# Star Pusher (a Sokoban clone)

# By Al Sweigart al@inventwithpython.com

# http://inventwithpython.com/pygame

# Released under a "Simplified BSD" license

import random, sys, copy, os, pygame

from pygame.locals import \*

FPS = 30 # frames per second to update the screen

WINWIDTH = 1024 # width of the program's window, in pixels

WINHEIGHT = 768 # height in pixels

HALF\_WINWIDTH = int(WINWIDTH / 2)

HALF\_WINHEIGHT = int(WINHEIGHT / 2)

# The total width and height of each tile in pixels.

TILEWIDTH = 50

TILEHEIGHT = 85

TILEFLOORHEIGHT = 40

CAM\_MOVE\_SPEED = 3 # how many pixels per frame the camera moves

# The percentage of outdoor tiles that have additional

# decoration on them, such as a tree or rock.

OUTSIDE\_DECORATION\_PCT = 20

BRIGHTBLUE = ( 0, 170, 255)

WHITE = (255, 255, 255)

BGCOLOR = BRIGHTBLUE

TEXTCOLOR = WHITE

UP = 'up'

DOWN = 'down'

LEFT = 'left'

RIGHT = 'right'

def main():

global FPSCLOCK, DISPLAYSURF, IMAGESDICT, TILEMAPPING, OUTSIDEDECOMAPPING, BASICFONT, PLAYERIMAGES, currentImage

# Pygame initialization and basic set up of the global variables.

pygame.init()

FPSCLOCK = pygame.time.Clock()

# Because the Surface object stored in DISPLAYSURF was returned

# from the pygame.display.set\_mode() function, this is the

# Surface object that is drawn to the actual computer screen

# when pygame.display.update() is called.

DISPLAYSURF = pygame.display.set\_mode((WINWIDTH, WINHEIGHT))

pygame.display.set\_caption('Star Pusher')

BASICFONT = pygame.font.Font('freesansbold.ttf', 18)

# A global dict value that will contain all the Pygame

# Surface objects returned by pygame.image.load().

IMAGESDICT = {'uncovered goal': pygame.image.load('RedSelector.png'),

'covered goal': pygame.image.load('Selector.png'),

'star': pygame.image.load('Star.png'),

'corner': pygame.image.load('Wall\_Block\_Tall.png'),

'wall': pygame.image.load('Wood\_Block\_Tall.png'),

'inside floor': pygame.image.load('Plain\_Block.png'),

'outside floor': pygame.image.load('Grass\_Block.png'),

'title': pygame.image.load('star\_title.png'),

'solved': pygame.image.load('star\_solved.png'),

'princess': pygame.image.load('princess.png'),

'boy': pygame.image.load('boy.png'),

'catgirl': pygame.image.load('catgirl.png'),

'horngirl': pygame.image.load('horngirl.png'),

'pinkgirl': pygame.image.load('pinkgirl.png'),

'rock': pygame.image.load('Rock.png'),

'short tree': pygame.image.load('Tree\_Short.png'),

'tall tree': pygame.image.load('Tree\_Tall.png'),

'ugly tree': pygame.image.load('Tree\_Ugly.png')}

# These dict values are global, and map the character that appears

# in the level file to the Surface object it represents.

TILEMAPPING = {'x': IMAGESDICT['corner'],

'#': IMAGESDICT['wall'],

'o': IMAGESDICT['inside floor'],

' ': IMAGESDICT['outside floor']}

OUTSIDEDECOMAPPING = {'1': IMAGESDICT['rock'],

'2': IMAGESDICT['short tree'],

'3': IMAGESDICT['tall tree'],

'4': IMAGESDICT['ugly tree']}

# PLAYERIMAGES is a list of all possible characters the player can be.

# currentImage is the index of the player's current player image.

currentImage = 0

PLAYERIMAGES = [IMAGESDICT['princess'],

IMAGESDICT['boy'],

IMAGESDICT['catgirl'],

IMAGESDICT['horngirl'],

IMAGESDICT['pinkgirl']]

startScreen() # show the title screen until the user presses a key

# Read in the levels from the text file. See the readLevelsFile() for

# details on the format of this file and how to make your own levels.

levels = readLevelsFile('starPusherLevels.txt')

currentLevelIndex = 0

# The main game loop. This loop runs a single level, when the user

# finishes that level, the next/previous level is loaded.

while True: # main game loop

# Run the level to actually start playing the game:

result = runLevel(levels, currentLevelIndex)

if result in ('solved', 'next'):

# Go to the next level.

currentLevelIndex += 1

if currentLevelIndex >= len(levels):

# If there are no more levels, go back to the first one.

currentLevelIndex = 0

elif result == 'back':

# Go to the previous level.

currentLevelIndex -= 1

if currentLevelIndex < 0:

# If there are no previous levels, go to the last one.

currentLevelIndex = len(levels)-1

elif result == 'reset':

pass # Do nothing. Loop re-calls runLevel() to reset the level

def runLevel(levels, levelNum):

global currentImage

levelObj = levels[levelNum]

mapObj = decorateMap(levelObj['mapObj'], levelObj['startState']['player'])

gameStateObj = copy.deepcopy(levelObj['startState'])

mapNeedsRedraw = True # set to True to call drawMap()

levelSurf = BASICFONT.render('Level %s of %s' % (levelNum + 1, len(levels)), 1, TEXTCOLOR)

levelRect = levelSurf.get\_rect()

levelRect.bottomleft = (20, WINHEIGHT - 35)

mapWidth = len(mapObj) \* TILEWIDTH

mapHeight = (len(mapObj[0]) - 1) \* TILEFLOORHEIGHT + TILEHEIGHT

MAX\_CAM\_X\_PAN = abs(HALF\_WINHEIGHT - int(mapHeight / 2)) + TILEWIDTH

MAX\_CAM\_Y\_PAN = abs(HALF\_WINWIDTH - int(mapWidth / 2)) + TILEHEIGHT

levelIsComplete = False

# Track how much the camera has moved:

cameraOffsetX = 0

cameraOffsetY = 0

# Track if the keys to move the camera are being held down:

cameraUp = False

cameraDown = False

cameraLeft = False

cameraRight = False

while True: # main game loop

# Reset these variables:

playerMoveTo = None

keyPressed = False

for event in pygame.event.get(): # event handling loop

if event.type == QUIT:

# Player clicked the "X" at the corner of the window.

terminate()

elif event.type == KEYDOWN:

# Handle key presses

keyPressed = True

if event.key == K\_LEFT:

playerMoveTo = LEFT

elif event.key == K\_RIGHT:

playerMoveTo = RIGHT

elif event.key == K\_UP:

playerMoveTo = UP

elif event.key == K\_DOWN:

playerMoveTo = DOWN

# Set the camera move mode.

elif event.key == K\_q:

cameraLeft = True

elif event.key == K\_d:

cameraRight = True

elif event.key == K\_z:

cameraUp = True

elif event.key == K\_s:

cameraDown = True

elif event.key == K\_n:

return 'next'

elif event.key == K\_b:

return 'back'

elif event.key == K\_ESCAPE:

terminate() # Esc key quits.

elif event.key == K\_BACKSPACE:

return 'reset' # Reset the level.

elif event.key == K\_p:

# Change the player image to the next one.

currentImage += 1

if currentImage >= len(PLAYERIMAGES):

# After the last player image, use the first one.

currentImage = 0

mapNeedsRedraw = True

elif event.type == KEYUP:

# Unset the camera move mode.

if event.key == K\_q:

cameraLeft = False

elif event.key == K\_d:

cameraRight = False

elif event.key == K\_z:

cameraUp = False

elif event.key == K\_s:

cameraDown = False

if playerMoveTo != None and not levelIsComplete:

# If the player pushed a key to move, make the move

# (if possible) and push any stars that are pushable.

moved = makeMove(mapObj, gameStateObj, playerMoveTo)

if moved:

# increment the step counter.

gameStateObj['stepCounter'] += 1

mapNeedsRedraw = True

if isLevelFinished(levelObj, gameStateObj):

# level is solved, we should show the "Solved!" image.

levelIsComplete = True

keyPressed = False

DISPLAYSURF.fill(BGCOLOR)

if mapNeedsRedraw:

mapSurf = drawMap(mapObj, gameStateObj, levelObj['goals'])

mapNeedsRedraw = False

if cameraUp and cameraOffsetY < MAX\_CAM\_X\_PAN:

cameraOffsetY += CAM\_MOVE\_SPEED

elif cameraDown and cameraOffsetY > -MAX\_CAM\_X\_PAN:

cameraOffsetY -= CAM\_MOVE\_SPEED

if cameraLeft and cameraOffsetX < MAX\_CAM\_Y\_PAN:

cameraOffsetX += CAM\_MOVE\_SPEED

elif cameraRight and cameraOffsetX > -MAX\_CAM\_Y\_PAN:

cameraOffsetX -= CAM\_MOVE\_SPEED

# Adjust mapSurf's Rect object based on the camera offset.

mapSurfRect = mapSurf.get\_rect()

mapSurfRect.center = (HALF\_WINWIDTH + cameraOffsetX, HALF\_WINHEIGHT + cameraOffsetY)

# Draw mapSurf to the DISPLAYSURF Surface object.

DISPLAYSURF.blit(mapSurf, mapSurfRect)

DISPLAYSURF.blit(levelSurf, levelRect)

stepSurf = BASICFONT.render('Steps: %s' % (gameStateObj['stepCounter']), 1, TEXTCOLOR)

stepRect = stepSurf.get\_rect()

stepRect.bottomleft = (20, WINHEIGHT - 10)

DISPLAYSURF.blit(stepSurf, stepRect)

if levelIsComplete:

# is solved, show the "Solved!" image until the player

# has pressed a key.

solvedRect = IMAGESDICT['solved'].get\_rect()

solvedRect.center = (HALF\_WINWIDTH, HALF\_WINHEIGHT)

DISPLAYSURF.blit(IMAGESDICT['solved'], solvedRect)

if keyPressed:

return 'solved'

pygame.display.update() # draw DISPLAYSURF to the screen.

FPSCLOCK.tick()

def isWall(mapObj, x, y):

"""Returns True if the (x, y) position on

the map is a wall, otherwise return False."""

if x < 0 or x >= len(mapObj) or y < 0 or y >= len(mapObj[x]):

return False # x and y aren't actually on the map.

elif mapObj[x][y] in ('#', 'x'):

return True # wall is blocking

return False

def decorateMap(mapObj, startxy):

"""Makes a copy of the given map object and modifies it.

Here is what is done to it:

\* Walls that are corners are turned into corner pieces.

\* The outside/inside floor tile distinction is made.

\* Tree/rock decorations are randomly added to the outside tiles.

Returns the decorated map object."""

startx, starty = startxy # Syntactic sugar

# Copy the map object so we don't modify the original passed

mapObjCopy = copy.deepcopy(mapObj)

# Remove the non-wall characters from the map data

for x in range(len(mapObjCopy)):

for y in range(len(mapObjCopy[0])):

if mapObjCopy[x][y] in ('$', '.', '@', '+', '\*'):

mapObjCopy[x][y] = ' '

# Flood fill to determine inside/outside floor tiles.

floodFill(mapObjCopy, startx, starty, ' ', 'o')

# Convert the adjoined walls into corner tiles.

for x in range(len(mapObjCopy)):

for y in range(len(mapObjCopy[0])):

if mapObjCopy[x][y] == '#':

if (isWall(mapObjCopy, x, y-1) and isWall(mapObjCopy, x+1, y)) or \

(isWall(mapObjCopy, x+1, y) and isWall(mapObjCopy, x, y+1)) or \

(isWall(mapObjCopy, x, y+1) and isWall(mapObjCopy, x-1, y)) or \

(isWall(mapObjCopy, x-1, y) and isWall(mapObjCopy, x, y-1)):

mapObjCopy[x][y] = 'x'

elif mapObjCopy[x][y] == ' ' and random.randint(0, 99) < OUTSIDE\_DECORATION\_PCT:

mapObjCopy[x][y] = random.choice(list(OUTSIDEDECOMAPPING.keys()))

return mapObjCopy

def isBlocked(mapObj, gameStateObj, x, y):

"""Returns True if the (x, y) position on the map is

blocked by a wall or star, otherwise return False."""

if isWall(mapObj, x, y):

return True

elif x < 0 or x >= len(mapObj) or y < 0 or y >= len(mapObj[x]):

return True # x and y aren't actually on the map.

elif (x, y) in gameStateObj['stars']:

return True # a star is blocking

return False

def makeMove(mapObj, gameStateObj, playerMoveTo):

"""Given a map and game state object, see if it is possible for the

player to make the given move. If it is, then change the player's

position (and the position of any pushed star). If not, do nothing.

Returns True if the player moved, otherwise False."""

# Make sure the player can move in the direction they want.

playerx, playery = gameStateObj['player']

# This variable is "syntactic sugar". Typing "stars" is more

# readable than typing "gameStateObj['stars']" in our code.

stars = gameStateObj['stars']

# The code for handling each of the directions is so similar aside

# from adding or subtracting 1 to the x/y coordinates. We can

# simplify it by using the xOffset and yOffset variables.

if playerMoveTo == UP:

xOffset = 0

yOffset = -1

elif playerMoveTo == RIGHT:

xOffset = 1

yOffset = 0

elif playerMoveTo == DOWN:

xOffset = 0

yOffset = 1

elif playerMoveTo == LEFT:

xOffset = -1

yOffset = 0

# See if the player can move in that direction.

if isWall(mapObj, playerx + xOffset, playery + yOffset):

return False

else:

if (playerx + xOffset, playery + yOffset) in stars:

# There is a star in the way, see if the player can push it.

if not isBlocked(mapObj, gameStateObj, playerx + (xOffset\*2), playery + (yOffset\*2)):

# Move the star.

ind = stars.index((playerx + xOffset, playery + yOffset))

stars[ind] = (stars[ind][0] + xOffset, stars[ind][1] + yOffset)

else:

return False

# Move the player upwards.

gameStateObj['player'] = (playerx + xOffset, playery + yOffset)

return True

def startScreen():

"""Display the start screen (which has the title and instructions)

until the player presses a key. Returns None."""

# Position the title image.

titleRect = IMAGESDICT['title'].get\_rect()

topCoord = 50 # topCoord tracks where to position the top of the text

titleRect.top = topCoord

titleRect.centerx = HALF\_WINWIDTH

topCoord += titleRect.height

# Unfortunately, Pygame's font & text system only shows one line at

# a time, so we can't use strings with \n newline characters in them.

# So we will use a list with each line in it.

instructionText = ['Push the stars over the marks.',

'Arrow keys to move, QZSD for camera control, P to change character.',

'Backspace to reset level, Esc to quit.',

'N for next level, B to go back a level.']

# Start with drawing a blank color to the entire window:

DISPLAYSURF.fill(BGCOLOR)

# Draw the title image to the window:

DISPLAYSURF.blit(IMAGESDICT['title'], titleRect)

# Position and draw the text.

for i in range(len(instructionText)):

instSurf = BASICFONT.render(instructionText[i], 1, TEXTCOLOR)

instRect = instSurf.get\_rect()

topCoord += 10 # 10 pixels will go in between each line of text.

instRect.top = topCoord

instRect.centerx = HALF\_WINWIDTH

topCoord += instRect.height # Adjust for the height of the line.

DISPLAYSURF.blit(instSurf, instRect)

while True: # Main loop for the start screen.

for event in pygame.event.get():

if event.type == QUIT:

terminate()

elif event.type == KEYDOWN:

if event.key == K\_ESCAPE:

terminate()

return # user has pressed a key, so return.

# Display the DISPLAYSURF contents to the actual screen.

pygame.display.update()

FPSCLOCK.tick()

def readLevelsFile(filename):

assert os.path.exists(filename), 'Cannot find the level file: %s' % (filename)

mapFile = open(filename, 'r')

# Each level must end with a blank line

content = mapFile.readlines() + ['\r\n']

mapFile.close()

levels = [] # Will contain a list of level objects.

levelNum = 0

mapTextLines = [] # contains the lines for a single level's map.

mapObj = [] # the map object made from the data in mapTextLines

for lineNum in range(len(content)):

# Process each line that was in the level file.

line = content[lineNum].rstrip('\r\n')

if ';' in line:

# Ignore the ; lines, they're comments in the level file.

line = line[:line.find(';')]

if line != '':

# This line is part of the map.

mapTextLines.append(line)

elif line == '' and len(mapTextLines) > 0:

# A blank line indicates the end of a level's map in the file.

# Convert the text in mapTextLines into a level object.

# Find the longest row in the map.

maxWidth = -1

for i in range(len(mapTextLines)):

if len(mapTextLines[i]) > maxWidth:

maxWidth = len(mapTextLines[i])

# Add spaces to the ends of the shorter rows. This

# ensures the map will be rectangular.

for i in range(len(mapTextLines)):

mapTextLines[i] += ' ' \* (maxWidth - len(mapTextLines[i]))

# Convert mapTextLines to a map object.

for x in range(len(mapTextLines[0])):

mapObj.append([])

for y in range(len(mapTextLines)):

for x in range(maxWidth):

mapObj[x].append(mapTextLines[y][x])

# Loop through the spaces in the map and find the @, ., and $

# characters for the starting game state.

startx = None # The x and y for the player's starting position

starty = None

goals = [] # list of (x, y) tuples for each goal.

stars = [] # list of (x, y) for each star's starting position.

for x in range(maxWidth):

for y in range(len(mapObj[x])):

if mapObj[x][y] in ('@', '+'):

# '@' is player, '+' is player & goal

startx = x

starty = y

if mapObj[x][y] in ('.', '+', '\*'):

# '.' is goal, '\*' is star & goal

goals.append((x, y))

if mapObj[x][y] in ('$', '\*'):

# '$' is star

stars.append((x, y))

# Basic level design sanity checks:

assert startx != None and starty != None, 'Level %s (around line %s) in %s is missing a "@" or "+" to mark the start point.' % (levelNum+1, lineNum, filename)

assert len(goals) > 0, 'Level %s (around line %s) in %s must have at least one goal.' % (levelNum+1, lineNum, filename)

assert len(stars) >= len(goals), 'Level %s (around line %s) in %s is impossible to solve. It has %s goals but only %s stars.' % (levelNum+1, lineNum, filename, len(goals), len(stars))

# Create level object and starting game state object.

gameStateObj = {'player': (startx, starty),

'stepCounter': 0,

'stars': stars}

levelObj = {'width': maxWidth,

'height': len(mapObj),

'mapObj': mapObj,

'goals': goals,

'startState': gameStateObj}

levels.append(levelObj)

# Reset the variables for reading the next map.

mapTextLines = []

mapObj = []

gameStateObj = {}

levelNum += 1

return levels

def floodFill(mapObj, x, y, oldCharacter, newCharacter):

"""Changes any values matching oldCharacter on the map object to

newCharacter at the (x, y) position, and does the same for the

positions to the left, right, down, and up of (x, y), recursively."""

# In this game, the flood fill algorithm creates the inside/outside

# floor distinction. This is a "recursive" function.

# For more info on the Flood Fill algorithm, see:

# http://en.wikipedia.org/wiki/Flood\_fill

if mapObj[x][y] == oldCharacter:

mapObj[x][y] = newCharacter

if x < len(mapObj) - 1 and mapObj[x+1][y] == oldCharacter:

floodFill(mapObj, x+1, y, oldCharacter, newCharacter) # call right

if x > 0 and mapObj[x-1][y] == oldCharacter:

floodFill(mapObj, x-1, y, oldCharacter, newCharacter) # call left

if y < len(mapObj[x]) - 1 and mapObj[x][y+1] == oldCharacter:

floodFill(mapObj, x, y+1, oldCharacter, newCharacter) # call down

if y > 0 and mapObj[x][y-1] == oldCharacter:

floodFill(mapObj, x, y-1, oldCharacter, newCharacter) # call up

def drawMap(mapObj, gameStateObj, goals):

"""Draws the map to a Surface object, including the player and

stars. This function does not call pygame.display.update(), nor

does it draw the "Level" and "Steps" text in the corner."""

# mapSurf will be the single Surface object that the tiles are drawn

# on, so that it is easy to position the entire map on the DISPLAYSURF

# Surface object. First, the width and height must be calculated.

mapSurfWidth = len(mapObj) \* TILEWIDTH

mapSurfHeight = (len(mapObj[0]) - 1) \* TILEFLOORHEIGHT + TILEHEIGHT

mapSurf = pygame.Surface((mapSurfWidth, mapSurfHeight))

mapSurf.fill(BGCOLOR) # start with a blank color on the surface.

# Draw the tile sprites onto this surface.

for x in range(len(mapObj)):

for y in range(len(mapObj[x])):

spaceRect = pygame.Rect((x \* TILEWIDTH, y \* TILEFLOORHEIGHT, TILEWIDTH, TILEHEIGHT))

if mapObj[x][y] in TILEMAPPING:

baseTile = TILEMAPPING[mapObj[x][y]]

elif mapObj[x][y] in OUTSIDEDECOMAPPING:

baseTile = TILEMAPPING[' ']

# First draw the base ground/wall tile.

mapSurf.blit(baseTile, spaceRect)

if mapObj[x][y] in OUTSIDEDECOMAPPING:

# Draw any tree/rock decorations that are on this tile.

mapSurf.blit(OUTSIDEDECOMAPPING[mapObj[x][y]], spaceRect)

elif (x, y) in gameStateObj['stars']:

if (x, y) in goals:

# A goal AND star are on this space, draw goal first.

mapSurf.blit(IMAGESDICT['covered goal'], spaceRect)

# Then draw the star sprite.

mapSurf.blit(IMAGESDICT['star'], spaceRect)

elif (x, y) in goals:

# Draw a goal without a star on it.

mapSurf.blit(IMAGESDICT['uncovered goal'], spaceRect)

# Last draw the player on the board.

if (x, y) == gameStateObj['player']:

# Note: The value "currentImage" refers

# to a key in "PLAYERIMAGES" which has the

# specific player image we want to show.

mapSurf.blit(PLAYERIMAGES[currentImage], spaceRect)

return mapSurf

def isLevelFinished(levelObj, gameStateObj):

"""Returns True if all the goals have stars in them."""

for goal in levelObj['goals']:

if goal not in gameStateObj['stars']:

# Found a space with a goal but no star on it.

return False

return True

def terminate():

pygame.quit()

sys.exit()

if \_\_name\_\_ == '\_\_main\_\_':

main()

; Star Pusher (Sokoban clone)

; http://inventwithpython.com/blog

; By Al Sweigart al@inventwithpython.com

;

; Everything after the ; is a comment and will be ignored by the game that reads in this file.

;

; The format is described at: ; http://sokobano.de/wiki/index.php?title=Level\_format

; @ - The starting position of the player.

; $ - The starting position for a pushable star.

; . - A goal where a star needs to be pushed.

; + - Player & goal

; \* - Star & goal

; (space) - an empty open space.

; # - A wall.

;

; Level maps are separated by a blank line (I like to use a ; at the start of the line since it is more visible.)

; I tried to use the same format as other people use for their Sokoban games, so that loading new levels is easy. Just place the levels

; in a text file and name it "starPusherLevels.txt" (after renaming this file, of course).

; Starting demo level:

########

## #

# . #

# $ #

# .$@$. #

####$ #

#. #

# ##

#####

; Sasquatch Set I

; 1

###

## # ####

## ### #

## $ #

# @$ # #

### $### #

# #.. #

## ##.# ##

# ##

# ##

#######

; 2

## #####

## ## . #

# ## $. #

## $ #

## $@ ###

# $ ##

#.. ## ##

# # ##

##### #