

Ghost Home Modifications

At this point, Inky and Clyde will bounce around in the ghost home without ever leaving, and if you eat Blinky or Pinky, they'll only circle around the home indefinitely since we restricted their access.

Make sure 'init' in pacman.py has a "self.alive = True"

Run.py

```
import pygame
from pygame.locals import *
from constants import *
from pacman import Pacman
from nodes import NodeGroup
from pellets import PelletGroup
from ghosts import GhostGroup
from fruit import Fruit
from pauser import Pause

class GameController(object):
    def __init__(self):
        pygame.init()
        self.screen = pygame.display.set_mode(SCREENSIZE, 0, 32)
        self.background = None
        self.clock = pygame.time.Clock()
        self.fruit = None
        self.pause = Pause(True)
        self.level = 0
        self.lives = 5

    def setBackground(self):
        self.background = pygame.surface.Surface(SCREENSIZE).convert()
        self.background.fill(BLACK)

    def startGame(self):
        self.setBackground()
        self.nodes = NodeGroup("maze01.txt")
        self.nodes.setPortalPair((0,17), (27,17))
        homekey = self.nodes.createHomeNodes(11.5, 14)
        self.nodes.connectHomeNodes(homekey, (12,14), LEFT)
        self.nodes.connectHomeNodes(homekey, (15,14), RIGHT)
        self.pacman = Pacman(self.nodes.getNodeFromTiles(15, 26))
        self.pellets = PelletGroup("maze01.txt.")
        self.ghosts = GhostGroup(self.nodes.getStartTempNode(), self.pacman)
        self.ghosts.blinky.setStartNode(self.nodes.getNodeFromTiles(2+11.5, 0+14))
        self.ghosts.pinky.setStartNode(self.nodes.getNodeFromTiles(2+11.5, 3+14))
        self.ghosts.inky.setStartNode(self.nodes.getNodeFromTiles(0+11.5, 3+14))
        self.ghosts.clyde.setStartNode(self.nodes.getNodeFromTiles(4+11.5, 3+14))
        self.ghosts.setSpawnNode(self.nodes.getNodeFromTiles(2+11.5, 3+14))
```

```

self.nodes.denyHomeAccess(self.pacman)
self.nodes.denyHomeAccessList(self.ghosts)
self.nodes.denyAccessList(2+11.5, 3+14, LEFT, self.ghosts)
self.nodes.denyAccessList(2+11.5, 3+14, RIGHT, self.ghosts)
self.ghosts.inky.startNode.denyAccess(RIGHT, self.ghosts.inky)
self.ghosts.clyde.startNode.denyAccess(LEFT, self.ghosts.clyde)
self.nodes.denyAccessList(12, 14, UP, self.ghosts)
self.nodes.denyAccessList(15, 14, UP, self.ghosts)
self.nodes.denyAccessList(12, 26, UP, self.ghosts)
self.nodes.denyAccessList(15, 26, UP, self.ghosts)

```

```

def update(self):
    dt = self.clock.tick(30) / 1000.0
    self.pellets.update(dt)
    if not self.pause.paused:
        self.pacman.update(dt)
        self.ghosts.update(dt)
        if self.fruit is not None:
            self.fruit.update(dt)
        self.checkGhostEvents()
        self.checkPelletEvents()
        self.checkFruitEvents
    afterPauseMethod = self.pause.update(dt)
    if afterPauseMethod is not None:
        afterPauseMethod()
    self.checkEvents()
    self.render()

```

```

def checkEvents(self):
    for event in pygame.event.get():
        if event.type == QUIT:
            exit()
        elif event.type == KEYDOWN:
            if event.key == K_SPACE:
                if self.pacman.alive:
                    self.pause.setPause(playerPaused=True)
                    if not self.pause.paused:
                        self.showEntities()
                else:
                    self.hideEntities()

```

```

def checkGhostEvents(self):
    for ghost in self.ghosts:
        if self.pacman.collideGhost(ghost):
            if ghost.mode.current is FREIGHT:
                self.pacman.visible = False
                ghost.visible = False
                self.pause.setPause(pauseTime=1, func=self.showEntities)
                ghost.startSpawn()
            elif ghost.mode.current is not SPAWN:
                if self.pacman.alive:
                    self.lives -= 1

```

```

        self.pacman.die()
        self.ghosts.hide()
        if self.lives <= 0:
            self.pause.setPause(pauseTime=3, func=self.restartGame)
        else:
            self.pause.setPause(pauseTime=3, func=self.resetLevel)

def checkPelletEvents(self):
    pellet = self.pacman.eatPellets(self.pellets.pelletList)
    if pellet:
        self.pellets.numEaten += 1
        self.pellets.pelletList.remove(pellet)
        if pellet.name == POWERPELLET:
            self.ghosts.startFreight()
        if self.pellets.isEmpty():
            self.hideEntities()
            self.pause.setPause(pauseTime=3, func=self.nextLevel)

def checkFruitEvents(self):
    if self.pellets.numEaten == 50 or self.pellets.numEaten == 140:
        if self.fruit is None:
            self.fruit = Fruit(self.nodes.getNodeFromTiles(9, 20))
        if self.fruit is not None:
            if self.pacman.collideCheck(self.fruit):
                self.fruit = None
            elif self.fruit.destroy:
                self.fruit = None

def showEntities(self):
    self.pacman.visible = True
    self.ghosts.show()

def hideEntities(self):
    self.pacman.visible = False
    self.ghosts.hide()

def nextLevel(self):
    self.showEntities()
    self.level += 1
    self.pause.paused = True
    self.startGame()

def restartGame(self):
    self.lives = 5
    self.level = 0
    self.pause.paused = True
    self.fruit = None
    self.startGame()

def resetLevel(self):
    self.pause.paused = True
    self.pacman.reset()
    self.ghosts.reset()

```

```

self.fruit = None

def render(self):
    self.screen.blit(self.background, (0,0))
    self.nodes.render(self.screen)
    self.pellets.render(self.screen)
    if self.fruit is not None:
        self.fruit.render(self.screen)
    self.pacman.render(self.screen)
    self.ghosts.render(self.screen)
    pygame.display.update()

if __name__ == "__main__":
    game = GameController()
    game.startGame()
    while True:
        game.update()

```

Nodes.py

```

import pygame
from vector import Vector2
from constants import *
import numpy as np

class Node(object):
    def __init__(self, x, y):
        self.position = Vector2(x, y)
        self.neighbors = {UP:None, DOWN:None, LEFT:None, RIGHT:None, PORTAL:None}
        self.access = {UP:[PACMAN, BLINKY, PINKY, INKY, CLYDE, FRUIT],
                        DOWN:[PACMAN, BLINKY, PINKY, INKY, CLYDE, FRUIT],
                        LEFT:[PACMAN, BLINKY, PINKY, INKY, CLYDE, FRUIT],
                        RIGHT:[PACMAN, BLINKY, PINKY, INKY, CLYDE, FRUIT]}

    def denyAccess(self, direction, entity):
        if entity.name in self.access[direction]:
            self.access[direction].remove(entity.name)

    def allowAccess(self, direction, entity):
        if entity.name not in self.access[direction]:
            self.access[direction].append(entity.name)

    def render(self, screen):
        for n in self.neighbors.keys():
            if self.neighbors[n] is not None:
                line_start = self.position.asTuple()
                line_end = self.neighbors[n].position.asTuple()
                pygame.draw.line(screen, WHITE, line_start, line_end, 4)
                pygame.draw.circle(screen, RED, self.position.asInt(), 12)

```

```

class NodeGroup(object):
    def __init__(self, level):
        self.level = level
        self.nodesLUT = {}
        self.nodeSymbols = ['+', 'P', 'n']
        self.pathSymbols = ['.', '-', '|', 'p']
        data = self.readMazeFile(level)
        self.createNodeTable(data)
        self.connectHorizontally(data)
        self.connectVertically(data)
        self.homekey = None

    def render(self, screen):
        for node in self.nodesLUT.values():
            node.render(screen)

    def readMazeFile(self, textfile):
        return np.loadtxt(textfile, dtype='<U1')

    def createNodeTable(self, data, xoffset=0, yoffset=0):
        for row in list(range(data.shape[0])):
            for col in list(range(data.shape[1])):
                if data[row][col] in self.nodeSymbols:
                    x, y = self.constructKey(col+xoffset, row+yoffset)
                    self.nodesLUT[(x, y)] = Node(x, y)

    def constructKey(self, x, y):
        return x * TILEWIDTH, y * TILEHEIGHT

    def connectHorizontally(self, data, xoffset=0, yoffset=0):
        for row in list(range(data.shape[0])):
            key = None
            for col in list(range(data.shape[1])):
                if data[row][col] in self.nodeSymbols:
                    if key is None:
                        key = self.constructKey(col+xoffset, row+yoffset)
                    else:
                        otherkey = self.constructKey(col+xoffset, row+yoffset)
                        self.nodesLUT[key].neighbors[RIGHT] = self.nodesLUT[otherkey]
                        self.nodesLUT[otherkey].neighbors[LEFT] = self.nodesLUT[key]
                        key = otherkey
                elif data[row][col] not in self.pathSymbols:
                    key = None

    def connectVertically(self, data, xoffset=0, yoffset=0):
        dataT = data.transpose()
        for col in list(range(dataT.shape[0])):
            key = None
            for row in list(range(dataT.shape[1])):
                if dataT[col][row] in self.nodeSymbols:
                    if key is None:

```

```

        key = self.constructKey(col+xoffset, row+yoffset)
    else:
        otherkey = self.constructKey(col+xoffset, row+yoffset)
        self.nodesLUT[key].neighbors[DOWN] = self.nodesLUT[otherkey]
        self.nodesLUT[otherkey].neighbors[UP] = self.nodesLUT[key]
        key = otherkey
    elif dataT[col][row] not in self.pathSymbols:
        key = None

def getNodeFromPixels(self, xpixel, ypixel):
    if (xpixel, ypixel) in self.nodesLUT.keys():
        return self.nodesLUT[(xpixel, ypixel)]
    return None

def getNodeFromTiles(self, col, row):
    x, y = self.constructKey(col, row)
    if (x, y) in self.nodesLUT.keys():
        return self.nodesLUT[(x, y)]
    return None

def getStartTempNode(self):
    nodes = list(self.nodesLUT.values())
    return nodes[0]

def setPortalPair(self, pair1, pair2):
    key1 = self.constructKey(*pair1)
    key2 = self.constructKey(*pair2)
    if key1 in self.nodesLUT.keys() and key2 in self.nodesLUT.keys():
        self.nodesLUT[key1].neighbors[PORTAL] = self.nodesLUT[key2]
        self.nodesLUT[key2].neighbors[PORTAL] = self.nodesLUT[key1]

def createHomeNodes(self, xoffset, yoffset):
    homedata = np.array([[['X','X','+', 'X','X'],
                           ['X','X','.', 'X','X'],
                           ['+', 'X','.', 'X','+'],
                           ['+', '.', '+', '.', '+'],
                           ['+', 'X','X','X','+']]])

    self.createNodeTable(homedata, xoffset, yoffset)
    self.connectHorizontally(homedata, xoffset, yoffset)
    self.connectVertically(homedata, xoffset, yoffset)
    self.homekey = self.constructKey(xoffset+2, yoffset)
    return self.homekey

def connectHomeNodes(self, homekey, otherkey, direction):
    key = self.constructKey(*otherkey)
    self.nodesLUT[homekey].neighbors[direction] = self.nodesLUT[key]
    self.nodesLUT[key].neighbors[direction*-1] = self.nodesLUT[homekey]

def denyAccess(self, col, row, direction, entity):
    node = self.getNodeFromTiles(col, row)

```

```

    if node is not None:
        node.denyAccess(direction, entity)

def allowAccess(self, col, row, direction, entity):
    node = self.getNodeFromTiles(col, row)
    if node is not None:
        node.allowAccess(direction, entity)

def denyAccessList(self, col, row, direction, entities):
    for entity in entities:
        self.denyAccess(col, row, direction, entity)

def allowAccessList(self, col, row, direction, entities):
    for entity in entities:
        self.allowAccess(col, row, direction, entity)

def denyHomeAccess(self, entity):
    self.nodesLUT[self.homekey].denyAccess(DOWN, entity)

def allowHomeAccess(self, entity):
    self.nodesLUT[self.homekey].allowAccess(DOWN, entity)

def denyHomeAccessList(self, entities):
    for entity in entities:
        self.denyHomeAccess(entity)

def allowHomeAccessList(self, entities):
    for entity in entities:
        self.allowHomeAccess(entity)

```

Entity.py

```

import pygame
from pygame.locals import *
from vector import Vector2
from constants import *
from random import randint

class Entity(object):
    def __init__(self, node):
        self.name = None
        self.directions = {UP:Vector2(0, -1),DOWN:Vector2(0, 1),
                           LEFT:Vector2(-1, 0), RIGHT:Vector2(1, 0), STOP:Vector2()}
        self.direction = STOP
        self.setSpeed(100)
        self.radius = 10
        self.collideRadius = 5
        self.color = WHITE
        self.visible = True
        self.disablePortal = False
        self.goal = None

```

```

self.directionMethod = self.randomDirection
self.setStartNode(node)

def setStartNode(self, node):
    self.node = node
    self.startNode = node
    self.target = node
    self.setPosition()

def setPosition(self):
    self.position = self.node.position.copy()

def validDirection(self, direction):
    if direction is not STOP:
        if self.name in self.node.access[direction]:
            if self.node.neighbors[direction] is not None:
                return True
    return False

def getNewTarget(self, direction):
    if self.validDirection(direction):
        return self.node.neighbors[direction]
    return self.node

def overshotTarget(self):
    if self.target is not None:
        vec1 = self.target.position - self.node.position
        vec2 = self.position - self.node.position
        node2Target = vec1.magnitudeSquared()
        node2Self = vec2.magnitudeSquared()
        return node2Self >= node2Target
    return False

def reverseDirection(self):
    self.direction *= -1
    temp = self.node
    self.node = self.target
    self.target = temp

def oppositeDirection(self, direction):
    if direction is not STOP:
        if direction == self.direction * -1:
            return True
    return False

def setSpeed(self, speed):
    self.speed = speed * TILEWIDTH / 16

def render(self, screen):
    if self.visible:
        p = self.position.asInt()
        pygame.draw.circle(screen, self.color, p, self.radius)

```



```

def update(self, dt):
    self.position += self.directions[self.direction]*self.speed*dt

    if self.overshotTarget():
        self.node = self.target
        directions = self.validDirections()
        direction = self.directionMethod(directions)
        if not self.disablePortal:
            if self.node.neighbors[PORTAL] is not None:
                self.node = self.node.neighbors[PORTAL]
        self.target = self.getNewTarget(direction)
        if self.target is not self.node:
            self.direction = direction
        else:
            self.target = self.getNewTarget(self.direction)

    self.setPosition()

def validDirections(self):
    directions = []
    for key in [UP, DOWN, LEFT, RIGHT]:
        if self.validDirection(key):
            if key != self.direction * -1:
                directions.append(key)
    if len(directions) == 0:
        directions.append(self.direction * -1)
    return directions

def randomDirection(self, directions):
    return directions[randint(0, len(directions)-1)]

def goalDirection(self, directions):
    distances = []
    for direction in directions:
        vec = self.node.position + self.directions[direction]*TILEWIDTH - self.goal
        distances.append(vec.magnitudeSquared())
    index = distances.index(min(distances))
    return directions[index]

def setBetweenNodes(self, direction):
    if self.node.neighbors[direction] is not None:
        self.target = self.node.neighbors[direction]
        self.position = (self.node.position + self.target.position) / 2.0

def reset(self):
    self.setStartNode(self.startNode)
    self.direction = STOP
    self.speed = 100
    self.visible = True

```