# KH: All code in this file is original.

# KH: Various Tkinter / matplotlib usage was adapted from various sources,

# KH: but no significant portions were used directly.

import sys

import Tkinter as tk

import numpy as np

import Image

import ImageTk

import diffuse

import matplotlib.pyplot as plt

from matplotlib.colors import LogNorm, Colormap

from random import randint

moves = [(-1, -1), (-1, 0), (-1, 1),

(0, -1), (0, 0), (0, 1),

(1, -1), (1, 0), (1, 1)]

emptyNeighborhood = np.empty((3,3)) # used to avoid array instantiation in getMove()

noneNeighborhood = np.empty((3,3),dtype=object) # used to see where entities exist in neighborhood

gameData = None

# Hotkeys for plotting and their corresponding metrics

plotKeys = {'f':'FoodMetric', 'b':'BirdMetric', 'h':'HawkMetric', 'a':'All'};

class Entity():

"""This class represents a game entity which can interact with environment metrics and other entities"""

def \_\_init\_\_(self, size, name):

"""

This constructor builds an entity

@param size: a two tuple of the form (height, width) which specifies the size of this entities icon on the game map

@param name: A string value indicating the type of entitiy. Must be a key in the gameData['Entities'] dictionary

"""

self.alive = True #a boolean to indicate whether this entity is alive

self.name = name #a name for the type of entity, used as a key into the game data structure

self.canvasItemId = -1 #the integer identifier of a tkinter canvas item corresponding to this entity

self.skill = 10 #a general metric for the skill level of an entity

#Load the image file for this entity

im = Image.open(gameData['Entities'][self.name]['Image'])

im = im.resize(size)

self.image = ImageTk.PhotoImage(image=im)

def getMove(self, neighborhood):

if not gameData['Entities'][self.name]['Moves']:

return(0,0)

a = emptyNeighborhood.copy() # copy array of zeros

for k, v in gameData['Entities'][self.name]['Weights'].items():

a += v\*neighborhood[k]

# -infinity value for obstacles and entities

a[np.where(neighborhood['Obstacles'] == 0)] = -sys.maxint

a[np.where(neighborhood['Entities'] != noneNeighborhood)] = -sys.maxint

maxAt = np.argmax(a)

coords = np.nonzero(neighborhood['Entities'])

for row, col in zip(coords[0], coords[1]):

if neighborhood['Entities'][row,col].name in gameData['Entities'][self.name]['Eats']:

self.skill += neighborhood['Entities'][row,col].skill

neighborhood['Entities'][row,col] = None

move = (row-1,col-1)

move = moves[maxAt]

if move[0] == 0 and move[1] == 0:

# never just stay put. choose random direction instead.

move = (randint(-1,1),randint(-1,1))

return move

def getMetrics(self):

return gameData['Entities'][self.name]['Affects']

class Game(tk.Frame):

def \_\_init\_\_(self, master=tk.Tk(), height=700, width=700):

tk.Frame.\_\_init\_\_(self, master)

self.master = master

self.deltaT = 1 #:Time delay in ms between frame updates, not guaranteed

self.text = [None, None] # holds Tkinter text items

self.paused = False

plt.ion()

self.showPlot = True

self.plotVal = "FoodMetric"

self.height=height

self.width=width

self.loadMap(mapFile = "map2.map")

self.createWidgets()

self.pack()

self.initEntities()

self.initObstacles()

self.initMetrics()

self.update()

self.draw()

def createWidgets(self):

self.Surface = tk.Canvas(self, width=self.width, height=self.height, bg="#FFFFFF")

self.Surface.bind("<Button-1>", self.leftClick)

self.Surface.bind("<Button-3>", self.rightClick)

self.master.bind("<Key>",self.keyPress)

self.Surface.pack()

def loadMap(self, mapFile='map1.map'):

global gameData

dataFile = 'map.data'#mapFile.replace('.map','.data')

f = open(dataFile)

gameData = eval(f.read())

f.close()

entityIDMapping = {}

for k, v in gameData['Entities'].items():

entityIDMapping[v['MapChar']] = k

gameMap = np.loadtxt(mapFile,dtype='c')

self.shape = gameMap.shape

self.entities = np.empty(gameMap.shape, dtype=object)

self.obstacles = np.ones(gameMap.shape)

self.cellsize = self.cellToPixel(1, 1)

for row in range(self.shape[0]):

for col in range(self.shape[1]):

if gameMap[row,col] == 'O': # 0 == obstacle

self.obstacles[row,col] = 0

elif gameMap[row,col] != '.': # . == normal land

self.entities[row,col] = Entity(self.cellsize, entityIDMapping[gameMap[row,col]])

def cellToPixel(self, row, col):

"""Returns tuple (x,y) representing the upper left corner of the given cell in pixel coordinates"""

return (int(col\*float(self.width)/self.shape[1]), int(row\*float(self.height)/self.shape[0]))

def pixelToCell(self,x,y):

"""returns tuple (row,col) representing the cell which contains the given x,y coordinate"""

return (int(y\*self.shape[0]/float(self.height)),int(x\*self.shape[1]/float(self.width)))

def initEntities(self):

"""Initialize all entities that are encoded in the map file.

NOTE: use loadMap() before calling"""

coords = np.nonzero(self.entities)

for row, col in zip(coords[0], coords[1]):

img = self.Surface.create\_image(self.cellToPixel(row,col), image=self.entities[row,col].image, anchor="nw")

self.entities[row,col].canvasItemId = img

self.Surface.update()

def initObstacles(self):

"""Initialize all obstacles that are encoded in the map file.

NOTE: use loadMap() before calling"""

coords = np.nonzero(np.logical\_not(self.obstacles))

for row, col in zip(coords[0], coords[1]):

self.Surface.create\_rectangle(self.cellToPixel(row,col), self.cellToPixel(row+1,col+1), fill="#000000")

self.Surface.update()

def initMetrics(self):

"""Initialize metric seeds and diffusion arrays to zeros.

NOTE: use loadMap() before calling"""

self.metrics = {}

for k,v in gameData['Metrics'].items():

self.metrics[k] = v

self.metrics[k]['seed'] = np.zeros(self.shape)

self.metrics[k]['diffused'] = np.zeros(self.shape)

def seedMetrics(self):

"""For all entities, set their positions in their seed array to their 'Affects' values (specified in data file)"""

for name, data in self.metrics.items():

self.metrics[name]['seed'].fill(0)

coords = np.nonzero(self.entities)

for row, col in zip(coords[0], coords[1]):

entMetrics = self.entities[row,col].getMetrics()

for k, v in entMetrics.items():

self.metrics[k]['seed'][row,col] = v

def diffuseMetrics(self):

"""Diffuse each metric layer"""

for name, data in self.metrics.items():

# call C diffusion extension

data['diffused'] = diffuse.diffuse(data['iters'], data['rate'], data['seed'], self.obstacles)

def getNeighborhood(self,row,col):

"""Returns a 3 X 3 matrix centered around row, col with all metric diffusion values"""

neighborhood = {}

neighborhood['Entities'] = self.entities[row-1:row+2, col-1:col+2]

neighborhood['Obstacles'] = self.obstacles[row-1:row+2, col-1:col+2]

for layer, data in self.metrics.items():

neighborhood[layer] = data['diffused'][row-1:row+2, col-1:col+2]

return neighborhood

def update(self):

"""Main game logic update method. Call once per frame."""

if not self.paused:

self.seedMetrics()

self.diffuseMetrics()

coords = np.nonzero(self.entities)

for row, col in zip(coords[0], coords[1]):

if not self.entities[row,col] or not self.entities[row,col].alive:

continue

move = self.entities[row,col].getMove(self.getNeighborhood(row,col))

newPos = ((row+move[0])%self.entities.shape[0], (col+move[1])%self.entities.shape[1])

if self.obstacles[newPos] and not (self.entities[newPos] and self.entities[newPos].alive):

self.entities[newPos] = self.entities[row, col]

self.entities[row,col] = None

self.draw()

def draw(self):

"""Takes care of all visual rendering/updating"""

coords = np.nonzero(self.entities)

for row, col in zip(coords[0], coords[1]):

item = self.entities[row,col].canvasItemId

pos = self.cellToPixel(row,col)

self.Surface.coords(item, pos)

self.drawText()

if self.showPlot: self.plot()

self.Surface.update()

def drawText(self):

"""Draw text indicating the ammount of each Insert Entity left"""

for i in range(len(gameData['InsertEntity'])):

entityInfo = gameData['InsertEntity'][i]

count = entityInfo['count']

label = entityInfo['label']

countString = str(count) if count > 0 else "No"

clickString = "Left" if i == 0 else "Right"

ammoString = clickString + "-Click to add " + label + ". (" + countString + " " + label + " remaining)"

if self.text[i]: # create the item if it doesn't exist, otherwise set its text

self.Surface.itemconfig(self.text[i], text=ammoString)

else:

self.text[i] = self.Surface.create\_text(20, 20 \* (i + 1), anchor=tk.W, fill='blue', text=ammoString)

def plot(self):

"""Use matplotlib to draw pretty graphs of user-specified metric layers"""

plt.figure(0)

plt.clf()

plt.suptitle(self.plotVal)

if self.plotVal == "All":

m = np.zeros(self.shape)

for k, v in self.metrics.items():

m += v['diffused']

else:

m = self.metrics[self.plotVal]['diffused']

if np.max(m) != 0:

ln = LogNorm()

plt.imshow(m, norm=ln)

plt.contour(m, norm=ln, colors='black', linewidth=.5)

else:

plt.imshow(m)

plt.show()

def click(self, event, insertId):

"""This function is called from click event handlers, with insertId based on left/right click.

Inserts the appropriate entity, if valid."""

row, col = self.pixelToCell(event.x, event.y)

entityInfo = gameData['InsertEntity'][insertId]

# if the position we clicked on is not an obstacle or another entity,

# and we have 'ammo' for that entity remaining, add the entity and decrease the ammo

if self.obstacles[row, col] and self.entities[row, col] == None and entityInfo['count'] > 0:

self.entities[row,col] = Entity(self.cellsize, entityInfo['entity'])

img = self.Surface.create\_image(self.cellToPixel(row,col), image=self.entities[row,col].image, anchor="nw")

self.entities[row,col].canvasItemId = img

entityInfo['count'] -= 1

def keyPress(self, event):

"""Handles keypress events"""

if event.char == "p":

self.paused = not self.paused

elif event.char == "t":

self.showPlot = not self.showPlot

elif event.char == "s":

self.paused = True

self.update()

elif event.char in plotKeys:

self.showPlot = True

self.plotVal = plotKeys[event.char]

self.plot()

def leftClick(self, event):

"""Handles left click events"""

self.click(event, 0)

def rightClick(self, event):

"""Handles right click events"""

self.click(event, 1)

g = Game()

while True:

g.update()