(formatColor(.9,0,0)) # Red
GHOST_COLORS.append(formatColor(0,.3,.9)) # Blue
GHOST_COLORS.append(formatColor(.98,.41,.07)) # Orange
GHOST_COLORS.append(formatColor(.1,.75,.7)) # Green
GHOST_COLORS.append(formatColor(1.0,0.6,0.0)) # Yellow
GHOST_COLORS.append(formatColor(.4,0.13,0.91)) # Purple

TEAM_COLORS = GHOST_COLORS[:2]

GHOST_SHAPE = [
    ( 0,    0.3 ),
    ( 0.25, 0.75 ),
    ( 0.5,  0.3 ),
    ( 0.75, 0.75 ),
    ( 0.75, -0.5 ),
    ( 0.5,  -0.75 ),
    (-0.5,  -0.75 ),
    (-0.75, -0.5 ),
    (-0.75, 0.75 ),
    (-0.5,  0.3 ),
    (-0.25, 0.75 )
  ]
GHOST_SIZE = 0.65
SCARED_COLOR = formatColor(1,1,1)

GHOST_VEC_COLORS = map(colorToVector, GHOST_COLORS)

PACMAN_COLOR = formatColor(255.0/255.0,255.0/255.0,61.0/255)
PACMAN_SCALE = 0.5
#pacman_speed = 0.25

# Food
FOOD_COLOR = formatColor(1,1,1)
```

```
FOOD_SIZE = 0.1

# Laser
LASER_COLOR = formatColor(1,0,0)
LASER_SIZE = 0.02

# Capsule graphics
CAPSULE_COLOR = formatColor(1,1,1)
CAPSULE_SIZE = 0.25

# Drawing walls
WALL_RADIUS = 0.15

class InfoPane:
  def __init__(self, layout, gridSize):
    self.gridSize = gridSize
    self.width = (layout.width) * gridSize
    self.base = (layout.height + 1) * gridSize
    self.height = INFO_PANE_HEIGHT
    self.fontSize = 24
    self.textColor = PACMAN_COLOR
    self.drawPane()

  def toScreen(self, pos, y = None):
    """
      Translates a point relative from the bottom left of the info pane.
    """
    if y == None:
      x,y = pos
    else:
      x = pos

    x = self.gridSize + x # Margin
    y = self.base + y
    return x,y

  def drawPane(self):
    self.scoreText = text( self.toScreen(0, 0  ), self.textColor, "SCORE:    0",
"Times", self.fontSize, "bold")

  def initializeGhostDistances(self, distances):
    self.ghostDistanceText = []

    size = 20
    if self.width < 240:
      size = 12
    if self.width < 160:
      size = 10

    for i, d in enumerate(distances):
      t = text( self.toScreen(self.width/2 + self.width/8 * i, 0), GHOST_COLORS[i+1],
d, "Times", size, "bold")
      self.ghostDistanceText.append(t)

  def updateScore(self, score):
    changeText(self.scoreText, "SCORE: % 4d" % score)

  def setTeam(self, isBlue):
    text = "RED TEAM"
    if isBlue: text = "BLUE TEAM"
    self.teamText = text( self.toScreen(300, 0  ), self.textColor, text, "Times",
self.fontSize, "bold")

  def updateGhostDistances(self, distances):
    if len(distances) == 0: return
    if 'ghostDistanceText' not in dir(self): self.initializeGhostDistances(distances)
    else:
      for i, d in enumerate(distances):
        changeText(self.ghostDistanceText[i], d)
```

```
  def drawGhost(self):
    pass

  def drawPacman(self):
    pass

  def drawWarning(self):
    pass

  def clearIcon(self):
    pass

  def updateMessage(self, message):
    pass

  def clearMessage(self):
    pass


class PacmanGraphics:
  def __init__(self, zoom=1.0, frameTime=0.0, capture=False):
    self.have_window = 0
    self.currentGhostImages = {}
    self.pacmanImage = None
    self.zoom = zoom
    self.gridSize = DEFAULT_GRID_SIZE * zoom
    self.capture = capture
    self.frameTime = frameTime

  def initialize(self, state, isBlue = False):
    self.isBlue = isBlue
    self.startGraphics(state)

    # self.drawDistributions(state)
    self.distributionImages = None  # Initialized lazily
    self.drawStaticObjects(state)
    self.drawAgentObjects(state)

    # Information
    self.previousState = state

  def startGraphics(self, state):
    self.layout = state.layout
    layout = self.layout
    self.width = layout.width
    self.height = layout.height
    self.make_window(self.width, self.height)
    self.infoPane = InfoPane(layout, self.gridSize)
    self.currentState = layout

  def drawDistributions(self, state):
    walls = state.layout.walls
    dist = []
    for x in range(walls.width):
      distx = []
      dist.append(distx)
      for y in range(walls.height):
          ( screen_x, screen_y ) = self.to_screen( (x, y) )
          block = square( (screen_x, screen_y),
                          0.5 * self.gridSize,
                          color = BACKGROUND_COLOR,
                          filled = 1, behind=2)
          distx.append(block)
    self.distributionImages = dist

  def drawStaticObjects(self, state):
    layout = self.layout
    self.drawWalls(layout.walls)
```

```
    self.food = self.drawFood(layout.food)
    self.capsules = self.drawCapsules(layout.capsules)
    refresh()

  def drawAgentObjects(self, state):
    self.agentImages = [] # (agentState, image)
    for index, agent in enumerate(state.agentStates):
      if agent.isPacman:
        image = self.drawPacman(agent, index)
        self.agentImages.append( (agent, image) )
      else:
        image = self.drawGhost(agent, index)
        self.agentImages.append( (agent, image) )
    refresh()

  def swapImages(self, agentIndex, newState):
    """
      Changes an image from a ghost to a pacman or vis versa (for capture)
    """
    prevState, prevImage = self.agentImages[agentIndex]
    for item in prevImage: remove_from_screen(item)
    if newState.isPacman:
      image = self.drawPacman(newState, agentIndex)
      self.agentImages[agentIndex] = (newState, image )
    else:
      image = self.drawGhost(newState, agentIndex)
      self.agentImages[agentIndex] = (newState, image )
    refresh()

  def update(self, newState):
    agentIndex = newState._agentMoved
    agentState = newState.agentStates[agentIndex]

    if self.agentImages[agentIndex][0].isPacman != agentState.isPacman:
self.swapImages(agentIndex, agentState)
    prevState, prevImage = self.agentImages[agentIndex]
    if agentState.isPacman:
      self.animatePacman(agentState, prevState, prevImage)
    else:
      self.moveGhost(agentState, agentIndex, prevState, prevImage)
    self.agentImages[agentIndex] = (agentState, prevImage)

    if newState._foodEaten != None:
      self.removeFood(newState._foodEaten, self.food)
    if newState._capsuleEaten != None:
      self.removeCapsule(newState._capsuleEaten, self.capsules)
    self.infoPane.updateScore(newState.score)
    if 'ghostDistances' in dir(newState):
      self.infoPane.updateGhostDistances(newState.ghostDistances)

  def make_window(self, width, height):
    grid_width = (width-1) * self.gridSize
    grid_height = (height-1) * self.gridSize
    screen_width = 2*self.gridSize + grid_width
    screen_height = 2*self.gridSize + grid_height + INFO_PANE_HEIGHT

    begin_graphics(screen_width,
                   screen_height,
                   BACKGROUND_COLOR,
                   "CS188 Pacman")

  def drawPacman(self, pacman, index):
    position = self.getPosition(pacman)
    screen_point = self.to_screen(position)
    endpoints = self.getEndpoints(self.getDirection(pacman))

    width = PACMAN_OUTLINE_WIDTH
    outlineColor = PACMAN_COLOR
    fillColor = PACMAN_COLOR
```

```
    if self.capture:
      outlineColor = TEAM_COLORS[index % 2]
      fillColor = GHOST_COLORS[index]
      width = PACMAN_CAPTURE_OUTLINE_WIDTH

    return [circle(screen_point, PACMAN_SCALE * self.gridSize,
                   fillColor = fillColor, outlineColor = outlineColor,
                   endpoints = endpoints,
                   width = width)]

  def getEndpoints(self, direction, position=(0,0)):
    x, y = position
    pos = x - int(x) + y - int(y)
    width = 30 + 80 * math.sin(math.pi* pos)

    delta = width / 2
    if (direction == 'West'):
      endpoints = (180+delta, 180-delta)
    elif (direction == 'North'):
      endpoints = (90+delta, 90-delta)
    elif (direction == 'South'):
      endpoints = (270+delta, 270-delta)
    else:
      endpoints = (0+delta, 0-delta)
    return endpoints

  def movePacman(self, position, direction, image):
    screenPosition = self.to_screen(position)
    endpoints = self.getEndpoints( direction, position )
    r = PACMAN_SCALE * self.gridSize
    moveCircle(image[0], screenPosition, r, endpoints)
    refresh()

  def animatePacman(self, pacman, prevPacman, image):
    if self.frameTime < 0:
      print 'Press any key to step forward, "q" to play'
      keys = wait_for_keys()
      if 'q' in keys:
        self.frameTime = 0.1
    if self.frameTime > 0.01 or self.frameTime < 0:
      start = time.time()
      fx, fy = self.getPosition(prevPacman)
      px, py = self.getPosition(pacman)
      frames = 4.0
      for i in range(1,int(frames) + 1):
        pos = px*i/frames + fx*(frames-i)/frames, py*i/frames + fy*(frames-i)/frames
        self.movePacman(pos, self.getDirection(pacman), image)
        refresh()
        sleep(abs(self.frameTime) / frames)
    else:
      self.movePacman(self.getPosition(pacman), self.getDirection(pacman), image)
    refresh()

  def getGhostColor(self, ghost, ghostIndex):
    if ghost.scaredTimer > 0:
      return SCARED_COLOR
    else:
      return GHOST_COLORS[ghostIndex]

  def drawGhost(self, ghost, agentIndex):
    pos = self.getPosition(ghost)
    dir = self.getDirection(ghost)
    (screen_x, screen_y) = (self.to_screen(pos) )
    coords = []
    for (x, y) in GHOST_SHAPE:
      coords.append((x*self.gridSize*GHOST_SIZE + screen_x,
y*self.gridSize*GHOST_SIZE + screen_y))
```

```
    colour = self.getGhostColor(ghost, agentIndex)
    body = polygon(coords, colour, filled = 1)
    WHITE = formatColor(1.0, 1.0, 1.0)
    BLACK = formatColor(0.0, 0.0, 0.0)

    dx = 0
    dy = 0
    if dir == 'North':
      dy = -0.2
    if dir == 'South':
      dy = 0.2
    if dir == 'East':
      dx = 0.2
    if dir == 'West':
      dx = -0.2
    leftEye = circle((screen_x+self.gridSize*GHOST_SIZE*(-0.3+dx/1.5), screen_y-
self.gridSize*GHOST_SIZE*(0.3-dy/1.5)), self.gridSize*GHOST_SIZE*0.2, WHITE, WHITE)
    rightEye = circle((screen_x+self.gridSize*GHOST_SIZE*(0.3+dx/1.5), screen_y-
self.gridSize*GHOST_SIZE*(0.3-dy/1.5)), self.gridSize*GHOST_SIZE*0.2, WHITE, WHITE)
    leftPupil = circle((screen_x+self.gridSize*GHOST_SIZE*(-0.3+dx), screen_y-
self.gridSize*GHOST_SIZE*(0.3-dy)), self.gridSize*GHOST_SIZE*0.08, BLACK, BLACK)
    rightPupil = circle((screen_x+self.gridSize*GHOST_SIZE*(0.3+dx), screen_y-
self.gridSize*GHOST_SIZE*(0.3-dy)), self.gridSize*GHOST_SIZE*0.08, BLACK, BLACK)
    ghostImageParts = []
    ghostImageParts.append(body)
    ghostImageParts.append(leftEye)
    ghostImageParts.append(rightEye)
    ghostImageParts.append(leftPupil)
    ghostImageParts.append(rightPupil)

    return ghostImageParts

  def moveEyes(self, pos, dir, eyes):
    (screen_x, screen_y) = (self.to_screen(pos))
    dx = 0
    dy = 0
    if dir == 'North':
      dy = -0.2
    if dir == 'South':
      dy = 0.2
    if dir == 'East':
      dx = 0.2
    if dir == 'West':
      dx = -0.2
    moveCircle(eyes[0],(screen_x+self.gridSize*GHOST_SIZE*(-0.3+dx/1.5), screen_y-
self.gridSize*GHOST_SIZE*(0.3-dy/1.5)), self.gridSize*GHOST_SIZE*0.2)
    moveCircle(eyes[1],(screen_x+self.gridSize*GHOST_SIZE*(0.3+dx/1.5), screen_y-
self.gridSize*GHOST_SIZE*(0.3-dy/1.5)), self.gridSize*GHOST_SIZE*0.2)
    moveCircle(eyes[2],(screen_x+self.gridSize*GHOST_SIZE*(-0.3+dx), screen_y-
self.gridSize*GHOST_SIZE*(0.3-dy)), self.gridSize*GHOST_SIZE*0.08)
    moveCircle(eyes[3],(screen_x+self.gridSize*GHOST_SIZE*(0.3+dx), screen_y-
self.gridSize*GHOST_SIZE*(0.3-dy)), self.gridSize*GHOST_SIZE*0.08)

  def moveGhost(self, ghost, ghostIndex, prevGhost, ghostImageParts):
    old_x, old_y = self.to_screen(self.getPosition(prevGhost))
    new_x, new_y = self.to_screen(self.getPosition(ghost))
    delta = new_x - old_x, new_y - old_y

    for ghostImagePart in ghostImageParts:
      move_by(ghostImagePart, delta)
    refresh()

    if ghost.scaredTimer > 0:
      color = SCARED_COLOR
    else:
      color = GHOST_COLORS[ghostIndex]
    edit(ghostImageParts[0], ('fill', color), ('outline', color))
    self.moveEyes(self.getPosition(ghost), self.getDirection(ghost),
ghostImageParts[-4:])
```

```
    refresh()

  def getPosition(self, agentState):
    if agentState.configuration == None: return (-1000, -1000)
    return agentState.getPosition()

  def getDirection(self, agentState):
    if agentState.configuration == None: return Directions.STOP
    return agentState.configuration.getDirection()

  def finish(self):
    end_graphics()

  def to_screen(self, point):
    ( x, y ) = point
    #y = self.height - y
    x = (x + 1)*self.gridSize
    y = (self.height  - y)*self.gridSize
    return ( x, y )

  # Fixes some TK issue with off-center circles
  def to_screen2(self, point):
    ( x, y ) = point
    #y = self.height - y
    x = (x + 1)*self.gridSize
    y = (self.height  - y)*self.gridSize
    return ( x, y )

  def drawWalls(self, wallMatrix):
    wallColor = WALL_COLOR
    for xNum, x in enumerate(wallMatrix):
      if self.capture and (xNum * 2) < wallMatrix.width: wallColor = TEAM_COLORS[0]
      if self.capture and (xNum * 2) >= wallMatrix.width: wallColor = TEAM_COLORS[1]

      for yNum, cell in enumerate(x):
        if cell: # There's a wall here
          pos = (xNum, yNum)
          screen = self.to_screen(pos)
          screen2 = self.to_screen2(pos)

          # draw each quadrant of the square based on adjacent walls
          wIsWall = self.isWall(xNum-1, yNum, wallMatrix)
          eIsWall = self.isWall(xNum+1, yNum, wallMatrix)
          nIsWall = self.isWall(xNum, yNum+1, wallMatrix)
          sIsWall = self.isWall(xNum, yNum-1, wallMatrix)
          nwIsWall = self.isWall(xNum-1, yNum+1, wallMatrix)
          swIsWall = self.isWall(xNum-1, yNum-1, wallMatrix)
          neIsWall = self.isWall(xNum+1, yNum+1, wallMatrix)
          seIsWall = self.isWall(xNum+1, yNum-1, wallMatrix)

          # NE quadrant
          if (not nIsWall) and (not eIsWall):
            # inner circle
            circle(screen2, WALL_RADIUS * self.gridSize, wallColor, wallColor,
(0,91), 'arc')
          if (nIsWall) and (not eIsWall):
            # vertical line
            line(add(screen, (self.gridSize*WALL_RADIUS, 0)), add(screen,
(self.gridSize*WALL_RADIUS, self.gridSize*(-0.5)-1)), wallColor)
          if (not nIsWall) and (eIsWall):
            # horizontal line
            line(add(screen, (0, self.gridSize*(-1)*WALL_RADIUS)), add(screen,
(self.gridSize*0.5+1, self.gridSize*(-1)*WALL_RADIUS)), wallColor)
          if (nIsWall) and (eIsWall) and (not neIsWall):
            # outer circle
            circle(add(screen2, (self.gridSize*2*WALL_RADIUS, self.gridSize*(-
2)*WALL_RADIUS)), WALL_RADIUS * self.gridSize-1, wallColor, wallColor, (180,271),
'arc')
            line(add(screen, (self.gridSize*2*WALL_RADIUS-1, self.gridSize*(-
```

```
1)*WALL_RADIUS)), add(screen, (self.gridSize*0.5+1, self.gridSize*(-1)*WALL_RADIUS)),
wallColor)
            line(add(screen, (self.gridSize*WALL_RADIUS, self.gridSize*(-
2)*WALL_RADIUS+1)), add(screen, (self.gridSize*WALL_RADIUS, self.gridSize*(-0.5))),
wallColor)

          # NW quadrant
          if (not nIsWall) and (not wIsWall):
            # inner circle
            circle(screen2, WALL_RADIUS * self.gridSize, wallColor, wallColor,
(90,181), 'arc')
          if (nIsWall) and (not wIsWall):
            # vertical line
            line(add(screen, (self.gridSize*(-1)*WALL_RADIUS, 0)), add(screen,
(self.gridSize*(-1)*WALL_RADIUS, self.gridSize*(-0.5)-1)), wallColor)
          if (not nIsWall) and (wIsWall):
            # horizontal line
            line(add(screen, (0, self.gridSize*(-1)*WALL_RADIUS)), add(screen,
(self.gridSize*(-0.5)-1, self.gridSize*(-1)*WALL_RADIUS)), wallColor)
          if (nIsWall) and (wIsWall) and (not nwIsWall):
            # outer circle
            circle(add(screen2, (self.gridSize*(-2)*WALL_RADIUS, self.gridSize*(-
2)*WALL_RADIUS)), WALL_RADIUS * self.gridSize-1, wallColor, wallColor, (270,361),
'arc')
            line(add(screen, (self.gridSize*(-2)*WALL_RADIUS+1, self.gridSize*(-
1)*WALL_RADIUS)), add(screen, (self.gridSize*(-0.5), self.gridSize*(-
1)*WALL_RADIUS)), wallColor)
            line(add(screen, (self.gridSize*(-1)*WALL_RADIUS, self.gridSize*(-
2)*WALL_RADIUS+1)), add(screen, (self.gridSize*(-1)*WALL_RADIUS, self.gridSize*(-
0.5))), wallColor)

          # SE quadrant
          if (not sIsWall) and (not eIsWall):
            # inner circle
            circle(screen2, WALL_RADIUS * self.gridSize, wallColor, wallColor,
(270,361), 'arc')
          if (sIsWall) and (not eIsWall):
            # vertical line
            line(add(screen, (self.gridSize*WALL_RADIUS, 0)), add(screen,
(self.gridSize*WALL_RADIUS, self.gridSize*(0.5)+1)), wallColor)
          if (not sIsWall) and (eIsWall):
            # horizontal line
            line(add(screen, (0, self.gridSize*(1)*WALL_RADIUS)), add(screen,
(self.gridSize*0.5+1, self.gridSize*(1)*WALL_RADIUS)), wallColor)
          if (sIsWall) and (eIsWall) and (not seIsWall):
            # outer circle
            circle(add(screen2, (self.gridSize*2*WALL_RADIUS,
self.gridSize*(2)*WALL_RADIUS)), WALL_RADIUS * self.gridSize-1, wallColor, wallColor,
(90,181), 'arc')
            line(add(screen, (self.gridSize*2*WALL_RADIUS-1,
self.gridSize*(1)*WALL_RADIUS)), add(screen, (self.gridSize*0.5,
self.gridSize*(1)*WALL_RADIUS)), wallColor)
            line(add(screen, (self.gridSize*WALL_RADIUS,
self.gridSize*(2)*WALL_RADIUS-1)), add(screen, (self.gridSize*WALL_RADIUS,
self.gridSize*(0.5))), wallColor)

          # SW quadrant
          if (not sIsWall) and (not wIsWall):
            # inner circle
            circle(screen2, WALL_RADIUS * self.gridSize, wallColor, wallColor,
(180,271), 'arc')
          if (sIsWall) and (not wIsWall):
            # vertical line
            line(add(screen, (self.gridSize*(-1)*WALL_RADIUS, 0)), add(screen,
(self.gridSize*(-1)*WALL_RADIUS, self.gridSize*(0.5)+1)), wallColor)
          if (not sIsWall) and (wIsWall):
            # horizontal line
            line(add(screen, (0, self.gridSize*(1)*WALL_RADIUS)), add(screen,
(self.gridSize*(-0.5)-1, self.gridSize*(1)*WALL_RADIUS)), wallColor)
```

```
          if (sIsWall) and (wIsWall) and (not swIsWall):
            # outer circle
            circle(add(screen2, (self.gridSize*(-2)*WALL_RADIUS,
self.gridSize*(2)*WALL_RADIUS)), WALL_RADIUS * self.gridSize-1, wallColor, wallColor,
(0,91), 'arc')
            line(add(screen, (self.gridSize*(-2)*WALL_RADIUS+1,
self.gridSize*(1)*WALL_RADIUS)), add(screen, (self.gridSize*(-0.5),
self.gridSize*(1)*WALL_RADIUS)), wallColor)
            line(add(screen, (self.gridSize*(-1)*WALL_RADIUS,
self.gridSize*(2)*WALL_RADIUS-1)), add(screen, (self.gridSize*(-1)*WALL_RADIUS,
self.gridSize*(0.5))), wallColor)

  def isWall(self, x, y, walls):
    if x < 0 or y < 0:
      return False
    if x >= walls.width or y >= walls.height:
      return False
    return walls[x][y]

  def drawFood(self, foodMatrix ):
    foodImages = []
    color = FOOD_COLOR
    for xNum, x in enumerate(foodMatrix):
      if self.capture and (xNum * 2) <= foodMatrix.width: color = TEAM_COLORS[0]
      if self.capture and (xNum * 2) > foodMatrix.width: color = TEAM_COLORS[1]
      imageRow = []
      foodImages.append(imageRow)
      for yNum, cell in enumerate(x):
        if cell: # There's food here
          screen = self.to_screen((xNum, yNum ))
          dot = circle( screen,
                        FOOD_SIZE * self.gridSize,
                        outlineColor = color, fillColor = color,
                        width = 1)
          imageRow.append(dot)
        else:
          imageRow.append(None)
    return foodImages

  def drawCapsules(self, capsules ):
    capsuleImages = {}
    for capsule in capsules:
      ( screen_x, screen_y ) = self.to_screen(capsule)
      dot = circle( (screen_x, screen_y),
                    CAPSULE_SIZE * self.gridSize,
                    outlineColor = CAPSULE_COLOR,
                    fillColor = CAPSULE_COLOR,
                    width = 1)
      capsuleImages[capsule] = dot
    return capsuleImages

  def removeFood(self, cell, foodImages ):
    x, y = cell
    remove_from_screen(foodImages[x][y])

  def removeCapsule(self, cell, capsuleImages ):
    x, y = cell
    remove_from_screen(capsuleImages[(x, y)])

  def drawExpandedCells(self, cells):
    """
    Draws an overlay of expanded grid positions for search agents
    """
    n = float(len(cells))
    baseColor = [1.0, 0.0, 0.0]
    self.clearExpandedCells()
    self.expandedCells = []
    for k, cell in enumerate(cells):
       screenPos = self.to_screen( cell)
```

```
        cellColor = formatColor(*[(n-k) * c * .5 / n + .25 for c in baseColor])
        block = square(screenPos,
                 0.5 * self.gridSize,
                 color = cellColor,
                 filled = 1, behind=2)
        self.expandedCells.append(block)
        if self.frameTime < 0:
          refresh()

    def clearExpandedCells(self):
      if 'expandedCells' in dir(self) and len(self.expandedCells) > 0:
        for cell in self.expandedCells:
          remove_from_screen(cell)


    def updateDistributions(self, distributions):
      "Draws an agent's belief distributions"
      if self.distributionImages == None:
        self.drawDistributions(self.previousState)
      for x in range(len(self.distributionImages)):
        for y in range(len(self.distributionImages[0])):
          image = self.distributionImages[x][y]
          weights = [dist[ (x,y) ] for dist in distributions]

          if sum(weights) != 0:
            pass
          # Fog of war
          color = [0.0,0.0,0.0]
          colors = GHOST_VEC_COLORS[1:] # With Pacman
          if self.capture: colors = GHOST_VEC_COLORS
          for weight, gcolor in zip(weights, colors):
            color = [min(1.0, c + 0.95 * g * weight ** .3) for c,g in zip(color,
gcolor)]
          changeColor(image, formatColor(*color))
      refresh()

class FirstPersonPacmanGraphics(PacmanGraphics):
    def __init__(self, zoom = 1.0, showGhosts = True, capture = False, frameTime=0):
      PacmanGraphics.__init__(self, zoom, frameTime=frameTime)
      self.showGhosts = showGhosts
      self.capture = capture

    def initialize(self, state, isBlue = False):

      self.isBlue = isBlue
      PacmanGraphics.startGraphics(self, state)
      # Initialize distribution images
      walls = state.layout.walls
      dist = []
      self.layout = state.layout

      # Draw the rest
      self.distributionImages = None  # initialize lazily
      self.drawStaticObjects(state)
      self.drawAgentObjects(state)

      # Information
      self.previousState = state

    def lookAhead(self, config, state):
      if config.getDirection() == 'Stop':
        return
      else:
        pass
        # Draw relevant ghosts
        allGhosts = state.getGhostStates()
        visibleGhosts = state.getVisibleGhosts()
        for i, ghost in enumerate(allGhosts):
          if ghost in visibleGhosts:
```

```
          self.drawGhost(ghost, i)
        else:
          self.currentGhostImages[i] = None

  def getGhostColor(self, ghost, ghostIndex):
    return GHOST_COLORS[ghostIndex]

  def getPosition(self, ghostState):
    if not self.showGhosts and not ghostState.isPacman and ghostState.getPosition()
[1] > 1:
      return (-1000, -1000)
    else:
      return PacmanGraphics.getPosition(self, ghostState)

def add(x, y):
  return (x[0] + y[0], x[1] + y[1])


# Saving graphical output
# -----------------------
# Note: to make an animated gif from this postscript output, try the command:
# convert -delay 7 -loop 1 -compress lzw -layers optimize frame* out.gif
# convert is part of imagemagick (freeware)

SAVE_POSTSCRIPT = False
POSTSCRIPT_OUTPUT_DIR = 'frames'
FRAME_NUMBER = 0
import os

def saveFrame():
  "Saves the current graphical output as a postscript file"
  global SAVE_POSTSCRIPT, FRAME_NUMBER, POSTSCRIPT_OUTPUT_DIR
  if not SAVE_POSTSCRIPT: return
  if not os.path.exists(POSTSCRIPT_OUTPUT_DIR): os.mkdir(POSTSCRIPT_OUTPUT_DIR)
  name = os.path.join(POSTSCRIPT_OUTPUT_DIR, 'frame_%08d.ps' % FRAME_NUMBER)
  FRAME_NUMBER += 1
  writePostscript(name) # writes the current canvas
```