import pygame

import math

from queue import PriorityQueue

import random

WIDTH = 800

WIN = pygame.display.set\_mode((WIDTH, WIDTH))

pygame.display.set\_caption("A\* Path Finding Algorithm")

RED = (255, 0, 0) #(state visited)

GREEN = (0, 255, 0)

BLUE = (0, 255, 0)

YELLOW = (255, 255, 0)

WHITE = (255, 255, 255) #(state not yet visited)

BLACK = (0, 0, 0) #(barrier that have to avoid)

PURPLE = (128, 0, 128) #(path)

ORANGE = (255, 165 ,0) #(start node)

GREY = (128, 128, 128)

TURQUOISE = (64, 224, 208)

# to keep track of this nodes and create grid nodes/spots

#methods that tells the state of the spot

class Spot:

    def \_\_init\_\_(self, row, col, width, total\_rows):

        self.row = row

        self.col = col

        self.x = row \* width

        self.y = col \* width

        self.color = BLACK

        self.neighbours=[]

        self.width = width

        self.total\_rows = total\_rows

    #indexig for rows and columns of grid

    def get\_pos(self):

        return self.row, self.col

    def is\_closed(self):

        return self.color == RED

    def is\_open(self):

        return self.color == GREEN

    def is\_barrier(self):

        return self.color == BLACK

    def is\_start(self):

        return self.color == ORANGE

    def is\_end(self):

        return self.color == PURPLE

    def reset(self):

        self.color = WHITE

    def make\_start(self):

        self.color=ORANGE

    def make\_closed(self):

        self.color = RED

    def make\_open(self):

        self.color = GREEN

    def make\_barrier(self):

        self.color = BLACK

    def make\_end(self):

        self.color = TURQUOISE

    def make\_path(self):

        self.color = PURPLE

    def draw(self, win): #where we want to draw the grid(window=win)

        pygame.draw.rect(win, self.color,(self.x,self.y, self.width, self.width))

    def update\_neighbours(self,grid):

        self.neighbours=[]

        #checks for any barriers

        if self.row<self.total\_rows-1 and not grid[self.row+1][self.col].is\_barrier():#down

            self.neighbours.append(grid[self.row +1][self.col])

        if self.row>0 and not grid[self.row-1][self.col].is\_barrier():#up

            self.neighbours.append(grid[self.row -1][self.col])

        if self.col<self.total\_rows-1 and not grid[self.row][self.col+1].is\_barrier():#left

            self.neighbours.append(grid[self.row ][self.col +1])

        if self.col>0 and not grid[self.row][self.col-1].is\_barrier():#right

            self.neighbours.append(grid[self.row ][self.col-1])

    def \_\_lt\_\_(self,other):

        return False

#define the heuristic function

#finding the distance between point1 and point 2

#using manhattan distance

def h(p1,p2):

    x1,y1=p1

    x2,y2=p2

    return abs(x1-x2)+ abs(y1-y2)

def reconstruct\_path(came\_from,current,draw):

    while current in came\_from:

        current=came\_from[current]

        current.make\_path()

        draw()

def algorithm(draw,grid,start,end):

    count=0#used if we hv 2 vqalues with the same fscore so to consider which we insert in the queue first

    open\_set=PriorityQueue()

    #include the start node with the fscore(f(x)=g(x)+h(x))

    open\_set.put((0,count,start))

    came\_from ={} #keeps track which node came from where

    g\_score = { spot: float("inf") for row in grid for spot in row} #keeps track of from start node to this node

    g\_score[start] = 0 # to estimate how for the end node is from start node

    f\_score = {Spot: float("inf") for row in grid for spot in row}# keeps track of predicted distance from this node to the end node

    f\_score[start] = h(start.get\_pos(),end.get\_pos()) #heuristic distance

    open\_set\_hash = {start}#keep the track of all the items in and out of  the priority\_queue

    while not open\_set.empty():

    #if openset is empty then its considered all possibble path it is going to and if not then path does'nt exist

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                pygame.quit()

        current =open\_set.get()[2] #current node we are looking at

        open\_set\_hash.remove(current)

        if current == end: #where the shortest path is found

            reconstruct\_path(came\_from,end,draw)

            end.make\_end()

            start.make\_start()

            return True

        for neighbor in current.neighbours:

            temp\_g\_score = g\_score[current] + 1 #to know the temporary g score of this neighbor we take current node and add 1

            if temp\_g\_score < g\_score[neighbor]: #update the value say this is the better path

                came\_from[neighbor]= current

                g\_score[neighbor]= temp\_g\_score

                f\_score[neighbor]= temp\_g\_score + h(neighbor.get\_pos(),end.get\_pos())

                if neighbor not in open\_set\_hash:

                    count += 1

                    open\_set.put((f\_score[neighbor],count, neighbor))

                    open\_set\_hash.add(neighbor)

                    neighbor.make\_open()

        draw()

        if current != start:

            current.make\_closed()

    return False #we do not find the path

#make grid

def make\_grid(rows,width):

    grid=[]

    gap=width//rows

    for i in range(rows):

        grid.append([])#prepares a ew row in the grid

        for j in range(rows):

            spot=Spot(i,j,gap,rows)

            grid[i].append(spot)#appends newlycreated spot to the current row

    return grid

def draw\_grid(win,rows,width):

    gap=width//rows

    for i in range (rows):

        pygame.draw.line(win,GREY,(0,i\*gap),(width,i\*gap))

        for j in range (rows):

            pygame.draw.line(win,GREY,(j\*gap,0),(j\*gap,width))

#generating the maze

def generate\_maze(grid, visited, x, y):

    visited.add((x, y))  # Mark the cell as visited

    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

    random.shuffle(directions)

    for dx, dy in directions:

        nx, ny = x + dx \* 2, y + dy \* 2

        if (nx, ny) not in visited:

            if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]):

                visited.add((nx, ny))  # Mark the new cell as visited

                #to make it a passage

                pass1=grid[nx][ny]

                pass1.reset()

                pass2= grid[x + dx][y + dy]

                pass2.reset()

                generate\_maze(grid, visited, nx, ny)  # Recursive call

def create\_maze(rows,grid):

    visited = set()  # Keep track of visited cells

    start\_x, start\_y = random.randint(0, rows - 1), random.randint(0, rows - 1)

    generate\_maze(grid, visited, start\_x, start\_y)  # Generate maze recursively

def draw(win,grid,rows,width):

    #fill entire screen with one color

    #win.fill(WHITE)

    for row in grid:

        for spot in row:

            spot.draw(win)

    #draw\_grid(win,rows,width)

    pygame.display.update()

def get\_clicked\_pos(pos,rows,width): #click

    gap = width // rows

    y,x = pos

    row = y // gap

    col = x // gap

    return row, col

def main(win, width):

    ROWS = 50

    grid = make\_grid(ROWS,width)

    create\_maze(ROWS,grid)

    start = None

    end = None

    run = True

    started = False

    while run:

        draw(win,grid,ROWS,width)

        for event in pygame.event.get():

            if event.type == pygame.QUIT: #if user presses exit

                run = False

            if started: #no interference of user

                continue

            if pygame.mouse.get\_pressed()[0]: #leftmost button

                pos= pygame.mouse.get\_pos()

                row,col=get\_clicked\_pos(pos,ROWS,width)

                spot=grid[row][col]

                if not start and spot!=end and not spot.is\_barrier():

                    start=spot

                    start.make\_start()

                elif not end and spot!=start and not spot.is\_barrier():

                    end=spot

                    end.make\_end()

                # elif spot!=end and spot !=start:

                #     spot.make\_barrier()

            elif pygame.mouse.get\_pressed()[2]: # right button to erase any barriers of if we want to change the start or end positions

                pos= pygame.mouse.get\_pos()

                row,col=get\_clicked\_pos(pos,ROWS,width)

                spot=grid[row][col]

                spot.reset()

                if spot==start:

                    start=None

                elif spot==end:

                    end=None

            if event.type==pygame.KEYDOWN:#checks if key is pressed

                if event.key== pygame.K\_SPACE and not started:#checks if key pressed is space and started is false

                    for row in grid:

                        for spot in row:

                            spot.update\_neighbours(grid)

                    #using anonymous function lambda to call draw as a parameter

                    algorithm(lambda:draw(win,grid,ROWS,width),grid,start,end)

    pygame.quit()

main(WIN,WIDTH)