**Game File**

**import** pygame  
**import** sys  
**from** pygame.locals **import** \*  
**from** Grid **import** \*  
**from** Player **import** \*  
**from** Enemy **import** \*  
**from** Powerup **import** \*  
**from** Music **import** \*  
**import** time  
**import** random  
**from** random **import** shuffle  
  
**"""Pygame.mixer is used to initialise the audio for my game. 44100 is the frequency, the size is 16, 2 is amount of   
channels (multiple channels allow different sounds to be paused at once, rather than having to pause all at once), and  
4096 is the buffer size"""  
  
"""Pygame.init is used to initialize pygame and the pygame window. The caption is also set here, alongside the  
resolution (1260 x 744). The window is then initialized, which automatically puts the game in Fullscreen. Users will be  
able to exit out of Fullscreen"""**pygame.mixer.pre\_init(44100, 16, 2, 4096)  
pygame.init()  
pygame.display.set\_caption(**"Pacman"**)  
screen\_width = 1260  
screen\_height = 744  
window = pygame.display.set\_mode((screen\_width, screen\_height), FULLSCREEN)  
  
  
*#-----------------------------------------------Menu\_State-----------------------------------------------#***""" This class is the Menu Class. This class is for players to navigate to different parts of the game (classes).  
The Menu Class detects mouse clicking and positioning, then depending on which button is clicked, the game will change  
state and run the game loop for that state."""  
  
"""The attributes for class Menu indicate the game state, window display for the background image, clock tick which  
calculates the frames per second, a self.terminate attribute which is set to false unless the user exits the main menu,  
and a button list which holds the top left corner of each button so pygame can draw the rectangle shape"""  
  
  
class** Menu(object):  
 **def** \_\_init\_\_(self, terminate):  
 self.window = window  
 self.state = **"Menu"** self.terminate = terminate  
 self.clock = pygame.time.Clock()  
 self.button\_list = [(155, 152), (155, 272), (155, 392), (155, 512), (155, 632)]  
  
 **"""The function menu\_event is the first part of the game loop when the game state is in Menu. The function handles  
 all events on the menu"""  
  
 def** menu\_event(self):  
 *"""self.clock.tick represents the frames per second the game is ticking at"""* self.clock.tick(120)  
 **"""This is the event loop"""  
 for** event **in** pygame.event.get():  
 **"""mouse\_pos gets the exact position coordinates of the mouse and represents the coordinates as a tuple."""** mouse\_pos = pygame.mouse.get\_pos()  
 **"""This event type checks for the motion of the mouse and if it is hovering over a button. The first if  
 statement will always be executed as long as the mouse is moving. If the mouse is not within the  
 boundaries of a button, the buttons will just be redrawn with the normal colour. However, if the  
 mouse position is within the boundaries of a button, the buttons will be redrawn but shaded."""  
 if** event.type == pygame.MOUSEMOTION:  
 **if** self.button\_collisions(mouse\_pos):  
 **"""If the mouse is hovering over a button, that button will be shaded."""** self.hover()  
 self.button\_text()  
 pygame.display.update()  
 **else**:  
 self.draw\_buttons()  
  
 **"""Depend on which button the user clicks, it will change the state of the game, and therefore change the  
 game loop. The last if statement in the nested-if statement will terminate the game if the user clicks (with  
 the left mouse button) the quit button."""  
  
 if** event.type == pygame.MOUSEBUTTONDOWN:  
 **if** self.button\_collisions(mouse\_pos) **and** self.button\_count == 0:  
 self.state = **"Play"  
 return** self.state  
 **if** self.button\_collisions(mouse\_pos) **and** self.button\_count == 1:  
 self.state = **"Two\_Play"  
 return** self.state  
 **if** self.button\_collisions(mouse\_pos) **and** self.button\_count == 3:  
 self.state = **"Settings"  
 return** self.state  
 **if** self.button\_collisions(mouse\_pos) **and** self.button\_count == 4:  
 self.terminate = **True  
 break  
  
 """If the user is not in fullscreen and clicks the x button on the window, the code will terminate"""  
  
 if** event.type == pygame.QUIT **or** (self.terminate **is True**):  
 self.terminate = **True  
 break  
  
 """if the user presses the Escape button in the main menu, the game will terminate"""  
  
 if** event.type == pygame.KEYDOWN:  
 **if** event.key == K\_ESCAPE:  
 self.terminate = **True  
  
 """This function is checks if the position of the mouse is within the boundaries of a button. If the mouse is within  
 the boundaries and the user clicks their mouse, it will exit out of the main menu.If so, it will return true to the   
 main event loop where more functions are further called."""  
  
 def** button\_collisions(self, mouse\_pos):  
 self.button\_count = -1  
 **for** pos **in** self.button\_list:  
 self.button\_count += 1  
 **if** (mouse\_pos[0] > pos[0]) **and** (mouse\_pos[0] < (pos[0] + 200)):  
 **if** (mouse\_pos[1] > pos[1]) **and** (mouse\_pos[1] < (pos[1] + 100)):  
 **return True  
  
 """Function is here just in case other objects/things need to be drawn to the menu (for future development)"""  
  
 def** menu\_draw(self):  
 **None  
  
 """menu\_update, updates the screen display for the user (120 frames per second)"""  
  
 def** menu\_update(self):  
 pygame.display.update()  
  
 **"""This function is always called on the menu. It draws the text onto the buttons."""  
  
 def** button\_text(self):  
 display\_list = [**"Single-player"**, **"Local-Multiplayer"**, **"Leaderboard"**, **"Settings"**, **"Quit"**]  
 height = 8  
 **for** x **in** display\_list:  
 button\_font = pygame.font.Font(**None**, 33)  
 button\_surf = button\_font.render(x, 1, (255, 255, 255))  
 button\_pos = [3.7 \* 45, height \* 24]  
 height += 5  
 self.window.blit(button\_surf, button\_pos)  
  
 **"""This function is only called if the mouse position is not on the button. This means the button colour will be  
 its normal colour (gold/yellow) """  
  
 def** draw\_buttons(self):  
 y = 154  
 **for** x **in** range(0, 5):  
 pygame.draw.rect(self.window, (0, 0, 0), (157, y, 204, 104), 0)  
 pygame.draw.rect(self.window, (255, 205, 0), (155, y-2, 200, 100), 0)  
 y += 120  
 self.button\_text()  
  
 **"""This function is only called if the mouse position is on the button. This means the button colour will be  
 turned to a darker gold/yellow. This is to indicate to the user that they are hovering over the button"""  
  
 def** hover(self):  
 **for** index, pos **in** enumerate(self.button\_list):  
 **if** self.button\_count == index:  
 pygame.draw.rect(self.window, (178, 143, 0), (pos[0], pos[1], 200, 100), 0)  
  
 **"""This function is called only once, before the game-loop. The function loads the background, scales it to the  
 users screen and blits it to the screen so it does not change unless the game state changes. The buttons are also  
 drawn and the screen is updated using pygame.display.update()"""  
  
 def** load(self):  
 self.background = pygame.image.load(**"menuscreen.jpg"**)  
 self.background = pygame.transform.smoothscale(self.background, (screen\_width, screen\_height))  
 self.window.blit(self.background, (0, 0))  
 self.draw\_buttons()  
 pygame.display.update()  
  
*#-----------------------------------------------Playing\_State-----------------------------------------------#***"""This is the Board class for single-player. The attributes in this class all contribute to how the board and   
functionality of the game work. This class is where we get and set attributes from other classes. All board events,   
drawings to the screen, and updates occur"""  
  
"""This board consists of some key attributes.   
Self.Maze stores the 'list of lists' which is a 2D matrix of my pacman maze.   
Self.window stores the display criteria.   
Self.terminate is a bool, which when turned to True, exits out of the game.   
The attributes, spawn\_count, power\_count, x\_coord and y\_coord are just used as counters for certain board functions.   
They store int values.  
Self.clock also stores the frames/ticks per second.  
The list walls, stores the coordinates of the walls in terms of pixels, as tuples in a list(x, y). The list   
walls\_pos stores the coordinates of the walls in terms of vectors, as tuples in a list. The list free\_cells and  
free\_pos follow the same pattern as wall lists, but instead store the free cells/spaces rather than the walls.  
The attributes self.cell\_width and self.cell\_height, store the width and height of a cell respectively.  
The attributes self.offset\_width and self.offset\_height, store the width and height to get to the centre of a free cell.  
Attribute self.base\_time stores the spawn timer for the enemies (ghosts) to come out.  
Attribute self.pause stores a bool, which allows the user to pause the single-player game using the key 'p'/'P'.  
Attribute self.intersections, stores all the intersection points on the maze."""  
  
"""We initialized several objects from other classes.   
I initialized the player object which holds the attributes of the user such as score and lives, etc...  
I initialized the music object which stores all the sounds of the game as separate methods.  
I initialized the power object which stores the attributes and methods for all the power-ups in the game.  
I initialized the setting object which stores all the methods and attribute for when the game state is at settings and  
the settings game loop is running.  
I initialized all four ghosts as enemy objects (inky, pinky, blinky, clyde). Besides the board and player object being  
passed as an argument, the colour, initial direction, 'if spawned', and name are all passed as arguments.  
All objects have the board object passed to them to allow them to be manipulate the board from their class.  
The list players and enemy, store all the player and enemy objects in a list respectively."""  
  
  
class** Board(object):  
 **def** \_\_init\_\_(self, terminate):  
 self.Maze = grid  
 self.window = window  
 self.clock = pygame.time.Clock()  
 self.terminate = terminate  
 self.spawn\_count = 0  
 self.power\_count = 0  
 self.x\_coord = 0  
 self.y\_coord = 0  
 self.state = **"Single"** self.walls = []  
 self.walls\_pos = []  
 self.free\_cells = []  
 self.free\_pos = []  
 self.enemy\_spawn = []  
 self.dots = []  
 self.cell\_width = 45  
 self.offset\_width = self.cell\_width // 2  
 self.cell\_height = 24  
 self.offset\_height = self.cell\_height // 2  
 self.base\_time = time.time()  
 self.paused = **False** *########Initialization###########* self.music = Music()  
 self.power = Items(self)  
 self.setting = Settings(terminate, self)  
 self.player = Player(self, **"Player1"**)  
 self.intersections = []  
 self.inky = Enemy(self, self.player, (178, 225, 255), **'L'**, **False**, **"inky"**)  
 self.blinky = Enemy(self, self.player, (178, 225, 120), **'R'**, **False**, **"blinky"**)  
 self.pinky = Enemy(self, self.player, (93, 5, 120), **'L'**, **False**, **"pinky"**)  
 self.clyde = Enemy(self, self.player, (154, 253, 78), **'R'**, **False**, **"clyde"**)  
 self.players = [self.player]  
 self.enemy = [self.inky, self.pinky, self.blinky, self.clyde]  
  
 **"""This function handles all the boards events. Some events are just single-player and some are both single-player  
 and multi-player"""  
  
 def** play\_event(self):  
 *"""self.clock.tick represents the frames per second the game is ticking at"""* self.clock.tick(120)  
 **"""The variable keys is used to store any buttons pressed on the keyboard (pygame.key.get\_pressed is an in-built  
 pygame function"""** keys = pygame.key.get\_pressed()  
 **"""This for loop is used to detect any keyboard events other than movement. This could be quitting the game,  
 using a power up or even pausing the game"""  
 for** event **in** pygame.event.get():  
 **if** event.type == pygame.QUIT **or** (self.terminate **is True**):  
 self.terminate = **True  
 break  
 """pygame.KEYDOWN is an in-built function used to detect key presses."""  
 """If the user presses 'f' or 'F' on their keyboard the power-up of the user will activate, if the user is  
 carrying one. If the user presses 'p' or 'P' on their keyboard, the game will be paused."""  
 if** event.type == pygame.KEYDOWN:  
 **if** event.key == pygame.K\_f:  
 self.power.activate\_power\_up()  
 **if** event.key == pygame.K\_p:  
 self.paused = **not** self.paused  
 **break  
  
 """Depending on the difficulty selection in the settings, the game will change difficulty. The difficulty change  
 changes the logarithmic function I have implemented, which is the cost function."""  
  
 """self.difficulty = 0, is easy. self.difficulty = 1, is medium. self.difficulty = 2, is hard."""  
  
 for** enemy **in** self.enemy:  
 **if** self.difficulty == 0:  
 enemy.move\_difficulty = 5  
 **elif** self.difficulty == 1:  
 enemy.move\_difficulty = 3  
 **elif** self.difficulty == 2:  
 enemy.move\_difficulty = 2  
  
 **"""self.player\_collision() is an event that detects enemy movement and checks if the movement being made will  
 collide with any walls/non-free-cells"""** self.player\_collision()  
  
 **"""This function updates the board every game loop by calling other functions. It updates player and enemy   
 locations. It checks the spawn timer for the enemies and if the player has died, which then makes the player immune   
 for a specific amount of time. In addition, it checks for power-ups and items on the board"""  
  
 def** play\_update(self):  
 self.player.update()  
 self.check\_timer()  
 self.power.check\_items()  
 self.power.check\_power\_count()  
 self.player.immunity()  
 self.inky.update()  
 self.blinky.update()  
 self.pinky.update()  
 self.clyde.update()  
 pygame.display.update()  
  
 **"""This function draws onto the board every game loop by calling other functions. It draws the dots, power\_ups,  
 player(s) amd enemies. In addition it checks for death, which re-draws the user onto the spawn location. There is  
 also a timer implemented for when enemies are in the spawn location."""  
  
 def** play\_draw(self):  
 *"""self.window.fill and self.window.blit, re draws the background to the screen, so it clears all  
 previous drawings"""* self.window.fill((0, 0, 0))  
 self.window.blit(self.background, (0, 0))  
 *#self.draw\_grid()* self.draw\_pops()  
 self.power.draw\_items()  
 self.player.draw()  
 self.player.check\_death()  
 **if** self.check\_timer():  
 **for** enemy **in** self.enemy:  
 **if** enemy.spawned **is False**:  
 enemy.spawned = **True** enemy.y = 276  
 **break  
 """These three functions check for enemy location to make sure they do not overlap. Also they check for enemy  
 collision and then move the enemies bases on certain/specific criteria, depending what enemy is being moved."""** self.check\_enemy\_location()  
 self.enemy\_moves()  
 self.check\_enemy\_location()  
  
 **"""This function is used for when a player loses all their lives, or presses the escape button. They will then  
 return to the game menu. This is why the state changes back to menu. The function self.check\_score() is also called.  
 This function checks if the current score is greater than the current high score, if so, the current score becomes  
 the new high score."""  
  
 """self.game\_over is called when a player loses all three lives"""  
  
 def** back\_menu(self):  
 keys = pygame.key.get\_pressed()  
 **if** self.player.player\_lives == 0:  
 self.check\_score()  
 self.game\_over(**"GAME OVER"**)  
 self.state = **"Menu"  
 return** self.state  
 *#self.terminate = True* **if** keys[K\_ESCAPE]:  
 self.check\_score()  
 self.state = **"Menu"  
 return** self.state  
 *#self.terminate = True* **"""When the player chooses to start a new game, all board attributes will be reset to their initial values"""  
  
 def** game\_reset(self):  
 self.spawn\_count = 0  
 self.power\_count = 0  
 self.x\_coord = 0  
 self.y\_coord = 0  
 self.player.direction = **" "** self.state = **"Single"** self.walls = []  
 self.walls\_pos = []  
 self.free\_cells = []  
 self.free\_pos = []  
 self.enemy\_spawn = []  
 self.dots = []  
  
 **"""In this function i demonstrate opening, reading and overwriting/writing a file. The file 'highscore.txt' stores  
 a numerical value as a string in the file. This function checks if the current in-game score is higher than the  
 score stored in the file. If so, the score will replace the score in the file. This function is only called when the  
 user exits the single-player game to the main menu."""  
  
 def** check\_score(self):  
 *"""current score stores the string value of the current game score, when the game has ended"""* current\_score = str(self.player.score)  
 **"""This opens the file in read only and stores the file as the variable high\_score. text\_high\_score then  
 interprets the first line of the text, which is the numerical string, and stores it."""  
 with** open(**'highscore.txt'**, **'r'**) **as** high\_score:  
 text\_high\_score = high\_score.readline()  
 **"""converts numerical string to an integer and stores it"""** int\_high\_score = int(text\_high\_score)  
 **"""replaces current high score with a new high score, if the players last game exceeded it."""  
 if** self.player.score > int\_high\_score:  
 text\_high\_score = text\_high\_score.replace(text\_high\_score, current\_score)  
 **"""Here i demonstrate closing a file"""** high\_score.close()  
 **"""Writes the new high score in the file, after replacing it"""  
 with** open(**'highscore.txt'**, **'w'**) **as** high\_score:  
 high\_score.write(text\_high\_score)  
 high\_score.close()  
  
 **"""This function checks if there are other enemies in a connected free cell next to the current enemy. If there is  
 an enemy in the adjacent free cell, the current enemy will move in the opposite direction. This prevents enemies  
 overlapping in the same cell."""  
 """This is the enemy detection algorithm."""  
  
 def** check\_enemy\_location(self):  
 *"""nested for loop is used to test every enemy's position against all other enemies position"""* **for** enemy **in** self.enemy:  
 **for** other\_enemy **in** self.enemy:  
 **"""if enemy is to the right, move left"""  
 if** (enemy.pos[0] + 45 == other\_enemy.pos[0]) **and** (enemy.pos[1] == other\_enemy.pos[1]):  
 enemy.direction = **"L"  
 """checks for enemy collision with walls"""** self.enemy\_collision(enemy.direction, enemy)  
 **"""if enemy is to the left, move right"""  
 if** (enemy.pos[0] - 45 == other\_enemy.pos[0]) **and** (enemy.pos[1] == other\_enemy.pos[1]):  
 enemy.direction = **"R"  
 """checks for enemy collision with walls"""** self.enemy\_collision(enemy.direction, enemy)  
 **"""if enemy is below, move up"""  
 if** (enemy.pos[1] + 24 == other\_enemy.pos[1]) **and** (enemy.pos[0] == other\_enemy.pos[0]):  
 enemy.direction = **"U"  
 """checks for enemy collision with walls"""** self.enemy\_collision(enemy.direction, enemy)  
 **"""if enemy is above, move down"""  
 if** (enemy.pos[1] - 24 == other\_enemy.pos[1]) **and** (enemy.pos[0] == other\_enemy.pos[0]):  
 enemy.direction = **"D"  
 """checks for enemy collision with walls"""** self.enemy\_collision(enemy.direction, enemy)  
  
 **"""This function contains all the different types of movements each ghost will make. Inky will use Dijkstra with a,  
 dynamic matrix that changes every game loop to try trap the player. Pinky will use breadth-first search. Blinky  
 will use random movement but Blinky can track Pacman if he sees him in his line of sight, by using the line of   
 sight algorithm. Clyde will just move intersection to intersection, but will not be able to track Pacman."""  
  
 def** enemy\_moves(self):  
 *"""I put a time delay, otherwise the game speed would be too quick, an in face unplayable. The value 120 is the  
 time in milliseconds. So there is a delay of 120 milliseconds per game loop."""* pygame.time.delay(120)  
  
 **for** enemy **in** self.enemy:  
 **"""for each enemy, the function change\_matrix() is called. This function gets the previous intersection of  
 every enemy, and adjusts Inky's adjacency matrix accordingly"""** enemy.change\_matrix()  
 **"""This if statement check if the enemy position has reached an intersection. If so, a tuple is appended to   
 a list. The reason we need the enemy's last intersection, is to see where the path for the searching  
 algorithms will start for Inky and Pinky, and where the next possible intersection can be for Clyde. Also  
 the line of sight algorithm for Blinky, Inky and Pinky. In addition, we need the last intersection of each   
 enemy to be able to change the adjacency matrix for Inky."""  
  
 if** enemy.pos **in** self.intersections:  
 enemy.last\_intersection.append(enemy.pos)  
 **"""When the enemies spawn and the player spawns, they have not yet touched an intersection, so all four   
 algorithm's will not be able to work. Therefore the enemies will move randomly until they   
 touch one and the player touches one."""  
 """If the player uses the invisibility potion, all the ghosts will have to move randomly"""  
  
 if** ((len(enemy.last\_intersection) **and** len(self.player.last\_intersection)) != 0) **and** (self.player.cloak **is False**):  
  
 **"""If the enemy in the for loop is inky, the dijkstra algorithm will take place"""  
 if** enemy.name == **"inky"**:  
 cords, cords\_next = enemy.dijkstra()  
 self.searching\_location(enemy, cords, cords\_next)  
  
 **"""If the enemy in the for loop is blinky, blinky will move randomly but the line of sight algorithm   
 will take place"""  
 if** enemy.name == **"blinky"**:  
 bool\_value = self.in\_line(enemy)  
 **if** bool\_value **is True**:  
 enemy.changeLocation(random.choice([**'L'**, **'U'**, **'D'**, **'R'**]))  
 self.enemy\_collision(enemy.direction, enemy)  
 **else**:  
 **if** enemy.pos[0] > self.player.pos[0] **and** enemy.pos[1] == self.player.pos[1]:  
 enemy.changeLocation(**'L'**)  
 self.enemy\_collision(enemy.direction, enemy)  
 **if** enemy.pos[0] < self.player.pos[0] **and** enemy.pos[1] == self.player.pos[1]:  
 enemy.changeLocation(**'R'**)  
 self.enemy\_collision(enemy.direction, enemy)  
 **if** enemy.pos[1] < self.player.pos[1] **and** enemy.pos[0] == self.player.pos[0]:  
 enemy.changeLocation(**'D'**)  
 self.enemy\_collision(enemy.direction, enemy)  
 **if** enemy.pos[1] > self.player.pos[1] **and** enemy.pos[0] == self.player.pos[0]:  
 enemy.changeLocation(**'U'**)  
 self.enemy\_collision(enemy.direction, enemy)  
  
 **"""If the enemy in the for loop is pinky, the breadth-first search algorithm will take place"""  
 if** enemy.name == **"pinky"**:  
 cords, cords\_next = enemy.breadth\_first()  
 self.searching\_location(enemy, cords, cords\_next)  
  
 **"""If the enemy in the for loop is clyde, the intersection to intersection algorithm will take place."""  
 if** enemy.name == **"clyde"**:  
 **if** enemy.pos **in** self.intersections:  
 enemy.changeLocation(random.choice([**'L'**, **'U'**, **'D'**, **'R'**]))  
 self.enemy\_collision(enemy.direction, enemy)  
 **else**:  
 self.enemy\_collision(enemy.direction, enemy)  
  
 **else**:  
 enemy.changeLocation(random.choice([**'L'**, **'U'**, **'D'**, **'R'**]))  
 self.enemy\_collision(enemy.direction, enemy)  
  
 **"""This try and except is used to detect whether an enemy is within the boundary of the laser, if the user  
 has activated it. The laser has a start position and an end position. If the enemy is between those two  
 coordinate positions or directly on top of the position, the enemy will die"""  
 """We have to put an except for IndexError because if the player has activated the laser and has run into  
 a wall, the starting and ending position will have an empty tuple."""  
  
 """self.player.power stores the users power up. self.player.laser stores a boolean to check if the player  
 has activated the power\_up. enemy.spawned also stores a boolean operation, and is used as condition to  
 prevent a possibility of players killing ghosts before they have even spawned"""  
 try**:  
 **if** (self.player.power == **"laser"**) **and** (self.player.laser **is True**) **and** (enemy.spawned **is True**):  
 **if** (self.player.direction == **"L"**) **or** (self.player.direction == **"R"**):  
 **if** (self.power.start\_position[0] >= enemy.x >= self.power.end\_position[0]) **or** (self.power.start\_position[0] <= enemy.x <= self.power.end\_position[0]):  
 **if** enemy.y == self.player.y:  
 **"""calls the ghost\_death function to reset some of the ghosts attributes"""** self.ghost\_death(enemy)  
 **if** (self.player.direction == **"D"**) **or** (self.player.direction == **"U"**):  
 **if** (self.power.start\_position[1] >= enemy.y >= self.power.end\_position[1]) **or** (self.power.start\_position[1] <= enemy.y <= self.power.end\_position[1]):  
 **if** enemy.x == self.player.x:  
 **"""calls the ghost\_death function to reset some of the ghosts attributes"""** self.ghost\_death(enemy)  
 **except** IndexError:  
 **return None  
  
 """This function is called when a ghost dies by a user. It resets attributes and plays the 'ghost death' music"""  
  
 def** ghost\_death(self, enemy):  
 *"""Calls ghost death music"""* self.music.enemy\_death\_music()  
 **"""Reset ghost spawn point. Spawn point is stored as a tuple which contains x and y integers (x, y)"""** enemy.x = 607  
 enemy.y = 324  
 **"""Sets the spawned attribute to false. This stops players killing enemies before spawning. In addition by  
 setting the boolean value to false, the if condition of the function self.check\_timer() will pass, which starts a  
 timer for the enemy to spawn again. Once the timer is done, enemy.spawned turns to True."""** enemy.spawned = **False** enemy.last\_intersection.clear()  
  
 **"""This function is called when the path from Dijkstra and Breadth-First Search is found, for Inky and Pinky  
 respectively. Once the path is found, the current intersection cords of the ghost and the next intersection cords,  
 alongside what ghost it is, are passed as arguments into the function.."""  
 """This function is how Inky and Blinky move, after using Dijkstra or Breadth-First Search."""  
 def** searching\_location(self, enemy, cords, cords\_next):  
 *"""If the cords or cords\_next are none, the else condition will be executed. The reason these may be none, is  
 because the enemy is very close to Pacman meaning that there are no further intersections to go to, which means  
 the line of sight algorithm will be used instead. So in that case the ghost goes straight towards Pacman, as  
 there is no intersection between the enemy and Pacman."""* **if** (cords **or** cords\_next) **is not None**:  
 **if** enemy.pos[0] > cords\_next[0]:  
 enemy.changeLocation(**'L'**)  
 self.enemy\_collision(enemy.direction, enemy)  
 **if** enemy.pos[0] < cords\_next[0]:  
 enemy.changeLocation(**'R'**)  
 self.enemy\_collision(enemy.direction, enemy)  
 **if** enemy.pos[1] < cords\_next[1]:  
 enemy.changeLocation(**'D'**)  
 self.enemy\_collision(enemy.direction, enemy)  
 **if** enemy.pos[1] > cords\_next[1]:  
 enemy.changeLocation(**'U'**)  
 self.enemy\_collision(enemy.direction, enemy)  
 **else**:  
 **if** enemy.pos[0] > self.player.pos[0]:  
 enemy.changeLocation(**'L'**)  
 self.enemy\_collision(enemy.direction, enemy)  
 **if** enemy.pos[0] < self.player.pos[0]:  
 enemy.changeLocation(**'R'**)  
 self.enemy\_collision(enemy.direction, enemy)  
 **if** enemy.pos[1] < self.player.pos[1]:  
 enemy.changeLocation(**'D'**)  
 self.enemy\_collision(enemy.direction, enemy)  
 **if** enemy.pos[1] > self.player.pos[1]:  
 enemy.changeLocation(**'U'**)  
 self.enemy\_collision(enemy.direction, enemy)  
  
 **"""This is the spawn timer for the ghosts. Self.base\_time is the initial time. Spawn time is current time.  
 Final time is the current time subtract the base time. We return a bool. True if final\_time == 0, and false if it  
 does not. If true is returned, the enemy spawns in during the game loop. Self.spawn\_count is used to delay the  
 spawn time (this may not be necessary to have the spawn timer working)."""  
  
 def** check\_timer(self):  
 spawn\_time = time.time()  
 final\_time = int(spawn\_time - self.base\_time) % 20  
 self.spawn\_count += 1  
 **if** self.spawn\_count % 5 == 0:  
 **return** final\_time == 0  
  
 **"""The function draw\_grid was created to help me calculate the exact amount of free cells and specific locations   
 I needed for my game (e.g. enemy spawn location). The function draws lines horizontally and vertically to created  
 boxes around the free cells. This function does not run in the game loop, only when I am changing some functionality  
 to the game using location coordinates."""  
  
 def** draw\_grid(self):  
 *"""Draws horizontal lines"""* **for** line **in** range(screen\_width // 45):  
 pygame.draw.line(self.window, (107, 107, 107), (line \* self.cell\_width, 0),  
 (line \* self.cell\_width, screen\_height))  
 **"""Draws vertical lines"""  
 for** line **in** range(screen\_height // 24):  
 pygame.draw.line(self.window, (107, 107, 107), (0, line \* self.cell\_height),  
 (screen\_width, line \* self.cell\_height))  
  
 **"""This is my line of sight algorithm. If the function returns True, the player is not in the line of sight of the  
 enemy. If the function returns False, the enemy can see the player in their line of sight"""  
  
 def** in\_line(self, enemy):  
 *"""This if statement checks if the enemy is on the same x or y coordinate as the player."""* **if** (enemy.pos[0] == self.player.pos[0]) **or** (enemy.pos[1] == self.player.pos[1]):  
 **"""This for loop checks if there is a wall between the enemy and player because even though they may be on   
 the same x or y coordinate, a wall will stop the enemy from seeing the player"""  
 for** value **in** self.walls:  
 **if** (value[0] == enemy.pos[0]) **or** (value[1] == enemy.pos[1]):  
 **if** (enemy.pos[1] < value[1] < self.player.pos[1]) **or** (enemy.pos[1] > value[1] > self.player.pos[1]):  
 **return True  
 if** (enemy.pos[0] < value[0] < self.player.pos[0]) **or** (enemy.pos[0] > value[0] > self.player.pos[0]):  
 **return True  
 return False  
 return True  
  
 """The cells function calculates the free cells in coordinates and in vectors form and stores them in lists   
 free\_cells and free\_pos respectively. In addition it stores all the free cell positions into a dots list. Dots are  
 what Pacman collects as he moves along (dots accumulate the points). As Pacman collects the dots, they get removed  
 from the dots list. Furthermore, the function calculates the wall cells in coordinates and in vectors form and  
 stores them in lists walls\_pos and walls respectively. The enemy spawn location is also calculated."""  
  
 """The reason I need to calculate the free cells as vectors is for the Dijkstra, Breadth-First Search and Line of  
 Sight algorithm."""  
  
 """The function also calculates each intersection on the maze in coordinate form."""  
  
 def** cells(self):  
 *"""self.Maze stores the grid, which is a list of lists. row takes one of the lists, from the lists of lists.  
 cell takes each element from row."""* **for** y, row **in** enumerate(self.Maze):  
 **for** x, cell **in** enumerate(row):  
 **"""0 on the grid represents a free cell"""  
 if** cell == 0:  
 self.free\_pos.append((x, y))  
 self.free\_cells.append((self.x\_coord + self.offset\_width, self.y\_coord + self.offset\_height))  
 self.dots.append((self.x\_coord + self.offset\_width, self.y\_coord + self.offset\_height))  
 **"""1 on the grid represents a wall"""  
 elif** cell == 1:  
 self.walls\_pos.append((x, y))  
 self.walls.append((self.x\_coord + self.offset\_width, self.y\_coord + self.offset\_height))  
 **"""Any other value on the grid represents the enemy spawn"""  
 else**:  
 self.enemy\_spawn.append((self.x\_coord + self.offset\_width, self.y\_coord + self.offset\_height))  
 **"""As the cell goes through every element per list (row), we must add 45 to the x coordinate because  
 every cell is 45 pixels in width."""** self.x\_coord += 45  
 **"""As the list (row) ends and it goes to the next row, we need to reset the x coordinate"""  
 if** self.x\_coord == 1260:  
 self.x\_coord = 0  
 **"""Every row down, the y coordinate increase by 24 because every cell has a height of 24"""** self.y\_coord += 24  
  
 **"""Here is where we append each intersection on the maze to a list, self.intersections"""  
  
 for** value **in** self.free\_cells:  
 **"""If there is minimum an empty cell horizontally and vertically next to the free cell, it must be an  
 intersection."""  
 """free cell to the right"""** x1 = (value[0] + 45, value[1])  
 **"""free cell to the left"""** x2 = (value[0] - 45, value[1])  
 **"""free cell down"""** y1 = (value[0], value[1] + 24)  
 **"""free cell up"""** y2 = (value[0], value[1] - 24)  
 **if** x1 **in** self.free\_cells **or** x2 **in** self.free\_cells:  
 **if** y1 **in** self.free\_cells **or** y2 **in** self.free\_cells:  
 **if** value **not in** self.intersections:  
 self.intersections.append((value[0], value[1]))  
  
 **"""This function draws the dots to the screen"""  
  
 def** draw\_pops(self):  
 *"""Value takes the coordinates as a tuple and draws the dots onto the screen"""* **for** value **in** self.dots:  
 pygame.draw.circle(self.window, (255, 215, 0), (value[0], value[1]), 5)  
 **"""If all the dots have been collected on the maze, all possible dot positions get appended back to the list and  
 all dots will be redrawn on the board."""  
 if** len(self.dots) == 0:  
 **for** value **in** self.free\_cells:  
 self.dots.append((value[0], value[1]))  
 **"""There are two power ups, and they spawn alongside the dots, in place of two of them."""** self.power.spawn()  
 **"""Value takes the coordinates as a tuple and draws the dots onto the screen"""  
 for** value **in** self.dots:  
 pygame.draw.circle(self.window, (255, 215, 0), (value[0], value[1]), 5)  
  
 **"""This function is how a player moves during the game. This function covers player collision, to make sure  
 that players can only move within free cells"""  
  
 def** player\_collision(self):  
 keys = pygame.key.get\_pressed()  
 **"""tup[0] stores the x coordinate and tup[1] stores the y coordinate"""  
 """self.player.direction stores the direction the player is moving in the player class."""  
 """self.player.movement calls the movement function in the player class which actually changes the coordinates  
 of the player depending on the arguments given."""  
  
 """If the player presses the left arrow key, it will check if there is a free cell to the left of the player"""  
 if** keys[pygame.K\_LEFT]:  
 **for** tup **in** self.free\_cells:  
 **if** (tup[0] == self.player.x - self.cell\_width) **and** (tup[1] == self.player.y):  
 self.player.direction = **"L"** self.player.movement(-self.cell\_width, 0)  
 **return None  
 """If the player presses the right arrow key, it will check if there is a free cell to the right of the   
 player"""  
 if** keys[pygame.K\_RIGHT]:  
 **for** tup **in** self.free\_cells:  
 **if** (tup[0] == self.player.x + self.cell\_width) **and** (tup[1] == self.player.y):  
 self.player.direction = **"R"** self.player.movement(self.cell\_width, 0)  
 **return None  
 """If the player presses the up arrow key, it will check if there is a free cell above the player"""  
 if** keys[pygame.K\_UP]:  
 **for** tup **in** self.free\_cells:  
 **if** (tup[0] == self.player.x) **and** (tup[1] == self.player.y - self.cell\_height):  
 self.player.direction = **"U"** self.player.movement(0, -self.cell\_height)  
 **return None  
 """If the player presses the down arrow key, it will check if there is a free cell below the player"""  
 if** keys[pygame.K\_DOWN]:  
 **for** tup **in** self.free\_cells:  
 **if** (tup[0] == self.player.x) **and** (tup[1] == self.player.y + self.cell\_height):  
 self.player.direction = **"D"** self.player.movement(0, self.cell\_height)  
 **return None  
 """Nothing occurs from this function if the player tries to move into a non free cell"""  
  
 """This is how all enemies move. This function checks for collision. It makes sure when all enemies move in a  
 certain direction that it is a free cell and not any other cells. The enemy will not be able to move if it is not  
 a free cell. This function takes the direction and what enemy is moving as the arguments."""  
  
 def** enemy\_collision(self, direction, enemy):  
 *"""If the direction the enemy wants to move is left, it will check if there is a free cell to the left of the  
 enemy. If so the function enemy.moves() is called, which is a function in the Enemy class."""* **if** direction == **"L"**:  
 **for** tup **in** self.free\_cells:  
 **if** (tup[0] == enemy.x - self.cell\_width) **and** (tup[1] == enemy.y):  
 enemy.moves()  
 **break  
 """If the direction the enemy wants to move is right, it will check if there is a free cell to the right of the  
 enemy. If so the function enemy.moves() is called, which is a function in the Enemy class."""  
 if** direction == **"R"**:  
 **for** tup **in** self.free\_cells:  
 **if** (tup[0] == enemy.x + self.cell\_width) **and** (tup[1] == enemy.y):  
 enemy.moves()  
 **break  
 """If the direction the enemy wants to move is up, it will check if there is a free cell above the  
 enemy. If so the function enemy.moves() is called, which is a function in the Enemy class."""  
 if** direction == **"U"**:  
 **for** tup **in** self.free\_cells:  
 **if** (tup[0] == enemy.x) **and** (tup[1] == enemy.y - self.cell\_height):  
 enemy.moves()  
 **break  
 """If the direction the enemy wants to move is down, it will check if there is a free cell below the  
 enemy. If so the function enemy.moves() is called, which is a function in the Enemy class."""  
 if** direction == **"D"**:  
 **for** tup **in** self.free\_cells:  
 **if** (tup[0] == enemy.x) **and** (tup[1] == enemy.y + self.cell\_height):  
 enemy.moves()  
 **break  
  
 """If the game state stored is single, the game is running the single player mode. If so, when the player loses all  
 three lives, the game will be over and this function will be called."""  
  
 """If the game state stored is multi, the game is running the co-op multiplayer mode. If so, when a player loses all  
 three lives, the other player wins and this function is called, or if a player reaches a score of 500, this function  
 is also called and the player that reached score 500 wins."""  
  
 """This function has a parameter text. Depending on what state the game is in, a different argument for text will be  
 passed."""  
  
 """The function creates an animation for the text"""  
  
 def** game\_over(self, text):  
 *"""expansion stores an integer which is the font size of the text"""* expansion = 30  
 **"""This for loop means that the size of the text increases font size up to an extra 120 (so 150 in total).  
 This creates an animation of the text."""  
 for** value **in** range(0, 120):  
 **"""This creates a black screen before displaying the text."""** self.window.fill((0, 0, 0))  
 **"""button\_font is the font for the text"""** button\_font = pygame.font.Font(**None**, expansion)  
 **"""button\_surf stores the text that will be rendered in and the colour it will be"""** button\_surf = button\_font.render(text, 1, (236, 0, 0))  
 **"""button\_pos stores the position [x, y] of the text"""** button\_pos = [8 \* 45, 12 \* 24]  
 **"""Draws and blits the text to the screen using pixels"""** self.window.blit(button\_surf, button\_pos)  
 **"""Increases the font size by 1 every loop"""** expansion += 1  
 **"""By putting a delay of 5 milliseconds, the expansion of the text doesn't increase as fast. Slows the  
 animation down."""** pygame.time.delay(5)  
 **"""Updates the screen every loop."""** pygame.display.update()  
 **"""Once the animation has finished, the text is displayed for another 840 milliseconds."""** pygame.time.delay(840)  
  
 **"""This function, paused\_function, is only called in the game loop if the game is currently paused. The function  
 gets the event pygame.KEYDOWN, which detects if a key is pressed. I have declared that only if 'p' or 'P' is pressed  
 an event occurs. This event is unpausing the game."""  
  
 def** pause\_function(self):  
 **for** event **in** pygame.event.get():  
 **if** event.type == pygame.KEYDOWN:  
 **if** event.key == pygame.K\_p:  
 self.paused = **not** self.paused  
 **break  
  
 """If the game is paused, the text 'Paused' will be displayed on the screen and be blit onto the screen so it does  
 not flash off it."""  
  
 def** pause\_display(self):  
 pause\_font = pygame.font.Font(**None**, 150)  
 pause\_surf = pause\_font.render(**"Paused"**, 1, (255, 255, 255))  
 pause\_pos = [10 \* 45, 12 \* 24]  
 self.window.blit(pause\_surf, pause\_pos)  
 pygame.display.update()  
  
 **"""The load function is only run once, when the state is firstly changed/initialized. Function always runs before   
 the game loop starts. It takes the argument, 'difficulty' which can range from '0', being easy, '1', being medium,   
 and '2' being Hard."""  
  
 def** load(self, difficulty):  
 *"""self.background loads in the maze image and stores it"""* self.background = pygame.image.load(**"Maze.png"**)  
 **"""This in-built function (pygame.transform.smoothscale), scales and adjusts the the image resolution and size   
 to the screen it will be displayed on."""** self.background = pygame.transform.smoothscale(self.background, (screen\_width, screen\_height))  
 **"""blits the image and renders it permanently onto the screen, until the state changes or the game finishes"""** self.window.blit(self.background, (0, 0))  
 **"""Calls the self.cells() and self.power.spawn() function before the game loop starts"""** self.cells()  
 self.power.spawn()  
 **"""The difficulty is set so the cost function can be adjusted for the enemy movement before the game starts."""** self.difficulty = difficulty  
 **"""Updates the screen display before the game starts, so everything gets rendered and updated onto the screen"""** pygame.display.update()  
  
  
*#-----------------------------------------------Multi\_Player\_State-----------------------------------------------#***"""This class (MultiBoard) inherits all the attributes and methods from the class Board.  
The class MultiBoard then has its own attributes. One of which is a player class object, which takes the parameters  
'self'(class board) and the name of the player 'Player2'."""  
  
"""The attribute self.tile\_list store all the coordinates of the lava tiles as tuples in a list.  
The attribute self.rock\_tile\_list store all the coordinates of the rock tiles as tuples in a list.  
The attribute self.tile\_counter stores an integer and is used as a timer that spawns in the lava onto the maze at a  
specific interval.  
The attribute self.rock\_tile\_counter stores an integer and is used as a timer that converts the lava to rock in the at a  
specific interval.  
The attribute self.lava\_to\_rock\_counter is used to create a specific time interval in which lava turns to rock."""  
  
  
class** MultiBoard(Board):  
 **def** \_\_init\_\_(self, terminate):  
 super().\_\_init\_\_(terminate)  
 self.player\_two = Player(self, **"Player2"**)  
 self.tile\_list = []  
 self.rock\_tile\_list = []  
 self.tile\_counter = 1  
 self.rock\_tile\_counter = 0  
 self.lava\_to\_rock\_counter = 1000  
  
 **"""When the player chooses to start a new game, all board class and multiboard class attributes will be reset to   
 their initial values"""  
  
 def** multi\_reset(self):  
 self.tile\_list = []  
 self.rock\_tile\_list = []  
 self.tile\_counter = 1  
 self.rock\_tile\_counter = 0  
 self.lava\_to\_rock\_counter = 1000  
 self.spawn\_count = 0  
 self.power\_count = 0  
 self.x\_coord = 0  
 self.y\_coord = 0  
 self.state = **"Two\_Play"** self.player.direction = **" "** self.walls = []  
 self.walls\_pos = []  
 self.free\_cells = []  
 self.free\_pos = []  
 self.dots = []  
  
 **"""This is the event function in the game loop when the state stores the string 'Two\_Play'.  
 This event function checks and calls for player one and player two collision detection, as well as if there is   
 a winner of the game. In addition, there is an event loop which detects if the user has terminate the game and   
 quit from it."""  
  
 def** two\_play\_event(self):  
 self.clock.tick(120)  
 pygame.time.delay(120)  
 **for** event **in** pygame.event.get():  
 **if** event.type == pygame.QUIT **or** (self.terminate **is True**):  
 self.terminate = **True  
 break** self.check\_winner()  
 self.player\_collision()  
 self.two\_player\_collision()  
  
 **"""If a player loses all three lives or presses escape, the game state will change to 'Menu' and the function  
 self.game\_over will be called, where the Player that won will be displayed (if a player lost all three lives).   
 This function returns the game state to the main game loop."""  
  
 def** multi\_back\_menu(self):  
 keys = pygame.key.get\_pressed()  
 **if** self.player.player\_lives == 0:  
 self.game\_over(**"Player2 Wins"**)  
 self.state = **"Menu"  
 return** self.state  
 **if** self.player\_two.player\_lives == 0:  
 self.game\_over(**"Player1 Wins"**)  
 self.state = **"Menu"  
 return** self.state  
 **if** keys[K\_ESCAPE]:  
 self.state = **"Menu"  
 return** self.state  
  
 **"""This function checks if a player has reached 500 points. If a player has reached 500 points, the function  
 self.game\_over is called, and the winner is displayed onto the screen, before the game state 'Menu' is returned  
 to the game loop."""  
  
 def** check\_winner(self):  
 **for** player **in** self.players:  
 **if** player.score >= 500:  
 winner = player.name + **" Wins"** self.game\_over(winner)  
 self.state = **"Menu"  
 return** self.state  
  
 **"""This is the update function of the game loop. It calls all other functions that are related to the current game   
 state and checks and updates certain attributes related to the player or any other class in the game."""  
  
 def** multi\_play\_update(self):  
 self.check\_player\_location()  
 self.player.immunity()  
 self.player\_two.immunity()  
 self.player.update()  
 self.player\_two.update()  
 pygame.display.update()  
  
 **"""This is the draw function of the game loop. It draws objects and specific things to the screen."""  
  
 def** multi\_play\_draw(self):  
 *"""self.window.fill and self.window.blit, re draws the background to the screen, so it clears all  
 previous drawings"""* self.window.fill((0, 0, 0))  
 self.window.blit(self.background, (0, 0))  
 *#self.draw\_grid()* **"""Draws the dots, player one and player two to the screen. In addition, it checks if player one or player two  
 has died by lava, so they can re draw the player to a position where there is no lava or rock."""** self.draw\_pops()  
 self.player.draw()  
 self.player\_two.draw()  
 self.player.check\_death()  
 self.player\_two.check\_death()  
 **"""This function is called to draw lava or draw rocks and remove lava from the maze/board."""** self.multi\_player\_map\_spawn()  
  
 **"""The function multi\_player\_spawn is called once from the load function. It changes the game state to 'Two\_Play'.  
 It also appends the second player to the list of players, and then initialize each players spawn using pixel  
 coordinates."""  
  
 def** multi\_player\_spawn(self):  
 self.state = **"Two\_Play"** self.players.append(self.player\_two)  
 **for** player **in** self.players:  
 **if** player == self.player:  
 player.x = 1192  
 player.y = 36  
 **if** player == self.player\_two:  
 player.x = 67  
 player.y = 708  
  
 **"""This function is the exact same as player\_collision() function which I described above in the board class.  
 The only differences, is this function is only used for player two and the other is only used for player one, or  
 single player. In addition, instead of arrow keys, player two uses W A S D. """  
  
 def** two\_player\_collision(self):  
 keys = pygame.key.get\_pressed()  
 **if** keys[K\_a]:  
 **for** tup **in** self.free\_cells:  
 **if** (tup[0] == self.player\_two.x - self.cell\_width) **and** (tup[1] == self.player\_two.y):  
 self.player\_two.direction = **"L"** self.player\_two.movement(-self.cell\_width, 0)  
 **return None  
 if** keys[K\_d]:  
 **for** tup **in** self.free\_cells:  
 **if** (tup[0] == self.player\_two.x + self.cell\_width) **and** (tup[1] == self.player\_two.y):  
 self.player\_two.direction = **"R"** self.player\_two.movement(self.cell\_width, 0)  
 **return None  
 if** keys[K\_w]:  
 **for** tup **in** self.free\_cells:  
 **if** (tup[0] == self.player\_two.x) **and** (tup[1] == self.player\_two.y - self.cell\_height):  
 self.player\_two.direction = **"U"** self.player\_two.movement(0, -self.cell\_height)  
 **return None  
 if** keys[K\_s]:  
 **for** tup **in** self.free\_cells:  
 **if** (tup[0] == self.player\_two.x) **and** (tup[1] == self.player\_two.y + self.cell\_height):  
 self.player\_two.direction = **"D"** self.player\_two.movement(0, self.cell\_height)  
 **return None  
  
 """This function is called to draw lava or replace lava for rocks, on the maze/board."""  
  
 def** multi\_player\_map\_spawn(self):  
 *"""Every time self.tile\_counter reaches a multiple of 10, this if statement will be executed. This gives a  
 slight delay for when a new lava tile/cell will be placed."""* **if** self.tile\_counter % 10 == 0:  
 tile = random.choice(self.free\_cells)  
 **"""If there are 30 lava tiles on the map/board, no more will be placed. Once 30 lava tiles are placed, the  
 self.lava\_to\_rock\_counter will = 200. This means every 200 game loops a lava tile will change to a  
 rock tile, once 30 lava tiles have been placed onto the map."""  
 if** len(self.tile\_list) < 30:  
 self.tile\_list.append(tile)  
 shuffle(self.tile\_list)  
 **else**:  
 self.lava\_to\_rock\_counter = 200  
 **"""When the 'tile counter' is a multiple of the 'lava to rock counter' a lava tile will be changed to a  
 rock tile. It will call the function self.lava\_to\_rock()"""  
 if** self.tile\_counter % self.lava\_to\_rock\_counter == 0:  
 self.lava\_to\_rock()  
 self.tile\_counter += 1  
 **"""This for loop takes the position of every tile and draws it onto maze. The if condition makes sure the tile  
 does not get drawn onto a player, and it is a free cell and not in the list of rock tiles."""  
 for** tile **in** self.tile\_list:  
 **if** (self.player.pos != tile) **and** (tile **in** self.free\_cells) **and** (tile **not in** self.rock\_tile\_list) **and** (self.player\_two.pos != tile):  
 x\_cord\_one = tile[0] - self.offset\_width  
 y\_cord\_one = tile[1] - self.offset\_height  
 **"""Draws a rectangular lava tile"""** pygame.draw.rect(self.window, (255, 128, 0), (x\_cord\_one, y\_cord\_one, self.cell\_width, self.cell\_height), 0)  
 **"""Removes the dot from below the lava, otherwise it would be inaccessible by the players."""  
 if** (len(self.tile\_list) != 0) **and** (tile **in** self.dots):  
 self.dots.remove(tile)  
 **"""If the lava tile has turned into a rock tile. The rock tile will be drawn in place of the lava tile."""  
 for** tile **in** self.rock\_tile\_list:  
 x\_cord\_two = tile[0] - self.offset\_width  
 y\_cord\_two = tile[1] - self.offset\_height  
 pygame.draw.rect(self.window, (128, 128, 128), (x\_cord\_two, y\_cord\_two, self.cell\_width, self.cell\_height), 0)  
 **if** (len(self.rock\_tile\_list) != 0) **and** (tile **in** self.dots):  
 self.dots.remove(tile)  
 pygame.display.update()  
  
 **"""The variable rock\_tile stores the coordinates of a lava. If amount of rock tiles is equal to the amount of  
 lava tiles that there were, this if statement will no longer work.  
 self.rock\_tile\_counter starts at 0 and finishes at 29 (30 tiles). len(self.tile\_list) would be equal 30."""  
  
 def** lava\_to\_rock(self):  
 rock\_tile = self.tile\_list[self.rock\_tile\_counter]  
 **if** self.rock\_tile\_counter < len(self.tile\_list):  
 self.rock\_tile\_list.append(rock\_tile)  
 self.rock\_tile\_counter += 1  
 pygame.display.update()  
  
 **"""This function checks if a player is on a lava tile or rock tile."""  
  
 def** check\_player\_location(self):  
 *"""This while loop finds a new spawn location where there is no lava or rocks"""* spawn\_available = **False  
 while True**:  
 new\_spawn = random.choice(self.free\_cells)  
 **if** (new\_spawn **in** self.tile\_list) **or** (new\_spawn **in** self.rock\_tile\_list):  
 **continue  
 else**:  
 spawn\_available = **True  
 break  
  
 """Checks if a player is on a lava tile or rock tile and they are not immune. If so, they will be spawned in a   
 new location (new\_spawn) where there is no lava or rock tile."""  
 """new\_spawn holds a tuple which contains an x or y coordinate"""  
  
 if** spawn\_available **is True**:  
 **for** player **in** self.players:  
 **if** (player.pos **in** self.tile\_list) **and** (player.immune **is False**) **and** (player.pos **not in** self.rock\_tile\_list):  
 **"""If a player dies, the death\_music() function will be called. This plays Pacman dying music.  
 The player will also lose a life.  
 The player will lose all their points, if they have 100 or less points. Otherwise they will just  
 lose 100 points."""** self.music.death\_music()  
 player.player\_lives -= 1  
 **if** player.score >= 100:  
 player.score -= 100  
 **else**:  
 player.score -= player.score  
 **"""Player is moved to the new location after they die."""** player.x = new\_spawn[0]  
 player.y = new\_spawn[1]  
 player.pos = new\_spawn  
 **"""The player becomes immune from all harm for 60 game loops as soon as they die."""** player.immune = **True  
  
 """This function only gets called once, and before the game loop."""  
  
 def** multi\_load(self):  
 *"""self.background loads in the maze image and stores it"""* self.background = pygame.image.load(**"Maze.png"**)  
 **"""This in-built function (pygame.transform.smoothscale), scales and adjusts the the image resolution and size   
 to the screen it will be displayed on."""** self.background = pygame.transform.smoothscale(self.background, (screen\_width, screen\_height))  
 **"""blits the image and renders it permanently onto the screen, until the state changes or the game finishes"""** self.window.blit(self.background, (0, 0))  
 **"""Calls the self.cells() and self.multi\_player\_spawn() function before the game loop starts"""** self.cells()  
 self.multi\_player\_spawn()  
 **"""Updates the screen display before the game starts, so everything gets rendered and updated onto the screen"""** pygame.display.update()  
  
  
*#-----------------------------------------------Settings\_State-----------------------------------------------#***"""This class is the Settings Class. It is accessed when the user clicks on the Settings button in the menu.  
The attribute self.difficulty\_count stores the difficulty of the game the user has chosen (as an integer). 0 is easy,  
1 is medium and 2 is hard.  
The attribute self.state would store the strings 'Settings', so the settings game loop can run.  
self.terminate stores a boolean, and when True will close the program.  
self.board is an attribute that can be used to access all attributes and methods from the Board class.  
self.window stores the properties of the display screen (e.g. resolution etc..).  
self.button\_list stores a list of tuples, which are the positions of all the buttons which will be displayed onto the  
screen.  
self.clock tracks time and is used to set how many frames per second the program will aim to run at."""  
  
  
class** Settings:  
 **def** \_\_init\_\_(self, terminate, board):  
 self.terminate = terminate  
 self.window = window  
 self.board = board  
 self.state = **"Settings"** self.difficulty\_count = 0  
 self.clock = pygame.time.Clock()  
 self.button\_list = [(155, 152), (155, 272), (155, 392), (155, 512), (155, 632)]  
  
 **"""This is the events function of the settings game loop"""  
  
 def** settings\_events(self):  
 *"""60 FPS"""* self.clock.tick(60)  
 mouse\_pos = pygame.mouse.get\_pos()  
  
 **"""This for loop calls and detects all the events in the game loop."""  
  
 for** event **in** pygame.event.get():  
 **"""If the user closes the window, the game will terminate"""  
 if** event.type == pygame.QUIT **or** (self.terminate **is True**):  
 self.terminate = **True  
 break  
 """If the user holds down the escape key, the setting state changes to a menu state, which is returned."""  
 if** event.type == pygame.KEYDOWN:  
 **if** event.key == K\_ESCAPE:  
 self.state = **"Menu"  
 return** self.state  
 **"""This event type checks for the motion of the mouse and if it is hovering over a button. The first if  
 statement will always be executed as long as the mouse is moving. If the mouse is not within the  
 boundaries of a button, the buttons will just be redrawn with the normal colour. However, if the  
 mouse position is within the boundaries of a button, the buttons will be redrawn but shaded."""  
 if** event.type == pygame.MOUSEMOTION:  
 **"""self.settings\_button\_collision(mouse\_pos) checks if the mouse position is within the boundaries  
 of a button(rectangle)"""  
 if** self.settings\_button\_collision(mouse\_pos):  
 pygame.display.update()  
 **else**:  
 self.settings\_draw()  
 **"""This event type checks if a player has pressed their left or right mouse button."""  
 if** event.type == pygame.MOUSEBUTTONDOWN:  
 **"""If this button is clicked on, the select\_difficulty function is called"""  
 if** self.settings\_button\_collision(mouse\_pos) **and** self.button\_count == 0:  
 self.select\_difficulty()  
 **elif** self.settings\_button\_collision(mouse\_pos) **and** self.button\_count == 1:  
 **"""If this button is clicked on, the show\_highscore function is called"""** self.show\_highscore()  
 **elif** self.settings\_button\_collision(mouse\_pos) **and** self.button\_count == 2:  
 **"""If this button is clicked on, the show\_help function is called"""** self.show\_help()  
 **"""If this button is clicked on, the volume of the music is turned to 0(off) or 0.1 (on). Volume is  
 an attribute of the Music class."""  
 elif** self.settings\_button\_collision(mouse\_pos) **and** self.button\_count == 3:  
 **if** self.board.music.volume != 0:  
 self.board.music.volume = 0  
 self.display\_music()  
 **else**:  
 self.board.music.volume = 0.1  
 self.display\_music()  
 **"""If this button is clicked, the state changes to 'Menu', and the user will be return to the Menu  
 and the Menu game loop will rune."""  
 elif** self.settings\_button\_collision(mouse\_pos) **and** self.button\_count == 4:  
 self.state = **"Menu"  
 return** self.state  
  
 **"""This function is called when the user selects the difficulty button. Depending on the count, a difficulty will be  
 selected. This difficulty will then be passed through from the settings game loop, to the single player game loop"""  
  
 def** select\_difficulty(self):  
 **if** self.difficulty\_count == 0:  
 self.difficulty\_count += 1  
 **"""Display function is called"""** self.display\_difficulty(**"Medium"**)  
 **elif** self.difficulty\_count == 1:  
 self.difficulty\_count += 1  
 **"""Display function is called"""** self.display\_difficulty(**"Hard"**)  
 **elif** self.difficulty\_count == 2:  
 self.difficulty\_count = 0  
 **"""Display function is called"""** self.display\_difficulty(**"Easy"**)  
  
 **"""Alongside actually selecting the difficulty, we must display onto the screen what difficulty the user has  
 selected. It takes one argument, which is the text that will be displayed."""  
  
 def** display\_difficulty(self, text):  
 button\_font = pygame.font.Font(**None**, 33)  
 button\_surf = button\_font.render(text, 1, (255, 0, 0))  
 button\_pos = [13 \* 45, 1 \* 24]  
 self.window.blit(button\_surf, button\_pos)  
 pygame.display.update()  
 pygame.time.delay(210)  
  
 **"""Alongside toggling the music on and off by clicking the music button, we must display we the music onto the  
 screen the current state of the volume."""  
  
 def** display\_music(self):  
 button\_font = pygame.font.Font(**None**, 33)  
 **"""If the volume is not zero, the music is shown to be turned on"""  
 if** self.board.music.volume != 0:  
 button\_surf = button\_font.render(**"Music On"**, 1, (255, 0, 0))  
 **else**:  
 **"""If the volume is zero, the music is shown to be turned off"""** button\_surf = button\_font.render(**"Music Off"**, 1, (255, 0, 0))  
 button\_pos = [13 \* 45, 1 \* 24]  
 self.window.blit(button\_surf, button\_pos)  
 pygame.display.update()  
 pygame.time.delay(210)  
  
 **"""In this function I open and read the 'highscore.txt' file. I store the first line of the file in the variable  
 high\_score as a string. I then assign the font, how it will be rendered and the position of the text, before  
 displaying it onto the screen."""  
  
 def** show\_highscore(self):  
 high\_score\_file = open(**'highscore.txt'**, **'r'**)  
 high\_score = high\_score\_file.readline()  
 high\_score\_font = pygame.font.Font(**None**, 66)  
 high\_score\_surf = high\_score\_font.render(high\_score, 1, (255, 0, 0))  
 high\_score\_pos = [13 \* 45, 1 \* 24]  
 self.window.blit(high\_score\_surf, high\_score\_pos)  
 pygame.display.update()  
 pygame.time.delay(210)  
 **"""Here I close the file once we are done displaying the high score."""** high\_score\_file.close()  
  
 **"""Here I open and read the 'help.txt' file. I then displaying the text line by line. """  
  
 def** show\_help(self):  
 help\_file = open(**'help.txt'**, **'r'**)  
 **"""Here I store each line of text as an element in a list (help\_list)."""** help\_list = help\_file.readlines()  
 **"""First line of text is 3 vectors in height down, from the top of the screen."""** height = 3  
 **"""This for loop takes every element (line of text) from the list and displays it line by line. Text will only  
 disappear when all the text lines are finished displaying, as we blit every line to the screen."""  
 for** line **in** help\_list:  
 help\_font = pygame.font.Font(**None**, 66)  
 help\_surf = help\_font.render(line, 1, (255, 0, 0))  
 help\_pos = [13 \* 45, height \* 24]  
 self.window.blit(help\_surf, help\_pos)  
 **"""I increase the height by 2 every time, so as every line is being displayed onto the screen, it is being  
 displayed to vector heights down from the previous line."""** height += 2  
 pygame.display.update()  
 **"""Each time after I display a line of text, before the function displays the next line, pygame delays the  
 program by 840 milliseconds."""** pygame.time.delay(840)  
 **"""When all the lines have been displayed, the file is then closed."""** help\_file.close()  
  
 **"""This function is checks if the position of the mouse is within the boundaries of a button. If the mouse is within  
 the boundaries and the user clicks their mouse, it will change the state/value of certain setting option   
 (e.g. difficulty etc...).If so, it will return true to the main event loop where more functions are   
 further called."""  
  
 def** settings\_button\_collision(self, mouse\_pos):  
 self.button\_count = -1  
 **for** pos **in** self.button\_list:  
 self.button\_count += 1  
 **if** (mouse\_pos[0] > pos[0]) **and** (mouse\_pos[0] < (pos[0] + 200)):  
 **if** (mouse\_pos[1] > pos[1]) **and** (mouse\_pos[1] < (pos[1] + 100)):  
 **return True  
  
 """This is the draw function of the game loop when the state stores 'settings'."""  
  
 def** settings\_draw(self):  
 self.window.fill((0, 0, 0))  
 self.window.blit(self.background, (0, 0))  
 **"""All rectangular buttons are drawn onto the screen"""  
 for** button **in** self.button\_list:  
 pygame.draw.rect(self.window, (0, 0, 255), (button[0], button[1], 200, 100), 0)  
 display\_list = [**"Difficulty"**, **"High Score"**, **"Help"**, **"Music"**, **"Exit To Menu"**]  
 height = 8  
 **"""All text onto the buttons are drawn"""  
 for** display **in** display\_list:  
 button\_font = pygame.font.Font(**None**, 33)  
 button\_surf = button\_font.render(display, 1, (255, 0, 0))  
 button\_pos = [3.7 \* 45, height \* 24]  
 height += 5  
 self.window.blit(button\_surf, button\_pos)  
  
 **"""This is the update functions of the game loop. It updates all displays on the screen."""  
  
 def** settings\_update(self):  
 pygame.display.update()  
  
 **"""This function only gets called once, and before the game loop."""  
  
 def** load(self):  
 *"""self.background loads in the settings image and stores it"""* self.background = pygame.image.load(**"Settings.jpg"**)  
 **"""This in-built function (pygame.transform.smoothscale), scales and adjusts the the image resolution and size   
 to the screen it will be displayed on."""** self.background = pygame.transform.smoothscale(self.background, (screen\_width, screen\_height))  
 **"""blits the image and renders it permanently onto the screen, until the state changes or the game finishes"""** self.window.blit(self.background, (0, 0))  
 **"""Updates the screen display before the game starts, so everything gets rendered and updated onto the screen"""** pygame.display.update()

**Enemy File**

**import** pygame  
**import** math  
**import** time  
**from** collections **import** defaultdict  
  
**"""This is the Enemy Class. It contains all the methods and attributes related to enemies in my Pacman game.  
self.player is an argument in the constructor of this class. It stores all the attributes and methods of the  
Player Class.  
self.board is an argument in the constructor of this class. It stores all the attributes and methods of the  
Board Class.  
self.colour is an argument in the constructor of this class. It stores the colour of the enemy.  
self.direction is an argument in the constructor of this class. It stores the direction (L, R, U, D) the enemy is   
moving. It is stored as a string.  
self.spawned is an argument in the constructor of this class. It stores a boolean value which determines whether an  
enemy is in the spawning area waiting to be spawned in, or is currently out of spawn chasing Pacman.  
self.name stores the name of the Ghost as a string.  
self.move\_counter is used as a time interval for enemy movement. self.move\_counter is used alongside the cost function   
to create a speed increase during the game that slows down as the game goes on due to the logarithmic function.  
self.move\_difficulty stores an integer value which represents the difficulty of the game. Depending on the difficulty  
a different value is passed through to the logarithmic function. 0 represents easy, 1 represents medium and 2 represents  
hard.  
self.x and self.y stores the x and y coordinates in pixels respectively. self.pos stores self.x and self.y as a tuple.  
self.intersections stores all the intersections on the board as vectors.  
self.last\_intersection stores the last intersection the enemy reached, as a vector.  
self.two\_exits stores the intersections where there are two possible exits/entrances.  
self.three\_y\_exits stores the intersections where there are two possible exits/entrances in the y direction  
(up and down) and one possible exits/entrances in the x direction (left or right).  
self.three\_x\_exits stores the intersections where there are two possible exits/entrances in the x direction  
(left and right) and one possible exits/entrances in the y direction (up or down).  
self.four\_exits stores the intersections where there are four (all directions) possible exits/entrances.  
self.matrix is the adjacency matrix that stores all the vertices between nodes. The nodes being the intersections.  
self.matrix\_copy is used to copy the original matrix every game loop. This is for dynamically changing the adjacency  
matrix to allow the ghost Inky to try trap Pacman.  
self.adjacency\_list stores a dictionary and is used to convert the adjacency matrix to an adjacency list.  
self.matrix\_equivalent is used to create dictionary that stores key-value pairs, where the key is the intersection and  
the value is the number each intersection is assigned.  
self.move\_available is used to store whether an enemy move is available. Can be used with the move counter to allow  
the enemy to move after the time interval, but currently not needed."""  
  
  
class** Enemy:  
 **def** \_\_init\_\_(self, board, player, colour, direction, spawned, name):  
 self.player = player  
 self.board = board  
 self.name = name  
 self.move\_counter = 2  
 self.move\_difficulty = 0  
 self.x = 607  
 self.y = 324  
 self.pos = (self.x, self.y)  
 self.direction = direction  
 self.colour = colour  
 self.spawned = spawned  
 self.intersections = []  
 self.last\_intersection = []  
 self.two\_exits = []  
 self.three\_y\_exits = []  
 self.three\_x\_exits = []  
 self.four\_exits = []  
 self.matrix = []  
 self.matrix\_copy = []  
 self.adjacency\_list = defaultdict(list)  
 self.matrix\_equivalent = {}  
 self.move\_available = **True  
  
 """function changeLocation is called whenever the enemy is moving, so it updates the enemy's direction."""  
  
 def** changeLocation(self, direction):  
 self.direction = direction  
  
 **"""When the player chooses to start a new game, all enemy attributes will be reset to their initial values"""  
  
 def** enemy\_reset(self):  
 self.spawned = **False** self.x = 607  
 self.y = 324  
 self.pos = (self.x, self.y)  
 self.move\_counter = 2  
 self.move\_available = **True  
  
 """This moves function is used to move the enemy into a free cell. This function is called after the enemy collision  
 from the Board Class. This board uses a cost function for moving the enemy"""  
  
 def** moves(self):  
 *"""cost\_function has two input values depending on the the players score and the games difficulty. Depending on  
 the difficulty, the self.move\_difficulty value will be different. The harder the difficulty, the lower the  
 value of self.move\_difficulty. self.player.cost\_speed represents the x value of the logarithmic function. This  
 value increases by one every time the players score gets to a multiple of 100. This function makes sure the game  
 constantly increases in speed but at a lower (increase) rate over time. If the value is a float value it will  
 always round."""* cost\_function = round(self.move\_difficulty/math.log10(self.player.cost\_speed))  
 **"""This makes sure that lowest value the cost function can be is 1"""  
 if** cost\_function < 1:  
 cost\_function = 1  
 **"""The cost\_function is used with the move\_counter to decide how often the enemy moves. For example, if  
 cost\_function equals 5, the enemy will move 5 times slower than the player."""  
 if** self.move\_counter % cost\_function == 0:  
 **"""These if statements are executed only if the enemy is spawned, and is dependant on the direction the  
 enemy is moving."""  
 if** self.direction == **'L' and** self.spawned **is True**:  
 self.x -= self.board.cell\_width  
 pygame.draw.circle(self.board.window, (self.colour), (self.x, self.y), 8)  
 **if** self.direction == **'R' and** self.spawned **is True**:  
 self.x += self.board.cell\_width  
 pygame.draw.circle(self.board.window, (self.colour), (self.x, self.y), 8)  
 **if** self.direction == **'U' and** self.spawned **is True**:  
 self.y -= self.board.cell\_height  
 pygame.draw.circle(self.board.window, (self.colour), (self.x, self.y), 8)  
 **if** self.direction == **'D' and** self.spawned **is True**:  
 self.y += self.board.cell\_height  
 pygame.draw.circle(self.board.window, (self.colour), (self.x, self.y), 8)  
 self.pos = (self.x, self.y)  
 self.move\_counter += 1  
 **else**:  
 self.move\_counter += 1  
  
 **"""This function draws and updates the enemy's position onto the screen when the enemy has moved."""  
  
 def** update(self):  
 pygame.draw.circle(self.board.window, (self.colour), (self.x, self.y), 8)  
 pygame.display.update()  
  
 **"""This function is one of the most complicated functions in my program. This function is used to create a matrix  
 and parse the intersections into the adjacency matrix. By parsing the intersections rather than parsing every single  
 free cell, I am making the game easier for the user. If I did not do this it would be almost impossible for the  
 for the player to get a good score. In addition by parsing the intersections my game is very much more efficient.  
 Parsing the intersections means there are 4096 possible paths. If I just parsed every single free cell, there would  
 be 82944 possible paths which is a lot more paths to be calculated."""  
  
 def** create\_matrix(self):  
 self.matrix = []  
 **"""Here I append all the intersection locations in terms of vectors into a list called self.intersections."""  
 for** value **in** self.board.free\_pos:  
 x1 = (value[0] + 1, value[1])  
 x2 = (value[0] - 1, value[1])  
 y1 = (value[0], value[1] + 1)  
 y2 = (value[0], value[1] - 1)  
 **if** x1 **in** self.board.free\_pos **or** x2 **in** self.board.free\_pos:  
 **if** y1 **in** self.board.free\_pos **or** y2 **in** self.board.free\_pos:  
 **if** value **not in** self.intersections:  
 self.intersections.append((value[0], value[1]))  
  
 **"""The function self.nearest\_neighbour() appends each intersection to either a list, self.two\_exits,  
 self.three\_y\_exits, self.three\_x\_exits or self.four\_exits."""** self.nearest\_neighbour()  
  
 **"""Here I create the initial matrix where all the vertices are given initial values of infinity"""** index = 0  
 **for** value **in** self.intersections:  
 self.matrix\_equivalent[value] = index  
 index += 1  
 **for** x **in** range(0, len(self.intersections)):  
 self.matrix.append([])  
 **for** y **in** range(0, len(self.intersections)):  
 self.matrix[x].append(float(**"inf"**))  
  
 **"""Here I adjust the adjacency matrix to parse the intersections. So add weights for the vertices between the  
 intersections (Which are the nodes)."""  
  
 for** index, value **in** enumerate(self.intersections):  
 x\_count = 0  
 y\_count = 0  
 cords\_y\_list = []  
 cords\_x\_list = []  
 weight\_y\_list = []  
 weight\_x\_list = []  
 **for** index2, value2 **in** enumerate(self.intersections):  
 **if** value != value2:  
 **"""Checks if two intersections (value and value2) have the same x coordinate."""  
 if** value[0] == value2[0]:  
 **"""Checks if there is a wall between those two intersections"""** wall\_count = 0  
 **for** index3, value3 **in** enumerate(self.board.walls\_pos):  
 **if** value3[0] == value[0]:  
 **if** (value2[1] >= value3[1] >= value[1]) **or** (value[1] >= value3[1] >= value2[1]):  
 wall\_count += 1  
 **if** wall\_count == 0:  
 **"""Calculates the weight and appends it to the list weight\_y\_list."""** weight = abs(value2[1] - value[1])  
 weight\_y\_list.append(weight)  
 **"""Appends index of the second intersection to the list cords\_y\_list."""** cords\_y\_list.append(index2)  
 **"""Checks if two intersections (value and value2) have the same y coordinate."""  
 if** value[1] == value2[1]:  
 **"""Checks if there is a wall between those two intersections"""** wall\_count = 0  
 **for** index3, value3 **in** enumerate(self.board.walls\_pos):  
 **if** value3[1] == value[1]:  
 **if** (value2[0] >= value3[0] >= value[0]) **or** (value[0] >= value3[0] >= value2[0]):  
 wall\_count += 1  
 **if** wall\_count == 0:  
 **"""Calculates the weight and appends it to the list weight\_y\_list."""** weight = abs(value2[0] - value[0])  
 weight\_x\_list.append(weight)  
 **"""Checks if two intersections (value and value2) have the same y coordinate."""** cords\_x\_list.append(index2)  
  
 **"""Depending on how many exits the intersection has, is how many weights will be added to the adjacency  
 matrix. In addition depending on if the exits are on the x or y axis, weights will be taken from either the  
 weight\_x\_list list or weight\_y\_list list respectively."""  
 if** value **in** self.two\_exits:  
 **while** x\_count < 1 **and** y\_count < 1:  
 min\_x\_weight = min(weight\_x\_list)  
 min\_x\_index = weight\_x\_list.index(min\_x\_weight)  
 index\_x\_value = cords\_x\_list[min\_x\_index]  
 min\_y\_weight = min(weight\_y\_list)  
 min\_y\_index = weight\_y\_list.index(min\_y\_weight)  
 index\_y\_value = cords\_y\_list[min\_y\_index]  
 self.matrix[index][index\_x\_value] = min\_x\_weight  
 self.matrix[index][index\_y\_value] = min\_y\_weight  
 weight\_x\_list[min\_x\_index] = 100  
 weight\_y\_list[min\_y\_index] = 100  
 x\_count += 1  
 y\_count += 1  
 **elif** value **in** self.four\_exits:  
 **while** x\_count < 2 **and** y\_count < 2:  
 min\_x\_weight = min(weight\_x\_list)  
 min\_x\_index = weight\_x\_list.index(min\_x\_weight)  
 index\_x\_value = cords\_x\_list[min\_x\_index]  
 min\_y\_weight = min(weight\_y\_list)  
 min\_y\_index = weight\_y\_list.index(min\_y\_weight)  
 index\_y\_value = cords\_y\_list[min\_y\_index]  
 self.matrix[index][index\_x\_value] = min\_x\_weight  
 self.matrix[index][index\_y\_value] = min\_y\_weight  
 weight\_x\_list[min\_x\_index] = 100  
 weight\_y\_list[min\_y\_index] = 100  
 x\_count += 1  
 y\_count += 1  
 **elif** value **in** self.three\_x\_exits:  
 **while** x\_count < 2:  
 min\_x\_weight = min(weight\_x\_list)  
 min\_x\_index = weight\_x\_list.index(min\_x\_weight)  
 index\_x\_value = cords\_x\_list[min\_x\_index]  
 **if** y\_count < 1:  
 min\_y\_weight = min(weight\_y\_list)  
 min\_y\_index = weight\_y\_list.index(min\_y\_weight)  
 index\_y\_value = cords\_y\_list[min\_y\_index]  
 self.matrix[index][index\_y\_value] = min\_y\_weight  
 weight\_y\_list[min\_y\_index] = 100  
 self.matrix[index][index\_x\_value] = min\_x\_weight  
 weight\_x\_list[min\_x\_index] = 100  
 x\_count += 1  
 y\_count += 1  
 **elif** value **in** self.three\_y\_exits:  
 **while** y\_count < 2:  
 min\_y\_weight = min(weight\_y\_list)  
 min\_y\_index = weight\_y\_list.index(min\_y\_weight)  
 index\_y\_value = cords\_y\_list[min\_y\_index]  
 **if** x\_count < 1:  
 min\_x\_weight = min(weight\_x\_list)  
 min\_x\_index = weight\_x\_list.index(min\_x\_weight)  
 index\_x\_value = cords\_x\_list[min\_x\_index]  
 self.matrix[index][index\_x\_value] = min\_x\_weight  
 weight\_x\_list[min\_x\_index] = 100  
 self.matrix[index][index\_y\_value] = min\_y\_weight  
 weight\_y\_list[min\_y\_index] = 100  
 x\_count += 1  
 y\_count += 1  
  
 **"""This function makes a copy of the original matrix every time it is called. After it makes a copy, it adjusts the  
 weights of the vertices of the matrix for the ghost/enemy Inky. This is based on the last intersection of the other   
 three ghosts. This makes the adjacency matrix dynamic."""  
  
 def** change\_matrix(self):  
 self.matrix\_copy = self.matrix[:]  
 **for** enemy **in** self.board.enemy:  
 row = len(self.matrix)  
 **if** (enemy.name != **"inky"**) **and** (len(enemy.last\_intersection) != 0):  
 enemy\_intersection = self.board.intersections.index(enemy.last\_intersection[-1])  
 **for** non\_inf\_value **in** range(0, row):  
 matrix\_value = self.board.inky.matrix\_copy[enemy\_intersection][non\_inf\_value]  
 **if** matrix\_value < 1000:  
 self.board.inky.matrix\_copy[enemy\_intersection][non\_inf\_value] = 100  
  
 **"""This function calculates which intersections have two, three or four exits. It uses XOR to calculate this."""  
  
 def** nearest\_neighbour(self):  
 **for** value **in** self.intersections:  
 a = (value[0] + 1, value[1]) *#right* b = (value[0] - 1, value[1]) *#left* c = (value[0], value[1] + 1) *#down* d = (value[0], value[1] - 1) *#up* **"""For example this first if statement states, (a and not b) or (b and not a)"""  
 if** (bool(a **in** self.board.free\_pos)) ^ (bool(b **in** self.board.free\_pos)):  
 **if** (bool(c **in** self.board.free\_pos)) ^ (bool(d **in** self.board.free\_pos)):  
 self.two\_exits.append(value)  
 **if** bool(c **in** self.board.free\_pos) == bool(d **in** self.board.free\_pos):  
 self.three\_y\_exits.append(value)  
 **if** (bool(c **in** self.board.free\_pos)) ^ (bool(d **in** self.board.free\_pos)):  
 **if** bool(a **in** self.board.free\_pos) == bool(b **in** self.board.free\_pos):  
 self.three\_x\_exits.append(value)  
 **if** bool(a **in** self.board.free\_pos) == bool(b **in** self.board.free\_pos) **and** bool(c **in** self.board.free\_pos) == bool(d **in** self.board.free\_pos):  
 self.four\_exits.append(value)  
  
 **"""This calculates the minimum distance from the source node to the available nodes, using Dijkstra Algorithm. This  
 is because Dijkstra is weighted."""  
  
 def** minDistance(self, dist, sptSet):  
 mini = float(**"inf"**)  
 min\_index = -1  
 row = len(self.matrix)  
 **"""In this for loop the minimum distance/vertex index is searched for from the set of vertices that have not yet   
 been visited."""  
 for** v **in** range(row):  
 **if** dist[v] < mini **and** sptSet[v] **is False**:  
 mini = dist[v]  
 min\_index = v  
 **return** min\_index  
  
 **"""This function calculates the shortest path to Pacman using Dijkstra.   
 This is path is created for the ghost Inky."""  
  
 def** dijkstra(self):  
 **if** (len(self.last\_intersection) **and** len(self.player.last\_intersection)) != 0:  
 row = len(self.matrix)  
 col = len(self.matrix[0])  
  
 **"""source\_cords is the ghosts last intersection."""  
 """destination\_cords is the players last intersection."""** source\_cords = self.last\_intersection[-1]  
 destination\_cords = self.player.last\_intersection[-1]  
 **"""Getting the index of both gets them in terms of vectors rather than pixel coordinates."""** source = self.board.intersections.index(source\_cords)  
 destination = self.board.intersections.index(destination\_cords)  
  
 **"""dist stores elements of the weights from the source node to all other nodes"""** dist = [float(**"inf"**)] \* row  
 dist[source] = 0  
 **"""sptSet stores boolean values to check if a node has been visited from the source node"""** sptSet = [**False**] \* row  
  
 dict\_nodes = {}  
  
 **for** count **in** range(row):  
  
 **"""u stores the minimum distance index."""** u = self.minDistance(dist, sptSet)  
 **"""stores that u has been visited by changing the bool value from False to True."""** sptSet[u] = **True** *# Update dist value of the adjacent vertices  
 # of the picked vertex only if the current  
 # distance is greater than new distance and  
 # the vertex in not in the shotest path tree* **"""Updates the dist value of the adjacent vertices from the current vertex, u."""  
 """self.matrix[u][v] makes sure the path being checked is not zero. Then makes sure the next node has   
 not yet been visited and then the condition, dist[v] > (dist[u] + self.matrix[u][v]) checks to see if   
 the current distance is greater than the new distance. This is to find the shortest distance."""  
 for** v **in** range(row):  
 **if** self.matrix[u][v] > 0 **and** (sptSet[v] **is False**) **and** dist[v] > (dist[u] + self.matrix[u][v]):  
 dist[v] = dist[u] + self.matrix[u][v]  
 dict\_nodes[v] = u  
  
 **"""The next part of Dijkstra is where the path is formed"""** n = destination  
 path = [n]  
 **"""The while loop appends the nodes required to get to the destination nodes using the shortest path."""  
 while** n != source:  
 path.append(dict\_nodes[n])  
 n = dict\_nodes[n]  
 path.reverse()  
 **"""returns the current cords of the enemy and the next cords the enemy needs to go to.  
 I use a try and except method because if the enemy is close enough to the player that there are no next  
 cords, an index error will occur so the function will return None, None. In addition this is where 'line of  
 sight' algorithm is used, because if the enemy (Inky) is that close to the player, the player will always be  
 in the sight of the enemy."""  
 try**:  
 cords = self.board.intersections[path[0]]  
 cords\_next = self.board.intersections[path[1]]  
 **return** cords, cords\_next  
 **except** IndexError:  
 **return None**, **None  
  
 """This function converts an adjacency matrix to an adjacency list."""  
  
 def** matrix\_to\_\_list(self):  
 self.adjacency\_list.clear()  
 **for** i, value **in** enumerate(self.matrix, 0):  
 **for** j, value2 **in** enumerate(value, 0):  
 **if** value2 != float(**"inf"**):  
 self.adjacency\_list[i].append(j)  
  
 **"""This function calculates a path to Pacman using Breadth-First Search. This is unweighted so it may not take the  
 shortest path, just the first path found. This is path is created for the ghost Pinky."""  
  
 def** breadth\_first(self):  
 **if** (len(self.last\_intersection) **and** len(self.player.last\_intersection)) != 0:  
  
 **"""source\_cords is the ghosts last intersection."""  
 """destination\_cords is the players last intersection."""** source\_cords = self.last\_intersection[-1]  
 destination\_cords = self.player.last\_intersection[-1]  
 **"""Getting the index of both gets them in terms of vectors rather than pixel coordinates."""** source = self.board.intersections.index(source\_cords)  
 destination = self.board.intersections.index(destination\_cords)  
  
 **"""Creates a queue for Breadth-First Search"""** queue = [[source]]  
 **"""stores all nodes that are visited"""** visited = set()  
  
 **while** queue:  
  
 path = queue.pop(0)  
 vertex = path[-1]  
  
 **if** vertex == destination:  
 **"""returns the current cords of the enemy and the next cords the enemy needs to go to.  
 I use a try and except method because if the enemy is close enough to the player that there are   
 no next cords, an index error will occur so the function will return None, None. In addition this   
 is where 'line of sight' algorithm is used, because if the enemy (Pinky) is that close to the   
 player, the player will always be in the sight of the enemy."""  
 try**:  
 cords = self.board.intersections[path[0]]  
 cords\_next = self.board.intersections[path[1]]  
 **return** cords, cords\_next  
 **except** IndexError:  
 **return None**, **None  
 elif** vertex **not in** visited:  
 **"""Gets all the adjacent nodes of vertex, provided that intersection (vertex) has not been   
 visited."""  
 for** current\_neighbour **in** self.adjacency\_list.get(vertex, []):  
 new\_path = list(path)  
 new\_path.append(current\_neighbour)  
 queue.append(new\_path)  
  
 visited.add(vertex)  
 **"""If queue is empty it will return None, None."""  
 return None**, **None**

**Player File**

**import** pygame  
**from** Powerup **import** \*  
  
**"""This is the Player class. It holds all the attributes and methods that represents a player and is related to   
a player"""  
  
"""self.board stores the board classes attributes and methods. This allows me to call attributes and methods from the  
board class in the player class.  
self.name stores the name of the player as a string. Either 'Player1' or 'Player2'.  
self.direction stores the last movement key the user pressed (L, R, U, D).  
self.power stores the current power up the user has collected as a string.  
self.immune stores a boolean value which checks if the user has recently died, so they can be for a short amount of time  
after death.  
self.immunity\_count stores an integer value that is used to determine the time interval when the player is immune.  
self.x and self.y is the starting position of the player (only for single player).  
self.pos stores self.x and self.y as a tuple.  
self.music\_count determines the time interval of when music is played when the user is eating the dots.  
self.score stores an integer value of the users score.  
self.player\_lives stores the current amount of lives a player has. This value gets deducted by 1 every time they die and  
when it reaches 0, the game ends and the state changes back to menu.  
self.cost\_speed represents the value inputted in the logarithmic function, which is the cost function for the speed of  
the game (as the game goes on it speeds up but at a slower rate every time).  
self.last\_intersection stores a list of the last intersection a player has visited. This is needed for the Dijkstra and  
Breadth-First Search algorithm.  
self.cloak and self.laser both store bool values and are only changed to True when a user has activated that power up,  
using the key 'f' or 'F'."""  
  
  
class** Player(object):  
 **def** \_\_init\_\_(self, board, name):  
 self.board = board  
 self.name = name  
 self.direction = **" "** self.power = **"empty"** self.immune = **False** self.immunity\_count = 1  
 self.x = 607  
 self.y = 420  
 self.pos = [(self.x, self.y)]  
 self.music\_count = 0  
 self.score = 0  
 self.player\_lives = 3  
 self.cost\_speed = 2  
 self.last\_intersection = []  
 self.cloak = **False** self.laser = **False  
  
 """This function checks if the user is in the same position as an enemy. If so, the player will die."""  
  
 def** check\_death(self):  
 *"""If the player is in the same position as an enemy, death music is played, the player loses 1 life and their  
 power up if they have once. In addition they become immune for a short period after respawning at the starting  
 point (607, 420)."""* **"""self.board.enemy stores all four enemies, which are objects, in a list."""  
 for** tup2 **in** self.board.enemy:  
 **if** (tup2.x == self.x) **and** (tup2.y == self.y) **and** (self.immune **is False**):  
 self.board.music.death\_music()  
 self.player\_lives -= 1  
 self.board.player.x = 607  
 self.board.player.y = 420  
 self.power = **"empty"** self.immune = **True  
 """If the user has any power ups activated, they get deactivated here."""** self.cloak = **False** self.laser = **False  
 break  
 """The new amount of lives a player has (which will be 1 less than before) will get displayed."""** self.lives\_system()  
  
 **"""This function updates the player position before redrawing the player to the screen.  
 It also appends the last intersection position the player has visited."""  
  
 def** movement(self, x, y):  
 **if** self.pos **in** self.board.intersections:  
 self.last\_intersection.append(self.pos)  
 self.x += x  
 self.y += y  
 self.pos = (self.x, self.y)  
  
 **"""This is the draw function of the Player Class. It draws, not only the player, but the dots the screen, because  
 every time a player eats a dot the screen will need to be refreshed."""  
  
 def** draw(self):  
 *"""This if statement draws a yellow Pacman, when the game state is in single player and the user has not  
 activated any power ups"""* **if** (self.board.state == **"Single"**) **and** (self.cloak **is False**) **and** (self.laser **is False**):  
 pygame.draw.circle(self.board.window, (255, 255, 0), (self.x, self.y), 8)  
 **"""This elif statement draws a yellow Pacman, when the game state is in single player and the user has not   
 collected a power up, and they are immune to enemy death."""  
 elif** (self.board.state == **"Single"**) **and** (self.immune **is True**) **and** (self.power == **"empty"**):  
 pygame.draw.circle(self.board.window, (255, 255, 0), (self.x, self.y), 8)  
 **"""This elif statement draws a red Pacman, when the game state is in single player and the user has  
 activated any power up"""  
 elif** (self.board.state == **"Single"**) **and** ((self.cloak **is True**) **or** (self.laser **is True**)):  
 pygame.draw.circle(self.board.window, (255, 0, 0), (self.x, self.y), 8)  
  
 **"""This if statement draws red Pacman, when the game state is in multi player and it is Player One"""  
 if** (self.board.state == **"Two\_Play"**) **and** (self.name == **"Player1"**):  
 pygame.draw.circle(self.board.window, (255, 0, 0), (self.x, self.y), 8)  
 **"""This if statement draws blue Pacman, when the game state is in multi player and it is Player Two"""  
 if** (self.board.state == **"Two\_Play"**) **and** (self.name == **"Player2"**):  
 pygame.draw.circle(self.board.window, (0, 0, 255), (self.x, self.y), 8)  
  
 **"""This for loop checks if a user is in the same position as a dot. If so music will be played, the dot will be  
 removed from the screen and the score of the suer will be increased by 1 and displayed onto the screen using  
 the function self.score\_system()."""  
  
 for** tup2 **in** self.board.dots:  
 **if** (tup2[0] == self.x) **and** (tup2[1] == self.y):  
 **"""The music only works every time music\_count is a multiple of 4. The reason for this is because the  
 sound is four seconds long. So if a player was constantly collecting the dots and there was no music  
 count, only the first second of the sound would play every time."""  
 if** self.music\_count % 4 == 0:  
 self.board.music.eating\_music()  
 self.board.dots.remove(tup2)  
 self.score += 1  
 self.score\_system()  
 self.music\_count += 1  
 **break** self.score\_system()  
  
 **"""Every time the user gets a score which is a multiple of 100 (e.g. 100, 200, 300 etc...) the cost function  
 increases. This means we increase the input value by 1 in the logarithmic function.   
 The input value represents the x value of the logarithmic function on a graph."""  
  
 if** self.score % 100 == 0:  
 **if** self.cost\_speed <= 30:  
 self.cost\_speed += 1  
  
 **"""Display the current players score to the screen by calling the score\_display() function"""  
  
 def** score\_system(self):  
 **if** self.name == **"Player1"**:  
 self.score\_display(3)  
 **if** self.name == **"Player2"**:  
 self.score\_display(24)  
  
 **"""score\_display updates and blits to the screen the current score of the player.  
 It takes in the argument, value, which is the x coordinate, in vectors, of where the score will be displayed. This  
 depends on each player."""  
  
 def** score\_display(self, value):  
 score\_font = pygame.font.Font(**None**, 50)  
 score\_surf = score\_font.render(str(self.score), 1, (255, 255, 255))  
 score\_pos = [value \* self.board.cell\_width, 13 \* self.board.cell\_height]  
 self.board.window.blit(score\_surf, score\_pos)  
  
 **"""Display the current players lives to the screen by calling the lives\_display() function"""  
  
 def** lives\_system(self):  
 **if** self.name == **"Player1"**:  
 self.lives\_display(3)  
 **if** self.name == **"Player2"**:  
 self.lives\_display(24)  
  
 **"""lives\_display updates and blits to the screen the current lives of the player.  
 It takes in the argument, value, which is the x coordinate, in vectors, of where the lives will be displayed. This  
 depends on each player."""  
  
 def** lives\_display(self, value):  
 lives\_font = pygame.font.Font(**None**, 50)  
 lives\_surf = lives\_font.render(str(self.player\_lives), 1, (255, 255, 255))  
 lives\_pos = [value \* self.board.cell\_width, 17 \* self.board.cell\_height]  
 self.board.window.blit(lives\_surf, lives\_pos)  
  
 **"""This function updates the text on the screen that indicates what the attributes displayed mean (e.g. Score:)."""  
  
 def** update(self):  
 **if** self.name == **"Player1"**:  
 self.update\_display(2)  
 **if** self.name == **"Player2"**:  
 self.update\_display(24)  
  
 **"""update\_display updates and blits to the screen the text description of the displayed attributes of the player.  
 It takes in the argument, value, which is the x coordinate, in vectors, of where the lives will be displayed. This  
 depends on each player.  
 In addition, the players power up is displayed on the right side of the screen, if the game state is in  
 singe player."""  
  
 def** update\_display(self, value):  
 score\_lives = [**"Score"**, **"Lives"**]  
 height = 11.5  
 **for** text **in** score\_lives:  
 button\_font = pygame.font.Font(**None**, 50)  
 button\_surf = button\_font.render(text, 1, (255, 255, 255))  
 button\_pos = [value \* self.board.cell\_width, height \* self.board.cell\_height]  
 self.board.window.blit(button\_surf, button\_pos)  
 height += 4  
 **if** (self.name == **"Player1"**) **and** (self.board.state == **"Single"**):  
 power\_font = pygame.font.Font(**None**, 50)  
 power\_surf = power\_font.render(self.power, 1, (255, 255, 255))  
 power\_pos = [24 \* self.board.cell\_width, 11.5 \* self.board.cell\_height]  
 self.board.window.blit(power\_surf, power\_pos)  
  
 **"""This function is called whenever a player dies. It gives the player a short interval of being immune from dying.  
 This function starts counting (and is called) as soon as the player dies, and after 50 game loops the immunity  
 runs out."""  
  
 def** immunity(self):  
 **if** self.immunity\_count % 50 == 0:  
 self.immunity\_count = 1  
 self.immune = **False  
 elif** self.immune **is True**:  
 self.immunity\_count += 1  
  
 **"""When the player chooses to start a new single player or co-op game, all player class attributes will be reset to   
 their initial values"""  
  
 def** player\_reset(self):  
 self.power = **"empty"** self.immune = **False** self.immunity\_count = 1  
 self.x = 607  
 self.y = 420  
 self.pos = [(self.x, self.y)]  
 self.music\_count = 0  
 self.score = 0  
 self.player\_lives = 3  
 self.cost\_speed = 2  
 self.last\_intersection = []  
 self.cloak = **False** self.laser = **False**

**Powerup File**

**import** pygame  
**import** random  
  
**"""These are the RGB colour values for the power up when they get spawned onto the map."""**invisibility = (153, 0, 76)  
laser = (0, 208, 145)  
  
**"""This is the Items class. It stores all the attributes and methods related to the power ups in my game.  
self.name\_power\_ups stores the names of each power up that will spawn, as strings.  
self.board stores the attributes and methods of the board class, so they can be accessed and changes from the Items  
class.  
self.carry\_item stores a bool value. If false, the user has not picked up a power up. If true, the user has picked up  
a power up.  
self.power\_ups stores a key, value pair. The key being the name of the power up and the value being the location of the  
power up.  
self.colours stores the colours of the power up in a list.  
self.collect\_power stores the name of the power up that is collected, in a list. The element is a string.  
self.start\_position and self.end\_position stores the coordinates as tuples of the starting and end position of the  
laser beam."""  
  
  
class** Items(object):  
 **def** \_\_init\_\_(self, board):  
 self.name\_power\_ups = [**"invisibility"**, **"laser"**]  
 self.board = board  
 self.carry\_item = **False** self.power\_ups = {}  
 self.colours = [invisibility, laser]  
 self.collect\_power = []  
 self.start\_position = ()  
 self.end\_position = ()  
  
 **"""This function is called once at the start of the game, and every time all the dots are removed from the board  
 which means everything needs to respawn."""  
  
 def** spawn(self):  
 self.colours = [invisibility, laser]  
 **for** name **in** self.name\_power\_ups:  
 location = random.choice(self.board.free\_cells)  
 **"""self.power\_ups[name] adds the location of the power up as a value to the key (name of power up)."""** self.power\_ups[name] = location  
 **"""Removes the dot in that location and replaces it with a power up."""** self.board.dots.remove(location)  
  
 **"""This function draws all the power ups with their designated colour to the screen."""  
  
 def** draw\_items(self):  
 *"""colour index represents the designated colour index in the list self.colours"""* colour\_index = 0  
 **for** index, name **in** enumerate(self.power\_ups):  
 colour = self.colours[colour\_index]  
 game\_location = (self.power\_ups[name])  
 **"""game\_location is a ruple that holds the x and y coordinate of the power up location.  
 self.board.window represents the window we are drawing to.  
 The value 3 represents the radius of the power up."""** pygame.draw.circle(self.board.window, colour, (game\_location[0], game\_location[1]), 3)  
 colour\_index += 1  
  
 **"""This function collects a power up by looping through the self.collect\_power list."""  
  
 def** collect(self):  
 self.collect\_power = []  
 **"""checks the invisibility and laser functions to see if the user has obtained either power up."""** power\_up = self.invisibility\_power()  
 power\_up2 = self.laser\_power()  
 self.collect\_power.append(power\_up)  
 self.collect\_power.append(power\_up2)  
 **for** value **in** self.collect\_power:  
 power\_up\_value = value  
 **"""If the current power up the user has is not empty, the users power up attribute will be updated."""  
 if** value != **"empty"**:  
 self.board.player.power = power\_up\_value  
 **break  
  
 """This function is for removing a power up from the board if a user is in the same location as one"""  
  
 def** check\_items(self):  
 self.collect()  
 self.beam()  
 **for** index, name **in** enumerate(self.power\_ups):  
 game\_location = (self.power\_ups[name])  
 **"""User can only collect a power up if they currently are not carrying one."""  
 if** (self.board.player.pos == game\_location) **and** (self.board.power\_count == 0):  
 **if** name == **"invisibility"**:  
 self.colours.remove(invisibility)  
 **if** name == **"laser"**:  
 self.colours.remove(laser)  
 **"""key value pair of the power up is deleted from the dictionary, which therefore means it gets  
 deleted from the board."""  
 del** self.power\_ups[name]  
 **break  
  
 """In this function I check if the user is able (is eligible) to collect the invisibility power up."""  
  
 def** invisibility\_power(self):  
 *"""I have to do a try and except because if the user has already collected the invisibility power up, it will  
 have been removed from the dictionary, so as the function runs, it will give a Key Error."""* **try**:  
 **"""If the user is in the same location as the power up, is not holding the previous power up and if the user   
 did have a power up, the activation time of it has finished."""  
 if** (self.board.player.pos == self.power\_ups[**"invisibility"**]) **and** (self.board.power\_count == 0) **and** (self.board.player.power == **"empty"**):  
 self.carry\_item = **True** power\_up = **"invisibility"** self.board.music.eating\_powerup\_music()  
 **return** power\_up  
 **elif** self.board.player.power == **"invisibility"**:  
 power\_up = **"invisibility"  
 return** power\_up  
 **else**:  
 power\_up = **"empty"  
 return** power\_up  
 **except** KeyError:  
 **return** self.board.player.power  
  
 **"""In this function I check if the user is able (is eligible) to collect the laser power up."""  
  
 def** laser\_power(self):  
 *"""I have to do a try and except because if the user has already collected the laser power up, it will  
 have been removed from the dictionary, so as the function runs, it will give a Key Error."""* **try**:  
 **"""If the user is in the same location as the power up, is not holding the previous power up and if the user   
 did have a power up, the activation time of it has finished."""  
 if** (self.board.player.pos == self.power\_ups[**"laser"**]) **and** (self.board.power\_count == 0) **and** (self.board.player.power == **"empty"**):  
 self.carry\_item = **True** power\_up = **"laser"** self.board.music.eating\_powerup\_music()  
 **return** power\_up  
 **elif** self.board.player.power == **"laser"**:  
 power\_up = **"laser"  
 return** power\_up  
 **else**:  
 power\_up = **"empty"  
 return** power\_up  
 **except** KeyError:  
 **return** self.board.player.power  
  
 **"""If the user activates the invisibility power up (by pressing 'f' or 'F') the attribute in the Player Class,  
 self.cloak will change to True."""  
  
 def** activate\_invisibility(self):  
 self.board.player.cloak = **True  
  
 """If the user activates the laser power up (by pressing 'f' or 'F') the attribute in the Player Class,  
 self.laser will change to True."""  
  
 def** activate\_laser(self):  
 self.board.player.laser = **True  
  
 """This function checks if the user is currently carrying the laser power up and has activated it. If so,  
 the first if statement will execute, and the beam will be drawn."""  
  
 def** beam(self):  
 **if** (self.board.player.power == **"laser"**) **and** (self.board.player.laser **is True**):  
 **"""direction stores a string of the direction the player is moving/facing"""** direction = self.board.player.direction  
 **"""initializes the current distance of the laser, as an integer"""** current\_distance = 10000  
 **"""initializes the end position of the laser."""** self.end\_position = self.board.player.pos  
 **"""Draws the beam when user is moving left."""  
 if** direction == **"L"**:  
 **"""initializes the start position of the laser."""** self.start\_position = self.board.player.pos  
 **"""value stores each free cell coordinates form the list of free cells."""  
 for** value **in** self.board.free\_cells:  
 **if** (value[1] == self.start\_position[1]) **and** (value[0] <= self.start\_position[0]):  
 **"""checks if the end position of the last will be reach the last point before it hits a wall."""  
 if** (value[0] - 45, value[1]) **in** self.board.walls:  
 distance = self.start\_position[0] - value[0]  
 **"""By getting the smallest distance the laser can reach before hitting a wall makes sure  
 that the laser is not going through walls and hitting walls further away."""  
 if** distance < current\_distance:  
 current\_distance = distance  
 **"""end position of the laser stores a tuple of coordinates"""** self.end\_position = value  
 pygame.draw.line(self.board.window, (255, 0, 0), self.start\_position, self.end\_position, 5)  
  
 **"""The variables described above are repeated for also moving Right, Up and Down, just with different  
 inequality symbols when comparing the value and start position in the for loop. Also when calculating the  
 distance."""  
  
 if** direction == **"R"**:  
 self.start\_position = self.board.player.pos  
 **for** value **in** self.board.free\_cells:  
 **if** (value[1] == self.start\_position[1]) **and** (value[0] >= self.start\_position[0]):  
 **if** (value[0] + 45, value[1]) **in** self.board.walls:  
 distance = value[0] - self.start\_position[0]  
 **if** distance < current\_distance:  
 current\_distance = distance  
 self.end\_position = value  
 pygame.draw.line(self.board.window, (255, 0, 0), self.start\_position, self.end\_position, 5)  
 **if** direction == **"U"**:  
 self.start\_position = self.board.player.pos  
 **for** value **in** self.board.free\_cells:  
 **if** (value[0] == self.start\_position[0]) **and** (value[1] <= self.start\_position[1]):  
 **if** (value[0], value[1] - 24) **in** self.board.walls:  
 distance = self.start\_position[1] - value[1]  
 **if** distance < current\_distance:  
 current\_distance = distance  
 self.end\_position = value  
 pygame.draw.line(self.board.window, (255, 0, 0), self.start\_position, self.end\_position, 5)  
 **if** direction == **"D"**:  
 self.start\_position = self.board.player.pos  
 **for** value **in** self.board.free\_cells:  
 **if** (value[0] == self.start\_position[0]) **and** (value[1] >= self.start\_position[1]):  
 **if** (value[0], value[1] + 24) **in** self.board.walls:  
 distance = value[1] - self.start\_position[1]  
 **if** distance < current\_distance:  
 current\_distance = distance  
 self.end\_position = value  
 pygame.draw.line(self.board.window, (255, 0, 0), self.start\_position, self.end\_position, 5)  
  
 **"""This function checks how long the user has held the power up for. If the user has collected a power up, after a  
 certain amount of time, whether the power up has been activated or not, it will be cleared from the user."""  
  
 def** check\_power\_count(self):  
 **if** self.board.player.power != **"empty"**:  
 self.board.power\_count += 1  
 **if** self.board.power\_count % 200 == 0:  
 **if** self.board.player.power == **"laser"**:  
 self.board.player.laser = **False  
 if** self.board.player.power == **"invisibility"**:  
 self.board.player.cloak = **False** self.board.player.power = **"empty"** self.board.power\_count = 0  
 self.carry\_item = **False  
 else**:  
 **return None  
  
 """If the user is carrying a power up and they press the key 'f' or 'F', this function will be called which calls  
 the power up to be activated."""  
  
 def** activate\_power\_up(self):  
 **if** self.board.player.power == **"laser"**:  
 self.activate\_laser()  
 **if** self.board.player.power == **"invisibility"**:  
 self.activate\_invisibility()  
  
 **"""When the player chooses to start a new single player or co-op game, all Items class attributes will be reset to   
 their initial values"""  
  
 def** power\_reset(self):  
 self.name\_power\_ups = [**"invisibility"**, **"laser"**]  
 self.carry\_item = **False** self.power\_ups = {}  
 self.colours = [invisibility, laser]  
 self.collect\_power = []  
 self.start\_position = ()  
 self.end\_position = ()

**Music File**

**import** pygame  
  
**"""This is the Music Class. It only has one attribute, which is volume. All the methods call different sound effects."""  
  
  
class** Music(object):  
 **def** \_\_init\_\_(self):  
 self.volume = 0.1  
  
 **"""pygame.mixer.music.load loads in the music file"""  
  
 """pygame.mixer.music.set\_volume takes the attribute self.volume. This in-built pygame function sets the volume  
 for the current music playing."""  
  
 """pygame.mixer.music.play sets the amount of times the sound effect should be repeated. If the value is -1, the  
 sound will be replayed indefinitely."""  
  
 def** menu\_music(self):  
 pygame.mixer.music.load(**"pacman\_beginning.wav"**)  
 pygame.mixer.music.set\_volume(self.volume)  
 pygame.mixer.music.play(-1)  
  
 **def** death\_music(self):  
 pygame.mixer.music.load(**"pacman\_death.wav"**)  
 pygame.mixer.music.set\_volume(self.volume)  
 pygame.mixer.music.play(1)  
  
 **def** enemy\_death\_music(self):  
 pygame.mixer.music.load(**"pacman\_eatghost.wav"**)  
 pygame.mixer.music.set\_volume(self.volume)  
 pygame.mixer.music.play(1)  
  
 **def** eating\_music(self):  
 pygame.mixer.music.load(**"pacman\_chomp.wav"**)  
 pygame.mixer.music.set\_volume(self.volume)  
 pygame.mixer.music.play(1)  
  
 **def** eating\_powerup\_music(self):  
 pygame.mixer.music.load(**"pacman\_eatfruit.wav"**)  
 pygame.mixer.music.set\_volume(self.volume)  
 pygame.mixer.music.play(1)

**Grid File**

**import** numpy **as** np  
  
**"""Stores a numpy array (which would allow me to manipulate the grid for future game updates). The '1' values represent  
a wall, the '0' values represents a free cell/tile and the '2' values represent enemy spawn."""**grid = np.array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],  
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],  
 [1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1],  
 [1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1],  
 [1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1],  
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],  
 [1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1],  
 [1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1],  
 [1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1],  
 [1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 2, 2, 2, 2, 2, 2, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 2, 2, 2, 2, 2, 2, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 2, 2, 2, 2, 2, 2, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1],  
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],  
 [1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1],  
 [1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1],  
 [1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1],  
 [1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1],  
 [1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1],  
 [1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1],  
 [1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1],  
 [1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1],  
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],  
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])

**Run File**

**from** Game **import** \*  
**from** Enemy **import** \*  
**from** Grid **import** \*  
**from** Player **import** \*  
**from** Powerup **import** \*  
**from** pygame **import** \*  
  
**"""This is the main file where I run all the game loops."""  
  
if** \_\_name\_\_ == **'\_\_main\_\_'**:  
  
 **"""Here I am creating 4 objects.  
 menu which passes the argument false (for self.terminate)  
 board which passes the argument false (for self.terminate)  
 multi\_board which passes the argument false (for self.terminate)  
 setting which passes the argument false (for self.terminate) and board which is an object for the class Board"""** menu = Menu(**False**)  
 board = Board(**False**)  
 multi\_board = MultiBoard(**False**)  
 setting = Settings(**False**, board)  
  
 **"""This is the load function. It loads in the game map and all the starting methods are called. Then the game loop  
 is initialized. It has the parameters state and difficulty. The program passes two arguments which are the state of  
 the game and the difficulty of the game."""  
  
 def** load(state, difficulty):  
 *"""If the state is Menu, the load function for the Menu class will be loaded, followed by the game loop for the  
 Menu."""* **if** state == **"Menu"**:  
 menu.load()  
 game\_loop(state, **False**)  
 **"""If the state is Play, the load function for the Board class will be loaded, followed by the game loop for  
 the Single Player game."""  
 if** state == **"Play"**:  
 **"""Before loading in the game, all the attributes of the classes are reset. This is so if the user entered a  
 game and they quit, they are able to start a brand new game without having to exit the whole program."""** board.player.player\_reset()  
 board.power.power\_reset()  
 **for** enemy **in** board.enemy:  
 enemy.enemy\_reset()  
 board.game\_reset()  
 board.load(difficulty)  
 game\_loop(state, **False**)  
 **"""If the state is Two\_Play, the load function for the MultiBoard class will be loaded, followed by the game   
 loop for the Local MultiPlayer game."""  
 if** state == **"Two\_Play"**:  
 **"""Before loading in the game, all the attributes of the classes are reset. This is so if the user entered a  
 game and they quit, they are able to start a brand new game without having to exit the whole program."""** multi\_board.player.player\_reset()  
 multi\_board.player\_two.player\_reset()  
 multi\_board.multi\_reset()  
 multi\_board.multi\_load()  
 game\_loop(state, **False**)  
 **"""If the state is Settings, the load function for the Setting class will be loaded, followed by the game   
 loop for the settings."""  
 if** state == **"Settings"**:  
 setting.load()  
 game\_loop(state, **False**)  
  
 **"""The game\_loop function takes the arguments, state and terminate. State takes the current game state being loaded  
 in, as a string. Terminate takes a boolean value that determines if the game loop should terminate or not (always  
 will start as False)."""  
  
 def** game\_loop(state, terminate):  
 *"""create\_count is used to make sure the adjacency matrix for the enemies is only created once."""* create\_count = 0  
 **"""music\_count is used to measure the time interval the music is played at while the Menu game loop   
 is running."""** music\_count = 0  
 **"""Game loop (event, draw, update) is run here"""  
 while not** terminate **and** state == **"Menu"**:  
 **"""stores the game difficulty as an integer"""** difficulty = setting.difficulty\_count  
 **"""Plays music indefinitely in the menu and restarts the soundtrack after 120 loops. """  
 if** music\_count % 120 == 0:  
 board.music.menu\_music()  
 menu.menu\_event()  
 menu.menu\_draw()  
 menu.menu\_update()  
 music\_count += 1  
 **"""If the user clicks a button within the main menu, a new state will be returned from menu.menu\_event()  
 or the program will terminate"""  
 """The difficulty is also passed so the Enemy Class and the cost function can be adjusted for the   
 difficulty selected."""  
 """If the user selects Single player, Single player will be loaded in and the game loop will start."""  
 if** menu.menu\_event() == **"Play"**:  
 state = **"Play"** load(**"Play"**, difficulty)  
 **"""If the user selects Local MultiPlayer, Local MultiPlayer will be loaded in and the game loop   
 will start."""  
 if** menu.menu\_event() == **"Two\_Play"**:  
 state = **"Two\_Play"** load(**"Two\_Play"**, difficulty)  
 **"""If the user selects Settings, Settings will be loaded in and the game loop will start."""  
 if** menu.menu\_event() == **"Settings"**:  
 state = **"Settings"** load(**"Settings"**, difficulty)  
 **"""If the user selects Exit, the game will terminate."""  
 if** menu.terminate:  
 terminate = **True  
 """Single Player Game loop (event, draw, update) is run here"""  
 while not** terminate **and** state == **"Play"**:  
 **"""Here I create an adjacency matrix for both enemies, Inky and Pinky. In addition I convert the adjacency   
 matrix to an adjacency list for Pinky as Breadth-First Search using an adjacency list."""  
 if** create\_count < 1:  
 board.inky.create\_matrix()  
 board.pinky.create\_matrix()  
 board.pinky.matrix\_to\_\_list()  
 **"""Here the game loop occurs as long as the game is not paused."""  
 if not** board.paused:  
 board.play\_event()  
 board.play\_draw()  
 board.play\_update()  
 **"""If the game is paused using the key 'p' or 'P', only two functions run.  
 board.pause\_display() updates the screen to show the text 'Paused' and board.pause\_function detects if the  
 user presses the key 'p' or 'P' to unpause the game."""  
 if** board.paused:  
 board.pause\_display()  
 board.pause\_function()  
 create\_count += 1  
 **"""If the user goes back to the main menu, the game state 'Menu' changes and the menu is loaded back in."""  
 if** board.back\_menu() == **"Menu"**:  
 difficulty = setting.difficulty\_count  
 state = **"Menu"** load(**"Menu"**, difficulty)  
 **"""If the user decides to exit from the whole game, the program will terminate"""  
 if** board.terminate:  
 terminate = **True  
 """MultiPlayer Game loop (event, draw, update) is run here"""  
 while not** terminate **and** state == **"Two\_Play"**:  
 **"""If the user turns off the Music in the settings, this if statement makes sure that Music is off in all  
 game states of the program"""  
 if** board.music.volume == 0:  
 multi\_board.music.volume = 0  
 **"""Here the main part of the game loop occurs."""** multi\_board.two\_play\_event()  
 multi\_board.multi\_play\_draw()  
 multi\_board.multi\_play\_update()  
 **"""If the user goes back to the main menu, the game state 'Menu' changes and the menu is loaded back in."""  
 if** multi\_board.multi\_back\_menu() == **"Menu" or** multi\_board.check\_winner() == **"Menu"**:  
 difficulty = setting.difficulty\_count  
 state = **"Menu"** load(**"Menu"**, difficulty)  
 **"""If the user decides to exit from the whole game, the program will terminate"""  
 if** multi\_board.terminate:  
 terminate = **True  
 """Settings Game loop (event, draw, update) is run here"""  
 while not** terminate **and** state == **"Settings"**:  
 setting.settings\_events()  
 setting.settings\_draw()  
 setting.settings\_update()  
 **"""If the user goes back to the main menu, the game state 'Menu' changes and the menu is loaded back in."""  
 if** setting.settings\_events() == **"Menu"**:  
 difficulty = setting.difficulty\_count  
 state = **"Menu"** load(**"Menu"**, difficulty)  
 **"""If the user decides to exit from the whole game, the program will terminate"""  
 if** setting.terminate:  
 terminate = **True** pygame.quit()  
 sys.exit()  
  
 **"""This game starts by calling this function. It takes the arguments 'Menu' and '0'. 'Menu' is the game state and  
 '0' is the difficulty the game loads in (which is easy)."""** load(**"Menu"**, 0)

**Extra Files:**

* Music Files:
  + I imported 5 different .wav music files
    - Pacman\_beginning.wav
    - Pacman\_chomp.wav
    - Pacman\_death.wav
    - Pacman\_eatfruit.wav
    - Pacman\_eatghost.wav
* Image Files:
  + I imported 3 images, of .png and .jpg
    - Settings.jpg
    - Maze.png
    - Menuscreen.jpg
* Text Files:
  + I have two text files.
    - One text file contains the high score of the user.
    - The second text file contains the help text that the user will be able to see in the settings in the game.