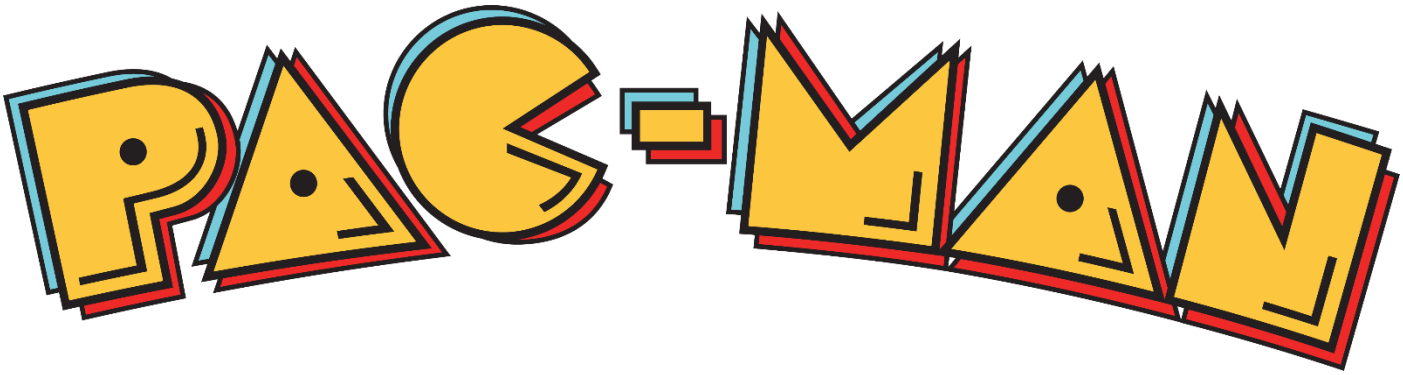


A MULTIPLAYER
EXPERIENCE

CREATED

BY WILLIAM EVANS

TABLE OF CONTENTS

Analysis 1
Problem Definition 1
Background Research 1
Maze design 2
Path finding 3
Pac-Man VS. 4
Identification of End Users 5
Objectives 5
1 - Start Screen 5
2 - Sign-up Screen 6
3 - Login Screen 6
Models 10
Basic Multiplayer Model 10
Maze representation 11
Networking 11
A* 13
Design 14
Project Technology 14
High-level overview 15
Mainloop 15
Menu System 16
Database 18
High Level Description 18
Main 18
Splash Screens 19
Tutorial 19
Single Player 20
Multiplayer 20
Sprites 21
Pathfinding 21
Data Structures 21
Multiplayer Sprites 22
Networking 22
GUI 22

Local Database 23
Local Settings 23
User Interface 24
Start Screen 24
Settings 25
Classic Mode 26
Tutorial 27
High Scores 28
Sign Up 29
Login 30
Accounts 31
Multiplayer 32
Multiplayer – Host 33
Multiplayer – Client 35
Class Descriptions 37
Class Diagram 37
Class Description Table 40
Function Description Table 58
Key Algorithms 60
A* 60
SQL 62
Networking 62
Data Structures 68
Priority Queue 68
Maze 69
Technical solution 71
Key Algorithms 71
Code 71
Main 71
Splash Screens 73
Tutorial 85
Single Player 96
Multiplayer 104
Sprites 104
Pathfinding 149
Data structures 152

Multiplayer Sprites 157
Networking 167
GUI 174
Local Database 189
Local Settings 196
Testing 197
Strategy 197
Video 198
Plan 198
Key 198
Table 199
Evaluation 209
Objective Analysis 209
User Feedback 213
Possible Extensions 214

ANALYSIS

PROBLEM DEFINITION

Versions of the game Pac-Man pop up everywhere nowadays, usually, with the same core principals as the original game but with slightly different maze design and graphics. On occasion, they include extra features which make the game more appealing to a younger audience, but this loses the simple beauty of the original Pac-Man games. Therefore, I will create a multiplayer game based on Pac-Man that will address both the spinoffs' issues – (by keeping the core principals of the game) and the originals' by applying the tried and tested mechanics to a more exciting multiplayer platform.

While multiplayer versions of Pac-Man have been made in the past, the majority (if not all) revolve around 2 player versions in which both players play as Pac-Man. On the contrary, I believe the most effective design will come from a 5-player game that preserves the original gameplay by allowing the other human players to play as the ghosts instead. It will also incorporate even more features, such as maze generation (in keeping with the original map design); local accounts (storing game statistics), and another game mode: 'Tutorial'. This will allow beginners to learn how to effectively play against the ghost AI.

Further to my proposal to expand the functionality with a multiplayer mode, it is vital that my game achieves a balance at all times (meaning it is equally easy/difficult to play the game regardless of what 'team' you are on). In the single-player version, this will not be a problem as it will feature various AIs controlling the ghosts that all have certain flaws (based on the tried and tested original Pac-Man); one human ghost, however, will not (let alone four). To solve this problem, I will introduce a mechanic that restricts the vision of the ghosts. The game will have the same dimensions, but the majority of the tilemap will be in darkness for them. The ghosts, therefore, will require a lot of teamwork in order to locate and catch Pac-Man and allows some re-playability with teams able to experiment with different strategies.

It is also important that my game has a built-in account system. This will allow the player to connect more easily with friends and encourage competition within the single-player modes through high scores. In order to accomplish this, I will have to build a local login and signup system, using a database with a 'User' table (to store usernames and passwords) and also a 'Game' table, which will store information about games played.

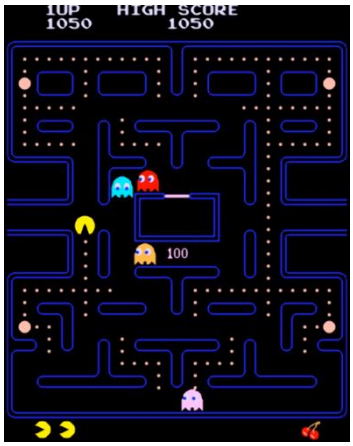
In order to create my game, I will use the python library 'PyGame', as this allows me to create a perfectly sized window based on a tilemap and add sprites and sound effects to mimic the quality of the original game. The library also makes moving and logging interactions between multiple objects across the screen many times a second very easy. These features make 'PyGame' a very capable alternative to another (more basic) python GUI framework like 'TkInter'. For the networking aspects, I will use python's 'Socket' library. While not the easiest networking library to use, its basic approach means I can accomplish more specific and advanced features when compared with a higher-level library like 'PyGame'. When approaching the database design, I thought it would be most appropriate to use 'Sqlite3'. It is easier to use than other, comparable libraries and I already have some experience with it. Whilst it does lack some features, I don't plan on implementing anything too advanced within the database so it will suffice.

BACKGROUND RESEARCH

In this section, I will explore the key components that make up the game 'Pac-Man' and briefly comment on how I could implement these into my game.

My game will build on the core features in the original Pac-Man (unlike many other spinoffs). Pac-Man is a game that runs using hidden tiles usually 28x31 but with 3 tiles above and 2 below for game information making the total screen size 28x36. You play as a yellow circle (Pac-Man); the objective is to eat all the smaller circles which appear in the middle of every tile (excluding wall pieces) at the start of each game/round. There are four ghosts that each, with different methods, try and catch Pac-Man by touching him.

In order to balance the game, the ghosts are unable to make a 180° turn (unless switching modes), and there are larger circles in the corners of the maze that (when eaten) allow Pac-Man to eat the ghosts for a limited time – giving the player points and putting the ghosts out of action of a few seconds while they return to the centre. This ghost 'mode' is called 'scared'.



Mode	Level 1	Levels 2-4	Levels 5+
Scatter	7	7	5
Chase	20	20	20
Scatter	7	7	5
Chase	20	20	20
Scatter	5	5	5
Chase	20	1033	1037
Scatter	5	1/60	1/60
Chase	indefinite	indefinite	indefinite

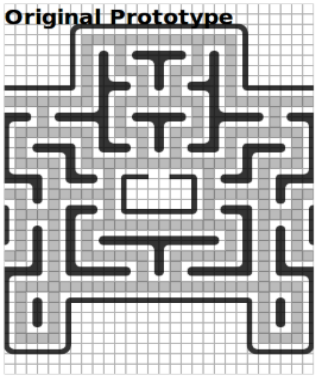
There are 3 modes: chase, scatter and frightened. The ghosts chase for a set time and then switch to scatter mode in which they no longer chase Pac-Man and instead head straight for one of the four corners each. When a power pellet is eaten, the ghosts change direction and enter scared mode whereby they make random movements and can be eaten by Pac-Man. The amount of time spent in these modes vary as each level progresses and from level to level (as shown in the image on the right).

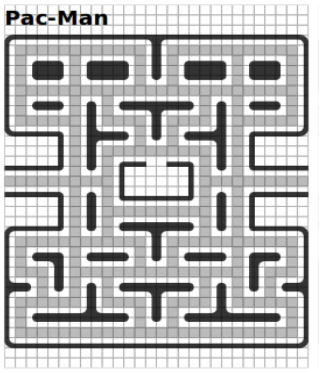
Furthermore, Pac-Man is also able to take advantage of a technique known as cornering. As Pac-Man's hitbox is smaller than the tiles, he can 'hug' walls and turn corners more quickly than the ghosts who are required to centre themselves in the middle of a tile before they are able to change direction.

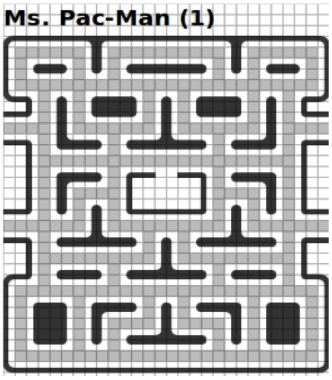
MAZE DESIGN

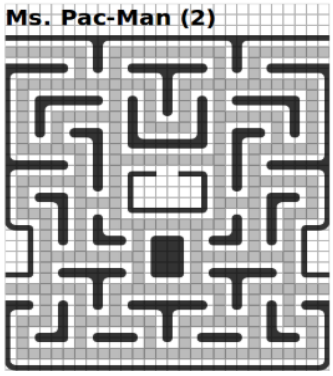
In Pac-Man, the mazes are designed and hard-coded into the game with the following set of rules:

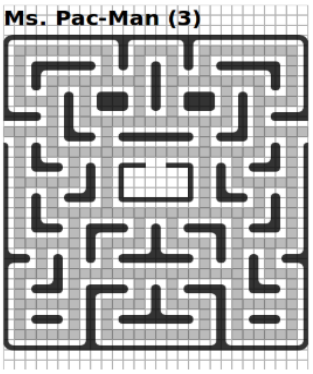
- The maze is 28x31 tiles
- Paths are only 1 tile thick
- No sharp turns (i.e. intersections are separated by at least 2 tiles)
- There are 1 or 2 tunnels
- No dead-ends
- Only I, L, T, or + wall shapes are allowed, including the occasional rectangular wall
- Any non-rectangular wall pieces must only be 2 tiles thick
- They are symmetrical down the centre

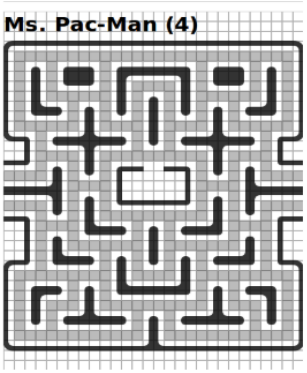












These constraints are visualized in the above abstractions of the maps in various evolutions of Pac-Man (not including the prototype which is an example of a poor maze - as a result of not following the rules). If I am to create an algorithm that can randomly generate mazes that resemble Pac-Man (and thus keep the core gameplay), all I need to do is program the algorithm to comply with the above rules.

This idea sounds simple, but after doing some more research I do not believe it is feasible to create such an algorithm in my given timeframe. In fact, there have been many papers written on the subject, and the most elegant solution I found online is roughly 784 lines long and took the experienced programmer around 2 months to implement. However, I would still like a randomly generated maze algorithm so I will look into machine learning – more specifically a genetic approach to solving this problem.

After looking into machine learning, it appears a lot of knowledge would be required to pursue this method of maze generation. This knowledge will simply take too long so instead I will experiment with my own simplified version of the maze generation algorithm in which random pieces are chosen and added to a half a blank maze. If it doesn't fit, it is moved, or another piece is chosen until one side is complete. The maze will then be mirrored onto the other side. This should still give a random maze every time, but there will be a smaller total number of possible mazes.

PATHFINDING

In Pac-Man, there are four ghosts, each with different approaches to catching Pac-Man. A graph is generated to represent the maze, then the ghosts navigate using slightly different versions of a graph traversal algorithm based on the Euclidean distance to catch Pac-Man. In simple terms, it is the shortest path algorithm and the four ghosts use it in different ways in order to catch Pac-Man.

'Blinky' uses the simplest form of the algorithm using Pac-Man's tile as the target tile.

'Pinky' uses a target tile 4 tiles ahead of Pac-Man.

'Inky' uses the tile two in front of Pac-Man, finds the vector between Blinky and that tile, doubles the vector and uses the other end of that vector as the target tile.

'Clyde' uses a tile in the bottom left corner when not within 8 tiles of Pac-Man. Otherwise, he uses Pac-Man's.

Whilst even the original game ran at 60fps on very lacking hardware, I will use a very efficient pathfinding algorithm known as A* in order to reduce the games hardware demands. This is because, at this stage of development, I am not certain how taxing the maze generation algorithm will be on the hardware and therefore I will need all the processing power I can get to ensure the final product runs as smooth as possible. It's worth noting that the original game was written in a very efficient language: C. On the contrary, I am coding in python which is a notoriously inefficient language (all the more reason to focus on efficiency in my implementations at this stage). A* is different from the Euclidean distance method as instead of checking every possible path it ignores extremes and so can find the most efficient path much more quickly. Here is a more in-depth explanation:

To find the fastest route to a given tile, A* must first receive a weighted graph of the maze. Then, you would pass through the start tile and end tile. A tree of paths is created which will store all the possible paths as they are created (the first path will simply be the start tile). A* then follows these steps.

1. Children (possible next tiles) are created at the closest tile in the tree to the target tile.
2. Repeat step 1 until the closest tile to the target tile is the target tile.

As you can see the steps are extremely simple and allow the algorithm to backtrack if there are no available tiles (children) on the closest tile to the end tile. The backtracking allows the algorithm to navigate around obstacles and by choosing the tile closest to the target tile the algorithm can ignore inefficient paths.

The shortest route is calculated in step 1. It does this based on the cost of the path and the estimated cost of extending the path all the way to the target node, specifically by minimizing $f(n) = g(n) + h(n)$ where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n and $h(n)$ is the estimated distance to the target node from the next node. This estimated distance is calculated using a heuristic (sloppy, but very fast) function. As long as the heuristic function is at least admissible (returns either the cost of getting to the final node or a cost lower) A* will always return the most efficient path to the end node. This can be done by simply ignoring all walls and using the number of squares there are between the start and end node.

PAC-MAN VS.

Pac-Man Vs. is one of the most popular Pac-Man spinoffs to incorporate multiplayer functionality. It does this by allowing 4 local players (1 as Pac-Man and 3 as ghosts). The graphics of the game are different from the original game in that the characters have been redrawn in a 'higher quality' 3D environment, but also with slightly different aesthetics to the originals. The game mode is also different on Pac-Man Vs.

The winner of the game is the first to reach a predetermined number of points. You earn points as Pac-Man by eating ghosts, fruit, pellets and power pellets. Once Pac-Man is caught by the ghosts, whichever ghost caught Pac-Man will become Pac-Man in the following round. In order to balance the game, the ghosts can only view a small radius of the map around them.

I will build on the success of this game which is the well balanced and very replayable game mode. Like Pac-Man VS. I will include a feature that limits the ghosts' vision and include a game mode very similar to its own. However, unlike the spin-off I will stay true to the original game; I will use the original character sprites as I want to be able to keep some of the elements of the original Pac-Man game. Furthermore, I will also expand on the multiplayer functionality by offering an online (multiple machines) rather than local (single machine) multiplayer experience. This has never been done before in a Pac-Man game and I feel it is a necessary step for the franchise in order to stay relevant nowadays.

IDENTIFICATION OF END USERS

Pac-Man has no age barrier. This is because even the youngest of people can enjoy the game thanks to its simple design and controls. I would say that my game will be aimed at 4+, but of course, it is not limited to this age range. In fact, the beauty of this game is that it appeals to children and adults alike. In order to accommodate the age range of 4 – 6 years (according to Debra Levin Gelman's Design for Kids – below), I need to break up instructions, allow invention and self-expression and provide immediate feedback. In order to accomplish these things, I will incorporate the following features:

- A short tutorial (which will be my tutorial mode) with broken up instructions (the tutorial will exist over multiple levels).
- The multiplayer will credit all players with the place they came (allowing for clear accomplishment).
- Sounds and little score indicators that pop up when Pac-Man eats an enemy and sound effects to reward the user (providing immediate feedback)

By following these key principles, I can not only accommodate the younger generation, but also the older and more competitive generation. Furthermore, high score system can be very easily implemented using a relational database that will encourage healthy competition in these higher skill/age bands.

Table 2—Design guidelines for developmental categories

2–4 year olds	4–6 year olds	7–9 year olds	10–12 year olds
Highlight only things with which a child can interact.	Break up instructions into manageable chunks.	Clearly show goals and purpose.	Provide contextual Help after a failure.
Use few colors.	Allow invention and self-expression.	Offer a clear set of rules, with opportunities to interpret and expand them.	Emphasize self-expression and accomplishment.
Use icons that are highly representational.	Provide immediate positive feedback.	Let kids earn and collect awards.	Invite silliness or irreverence.

OBJECTIVES

In this section I will outline and further expand on the main points and objectives I have for my project, starting with how the role of each of my pages, and then moving on to gameplay and game modes.

1- Start Screen

1.1 - Music that plays whenever the start screen is running.
1.2 - A list of choices that become highlighted when a mouse hovers over them and can be clicked to choose the game mode.

1.3 - Icons that can be used for extra functions such as muting the music or launching other modes like the sign-in pages.

2- Sign-up Screen

2.1 - The screen should have three input boxes that can be highlighted by either by clicking or pressing tab. These will have names: 'Username', 'Password', 'Confirm Password' which display inside the box until the user enters something.

2.2 - There should be a 'Sign Up' button that the user can click in order to submit their inputs. This should query the database to ensure the inputs are valid. If not, an appropriate error message should be displayed, or, if it is correct then the user should be taken to the login screen. The password should follow the following rules: be seven characters or more; contain an upper-case letter; contain a lower-case letter and contain a number.

3- Login Screen

3.1 - Like the sign-up screen, this page should have input boxes, but this time only the 'Username' and 'Password' boxes are needed.

3.2 - There should be a sign-up button that changes to a login button when the user enters something into the username field.

4 - Settings Screen

4.1 - There should be sliders that control, semi-discreet values, such as music volume, game volume etc. These settings should be updated in a text file which the rest of the game can then use to apply the changes.

4.2 - Any changes made in the settings should be saved in a settings JSON file.

5- Gameplay

5.1 - The maze

5.1.1 - The maze should be saved as a two-dimensional list in a text file or database.

5.1.2 - My program must then be able to take this two-dimensional list and convert each item (a number) and turn this into an appropriate tile object, which will be in a similar two-dimensional list.

5.1.3 - Each of the tile objects must contain which image of the 36 different tile pieces it requires, that will be assigned to it by an algorithm that checks all adjacent tiles to find the piece that will connect graphically with the ones around it.

5.1.4 - The tile object should also contain its position within the maze that means it can be quickly displayed (60 times a second) in every frame along with the other tiles.

5.1.5 - When the user wins a level by eating all the pellets, the maze should flash white and blue.

5.2 - Pac-Man

5.2.1 - Pac-Man must be able to respond to keyboard input and validate it. For example, if the user wants to go north (using the up-arrow key), then the Pac-Man object should check what tiles are above (by referencing the 2D maze list and its own position) and decide whether this is a valid move (if there is a free space above).

5.2.2 - Pac-Man must be able to stop when he collides with a wall. When a wall tile is in the direction that Pac-Man is facing, a check should be made to see whether Pac-Man is colliding with the wall.

5.2.3 - Pac-Man should be able to collide (and eat) pellets and power pellets, although I feel it would be easier to handle this within the pellet object.

5.2.4 - The classic 'waka-waka' sound should play only when Pac-Man is moving.

5.2.5 - When moving, Pac-Man's skin should also alternate between his slightly open and fully open mouth.

5.2.6 - Pac-Man should also be able to travel through the tunnel at normal speed but not into the ghost hut.

5.3 - Ghosts

5.3.1 - Ghosts will have 4 modes: chase (this is when the ghosts will target Pac-Man), scatter (this is when the ghosts will target their home tiles), scared (this is when the ghosts make random moves and can be eaten by Pac-Man after a power pellet has been eaten) and finally dead (this is when the ghosts move very quickly and return to the 'ghost hut'. These modes must last the correct lengths of time (this will be highlighted in the design) and have the correct skins.

5.3.2 - The ghosts must all follow their own unique chase algorithm as follows: Blinky - targets Pac-Man; Pinky - targets 4 tiles ahead of Pac-Man; Inky - Takes the vector between Blinky and Pac-Man, doubles it, then targets that tile; Clyde - targets Pac-Man until he gets close to Pac-Man.

5.3.3 - The ghosts must use the A* pathfinding algorithm when searching for a tile, as this will allow the game to run smoothly at 60fps.

5.3.4 - The ghosts must only change direction when they are changing modes.

5.3.5 - The ghosts must play a death sound when eaten.

5.4 - Pellets

5.4.1 - There should be two kinds of pellets: the normal pellet (small square skin), and the large power pellet (larger circular skin).

5.4.2 - The small pellet should remain static, while the power pellets should flash.

5.4.3 - The small pellet should grant the user 10 pts, whereas the power pellets should grant the user 50pts and change the mode of the ghosts to 'scared' mode.

5.4.4 - There should be 4 power pellets on each maze with the majority of the other non-wall tiles containing normal pellets. The spaces around the ghost hut, inside the 'ghost hut' and the tunnel should all be empty (no pellets).

6 - Tutorial Mode

6.1 - The tutorial mode must include a narrator that explains the game's mechanics to the user.

6.1.1 - This should be in the form of a scrolling message.

6.1.2 - When the message fills the box, the user should be able to press the space bar to skip to the next part of the message.

6.1.3 - If the user presses space before the message has finished then the message should skip to the end and immediately fill the box.

6.2 - The levels should progressively get harder and each should focus on a different aspect of the full game.

6.2.1 - The first level should focus on teaching the user about how to move Pac-Man with the arrow keys in order to collect all the pellets.

6.2.2 - The second level should introduce the ghost Clyde and explain his behaviour (that he only targets Pac-Man when he's at least 8 tiles away.

6.2.3 - The third level should introduce Pinky.

6.2.4 - The fourth level will include the most aggressive ghost: Blinky, and explain that when there are few pellets left, Blinky turns into Elroy.

6.2.5 - Level five will introduce power pellets, as a way to counter the ghosts (specifically Blinky).

6.2.6 - Level six will add the final ghost: 'Inky', who must be accompanied by Blinky due to his chase mechanic.

6.3 - Pac-Man should not have any lives in tutorial mode. Instead, he will simply respawn an infinite amount of times, with the pellets carrying on to each subsequent level.

6.4 - After the scripted levels have finished the user should be able to play on randomly generated mazes.

7 - Classic Mode

7.1 - The classic mode should have a high score at the top of the screen that is taken from the local database.

7.2 - This should also see (like the original game) Pac-Man with just 3 lives (gaining a 4^{th} at 10k pts).

7.3 - The mode should feature just one maze (the original) in order to as closely resemble the original game as possible.

7.4 - When the user has run out of lives, they will be prompted to enter three initials. These will then be associated with the game they just completed and will also appear on the high scores (provided the score of that game is in the top ten).

8 - High scores

8.1 - The high score page should feature the top ten scores achieved in classic mode (as long as the games have initials attached to them).

8.2 - This information should be formatted into 3 columns in the following order. Place (i.e. 1^{st}, 2^{nd}, 3^{rd}), score then initials. This, again, is to stay true to the original game.

8.3 - The top 3 scores should be coloured gold, silver and bronze.

8.4 - When any key is pressed on this page, the user will be returned to the menu screen.

9 - Multiplayer

9.1 - Multiplayer menu

9.1.1 - As with all the multiplayer menus, there should 5 avatars on the screen. 4 in small boxes at the top and a central larger one. At this stage, the top four will be ghosts and the large central one will be Pac-Man. They will all be greyed out.

9.1.2 - There will be two options underneath this: ‘Create Game’ and ‘Join game’

9.2 - Create Game menu

9.2.1 - The central avatar should now be coloured in yellow (as Pac-Man).

9.2.2 - The current user’s username should be displayed beneath the Pac-Man avatar, along with a score of 0.

9.2.3 - This should instantiate a server object and thus allow other users to join the lobby.

9.2.4 - There will be a box containing the game ID (host’s local IPV4 address) that other users can use to join the lobby.

9.2.5 - When a user joins, they should take one of the avatar slots up (and thus the appropriate coloured-in ghost skin should appear there along with their name and a score of 0).

9.2.6 - At any time, the host is able to start the game by clicking the start button in the bottom right-hand corner. This will stop any other users from joining and begin a countdown.

9.3 - Join Game menu

9.3.1 - The avatars remain the same as the previous multiplayer menu.

9.3.2 - There is an input box underneath the central avatar that the user can input a game ID into.

9.3.3 - When the user joins a game, the central avatar box will be filled in with the ghost they have been allocated and their name and score of 0 will appear underneath in place of the game ID box. Any other user’s in the lobby will appear along with their avatars, names and scores along the top row.

9.3.4 - The user will have the option to ready up in this position. This will add the word ‘Ready’ under their score. This will then be visible to anyone else in the lobby.

9.4 - Gameplay

9.4.1 - Unlike the Classic game mode, there should be no lives nor a high score text at the top of the screen. The only indicator will be the user’s current score.

9.4.2 - Each user should start as the avatar that was in their central box in the lobby.

9.4.3 - Pac-Man should be able to gain points by collecting pellets and eating the other players.

9.4.4 - The ghosts should be able to work together in order to catch Pac-Man. They should also receive points for being close to Pac-Man and for eating him.

9.4.5 - When a ghost eats Pac-Man they gain 1600 pts, and everyone is returned to the lobby where the countdown is automatically started and whoever ate Pac-Man swaps characters with Pac-Man.

9.4.6 - Whoever gets to 20k points first will win the game. When this score is reached in a game, after the round has finished, the users are returned to the lobby where their respective places are displayed.

9.4.7 - 1st, 2nd and 3rd should be coloured gold, silver and bronze respectively.

10 - Database

10.1 - The database should store information about every game that is played on a computer.

10.1.1 - If a Classic game is played there should be an entry in a ‘Game’ table that will store information about the user that played the game (if logged in) and what the game mode was (in this case classic).

10.1.2 - There should also be an entry into the database for each level played. This will store in-game stats like the number of pellets eaten, how long the level took etc.

10.1.3 - There should also be another table: 'GameLevel'. This should simply link the two tables together which will allow complex queries to be carried out about a single game regarding statistics built up within each level.

10.2 - The database should store information about users.

10.2.1 - When a user is created using the sign-up menu an entry into the 'Users' table should be added.

10.3 - The database should be able to carry out complex queries.

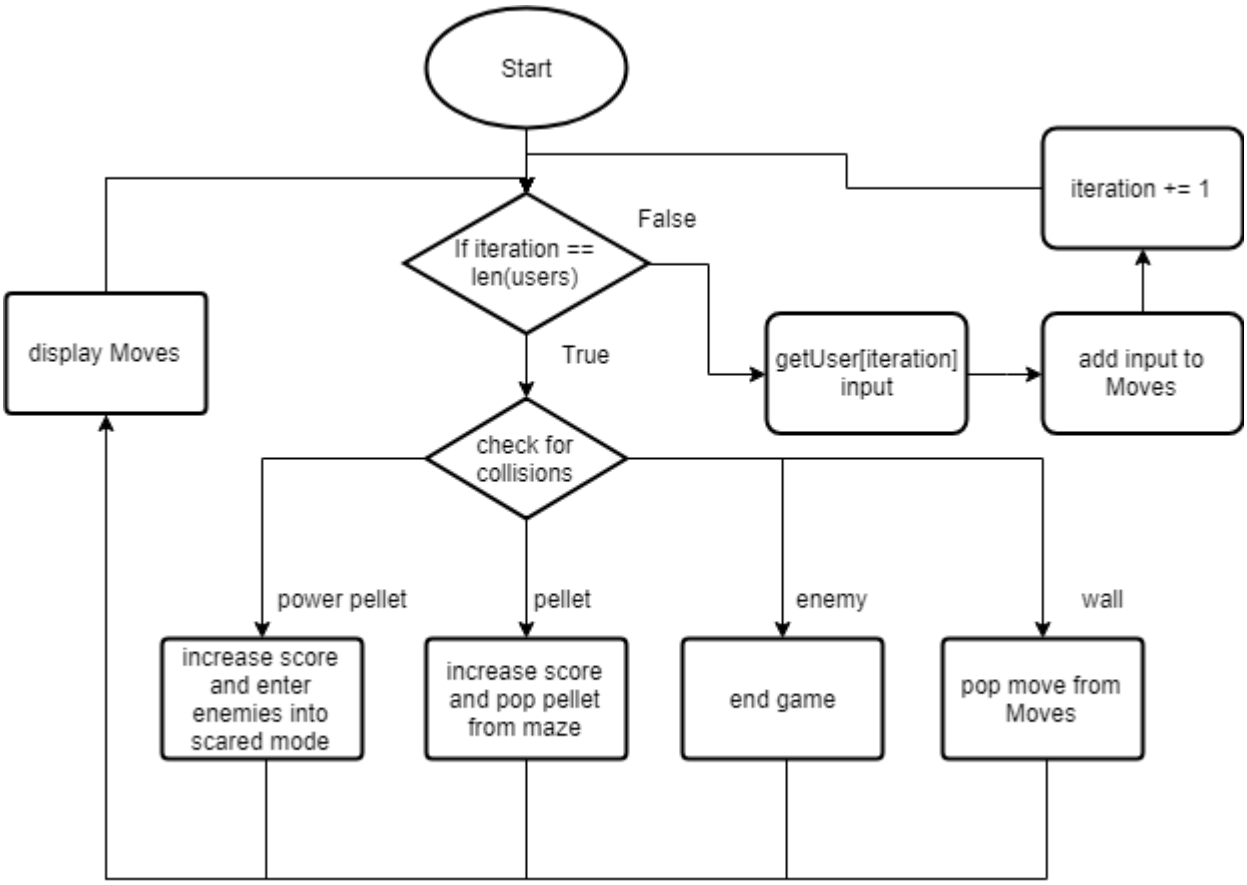
10.3.1 - We should be able to gather the top 10 scores (in order) in Classic games from the database.

10.3.2 - The database should be able to take login details and verify them.

10.3.3 - The database should also be able to take sign up details and verify that they fit the correct format for the database and that there will be no duplicates.

MODELS

BASIC MULTIPLAYER MODEL



MAZE REPRESENTATION

In order to draw the first level or the test level (before I develop the maze generation algorithm). I need a way of quickly creating the Pac-Man map. To do this I created a program that allows you to draw maps. As the map is symmetrical, I only drew half of it and then used another algorithm to flip it.

```
import pygame
from sys import exit

class Maze:
    def __init__(self):

        self.__maze = [list(x) for x in [[int(x) for x in '0' * 28]]*31] # Makes sure lists
aren't duplicates

    def display(self, win):
        win.fill((0, 0, 0))
        for x, row in enumerate(self.__maze):
            for y, tile in enumerate(row):
                if tile == 1:
                    colour = (20, 10, 255)
                else:
                    colour = (000, 0, 0)
                pygame.draw.rect(win, colour, (y * 10, x * 10, 10, 10))

        pygame.display.update()

    def update(self, win, ev):
        for event in ev:
            if event.type == pygame.QUIT:
                print(self.__maze)
                exit(0)

        for event in ev:
            if event.type == pygame.MOUSEBUTTONUP:
                pos = pygame.mouse.get_pos()
                xmouse = int(round(pos[0], -1) / 10)
                ymouse = int(round(pos[1], -1) / 10)
                self.__maze[ymouse][xmouse] = abs(self.__maze[ymouse][xmouse] - 1)


    def getMaze(self):
        return self.__maze

def run(win, maze, ev = []):
    clock.tick(10)
    maze.display(win)
    maze.update(win, ev)
    return win, maze

if __name__ == "__main__":
    pygame.init()
    win = pygame.display.set_mode((280, 310))
    clock = pygame.time.Clock()
    maze = Maze()
    while True:
        win, maze = run(win, maze, pygame.event.get())
```

NETWORKING

I came up with the following idea for the use of networking.

I would have a Server (whoever creates the game) and multiple clients. When the clients make a move i.e press a key it updates their move variable. This is a constant process. On another thread, the client is always listening for the server to send a maze that it can display. When it receives the maze, it sends the client's next move and displays it.

The server listens for moves from the clients all the time and updates its move variables for each player when it receives one. When the server has processed the maze, it sends it to all clients and then uses its current move variables to draw the next one.

This model disconnects various processes so that they cannot cause each other to crash. For example, if a client was to disconnect, then, the server would simply continue using their last move to draw the new board. By not waiting for the user to send the next move, the game will continue to run smoothly for the other clients. Of course, the server can still detect when someone has disconnected because it expects a reply from all users after it sends a maze, but, it does not require nor wait for it. This, again, ensures that if a particular user is experiencing high latency, then the other users are not affected.

I have written the following python code to express this model:

```
import threading, socket, json


class Connection:
    def __init__(self, userIP, conn, ID):
        self.__ID = ID
        self.__HOST = userIP
        self.__PORT = 50007
        self.__fps = 60
        self.__conn = conn
        self.__move = {}

    def receiving(self):
        while True:
            try:
                data = self.__conn.recv(1024)
                self.__move = json.loads(data)# {'left arrow key': True, 'spacebar' : True}
etc...
            except ConnectionResetError:
                return "Connection Lost"

    def sendBoard(self, board):
        self.__conn.sendall(bytes(json.dumps(board), 'utf-8'))

    def getMove(self):
        return self.__move

    def getID(self):
        return self.__ID


class Server: #Instanciates whenever a user clicks 'create game'.
    def __init__(self):
        self.__IP = socket.gethostbyname(socket.gethostname())
        self.__port = 50007
        self.__connections = []
        self.__Threads = []
        self.__s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.__s.bind(('127.0.0.1', self.__port))

    def connect(self, ID):
        self.__s.listen(1)
        conn, addr = self.__s.accept()
        connection = Connection(addr[0], conn, ID)
        self.__connections.append(connection)

    def startReceiving(self):
        for connection in self.__connections:
            self.__Threads.append(threading.Thread(target=connection.receiving).start())

    def getMoves(self):
        moves = {}
        for connection in self.__connections:
```

```
            moves.update({connection.getID(): connection.getMove()})
        return moves

    def sendBoard(self, board):
        for connection in self.__connections:
            connection.sendBoard(board)

    def endGame(self):
        pass


class Client:
    def __init__(self, hostIP):
        self.__host = hostIP
        self.__port = 50007
        self.__s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.__move = {}
        self.__board = {}

    def connect(self):
        self.__s.connect((self.__host, self.__port))
        threading.Thread(target=self.receiving).start()

    def receiving(self):
        while True:
            data = self.__s.recv(1024)
            self.__board = json.loads(data)
            self.sendMove()

    def updateMove(self, move):
        self.__move = move

    def sendMove(self):
        print(self.__move)
        self.__s.sendall(bytes(json.dumps(self.__move), 'utf-8'))

    def getBoard(self):
        return self.__board

    def endGame(self):
        self.__s.close()
```

A*

I will develop my own A* algorithm that works with the way I have formatted my maze and will base it on following pseudocode (sourced from Wikipedia) which shows the basic theory behind the A* search algorithm.

```
function A_Star(start, goal)
# The set of nodes already evaluated
closedSet := {}

# The set of currently discovered nodes that are not evaluated yet.
# Initially, only the start node is known.
openSet := {start}

# For each node, which node it can most efficiently be reached from.
# If a node can be reached from many nodes, cameFrom will eventually contain the
# most efficient previous step.
cameFrom := an empty map

# For each node, the cost of getting from the start node to that node.
gScore := map with default value of Infinity

# The cost of going from start to start is zero.
gScore[start] := 0
```

```
# For each node, the total cost of getting from the start node to the goal
# by passing by that node. That value is partly known, partly heuristic.
fScore := map with default value of Infinity

# For the first node, that value is completely heuristic.
fScore[start] := heuristic_cost_estimate(start, goal)

while openSet is not empty
   current := the node in openSet having the lowest fScore[] value
   if current = goal
      return reconstruct_path(cameFrom, current)

   openSet.Remove(current)
   closedSet.Add(current)

   for each neighbour of current
      if neighbour in closedSet
         continue            # Ignore the neighbour which is already evaluated.

      # The distance from start to a neighbour
      tentative_gScore := gScore[current] + dist_between(current, neighbour)

      if neighbour not in openSet   # Discover a new node
         openSet.Add(neighbour)
      else if tentative_gScore >= gScore[neighbour]
         continue

      # This path is the best until now. Record it!
      cameFrom[neighbour] := current
      gScore[neighbour] := tentative_gScore
      fScore[neighbour] := gScore[neighbour] + heuristic_cost_estimate(neighbour, goal)
```

DESIGN

PROJECT TECHNOLOGY

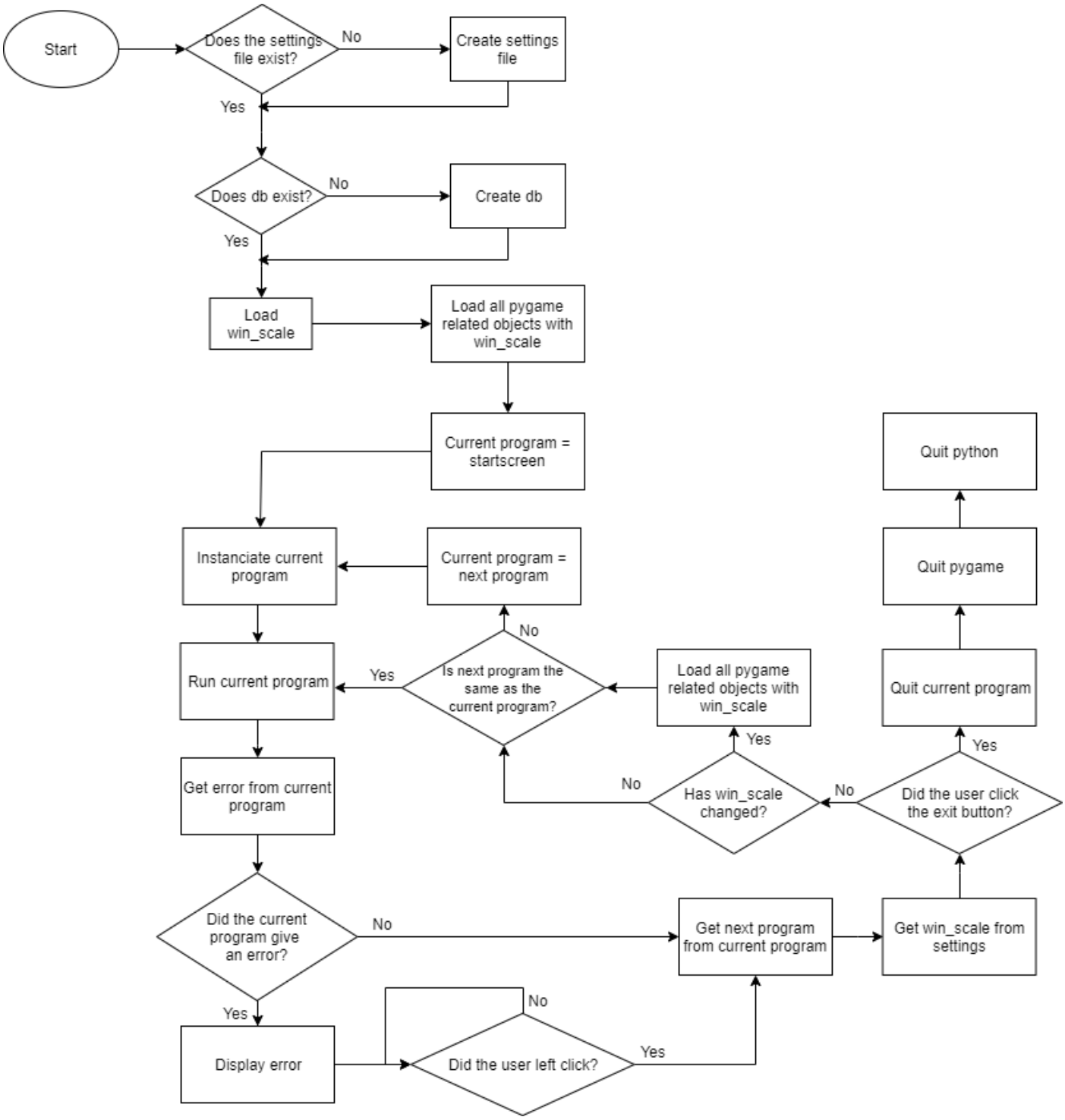
For this project, I will be utilising the python programming language (mostly version 3.7) and will be using the PyCharm integrated development environment to help support the final implementation of the project. I will also be using the following libraries:

- PyGame; this will allow me to display my game and handle various game-related tasks.
- Socket; python sockets will allow me to create a multiplayer game mode.
- Threading; using threads is essential when sending and receiving data on a network, but the pathfinding algorithm could also be run off of a thread to avoid any lag within the game that could occur.
- JSON; this will make saving mazes in the database easier and will also allow me to send dictionaries via sockets.
- Random; this will allow ghosts to make random moves when they are in scared mode.
- Copy; this will allow me to use deepcopy() which is needed when copying multidimensional data structures. This will be especially useful when resetting player data in my networking script, as it is formatted as a multidimensional dictionary.
- os; I will use os to check whether the database/settings need to be rebuilt and to create paths in a more concise manner than using just strings.

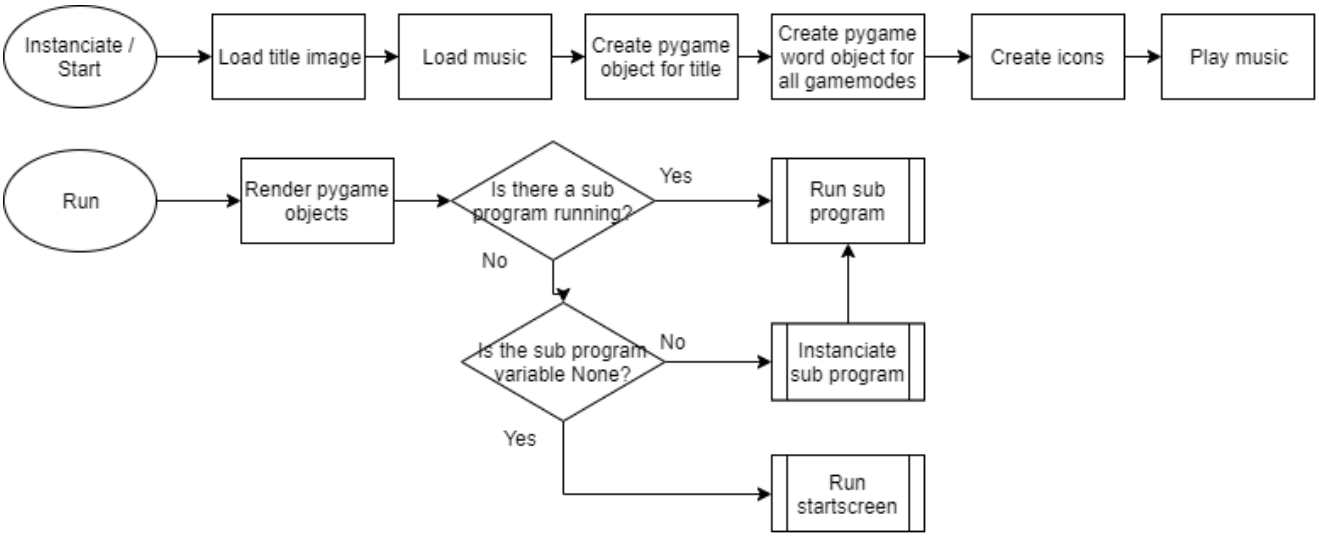
HIGH-LEVEL OVERVIEW

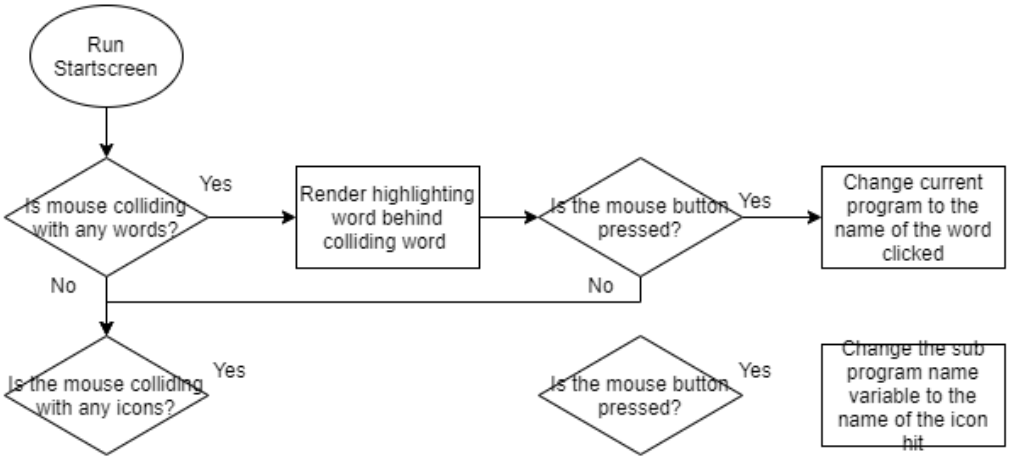
This section will contain flow charts outlining the key events and structure of my game. Due to the various menus and game modes, I separate each game mode and menu area into a separate chart.

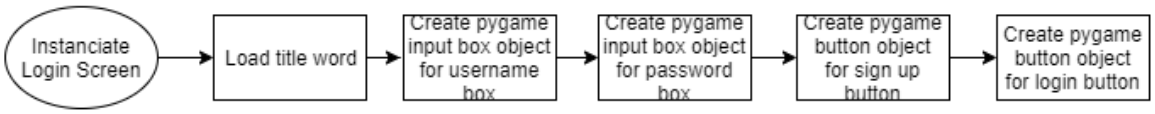
MAINLOOP

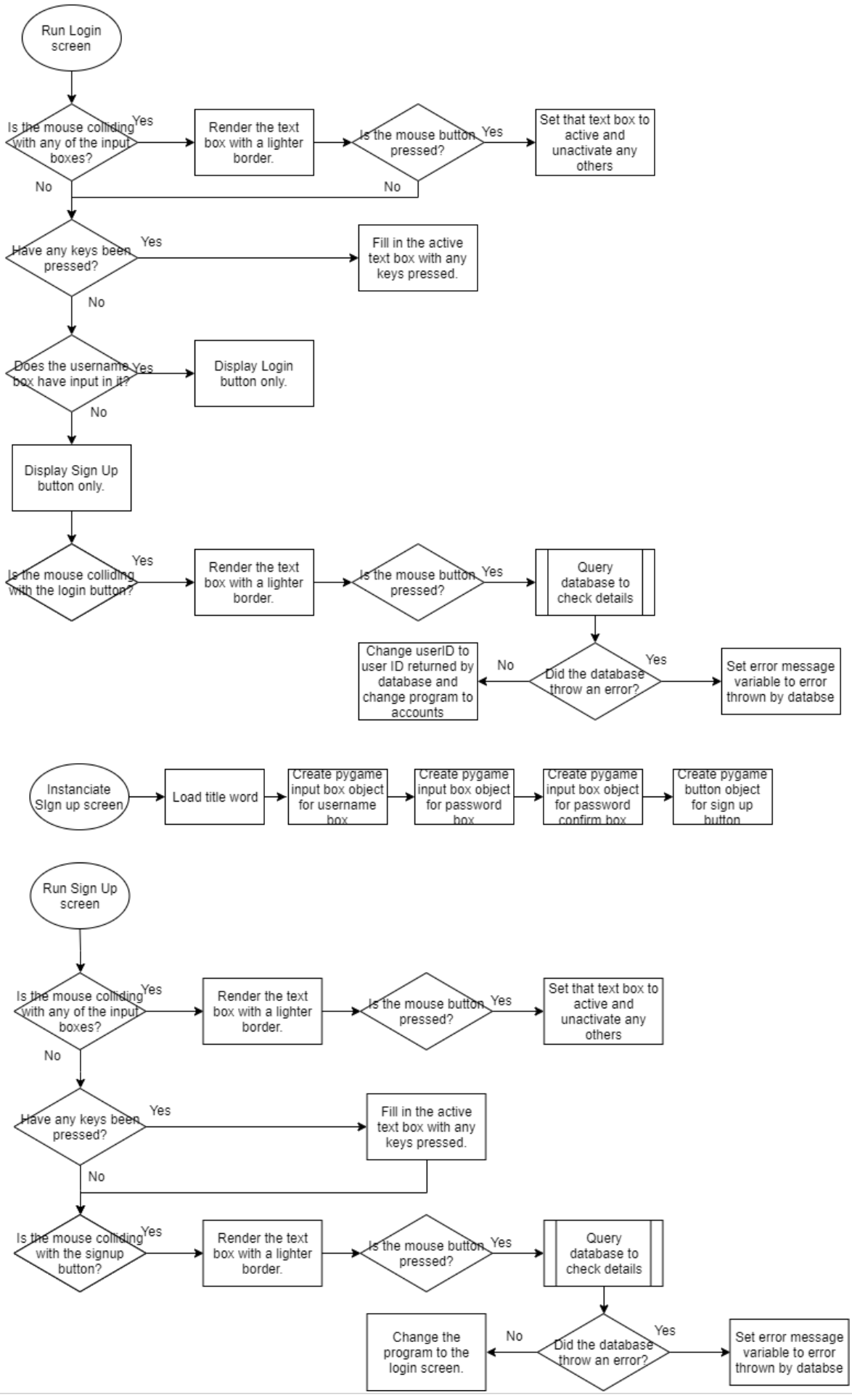


MENU SYSTEM

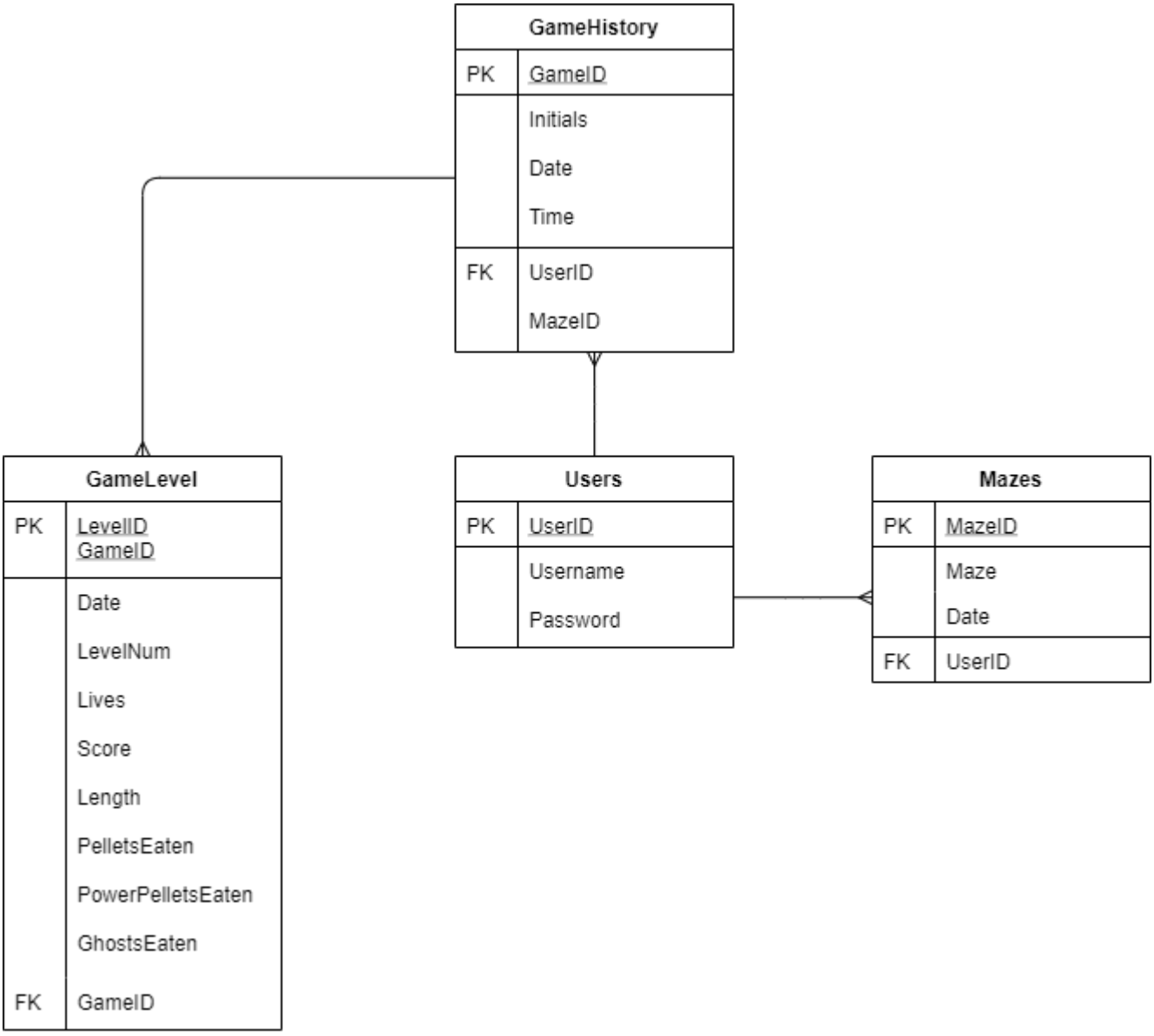

Instanciate / Start
Load title image
Load music
Create pygame object for title
Create pygame word object for all gamemodes
Create icons
Play music
Run
Render pygame objects
Is there a sub program running?
Yes
Run sub program
No
Is the sub program variable None?
No
Instanciate sub program
Yes
Run startscreen


Run Startscreen
Is mouse colliding with any words?
Yes
Render highlighting word behind colliding word
Is the mouse button pressed?
Yes
Change current program to the name of the word clicked
No
No
Is the mouse colliding with any icons?
Yes
Is the mouse button pressed?
Yes
Change the sub program name variable to the name of the icon hit


Instanciate Login Screen
Load title word
Create pygame input box object for username box
Create pygame input box object for password box
Create pygame button object for sign up button
Create pygame button object for login button


Run Login screen
Is the mouse colliding with any of the input boxes?
Yes
No
Render the text box with a lighter border.
Is the mouse button pressed?
Yes
No
Set that text box to active and unactivate any others
Have any keys been pressed?
Yes
No
Fill in the active text box with any keys pressed.
Does the username box have input in it?
Yes
No
Display Login button only.
Display Sign Up button only.
Is the mouse colliding with the login button?
Yes
Render the text box with a lighter border.
Is the mouse button pressed?
Yes
Query database to check details
Did the database throw an error?
Yes
No
Set error message variable to error thrown by databse
Change userID to user ID returned by database and change program to accounts
Instanciate SIgn up screen
Load title word
Create pygame input box object for username box
Create pygame input box object for password box
Create pygame input box object for password confirm box
Create pygame button object for sign up button
Run Sign Up screen
Is the mouse colliding with any of the input boxes?
Yes
No
Render the text box with a lighter border.
Is the mouse button pressed?
Yes
Set that text box to active and unactivate any others
Have any keys been pressed?
Yes
No
Fill in the active text box with any keys pressed.
Is the mouse colliding with the signup button?
Yes
Render the text box with a lighter border.
Is the mouse button pressed?
Yes
Query database to check details
Did the database throw an error?
Yes
No
Set error message variable to error thrown by databse
Change the program to the login screen.

DATABASE



HIGH-LEVEL DESCRIPTION

Below is a high-level description of each of my project files that will show what each file should achieve in the final solution. You may notice there is substantial detail regarding the main script. This is only because other areas of my code are written as functions and classes and are therefore elaborated on in the class and function descriptions, whereas my main loop is neither.

MAIN

The main script should be the target whenever starting the game. It will handle the running of all other scripts in a manner outlined above by the high-level overview describing the main loop. To reiterate, its main tasks will be controlling which scripts are able to run and executing key PyGame code. This will include things like controlling the frame rate and also running the code 'pg.display.update()' that will allow any changes made by the scripts to actually be displayed.

The Mainloop will be able to run the following game areas (that are controlled by objects): 'Start Screen', 'Story', 'Classic', 'Multiplayer', 'Create Game', 'Join Game' and 'High scores'. As you can see, not all of the game's features are listed here. I have done this as for the classes that require communication with the list of objects here, it is very difficult to facilitate this communication through the main loop. In fact, to do this, the Mainloop would need to be programmed to accept every specific piece of information that theoretically would need to be sent by any of these classes. That is why I have chosen to allow these high-level classes, such as 'Classic' to be able to run the lower-level ones themselves which in that case would be 'Level'.

There is, however, some information that the Mainloop can and must receive from each of the classes listed. That is the 'program', 'error message' and 'userID'. The program is the program that should be executed by the main script (as programs are allowed to decide if the target program should be changed i.e. returning to the main menu). By allowing the classes to choose what these programs are, we are able to easily create navigation menus within the classes without having to run them within that class and filling up the call stack. If the class runs into an error, then it can specify what error this was and the Mainloop can then display it. The reason this is handled in the Mainloop and not the class is not only to save duplicated code but also so that as soon as the error message is clicked, we are able to switch the control to a different program. An example of this is when the Multiplayer class throws an error if the user is not signed in and thus kicks them back into the main menu where they can then navigate to the accounts area. The userID is actually something that only the login subprogram can change, but it is handled by the main script as it is needed for multiplayer and there is no other way for these 2 areas of my code to communicate.

SPLASH SCREENS

This file should contain the classes responsible for displaying the various splash screens. Whilst not containing the screens relating specifically to the multiplayer mode (as these splash screens need direct communication with aspects of the multiplayer mode), it will include all others: 'Start Screen', 'Sign Up Screen', 'Login Screen', 'Settings', 'Accounts', and 'High-scores'.

TUTORIAL

This script will look after the tutorial mode section of the game. It must utilise a general level class which can be adapted (through inheritance) to be used for the many different levels. The parent level class consists of running the game itself, by keeping track of all the sprites (the same as the level class in Classic mode). The parent class must also keep track of winning conditions (i.e. when all pellets have been eaten). When Pac-Man is eaten by a ghost the level also handles this event by restarting the level (since there are an unlimited amount of lives in the tutorial mode).

On the other hand, the higher-level classes (specific to each level) manipulate the main level class by only adding certain ghosts. This is useful as the user can learn about each ghost one by one. A big feature of the tutorial mode will be the narrator. The narrator will share tips with the user for defeating each ghost and will guide the user throughout the level. In order to implement the latter, the messages and more specifically the conditions for when these should be displayed must be hardcoded into the individual level classes.

The individual levels will follow the following description:

Level 1	A tutorial on how to use the arrow keys to move Pac-Man and collect all the pellets in order to win a game.
Level 2	This level will introduce Clyde.
Level 3	This level will consist of Pinky.
Level 4	This level will introduce Blinky.
Level 5	I believe level 5 will be a good point to introduce power pellets.
Level 6	Level 6 will introduce inky and Blinky (as inky requires Blinky to function).
Level 7	This will be the final tutorial-like level that will feature all of the ghosts and allow the player to use all of the knowledge they have learnt/been given on the previous levels.

SINGLE PLAYER

The single-player script controls the 'Classic' game mode. While my other game modes offer something different over the original game, the 'Classic' mode stays true to the original. Since the original game is an infinite loop, the code for this game mode isn't too complicated (especially since a lot of the logic is stored within the sprite objects).

Contained in the script are two classes: the 'Classic' class and the 'Level' class. The level class contains the references to all the sprites and can be run by the main script which in turn calls various methods within the sprites. Other responsibilities of the Level class include but are not limited to: control of the delay at the start of a game; checking when a game is won (all pellets have been eaten) and thus telling the maze object to start flashing; deciding when a game is lost and telling the Pac-Man sprite to play the death animation; counting the score.

The 'Classic' class, on the other hand, seeks to control the level class instances. Its main responsibilities are again as follows: how many lives Pac-Man has left; when the game ends (if Pac-Man has no lives); the saving of high scores; the correct shutting down of the level class (closing of all threads correctly).

MULTIPLAYER

Like the previous two descriptions, the multiplayer controls relatively little of the game logic (as most of this is contained in the sprites that these scripts utilise), however unlike the previous two, the multiplayer contains many different menus. For a graphical analysis of the menus, you can read the 'User Interface' section. These menus make up 3 of the 5 classes in this section. They allow the user to choose between creating or joining an online game and also execute these two choices, guiding the user through the process through well-positioned visual cues.

These menus also act as controllers for the two other classes 'HostLevel' and 'ClientLevel' in a similar way that the 'Classic' class controls the levels in single-player, however, the multiplayer classes are required to handle the player lobby and usage of the server. They also (as well as the 'Multiplayer' class) use the functions in the script to gather the necessary GUI components to visually represent the lobby.

Unlike the single-player mode, 'ClientLevel' and 'HostLevel' use different sprites (that are able to receive input from a server which is also controlled by these level classes). These multiplayer sprites inherit all the methods from the sprites used for the single-player modes, but a few are overridden (such as the method of receiving a move) and some are added (such as updating the server with the move). There is more on the adaptations of the multiplayer sprites in the dedicated section below.

SPRITES

This script looks after any object within the gameplay that has collision mechanics. This includes Pac-Man, all the ghosts and pellets.

Pac-Man and the ghosts are based on a single Class: 'Sprite'. This includes all the necessary methods to display something on the screen. However, this sprite class cannot function on its own as it only includes the methods that both Pac-Man and the ghosts have in common, which isn't enough to create an independent sprite. Instead, the basic methods are inherited by the higher-level classes 'Pac-Man' and 'Ghost'. Then, the movement methods are overridden and other methods completely unique to the two are added (more detail on the specific methods in the class description table).

While the Pac-Man class can now be used to create a Pac-Man object, the Ghost class must go through one more level in order to be usable. This is because there are 4 unique ghosts that each hunt Pac-man in a different way (as explained in the analysis). There are four more classes ('Blinky, 'Pinky', 'Inky' and 'Clyde) which override the method of chasing Pac-Man and also their start position, home tile and behaviour at the start of the game (when they leave the ghost hut).

The final sprite in the script is the 'Pellet'. The pellet class simply controls the normal pellets and power pellets in the game. They keep track of collisions between themselves and the 'Predator' assigned to them (which is Pac-Man). The reason I had to include this argument is that the instance of Pac-Man changes between levels, but the Pellets do not (and thus their predator must be changed between levels). When a pellet collides with Pac-Man it is set to 'dead' and thus removed from the list of pellets.

PATHFINDING

As shown in the previous models, the pathfinding script will include an adaptation of the A* pathfinding algorithm represented by a class (so that the maze can be saved). This script also uses inheritance to override the 'Search' class' heuristic function. This is to allow multiple heuristics to be tested, I have included these in the design so that I may use them in the testing stage later in order to prove the efficiency benefits of using Manhattan over Euclidean.

The script also includes two functions 'get_children' and 'in_closed' as these are used frequently in the main loop of A*. As they are static (they do not use any class attributes) it is good to practise to include these as functions and not methods.

The script makes use of the 'Priority Queue' data structure (defined as a class in the 'data structures' script) along with the maze class that is given as an argument from the game modes the pathfinding algorithm is used in.

DATA STRUCTURES

The data structures script contains the code for the 'Priority Queue', 'Node', 'Maze' and 'Tile'.

The priority queue is used when deciding which nodes to explore next in the A* pathfinding algorithm. They are given numerical priority based on their f score which is partly calculated by the heuristic used in the algorithm. The node simply stores this score, its position in the maze, its parent and an attribute called facing. This is the direction the ghost would be facing if it were to choose that path and it prevents nodes behind it being evaluated. This allows the mechanic that is 'ghosts cannot turn around' to be implemented.

The maze and tile objects are used by many parts of the code to represent the maze. The tile object stores the type of tile (which is decided based on the 2D list tilemap), the tile's position, the hitbox (used for collisions) and the skin (the image that should be used when displaying that specific tile). The maze simply

groups these tiles into a 2D list in the same format as the tilemap. It also has methods that allow the creation of these tiles (by determining what skin to use based on the tile's position) as well as two methods that enable the displaying of the maze using either white or blue tile skins.

MULTIPLAYER SPRITES

The multiplayer sprites script (as touched on before) contains sprites that (using the classes defined in the sprites class) are able to be used in multiplayer. To do this they are given methods that allow them to update the server data.

There are two main types of multiplayer sprite: 'Server' and 'Client. Within these, there are classes controlled by humans and ones controlled by a computer. Within those, there are Pac-Man and ghost. This totals to 13 different sprites (including the ones for individual ghosts i.e. Blinky, Pinky etc.). There is a method in the multiplayer script that sorts out which sprites must utilise which class in order to function.

As an example: if I was hosting a match and playing as Pac-Man for the first game, then my sprite would be 'ServerPlayerPacMan'. This would mean I could use keyboard inputs to control Pac-Man and whatever score was given to the sprite would be updated to the server (as Client sprites cannot edit the server's score- this is to improve the robustness of the system, reducing the ways the scoring system could be exploited by players). Furthermore, on my machine, I would need a 'ServerGhost' sprite that would not take input from the server as these pathfinding calculations occur on the host machine. If I was playing with other players, however, I would also need some 'ServerPlayerGhost' sprites, in order to take input from their machines.

NETWORKING

The networking script contains 3 classes: 'Server', 'Connection' and 'Client'. These classes are used

The first two ('Server' and 'Connection') are used together to form to allow networking on the host machine (someone who creates a game). The server stores the main dictionary, storing the data that the multiplayer uses in order to display the game and communicate with the other machines. The server's job is to allow editing and transmission of this data. It also has a thread that continuously listens for new connections (until the lobby is full i.e. 4 connections have been made). When a new connection object is created, and the connection is stored in this. After the initial trade of information about the host and client, another thread listens for player data from the client, whilst the server sends out data along with another method in the connection that is triggered by the 'HostMenu' class on the host machine.

There are many other methods in the Server class that manipulate the data in order to change things about the lobby (such which machine is playing as which character and more). There are also several 'get' methods that are no apparent in many other areas of my code. They appear here as it is absolutely vital that no data is edited outside of the class, whereas in other areas it is somewhat essential to have this feature in order to keep my code clear and concise.

The 'Client' class also uses a thread to receive data from the server and (like the server) updates its own copy of the data so that it can be accessed then by the 'ClientLevel' class. Data from the multiplayer sprite controlled by the client is also sent over the client class so that it can be evaluated by the server (this includes making sure it is a valid move and sending back information as to whether it is or not).

GUI

Arguably the most important aspect of my game is its graphical representation. This will be stored It is what the end-user will interact with and forms the basis of what the game actually is. For the game itself, I will mostly use a sprite sheet, however, I will also need to construct a GUI, using my own basic framework to

form the various menu screens. All the necessary classes to achieve this will be stored in the 'GUI' script, which will include the following classes:

- Live word - This will control any large title like words that can react to a hovering mouse and mouse clicks.
- Icon - This will control any interactive pictures (icons) in my game.

These are the classes I have prototyped so far, however, I feel I will need the following also:

- Word - This will be used for any basic words that do not need to react to any input.
- Box - This class would simply describe a box that could respond to mouse input.
- Buttons - This could be used in the future to begin a game or ready up in a multiplayer match and would use the programming technique of aggregation by combining the word and box class.
- Input box - Again this class would be an example of aggregation (by combining the box and word class) and could be used for sign-up and login forms.
- Sliders - This class could be used for settings requiring a semi-continuous input. i.e. volume. Again, we could use aggregation here by incorporating the box and word class.
- Scrolling word - This class would most likely use the box and word-class and would work by slowly revealing whatever sentence needs to be output to the screen.

These classes will be used to together to quickly build and efficiently run a graphical menu system. They can also be used in aspects of the core Pac-Man gameplay.

LOCAL DATABASE

This script is relatively simple. It contains all the functions to recreate the database if it is not detected, this includes a function for each table in the database. There are also more functions for each query and complex that is needed for other areas of my code. A few of the more complicated queries will be explored later, but there are also many more, such as: 'login' which checks whether a user's login request matches the records in the database'; 'check_sign_up' which checks whether the password meets the requirements and if the username is already taken. Almost all of the queries in this script use a function called 'query' which can handle data fetches or just normal executions. It handles sqwlite3 header to keep the code concise.

LOCAL SETTINGS

The settings script, like the database, contains a function that creates a settings file (formatted as JSON to allow to more easily be worked with).

There is another function for saving settings which reads the file, changes the value using dictionary manipulation and then re-saves all the settings again. I believe this is a suitable way of doing this, as while it is not the most efficient way, as the amount of settings (win_scale, music_volume, game_volume, and saved login details) is very small I have favoured simpler code.

The final function is 'get_setting', which simply retrieves a setting by again converting the file into a dictionary object and then returning the data corresponding to the settings requested.

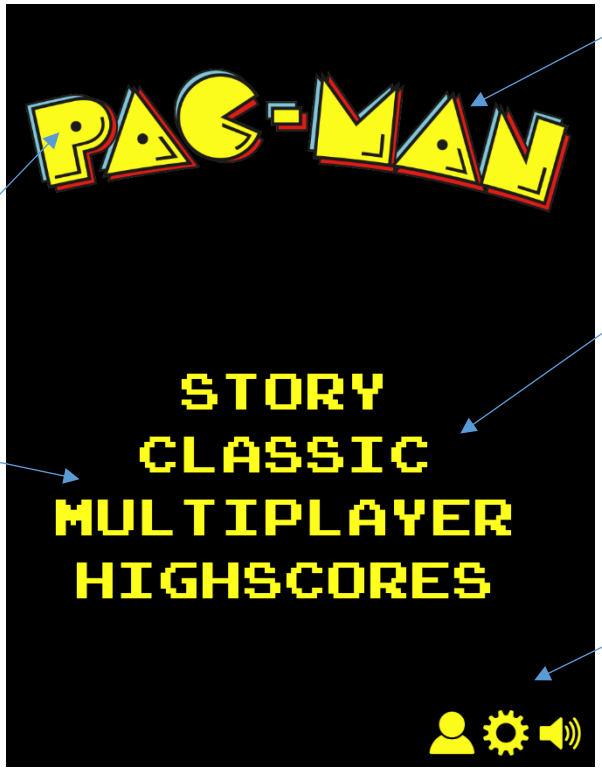
USER INTERFACE

Outlined below are all the main pages of my game including how they are designed, why they are designed this way and screenshots of prototypes.

START SCREEN

The start screen is the most important 'splash screen' in my game. It must look clean and exact, whilst allowing the user to easily navigate the game. I will do this by outlining the 5 main options: 'Tutorial', 'Classic', 'Multiplayer', 'Creator' and 'High scores'. This is different from my analysis as I have removed the settings and achievements sections. Instead, these sections are accessible from the icons (screenshots below) as this gives space for the creator and high scores sections without making the start screen too cluttered.

The yellow colour on the logo and words is slightly different to the original prototype. I made this adjustment to match the colour of Pac-Man more accurately and also to make the screen a little brighter / more exciting.

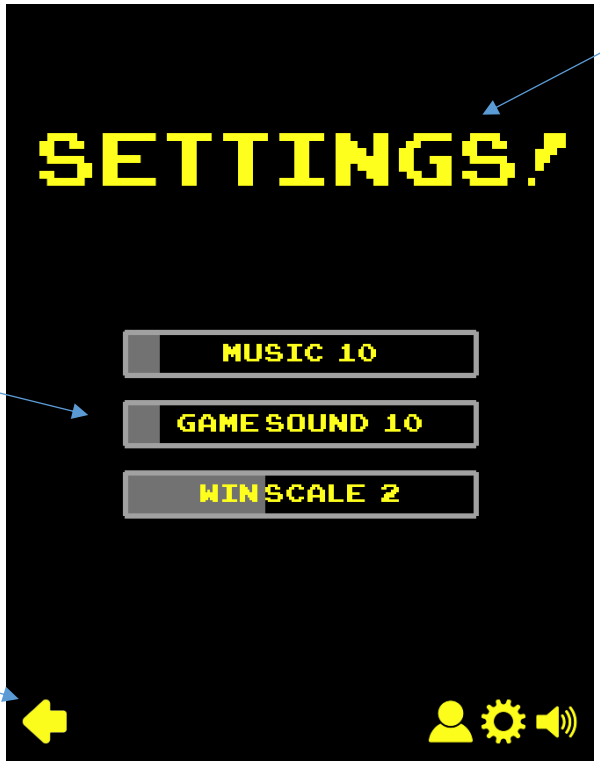


The options are interactive. When the user hovers over one with their mouse a highlighting colour will appear behind it. This alternates between blue and red as you go down in order to match the logo.

The options have been reshuffled. The mute icon is still present along with an accounts and settings icon. This is to reduce the number of options. I chose these things to be icons as they both have images associated with them that are easy for a user to recognise.

SETTINGS

The settings screen a subprogram of the main menu. For this reason, I chose to keep the icons in order to be able to navigate easily between the main menu, accounts and settings. When thinking of how the user should be able to interact with the settings, I felt the cleanest but also most intuitive way would be to use sliders. Further to this, I chose to only include the 3 settings I felt would be most important: the sound on the main menu, the in-game sound and the win-scale.



Large title that matches the majority of the other splash screens to maintain a continuity within the program.

These sliders control the 3 most important settings.

Icons that match the main menu and other programs connected to the main menu to prevent the user feeling overwhelmed by the menu system.

After repeated comments during interviews, I have also added a back arrow so that users can operate the menu system using only the mouse.

CLASSIC MODE

There wasn't a massive amount of creative innovation in the design of this area. The screen is based entirely on the original game and while the sprite sheet is slightly different (offering ironically, I slightly more pixelated/retro look than the original), it is fairly akin to the original.

Current score.

1 up indicator, flashes until the extra life has been earned.

High score.

Black background to contrast the brightly coloured ghosts and Pac-Man.

Life indicators. These let the user know how many lives they have left.

TUTORIAL

The tutorial is a diluted version of the core gameplay mechanics that allows the user to learn the various aspects of the main game. For this reason, I removed a lot of the features of the original game graphically and also added a friendly narrator in 'Ms. Pac-Man' who talks the user through the game.

The only indicator I have kept is the score, so that the user can still get a feel for what actions gain the most points.

The sprite Ms.Pac-Man is the tutorials' narrator. She has two sprites (open mouth and closed mouth) so she chomps like Pac-Man when the text is scrolling

I have added tutorial messages over the top of the game as to not intrude the game too much, whilst still making it stand out enough to seem important to the user.

HIGHSCORES

Whilst the high score page is meant to mimic the original, I have made some changes in order to fit more closely with the theme I set out when designing my menu screen.

Large title that matches the majority of the other splash screens to maintain a continuity within my program.

Gold, silver and bronze colours for 1st, 2nd and 3rd respectively.

Message written in the yellow colour and arcade classic font in order to match the rest of the game.

SIGN UP

I wanted to keep the game's GUI as clean as possible, which is one of the reasons I chose to format the sign-up screen in this way. There is the usual large title that accompanies every splash screen (and then the button and icons since the sign-up screen is part of the menu system), but also very little in the way of an explanation as to how the user should manoeuvre this page. This idea heavily relies on the user being able to figure it out for themselves which is also why I have designed this screen to be as similar to other sign up screens as possible. There are boxes with their desired inputs pre-typed. These boxes light up when the user hovers over them and can be clicked to enable typing in them. The user can also press tab to select the username and then to switch between the boxes.

Large title that matches the majority of the other splash screens to maintain a continuity within my program.

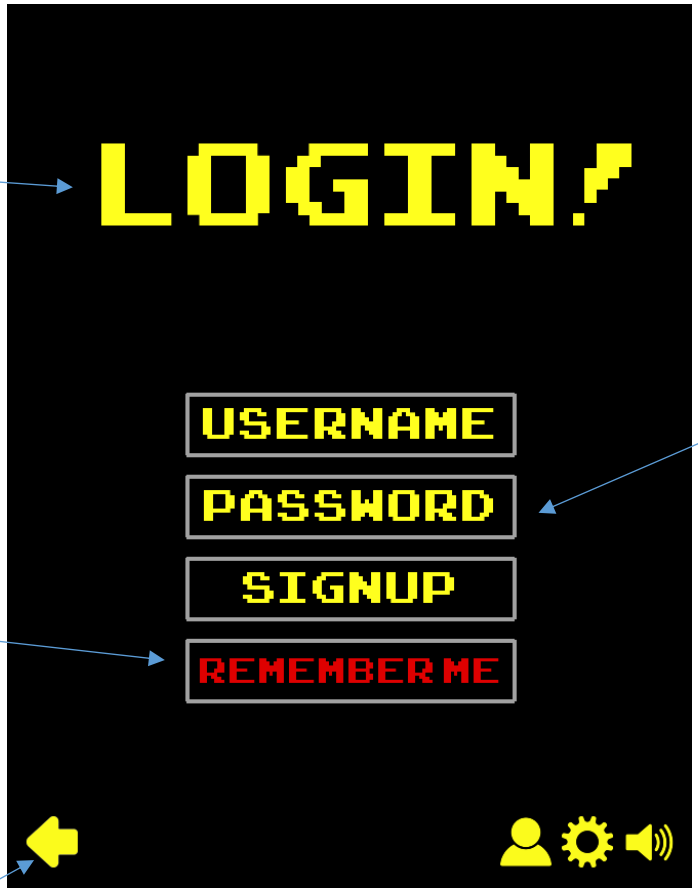
Equally sized and spaced input boxes to improve readability.

Sign Up button beneath the input boxes and in the same style as the input boxes to provide even more continuity.

Back button to enable easy traversal of the menu system.

LOGIN

My login screen is formatted in a very similar way to my sign-up screen. As you can see there is a couple of differences (other than the title at the top). First of all, there is only one password box (as on the sign-up screen you had to confirm the password); the button has been moved up and a remember me toggle button has been placed underneath. This button has colours matching that of the main menu screen. You may also notice that there is also no login button. In order to keep the login screen as clean as possible, whenever the user enters something in the login boxes the signup button changes to a login button. If the user clicks the signup button before this then they will be taken to the sign-up screen.



Large title that matches the majority of the other splash screens to maintain a continuity within my program.

Equally sized and spaced input boxes.

Remember me button with colours matching the menu screen.

Back button to enable easy traversal of the menu system.

ACCOUNTS

The accounts section allows the user to view who they are logged in as and also view their statistics. Aligned on the right are the names of each statistic and aligned to the left are those numbers. There is a logout button at the bottom and also a remember me button with the same colours as the one on the login screen.

Large title that matches the majority of the other splash screens to maintain a continuity within my program.

Statistic numbers aligned to the right.

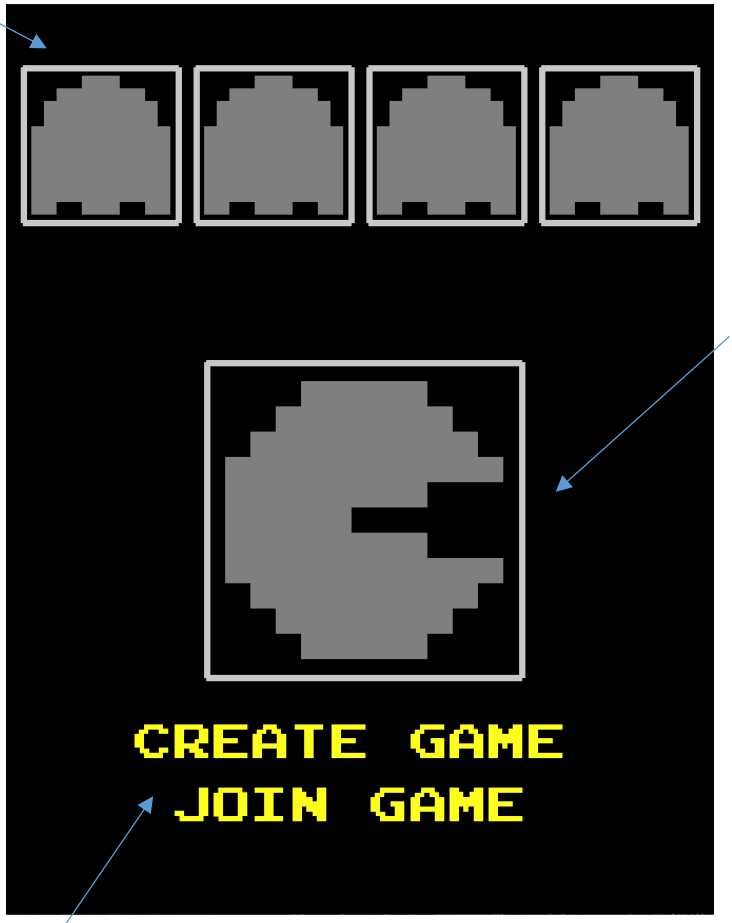
Back button to enable easy traversal of the menu system.

Log out button.

MULTIPLAYER

Being the main (and most complex) feature, it is very important that the multiplayer aspect of the game is polished and consistent. To achieve this, I have decided to have a basic GUI structure that would remain constant in all multiplayer menu screens. This involved (as shown below) having 5 boxes with sprites inside. (the default is 4 ghosts along the top and a central box with a greyed-out version of Pac-Man. This would later form the lobby but including this on the first screen reduces the perceived number of different multiplayer screens the user must navigate through.

4 equally sized and space boxes containing greyed-out ghost sprites.



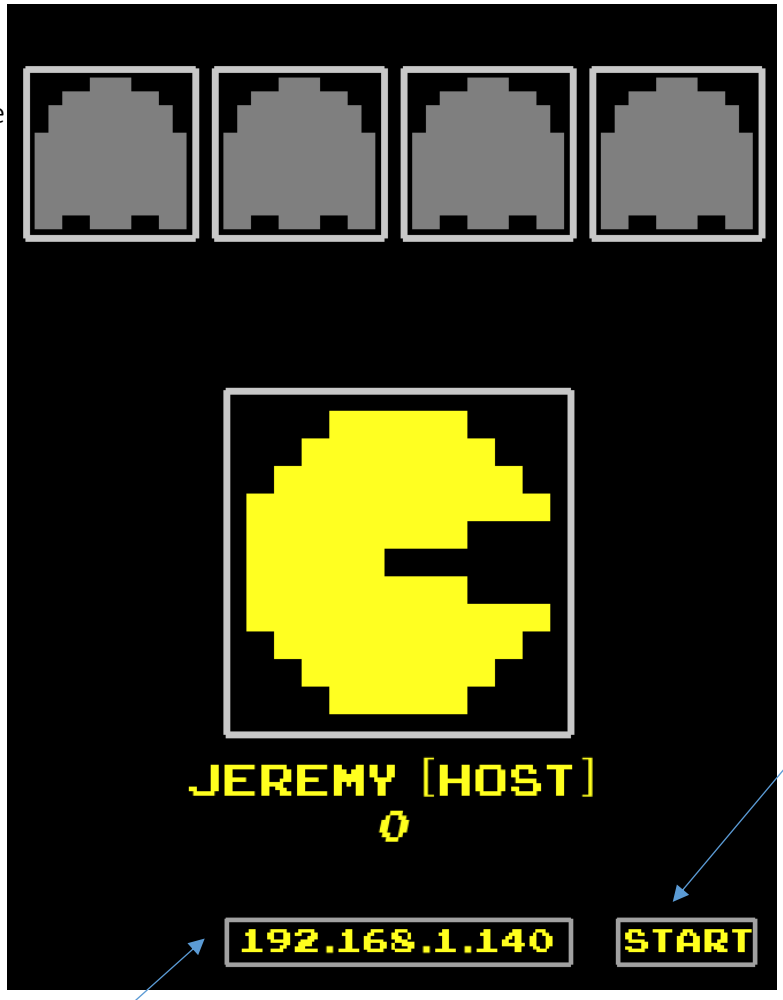
A single large box that will eventually contain the user's character.

Two simple options using the same interactive word objects as the menu screen, but this time using a white highlight behind instead of red and blue. This is to differentiate it from the menu screen.

MULTIPLAYER – HOST

If the user selects the 'Create Game' option, they will be taken to their own lobby. Since they have entered the lobby their name appears under the central avatar, which has now become full colour. As they are the first person in the lobby they will get to play as Pac-Man. The 'Start' button has also appeared which acts in the same way as the toggles (so that the user can stop the start timer). As shown below the start yellow start button turns into a red cancel button when clicked and a timer is started in place of the game id box. The GameID box is positioned in line with the buttons and centralised.

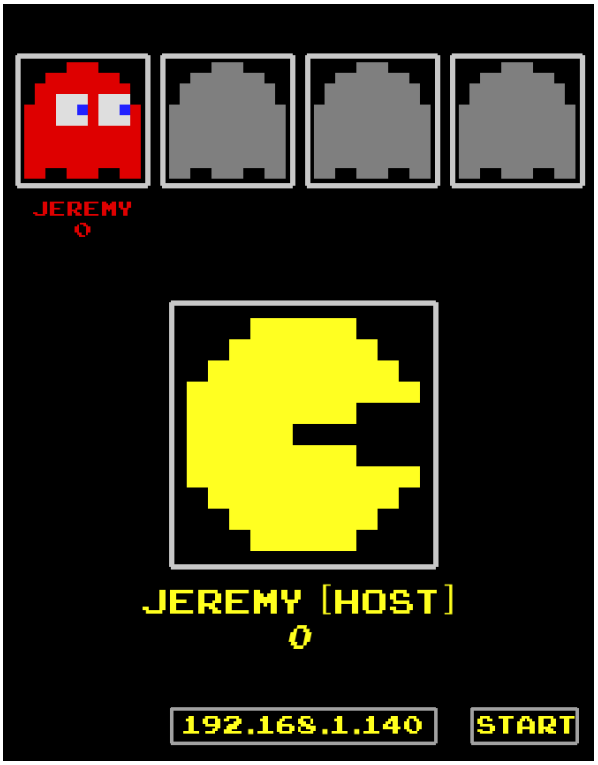
4 equally sized and space boxes containing greyed out ghost sprites. They will stay greyed out until someone else joins the lobby.



Start button that reacts to user's mouse hovering to further convey that it is a clickable button.

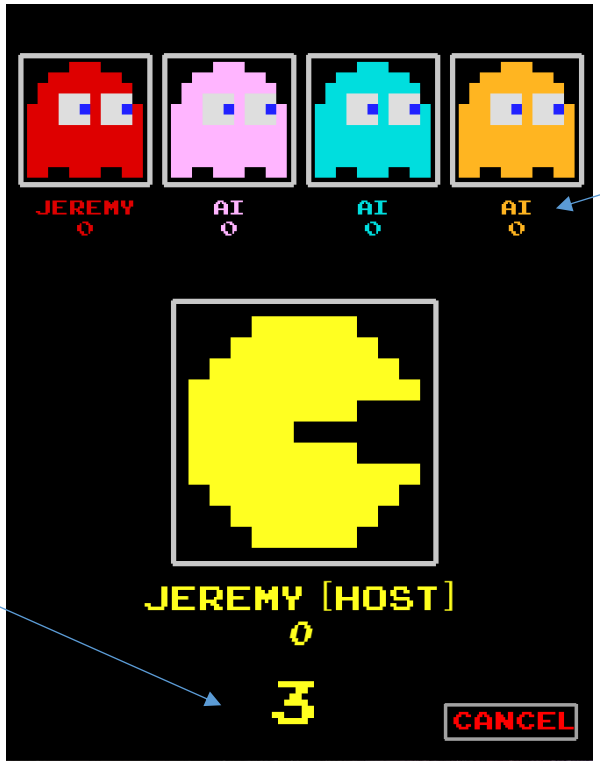
This is the GameID box. There is very little information to indicate what it is, but this is deliberately to keep the UI as clean as possible. The box itself is centralised and in line with the start button. Furthermore, the colours are identical to the other buttons, but you cannot interact with it. This is to imply it is something to be used and not changed.

When a user joins, the only indicator is that their name appears here, and an avatar appears in the box. This change is to keep the lobby clean while still showing the user exactly what is happening.



When the host starts the game, the other slots are filled up by AI and (to indicate extra players are in the game) the coloured sprites are revealed.

A countdown of 5 seconds is started which replaces the GameID box. This implies no other users can join as the GameID used to join is taken away- which is true at this point.

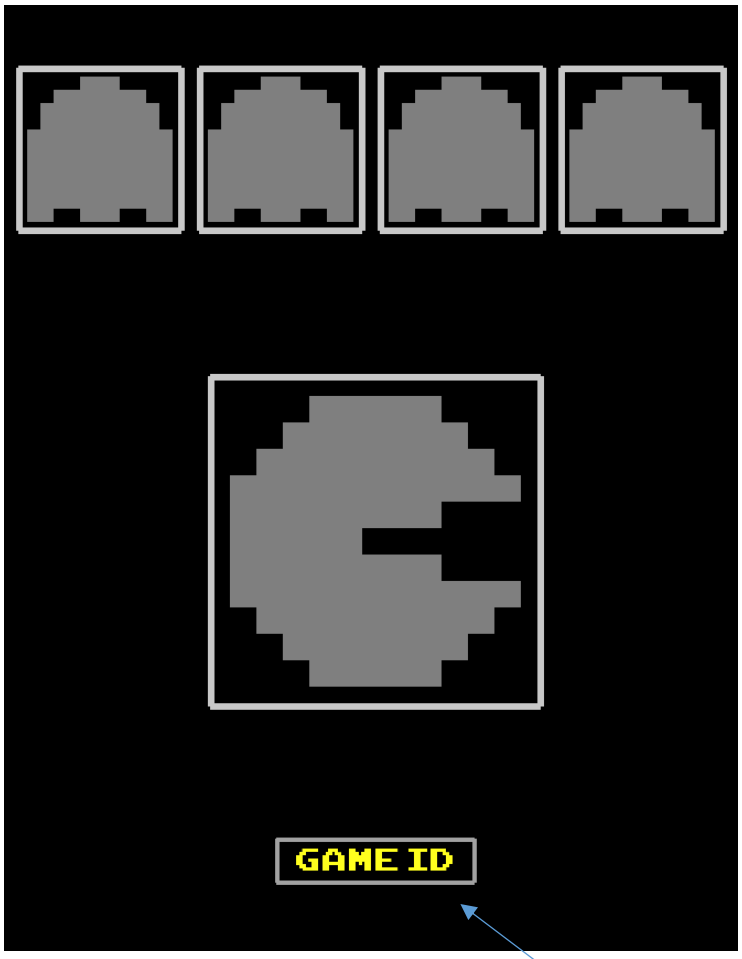


When the host starts a game, the start button changes into a cancel button with a red colour instead of yellow.

MULTIPLAYER – CLIENT

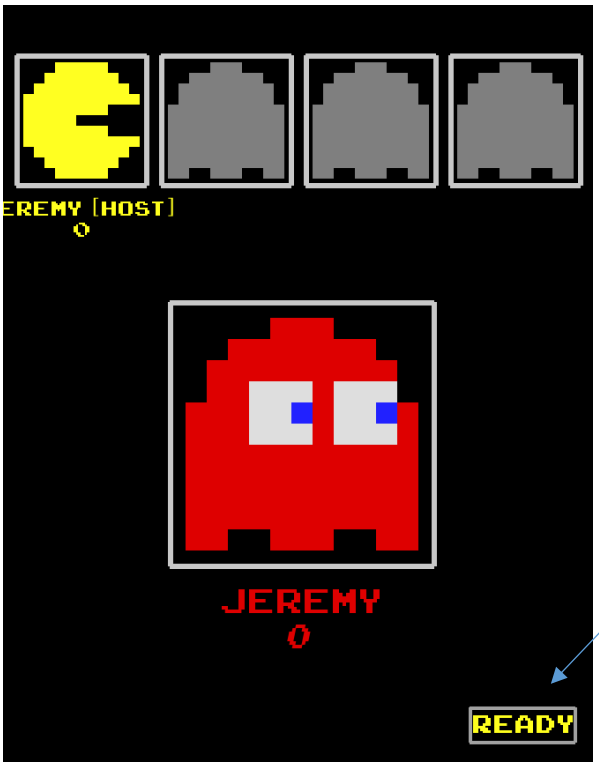
If the user instead selects 'Join Game' they will be taken to the screen below. This screen is very bare as there is not much you can do without a lobby. Once the user joins a lobby using the host's GameID, they will be taken to the other screens showed below. At this point, any other users in the lobby (including the host) will be displayed along the top with the user's assigned avatar then appearing in the central box. Here, they can ready up and view the countdown when the host begins it.

4 equally sized and space boxes containing greyed out ghost sprites. They will stay greyed out until someone else joins the lobby.



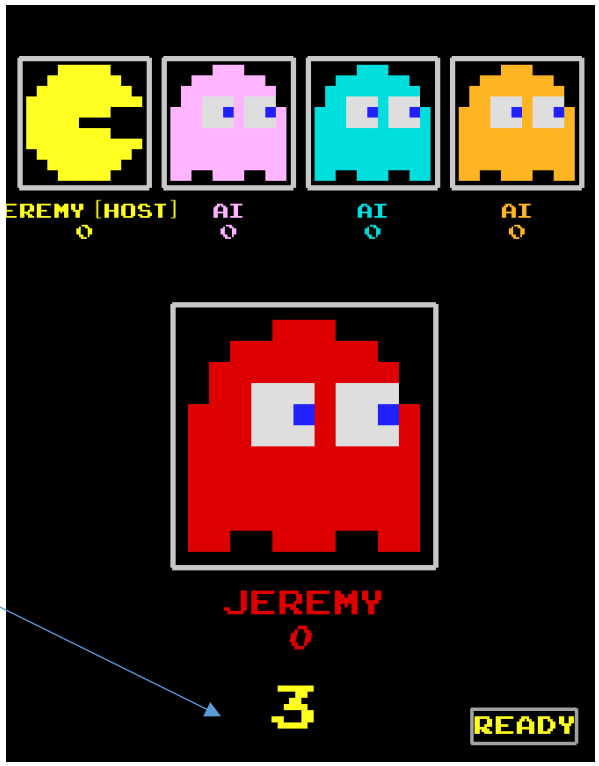
The Game ID box is the only input box on the whole page. This is done to bring attention to needing to join a lobby. There is also no central avatar on this screen as the user is given one when joining a lobby. This will also be assigned by the host so we cannot guess what the avatar will be at this point.

The host avatar and name (along with a badge to show that it is the host) will appear in the first box on the top row.



Here, there is a ready button. When clicked, the word 'Ready' appears under the user's avatar on everyone's machine.

When the host starts the game, the other slots are filled up by AI and (to indicate extra players are in the game) the coloured sprites are revealed.



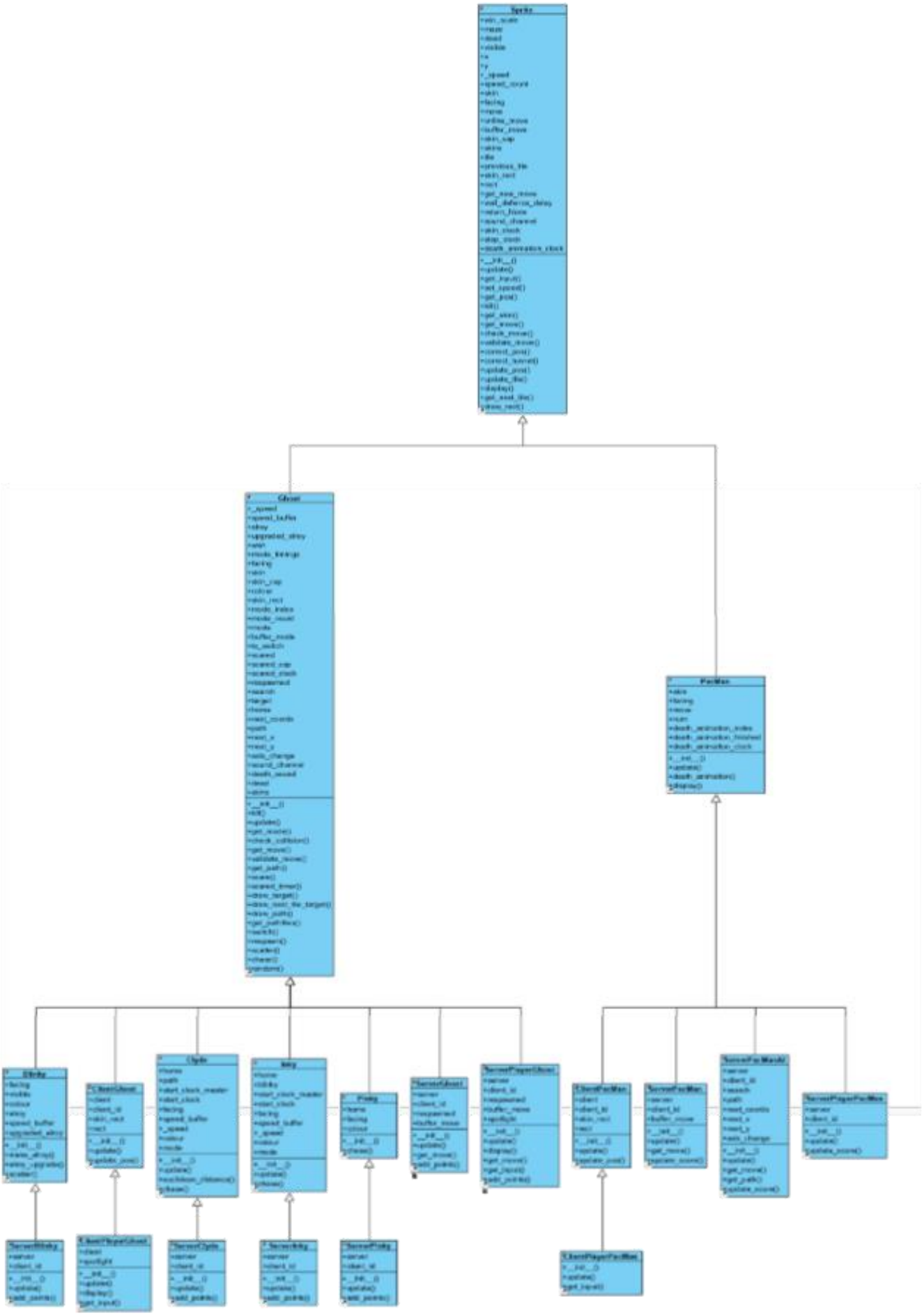
A countdown of 5 seconds is started which replaces the GameID box. This implies no other users can join as the box is taken away, which is true at this point.

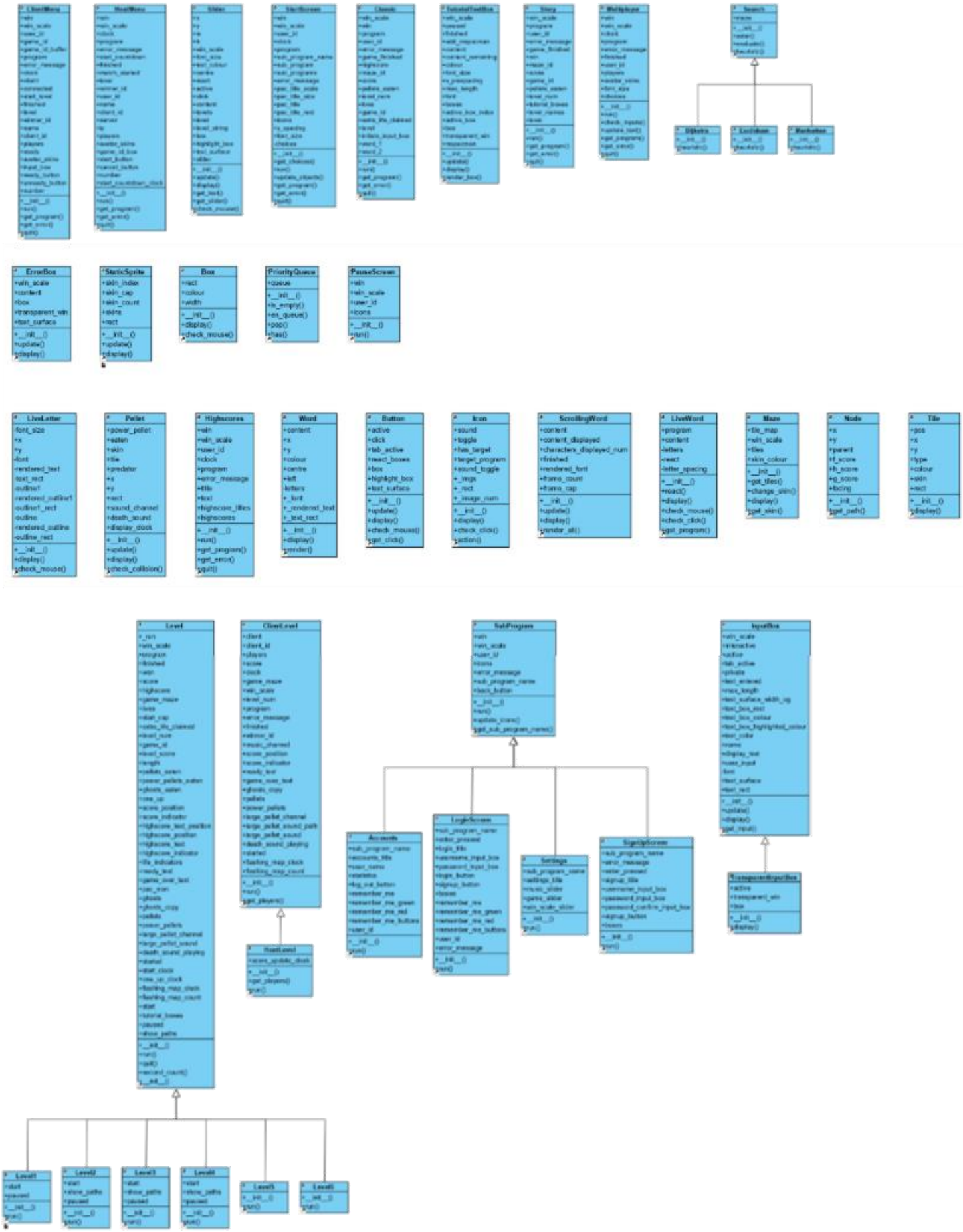
CLASS DESCRIPTIONS

Below are the class diagrams for my project which show my use of aggregation and inheritance in order to increase efficiency and readability. Also shown is a table depicting what each method and class is responsible for in higher detail than the high-level descriptions.

CLASS DIAGRAM

Below are class diagrams for the most advanced uses of inheritance in my project.





CLASS DESCRIPTION TABLE

“” - Unchanged signature from other definition of a method with the same name.

Class	Name	Parameters	Return	Description
StartScreen	__init__	“”	none	Displays the StartScreen and keeps track of all the objects on the screen and user interactions with them.
	get_choices	string_choices, win_scale	choices	Returns a list of live word objects. One for each choice in the string choices parameter.
	run	win, events	none	Controls all the essential tasks, such as displaying all objects, and deciding if and what subprogram should be run.
	update_objects	events	none	Updates objects according to mouse input.
	get_program	none	program	Gets program attribute.
	get_error	none	error	Gets error attribute.
	quit	none	none	Stops all threads in the object.
SubProgram	__init__	“” icons_list	none	A program within the menu screen. This parent class contains the methods that run / update each sub menu as all the subprograms operate in a similar way.
	run	win, events	none	Controls all the essential tasks, such as displaying all objects.
	update_icons	events	none	Updates icons according to mouse input.
	get_sub_ program_name	none	sub program name	Fetches sub program name.
SignUpScreen	__init__	“” icons	none	Inherits methods from SubProgram.
	run	“”	“”	Overridden from the parent class to include extra objects unique to the program.
LoginScreen	__init__	“” icons	none	Class for the login screen.
	run	win, events	none	Overridden from the parent class to include extra objects unique to the program.
Settings	__init__	“” icons	none	Class for the settings menu.
	run	“”	“”	Overridden from the parent class to include extra objects unique to the program.

Accounts	__init__	“” icons	none	Class for accounts menu.
	run	“”	“”	Overridden from the parent class to include extra objects unique to the program.
HighScores	__init__	“”	none	Displays all the high scores that are saved in the database.
	run	win, events	none	Allows the displaying of all objects on the screen.
	get_program	none	program	Gets program attribute.
	get_error	none	error	Gets the error attribute.
	quit	“”	“”	Quit has no function as there are no threads here, however it is still required that all objects run from the main loop have the same quit class.
Story	__init__	“”	none	Controls the running of each level and database queries.
	run	“”	none	This method is run directly from the main script and controls the running of each new level.
	get_program	“”	“”	“”
	get_error	“”	“”	“”
	quit	“”	“”	Quits all threads, it quits the level and the music.
Level	__init__	win_scale, game_id, level_num. maze_id, pellets, power_pellets, score, tutorial_boxes	none	Responsible for running each level by calling sprite objects and handling their updates. The class also controls other things, such as score, displaying maze etc.
	run	“”	“”	This method is run from the Tutorial class and updates and displays by one frame.
	quit	“”	“”	Quits the level. Stops music playing and saves that level to the database.
	second_count	none	none	Ran on a separate thread, counts seconds, to be saved in the database at the end of each level.
Level1	__init__	“”	none	In level one the user is introduced to the movement mechanics (using arrow keys). They are also introduced to first and least threatening ghost: Clyde. They are shown the behaviours that all

				ghosts have and the behaviour of Clyde. This is shown through the paths that are displayed.
Level2	__init__	“”	none	This level will consist of Pinky.
Level3	__init__	“”	none	This level will introduce Blinky.
Level4	__init__	“”	none	I believe level 5 will be a good point to introduce power pellets.
Level5	__init__	“”	none	Level 6 will introduce inky and Blinky (as inky requires Blinky to function).
Level6	__init__	“”	none	This will be the final tutorial-like level that will feature all of the ghosts and allow the player to use all of the knowledge they have learnt/been given on the previous levels.
Classic	__init__	“”	“”	Controls the running of each level and database queries.
	run	“”	“”	This method keeps track of scores and whether a level has been won or lost.
	get_program	“”	“”	“”
	get_error	“”	“”	“”
	quit	“”	“”	This quit method stops the music and quits the current level.
Level	__init__	win_scale, game_id, level_num, maze_id, pellets, power_pellets, lives, score, high score, start_cap, extra_life_ claimed	“”	Responsible for storing information about sprite objects. The class stores other things, such as score, maze etc.
	run	“”	“”	Responsible for running each level by calling sprite objects and handling their updates. The class also controls other things, such as score, displaying maze etc.
	quit	“”	“”	Saves current level to the database, sets run attribute to false and stops the music.
	second_count	“”	“”	“”
Multiplayer	__init__	“”	“”	Menu for multiplayer (contains a method for creating avatars that

				is used in sub-menus). It controls the multiplayer menu screen. It simply prompts the user to choose either 'create game' or 'join game'. They will be taken to the appropriate pages.
	run	“”	“”	Updates the two choices based on mouse input and displays all objects.
	check_inputs	events	none	Checks whether any of the choices have been clicked.
	update_text	none	none	Updates the choices (highlights when the mouse passes over).
	get_program	“”	“”	“”
	get_error	“”	“”	“”
	get_mouse_ input	events	mouse pos	Fetches mouse position at the time of call.
	quit	none	none	This quit does nothing as there are no threads in the Multiplayer object.
HostMenu	__init__	“”	“”	Host menu screen. Pac-Man will be coloured in and the hosts local IP address will be displayed in the bottom. This can then be used by other players to connect to the host.
	run	“”	“”	This method is run directly from the main script. It updates and displays all of the components on the screen. It also calls the server object to ensure all the data is up to date.
	get_program	“”	“”	“”
	get_error	“”	“”	“”
	quit	“”	“”	Quits the server and any sounds from playing.
Client Menu	__init__	“”	“”	This is very similar to the host menu in that there will be colour avatars when players join the lobby, but there will be a ghost in the centre instead of Pac-Man. They will also have the option to ready up instead of starting the game.
	get_program	“”	“”	“”
	get_error	“”	“”	“”
	quit	“”	“”	Quits the server and any sounds from playing.

ClientLevel	__init__	win_scale, level_num, game_maze, score, client	none	Responsible for running each level by calling sprite objects and handling their updates. The class also controls other things, such as score, displaying maze etc.
	run	“”	“”	This method is run from the Client menu class (as it needs to be able to transfer data form level object to level object which can't be done when running directly from main. It controls the updates and displaying of all objects.
	get_players	players, game_maze, win_scale, client	pac_man, ghosts	This takes the list of players and assigns each of them the appropriate multiplayer sprite based on whether they are Pac-Man or a ghost and based on whether they are a client or server.
HostLevel	__init__	win_scale, level_num, game_maze, score, server	none	Responsible for running each level by calling sprite objects and handling their updates. The class also controls other things, such as score, displaying maze etc.
	get_players	players, game_maze, win_scale, server	pac_man, ghosts	This takes the list of players and assigns each of them the appropriate multiplayer sprite based on whether they are Pac-Man or a ghost.
	run	“”	“”	Does the same as client level except it also keeps track of each players score.
Sprite	__init__	rescource_pack, position, maze, win_scale	none	Template for sub-classes: 'Pac-Man' and 'Ghost'.
	update	move	none	Contains all the calls needed to update any sprite once called (60 times a second).
	get_input	events	move	Default input method for the object using keyboard events (arrow keys).
	set_speed	speed	none	Sets private attribute speed. This is private as it is very important to keep that it is not incorrect.
	get_pos	none	x, y	Returns position of the sprite.
	kill	none	none	Kills the sprite by setting the dead attribute to True.

	get_skin	move	move, string number	Returns the skin reference (direction and number) based on how and if the sprite is moving. These correspond to image files.
	get_move	events	move	Gets move from input then checks to see if that move is valid.
	check_move	move	move	Performs checks on the move argument and returns a valid move.
	validate_move	move	move	Checks specifically whether the move will cause the player to collide with a wall or whether they are colliding.
	correct_pos	none	none	If the sprite is colliding with a wall it will work out how far the sprites (x,y) co-ords differ from the tile is currently on and gradually bring them closer together. This keeps the sprites centred and prevents sprites from clipping through walls.
	correct_tunnel	none	none	Allows players to go through the tunnels by changing their x coordinate when they go off the screen.
	update_pos	move	none	Updates sprite's current position, according to the move, current speed and win_scale.
	update_tile	none	none	Updates what tile the sprite is on. This is used by many methods to determine whether the sprite is going to collide with walls in the future.
	display	win	none	Displays the sprite with the skin that corresponds with the direction and also the skin_number (skin attribute) which is either a 0 or a 1.
	get_next_tile	move	tile	This is used by the move validating methods by returning the next tile the sprite will collide with if it out the move passed through.
PacMan	__init__	“”	“”	Contains all of the extra information specific to Pac-Man (not shared with ghosts).
	update	“”	“”	Run once a frame, this is the method that controls everything to do with Pac-Man.

	death_animation	none	none	Cycles through a series of death animation skins and plays the death animation sound.
	display	“”	“”	This display is slightly different as it either displays normally if Pac-Man is alive or displays a death skin if he is dead.
Ghost	__init__	“” target, level	“”	Contains all of the extra information specific to Pac-Man (not shared with Pac-Man). Target is sprite the ghost will target and level is the current level number which decides the difficulty of the ghost.
	kill	“”	“”	When a ghost is killed this is run.
	update	“”	“”	Run once a frame, this is the method that controls everything to do with the Ghost.
	get_mode	none	mode	Determines which mode should be used to get the Ghosts next target co-ords.
	check_collision	none	none	Checks whether the ghost is colliding with the target (Pac-Man).
	get_move	events	move	Uses the current mode to get the next co-ords. Works out the next move based on the target co-ords.
	validate_move	“”	“”	Checks to see if a move is valid by checking whether it is opposite the current direction (facing). And by the Sprite's validate_move above. This is not needed in the classic mode as the pathfinding algorithm does not produce paths that require a 180-degree change in direction, however, this is needed for online.
	get_path	mode	path	Uses the mode to get a path. This middle man is needed in case the target is unreachable (in which case the path is None and instead the chase mode is used (which is always reachable).
	scare	none	none	Sets the ghost into scared mode when called (which is when Pac-Man collides with a power pellet).

	scared_timer	none	none	Keeps track of how long the ghost is scared and adjust attributes accordingly. It's run on a thread.
	draw_path	win	none	Used for the story mode to teach the users how the AI works. Simply draws the path that the AI will take at any given moment.
	get_pathtiles	path, pathtiles	pathtiles	Simply returns the PyGame rectangle responsible for displaying the path. This is in a separate method so it can be run recursively. It is static but I have included in the object, so it is easy to see how it is being called.
	switch	none	move	Changes the direction of the ghost.
	respawn	none	path	Pathfinding mode: It targets inside the centre, then targets outside once it has reached it.
	scatter	none	path	Pathfinding mode: It targets the specific ghost's home tile.
	chase	none	path	Pathfinding mode: Unique to ghosts: default (Blinky) targets Pac-Man's current tile.
	random	none	path	Pathfinding mode: Targets random row and random tile on that row as long as it's a pellet.
Blinky	__init__	""	""	Class for Blinky (contains home tile, starting position and can become Elroy).
	make_elroy	none	none	Turns Blinky into Elroy (faster).
	elroy_upgrade	none	none	Turns Blinky into upgraded Elroy (faster, and still targets Pac-Man in scatter mode).
	scatter	none	path	Pathfinding mode: It targets the Blinky's home tile unless Blinky is in Elroy mode, in which case this will function in the same way as the chase mode.
Pinky	__init__	""	""	Class for Pinky (contains home tile, starting position).
	chase	""	""	Pathfinding mode: Uses the tile 4 spaces ahead of Pac-Man to get the path. This decreases by one until the target reaches Pac-Man if the tiles in front are not reachable.
Clyde	__init__	""	""	Class for Clyde (contains home tile, starting position, start clock

				(Clyde doesn't leave centre straight away).
	update	“”	“”	Run once a frame, this is the method that controls the start (when Clyde is still inside the centre).
	euclidean_distance	target	distance	Needs this to work out how far from Pac-Man Clyde is. (Used in chase method).
	chase	“”	“”	Pathfinding mode: Targets Pac-Man's tile until the distance to him is less than 8 tiles, when Clyde, instead, targets his home corner.
Inky	__init__	“” Blinky	“”	Class for Inky (contains home tile, starting position).
	update	none	none	Pathfinding mode: Takes the vector between Blinky and Pac-Man and doubles it. Adds this vector to Blinky's position and target that tile.
	chase	“”	“”	Pathfinding mode: Takes the vector between Blinky and Pac-Man and doubles it. Adds this vector to Blinky's position and target that tile.
Pellet	__init__	skin, tile, predator, win_scale, death_sound, sound_channel, power_pellet	none	Class for every pellet in the game.
	update	none	none	Runs every frame, just checks whether the pellets are colliding.
	display	win	none	Displays the pellet using the skin. If it's a power pellet it will flash.
	check_collision	none	none	If pellet's and Pac-Man's rectangles are colliding, kill pellet and play death sound.
Search	__init__	maze	none	Search is an object so that we can save the maze.
	astar	start, end, facing	path	Mainloop of the search algorithm.
	evaluate	child, end	none	Assigns each child an h score and a g score, then combines these for the f score. These determine the fitness of the child, by

				considering how many nodes there have been before the child and how close the child is to the end node. This score is then used to choose the next child to expand.
	heuristic	node, end	score	Gives a score based on how close the tile is to the end child. In this case, it returns 0, as the default heuristic is Dijkstra's which checks every path.
Manhattan	__init__	maze	none	Stores maze object.
	heuristic	“”	“”	Gives a score based on how close the tile is to the end child. In this case, it returns the Manhattan distance between them.
Euclidean	__init__	maze	none	Stores maze object.
	heuristic	“”	“”	Gives a score based on how close the tile is to the end child. In this case, it returns the Euclidean distance between them.
Priority Queue	__init__	none	none	Stores objects that are popped according to priority or if priories are equal, by first in first out.
	is_empty	none	boolean	Returns whether the queue is empty or not.
	en_queue	node	none	Adds node to the queue.
	pop	none	node	Returns node with the lowest f_score.
	has	child	boolean	Returns True if the given child is already in the queue else False.
Node	__init__	facing, parent	none	Every tile in a path (or possible path) is called a node. It has tilex and tiley values and also stores the node object of the tile that came before it in the path.
	get_path	path	path	Recursive algorithm to get the path once the target node has been reached.
Tile	__init__	tile_x, tile_y, _type, win_scale, skin	none	Class for a tile, containing the type of tile, position, skin and how to blit it.
	display	win	none	Displays tile.
Maze	__init__	maze_id, win_scale	none	Class for the whole maze which contains details about all the tiles.

	get_tiles	skin_colour	tiles	Gets a list of tile objects based on the tile_map.
	change_skin	none	none	Changes the colour of all tiles in the 2D list.
	display	win	none	Displays the maze, by displaying each individual tile.
	get_skin	tile_x, tile_y, tile_map	skin	Works out what skin each tile should have based on the type of tile in the surrounding 8 spaces.
ClientPacMan	__init__	resource_pack, maze, win_scale, client, client_id	none	Client-side Pac-Man sprite that takes input from another client or possibly the server (not from keyboard).
	update	events	none	Gets input from Client object.
	update_pos	pos	none	Changes the sprite's position and updates tile passed on this.
ClientPlayer PacMan	__init__	“”	“”	Client-side Pac-Man sprite that is controlled by keyboard inputs.
	update	“”	“”	Sends client-move to the server.
	get_input	events	move	Gets inputs from keyboard events.
ClientGhost	__init__	“” position, target, level	none	Client-side ghost sprite that takes input from another client or possibly the server (not from keyboard). The position is the place where the ghost should start.
	update	“”	“”	Gets input from Client object.
	update_pos	“”	“”	“”
ClientPlayer Ghost	__init__	“” position, target, level	“”	Client-side Ghost sprite that is controlled by keyboard inputs.
	update	“”	“”	Sends client-move to the server.
	display	win	none	Blits sprite skin to the window, and also a spotlight (after the maze, pellets and Pac-Man), but before the other ghosts and score.
	get_input	“”	“”	“”
ServerPacMan	__init__	resource_pack, maze, win_scale, client, client_id	“”	Server-side Pac-Man sprite controlled by another client.

	update	“”	“”	Gets input from the Server object, as the get_move method has been overridden below.
	get_move	“”	“”	Gets move from server.
	update_score	score	none	Updates server with the current score for the sprite.
ServerGhost	__init__	“” position, target, level	none	The server-side ghost that is controlled by clients.
	update	“”	“”	Validates client input (from server object) and sends back a valid move.
	get_move	“”	“”	Gets move from the server object and returns a validated move.
	add_points	points	none	Updates points for that particular player based on interactions the sprite has server-side.
ServerPlayer PacMan	__init__	“”	“”	Server-side Pac-Man sprite controlled by keyboard inputs.
	update	“”	“”	Sends move and pos to the server object.
	update_score	“”	“”	“”
ServerPacMan AI	__init__	“”	“”	Server-side Pac-Man AI. Makes random moves and sends to the server object.
	update	“”	“”	Updates server with the move.
	get_move	“”	“”	Gets move by working out the move needed to reach the next tile in the path that is worked out by randomly selecting a tile beforehand.
	get_path	none	path	Gets the path to random pellet tile.
	update_score	“”	“”	“”
ServerPlayer Ghost	__init__	“” position, target, level	“”	Server-side ghost sprite controlled by keyboard inputs.
	update	“”	“”	Gets move from keyboard inputs and then sends to the server object.
	display	“”	“”	“”
	get_move	“”	“”	Gets move using keyboard events.
	get_input	“”	“”	“”

	add_points	points	none	Adds points (earned from being close to Pac-Man) to score and updates server.
ServerBlinky	__init__	“” position, target, level	“”	Class for Blinky (contains home tile, starting position and can become Elroy).
	update	“”	“”	Takes next move from path and updates server with it.
	add_points	“”	“”	“”
ServerPinky	__init__	“”	“”	Class for Pinky (contains home tile, starting position).
	update	“”	“”	“”
	add_points	“”	“”	“”
ServerInky	__init__	“”	“”	Class for Inky (contains home tile, starting position).
	update	“”	“”	“”
	add_points	“”	“”	“”
ServerClyde	__init__	“”	“”	Class for Clyde (contains home tile, starting position, start clock (Clyde doesn't leave centre straight away).
	update	“”	“”	“”
	add_points	“”	“”	“”
Connection	__init__	user_ip, conn, user_id players	“”	Class for each connection the server has with a client. Controls all information going from server to client.
	update	none	none	Runs once every frame, updates each item in player_data dictionary. Only used at the beginning of the connection to share basic information between the client and the host.
	receive	none	data	Receives data from client (player_data).
	send	data	none	Converts data into bytes and sends to the client. This is usually the 2D dictionary of player info.
	get_player_data	none	data	Gets player_data.
	get_id	none	id	Gets ID.
	close	none	none	Closes the connection so the server can be quit safely.
Server	__init__	name	none	Class for the server that controls the sending and receiving of game data for each player between the host and all clients connected.

	connect	none	none	Before the game starts this method will listen for new connections, and then create a connection when one is received.
	receive	none	none	This method updates the player data based on information sent to each client to each of the connections. Updates twice a second to account for any syncing issues between clients and server.
	update_data	client_id, key, value	none	Edits the data in the player data dictionary with that data passed in.
	send_data	none	none	Sends the player data to all connected players.
	check_connections	none	none	Check to see whether the players are still connected. If not, their data is wiped from the dictionary.
	get_data	client_id	data	Gets data for a specific player from the player data.
	get_players	none	data	Gets all player data.
	get_client_id	none	client_id	Gets client id.
	swap	client_id	none	Whichever ClientID is passed into this method will become the next Pac-Man. This happens when a ghost catches Pac-Man and is needed to swap the skins and start position of players around when they become Pac-Man.
	reset	none	none	At the end of each round the positions of each sprite need to be reset according to the player template in order for the sprites to spawn in the correct place in the following round.
	add_ai	none	none	Changes name of all available ghosts into 'AI', which will mean they become controlled by AI in the game. This method is run when the game countdown is started.
	remove_ai	none	none	This changes all AI ghosts back to None when the game countdown is stopped (to allow for more players to join).
	quit	none	none	Correctly adjusts attributes so that all threads and connections terminate when the server is no longer needed.

Client	__init__	host_ip name	none	Runs on clients when the user selects join game and enters a GameID.
	send	data	none	Sends data to the Server.
	receive	none	data	Listens for player data from the server.
	update	none	none	Receives data from the server and sets equal to players.
	update_data	“”	“”	“”
	get_data	“”	“”	“”
	send_player_data	none	none	Sends player data using the send method.
	get_players	none	player data	Corrects integer keys in players (as they are saved as strings when converted to and from bytes) and returns.
	get_client_id	“”	“”	“”
	end	none	none	Closes connection.
Word	__init__	content, pos, colour, font_size, win_scale, italic, bold, centre, left	none	The class used to put a word on the screen.
	display	win	none	Blits the rendered font to the screen as per the rect.
	render	none	none	Renders font. Takes the content and colour and converts this into a font object. Then a rect object is created based on the position and alignment instructions.
LiveWord	__init__	content, y, font_size, win_scale, highlight_colour	none	Live words have the ability to highlight when selected by the user. Separate to word-class as live words must be rendered letter by letter so that when the highlighted layer is shown, it correctly covers each individual letter.
	react	none	none	Sets private attribute react to True.
	display	win	none	Displays each letter in the word, sets react to False so that if the mouse stops colliding with the word, then the word will stop displaying as highlighted.

	check_mouse	x, y	boolean	Returns true if the x, y coordinates supplied are colliding with any of the letters' rects.
	check_click	x, y	boolean	Checks each letter to see if any of them have been clicked by the mouse.
LiveLetter	__init__	letter, x, y, font_size, win_scale, highlight_colour	none	Live letters are used by the live words class. They control one letter each.
	display	react, win	none	Blits the letter object to the screen. If react is True, then a black outline and another outline will be blitted behind to highlight the letter.
	check_mouse	x, y	boolean	Checks whether the x, y co-ords are colliding with the letter.
Scrolling Word	__init__	content, pos, colour, font_size, win_scale, frame_cap	none	Scrolling words uses the word class (constantly re-renders it) to give the appearance of scrolling / revealing text.
	update	events	none	Updates the word by changing the number of words displayed in each frame as per the speed assigned in the init.
	display	“”	“”	Displays all text.
	render_all	none	none	Render all text.
TutorialText Box	__init__	content, colour, win_scale, add_mspacman	none	Uses the scrolling word, and box class to display scrolling text that can go over onto many lines all in a neat and tidy box.
	update	“”	“”	Controls the various scrolling words, boxes and MsPacMan sprite.
	display	“”	“”	Displays box, MsPacMan and text.
	render_box	none	none	Renders text in the box.
Box	__init__	pos, dimensions, colour, width, win_scale, centre	none	Boxes are simply rectangles. They are capable of changing colour when a mouse is colliding with it however, this requires external support form code at the moment. Boxes are often aggregated. For example. they are used in input boxes etc.
	display	“”	“”	Blits the box to the screen.

	check_mouse	“”	“”	“”
InputBox	__init__	x, y, w, h, font_size, win_scale, name, interactive, centre, private, max_length	none	Input boxes use a box object and a word object (or will do). They are used to allow the user to allow the input of words from the user.
	update	“”	“”	Updates the textbox. This includes changing the size of the box if it gets too small for the text and handling events.
	display	“”	“”	Blits text and box to the screen.
	get_input	none	none	Returns user input attribute.
ErrorBox	__init__	content, win_scale	none	Error boxes are used when there is an error that affects the game in a way the user would not expect.
	update	“”	“”	The only check that the error box performs is to see if a mouse button has been lifted up or a key has been pressed down. This is because when the user presses something it will disappear.
	display	“”	“”	Blits the box and text to the surface.
Button	__init__	content, pos, dimensions, font_size, text_colour, width, win_scale, centre	none	Buttons are used to allow the user to select certain outcomes or events. Buttons use the word class and the box class.
	update	“”	“”	Checks whether the mouse is 'colliding' with the button's box and if the mouse has been clicked. It becomes active if it has been clicked (meaning the colour of the box will change). The program using the button must implement the button's function and can use the click attribute to decide when to execute the desired outcome.

	display	“”	“”	Blits the box, and text to the win surface.
	check_mouse	x, y	boolean	Returns True if the (x, y) co-ord is within the box's rect.
	get_click	none	boolean	Returns click attribute.
Slider	__init__	content, x, y, w, h, font_size, text_colour, box_width, win_scale, centre, level, levels	none	Uses a box, a word and a transparent box to create a slider. It can be used by the user to select a level for something. The range is from 0 to the level’s parameter. The default is 100 and this is mainly used for sound sliders, but the one for win scale, for example, only goes up to 5.
	update	“”	“”	Checks where on the slider the mouse has been clicked and adjusts the slider and word to match this value.
	display	“”	“”	Blits the transparent slider (PyGame surface object), the text, and the box to the screen. If react is True, then the highlighted (white) box will be blitted instead.
	get_text	none	text_surface	Instantiates a text object. This must be run every time the level is changed in order to update the number.
	get_slider	none	slider	Gets the transparent surface that will become the sliding element. This must be run every time the level changes in order to display that change graphically as well.
	check_mouse	x, y	boolean	Returns True is the mouse is colliding with the box else False.
Transparent InputBox	__init__	w, h, font_size, win_scale, name	none	This is a like a normal input box however there is a transparent surface behind it so that it can be used over the top of something other than a black background.
	display	“”	“”	“”
Icon	__init__	pos, imgs, win_scale, sound, toggle, target_program	none	Class for creating icons.

	display	“”	“”	Blits the icon to the screen using the position and imgs.
	check_click	“”	“”	“”
	action	none	none	This usually runs when the check_click function returns True and completes the icons desired action.

FUNCTION DESCRIPTION TABLE

Script	Name	Parameters	Return	Description
Main	create_window	win_scale	win	Takes the window_scale and returns a surface based on this and the screen ration of 7:9. It also sets the necessary icons and string for the window.
Splash Screens	get_mouse_input	events	mouse pos	Fetches mouse position at the time of call.
Multiplayer	get_avatar_skins	none	avatar_skins	Gets skins for each avatar from the resources folder. Chooses between grey and coloured depending on whether the player is connected.
	get_boxes	win_scale	boxes, large_box	Returns box objects for the menu screens. These are just the 4 empty boxes across the top of the screen and the large central box that are displayed before there are any sprites in them.
	get_avatars	players, finished, avatar_skins, boxes, large_box, win_scale, client_id	avatars, names, scores, ready_indicators, places	Returns all the graphical attributes of each player (also known as the avatars). It will add sprites, names, the current score, ready indicators and places (when the game has finished) into lists that can then be stored in the menu classes.
	get_distance_points	ghost, pac_man	points	Works out the Euclidean distance between the ghost and Pac-Man objects and returns a number of points to give the based on how far it is.

	get_mouse_input	events	mouse pos	Fetches mouse position at the time of call.
Pathfinding	get_children	node, maze	children	Returns next available tiles that haven't yet been evaluated from the current tile. This can then be added to the list of children.
	in_closed	child, closed	boolean	Checks if a child is already in the closed set (already been evaluated).
Database	create_users	cursor	none	Creates table 'Users'.
	create_game_history	cursor	none	Creates table 'GameHistory'.
	create_game_level	cursor	none	Creates table 'GameLevel'.
	create_multiplayer_ game_history	cursor	none	Creates the table 'MultiplayerGameHistory'.
	create_multiplayer_ player_history	cursor	none	Creates the table 'MultiplayerPlayerHistory'.
	create_mazes	cursor	none	Creates table 'Mazes'.
	create_db	cursor	none	Checks whether the database has already been created and if not creates it by running all of the create table functions.
	query	sql, data	results	Used to execute SQL statements, can also return data.
	get_game_id	user_id, maze_id	game_id	Creates a game history entry using the arguments and date, time. Returns the GameID just created.
	get_highscore	none	high_score	Returns the current high score! In one line! One complex query!
	save_level	level_num, game_id, lives, score, length, pellets_eaten, power_pellets_ eaten, ghosts_eaten	none	Saves Individual-level data to GameLevel.
	save_initials	game_id, initials	none	Saves initials to game history after the game has finished.
	save_maze	user_id, maze	none	After a user has created a maze it can be saved to

				their account using this function.
	save_user	username, password	none	When a user has completed the sign up from their information is used to create a user here.
	login	username, password	user_id	Checks user-provided details against the database.
	check_sign_up	user_name, password, password_confirm	boolean (successful login), error_message	Performs the following checks in order: Username length, username availability, password length, password strength, password matches.
	get_username	user_id	username	Returns username from database, given the user_id.
	get_statistics	user_id	statistics	Returns statistics in a list from the database corresponding to the given user_id.
	get_highscore	none	high_score	Returns highest score a user who inputted their initials has ever achieved.
	get_date	none	date	Returns the date in the format: dd/mm/yyyy.
	get_time	none	time	Returns the time in the format: hh:mm:ss.
Settings	write_settings	none	none	If there is not a settings file, one will be created.
	get_setting	setting	none	Reads the value from the chosen setting. If there is no settings file, one will be created.
	save_setting	setting, value	none	Saves a given value to a given setting.

KEY ALGORITHMS

Below are more detailed versions of the key algorithms in my project. Whilst these are not final, they will likely not change much in the technical solution as testing has shown they accomplish what is required from the objectives.

A*

As touched on in my analysis, the pathfinding algorithm I will be using is A*. However, to implement this within the sprites that I had already made as a prototype, and to accommodate some important gameplay mechanics, I had to adapt my early concept into the final algorithm. In this section, I will overview the challenges I faced in coming up with the final concept.

The first and most important problem I faced was preventing the algorithm from finding a path for the ghosts that involved performing a 180° turn. The ghosts cannot change direction unless they are changing modes, so it was imperative that I ignored paths that would give this outcome in the A* algorithm or I would risk not getting the optimum path. To do this I simply replaced the tile behind the ghost with a wall and removed it once the ghost had moved to another tile.

Choosing the heuristic for the algorithm was also vital. A heuristic is what allows the pathfinding algorithm to ignore paths that would end up taking longer (by estimating the distance of a node to the target node). There are many types of heuristic and the trick is to find the one that takes the minimum time to perform, whilst still allowing the algorithm to return the most efficient path. For this, the heuristic must never overestimate the distance to the target node. A heuristic that does this is known as an admissible heuristic. There are 2 options for a 2D game like mine. Manhattan and Euclidean.

The Euclidean heuristic is what the original Pac-Man game uses. It calculates the distance to the target node using the Pythagorean theorem. This is useful in games in which you can travel diagonally, as travelling diagonally is quicker than travelling right then up or down then left for example. However, in Pac-Man, this movement is impossible and so using the Euclidean distance is irrelevant. Whilst it will never overestimate the distance (as travelling diagonally is shorter) and so is admissible, working out the hypotenuse of a triangle takes a relatively long time (at least in comparison to Manhattan). The Euclidean heuristic works in python as follows:

```
1.  def Euclidean(self, node, end):
2.          x, y = end
3.          return ((node.x - x)**2 + (node.y - y)**2)**0.5
```

The Manhattan heuristic simply adds together the difference between the position of the current and target node vertically and horizontally and adds these together. This is far quicker than the Euclidean distance and is also admissible as it does not overestimate the actual path distance between the two nodes. As this is a more efficient heuristic, and it is still admissible, we can use it instead of the Euclidean distance and make our pathfinding algorithm more efficient than the original game! The Manhattan heuristic works in python as follows:

```
4.  def Manhattan(self, node, end):
5.          x, y = end
6.          return abs(node.x - x) + abs(node.y - y)
```

Once the algorithm has reached the target node, it uses a recursive method as follows:

```
7.  def get_path(self, path):
8.          if self.parent is not None:
9.              path = self.parent.get_path(path)
10.         path.append((self.x, self.y))
11.         return path
```

This method uses the call stack in order to undo the path that the algorithm has chosen, whilst adding each node is visits to a list of nodes (path). This works by getting the node's parent and adding it to the path. Then, (if the parent node has a parent) it adds this to the path and keeps visiting the parents until we reach a node that doesn't have a parent (the start node).

SQL

As discussed, I have a local database within my project that holds information about user logins and most importantly their high scores. Below are examples of a few of my more complex SQL queries.

```
SELECT Sum(score) as SCORE, GameHistory.Initials
FROM GameLevel, GameHistory
WHERE GameLevel.GameID = GameHistory.GameID
AND GameHistory.Initials IS NOT NULL
GROUP BY GameHistory.GameID
ORDER BY SCORE DESC
```

This query will be used to grab a list of all the high scores ordered so that the largest is first. This can then be used on my high scores page to display the top ten high scores.

It works by collecting all of the scores from each individual level whose GameID have initials attached. These are then grouped according to their GameID. Next, the scores for each of the levels in a game are summed. Finally, these sums are sorted in descending order to allow the largest ten to be used on the high score screen.

```
SELECT Max(Scores)
FROM(
        SELECT Sum(Score) as Scores
        FROM GameLevel
        GROUP BY GameID
        )
```

This query will be used at the start of a classic game to pull the high scores from the current version of my relational database. It works by first by grabbing all the scores from every level in the 'GameLevel' table. Then, these scores are grouped by their GameID. After these lists of scores for each level have been grouped by GroupID, the lists are summed to give a total for each game. Finally, the largest score in this final list of total scores is selected to give the current High score.

NETWORKING

My net code is based roughly on the model provided in the analysis of this project. Whilst the basic idea is present in my latest prototype, I have made a few adjustments to solve problems I had only actually encountered in early tests of the net code within my game.

Below is a print out of my latest prototype and an explanation into the technical workings and my thoughts regarding the implementation.

```
1.  import threading
2.  import socket
3.  import json
4.  import copy
5.  import time
6.
7.
8.  class Connection:
9.      def __init__(self, user_ip, conn, user_id, players):
10.         self.connected = True
11.         self.__player_data = {
12.
```

```
            'ready': None,
            'client_move': None,

        }

        self.__connected = False
        self.__IP = user_ip
        self.id = user_id
        self.__PORT = 50007
        self.__conn = conn

        self.send({

                    'client_id': user_id,
                    'players': players

                  })

        self.__player_data['name'] = self.receive()['name']

        threading.Thread(target=self.update).start()

    def update(self):
        while self.connected:
            data = self.receive()
            if data is not None:
                for attribute, value in data.items():
                    self.__player_data[attribute] = value

    def receive(self):
        try:
            data = self.__conn.recv(4096)
            return json.loads(data)
        except Exception as e:
            print("disconnected: {}".format(e))

        if self.connected:
            data = json.dumps(data)
            data = bytes(data, 'utf-8')
            try:
                self.__conn.sendall(data)
            except ConnectionResetError:
                print("disconnected")
                self.connected = False

    def get_player_data(self):
        return self.__player_data

    def get_id(self):
        return self.id

    def close(self):
        self.__conn.close()
        self.connected = False


class Server:
    def __init__(self, name):
        self.__run = True
        self.test_count = 0
        self.__players_template = {

            0:
                {
                    'name':     '{} [host]'.format(name),
                    'skin':     'pac-man',
                    'score':    0,
                    'ready':    None,
```

```
                    'pos':      [167, 318],
                    'move':     None,
                    'countdown': None,
                    'start':    False,
                    'end':      True,
                    'place':    None,
                    'finished': False
                },

            1:
                {
                    'name':     None,
                    'skin':     'blinky',
                    'score':    0,
                    'ready':    None,
                    'pos':      [168, 176],
                    'move':     None,
                    'client_move': None,
                    'place': None
                    },

                2:
                    {
                        'name':     None,
                        'skin':     'pinky',
                        'score':    0,
                        'ready':    None,
                        'pos':      [168, 214],
                        'move':     None,
                        'client_move': None,
                        'place': None
                    },

                3:
                    {
                        'name':     None,
                        'skin':     'inky',
                        'score':    0,
                        'ready':    None,
                        'pos':      [144, 214],
                        'move':     None,
                        'client_move': None,
                        'place': None
                    },
                4:
                    {
                        'name':     None,
                        'skin':     'clyde',
                        'score':    0,
                        'ready':    None,
                        'pos':      [192, 214],
                        'move':     None,
                        'client_move': None,
                        'place': None
                    }
                }

            self.__players = copy.deepcopy(self.__players_template)
            self.__IP = self.get_ip()
            self.__port = 50007
            self.__connections = []
            self.__s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.__s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            self.__s.bind((self.__IP, self.__port))

            self.has_ai = False

            self.searching_for_clients = True
```

```
149.                threading.Thread(target=self.connect).start()
150.                threading.Thread(target=self.receive).start()
151.                threading.Thread(target=self.check_connections).start()
152.
153.            def connect(self):
154.                while self.searching_for_clients and self.__run:
155.                    self.__s.listen(1)
156.                    if self.searching_for_clients:
157.                        try:
158.                            conn, addr = self.__s.accept()
159.                            available_ids = [k for k, v in self.__players.items() if v['name'] is N
    one]
160.                            client_id = available_ids[0]
161.                            connection = Connection(addr[0], conn, client_id, self.__players)
162.                            self.__connections.append(connection)
163.                        except Exception as e:
164.                            print("Connect: {}".format(e))
165.
166.            def receive(self):
167.
168.                while self.__run:
169.                    try:
170.                        for connection in self.__connections:
171.                            data = connection.get_player_data()
172.                            for attribute, value in data.items():
173.                                self.__players[connection.get_id()][attribute] = value
174.                    except Exception as e:
175.                        print(e)
176.                    time.sleep(1/120)
177.
178.            def update_data(self, client_id, key, value):
179.                self.__players[client_id][key] = value
180.
181.            def send_data(self):
182.                for connection in self.__connections:
183.                    connection.send(self.__players)
184.
185.            def check_connections(self):
186.                while self.__run:
187.                    for connection in self.__connections:
188.                        if not connection.connected:
189.                            connection.close()
190.                            self.__players[connection.get_id()] = self.__players_template[connectio
    n.get_id()].copy()
191.                            self.__connections.remove(connection)
192.                    time.sleep(1/120)
193.
194.            def get_data(self, client_id, type):
195.                return self.__players[client_id][type]
196.
197.            def get_players(self):
198.                return self.__players
199.
200.            def get_ip(self):
201.                return socket.gethostbyname(socket.gethostname())
202.
203.            def get_client_id(self):
204.                return 0
205.
206.            def swap(self, client_id):
207.                pac_man_id = [id for id, data in self.__players.items() if data['skin'] == 'pac-
    man'][0]
208.
209.                pac_man_skin = "{}".format(self.__players[pac_man_id]['skin'])
210.                client_skin = "{}".format(self.__players[client_id]['skin'])
211.
212.                self.__players[pac_man_id]['skin'] = client_skin
213.                self.__players[client_id]['skin'] = pac_man_skin
```

```
214.
215.            def reset(self):
216.                for client_id, client_data in self.__players.items():
217.                    og_player_data = [og_client_data
218.                                      for og_client_data in self.__players_template.values()
219.                                      if og_client_data['skin'] == client_data['skin']][0]
220.
221.                    client_data['pos'] = og_player_data['pos'][::]
222.
223.            def add_ai(self):
224.                for client_data in self.__players.values():
225.                    if client_data['name'] is None:
226.                        client_data['name'] = 'AI'
227.                self.has_ai = True
228.
229.            def remove_ai(self):
230.                for client_data in self.__players.values():
231.                    if client_data['name'] == 'AI':
232.                        client_data['name'] = None
233.                self.has_ai = False
234.
235.            def quit(self):
236.                self.searching_for_clients = False
237.                self.__run = False
238.                for connection in self.__connections:
239.                    connection.close()
240.                self.__s.close()
241.
242.
243.        class Client:
244.            def __init__(self, host_ip, name):
245.                self.__host_ip = host_ip
246.                self.__name = name
247.                self.connected = False
248.                self.connection_failed = None
249.
250.                self.__player_data = {
251.
252.                                      'ready':        None,
253.                                      'client_move':  None,
254.
255.                                     }
256.
257.                self.__port = 50007
258.                self.__s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
259.
260.                try:
261.                    self.__s.connect((self.__host_ip, self.__port))
262.                    self.connection_failed = False
263.                except (ConnectionRefusedError, socket.gaierror, TypeError):
264.                    self.connection_failed = True
265.
266.                if not self.connection_failed:
267.                    init_data = self.receive()
268.                    if init_data is not None:
269.                        self.__client_id = init_data['client_id']
270.                        self.__players = init_data['players']
271.                        self.send({'name': self.__name})
272.
273.                        self.connected = True
274.                        threading.Thread(target=self.update).start()
275.
276.            def send(self, data):
277.                data = json.dumps(data)
278.                data = bytes(data, 'utf-8')
279.                try:
280.                    self.__s.sendall(data)
281.                except OSError:
```

```
282.                 print("disconnected")
283.
284.         def receive(self):
285.             try:
286.                 data = self.__s.recv(1024)
287.                 return json.loads(data)
288.             except ConnectionResetError as e:
289.                 self.connected = False
290.                 print(e)
291.             except WindowsError as e:
292.                 self.connected = False
293.                 print(e)
294.             except Exception as e:
295.                 print(e)
296.
297.         def update(self):
298.             while self.connected:
299.                 data = self.receive()
300.                 if data is not None:
301.                     self.__players = data
302.
303.         def update_data(self, key, value):
304.             self.__player_data[key] = value
305.
306.         def get_data(self, client_id, type):
307.             return self.get_players()[client_id][type]
308.
309.         def send_player_data(self):
310.             self.send(self.__player_data)
311.
312.         def get_players(self):
313.             return {int(k): v for k, v in self.__players.items()}
314.
315.         def get_client_id(self):
316.             return self.__client_id
317.
318.         def end(self):
319.             self.connected = False
320.             self.__s.close()
```

As you can see my model has grown in size significantly since my first prototype. The majority of the bulk has been added to accommodate my lobby system (and any error handling necessary for such a system). This system allows new connections to join a lobby before the game is started. This allowing and storing of connections is handled using the connection class in aggregation with the server class. This server class looks after each thread for each of the clients in its lobby (or list object 'connections' as it is called in the class).

Another significant change is the way in which data is shared within the network. In my early model, the only thing that was sent was a byte string. When testing this model in my game I had to come up with some code that allowed the necessary elements of the game (all of the sprite classes) to be serialised and sent as a byte string in this way. After testing however, I quickly realised due to the large size of these objects, that this approach was inappropriate. Instead, I adapted the way my multiplayer game mode (both the server and client version) handled data in order to minimise what was sent over the network.

This allowed me to only send the most necessary data over my network, which is detailed above in the 'player data' dictionary. This dictionary is sent to every client in the network after every loop on the server-side version of the game (please see the latest networking model in the high-level overview section for a visualisation). In essence, the new net code works with the following idea:

- The server listens for new connections.
- If a new connection is accepted the client sends its personal data as part of the initial handshake.

- The server sends back the player data which may contain information about other user's already in the lobby for the new client to display.
- The server then overwrites the default player data with the data the client provides.
- Then the server adds this connection to its own object that has a single thread within it (as opposed to two in the original model). This thread continually runs the receive method, which listens for data from the client.
- When the client sends data in response from the server sending player data (which can only include the client's user's desired move and ready state - due to the player_data dictionary within the Client class) the server overwrites its player data with this information.
- The server (like in the Classic mode) runs the game - calculating all collisions and scores etc, however, instead of receiving the desired move of the ghosts from the pathfinding algorithm, it will fetch it from the net code (player data)
- The server then works out the position, direction and score of each player in the game and overwrites these parts of the player data dictionary. Before the server-side game displays the sprites for the user it tells the net code to send off the player data to each user - ensuring the server-side user sees the sprites at roughly the same time as the clients.
- When the clients then receive this new position and direction data, it sends back the latest input data from the user and the process repeats again.

By using only one thread (to receive data) as opposed to two we minimise the number of times the network is used to send data. Furthermore, by continuing to use the thread for receiving we continue to avoid the problem identified in my analysis which is the slowing or complete crashing of the server when and if it were to wait for any particular client to send its data. This is helpful as it prevents a user's poor connection affecting the rest of the user's gameplay.

DATA STRUCTURES

Below are the two most important data structures within my project.

PRIORITY QUEUE

As discussed before, the A* algorithm stores all evaluated nodes in a priority queue. This means that we can choose the conditions for the next node to be evaluated. In this case, we want to choose the node with the smallest fScore. In other words, the node that has the smallest combination of distance to the target node and amount of parent nodes (length of the path to that node) and thus has the greatest probability to lead us to the target node whilst giving us the shortest path.

The most demanding aspect of my game is the pathfinding algorithm, so it is vital that the priority queue is written perfectly in order to maximise efficiency (since it is accessed so many times). Below is a prototype of the queue:

```
1.  class PriorityQueue:
2.
3.      def __init__(self):
4.          self.queue = []
5.
6.      def is_empty(self):
7.          return len(self.queue) == 0
8.
9.      def en_queue(self, node):
10.         self.queue.append(node)
11.
12.     def pop(self):
```

```
13.         self.queue.sort(key=lambda x: x.f_score)
14.         return self.queue.pop(0)
15.
16.     def has(self, child):
17.         for node in self.queue:
18.             if node.x == child.x and node.y == child.y:
19.                 return True
20.         return False
```

As you can see the algorithm must access the queue to perform several operations:

If the queue is empty the program knows that there are no more nodes that have children that can be evaluated. This means the program is able to tell whether there is no path to the target node and adjust accordingly.

When new children are evaluated, they are added to the queue. No sorting is done at this stage as many children may be added at one time (sorting every time a child is added would be inefficient).

Instead, the queue is sorted via their fScore whenever the algorithm pops a node to be evaluated. Sorting at this time ensures a sort is only carried out when it is absolutely necessary.

MAZE

The maze object stores information about each tile in the maze. Before it can be used, information about the maze to be played must be queried from the database. This will be returned as a JSON string like the following:

{"Level1":
[[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, 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, 2, 1, 6, 6, 1, 0, 1, 6, 6, 6, 1, 0, 1, 1, 0, 1, 6, 6, 6, 1, 0, 1, 6, 6, 1, 2, 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, 4, 1, 1, 4, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[5, 5, 5, 5, 5, 1, 0, 1, 1, 1, 1, 1, 4, 1, 1, 4, 1, 1, 1, 1, 1, 0, 1, 5, 5, 5, 5, 5],
[5, 5, 5, 5, 5, 1, 0, 1, 1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 1, 1, 0, 1, 5, 5, 5, 5, 5],
[5, 5, 5, 5, 5, 1, 0, 1, 1, 4, 1, 1, 1, 3, 3, 1, 1, 1, 4, 1, 1, 0, 1, 5, 5, 5, 5, 5],
[1, 1, 1, 1, 1, 1, 0, 1, 1, 4, 1, 4, 4, 4, 4, 4, 4, 1, 4, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[4, 4, 4, 4, 4, 4, 0, 4, 4, 4, 1, 4, 4, 4, 4, 4, 4, 1, 4, 4, 4, 0, 4, 4, 4, 4, 4, 4],
[1, 1, 1, 1, 1, 1, 0, 1, 1, 4, 1, 4, 4, 4, 4, 4, 4, 1, 4, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[5, 5, 5, 5, 5, 1, 0, 1, 1, 4, 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 0, 1, 5, 5, 5, 5, 5],
[5, 5, 5, 5, 5, 1, 0, 1, 1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 1, 1, 0, 1, 5, 5, 5, 5, 5],
[5, 5, 5, 5, 5, 1, 0, 1, 1, 4, 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 0, 1, 5, 5, 5, 5, 5],
[1, 1, 1, 1, 1, 1, 0, 1, 1, 4, 1, 1, 1, 1, 1, 1, 1, 4, 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, 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, 2, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 2, 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]]}

Only the smallest amount of information necessary is stored this way as it reduces the number of memory accesses. The state of the tiles is stored simply by their number in the above 2D list. The key is as follows:

0 - Pellet
1 - Wall
2 - Power pellet
3 - Ghost barrier (impassable by Pac-Man, but traversable by ghosts)
4 - Empty tile
5 - This is an area that neither Pac-Man nor the ghosts can reach/target not enclosed by walls.
6 - This is an area that neither Pac-Man nor the ghosts can reach/ target enclosed by walls.

The first 5 types have clear skin differences, however, the last two seem indifferent from the empty tile, however as I'll explore later, having these differentiated allows the generation of tile objects with correct images.

Once the 'tilemap' is fetched from the database, the maze object can start to turn each number into its own unique tile object. This is done while adding each one to another 2D list in the same fashion as the tilemap. If the tile is a wall, then the directory leading to the appropriate image of that wall is fetched by analysing the surrounding tiles. Currently, I do not believe there to be an elegant solution to this problem, so instead have adopted the following brute force method:

```
        adjacent = {}
        vectors = {'s': (0, 1),
                   'e': (1, 0),
                   'w': (-1, 0),
                   'n': (0, -1),
                   'ne': (1, -1),
                   'se': (1, 1),
                   'sw': (-1, 1),
                   'nw': (-1, -1)
                   }

        for key, value in vectors.items():
            x, y = value
            try:
                new_x = tile_x + x
                new_y = tile_y + y

                if new_x < 0 or new_y < 0:
                    adjacent.update({key: None})
                    continue

                adjacent.update({key: tile_map[tile_y + y][tile_x + x]})
            except IndexError:
                adjacent.update({key: None})

        values = tuple(list(adjacent.values())[:4])
        values_diag = tuple(adjacent.values())

        # ghost edges
        if values == (4, 3, 1, 4):
            return 'left_end_ghost'

        if values == (4, 1, 3, 4):
            return 'right_end_ghost'

        if values == (4, 1, 1, 4) and tile_y == 12:
            return 'lower_boundary'

        if values == (4, 1, 1, 4) and tile_y == 16:
```

```
            return 'upper_boundary'

        if values == (1, 4, 4, 1) and tile_x == 10:
            return 'right_boundary'

        if values == (1, 4, 4, 1) and tile_x == 17:
            return 'left_boundary'
```

This model outlines the method of figuring out which of some 30 different tile images is the correct one. It's worth mentioning that the reason this method is necessary is so that multiple mazes can be created/stored within the databases without taking up too much space. It also allows users to create their own mazes in the 'Creator' mode and have the mazes automatically update according to the original sprites in real-time.

Once each tile has been created and added to the Maze objects new 2D list of tile objects, the data structure can be used by both for pathfinding and of course the displaying of the maze.

TECHNICAL SOLUTION

My solution is split up into 13 scripts, two json files and one database. As shown:

KEY ALGORITHMS

Below are the page numbers for the most technically accomplished aspects of my solution:

149 - A* Pathfinding
167 - Networking
152 - Datastructures
189 - Database

CODE

MAIN

```
__author__ = 'Will Evans'

import pygame as pg
import splash_screens
import tutorial
import single_player
import multiplayer
import local_database
import local_settings
import text
import os


def create_window(win_scale):
    x = int(28 * 12 * win_scale)
    y = int(36 * 12 * win_scale)

    win = pg.display.set_mode((x, y))
    # noinspection PyUnresolvedReferences
    icon_path = os.path.join('Resources', 'pacman.gif')
    icon = pg.image.load(icon_path)
    pg.display.set_icon(icon)
    pg.display.set_caption('Pac Man')

    return win
```

```
26.
27.
28. if __name__ == '__main__':
29.
30.     os.environ['SDL_VIDEO_WINDOW_POS'] = "{}, {}.".format(25, 50)
31.
32.     # Gets settings
33.     win_scale = local_settings.get_setting('win_scale')
34.     music_volume = local_settings.get_setting('music_volume')
35.
36.     # PyGame set up
37.     pg.mixer.pre_init(44100, -16, 1, 512)  # Sets up sounds to play with very little latency.
38.     pg.init()
39.     pg.mixer_music.set_volume(music_volume / 100)
40.
41.     clock = pg.time.Clock()
42.
43.     # Creates window
44.     win = create_window(win_scale)
45.
46.     # Creates database
47.     local_database.create_db()
48.
49.     run = True
50.
51.     # List of programmes that can be run directly from this script
52.     programmes = {
53.
54.                     'StartScreen':    splash_screens.StartScreen,
55.                     'Story':          tutorial.Story,
56.                     'Classic':        single_player.Classic,
57.                     'Multiplayer':    multiplayer.Multiplayer,
58.                     'Create Game':    multiplayer.HostMenu,
59.                     'Join Game':      multiplayer.ClientMenu,
60.                     'Highscores':     splash_screens.Highscores
61.
62.                     }
63.
64.     # Attempt to login from file
65.     username = local_settings.get_setting('user_name')
66.     password = local_settings.get_setting('password')
67.     user_id = local_database.login(username, password)
68.
69.     # Essential information for starting the game
70.     program_name = 'StartScreen'
71.     program = programmes[program_name]
72.     running_program = program(win, win_scale, user_id)
73.
74.     # Information on error messages (these must be stored in the main script).
75.     error_message = None
76.     error_box = None
77.     buffer = program
78.
79.     # Mainloop
80.     while run:
81.
82.         # Change window scale
83.         if win_scale != local_settings.get_setting('win_scale'):
84.             win_scale = local_settings.get_setting('win_scale')
85.             running_program.quit()
86.             running_program = program(win, win_scale, user_id)
87.             running_program.sub_program_name = 'settings'
88.             running_program.sub_program = splash_screens.Settings(win, win_scale, user_id, running
    _program.icons)
89.             win = create_window(win_scale)
90.
91.         # Hand control to another program
92.         if buffer != program:
```

```
            buffer = program
            running_program.quit()
            pg.mixer.stop()
            running_program = program(win, win_scale, user_id)

        # See if the user wants to close the application
        events = pg.event.get()
                for event in events:
                    if event.type == pg.QUIT:
                        run = False
                        running_program.quit()

                # Error message
                if error_message is not None and error_box is None:
                    error_box = text.ErrorBox(error_message, win_scale)
                    error_box.display(win)

                if error_box is None:
                    running_program.run(win, events)
                else:
                    if not error_box.update(events):
                        error_box = None
                        error_message = None
                        running_program.error_message = None

                # Get information for next cycle
                try:
                    program = programmes[running_program.get_program()]
                    error_message = running_program.get_error()
                    user_id = running_program.user_id
                except KeyError:
                    error_message = 'Not found'
                    program = programmes['StartScreen']
                    buffer = None

                # Pygame Essential
                clock.tick(60)
                pg.display.update()

                if error_box is None and error_message is None:
                    win.fill((0, 0, 0))

            pg.quit()
            quit()
```

SPLASH SCREENS

```
__author__ = 'Will Evans'

import pygame as pg
import text
import icons
import local_settings
import local_database
import os


class StartScreen:
    def __init__(self, win, win_scale, user_id):
        """
        Displays the StartScreen and keeps track of all the objects on the screen and user interac
    tions with them.
        :param win: The current window, all objects must be blitted to this window to be displayed
    .
```

```
        :param win_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
        :param user_id: UserID if the user has signed in (they don't have to be signed in for the startscreen).
        """

        # Essential Information
        self.win = win
        self.win_scale = win_scale
        self.user_id = user_id
        self.clock = pg.time.Clock()
        self.program = 'StartScreen'
        self.sub_program_name = None
        self.sub_program = None
        self.sub_programs = {

            'loginscreen':  LoginScreen,
            'signupscreen': SignUpScreen,
            'settings':     Settings,
            'accounts':     Accounts

        }

        self.error_message = None

        # Pac-Man logo
        # Change this into a class
        pac_man_logo_path = os.path.join('resources', 'pac_man_logo.png')
        self.pac_title_scale = 80 * win_scale
        self.pac_title_size = (int(self.pac_title_scale * 3.8), self.pac_title_scale)
        self.pac_title = pg.transform.smoothscale(pg.image.load(pac_man_logo_path), self.pac_title_size)
        self.pac_title_rect = self.pac_title.get_rect(center=(168 * win_scale, 80 * win_scale))

        # Music
        theme_music_path = os.path.join('resources', 'sounds', 'startscreen', 'theme.mp3')
        pg.mixer.music.load(theme_music_path)

        # Icons
        self.icons = []

        #   Sound Icon
        sound_img_paths = [
                            os.path.join('resources', 'icons', 'unmute.png'),
                            os.path.join('resources', 'icons', 'mute.png')
                        ]
        self.icons.append(icons.Icon((326, 422), sound_img_paths, win_scale, sound=True, toggle=True))

        #   Settings Icon
        settings_img_path = [os.path.join('resources', 'icons', 'settings.png')]
        self.icons.append(icons.Icon((296, 422), settings_img_path, win_scale, target_program='settings'))

        #   Accounts Icon
        accounts_img_path = [os.path.join('resources', 'icons', 'accounts.png')]
        self.icons.append(icons.Icon((266, 422), accounts_img_path, win_scale, target_program='loginscreen'))

        self.y_spacing = 30
        self.font_size = 40

        # Choices
        self.__choices = self.get_choices(['Story', 'Classic', 'Multiplayer', 'Creator', 'Highscores'], win_scale)

        # Actions
        pg.mixer.music.play()
```

```
77.
78.     def get_choices(self, string_choices, win_scale):
79.         """
80.         Returns a list of live word objects. One for each choice in the string choices parameter.
81.         :param string_choices: A list of strings with the names of the possible choices/options th
    at the user can
82.         :param string_choices: A list of strings with the names of the possible choices/options th
    at the user can
83.         choose.
84.         :param win_scale: Window Scale (How large the window is - must be multiplied by all size r
    elated variables).
85.         :return: List of options.
86.         """
87.
88.         choices = []
89.         for num, content in enumerate(string_choices):
90.             y = 220 + num * self.y_spacing
91.             colour = (160, 205, 217) if num in [1, 3, 5] else (188, 47, 39)
92.             choices.append(text.LiveWord(content, y, self.font_size, win_scale, colour))
93.         return choices
94.
95.     def run(self, win, events):
96.         """
97.         Updates the screen with the text and icon objects.
98.         :param win: The current window, all objects must be blitted to this window to be displayed
    .
99.         :param events: Contains events from the pg.event.get() call containing all keyboard events
    .
100.                :return: None
101.                """
102.
103.                if self.sub_program_name is not None:
104.                    if self.sub_program is None or self.sub_program.error_message is not None:
105.                        self.sub_program = self.sub_programs[self.sub_program_name](win,
106.                                                                                    self.win_scale,
107.                                                                                    self.user_id,
108.                                                                                    self.icons)
109.                    else:
110.                        self.sub_program.run(win, events)
111.
112.                    self.error_message = self.sub_program.error_message
113.                    self.user_id = self.sub_program.user_id
114.                    program_name = self.sub_program.get_sub_program_name()
115.
116.                    if program_name is None:
117.                        self.sub_program_name = None
118.                        self.sub_program = None
119.
120.                    elif program_name != self.sub_program_name:
121.                        self.sub_program_name = program_name
122.                        self.sub_program = None
123.
124.                else:
125.                    for text in self.__choices:
126.                        if text.check_mouse(*pg.mouse.get_pos()):
127.                            text.react()
128.
129.                    self.update_objects(events)
130.
131.                    win.blit(self.pac_title, self.pac_title_rect)
132.
133.                    for word in self.__choices:
134.                        word.display(win)
135.
136.                    for icon in self.icons:
137.                        icon.display(win)
138.
```

```
139.            def update_objects(self, events):
140.                """
141.                Updates objects.
142.                :param events: Contains events from the pg.event.get() call containing all keyboard
    events.
143.                :return: None
144.                """
145.
146.                pos = get_mouse_input(events)
147.                if pos is not None:
148.                    for icon in self.icons:
149.                        if pos is not None:
150.                            if icon.check_click(*pos):
151.                                icon.action()
152.                                if icon.has_target:
153.                                    self.sub_program_name = icon.target_program
154.
155.                    for word in self.__choices:
156.                        if word.check_click(*pos):
157.                            self.program = word.get_program()
158.
159.            def get_program(self):
160.                return self.program
161.
162.            def get_error(self):
163.                return self.error_message
164.
165.            def quit(self):
166.                pg.mixer.music.stop()
167.
168.
169.        class SubProgram:
170.            def __init__(self, win, win_scale, user_id, icons_list):
171.                """
172.                Program within the menu screen. This parent class contains the methods that run / u
    pdate each sub menu as they
173.                are the same.
174.                :param win: The current window, all objects must be blitted to this window to be di
    splayed.
175.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
    size related variables).
176.                :param user_id: UserID if the user has signed in (they don't have to be signed in t
    o play Classic).
177.                :param icons_list: List of icon objects from the StartScreen.
178.                """
179.
180.                # Essential
181.                self.win = win
182.                self.win_scale = win_scale
183.                self.user_id = user_id
184.                self.icons = icons_list
185.                self.error_message = None
186.
187.                # Defined in subclasses
188.                self.sub_program_name = None
189.
190.                # Back button
191.                back_button_path = os.path.join('resources', 'icons', 'back_arrow.png')
192.                self.back_button = icons.Icon((35, 422), [back_button_path], win_scale, target_prog
    ram=None)
193.
194.            def run(self, win, events):
195.
196.                # Events
197.                for event in events:
198.                    if event.type == pg.KEYDOWN:
199.                        if event.key == pg.K_ESCAPE:
200.                            self.sub_program_name = None
```

```
201.
202.                 for icon in self.icons:
203.                     icon.display(win)
204.                 self.back_button.display(win)
205.                 self.update_icons(events)
206.
207.             def update_icons(self, events):
208.                 icons = self.icons[::]
209.                 icons.append(self.back_button)
210.                 pos = get_mouse_input(events)
211.                 if pos is not None:
212.                     for icon in icons:
213.                         if pos is not None:
214.                             if icon.check_click(*pos):
215.                                 icon.action()
216.                                 if icon.has_target:
217.                                     self.sub_program_name = icon.target_program
218.
219.             def get_sub_program_name(self):
220.                 return self.sub_program_name
221.
222.
223.         class SignUpScreen(SubProgram):
224.             def __init__(self, win, win_scale, user_id, icons):
225.                 """
226.                 Uses words and input boxes to display a sign up screen. It also updates the input b
    oxes and calls the local
227.                 database to check whether the details are valid, then (if they are) asks the local
    database to save the details.
228.                 :param win: The current window, all objects must be blitted to this window to be di
    splayed.
229.                 :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
230.                 :param user_id: UserID if the user has signed in (they don't have to be signed in t
    o play Classic).
231.                 :param icons: List of icon objects from the StartScreen.
232.                 """
233.
234.                 super().__init__(win, win_scale, user_id, icons)
235.
236.                 self.sub_program_name = 'signupscreen'
237.                 self.error_message = None
238.
239.                 self.enter_pressed = False
240.
241.                 # Sign Up Title
242.                 self.signup_title = text.Word('Sign Up!', (188, 90), (255, 255, 30), 80, win_scale,
     centre=True)
243.
244.                 # Username Input Box
245.                 self.username_input_box = text.InputBox(168, 206, 160, 30, 30, win_scale, 'Username
    ', centre=True)
246.
247.                 # Password Input Box
248.                 self.password_input_box = text.InputBox(168, 246, 160, 30, 30, win_scale, 'Password
    ', centre=True, private=True)
249.
250.                 # Password Confirm Box
251.                 self.password_confirm_input_box = text.InputBox(168, 286, 160, 30, 30,
252.                                                                 win_scale,
253.                                                                 'Password',
254.                                                                 centre=True,
255.                                                                 private=True)
256.
257.                 # Sign Up Button
258.                 self.signup_button = text.Button('Sign Up',
259.                                                  (168, 326),
260.                                                  (160, 30),
```

```
                                                30,
                                                (255, 255, 30),
                                                2,
                                                win_scale,
                                                centre=True)

            self.boxes = [
                            self.username_input_box,
                            self.password_input_box,
                            self.password_confirm_input_box,
                            self.signup_button
                        ]

        def run(self, win, events):
            super().run(win, events)

            # Updates
            for box in self.boxes:
                box.update(events)

            #   Sign up
            self.signup_button.update(events)

            # Display
            self.signup_title.display(win)

            for box in self.boxes:
                box.display(win)

            self.signup_button.display(win)

            if self.signup_button.get_click() or self.enter_pressed:
                self.enter_pressed = False
                approved, self.error_message = local_database.check_sign_up(
                                                                            self.username_input_box.user_input.lower(),
                                                                            self.password_input_box.user_input,
                                                                            self.password_confirm_input_box.user_input
                                                                            )
                if approved:
                    self.sub_program_name = 'loginscreen'
                    local_database.save_user(self.username_input_box.user_input.lower(), self.password_input_box.user_input)

            # Events
            for event in events:
                if event.type == pg.KEYDOWN:
                    # Allows the user to cycle through input boxes using tab
                    if event.key == pg.K_TAB:
                        try:
                            active_box = [box for box in self.boxes if box.tab_active][0]
                            active_box_index = self.boxes.index(active_box)
                            active_box.tab_active = False
                            active_box.active = False
                            self.boxes[(active_box_index + 1) % len(self.boxes)].tab_active = True
                        except IndexError:
                            self.boxes[0].tab_active = True
                    elif event.key == pg.K_RETURN:
                        self.enter_pressed = True


    class LoginScreen(SubProgram):
        def __init__(self, win, win_scale, user_id, icons):
            """
            Class for the login screen.
```

```
324.                :param win: The current window, all objects must be blitted to this window to be di
    splayed.
325.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
326.                :param user_id: UserID if the user has signed in (they don't have to be signed in t
    o play Classic).
327.                :param icons: List of icon objects from the StartScreen.
328.                """
329.
330.                # Essential
331.                super().__init__(win, win_scale, user_id, icons)
332.                self.sub_program_name = 'loginscreen' if user_id is None else 'accounts'
333.
334.                self.enter_pressed = False
335.
336.                # Login Title
337.                self.login_title = text.Word('Login!', (188, 90), (255, 255, 30), 80, win_scale, ce
    ntre=True)
338.
339.                # Username Input Box
340.                self.username_input_box = text.InputBox(168, 206, 150, 30, 30, win_scale, 'Username
    ', centre=True)
341.
342.                # Password Input Box
343.                self.password_input_box = text.InputBox(168, 246, 150, 30, 30, win_scale, 'Password
    ', centre=True, private=True)
344.
345.                # Login Button
346.                self.login_button = text.Button('Login', (168, 286), (160, 30), 30, (255, 255, 30),
     2, win_scale, centre=True)
347.
348.                # SignUp Button
349.                self.signup_button = text.Button('SignUp', (168, 286), (160, 30), 30, (255, 255, 30
    ), 2, win_scale, centre=True)
350.
351.                self.boxes = [self.username_input_box, self.password_input_box, self.signup_button]

352.
353.                # Remember me Buttons
354.                self.remember_me = False
355.                self.remember_me_green = text.Button('Remember me',
356.                                                     (168, 326),
357.                                                     (160, 30),
358.                                                     24,
359.                                                     (0, 222, 222),
360.                                                     2,
361.                                                     win_scale,
362.                                                     centre=True)
363.
364.                self.remember_me_red = text.Button('Remember me',
365.                                                   (168, 326),
366.                                                   (160, 30),
367.                                                   24,
368.                                                   (222, 0, 0),
369.                                                   2,
370.                                                   win_scale,
371.                                                   centre=True)
372.
373.                self.remember_me_buttons = [self.remember_me_red, self.remember_me_green]
374.
375.            def run(self, win, events):
376.                super().run(win, events)
377.
378.                # Updates
379.                for box in self.boxes:
380.                    box.update(events)
381.
382.                if self.password_input_box.text_entered:
```

```
383.                 self.boxes = [self.username_input_box, self.password_input_box, self.login_butt
    on]
384.
385.             else:
386.                 self.boxes = [self.username_input_box, self.password_input_box, self.signup_but
    ton]
387.
388.             #   Sign Up
389.             active = self.signup_button.active or self.signup_button.tab_active
390.             if self.signup_button.get_click() or (self.enter_pressed and active):
391.                 self.sub_program_name = 'signupscreen'
392.
393.             #   Login
394.             active = self.signup_button.active or self.signup_button.tab_active
395.             if self.login_button.get_click() or (self.enter_pressed and not active):
396.                 user_name = self.username_input_box.user_input.lower()
397.                 password = self.password_input_box.user_input
398.                 self.user_id = local_database.login(user_name, password)
399.                 if self.user_id is None:
400.                     self.error_message = 'Incorrect login details'
401.                 else:
402.                     if self.remember_me:
403.                         local_settings.save_setting('user_name', user_name)
404.                         local_settings.save_setting('password', password)
405.                     self.sub_program_name = 'accounts'
406.
407.             #   Remember me buttons
408.             self.remember_me_buttons[0].update(events)
409.             if self.remember_me_buttons[0].get_click():
410.                 self.remember_me = not self.remember_me
411.                 self.remember_me_buttons.reverse()
412.
413.             # Display
414.             self.login_title.display(win)
415.
416.             for box in self.boxes:
417.                 box.display(win)
418.
419.             self.remember_me_buttons[0].display(win)
420.
421.             # Events
422.             for event in events:
423.                 if event.type == pg.KEYDOWN:
424.
425.                     # Tab moves the active box along in the list above (for ease of logging in)

426.                     if event.key == pg.K_TAB:
427.                         try:
428.                             active_box = [box for box in self.boxes if (box.tab_active or box.a
    ctive)][0]
429.                             active_box_index = self.boxes.index(active_box)
430.                             active_box.tab_active = False
431.                             active_box.active = False
432.                             self.boxes[(active_box_index + 1) % len(self.boxes)].tab_active = T
    rue
433.                         except IndexError:
434.                             self.boxes[0].tab_active = True
435.                     elif event.key == pg.K_RETURN:
436.                         self.enter_pressed = True
437.
438.
439.     class Settings(SubProgram):
440.         def __init__(self, win, win_scale, user_id, icons=None):
441.             """
442.             Class for settings menu. At the moment it only lets you change the win_scale and re
    loads the pygame window when
443.             this is done.
```

```
444.                :param win: Window Scale (How large the window is - must be multiplied by all size
    related variables).
445.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
446.                :param user_id: UserID if the user has signed in (they don't have to be signed in t
    o play Classic).
447.                :param icons: List of icon objects from the StartScreen.
448.                """
449.
450.                # Essential
451.                super().__init__(win, win_scale, user_id, icons)
452.                self.sub_program_name = 'settings'
453.
454.                # Settings Title
455.                self.settings_title = text.Word('Settings!', (178, 90), (255, 255, 30), 60, win_sca
    le, centre=True)
456.
457.                # Sliders
458.                self.music_slider = (text.Slider('Music  ',
459.                                                 168, 200, 200, 25,
460.                                                 20, (255, 255, 30), 3,
461.                                                 win_scale,
462.                                                 level=int(local_settings.get_setting('music_volume
    ')),
463.                                                 centre=True))
464.
465.                self.game_slider = (text.Slider('Game Sound  ',
466.                                                168, 240, 200, 25,
467.                                                20, (255, 255, 30), 3,
468.                                                win_scale,
469.                                                level=int(local_settings.get_setting('game_volume')
    ),
470.                                                centre=True))
471.
472.                self.win_scale_slider = (text.Slider('Win Scale  ',
473.                                                     168, 280, 200, 25,
474.                                                     20, (255, 255, 30), 3,
475.                                                     win_scale,
476.                                                     level=int(local_settings.get_setting('win_scal
    e')),
477.                                                     centre=True,
478.                                                     levels=5))
479.
480.            def run(self, win, events):
481.                super().run(win, events)
482.
483.                # Events
484.                for event in events:
485.                    if event.type == pg.KEYDOWN:
486.                        if event.key == pg.K_ESCAPE:
487.                            return None
488.
489.                # Updates
490.                self.music_slider.update(events)
491.                pg.mixer.music.set_volume(self.music_slider.level_string/100)
492.                if local_settings.get_setting('music_volume') != self.music_slider.level_string:
493.                    local_settings.save_setting('music_volume', self.music_slider.level_string)
494.
495.                self.game_slider.update(events)
496.                if local_settings.get_setting('game_volume') != self.game_slider.level_string:
497.                    local_settings.save_setting('game_volume', self.game_slider.level_string)
498.
499.                self.win_scale_slider.update(events)
500.                for event in events:
501.                    if event.type == pg.MOUSEBUTTONUP:
502.                        if local_settings.get_setting('win_scale') != self.win_scale_slider.level_s
    tring:
```

```
503.                         local_settings.save_setting('win_scale', self.win_scale_slider.level_st
    ring)
504.
505.                 # Display
506.                 self.settings_title.display(win)
507.
508.                 for icon in self.icons:
509.                     icon.display(win)
510.
511.                 self.music_slider.display(win)
512.                 self.game_slider.display(win)
513.                 self.win_scale_slider.display(win)
514.
515.
516.         class Accounts(SubProgram):
517.             def __init__(self, win, win_scale, user_id, icons):
518.
519.                 # Essential
520.                 super().__init__(win, win_scale, user_id, icons)
521.                 self.sub_program_name = 'accounts'
522.
523.                 # Login Title
524.                 self.accounts_title = text.Word('Accounts!', (180, 90), (255, 255, 30), 60, win_sca
    le, centre=True)
525.
526.                 # Logged in as
527.                 self.user_name = text.Word(
528.                                             'Logged in as:       {}'.format(local_database.get_user
    name(user_id)),
529.                                             (310, 120),
530.                                             (255, 255, 30),
531.                                             15,
532.                                             win_scale,
533.
534.                                         )
535.
536.                 # Statistics
537.                 self.statistics = []
538.                 colour = (255, 255, 30)
539.                 stats_dict = local_database.get_statistics(user_id)
540.                 if stats_dict is None:
541.                     string_1 = 'Statistics will appear'
542.                     string_2 = 'once you play a game'
543.                     self.statistics.append(text.Word(string_1, (336 / 2, 250), colour, 25, win_scal
    e, centre=True))
544.                     self.statistics.append(text.Word(string_2, (336 / 2, 270), colour, 25, win_scal
    e, centre=True))
545.                 else:
546.                     for num, (stat_name, stat) in enumerate(stats_dict.items()):
547.                         self.statistics.append(text.Word(stat_name.replace('_', ' '),
548.                                                          (20, 240 + 25 * num),
549.                                                          colour, 20, win_scale,
550.                                                          left=True))
551.
552.                         self.statistics.append(text.Word(str(stat), (311, 240 + 25 * num), colour,
    20, win_scale))
553.
554.                 # Log out button
555.                 self.log_out_button = text.Button('Log Out', (47, 400), (75, 20), 20, (255, 255, 30
    ), 2, win_scale)
556.
557.                 # Stay logged in buttons
558.                 self.remember_me = False
559.                 self.remember_me_green = text.Button('Remember me', (130, 400), (105, 20), 16, (0,
    222, 222), 2, win_scale)
560.                 self.remember_me_red = text.Button('Remember me', (130, 400), (105, 20), 16, (222,
    0, 0), 2, win_scale)
561.                 self.remember_me_buttons = [self.remember_me_green, self.remember_me_red]
```

```
562.                if local_settings.get_setting('user_name') is None:
563.                    self.remember_me_buttons.pop(0)
564.
565.            def run(self, win, events):
566.                super().run(win, events)
567.
568.                # Updates
569.                self.log_out_button.update(events)
570.
571.                self.remember_me_buttons[0].update(events)
572.                if self.remember_me_buttons[0].get_click():
573.                    if len(self.remember_me_buttons) is 2:
574.                        local_settings.save_setting('user_name', None)
575.                        local_settings.save_setting('password', None)
576.                        self.remember_me_buttons.pop(0)
577.
578.                # Display
579.                self.accounts_title.display(win)
580.                self.user_name.display(win)
581.                self.remember_me_buttons[0].display(win)
582.
583.                for stat in self.statistics:
584.                    stat.display(win)
585.                self.log_out_button.display(win)
586.
587.                # Events
588.                if self.log_out_button.get_click():
589.                    self.user_id = None
590.                    self.sub_program_name = 'loginscreen'
591.
592.
593.        class PauseScreen:
594.            def __init__(self, win, win_scale, user_id, icons):
595.                """
596.                Runs when a user selects the escape key during a game. Gives the user the option to
    resume or quit.
597.                :param win: The current window, all objects must be blitted to this window to be di
    splayed.
598.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
    size related variables).
599.                :param user_id: UserID if the user has signed in (they don't have to be signed in t
    o play Classic).
600.                :param icons: List of icon objects from the StartScreen.
601.                """
602.
603.                # Essential
604.                self.win = win
605.                self.win_scale = win_scale
606.                self.user_id = user_id
607.                self.icons = icons
608.
609.            def run(self, win, events):
610.
611.                # Events
612.                for event in events:
613.                    if event.type == pg.KEYDOWN:
614.                        if event.key == pg.K_ESCAPE:
615.                            return None
616.
617.
618.        def get_mouse_input(events):
619.            for event in events:
620.                if event.type == pg.MOUSEBUTTONUP:
621.                    return event.pos
622.
623.
624.        class Highscores:
625.            def __init__(self, win, win_scale, user_id):
```

```
626.                """
627.                Displays all the highscores that are saved in the database.
628.                :param win: The current window, all objects must be blitted to this window to be di
    splayed.
629.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
630.                :param user_id: UserID if the user has signed in (they don't have to be signed in t
    o play Classic).
631.                """
632.
633.                # Essential Information
634.                self.win = win
635.                self.win_scale = win_scale
636.                self.user_id = user_id
637.                self.clock = pg.time.Clock()
638.                self.program = 'Highscores'
639.                self.error_message = None
640.
641.                # Title
642.                self.title = text.Word('Highscores!', (175, 90), (255, 255, 30), 45, win_scale, cen
    tre=True)
643.                self.text = text.Word('Press any key to return...', (326, 417), (255, 255, 30), 15,
     win_scale)
644.
645.                # Highscores
646.                colour = (255, 255, 30)
647.                rank_word = text.Word('Rank', (20, 140), colour, 20, win_scale, left=True)
648.                score_word = text.Word('Score', (120, 140), colour, 20, win_scale, left=True)
649.                name_word = text.Word('Name', (311, 140), colour, 20, win_scale)
650.                self.highscore_titles = [rank_word, score_word, name_word]
651.
652.                self.highscores = []
653.                place_details = {
654.                    0:
655.                        {
656.                            'string': '1st',
657.                            'colour': (255, 215, 0)
658.                        },
659.
660.                    1:
661.                        {
662.                            'string': '2nd',
663.                            'colour': (220, 220, 220)
664.                        },
665.
666.                    2:
667.                        {
668.                            'string': '3rd',
669.                            'colour': (205, 127, 50)
670.                        }
671.                }
672.
673.                for num, (score, name) in enumerate(local_database.get_highscores()):
674.                    if num <= 2:
675.                        place_string = place_details[num]['string']
676.                        colour = place_details[num]['colour']
677.                    else:
678.                        place_string = '{}th'.format(num + 1)
679.                        colour = (169, 169, 169)
680.
681.                    self.highscores.append(text.Word(place_string, (20, 160 + 25 * num), colour, 20
    , win_scale, left=True))
682.                    self.highscores.append(text.Word(str(score), (120, 160 + 25 * num), colour, 20,
     win_scale, left=True))
683.                    self.highscores.append(text.Word(name, (311, 160 + 25 * num), colour, 20, win_s
    cale))
684.
685.                    if num is 9:
```

```
686.                     break
687.
688.         def run(self, win, events):
689.
690.             # Events
691.             for event in events:
692.                 if event.type in [pg.KEYDOWN, pg.MOUSEBUTTONUP]:
693.                     self.program = 'StartScreen'
694.
695.             # Display
696.             self.title.display(win)
697.
698.             for title in self.highscore_titles:
699.                 title.display(win)
700.
701.             for word in self.highscores:
702.                 word.display(win)
703.
704.             self.text.display(win)
705.
706.         def get_program(self):
707.             return self.program
708.
709.         def get_error(self):
710.             return self.error_message
711.
712.         def quit(self):
713.             pass
714.
715.
716.     if __name__ == '__main__':
717.         pass
```

TUTORIAL

```
1.  __author__ = 'Will Evans'
2.
3.  import pygame as pg
4.  import threading
5.  import os
6.  import text
7.  import sprites
8.  import maze
9.  import local_database
10. import local_settings
11. from time import sleep
12. import json
13.
14.
15. class Story:
16.     def __init__(self, win, win_scale, user_id):
17.         """
18.         Controls the running of each level and database queries.
19.         :param win: The current window, all objects must be blitted to this window to be displayed
    .
20.         :param win_scale: Window Scale (How large the window is - must be multiplied by all size r
    elated variables).
21.         :param user_id: UserID if the user has signed in (they don't have to be signed in to play
    Classic).
22.         """
23.
24.         # Essential
25.         self.win_scale = win_scale
26.         self.program = 'Story'
27.         self.user_id = user_id
```

```
        self.error_message = None
        self.game_finished = False
        self.win = win

        # Maze
        self.maze_id = 1

        # Variables
        self.score = 0

        #   Database
        self.game_id = local_database.get_game_id(self.user_id, self.maze_id)
        self.pellets_eaten = 0

        # Level Info
        self.level_num = 3

        # Tutorial Messages

        tutorial_prompts_file_path = os.path.join('data', 'tutorial.json')
        with open(tutorial_prompts_file_path, 'r') as file:
            self.tutorial_prompts = json.load(file)

        # Boxes for the first level
        self.tutorial_boxes = []
        for prompt in self.tutorial_prompts[str(self.level_num)]:
            self.tutorial_boxes.append(text.TutorialTextBox(prompt, (255, 255, 30), win_scale, add_mspacman=True))
        # Level
        self.level_names = {

                                1:    Level1,
                                2:    Level2,
                                3:    Level3,
                                4:    Level4,
                                5:    Level5,
                                6:    Level6,

                            }

        self.level = self.level_names[self.level_num](
                                self.win_scale,
                                self.game_id, self.level_num,
                                self.maze_id,
                                [],
                                [],
                                self.score,
                                self.tutorial_boxes,
                            )

    def run(self, win, events):
        """
        This method is run directly from the main script.
        :param win: The current window, all objects must be blitted to this window to be displayed.
        :param events: Contains events from the pg.event.get() call containing all keyboard events.
        :return: None
        """

        # Essential
        self.level.run(win, events)

        # Variables
        self.score = self.level.score
        self.pellets_eaten = self.level.pellets_eaten

        # Events
```

```
        for event in events:
            if event.type == pg.KEYDOWN:
                if event.key == pg.K_ESCAPE:
                    self.program = 'StartScreen'

        # If all the pellets have been eaten
        if self.level.finished and self.level.won:
                    self.level_num += 1
                    self.level.quit()

                    # Tutorial Messages
                    self.tutorial_boxes = []
                    for prompt in self.tutorial_prompts[str(self.level_num)]:
                        self.tutorial_boxes.append(text.TutorialTextBox(prompt,
                                                                        (255, 255, 30),
                                                                        self.win_scale,
                                                                        add_mspacman=True)
                                                   )

                    self.level = self.level_names[self.level_num](
                                        self.win_scale,
                                        self.game_id,
                                        self.level_num,
                                        self.maze_id,
                                        [],
                                        [],
                                        self.score,
                                        self.tutorial_boxes,
                                    )

                # If Pac-Man has died
                elif self.level.finished:
                    self.score = self.level.score
                    self.tutorial_boxes = self.level.tutorial_boxes
                    self.level.quit()

                    self.level = self.level_names[self.level_num](
                                        self.win_scale,
                                        self.game_id,
                                        self.level_num,
                                        self.maze_id,
                                        self.level.pellets,
                                        self.level.power_pellets,
                                        self.score,
                                        self.tutorial_boxes,
                                    )

            def get_program(self):
                return self.program

            def get_error(self):
                return self.error_message

            def quit(self):
                self.level.quit()
                pg.mixer.music.stop()


        class Level:
            def __init__(self, win_scale, game_id, level_num, maze_id, pellets, power_pellets, scor
    e, tutorial_boxes):
                """
                Responsible for running each level by calling sprite objects and handling their upd
    ates. The class
                also controls other things, such as score, displaying maze etc.
                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
                :type win_scale: Integer.
```

```
158.                :param game_id: GameID given to the game by the database.
159.                :type game_id: Integer.
160.                :param level_num: This controls difficulty of the ghosts. The larger the level numb
    er, the more levels Pac-Man
161.                has completed and the harder the ghosts become.
162.                :type level_num: Integer.
163.                :param maze_id: ID given to each maze. This can be used by the database to return t
    he 2D list associated with
164.                the ID.
165.                :type maze_id: Integer.
166.                :param pellets: A list of pellets. This is if Pac-
    Man dies and the level starts over. The pellets must be the
167.                same as the previous level instance so they are saved by the classic class.
168.                :type pellets: List.
169.                :param power_pellets: A list of power pellets. This is if Pac-
    Man dies and the level starts over. The power
170.                pellets must be the same as the previous level instance so they are saved by the cl
    assic class.
171.                :type power_pellets: List.
172.                :param score: Current score that should be displayed at the top.
173.                :type score: Integer.
174.                """
175.
176.                import text
177.                # Essential
178.                self._run = True
179.                self.win_scale = win_scale
180.                self.program = 'Story'
181.                self.finished = False
182.                self.won = False
183.                self.start = False
184.
185.                # Variables
186.                #   Level
187.                self.score = score
188.
189.                self.game_maze = maze.Maze(maze_id, win_scale)
190.
191.                #   Database
192.                self.level_num = level_num
193.                self.game_id = game_id
194.                self.level_score = 0
195.                self.length = 0
196.                self.pellets_eaten = 0
197.                self.power_pellets_eaten = 0
198.                self.ghosts_eaten = 0
199.
200.                # Tutorial Variables
201.                #   Messages
202.                self.tutorial_boxes = tutorial_boxes
203.                self.paused = False
204.
205.                #   Paths
206.                self.show_paths = False
207.
208.                # Length counting thread
209.                threading.Thread(target=self.second_count).start()
210.
211.                # Indicators
212.                self.score_position = (7 * 12, 2 * 12)
213.                self.score_indicator = text.Word('{}'.format(self.score), self.score_position, (234
    , 234, 234), 24, win_scale)
214.
215.                # Start and ready text
216.                self.ready_text = text.Word('ready!', (17.5 * 12, 20.5 * 12), (255, 255, 30), 23, w
    in_scale, italic=True)
217.                self.game_over_text = text.Word('game over', (18 * 12, 20.5 * 12), (255, 0, 0), 21,
     win_scale)
```

```
218.
219.                 # Points (displayed when Pac-Man eats a ghost).
220.                 self.points_texts = {}
221.                 for point_text in os.listdir('Resources\\sprites\\{}'.format('points')):
222.                     # noinspection PyUnresolvedReferences
223.                     self.points_texts.update({text: pg.transform.scale(
224.                         pg.image.load('Resources\\sprites\\{}\\{}'.format('points', point_text)),
225.                         ((24 * win_scale), (10 * win_scale)))})
226.
227.                 # Pac-Man
228.                 self.pac_man = sprites.PacMan('pac-man', self.game_maze, win_scale)
229.
230.                 # Ghosts
231.                 self.ghosts = []
232.                 self.ghosts_copy = self.ghosts[::]
233.                 blinky = sprites.Blinky('blinky', self.pac_man, self.game_maze, win_scale, level_nu
    m)
234.                 pinky = sprites.Pinky('pinky', self.pac_man, self.game_maze, win_scale, level_num)
235.                 clyde = sprites.Clyde('clyde', self.pac_man, self.game_maze, win_scale, level_num)
236.                 inky = sprites.Inky('inky', self.pac_man, self.game_maze, win_scale, level_num, bli
    nky)
237.
238.                 if level_num == 1:
239.                     pass
240.                 elif level_num == 2:
241.                     self.ghosts.append(clyde)
242.                 elif level_num == 3:
243.                     self.ghosts.append(pinky)
244.                 elif level_num == 4:
245.                     self.ghosts.append(blinky)
246.                 elif level_num == 5:
247.                     self.ghosts.append(blinky)
248.                     self.ghosts.append(inky)
249.                 elif level_num == 6:
250.                     self.ghosts.append(blinky)
251.                     self.ghosts.append(inky)
252.                 else:
253.                     self.ghosts.append(blinky)
254.                     self.ghosts.append(pinky)
255.                     self.ghosts.append(inky)
256.                     self.ghosts.append(clyde)
257.
258.                 # Pellets
259.                 self.pellets = pellets
260.                 self.power_pellets = power_pellets
261.
262.                 if pellets == [] and power_pellets == []:
263.                     pellet_skin_path = os.path.join('Resources', 'sprites', 'pellets', 'pellet.png'
    )
264.                     pellet_skin = pg.transform.scale(pg.image.load(pellet_skin_path), (4 * win_scal
    e, 4 * win_scale))
265.                     power_pellet_skin_path = os.path.join('Resources', 'sprites', 'pellets', 'power
    _pellet.png')
266.                     power_pellet_skin = pg.transform.scale(pg.image.load(power_pellet_skin_path),
267.                                                            (12 * win_scale, 12 * win_scale)
268.                                                            )
269.
270.                     pellet_sound_channel = pg.mixer.Channel(2)
271.                     pellet_sound_channel.set_volume(0.5 * (local_settings.get_setting('game_volume'
    )/100))
272.
273.                     power_pellet_death_sound_path = os.path.join('Resources', 'sounds', 'pellet', '
    death.wav')
274.                     pellet_death_sound = pg.mixer.Sound(power_pellet_death_sound_path)
275.
276.                     for row in self.game_maze.tiles:
```

```
                        for tile in row:
                            if tile.type == 'pellet':
                                self.pellets.append(sprites.Pellet(pellet_skin,
                                                                   tile,
                                                                   self.pac_man,
                                                                   win_scale,
                                                                   pellet_death_sound,
                                                                   pellet_sound_channel)
                                                    )

                            elif tile.type == 'power_pellet':
                                if level_num >= 5:
                                    self.power_pellets.append(sprites.Pellet(power_pellet_skin,
                                                                             tile,
                                                                             self.pac_man,
                                                                             win_scale,
                                                                             pellet_death_sound,
                                                                             pellet_sound_channel,

                                                                             power_pellet=True)
                                                              )

                                else:
                                    self.pellets.append(sprites.Pellet(pellet_skin,
                                                                       tile,
                                                                       self.pac_man,
                                                                       win_scale,
                                                                       pellet_death_sound,
                                                                       pellet_sound_channel)
                                                        )

                else:
                    for pellet in self.pellets:
                        pellet.predator = self.pac_man
                    for power_pellet in self.power_pellets:
                        power_pellet.predator = self.pac_man

                # Sound
                self.large_pellet_channel = pg.mixer.Channel(3)
                self.large_pellet_channel.set_volume(0.5 * (local_settings.get_setting('game_volume
')/100))
                large_pellet_sound_path = os.path.join('Resources', 'sounds', 'large_pellet_loop.wa
v')
                self.large_pellet_sound = pg.mixer.Sound(large_pellet_sound_path)

                self.death_sound_playing = False
                self.started = False

                # Clocks and counts
                self.start_clock = 0
                self.one_up_clock = 0
                self.flashing_map_clock = 0
                self.flashing_map_count = 0

            def run(self, win, events):
                """
                This method is run from the Tutorial class and updates and displays by one frame.
                :param win: The current window, all objects must be blitted to this window to be di
splayed.
                :type win: Surface.
                :param events: Contains events from the pg.event.get() call containing all keyboard
 events.
                :type events: Tuple.
                :return: Boolean returns True if level is won, False if level is lost, None otherwi
se
                """

                # Change to make sure there is a delay when there are no text boxes.
```

```
339.                 if not self.start:
340.
341.                     # Score Indicators
342.                     self.score_indicator.display(win)
343.
344.                     # Maze
345.                     self.game_maze.display(win)
346.
347.                     self.ready_text.display(win)
348.
349.                     # Pellets
350.                     for pellet in self.pellets:
351.                         pellet.display(win)
352.
353.                     #  Power pellets
354.                     for power_pellet in self.power_pellets:
355.                         power_pellet.display(win)
356.
357.                     # Ghosts
358.                     for ghost in self.ghosts:
359.                         ghost.display(win)
360.
361.                     # Pac-Man
362.                     self.pac_man.display(win)
363.
364.                 elif self.paused:
365.                     self.score_indicator.display(win)
366.                     self.game_maze.display(win)
367.
368.                     for pellet in self.pellets:
369.                         pellet.display(win)
370.                     for power_pellet in self.power_pellets:
371.                         power_pellet.display(win)
372.                     for ghost in self.ghosts:
373.                         ghost.display(win)
374.                     self.pac_man.display(win)
375.
376.                 # If there are no pellets the maze will flash
377.                 elif len(self.pellets) == 0:
378.                     self.flashing_map_clock += 1/60
379.                     if self.flashing_map_clock > 0.25:
380.                         self.flashing_map_clock = 0
381.                         self.flashing_map_count += 1
382.                         self.game_maze.change_skin()
383.                     if self.flashing_map_count == 7:
384.                         self.finished = True
385.                         self.won = True
386.
387.                     self.game_maze.display(win)
388.
389.                     self.score_indicator.display(win)
390.                     self.pac_man.display(win)
391.
392.                 # Mainloop of the level
393.                 else:
394.
395.                     # Score Indicators
396.                     self.score_indicator = text.Word('{}'.format(self.score),
397.                                                      self.score_position,
398.                                                      (234, 234, 234),
399.                                                      24,
400.                                                      self.win_scale
401.                                                      )
402.
403.                     self.score_indicator.display(win)
404.
405.                     # Maze
406.                     self.game_maze.display(win)
```

```
407.
408.                     # Ghost Paths
409.                     if self.show_paths:
410.                         for ghost in self.ghosts:
411.                             ghost.draw_path(win)
412.
413.                     # Pellets
414.                     for pellet in self.pellets:
415.                         pellet.update()
416.                         pellet.display(win)
417.                         if pellet.eaten:
418.                             self.score += 10
419.                             self.level_score += 10
420.                             self.pellets_eaten += 1
421.                             self.pellets.remove(pellet)
422.
423.                     #  Power pellets
424.                     for power_pellet in self.power_pellets:
425.                         power_pellet.update()
426.                         power_pellet.display(win)
427.                         if power_pellet.eaten:
428.                             self.score += 50
429.                             self.level_score += 50
430.                             self.power_pellets_eaten += 1
431.                             if not self.large_pellet_channel.get_busy():
432.                                 self.large_pellet_channel.play(self.large_pellet_sound, loops=-1)
433.                             for ghost in self.ghosts:
434.                                 ghost.scare()
435.                                 self.ghosts_copy = [ghost for ghost in self.ghosts if not ghost.dea
    d]
436.                             self.power_pellets.remove(power_pellet)
437.
438.                     # Pac-Man
439.                     self.pac_man.update(events)
440.                     self.pac_man.display(win)
441.
442.                     # Checks whether the level has ended
443.                     if self.pac_man.dead:
444.                         if not self.death_sound_playing:
445.                             pg.mixer.stop()
446.                             self.death_sound_playing = True
447.
448.                     if self.pac_man.death_animation_finished and not self.finished:
449.                         self.finished = True
450.
451.                     # Ghosts
452.                     if all([not ghost.scared for ghost in self.ghosts]):
453.                         self.large_pellet_channel.stop()
454.
455.                     for ghost in self.ghosts_copy:
456.                         if ghost.dead:
457.                             self.ghosts_eaten += 1
458.                             self.ghosts_copy.remove(ghost)
459.                             for ghost_ in self.ghosts_copy:
460.                                 ghost_.display(win)
461.                             points = 200 * 2 ** ((len(self.ghosts) - 1) - len(self.ghosts_copy))
462.                             self.level_score += points
463.                             self.score += points
464.
465.                             win.blit(self.points_texts['{}.png'.format(str(points))],
466.                                      (self.pac_man.x, self.pac_man.y - 8 * self.win_scale))
467.                             pg.display.update()
468.                             sleep(1)
469.
470.                     if not self.pac_man.dead:
471.                         for ghost in self.ghosts:
472.                             if ghost.__class__.__name__ == 'Blinky':
473.                                 if len(self.pellets) < 20 + 2 * self.level_num:
```

```
474.                                 ghost.make_elroy()
475.                             if len(self.pellets) < 10 + 2 * self.level_num:
476.                                 ghost.elroy_upgrade()
477.
478.                         ghost.update(events)
479.                         ghost.display(win)
480.
481.         def quit(self):
482.             """
483.             Quits the level. Stops music playing and saves that level to the databse.
484.             :return: None
485.             """
486.
487.             local_database.save_level(
488.                                       self.level_num,
489.                                       self.game_id,
490.                                       None,
491.                                       self.level_score,
492.                                       self.length,
493.                                       self.pellets_eaten,
494.                                       self.power_pellets_eaten,
495.                                       self.ghosts_eaten
496.                                   )
497.
498.             self._run = False
499.             pg.mixer.stop()
500.
501.         def second_count(self):
502.             """
503.             Counts seconds, so that the time can be recorded in the database.
504.             :return: None
505.             """
506.
507.             while self._run:
508.                 sleep(1)
509.                 self.length += 1
510.
511.
512.     class Level1(Level):
513.         """
514.         In level one the user is introduced to the movement mechanics (using arrow keys). They
    are also introudced to
515.         the first and least threatening ghost: clyde. They are shown the behaviours that all gh
    osts have and the
516.         behaviour of clyde. This is shown through the paths that are displayed.
517.         """
518.
519.         def __init__(self, win_scale, game_id, level_num, maze_id, pellets, power_pellets, scor
    e, start_cap=2):
520.             super().__init__(win_scale, game_id, level_num, maze_id, pellets, power_pellets, sc
    ore, start_cap)
521.
522.         def run(self, win, events):
523.             super().run(win, events)
524.
525.             # Tutorial Events
526.             self.pellets = self.pellets[-1:]
527.             if not self.start:
528.                 if self.tutorial_boxes[0].finished:
529.                     self.start = True
530.                 else:
531.                     self.tutorial_boxes[0].update(events)
532.                     self.tutorial_boxes[0].display(win)
533.
534.             if len(self.pellets) == 0 and not self.tutorial_boxes[1].finished:
535.                 self.paused = True
536.                 self.tutorial_boxes[1].update(events)
537.                 self.tutorial_boxes[1].display(win)
```

```
538.                     if self.tutorial_boxes[1].finished:
539.                         self.paused = False
540.
541.
542.         class Level2(Level):
543.             """
544.
545.             """
546.
547.             def __init__(self, win_scale, game_id, level_num, maze_id, pellets, power_pellets, scor
    e, tutorial_prompts):
548.                 super().__init__(win_scale, game_id, level_num, maze_id, pellets, power_pellets, sc
    ore, tutorial_prompts)
549.
550.             def run(self, win, events):
551.
552.                 super().run(win, events)
553.
554.                 # Tutorial Events
555.                 self.pellets = self.pellets[-2:]
556.                 if not self.start:
557.                     if self.tutorial_boxes[0].finished:
558.                         self.start = True
559.                         self.show_paths = True
560.                     else:
561.                         self.tutorial_boxes[0].update(events)
562.                         self.tutorial_boxes[0].display(win)
563.
564.                 if len(self.pellets) == 100 and not self.tutorial_boxes[1].finished:
565.                     self.paused = True
566.                     self.tutorial_boxes[1].update(events)
567.                     self.tutorial_boxes[1].display(win)
568.                     if self.tutorial_boxes[1].finished:
569.                         self.paused = False
570.
571.                 if len(self.pellets) == 0 and not self.tutorial_boxes[2].finished:
572.                     self.paused = True
573.                     self.tutorial_boxes[2].update(events)
574.                     self.tutorial_boxes[2].display(win)
575.                     if self.tutorial_boxes[2].finished:
576.                         self.paused = False
577.
578.
579.         class Level3(Level):
580.             """
581.
582.             """
583.
584.             def __init__(self, win_scale, game_id, level_num, maze_id, pellets, power_pellets, scor
    e, tutorial_prompts):
585.                 super().__init__(win_scale, game_id, level_num, maze_id, pellets, power_pellets, sc
    ore, tutorial_prompts)
586.
587.             def run(self, win, events):
588.                 super().run(win, events)
589.
590.                 # Tutorial Events
591.                 self.pellets = self.pellets[-2:]
592.                 if not self.start:
593.                     if self.tutorial_boxes[0].finished:
594.                         self.start = True
595.                         self.show_paths = True
596.                     else:
597.                         self.tutorial_boxes[0].update(events)
598.                         self.tutorial_boxes[0].display(win)
599.
600.                 if len(self.pellets) == 20 and not self.tutorial_boxes[1].finished:
601.                     self.paused = True
```

```
602.                     self.tutorial_boxes[1].update(events)
603.                     self.tutorial_boxes[1].display(win)
604.                     if self.tutorial_boxes[1].finished:
605.                         self.paused = False
606.
607.                 if len(self.pellets) == 0 and not self.tutorial_boxes[2].finished:
608.                     self.paused = True
609.                     self.tutorial_boxes[2].update(events)
610.                     self.tutorial_boxes[2].display(win)
611.                     if self.tutorial_boxes[2].finished:
612.                         self.paused = False
613.
614.
615.         class Level4(Level):
616.             """
617.
618.             """
619.
620.             def __init__(self, win_scale, game_id, level_num, maze_id, pellets, power_pellets, scor
    e, tutorial_prompts):
621.                 super().__init__(win_scale, game_id, level_num, maze_id, pellets, power_pellets, sc
    ore, tutorial_prompts)
622.
623.             def run(self, win, events):
624.                 super().run(win, events)
625.
626.                 # Tutorial Events
627.                 self.pellets = self.pellets[-30:]
628.                 if not self.start:
629.                     if self.tutorial_boxes[0].finished:
630.                         self.start = True
631.                         self.show_paths = True
632.                     else:
633.                         self.tutorial_boxes[0].update(events)
634.                         self.tutorial_boxes[0].display(win)
635.
636.                 if len(self.pellets) == 20 and not self.tutorial_boxes[1].finished:
637.                     self.paused = True
638.                     self.tutorial_boxes[1].update(events)
639.                     self.tutorial_boxes[1].display(win)
640.                     if self.tutorial_boxes[1].finished:
641.                         self.paused = False
642.
643.                 if len(self.pellets) == 0 and not self.tutorial_boxes[2].finished:
644.                     self.paused = True
645.                     self.tutorial_boxes[2].update(events)
646.                     self.tutorial_boxes[2].display(win)
647.                     if self.tutorial_boxes[2].finished:
648.                         self.paused = False
649.
650.
651.         class Level5(Level):
652.             """
653.
654.             """
655.
656.             def __init__(self, win_scale, game_id, level_num, maze_id, pellets, power_pellets, scor
    e, tutorial_prompts):
657.                 super().__init__(win_scale, game_id, level_num, maze_id, pellets, power_pellets, sc
    ore, tutorial_prompts)
658.
659.             def run(self, win, events):
660.                 super().run(win, events)
661.
662.
663.         class Level6(Level):
664.             """
665.
```

```
666.             """
667.
668.             def __init__(self, win_scale, game_id, level_num, maze_id, pellets, power_pellets, scor
    e, tutorial_prompts):
669.                 super().__init__(win_scale, game_id, level_num, maze_id, pellets, power_pellets, sc
    ore, tutorial_prompts)
670.
671.             def run(self, win, events):
672.                 super().run(win, events)
```

SINGLEPLAYER

```
1.  __author__ = 'Will Evans'
2.
3.  import os
4.  import pygame as pg
5.  from maze import *
6.  from sprites import *
7.  from time import sleep
8.  import local_database
9.  import threading
10. import local_settings
11. import text
12.
13.
14. class Classic:
15.     def __init__(self, win, win_scale, user_id):
16.         """
17.         Controls the running of each level and database queries.
18.         :param win: The current window, all objects must be blitted to this window to be displayed
    .
19.         :param win_scale: Window Scale (How large the window is - must be multiplied by all size r
    elated variables).
20.         :param user_id: UserID if the user has signed in (they don't have to be signed in to play
    Classic).
21.         """
22.
23.         # Essential
24.         self.win_scale = win_scale
25.         self.win = win
26.         self.program = 'Classic'
27.         self.user_id = user_id
28.         self.error_message = None
29.         self.game_finished = False
30.
31.         # Music
32.         intro_music_channel = pg.mixer.Channel(7)
33.         intro_music_channel.set_volume(0.5 * local_settings.get_setting('game_volume')/100)
34.         intro_music_channel.play(pg.mixer.Sound('Resources\\sounds\\intro_music.wav'))
35.
36.         # Highscore
37.         self.highscore = local_database.get_highscore()
38.         if self.highscore is None:
39.             self.highscore = 0
40.
41.         # Maze
42.         self.maze_id = 1
43.
44.         # Variables
45.         self.score = 0
46.         self.pellets_eaten = 0
47.         self.level_num = 1
48.         self.lives = 3
49.
50.         #   Database
51.         self.game_id = local_database.get_game_id(self.user_id, self.maze_id)
52.
```

```
        # Level Info
        self.level_num = 1
        self.extra_life_claimed = False

        # Level
        self.level = Level(
                            self.win_scale,
                            self.game_id, 1,
                            self.maze_id,
                            [],
                            [],
                            self.lives,
                            self.score,
                            self.highscore,
                            start_cap=4
                           )
        # Initials Input Box
        self.initials_input_box = text.TransparentInputBox(100, 30, 20, win_scale, 'Initials')
        self.word_1 = text.Word('Enter 3 initials', (168, 180), (255, 255, 30), 20, win_scale, centre=True)
        self.word_2 = text.Word('to save highscore', (168, 195), (255, 255, 30), 20, win_scale, centre=True)

    def run(self, win, events):
        """
        This method is run directly from the main script.
        :param win: The current window, all objects must be blitted to this window to be displayed.
        :param events: Contains events from the pg.event.get() call containing all keyboard events.
        :return: None
        """

        # Essential
        self.level.run(win, events)  # bool -: True if level won, false if pac-man dead and None if neither

        # Variables
        self.score = self.level.score
        self.pellets_eaten = self.level.pellets_eaten
        self.lives = self.level.lives

        self.extra_life_claimed = self.level.extra_life_claimed

        # Events
        for event in events:
            if event.type == pg.KEYDOWN:
                if event.key == pg.K_ESCAPE:
                    self.program = 'StartScreen'

        # If all the pellets have been eaten
        if self.level.finished and self.level.won:
                    self.level_num += 1
                    self.level.quit()
                    self.level = Level(
                                       self.win_scale,
                                       self.game_id,
                                       self.level_num,
                                       self.maze_id,
                                       [],
                                       [],
                                       self.lives,
                                       self.score,
                                       self.highscore,
                                       extra_life_claimed=self.extra_life_claimed
                                       )

                # If Pac-Man has died, but he still has lives
```

```
116.                elif self.level.finished and self.lives > 1:
117.                    self.lives -= 1
118.                    self.score = self.level.score
119.                    self.level.quit()
120.                    self.level = Level(
121.                                        self.win_scale,
122.                                        self.game_id,
123.                                        self.level_num,
124.                                        self.maze_id,
125.                                        self.level.pellets,
126.                                        self.level.power_pellets,
127.                                        self.lives,
128.                                        self.score,
129.                                        self.highscore,
130.                                        extra_life_claimed=self.extra_life_claimed
131.                                        )
132.
133.                elif self.lives == 0 and self.level.finished and not self.game_finished:
134.                    self.level.quit()
135.                    self.game_finished = True
136.
137.                if self.game_finished:
138.                    for event in events:
139.                        if event.type == pg.KEYDOWN:
140.                            if event.key == pg.K_RETURN:
141.                                local_database.save_initials(self.game_id, self.initials_input_box.
    get_input())
142.                                self.program = 'Highscores'
143.
144.                    self.initials_input_box.update(events)
145.                    self.initials_input_box.display(win)
146.                    self.word_1.display(win)
147.                    self.word_2.display(win)
148.
149.            def get_program(self):
150.                return self.program
151.
152.            def get_error(self):
153.                return self.error_message
154.
155.            def quit(self):
156.                pg.mixer.music.stop()
157.                self.level.quit()
158.
159.
160.        class Level:
161.            def __init__(self, win_scale, game_id, level_num, maze_id, pellets, power_pellets, live
    s, score, highscore,
162.                         start_cap=2, extra_life_claimed=False):
163.                """
164.                Responsible for running each level by calling sprite objects and handling their upd
    ates. The class
165.                also controls other things, such as score, displaying maze etc.
166.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
167.                :type win_scale: Integer.
168.                :param game_id: GameID given to the game by the database.
169.                :type game_id: Integer.
170.                :param level_num: This controls difficulty of the ghosts. The larger the level numb
    er, the more levels Pac-Man
171.                has completed and the harder the ghosts become.
172.                :type level_num: Integer.
173.                :param maze_id: ID given to each maze. This can be used by the database to return t
    he 2D list associated with
174.                the ID.
175.                :type maze_id: Integer.
176.                :param pellets: A list of pellets. This is if Pac-
    Man dies and the level starts over. The pellets must be the
```

```
177.                same as the previous level instance so they are saved by the classic class.
178.                :type pellets: List.
179.                :param power_pellets: A list of power pellets. This is if Pac-
    Man dies and the level starts over. The power
180.                pellets must be the same as the previous level instance so they are saved by the cl
    assic class.
181.                :type power_pellets: List.
182.                :param lives: Number of lives remaining.
183.                :type lives: Integer.
184.                :param score: Current score that should be displayed at the top.
185.                :type score: Integer.
186.                :param highscore: Highscore that should be displayed at the top.
187.                :type highscore: Integer.
188.                :param start_cap: How long the level should wait to start.
189.                :type start_cap: Integer.
190.                :param extra_life_claimed: Whether or not the extra life has been given to the play
    er.
191.                :type extra_life_claimed: Boolean.
192.                """
193.
194.                # Essential
195.                self._run = True
196.                self.win_scale = win_scale
197.                self.program = 'Classic'
198.                self.finished = False
199.                self.won = False
200.
201.                # Variables
202.                #   Level
203.                self.score = score
204.                self.highscore = highscore
205.
206.                self.game_maze = Maze(maze_id, win_scale)
207.
208.                self.lives = lives
209.
210.                self.start_cap = start_cap
211.
212.                self.extra_life_claimed = extra_life_claimed
213.
214.                #   Database
215.                self.level_num = level_num
216.                self.game_id = game_id
217.                self.level_score = 0
218.                self.length = 0
219.                self.pellets_eaten = 0
220.                self.power_pellets_eaten = 0
221.                self.ghosts_eaten = 0
222.
223.                # Length counting thread
224.                threading.Thread(target=self.second_count).start()
225.
226.                # Indicators
227.                import text
228.                self.one_up = text.Word('1UP', (6 * 12, 0.8 * 12), (234, 234, 234), 24, win_scale)

229.                self.score_position = (7 * 12, 2 * 12)
230.                self.score_indicator = text.Word('{}'.format(self.score), self.score_position, (234
    , 234, 234), 24, win_scale)
231.
232.                self.highscore_text_position = (19 * 12, 0.8 * 12)
233.                self.highscore_position = (17 * 12, 2 * 12)
234.
235.                self.highscore_text = text.Word('high score', self.highscore_text_position, (234, 2
    34, 234), 24, win_scale)
236.
237.                if self.score > self.highscore:
238.                    self.highscore_indicator = text.Word('{}'.format(self.score),
```

```
                                                    self.highscore_position,
                                                    (234, 234, 234),
                                                    24,
                                                    win_scale
                                                    )
            else:
                self.highscore_indicator = text.Word('{}'.format(self.highscore),
                                                    self.highscore_position,
                                                    (234, 234, 234),
                                                    24,
                                                    win_scale
                                                    )

            self.life_indicators = []
            skin = pg.transform.scale(pg.image.load('Resources\\sprites\\pac-man\\w_0.png'),
                                      (22 * win_scale, 22 * win_scale))

            for num in range(lives - 1):
                rect = skin.get_rect(center=((num * 24 * win_scale + 18 * win_scale), (35 * 12 * win_scale)))
                self.life_indicators.append(StaticSprite([skin], rect))

            # Start and ready text
            self.ready_text = text.Word('ready!', (17.5 * 12, 20.5 * 12), (255, 255, 30), 23, win_scale, italic=True)
            self.game_over_text = text.Word('game over', (18 * 12, 20.5 * 12), (255, 0, 0), 21, win_scale)

            # Points (displayed when Pac-Man eats a ghost).
            self.points_text = {}
            for text in os.listdir('Resources\\sprites\\{}'.format('points')):
                # noinspection PyUnresolvedReferences
                self.points_text.update({text: pg.transform.scale(
                    pg.image.load('Resources\\sprites\\{}\\{}'.format('points', text)),
                    ((24 * win_scale), (10 * win_scale)))})

            # Pac-Man
            self.pac_man = PacMan('pac-man', self.game_maze, win_scale)

            # Ghosts
            self.ghosts = []
            self.ghosts_copy = self.ghosts[::]
            blinky = Blinky('blinky', self.pac_man, self.game_maze, win_scale, level_num)
            self.ghosts.append(blinky)
            self.ghosts.append(Pinky('pinky', self.pac_man, self.game_maze, win_scale, level_num))
            self.ghosts.append(Clyde('clyde', self.pac_man, self.game_maze, win_scale, level_num))
            self.ghosts.append(Inky('inky', self.pac_man, self.game_maze, win_scale, level_num, blinky))

            # Pellets
            self.pellets = pellets
            self.power_pellets = power_pellets

            if pellets == [] and power_pellets == []:
                pellet_skin_path = os.path.join('Resources', 'sprites', 'pellets', 'pellet.png')
                pellet_skin = pg.transform.scale(pg.image.load(pellet_skin_path), (4 * win_scale, 4 * win_scale))

                power_pellet_skin_path = os.path.join('Resources', 'sprites', 'pellets', 'power_pellet.png')
                power_pellet_skin = pg.transform.scale(pg.image.load(power_pellet_skin_path),
                                                       (12 * win_scale, 12 * win_scale))

                pellet_sound_channel = pg.mixer.Channel(2)
```

```
297.                     pellet_sound_channel.set_volume(0.5 * (local_settings.get_setting('game_volume'
    ) / 100))
298.
299.                     power_pellet_death_sound_path = os.path.join('Resources', 'sounds', 'pellet', '
    death.wav')
300.                     pellet_death_sound = pg.mixer.Sound(power_pellet_death_sound_path)
301.
302.                     for row in self.game_maze.tiles:
303.                         for tile in row:
304.                             if tile.type == 'pellet':
305.                                 self.pellets.append(Pellet(pellet_skin,
306.                                                            tile,
307.                                                            self.pac_man,
308.                                                            win_scale,
309.                                                            pellet_death_sound,
310.                                                            pellet_sound_channel)
311.                                                    )
312.
313.                             elif tile.type == 'power_pellet':
314.                                 self.power_pellets.append(Pellet(power_pellet_skin,
315.                                                                  tile,
316.                                                                  self.pac_man,
317.                                                                  win_scale,
318.                                                                  pellet_death_sound,
319.                                                                  pellet_sound_channel,
320.                                                                  power_pellet=True)
321.                                                           )
322.                 else:
323.                     for pellet in self.pellets:
324.                         pellet.predator = self.pac_man
325.                     for power_pellet in self.power_pellets:
326.                         power_pellet.predator = self.pac_man
327.
328.                 # Sound
329.                 self.large_pellet_channel = pg.mixer.Channel(3)
330.                 self.large_pellet_channel.set_volume(0.5 * (local_settings.get_setting('game_volume
    ') / 100))
331.
332.                 large_pellet_sound_path = os.path.join('Resources', 'sounds', 'large_pellet_loop.wa
    v')
333.                 self.large_pellet_sound = pg.mixer.Sound(large_pellet_sound_path)
334.
335.                 self.death_sound_playing = False
336.                 self.started = False
337.
338.                 # Clocks and counts
339.                 self.start_clock = 0
340.                 self.one_up_clock = 0
341.                 self.flashing_map_clock = 0
342.                 self.flashing_map_count = 0
343.
344.             def run(self, win, events):
345.                 """
346.                 This method is run from the Classic class and updates and displays by one frame.
347.                 :param win: The current window, all objects must be blitted to this window to be di
    splayed.
348.                 :type win: Surface.
349.                 :param events: Contains events from the pg.event.get() call containing all keyboard
     events.
350.                 :type events: Tuple.
351.                 :return: Boolean returns True if level is won, False if level is lost, None otherwi
    se
352.                 """
353.
354.                 # Before the game starts (music)
355.                 if self.start_clock < self.start_cap:
356.                     self.start_clock += 1 / 60
357.                     self.one_up.display(win)
```

```
358.                     self.score_indicator.display(win)
359.                     self.highscore_text.display(win)
360.                     self.highscore_indicator.display(win)
361.                     self.game_maze.display(win)
362.
363.                     self.ready_text.display(win)
364.                     for pellet in self.pellets:
365.                         pellet.display(win)
366.                     for power_pellet in self.power_pellets:
367.                         power_pellet.display(win)
368.                     for life_indicator in self.life_indicators:
369.                         life_indicator.display(win)
370.                     for ghost in self.ghosts:
371.                         ghost.display(win)
372.                     self.pac_man.display(win)
373.
374.                 # If there are no pellets the maze will flash
375.                 elif len(self.pellets) == 0:
376.                     self.flashing_map_clock += 1/60
377.                     if self.flashing_map_clock > 0.25:
378.                         self.flashing_map_clock = 0
379.                         self.flashing_map_count += 1
380.                         self.game_maze.change_skin()
381.                     if self.flashing_map_count == 7:
382.                         self.finished = True
383.                         self.won = True
384.
385.                     self.game_maze.display(win)
386.
387.                     self.one_up.display(win)
388.                     self.score_indicator.display(win)
389.                     self.highscore_text.display(win)
390.                     self.highscore_indicator.display(win)
391.                     for life_indicator in self.life_indicators:
392.                         life_indicator.display(win)
393.                     self.pac_man.display(win)
394.
395.                 # Mainloop of the level
396.                 else:
397.                     # Score Indicators
398.                     #   Controls flashing of one up
399.                     self.one_up_clock += 1 / 60
400.                     if 0.4 > self.one_up_clock > 0.2:
401.                         self.one_up.display(win)
402.                     elif 0.4 < self.one_up_clock:
403.                         self.one_up_clock = 0
404.
405.                     self.score_indicator = text.Word('{}'.format(self.score),
406.                                                      self.score_position,
407.                                                      (234, 234, 234),
408.                                                      24,
409.                                                      self.win_scale
410.                                                      )
411.
412.                     self.score_indicator.display(win)
413.                     if self.score > self.highscore:
414.                         self.highscore_indicator = text.Word('{}'.format(self.score),
415.                                                              self.highscore_position,
416.                                                              (234, 234, 234),
417.                                                              24,
418.                                                              self.win_scale)
419.
420.                     if not self.extra_life_claimed:
421.                         if self.score > 10000:
422.                             self.lives += 1
423.                             skin = pg.transform.scale(pg.image.load('Resources\\sprites\\pac-
    man\\w_0.png'),
424.                                                       (22 * self.win_scale, 22 * self.win_scale))
```

```
                        for num in range(self.lives - 1):
                            rect = skin.get_rect(center=(
                                                         (num * 24 * self.win_scale + 18 * self.
    win_scale),
                                                         (35 * 12 * self.win_scale)
                                                          )
                                                 )

                            self.life_indicators.append(StaticSprite([skin], rect))
                        self.extra_life_claimed = True

                self.highscore_text.display(win)
                self.highscore_indicator.display(win)

                #   Life indicators
                for life_indicator in self.life_indicators:
                    life_indicator.display(win)

                # Maze
                self.game_maze.display(win)
                self.game_maze.display(win)

                # Pellets
                for pellet in self.pellets:
                    pellet.update()
                    pellet.display(win)
                    if pellet.eaten:
                        self.score += 10
                        self.level_score += 10
                        self.pellets_eaten += 1
                        self.pellets.remove(pellet)

                #  Power pellets
                for power_pellet in self.power_pellets:
                    power_pellet.update()
                    power_pellet.display(win)
                    if power_pellet.eaten:
                        self.score += 50
                        self.level_score += 50
                        self.power_pellets_eaten += 1
                        if not self.large_pellet_channel.get_busy():
                            self.large_pellet_channel.play(self.large_pellet_sound, loops=-1)
                        for ghost in self.ghosts:
                            ghost.scare()
                            self.ghosts_copy = [ghost for ghost in self.ghosts if not ghost.dea
    d]
                        self.power_pellets.remove(power_pellet)

                # Pac-Man
                self.pac_man.update(events)
                self.pac_man.display(win)

                # Checks whether the level has ended
                if self.pac_man.dead:
                    if self.death_sound_playing:
                        pass
                    else:
                        pg.mixer.stop()
                        self.death_sound_playing = True

                if self.pac_man.death_animation_finished and not self.finished:
                    self.finished = True
                    if self.lives == 1:
                        self.lives -= 1
                        self.game_over_text.display(win)
                        pg.display.update()
                        sleep(2)

```

```
                # Ghosts
                if all([not ghost.scared for ghost in self.ghosts]):
                    self.large_pellet_channel.stop()

                for ghost in self.ghosts_copy:

                    if ghost.dead:
                        self.ghosts_eaten += 1
                        self.ghosts_copy.remove(ghost)
                        for ghost_ in self.ghosts_copy:
                            ghost_.display(win)
                        points = 200 * 2 ** ((len(self.ghosts) - 1) - len(self.ghosts_copy))
                        self.score += points
                        self.level_score += points

                        win.blit(self.points_text['{}.png'.format(str(points))],
                                 (self.pac_man.x, self.pac_man.y - 8 * self.win_scale))
                        pg.display.update()
                        sleep(1)

                if not self.pac_man.dead:
                    for ghost in self.ghosts:
                        if ghost.__class__.__name__ == 'Blinky':
                            if len(self.pellets) < 20 + 2 * self.level_num:
                                ghost.make_elroy()
                            if len(self.pellets) < 10 + 2 * self.level_num:
                                ghost.elroy_upgrade()

                        ghost.update(events)
                        ghost.display(win)

            def quit(self):
                """
                Quits the level. Stops music playing and saves that level to the databse.
                :return: None
                """

                local_database.save_level(
                                            self.level_num,
                                            self.game_id,
                                            self.lives,
                                            self.level_score,
                                            self.length,
                                            self.pellets_eaten,
                                            self.power_pellets_eaten,
                                            self.ghosts_eaten
                                        )

                self._run = False
                pg.mixer.stop()

            def second_count(self):
                """
                Counts seconds, so that the time can be recorded in the database.
                :return: None
                """

                while self._run:
                    sleep(1)
                    self.length += 1
```

MULTIPLAYER

```
___author__ = 'Will Evans'

```

```
3.  import os
4.  import pygame as pg
5.  import networking
6.  import local_database
7.
8.  from sprites import *
9.  from multiplayer_sprites import *
10. from datastructures import Maze
11. from gui import *
12.
13.
14. class Multiplayer:
15.     def __init__(self, win, win_scale, user_id):
16.         """
17.         Menu for multiplayer (contains method for creating avatars that is used in sub menus). It
    controls the
18.         multiplayer menu screen. It simply prompts the user to choose either 'create game' or 'joi
    n game'. They will
19.         then be taken to the appropriate pages.
20.         :param win: The current window, all objects must be blitted to this window to be displayed
    .
21.         :param win_scale: Window Scale (How large the window is - must be multiplied by all size r
    elated variables).
22.         :param user_id: Unique to each user, required to play online.
23.         """
24.
25.         # Essential Information
26.         self.win = win
27.         self.win_scale = win_scale
28.         self.clock = pg.time.Clock()
29.         self.program = 'Multiplayer'
30.         self.error_message = None
31.         self.finished = False
32.         self.user_id = user_id
33.
34.         # Data
35.         self.players = None
36.
37.         # Player Boxes
38.         self.boxes, self.large_box = get_boxes(win_scale)
39.
40.         # Player Avatars
41.         #   Avatar Skins
42.         self.avatar_skins = get_avatar_skins()
43.         self.avatars, self.names, self.scores, self.ready, self.places = get_avatars(self.players,
44.                                                                                       self.finished
    ,
45.                                                                                       self.avatar_s
    kins,
46.                                                                                       self.boxes,
47.                                                                                       self.large_bo
    x,
48.                                                                                       win_scale
49.                                                                                       )
50.
51.         # Choices
52.         self.font_size = 32
53.         self.choices = []
54.         self.choices.append(LiveWord('Create Game', 350, self.font_size, win_scale))
55.
56.         self.choices.append(LiveWord('Join Game', 380, self.font_size, win_scale))
57.
58.         # Check user_id
59.         if user_id is None:
60.             self.program = 'StartScreen'
61.             self.error_message = 'Login Required'
62.         else:
```

```
63.             self.user_id = user_id
64.
65.     def run(self, win, events):
66.         """
67.         Main method, called from main loop.
68.         :param win: The current window, all objects must be blitted to this window to be displayed
    .
69.         :param events: Contains events from the pg.event.get() call containing all keyboard events
    .
70.         :return: None
71.         """
72.
73.         # Events
74.         for event in events:
75.             if event.type == pg.KEYDOWN:
76.                 if event.key == pg.K_ESCAPE:
77.                     self.program = 'StartScreen'
78.
79.         # Updating text reactions (i.e. highlighting)
80.         self.check_inputs(events)
81.         self.update_text()
82.
83.         # Displaying all objects
84.         for choice in self.choices:
85.             choice.display(win)
86.
87.         for box in self.boxes:
88.             box.display(win)
89.         self.large_box.display(win)
90.
91.         for avatar in self.avatars:
92.             avatar.display(win)
93.
94.         pg.display.update()
95.
96.     def check_inputs(self, events):
97.         """
98.         Checks whether any of the choices have been clicked.
99.         :param events: Contains events from the pg.event.get() call containing all keyboard events
    .
100.                :return: Target program of the word clicked or the current program.
101.                """
102.
103.                pos = get_mouse_input(events)
104.                if pos is not None:
105.                    for choice in self.choices:
106.                        if choice.check_click(*pos):
107.                            self.program = choice.get_program()
108.
109.            def update_text(self):
110.                """
111.                Updates the choices (highlights when mouse passes over).
112.                :return: None
113.                """
114.
115.                for choice in self.choices:
116.                    if choice.check_mouse(*pg.mouse.get_pos()):
117.                        choice.react()
118.
119.            def get_program(self):
120.                return self.program
121.
122.            def get_error(self):
123.                return self.error_message
124.
125.            def quit(self):
126.                pass
127.
```

```
128.
129.       class HostMenu:
130.           def __init__(self, win, win_scale, user_id):
131.               """
132.               Host menu screen. Pac-
    Man will be coloured in and the hosts local IP address will be displayed in the bottom.
133.               This can then be used by other players to connect to the host.
134.               :param win:
135.               :param win_scale:
136.               :param user_id:
137.               """
138.
139.               # Essential Information
140.               self.win = win
141.               self.win_scale = win_scale
142.               self.clock = pg.time.Clock()
143.               self.program = 'Create Game'
144.               self.error_message = None
145.               self.start_countdown = False
146.               self.finished = False
147.               self.match_started = False
148.               self.level = None
149.               self.winner_id = None
150.               self.user_id = user_id
151.
152.               # Host Data
153.               self.name = local_database.get_username(user_id)
154.
155.               self.client_id = 0
156.
157.               # Instantiate Server
158.               self.server = networking.Server(self.name)
159.               self.ip = self.server.get_ip()
160.
161.               self.players = self.server.get_players()
162.
163.               # Thread that handles connecting clients
164.               self.server.searching_for_clients = True
165.
166.               # Player Boxes
167.               self.boxes, self.large_box = get_boxes(win_scale)
168.
169.               # Player Avatars
170.               #   Avatar skins
171.               self.avatar_skins = get_avatar_skins()
172.               self.avatars, self.names, self.scores, self.ready_indicators, self.places = get_ava
    tars(self.players,
173.
        self.finished,
174.
        self.avatar_skins,
175.
        self.boxes,
176.
        self.large_box,
177.
        win_scale,
178.
        self.client_id
179.
        )
180.
181.               # Input Box
182.               self.game_id_box = InputBox(x=120,
183.                                           y=400,
184.                                           w=100,
185.                                           h=20,
186.                                           font_size=20,
```

```
                                        win_scale=win_scale,
                                        name=self.ip,
                                        interactive=False
                                        )

            # Start Button
            self.start_button = Button(content='Start',
                                       pos=(265, 400),
                                       dimensions=(60, 20),
                                       font_size=20,
                                       text_colour=(255, 255, 30),
                                       width=2,
                                       win_scale=win_scale
                                       )

            # Cancel Button
            self.cancel_button = Button(content='Cancel',
                                        pos=(250, 400),
                                        dimensions=(75, 20),
                                        font_size=20,
                                        text_colour=(255, 0, 0),
                                        width=2,
                                        win_scale=win_scale
                                        )

            # Number
            self.number = Word(None, (180, 400), (255, 255, 30), 48, win_scale)

            # Clock
            self.start_countdown_clock = 0

        def run(self, win, events):
            """
            This method is run directly from the main script. It updates and displays all of th
    e components on the screen.
            It also calls the server object to ensure all the data is up to date.
            :param win: The current window, all objects must be blitted to this window to be di
    splayed.
            :type win: Surface.
            :param events: Contains events from the pg.event.get() call containing all keyboard
     events.
            :type events: list.
            :return: None
            """

            # Events
            for event in events:
                if event.type == pg.KEYDOWN:
                    if event.key == pg.K_ESCAPE:
                        self.program = 'Multiplayer'
                        self.server.quit()

            if self.level is None:
                pg.mixer.stop()

                players = self.server.get_players()

                # Check Score
                for player_id, player_data in players.items():
                    if player_data['score'] > 20000:
                        self.finished = True
                if self.finished:
                    places = []
                    for id, data in players.items():
                        places.append({'id': id, 'score': data['score']})
                    places = sorted(places, key=lambda k: k['score'], reverse=True)
                    for place, data in enumerate(places):
                        self.server.update_data(data['id'], 'place', place)
```

```
252.
253.                     # Updating avatars as per how many people are connected
254.                     self.avatars, self.names, self.scores, self.ready_indicators, self.places = get
    _avatars(
255.
256.         players,
257.         self.finished,
258.         self.avatar_skins,
259.         self.boxes,
260.         self.large_box,
261.         self.win_scale,
262.         self.client_id
263.     )
264.
265.                     # Displaying boxes and avatars
266.                     for box in self.boxes:
267.                         box.display(win)
268.                     self.large_box.display(win)
269.
270.                     for avatar in self.avatars:
271.                         avatar.display(win)
272.
273.                     for name in self.names:
274.                         name.display(win)
275.
276.                     for score in self.scores:
277.                         score.display(win)
278.
279.                     for ready in self.ready_indicators:
280.                         ready.display(win)
281.
282.                     for place in self.places:
283.                         place.display(win)
284.
285.                     if not self.finished:
286.                         if self.start_countdown:
287.                             # Start button related objects / countdown
288.                             if not self.server.has_ai:
289.                                 self.server.add_ai()
290.                             self.start_countdown_clock += 1 / 60
291.                             self.number.content = str(5 - int(self.start_countdown_clock))
292.                             self.number.render()
293.
294.                             if self.start_countdown_clock > 3:
295.                                 if self.winner_id is not None:
296.                                     self.server.swap(self.winner_id)
297.
298.                             if self.start_countdown_clock > 4.8:
299.                                 self.start_countdown_clock = 0
300.                                 self.server.reset()
301.                                 self.server.send_data()
302.
303.                                 game_maze = Maze(1, self.win_scale)
304.
305.                                 score = [data['score'] for data in self.server.get_players().values
    () if data['skin'] == 'pac-man'][0]
306.
307.                                 self.level = HostLevel(self.win_scale, 5, game_maze, score, self.se
    rver)
308.                                 self.match_started = True
```

```
309.
310.                                 self.start_countdown = False
311.
312.                             self.number.display(win)
313.                             self.cancel_button.update(events)
314.                             self.start_countdown = not self.cancel_button.get_click()
315.                             self.cancel_button.display(win)
316.                         else:
317.                             self.start_countdown_clock = 0
318.                             self.number.content = None
319.
320.                             if self.server.has_ai and not self.match_started:
321.                                 self.server.remove_ai()
322.
323.                             self.start_button.update(events)
324.                             self.start_countdown = self.start_button.get_click()
325.                             self.start_button.display(win)
326.
327.                             if not self.match_started:
328.                                 self.game_id_box.update(events)
329.                                 self.game_id_box.display(win)
330.
331.                 else:
332.                     self.level.run(win, events)
333.                     if self.level.finished:
334.                         self.winner_id = self.level.winner_id
335.                         score = self.server.get_data(self.winner_id, 'score')
336.                         self.server.update_data(self.winner_id, 'score', score + 1600)
337.                         self.level = None
338.
339.                 self.server.update_data(0, 'countdown', self.number.content)
340.                 self.server.update_data(0, 'start', False if self.level is None else True)
341.                 self.server.update_data(0, 'finished', self.finished)
342.                 self.server.send_data()
343.
344.             def get_program(self):
345.                 return self.program
346.
347.             def get_error(self):
348.                 return self.error_message
349.
350.             def quit(self):
351.                 self.server.quit()
352.                 pg.mixer.stop()
353.
354.
355.         class ClientMenu:
356.             def __init__(self, win, win_scale, user_id):
357.                 """
358.                 This is very similar to the host menu in that there will be colour avatars when pla
    yers join the lobby, but
359.                 there will be a ghost in the centre instead of Pac-
    Man. They will also have the option to ready up instead of
360.                 starting the game.
361.                 :param win: The current window, all objects must be blitted to this window to be di
    splayed.
362.                 :type win: Surface.
363.                 :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
364.                 :type win_scale: Integer.
365.                 :param user_id: UserID if the user has signed in. This is used (at this stage) just
     to get the client's name
366.                 from the database.
367.                 """
368.
369.                 # Essentials
370.                 self.win = win
371.                 self.win_scale = win_scale
```

```
372.                self.user_id = user_id
373.                self.game_id = None
374.                self.game_id_buffer = None
375.                self.program = 'Join Game'
376.                self.error_message = None
377.                self.clock = pg.time.Clock()
378.                self.client = None
379.                self.connected = False
380.                self.start_level = False
381.                self.finished = False
382.                self.level = None
383.                self.winner_id = None
384.
385.                # Client Data
386.
387.                self.name = local_database.get_username(user_id)
388.                self.client_id = None
389.                self.players = None
390.                self.ready = False
391.
392.                # Player Boxes
393.                self.boxes, self.large_box = get_boxes(win_scale)
394.
395.                # Player Avatars
396.                #   Avatar skins
397.                self.avatar_skins = get_avatar_skins()
398.                self.avatars, self.names, self.scores, self.ready_indicators, self.places = get_ava
    tars(
399.
        self.players,
400.
        self.finished,
401.
        self.avatar_skins,
402.
        self.boxes,
403.
        self.large_box,
404.
        self.win_scale,
405.
        )
406.
407.                # Input Box
408.                self.input_box = InputBox(120, 380, 100, 20, 20, win_scale, 'Game ID')
409.
410.                # Ready Buttons
411.                self.ready_button = Button(content='Ready',
412.                                           pos=(265, 400),
413.                                           dimensions=(60, 20),
414.                                           font_size= 20,
415.                                           text_colour=(255, 255, 30),
416.                                           width=2,
417.                                           win_scale=win_scale
418.                                           )
419.
420.                self.unready_button = Button(content='Un ready',
421.                                             pos=(240, 400),
422.                                             dimensions=(85, 20),
423.                                             font_size=20,
424.                                             text_colour=(255, 255, 30),
425.                                             width=2,
426.                                             win_scale=win_scale
427.                                             )
428.
429.                # Countdown number
430.                self.number = Word(None, (180, 400), (255, 255, 30), 48, win_scale)
431.
```

```
432.           def run(self, win, events):
433.               """
434.               This method is run directly from the main script. It updates and displays all of th
    e components on the screen.
435.               It also calls the client object to ensure all the data is up to date.
436.               :param win: The current window, all objects must be blitted to this window to be di
    splayed.
437.               :type win: Surface.
438.               :param events: Contains events from the pg.event.get() call containing all keyboard
     events.
439.               :type events: list.
440.               :return: None
441.               """
442.
443.               # Events
444.               for event in events:
445.                   if event.type == pg.KEYDOWN:
446.                       if event.key == pg.K_ESCAPE:
447.                           try:
448.                               self.client.end()
449.                           except Exception as e:
450.                               print("Client escape {}".format(e))
451.                           self.program = 'Multiplayer'
452.
453.               # Avatars
454.               if self.level is None:
455.                   self.avatars, self.names, self.scores, self.ready_indicators, self.places = get
    _avatars(self.players,
456.
            self.finished,
457.
            self.avatar_skins,
458.
            self.boxes,
459.
            self.large_box,
460.
            self.win_scale,
461.
            self.client_id
462.
            )
463.
464.                   for box in self.boxes:
465.                       box.display(win)
466.                   self.large_box.display(win)
467.
468.                   for avatar in self.avatars:
469.                       avatar.display(win)
470.
471.                   for name in self.names:
472.                       name.display(win)
473.
474.                   for score in self.scores:
475.                       score.display(win)
476.
477.                   for ready in self.ready_indicators:
478.                       ready.display(win)
479.
480.                   for place in self.places:
481.                       place.display(win)
482.
483.               if not self.connected:
484.                   # Input Box
485.                   self.game_id_buffer = self.input_box.update(events)
486.                   if self.game_id_buffer != self.game_id:
487.                       self.game_id = self.game_id_buffer
488.
```

```
489.                         self.client = networking.Client(self.game_id, self.name)
490.                         self.connected = self.client.connected
491.
492.                         if self.connected:
493.                             self.client_id = self.client.get_client_id()
494.                         else:
495.                             self.error_message = 'Invalid IP'
496.                             self.program = 'Multiplayer'
497.                     self.input_box.display(win)
498.
499.                 else:
500.                     if not self.client.connected:
501.                         self.client.end()
502.                         self.program = 'Multiplayer'
503.                         self.error_message = 'Connection Lost'
504.
505.                     if not self.finished:
506.                         if not self.start_level:
507.                             self.level = None
508.
509.                             # Buttons
510.                             if self.ready:
511.                                 self.unready_button.update(events)
512.                                 self.ready = not self.unready_button.get_click()
513.                                 self.unready_button.display(win)
514.                             else:
515.                                 self.ready_button.update(events)
516.                                 self.ready = self.ready_button.get_click()
517.                                 self.ready_button.display(win)
518.
519.                             self.client.update_data('ready', self.ready)
520.                             self.client.send_player_data()
521.
522.                         self.number.render()
523.                         self.number.display(win)
524.                     else:
525.                         if self.level is None:
526.                             game_maze = Maze(1, self.win_scale)
527.                             score = self.client.get_data(self.client_id, 'score')
528.                             self.level = ClientLevel(self.win_scale, 5, game_maze, score, self.
    client)
529.
530.                         self.level.run(win, events)
531.                     else:
532.                         if not self.start_level:
533.                             self.level = None
534.
535.                     self.start_level = self.client.get_data(0, 'start')
536.                     self.number.content = self.client.get_data(0, 'countdown')
537.                     self.finished = self.client.get_data(0, 'finished')
538.                     self.players = self.client.get_players()
539.
540.             def get_program(self):
541.                 return self.program
542.
543.             def get_error(self):
544.                 return self.error_message
545.
546.             def quit(self):
547.                 try:
548.                     self.client.end()
549.                 except AttributeError:
550.                     pass
551.                 except Exception as e:
552.                     print(e)
553.
554.                 pg.mixer.stop()
555.
```

```
556.
557.       class ClientLevel:
558.           def __init__(self, win_scale, level_num, game_maze, score, client):
559.               """
560.               Responsible for running each level by calling sprite objects and handling their upd
    ates. The class
561.               also controls other things, such as score, displaying maze etc.
562.               :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
563.               :type win_scale: Integer.
564.               :param level_num: This controls difficulty of the ghosts. The larger the level numb
    er, the more levels Pac-Man
565.               has completed and the harder the ghosts become.
566.               :type level_num: Integer.
567.               :param game_maze: 2D list of maze.
568.               :type game_maze: 2D list.
569.               :param score: Current score that should be displayed at the top.
570.               :type score: Integer.
571.               :param client: Object used to send and receive data.
572.               :type client: Server.
573.               """
574.
575.               # Essential
576.               self.client = client
577.               self.client_id = self.client.get_client_id()
578.               self.players = self.client.get_players()
579.               self.score = score
580.               self.clock = pg.time.Clock()
581.               self.game_maze = game_maze
582.               self.win_scale = win_scale
583.               self.level_num = level_num
584.               self.program = 'Classic'
585.               self.error_message = None
586.               self.finished = False
587.               self.winner_id = None
588.
589.               # Music
590.               self.music_channel = pg.mixer.Channel(5)
591.               self.music_channel.set_volume(0.5 * (local_settings.get_setting('game_volume') / 10
    0))
592.               intro_music_path = os.path.join('Resources', 'sounds', 'intro_music.wav')
593.               self.music_channel.play(pg.mixer.Sound(intro_music_path))
594.
595.               # Indicators
596.               self.score_position = (7 * 12, 2 * 12)
597.               self.score_indicator = Word('{}'.format(self.score), self.score_position, (234, 234
    , 234), 24, win_scale)
598.
599.               self.ready_text = Word('ready!', (17.5 * 12, 20.5 * 12), (255, 255, 30), 23, win_sc
    ale, italic=True)
600.               self.game_over_text = Word('game over', (18 * 12, 20.5 * 12), (255, 0, 0), 21, win_
    scale)
601.
602.               self.points_text = {}
603.               for text in os.listdir('Resources\\sprites\\{}'.format('points')):
604.                   # noinspection PyUnresolvedReferences
605.                   self.points_text.update({text: pg.transform.scale(
606.                       pg.image.load('Resources\\sprites\\{}\\{}'.format('points', text)),
607.                       ((24 * win_scale), (10 * win_scale)))})
608.
609.               # Sprites
610.               self.pac_man, self.ghosts = self.get_players(self.players, self.game_maze, win_scal
    e, self.client)
611.               self.ghosts_copy = self.ghosts[::]
612.
613.               #   Pellets
614.               self.pellets = []
615.               self.power_pellets = []
```

```
616.                pellet_skin_path = os.path.join('Resources', 'sprites', 'pellets', 'pellet.png')
617.                pellet_skin = pg.transform.scale(pg.image.load(pellet_skin_path), (4 * win_scale, 4
     * win_scale))
618.                power_pellet_skin_path = os.path.join('Resources', 'sprites', 'pellets', 'power_pel
    let.png')
619.                power_pellet_skin = pg.transform.scale(pg.image.load(power_pellet_skin_path), (12 *
     win_scale, 12 * win_scale))
620.                pellet_sound_channel = pg.mixer.Channel(2)
621.                pellet_sound_channel.set_volume(0.5 * (local_settings.get_setting('game_volume') /
    100))
622.
623.                power_pellet_death_sound_path = os.path.join('Resources', 'sounds', 'pellet', 'deat
    h.wav')
624.                pellet_death_sound = pg.mixer.Sound(power_pellet_death_sound_path)
625.
626.                for row in self.game_maze.tiles:
627.                    for tile in row:
628.                        if tile.type == 'pellet':
629.                            self.pellets.append(
630.                                                Pellet(pellet_skin,
631.                                                       tile,
632.                                                       self.pac_man,
633.                                                       win_scale,
634.                                                       pellet_death_sound,
635.                                                       pellet_sound_channel)
636.                                               )
637.
638.                        elif tile.type == 'power_pellet':
639.                            self.power_pellets.append(
640.                                                Pellet(power_pellet_skin,
641.                                                       tile,
642.                                                       self.pac_man,
643.                                                       win_scale,
644.                                                       pellet_death_sound,
645.                                                       pellet_sound_channel,
646.                                                       power_pellet=True)
647.                                                )
648.
649.                self.large_pellet_channel = pg.mixer.Channel(3)
650.                self.large_pellet_channel.set_volume(0.5 * (local_settings.get_setting('game_volume
    ') / 100))
651.                self.large_pellet_sound_path = os.path.join('Resources', 'sounds', 'large_pellet_lo
    op.wav')
652.                self.large_pellet_sound = pg.mixer.Sound(self.large_pellet_sound_path)
653.
654.                # Misc
655.                self.death_sound_playing = False
656.                self.started = False
657.
658.                # Clocks and counts
659.                self.flashing_map_clock = 0
660.                self.flashing_map_count = 0
661.
662.            def run(self, win, events):
663.                """
664.                This method is run from the Client menu class (as it needs to be able to transfer d
    ata form level object to
665.                level object which can't be done when running directly from main. It controls the u
    pdates and displaying of all
666.                objects.
667.                :param win: The current window, all objects must be blitted to this window to be di
    splayed.
668.                :type win: Surface.
669.                :param events: Contains events from the pg.event.get() call containing all keyboard
     events.
670.                :type events: Tuple.
671.                :return: None
672.                """
```

```
673.
674.                # Intro Music
675.                if self.music_channel.get_busy():
676.
677.                    # Display (only)
678.                    self.game_maze.display(win)
679.
680.                    for pellet in self.pellets:
681.                        pellet.display(win)
682.                    for power_pellet in self.power_pellets:
683.                        power_pellet.display(win)
684.
685.                    self.pac_man.display(win)
686.
687.                    for ghost in self.ghosts:
688.                        ghost.display(win)
689.
690.                    self.score_indicator.display(win)
691.
692.                    self.ready_text.display(win)
693.
694.                # Check if Pac-Man has won
695.                elif len(self.pellets) == 0:
696.                    # Causes map to flash
697.                    self.flashing_map_clock += 1 / 60
698.
699.                    if self.flashing_map_clock > 0.25:
700.                        self.flashing_map_clock = 0
701.                        self.flashing_map_count += 1
702.                        self.game_maze.change_skin()
703.
704.                    # If map has finished flashing
705.                    if self.flashing_map_count == 7:
706.                        self.finished = True
707.                        self.winner_id = self.pac_man.client_id
708.
709.                    # Display (only and selective)
710.                    self.game_maze.display(win)
711.                    self.score_indicator.display(win)
712.                    self.pac_man.display(win)
713.
714.                # Mainloop
715.                else:
716.                    self.game_maze.display(win)
717.                    # Updates and display
718.                    for pellet in self.pellets:
719.                        pellet.update()
720.                        pellet.display(win)
721.                        if pellet.eaten:
722.                            self.score += 10
723.                            self.pellets.remove(pellet)
724.
725.                    for power_pellet in self.power_pellets:
726.                        power_pellet.update()
727.                        power_pellet.display(win)
728.
729.                        # When power pellet eaten
730.                        if power_pellet.eaten:
731.                            self.score += 50
732.
733.                            # Play sound on loop
734.                            if not self.large_pellet_channel.get_busy():
735.                                self.large_pellet_channel.play(self.large_pellet_sound, loops=-1)
736.
737.                            # Scare ghosts
738.                            for ghost in self.ghosts:
739.                                ghost.scare()
```

```
740.                             self.ghosts_copy = [ghost for ghost in self.ghosts if not ghost.dea
    d]
741.                         self.power_pellets.remove(power_pellet)
742.
743.                 # If all ghosts are not scared (i.e all pac_man_dead or scared timer ended) sto
    p playing sound
744.                 if all([not ghost.scared for ghost in self.ghosts]):
745.                     self.large_pellet_channel.stop()
746.
747.                 # Calculate how many points each ghost should give when eaten
748.                 for ghost in self.ghosts_copy:
749.                     if ghost.dead:
750.                         self.ghosts_copy.remove(ghost)
751.                         for ghost_ in self.ghosts_copy:
752.                             ghost_.display(win)
753.                         points = 200 * 2 ** (3 - len(self.ghosts_copy))
754.                         self.score += points
755.
756.                         win.blit(self.points_text['{}.png'.format(str(points))],
757.                                  (self.pac_man.x, self.pac_man.y - 8 * self.win_scale))
758.                         pg.display.update()
759.                         sleep(1)
760.
761.                 # Update and display Pac-Man
762.                 self.pac_man.display(win)
763.                 self.pac_man.update(events)
764.
765.                 if self.pac_man.dead:
766.                     if self.death_sound_playing:
767.                         pass
768.                     else:
769.                         pg.mixer.stop()
770.                         self.death_sound_playing = True
771.
772.                 # Dead is None when death-animation has finished
773.                 if self.pac_man.death_animation_finished:
774.                     self.game_over_text.display(win)
775.                     client_id = [ghost.client_id for ghost in self.ghosts if ghost.won][0]
776.                     pg.display.update()
777.                     sleep(2)
778.                     self.finished = True
779.                     self.winner_id = client_id
780.
781.                 # If Pac-Man is alive
782.                 if not self.pac_man.dead:
783.
784.                     # Update and display ghosts
785.                     for ghost in self.ghosts:
786.                         ghost.update(events)
787.                         ghost.display(win)
788.
789.                 self.score_indicator = Word('{}'.format(self.client.get_data(self.client.get_cl
    ient_id(), 'score')),
790.                                             self.score_position, (234, 234, 234), 24, self.win_
    scale)
791.                 self.score_indicator.display(win)
792.
793.             return True, None
794.
795.         def get_players(self, players, game_maze, win_scale, client):
796.             """
797.             This takes the list of players and assigns each of them the appropriate multiplayer
     sprite based on whether they
798.             are Pac-Man or a ghost and based on whether they are a client or server.
799.             :param players: Dictionary of players and their various attributes.
800.             :type players: 2D dictionary.
801.             :param game_maze: 2D list of maze.
802.             :type game_maze: 2D list.
```

```
803.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
    size related variables).
804.                :type win_scale: Integer.
805.                :param client: Object that is passed through to all the sprites in order for them t
    o send and receive player
806.                data.
807.                :type client: Server.
808.                :return: Pac-Man object and list of ghost objects.
809.                """
810.
811.                # Sprites
812.                ghosts = []
813.
814.                # Pac-Man
815.                for client_id, player in players.items():
816.                    skin = player['skin']
817.                    if skin is None:
818.                        continue
819.                    # Pac-Man
820.                    if skin == 'pac-man':
821.                        if client_id == self.client_id:
822.                            pac_man = ClientPlayerPacMan(skin, game_maze, win_scale, client, client
    _id)
823.                        else:
824.                            pac_man = ClientPacMan(skin, game_maze, win_scale, client, client_id)
825.
826.                # Blinky
827.                for client_id, player in players.items():
828.                    skin = player['skin']
829.
830.                    if skin == 'blinky':
831.                        continue
832.
833.                    if skin == 'blinky':
834.                        if client_id == self.client_id:
835.                            blinky = ClientPlayerGhost(skin,
836.                                                       player['pos'],
837.                                                       pac_man,
838.                                                       game_maze,
839.                                                       win_scale,
840.                                                       self.level_num,
841.                                                       client,
842.                                                       client_id
843.                                                       )
844.
845.                        elif skin == 'AI':
846.                            blinky = ClientBlinky(skin,
847.                                                  pac_man,
848.                                                  game_maze,
849.                                                  win_scale,
850.                                                  self.level_num,
851.                                                  client,
852.                                                  client_id
853.                                                  )
854.                        else:
855.                            blinky = ClientGhost(skin,
856.                                                 player['pos'],
857.                                                 pac_man,
858.                                                 game_maze,
859.                                                 win_scale,
860.                                                 self.level_num,
861.                                                 client,
862.                                                 client_id
863.                                                 )
864.                        ghosts.append(blinky)
865.
866.                # Ghosts
867.                for client_id, player in players.items():
```

```
                    skin = player['skin']
                    if skin != 'pac-man':
                        # Ghosts
                        if client_id == self.client_id:
                            ghosts.insert(0,
                                          ClientPlayerGhost(skin,
                                                            player['pos'],
                                                            pac_man,
                                                            game_maze,
                                                            win_scale,
                                                            self.level_num,
                                                            client,
                                                            client_id)
                                          )

                        else:
                            ghosts.append(
                                ClientGhost(skin,
                                            player['pos'],
                                            pac_man,
                                            game_maze,
                                            win_scale,
                                            self.level_num,
                                            client,
                                            client_id
                                            )
                            )

                return pac_man, ghosts


        class HostLevel(ClientLevel):
            def __init__(self, win_scale, level_num, game_maze, score, server):
                """
                Responsible for running each level by calling sprite objects and handling their updates. The class also
                controls other things, such as score, displaying maze etc.
                :param win_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
                :type win_scale: Integer.
                :param level_num: This controls difficulty of the ghosts. The larger the level number, the more levels Pac-Man
                has completed and the harder the ghosts become.
                :type level_num: Integer.
                :param game_maze: 2D list of maze.
                :type game_maze: 2D list.
                :param score: Current score that should be displayed at the top.
                :type score: Integer.
                :param server: Object used to send and receive data.
                :type server: Server.
                """

                super().__init__(win_scale, level_num, game_maze, score, server)

                # Clock
                self.score_update_clock = 0

            def get_players(self, players, game_maze, win_scale, server):
                """
                This takes the list of players and assigns each of them the appropriate multiplayer sprite based on whether they
                are Pac-Man or a ghost.
                :param players: Dictionary of players and their various attributes.
                :type players: 2D dictionary.
                :param game_maze: 2D list of maze.
                :type game_maze: 2D list.
                :param win_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
```

```
931.                :type win_scale: Integer.
932.                :param server: Object that is passed through to all the sprites in order for them t
    o send and receive player
933.                data.
934.                :type server: Server.
935.                :return: Pac-Man object and list of ghost objects.
936.                """
937.
938.                ghost_sprites = {
939.
940.                    'pinky': ServerPinky,
941.                    'clyde': ServerClyde
942.
943.                }
944.                # Sprites
945.                ghosts = []
946.
947.                # Pac-Man
948.                for client_id, player in players.items():
949.                    skin = player['skin']
950.                    if skin is None:
951.                        continue
952.                    # Pac-Man
953.                    if skin == 'pac-man':
954.                        if client_id is 0:
955.                            pac_man = ServerPlayerPacMan(skin, game_maze, win_scale, server)
956.                        elif player['name'] == 'AI':
957.                            pac_man = ServerPacManAI(skin, game_maze, win_scale, server, client_id)

958.                        else:
959.                            pac_man = ServerPacMan(skin, game_maze, win_scale, server, client_id)
960.
961.                # Blinky
962.                for client_id, player in players.items():
963.                    skin = player['skin']
964.                    if skin == 'blinky':
965.                        if client_id is 0:
966.                            blinky = ServerPlayerGhost(skin, player['pos'], pac_man, game_maze, win
    _scale, self.level_num, server)
967.                        elif player['name'] == 'AI':
968.                            blinky = ServerBlinky(skin, pac_man, game_maze, win_scale, self.level_n
    um, server, client_id)
969.                        else:
970.                            blinky = ServerGhost(skin, player['pos'], pac_man, game_maze, win_scale
    , self.level_num, server,
971.                                                 client_id)
972.                        ghosts.append(blinky)
973.
974.                # Ghosts
975.                for client_id, player in players.items():
976.                    skin = player['skin']
977.
978.                    if skin == 'blinky':
979.                        continue
980.
981.                    if player['name'] != 'AI' and skin != 'pac-man':
982.                        # Ghosts
983.                        if client_id is 0:
984.                            ghosts.append(
985.                                ServerPlayerGhost(skin,
986.                                                  player['pos'],
987.                                                  pac_man,
988.                                                  game_maze,
989.                                                  win_scale,
990.                                                  self.level_num,
991.                                                  server)
992.                                          )
993.                        else:
```

```
                            ghosts.append(
                                ServerGhost(skin,
                                            player['pos'],
                                            pac_man,
                                            game_maze,
                                            win_scale,
                                            self.level_num,
                                            server,
                                            client_id)
                                        )

                    elif player['name'] == 'AI' and skin != 'pac-man':
                        if skin == 'inky':
                            ghosts.append(ServerInky(player['skin'],
                                                     pac_man,
                                                     game_maze,
                                                     win_scale,
                                                     self.level_num,
                                                     blinky,
                                                     server,
                                                     client_id)
                                          )
                        else:
                            ghosts.append(ghost_sprites[skin](player['skin'],
                                                              pac_man,
                                                              game_maze,
                                                              win_scale,
                                                              self.level_num,
                                                              server,
                                                              client_id)
                                          )

                return pac_man, ghosts

            def run(self, win, events):
                """
                Does the same as client level except it also keeps track of each players score.
                :param win: The current window, all objects must be blitted to this window to be di
    splayed.
                :type win: Surface.
                :param events: Contains events from the pg.event.get() call containing all keyboard
     events.
                :type events: Tuple.
                :return: None
                """

                # Scores
                self.pac_man.update_score(self.score)
                if self.score_update_clock > 1:
                    self.score_update_clock = 0
                    for ghost in self.ghosts:
                        points = get_distance_points(ghost, self.pac_man)
                        ghost.add_points(points)
                else:
                    self.score_update_clock += 1 / 60
                super().run(win, events)


        def get_avatar_skins():
            """
            Gets skins for each avatar from resources folder. Chooses between grey and coloured dep
    ending on whether the player is
            connected.
            :return: Dictionary of skins.
            """

            avatar_skins = {}

```

```
1059.           # Adding grey avatars
1060.           for skin in os.listdir('Resources\\sprites\\{}'.format('grey_avatars')):
1061.               avatar_skins.update(
1062.                   {skin[:-
    4]: pg.image.load('Resources\\sprites\\{}\\{}'.format('grey_avatars', skin))})
1063.
1064.           # Adding coloured skins for when a connection to a client is made
1065.           for skin in os.listdir('Resources\\sprites\\{}'.format('coloured_avatars')):
1066.               avatar_skins.update(
1067.                   {skin[:-
    4]: pg.image.load('Resources\\sprites\\{}\\{}'.format('coloured_avatars', skin))})
1068.
1069.           return avatar_skins
1070.
1071.
1072.       def get_boxes(win_scale):
1073.           """
1074.           Returns box objects for the menu screens. These are just the 4 empty boxes across the t
    op of the screen and
1075.           the large central box that are displayed before there are any sprites in them.
1076.           :param win_scale:
1077.           :return: Small boxes and the large box.
1078.           """
1079.
1080.           boxes = []
1081.           y = 30
1082.           w = 74
1083.           h = 74
1084.           x_spacing = 8
1085.           line_width = 3
1086.           colour = (200, 200, 200)
1087.           for num in range(4):
1088.               x = x_spacing + (x_spacing + w) * num
1089.               boxes.append(Box((x, y), (w, h), colour, line_width, win_scale))
1090.
1091.           large_box_rect = ((95, 170), (150, 150))
1092.           large_box = Box(*large_box_rect, colour, line_width, win_scale)
1093.
1094.           return boxes, large_box
1095.
1096.
1097.       def get_avatars(players, finished, avatar_skins, boxes, large_box, win_scale, client_id=Non
    e):
1098.           """
1099.           This class returns all the graphical attributes of each player within the many screen (
    also known as the
1100.           avatars). It will add sprites, names, the current score, ready indicators and places (w
    hen the game has
1101.           finished) into lists that can then be stored in the menu classes below.
1102.           :param players: List of players and all attributes, used to generate all the objects.
1103.           :param finished: Boolean, whether or not the game has finished (whether to display the
    places).
1104.           :param avatar_skins:
1105.           :param boxes:
1106.           :param large_box:
1107.           :param win_scale:
1108.           :param client_id:
1109.           :return: Lists: avatars, names, scores, ready_indicators and places.
1110.           """
1111.           avatars = []
1112.           names = []
1113.           scores = []
1114.           ready_indicators = []
1115.           places = []
1116.
1117.           colours = {
1118.
1119.               'pac-man': (255, 255, 30),
```

```
1120.             'blinky': (222, 0, 0),
1121.             'pinky': (255, 181, 255),
1122.             'inky': (0, 222, 222),
1123.             'clyde': (255, 181, 33),
1124.
1125.         }
1126.
1127.         place_details = {
1128.             0:
1129.                 {
1130.                     'string': '1st',
1131.                     'colour': (255, 215, 0)
1132.                 },
1133.
1134.             1:
1135.                 {
1136.                     'string': '2nd',
1137.                     'colour': (220, 220, 220)
1138.                 },
1139.
1140.             2:
1141.                 {
1142.                     'string': '3rd',
1143.                     'colour': (205, 127, 50)
1144.                 },
1145.
1146.             3:
1147.                 {
1148.                     'string': '4th',
1149.                     'colour': (169, 169, 169)
1150.                 },
1151.
1152.             4:
1153.                 {
1154.                     'string': '5th',
1155.                     'colour': (169, 169, 169)
1156.                 }
1157.
1158.         }
1159.
1160.         large_box_boolean = False
1161.         if players is not None:
1162.             for num, box in enumerate(boxes):
1163.                 if num == client_id or (num == 3 and not large_box_boolean):
1164.
1165.                     x, y = large_box.rect.center
1166.                     x /= win_scale
1167.                     y /= win_scale
1168.                     colour = colours[players[client_id]['skin']]
1169.
1170.                     # Large Avatar
1171.                     skin = pg.transform.scale(avatar_skins[players[client_id]['skin']],
1172.                                               (132 * win_scale, 132 * win_scale))
1173.                     skin_rect = skin.get_rect(center=large_box.rect.center)
1174.                     avatars.append(StaticSprite([skin], skin_rect))
1175.
1176.                     # Large Name
1177.                     name = players[client_id]['name']
1178.                     names.append(Word(name, (x, y + 95), colour, 26, win_scale, centre=True))
1179.
1180.                     # Score
1181.                     if players[num]['score'] is not None:
1182.                         score = str(players[client_id]['score'])
1183.
1184.                         scores.append(Word(content=score, pos=(x, y + 115), colour=colour,
1185.                                            font_size=26, win_scale=win_scale, italic=True, cent
    re=True))
1186.
```

```
1187.                     # Large Ready
1188.                     if players[client_id]['ready'] and not finished:
1189.                         ready_indicators.append(
1190.                             Word(content='Ready!', pos=(x, y + 132), colour=(255, 255, 30),
1191.                                  font_size=26, win_scale=win_scale, italic=True, centre=True))
1192.
1193.                     # Place
1194.                     if finished:
1195.                         places.append(Word(content=place_details[players[client_id]['place']]['
    string'],
1196.                                            pos=(x, y + 132),
1197.                                            colour=place_details[players[client_id]['place']]['c
    olour'],
1198.                                            font_size=26,
1199.                                            win_scale=win_scale,
1200.                                            italic=True,
1201.                                            centre=True)
1202.                                       )
1203.
1204.                         large_box_boolean = True
1205.
1206.                 if large_box_boolean:
1207.                     num += 1
1208.
1209.                 # Avatar
1210.                 if players[num]['name'] is None:
1211.                     skin = pg.transform.scale(avatar_skins['grey_{}'.format(players[client_id][
    'skin'])],
1212.                                               (66 * win_scale, 66 * win_scale))
1213.
1214.                 else:
1215.                     skin = pg.transform.scale(avatar_skins[players[num]['skin']],
1216.                                               (66 * win_scale, 66 * win_scale))
1217.
1218.                     x, y = box.rect.center
1219.                     x /= win_scale
1220.                     y /= win_scale
1221.
1222.                     colour = colours[players[num]['skin']]
1223.
1224.                     # Name
1225.                     name = players[num]['name']
1226.                     names.append(Word(name, (x, y + 50), colour, 16, win_scale, centre=True))
1227.
1228.                     # Score
1229.                     if players[num]['score'] is not None:
1230.                         score = str(players[num]['score'])
1231.                         scores.append(
1232.                             Word(score, (x, y + 62), colour, 16, win_scale, centre=True))
1233.
1234.                     # Ready
1235.                     if players[num]['ready'] and not finished:
1236.                         ready_indicators.append(
1237.                             Word(content='Ready!', pos=(x, y + 75), colour=(255, 255, 30),
1238.                                  font_size=16, win_scale=win_scale, italic=True, centre=True))
1239.
1240.                     # Place
1241.                     if finished:
1242.                         places.append(Word(content=place_details[players[num]['place']]['string
    '],
1243.                                            pos=(x, y + 75),
1244.                                            colour=place_details[players[num]['place']]['colour'
    ],
1245.                                            font_size=16,
1246.                                            win_scale=win_scale,
1247.                                            centre=True)
```

```
                                        )

                rect = skin.get_rect(center=box.rect.center)
                avatars.append(StaticSprite([skin], rect))

        else:
            skin = pg.transform.scale(avatar_skins['grey_{}'.format('blinky')],
                                      (66 * win_scale, 66 * win_scale))
            for box in boxes:
                rect = skin.get_rect(center=box.rect.center)
                avatars.append(StaticSprite([skin], rect))

            skin = pg.transform.scale(avatar_skins['grey_pac-man'],
                                      (132 * win_scale, 132 * win_scale))
            rect = skin.get_rect(center=large_box.rect.center)
            avatars.append(StaticSprite([skin], rect))

        return avatars, names, scores, ready_indicators, places


    def get_distance_points(ghost, pac_man):
        """
        Works out the euclidean distance between the ghost and Pac-
   Man objects and returns an amount of points to give the
        ghost based on how far it is.
        :param ghost: Ghost object.
        :param pac_man: Pac-Man object.
        :return: Points.
        """

        # Gets manhattan distance from pac_man to ghost
        distance = abs(ghost.tile.x - pac_man.tile.x) + abs(ghost.tile.y - pac_man.tile.y)

        # Subtracts distance from 13 so the closer to pac_man the more points you get
        points = (13 - distance) * 4

        # Ensures points are not negative
        return max(0, points)


    def get_mouse_input(events):
        for event in events:
            if event.type == pg.MOUSEBUTTONDOWN:
                return event.pos
```

SPRITES

```
__author__ = 'Will Evans'

import os
from pathfinding import Manhattan as Search
import pygame as pg
import random
import local_settings
from threading import Thread
from time import sleep


class Sprite:
    def __init__(self, resource_pack, position, maze, win_scale):
        """
        Template for sub-classes: 'Pac-Man' and 'Ghost'.
        :param resource_pack: Contains the path to the folder containing the skins for the sprite.

        :param position: x, y co-ords for the position of the sprite on the screen.
```

```
        :param maze: Two dimensional list representation of the maze.
        :param win_scale: Window Scale (How large the window is - must be multiplied by all size r
    elated variables).
        """

        # Essential
        self.win_scale = win_scale
        self.maze = maze

        #
        self.dead = False
        self.visible = False

        # Position
        x, y = position
        self.x = x * win_scale
        self.y = y * win_scale

        # Speed
        self._speed = 1.5
        self.speed_count = 0

        self.skin = '0'
        self.facing = 'e'
        self.move = 'e'
        self.online_move = 'e'
        self.buffer_move = 'e'

        # Skins
        self.skin = '0'
        self.skin_cap = 4

        #   Normal Skins
        self.normal_skins = {}
        for skin in os.listdir('resources\\sprites\\{}'.format(resource_pack)):
            # noinspection PyUnresolvedReferences
            self.normal_skins.update({skin: pg.transform.scale(
                pg.image.load('resources\\sprites\\{}\\{}'.format(resource_pack, skin)),
                ((22 * win_scale), (22 * win_scale)))})

        self.skins = self.normal_skins

        # Tiles
        tilex = int(self.x / (12 * self.win_scale))
        tiley = int(self.y / (12 * self.win_scale))

        self.tile = self.maze.tiles[tiley - 3][tilex]
        self.previous_tile = self.tile

        # Rects
        self.skin_rect = self.skins['e_0.png'].get_rect(center=(self.x, self.y))
        self.rect = pg.Rect(self.x - int(8 * win_scale),
                            self.y - int(8 * win_scale),
                            17 * win_scale,
                            17 * win_scale)

        # Movement booleans
        self.get_new_move = True
        self.wall_defence_delay = True
        self.return_None = False

        # Sound
        self.sound_channel = pg.mixer.Channel(0)
        self.sound_channel.set_volume(0.5 * local_settings.get_setting('game_volume') / 100)

        # Clocks
        self.skin_clock = 0
        self.stop_clock = 0
```

```
85.         self.death_animation_clock = 0
86.
87.     def update(self, move):
88.         """
89.         Contains all the calls needed for every type of Sprite in an update.
90.         :param move: Sprite's checked move.
91.         :return: None
92.         """
93.
94.         self.correct_pos()
95.         self.correct_tunnel()
96.         self.facing, self.skin = self.get_skin(move)
97.         self.update_tile()
98.         self.update_pos(move)
99.
100.            def get_input(self, events):
101.                """
102.                Default input method for the object using keyboard events (arrow keys).
103.                :param events: Contains events from the pg.event.get() call containing all keyboard
    events.
104.                :return: Returns a move ('n', 'e', 's', 'w').
105.                """
106.
107.                move = self.move
108.                for event in events:
109.                    if event.type == pg.KEYDOWN:
110.                        if event.key == pg.K_UP:
111.                            move = 'n'
112.                        elif event.key == pg.K_RIGHT:
113.                            move = 'e'
114.                        elif event.key == pg.K_DOWN:
115.                            move = 's'
116.                        elif event.key == pg.K_LEFT:
117.                            move = 'w'
118.
119.                return move
120.
121.            def set_speed(self, speed):
122.                """
123.                Sets private attribute speed.
124.                :param speed: Speed value that will be stored in the private attribute speed.
125.                :return: None.
126.                """
127.
128.                self._speed = speed
129.
130.            def get_pos(self):
131.                """
132.                Returns position of the sprite.
133.                :return: Position of sprite (x,y).
134.                """
135.
136.                return self.x, self.y
137.
138.            def kill(self):
139.                """
140.                Kills the sprite by setting the dead attribute to True.
141.                :return: None.
142.                """
143.
144.                self.dead = True
145.
146.            def get_skin(self, move):
147.                """
148.                Returns the skin reference (direction and number) based on how and if the sprite is
    moving.
149.                :param move: Sprite's move.
150.                :return: Skin reference (direction, number).
```

```
151.                 """
152.
153.                 if move is None:  # Stops the sprite changing skin number when it is not moving.
154.                     move = self.facing
155.                 if self.skin_clock == self.skin_cap:  # Allows the amount of frames between every s
    kin change to be changed.
156.                     num = abs(int(self.skin) - 1)  # If self.skin is 1 num will become 0. If it is
    0 it will become 1.
157.                     self.skin_clock = 0
158.                 else:
159.                     num = self.skin
160.                 return move, str(num)
161.
162.             def get_move(self, events):
163.                 """
164.                 Gets move from input then checks to see if that move is valid.
165.                 :param events: Contains events from the pg.event.get() call containing all keyboard
     events.
166.                 :return: Output from self.check_move(move).
167.                 """
168.
169.                 move = self.get_input(events)
170.                 return self.check_move(move)
171.
172.             def check_move(self, move):
173.                 """
174.                 Performs checks on the move argument and returns a valid move.
175.                 :param move: Move that the sprite wants to use.
176.                 :return: A valid move (usually the user input unless it was invalid).
177.                 """
178.
179.                 self.move = move  # Saves the move
180.
181.                 # Sets return_None to false when Pac-man hits a wall so that Pac-
    Man can change direction once he has hit a
182.                 # wall.
183.                 if self.get_next_tile(self.validate_move(self.move)).type != 'wall':
184.                     self.return_None = False
185.
186.                 if self.return_None:
187.                     return None
188.
189.                 # get_new_move is set to True after a tile change is detected to stop the direction
     from changing too many
190.                 # times.
191.                 if self.get_new_move:
192.                     move = self.validate_move(self.move)
193.
194.                     # If return_none is false then self.facing is still a valid move
195.                     if move == self.facing:
196.                         return self.facing
197.
198.                     # If move is None it is because the sprite has collided with a wall
199.                     elif move is None:
200.                         return None
201.
202.                     else:
203.                         self.get_new_move = False
204.                         return move
205.
206.                 # Move in the current direction until the tile has changed
207.                 else:
208.                     return self.validate_move(self.facing)
209.
210.             def validate_move(self, move):
211.                 """
212.                 Checks specifically whether the move will cause the player to collide with a wall o
    r whether they are colliding.
```

```
213.                :param move: Move that the sprite wants to use.
214.                :return: A valid (won't collide with a wall) move.
215.                """
216.
217.                tile_facing = self.get_next_tile(self.facing)
218.                tile_move = self.get_next_tile(move)
219.
220.                if tile_move.type in ['wall', 'ghost_barrier']:
221.                    if tile_facing.type in ['wall', 'ghost_barrier']:
222.                        if self.rect.colliderect(tile_facing.rect):
223.                            self.return_None = True
224.                            return None
225.                        else:
226.                            return self.facing
227.                    else:
228.                        return self.facing
229.                else:
230.                    return move
231.
232.            def correct_pos(self):
233.                """
234.                If the sprite is colliding with a wall it will work out how far the sprites (x,y) c
    o-ords differ from the tile
235.                it is currently on and gradually bring them closer together. This keeps the sprites
     centred and prevents
236.                sprites from clipping through walls.
237.                :return: None
238.                """
239.
240.                pac_x, pac_y = self.tile.pos
241.
242.                tiles = []
243.                try:
244.                    # Gets a list of all the tiles surrounding the sprite
245.                    tiles = [self.maze.tiles[y + pac_y - 3][x + pac_x] for x, y in [(1, 0), (-
    1, 0), (0, 1), (0, -1)]]
246.                except IndexError as e:
247.                    print(e)
248.
249.                for tile in tiles:
250.                    if self.rect.colliderect(tile.rect) and tile.type == 'wall':
251.                        # Delay means that every other call of the correct_pos function the followi
    ng is executed. This means
252.                        # the animation appears much smoother
253.                        if self.wall_defence_delay:
254.                            self.wall_defence_delay = False
255.                            wall_x, wall_y = tile.pos
256.                            difference_x = wall_x - pac_x
257.                            difference_y = wall_y - pac_y
258.
259.                            self.x -= difference_x
260.                            self.y -= difference_y
261.
262.                        else:
263.                            self.wall_defence_delay = True
264.
265.            def correct_tunnel(self):
266.                """
267.                Allows players to go through the tunnels by changing their x co ordinate when they
    go off the screen
268.                :return: None
269.                """
270.
271.                # There are 12 pixels in each tile which is where the 12 comes from
272.                if self.x < -12 * self.win_scale and self.facing == 'w':
273.                    self.x = 29 * 12 * self.win_scale
274.
275.                if self.x > 29 * 12 * self.win_scale and self.facing == 'e':
```

```
276.                     self.x = 0 * 12 * self.win_scale
277.
278.             def update_pos(self, move):
279.                 """
280.                 Updates sprite's current position, according to the move, current speed and win_sca
    le.
281.                 :param move: Move that has now been checked can be used to move the sprite.
282.                 :return: None
283.                 """
284.
285.                 # Dictionary keeping track of what direction the moves will move the sprite and wit
    h what magnitude (in this
286.                 # case it is a predetermined speed which can change throughout the game
287.                 moves = {'n': (0, -self._speed),
288.                          'e': (self._speed, 0),
289.                          's': (0, self._speed),
290.                          'w': (-self._speed, 0),
291.                          None: (0, 0)
292.                          }
293.
294.                 x, y = moves[move]
295.
296.                 # This records what move has been used to move in the direction. Usually sent to th
    e server in a
297.                 # multiplayer game to correctly show the direction of a sprite on clients
298.                 self.online_move = move
299.
300.                 # Skin clock is incremented once every frame unless the sprite is not moving. This
    is to make sure the skin
301.                 # isn't changing while a sprite is stationary. The skin_clock attribute is used in
    the get_skin method
302.                 if move is not None:
303.                     self.skin_clock += 1
304.
305.                 # sprite position updated as per the above move and multiplied by win_scale to allo
    w different sized windows
306.                 self.x += x * self.win_scale
307.                 self.y += y * self.win_scale
308.
309.                 # the position is then used to form the new rectangles which are used for blitting
    to the screen (skin_rect) and
310.                 # for managing collisions (rect)
311.                 self.skin_rect = self.skins['{}_{}.png'.format(self.facing, self.skin)].get_rect(ce
    nter=(self.x, self.y))
312.                 self.rect = pg.Rect(self.x - int(6 * self.win_scale),
313.                                     self.y - int(6 * self.win_scale),
314.                                     12 * self.win_scale,
315.                                     12 * self.win_scale)
316.
317.             def update_tile(self):
318.                 """
319.                 Updates what tile the sprite is on. This is used by many methods to determine wheth
    er the sprite is going to
320.                 collide with walls in the future.
321.                 :return: None
322.                 """
323.
324.                 # Gets tile x,y coords as opposed to pixel x,y coords based on the sprites pixel po
    sition
325.                 rect_tile_x, rect_tile_y = self.rect.centerx / (12 * self.win_scale), self.rect.cen
    tery / (12 * self.win_scale)
326.
327.                 # Gets tile x,y coords for the sprites current tile
328.                 tile_x, tile_y = self.tile.pos
329.
330.                 # If we do the following when the sprite is off the screen (in the tunnel) we get m
    any errors
331.                 if rect_tile_x > 0:
```

```
332.
333.                     # If the tile x,y from the pixel position is not equal to the current tile, we
    update the current tile to
334.                     # whichever tile the current pixel coords are inside of
335.                     if not(int(rect_tile_x) == tile_x and int(rect_tile_y) == tile_y):
336.                         self.previous_tile = self.tile
337.                         try:
338.                             self.tile = self.maze.tiles[int(rect_tile_y) - 3][int(rect_tile_x)]
339.                         except IndexError as e:
340.                             print(e)
341.                         # If the tile has changed we can receive a new move
342.                         self.get_new_move = True
343.
344.             def display(self, win):
345.                 """
346.                 Displays the sprite with the skin that corresponds with the direction and also the
    skin_number (skin attribute)
347.                 which is either a 0 or a 1.
348.                 :param win: The current window, all objects must be blitted to this window to be di
    splayed.
349.                 :return: None
350.                 """
351.
352.                 win.blit(self.skins['{}_{}.png'.format(self.facing, self.skin)], self.skin_rect)
353.
354.             def get_next_tile(self, move):
355.                 """
356.                 This is used by the move validating methods by returning the next tile the sprite w
    ill collide with if it
357.                 carries out the move passed through.
358.                 :param move: The method will return the next tile after this move.
359.                 :return: The next tile that will be reached if the sprite continues with the move.
360.                 """
361.
362.                 # This is the dictionary storing which tiles (in relation to the current one) will
    need to be checked depending
363.                 # on the move argument
364.                 checks = {'n': (0, -1),
365.                           'e': (1, 0),
366.                           's': (0, 1),
367.                           'w': (-1, 0)}
368.
369.                 # if the move is None the method returns the move that
370.                 if move is None:
371.                     check = checks[self.facing]
372.                 else:
373.                     check = checks[move]
374.                 x, y = check
375.                 tile_x, tile_y = self.tile.pos
376.                 # All y values must have 3 subtracted from them as the game is 3 tiles below the to
    p of the window to allow
377.                 # space for indicators such as score and highscore
378.                 tile_y -= 3
379.
380.                 try:
381.                     # Because tiles is a two dimensional list the y value must go first
382.                     return self.maze.tiles[tile_y + y][tile_x + x]
383.                 except IndexError as e:
384.                     print(e)
385.                     return self.maze.tiles[14][0]
386.
387.             def draw_rect(self, win):
388.                 """
389.                 Used for debugging
390.                 :param win: the current window, all objects must be blitted to this window to be di
    splayed
391.                 :return: None
392.                 """
```

```
393.                 pg.draw.rect(win, (255, 0, 0), self.rect)
394.
395.
396.         class PacMan(Sprite):
397.             def __init__(self, resource_pack, maze, win_scale):
398.                 """
399.                 Contains all of the extra information specific to Pac-Man (not shared with ghosts)
400.                 :param resource_pack: Contains the path to the folder containing the skins for the
    sprite.
401.                 :param maze: Two dimensional list representation of the maze.
402.                 :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables.
403.                 """
404.
405.                 # Essential
406.                 position = (167, 318)
407.                 super().__init__(resource_pack, position, maze, win_scale)
408.
409.                 # Skins
410.                 self.skin = '0'
411.                 self.facing = 'e'
412.                 self.move = 'e'
413.                 self.num = 0
414.
415.                 #   Death animation
416.                 self.death_animation_index = 0
417.                 self.death_animation_skins = {}
418.                 self.death_animation_finished = False
419.
420.                 for skin in os.listdir('resources\\sprites\\{}'.format('death_animation')):
421.                     # noinspection PyUnresolvedReferences
422.                     self.death_animation_skins.update({skin: pg.transform.scale(
423.                         pg.image.load('resources\\sprites\\{}\\{}'.format('death_animation', skin))
    ,
424.                         ((22 * win_scale), (22 * win_scale)))})
425.
426.                 # Sounds
427.                 self.sounds = {}
428.                 for sound in os.listdir('resources\\{}\\{}'.format('sounds', resource_pack)):
429.                     # noinspection PyUnresolvedReferences
430.                     self.sounds.update({sound: pg.mixer.Sound('resources\\{}\\{}\\{}'.format('sound
    s', resource_pack, sound))})
431.
432.             def update(self, events):
433.                 """
434.                 Run once a frame, this is the method that controls everything to do with Pac-Man.
435.                 :param events: Contains events from the pg.event.get() call containing all keyboard
     events.
436.                 :return: State of Pac-
    Man (True: Death Animation Playing, None: Dead so no longer has state, False: Alive)
437.                 """
438.
439.                 if not self.dead:
440.                     move = self.get_move(events)
441.                     super().update(move)
442.                     if move is not None:
443.                         # Plays siren sound only if Pac-
    Man is moving and if there isn't already a siren sound playing
444.                         if not self.sound_channel.get_busy():
445.                             self.sound_channel.play(self.sounds['siren.wav'])
446.                 else:
447.                     self.death_animation()
448.
449.             def death_animation(self):
450.                 """
451.                 Cycles through a series of death animation skins and plays the death animation soun
    d
452.                 :return: True if death animation has finished
```

```
453.                 """
454.
455.                 # Plays the sound once the first skin has been displayed (after 1/8 of a second) un
    less it is already playing
456.                 if not self.sound_channel.get_busy() and self.death_animation_index == 1:
457.                     self.sound_channel.play(self.sounds['death.wav'])
458.                 self.death_animation_clock += 1/60
459.                 if self.death_animation_clock > 1/8 and not self.death_animation_finished:
460.                     self.death_animation_clock = 0
461.                     self.death_animation_index += 1
462.                 if self.death_animation_index == 13:
463.                     self.death_animation_finished = True
464.
465.             def display(self, win):
466.                 """
467.                 This display is slightly different as it either displays normally if Pac-
    Man is alive or displays a death
468.                 animation skin if he is dead.
469.                 :param win: the current window, all objects must be blitted to this window to be di
    splayed
470.                 :return: None
471.                 """
472.
473.                 if self.dead:
474.                     win.blit(self.death_animation_skins['{}.png'.format(self.death_animation_index)
    ], self.skin_rect)
475.                 elif not self.dead:
476.                     win.blit(self.skins['{}_{}.png'.format(self.facing, self.skin)], self.skin_rect
    )
477.
478.
479.         class Ghost(Sprite):
480.             def __init__(self, resource_pack, position, target, maze, win_scale, level):
481.                 """
482.                 Contains all of the extra information specific to Pac-Man (not shared with Pac-
    Man)
483.                 :param resource_pack: Contains the path to the folder containing the skins for the
    sprite.
484.                 :param position: Each ghost starts with a different position on the maze so this is
     required.
485.                 :param target: Pac-Man object. Required to receive updates on which tile Pac-
    Man is currently on.
486.                 :param maze: Two dimensional list representation of the maze.
487.                 :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
488.                 :param level: Level number, used to determine how long to wait between the two mode
    s: chase (length increases
489.                 over levels) to scatter (length decreases over levels) and how long between becomin
    g scared to returning to
490.                 normal (lowers over levels).
491.                 """
492.
493.                 super().__init__(resource_pack, position, maze, win_scale)
494.
495.                 # Speed
496.                 self._speed = 4 / 3
497.                 self.speed_buffer = self._speed
498.
499.                 # this is only used by Blinky. When there are are certain number of pellets (lowers
     as levels progress). Blinky
500.                 # will enter elroy mode and this will be set to True
501.                 self.elroy = False
502.                 # When there are even less pellets this will be set to True
503.                 self.upgraded_elroy = False
504.
505.                 # This is used to keep track of which ghost caught Pac-Man
506.                 self.won = False
507.
```

```
508.                # These are the times between mode changes i.e on level 1 ghost will scatter for 7
    seconds and chase for 20 etc.
509.                if level == 1:
510.                    self.mode_timings = [7, 20, 7, 20, 5, 20, 5, 9999]
511.                elif level < 5:
512.                    self.mode_timings = [7, 20, 7, 20, 5, 1033, 1/60, 9999]
513.                else:
514.                    self.mode_timings = [5, 20, 5, 20, 5, 1037, 1/60, 9999]
515.
516.                # Skins
517.                self.facing = 'e'
518.                self.skin = '0'
519.
520.                self.skin_cap = 10
521.
522.                skin_size = ((22 * win_scale), (22 * win_scale))
523.                self.scared_skins = {}
524.                for skin in os.listdir(os.path.join('resources', 'sprites', 'scared')):
525.                    scared_skin_path = os.path.join('resources', 'sprites', 'scared', skin)
526.                    self.scared_skins.update(
527.                        {skin: pg.transform.scale(pg.image.load(scared_skin_path), skin_size)}
528.                    )
529.
530.                self.dead_skins = {}
531.                for skin in os.listdir(os.path.join('resources', 'sprites', 'dead')):
532.                    dead_skin_path = os.path.join('resources', 'sprites', 'dead', skin)
533.                    self.dead_skins.update(
534.                        {skin: pg.transform.scale(pg.image.load(dead_skin_path), skin_size)}
535.                    )
536.
537.                self.scared_flashing_skins = {}
538.                for skin in os.listdir(os.path.join('resources', 'sprites', 'scared_flashing')):
539.                    scared_flashing_skin_path = os.path.join('resources', 'sprites', 'scared_flashi
    ng', skin)
540.                    self.scared_flashing_skins.update(
541.                        {skin: pg.transform.scale(pg.image.load(scared_flashing_skin_path), skin_si
    ze)}
542.                    )
543.
544.                self.colour = (255, 255, 255)
545.
546.                # Rect
547.                self.skin_rect = self.skins['e_0.png'].get_rect(center=(self.x, self.y))
548.
549.                self.mode_index = 0
550.                self.mode_count = 0
551.
552.                # Modes
553.                self.mode = self.scatter
554.                self.buffer_mode = self.scatter
555.
556.                self.to_switch = False
557.
558.                self.scared = False
559.                self.scared_cap = 5 - 0.3 * level
560.                if self.scared_cap < 0:
561.                    self.scared_cap = 0
562.                self.scared_clock = 0
563.
564.                self.respawned = False
565.
566.                # Path finding
567.                self.search = Search(maze.tile_map)
568.
569.                self.target = target
570.
571.                self.home = (26, 4)
572.
```

```
573.                self.next_coords = self.scatter()[1]
574.
575.                self.path = self.get_path(self.mode)
576.
577.                self.next_x = 0
578.                self.next_y = 0
579.
580.                self.axis_change = 'x'
581.
582.                # Sound
583.                self.sound_channel = pg.mixer.Channel(1)
584.                self.sound_channel.set_volume(0.5 * (local_settings.get_setting('game_volume')/100)
    )
585.                self.death_sound = pg.mixer.Sound('resources\\sounds\\ghost\\death.wav')
586.
587.            def kill(self):
588.                """
589.                When a ghost is killed this is run.
590.                :return: None
591.                """
592.
593.                self.dead = True
594.                self.scared = False
595.                self._speed = 2
596.                self.skins = self.dead_skins
597.                self.sound_channel.play(self.death_sound)
598.
599.            def update(self, events):
600.                """
601.                Run once a frame, this is the method that controls everything to do with the Ghost.

602.                :param events: Contains events from the pg.event.get() call containing all keyboard
     events.
603.                :return: None
604.                """
605.                self.mode = self.get_mode()
606.                if self.to_switch:
607.                    move = self.switch()
608.                    self.to_switch = False
609.                else:
610.                    move = self.get_move(events)
611.
612.                super().update(move)
613.                self.check_collision()
614.
615.            def get_mode(self):
616.                """
617.                Determines which mode should be used to get the Ghosts next target coords.
618.                :return: Name of the correct coord-getting method.
619.                """
620.
621.                # Keeps track of how long a mode has been active for
622.                self.mode_count += 1/60
623.
624.                # Once the current mode has been active for the preset amount of time
625.                if int(self.mode_count) == self.mode_timings[self.mode_index]:
626.                    self.mode_index += 1
627.                    # Reset count
628.                    self.mode_count = 0
629.
630.                    # Toggles buffer mode between scatter or chase. Buffer is needed as some modes
    (e.g. 'scared' mode) stay
631.                    # active even when ghosts have changed to a different base mode ('chase' or 'sc
    atter'). Switch is also set
632.                    # to True. This is a boolean and not a return as it is only used if the ghost i
    s not 'scared' and not 'dead'
633.                    # / 'respawning'.
634.                    if self.buffer_mode == self.scatter:
```

```
635.                         if not(self.scared or self.dead):
636.                             self.to_switch = True
637.                         self.buffer_mode = self.chase
638.                         return self.chase
639.                     else:
640.                         if not(self.scared or self.dead):
641.                             self.to_switch = True
642.                         self.buffer_mode = self.scatter
643.                         return self.scatter
644.
645.                 # Slows down the ghost when they are passing over a ghost barrier (like the origina
    l game)
646.                 if not self.dead:
647.                     if self.tile.type == 'ghost_barrier':
648.                         self._speed = 0.25
649.                     elif not self.scared:
650.                         self._speed = self.speed_buffer
651.
652.                 # When a ghost reaches either of these coords (outside the ghost area) their mode i
    s no longer 'respawn'. The
653.                 # active mode will then become either 'chase' or 'scatter' depending on the buffer
    mode.
654.                 if self.tile.pos in [(13, 14), (14, 14)]:
655.                     self.respawned = False
656.
657.                 # If a ghost is dead the active mode is 'respawn' which will direct them to the gho
    st area.
658.                 if self.dead:
659.                     # Once they reach the ghost area -
    (13, 18) and (14, 18) are in the ghost area- they become alive again.
660.                     # There skins and speed change to account for this
661.                     if self.tile.pos in [(13, 18), (14, 18)]:
662.                         self.dead = False
663.                         self.respawned = True
664.                         self.to_switch = True
665.                         self._speed = self.speed_buffer
666.                         self.skins = self.normal_skins
667.                     else:
668.                         return self.respawn
669.
670.                 elif self.respawned:
671.                     return self.respawn
672.
673.                 elif self.scared:
674.                     return self.random
675.
676.                 return self.buffer_mode
677.
678.             def check_collision(self):
679.                 """
680.                 Checks whether the ghost is colliding with the target (Pac-Man)
681.                 :return: None
682.                 """
683.
684.                 if self.rect.colliderect(self.target.rect):
685.                     if self.scared:
686.                         self.kill()
687.                     elif self.dead:
688.                         pass
689.                     else:
690.                         self.won = True
691.                         self.target.kill()
692.                 else:
693.                     self.won = False
694.
695.             def get_move(self, events):
696.                 """
```

```
697.                Uses the current mode to get the next coords. Works out the next move based on the
    target coords.
698.                :param events: Events not used, but keeps same method signature.
699.                :return: Ghost's move.
700.                """
701.
702.                # When a ghost is in a tunnel their speed must decrease to 0.8, they must continue
    in the direction they are
703.                # facing and as soon as they leave the tunnel they must get a new path
704.
705.                path = self.path
706.
707.                if self.tile.pos[1] == 17:
708.                    if self.tile.pos[0] <= 5 or self.tile.pos[0] >= 22:
709.                        self._speed = 0.8
710.                        if self.tile.pos[0] == 5 or self.tile.pos[0] == 22:
711.                            path = self.get_path(self.mode)
712.                            if path is None:
713.                                return self.facing
714.                        else:
715.                            return self.facing
716.                    else:
717.                        if not self.scared:
718.                            if not self.dead:
719.                                self._speed = self.speed_buffer
720.
721.                self.path = path
722.                self.next_coords = self.path[1]
723.
724.                # This bit tests to see if the ghost has reached the 'next coords'. If it has, then
     new coords are calculated
725.                tile_x, tile_y = self.tile.pos
726.                x, y = self.next_coords
727.
728.                if x == tile_x and y == tile_y - 3:
729.                    path = self.get_path(self.mode)
730.
731.                    if path is None:
732.                        path = self.chase()
733.
734.                    if path is None:
735.                        return self.facing
736.
737.                    self.path = path
738.                    self.next_coords = self.path[1]
739.
740.                # Works out which direction the next coordinates are
741.                x, y = self.next_coords
742.                y += 3
743.
744.                x *= 12 * self.win_scale
745.                x += 6 * self.win_scale
746.
747.                y *= 12 * self.win_scale
748.                y += 6 * self.win_scale
749.
750.                self.next_x = x
751.                self.next_y = y
752.
753.                # If we just had self.x < x here then when we have a larger a screen the pos will j
    ump above and below the
754.                # desired coords. Having self.x - x < -
    self.win_scale: means that we say the ghost has reached the correct
755.                # coords when it is within a few pixels of the exact coords pos
756.                if self.axis_change == 'x':
757.                    if self.x < x and self.x - x < -self.win_scale:
758.                        return 'e'
759.                    elif self.x > x and self.x - x > self.win_scale:
```

```
760.                         return 'w'
761.                     else:
762.                         self.axis_change = 'y'
763.
764.                 if self.axis_change == 'y':
765.                     if self.y < y and self.y - y < -self.win_scale:
766.                         return 's'
767.                     elif self.y > y and self.y - y > self.win_scale:
768.                         return 'n'
769.                     else:
770.                         self.axis_change = 'x'
771.
772.             def get_move(self, events):
773.                 """
774.                 Uses the current mode to get the next coords. Works out the next move based on the
    target coords.
775.                 :param events: Events not used, but keeps same method signature.
776.                 :return: Ghost's move.
777.                 """
778.
779.                 # When a ghost is in a tunnel their speed must decrease to 0.8, they must continue
    in the direction they are
780.                 # facing and as soon as they leave the tunnel they must get a new path
781.                 if self.tile.pos[1] == 17:
782.                     if self.tile.pos[0] <= 5 or self.tile.pos[0] >= 22:
783.                         self._speed = 0.8
784.                         if self.tile.pos[0] == 5 or self.tile.pos[0] == 22:
785.                             try:
786.                                 self.next_coords = self.get_path(self.mode)[1]
787.                             except Exception as e:
788.                                 print(e)
789.                         return self.facing
790.                     else:
791.                         if not self.scared:
792.                             if not self.dead:
793.                                 self._speed = self.speed_buffer
794.
795.                 # This bit tests to see if the ghost has reached the 'next coords'. If it has, then
     new coords are calculated
796.                 tile_x, tile_y = self.tile.pos
797.                 x, y = self.next_coords
798.
799.                 if x == tile_x and y == tile_y - 3:
800.                     try:
801.                         self.next_coords = self.get_path(self.mode)[1]
802.                     except TypeError as e:
803.                         print(e)
804.                         try:
805.                             self.next_coords = self.chase()[1]
806.                         except TypeError as e:
807.                             print(e)
808.
809.                 # Works out which direction the next coordinates are
810.                 x, y = self.next_coords
811.                 y += 3
812.
813.                 x *= 12 * self.win_scale
814.                 x += 6 * self.win_scale
815.
816.                 y *= 12 * self.win_scale
817.                 y += 6 * self.win_scale
818.
819.                 self.next_x = x
820.                 self.next_y = y
821.
822.                 # If we just had self.x < x here then when we have a larger a screen the pos will j
    ump above and below the
```

```
823.                # desired coords. Having self.x - x < -
    self.win_scale: means that we say the ghost has reached the correct
824.                # coords when it is within a few pixels of the exact coords pos
825.                if self.axis_change == 'x':
826.                    if self.x < x and self.x - x < -self.win_scale:
827.                        return 'e'
828.                    elif self.x > x and self.x - x > self.win_scale:
829.                        return 'w'
830.                    else:
831.                        self.axis_change = 'y'
832.
833.                if self.axis_change == 'y':
834.                    if self.y < y and self.y - y < -self.win_scale:
835.                        return 's'
836.                    elif self.y > y and self.y - y > self.win_scale:
837.                        return 'n'
838.                    else:
839.                        self.axis_change = 'x'
840.
841.            def validate_move(self, move):
842.                """
843.                Checks to see if a move is valid by checking whether it is opposite the current dir
    ection (facing). And by
844.                using the Sprite's validate_move above. This is not needed in classic mode as the p
    ath finding algorithm does
845.                not produce paths that require a 180 degree change in direction, however this is ne
    eded for online.
846.                :param move: Sprite's unchecked next move.
847.                :return: Validated move.
848.                """
849.
850.                directions = ['n', 'e', 's', 'w', 'n', 'e']
851.                if move == directions[directions.index(self.facing) + 2]:
852.                    return super().validate_move(self.facing)
853.                else:
854.                    return super().validate_move(move)
855.
856.            def get_path(self, mode):
857.                """
858.                Uses the mode to get a path. This middle man is needed in case the target is unreac
    hable (in which case the path
859.                is None and instead the chase mode is used (which is always reachable).
860.                :param mode: Method for retrieving the path.
861.                :return: Path.
862.                """
863.
864.                path = mode()
865.                if path is None:
866.                    path = self.chase()
867.                return path
868.
869.            def scare(self):
870.                """
871.                Sets the ghost into scared mode when called.
872.                :return: None.
873.                """
874.
875.                if self.dead:
876.                    # If the ghost is dead then they cannot become scared
877.                    pass
878.
879.                elif not self.scared:
880.                    # If the ghost is not already scared change the following
881.                    self._speed = 0.5
882.                    self.skins = self.scared_skins
883.                    self.scared_clock = 0
884.                    self.scared = True
885.                    self.switch()  # Changes the ghost's direction
```

```
886.                 Thread(target=self.scared_timer).start()  # Keeps track of how long the ghost i
    s scared
887.
888.             else:
889.                 # If the ghost is already scared, set the skins to scared and reset the clock
890.                 self.skins = self.scared_skins
891.                 self.scared_clock = 0
892.
893.         def scared_timer(self):
894.             """
895.             Keeps track of how long the ghost is scared and adjust attributes accordingly.
896.             :return: None.
897.             """
898.
899.             while self.scared:
900.                 if self.scared_clock > self.scared_cap:
901.                     # After the cap begin swapping the skins 4 times a second (rate based on ho
    w long the sleep is)
902.                     if self.skins == self.scared_skins:
903.                         self.skins = self.scared_flashing_skins
904.                     else:
905.                         self.skins = self.scared_skins
906.
907.                 if int(self.scared_clock) == 8:
908.                     self.scared = False
909.                     self._speed = 4/3
910.                     self.skins = self.normal_skins
911.                     break
912.
913.                 sleep(0.25)
914.                 self.scared_clock += 0.25
915.
916.         def draw_target(self, win):
917.             """
918.             At the moment used for debugging ghost paths and ensuring they are working correctl
    y. Displays ghost's path
919.             :param win: The current window, all objects must be blitted to this window to be di
    splayed.
920.             :return: None.
921.             """
922.
923.             try:
924.                 for x, y in self.path:
925.                     pg.draw.rect(win, (0, 255, 0), pg.Rect(x * 12 * self.win_scale, (y+3) * 12
    * self.win_scale, 15, 15))
926.             except Exception as e:
927.                 print(e)
928.
929.         def draw_next_tile_target(self, win):
930.             """
931.             Used for debugging ghost paths and ensuring they are working correctly. Displays gh
    ost's target tile.
932.             :param win: The current window, all objects must be blitted to this window to be di
    splayed.
933.             :return: None.
934.             """
935.
936.             pg.draw.rect(win, (0, 255, 0), pg.Rect(self.next_x, self.next_y, 4, 4))
937.
938.         def draw_path(self, win):
939.             """
940.             Used for the storymode to teach the use how the AI works. Simply draws the path tha
    t the AI will take at any
941.             given moment.
942.             :param win: The current window, all objects must be blitted to this window to be di
    splayed.
943.             :return:
944.             """
```

```
945.
946.                 try:
947.                     tile_path = [self.maze.tiles[tile_y][tile_x] for tile_x, tile_y in self.path]
948.
949.                     for pathtile_rect in self.get_pathtiles(tile_path, []):
950.                         pg.draw.rect(win, self.colour, pathtile_rect)
951.                 except IndexError as e:
952.                     print(f'{e} in draw_path')
953.
954.             def get_pathtiles(self, path, pathtiles):
955.                 """
956.                 Simply returns the pygame rectangle responsible for displaying the path. This is in a sepearte method so it can
957.                 be run recursivley. It is static but I have included in the object so it is easy to see how it is being called.
958.                 :return:
959.                 """
960.
961.                 if len(path) == 1:
962.                     return pathtiles
963.
964.                 previous_tile = path[0]
965.                 previous_x, previous_y = previous_tile.rect.center
966.
967.                 current_tile = path[1]
968.                 current_x, current_y = current_tile.rect.center
969.
970.                 rect_long = 14 * self.win_scale
971.                 rect_short = 2 * self.win_scale
972.
973.                 if previous_x == current_x:
974.                     # must be a change in the y
975.                     if previous_y < current_y:
976.                         rect = pg.Rect(current_x - self.win_scale, previous_y - self.win_scale, rect_short, rect_long)
977.                     else:
978.                         rect = pg.Rect(current_x - self.win_scale, current_y - self.win_scale, rect_short, rect_long)
979.                 else:
980.                     # must be a change in the x
981.                     if previous_x < current_x:
982.                         rect = pg.Rect(previous_x - self.win_scale, current_y - self.win_scale,  rect_long, rect_short)
983.                     else:
984.                         rect = pg.Rect(current_x - self.win_scale, current_y - self.win_scale,  rect_long, rect_short)
985.
986.                 pathtiles.append(rect)
987.
988.                 return self.get_pathtiles(path[1:], pathtiles)
989.
990.             def switch(self):
991.                 """
992.                 Changes the direction of the ghost.
993.                 :return: Returns the opposite of the current move
994.                 """
995.
996.                 directions = ['n', 'e', 's', 'w', 'n', 'e']
997.                 move = directions[directions.index(self.facing) + 2]
998.                 x, y = self.previous_tile.pos
999.                 self.next_coords = (x, y-3)
1000.                return move
1001.
1002.            def respawn(self):
1003.                """
1004.                Pathfinding mode: It targets inside the centre, then targets outside once it has reached it.
1005.                :return: The next path.
```

```
1006.                """
1007.
1008.                start_tile = self.tile.pos
1009.                if self.respawned:
1010.                    target_tile = (13, 14)
1011.                else:
1012.                    target_tile = (13, 18)
1013.                start_tile = (start_tile[0], start_tile[1] - 3)
1014.                target_tile = (target_tile[0], target_tile[1] - 3)
1015.                return self.search.astar(start_tile, target_tile, self.facing)
1016.
1017.            def scatter(self):
1018.                """
1019.                Pathfinding mode: It targets the specific ghost's home tile.
1020.                :return: The next path.
1021.                """
1022.
1023.                start_tile = self.tile.pos
1024.                target_tile = self.home
1025.                start_tile = (start_tile[0], start_tile[1] - 3)
1026.                target_tile = (target_tile[0], target_tile[1] - 3)
1027.                return self.search.astar(start_tile, target_tile, self.facing)
1028.
1029.            def chase(self):
1030.                """
1031.                Pathfinding mode: Unique to ghosts: default (blinky) targets Pac-
    Man's current tile.
1032.                :return: The next path.
1033.                """
1034.
1035.                start_tile = self.tile.pos
1036.                target_tile = self.target.tile.pos
1037.                start_tile = (start_tile[0], start_tile[1] - 3)
1038.                target_tile = (target_tile[0], target_tile[1] - 3)
1039.                return self.search.astar(start_tile, target_tile, self.facing)
1040.
1041.            def random(self):
1042.                """
1043.                Pathfinding mode: Targets random row and random tile on that row as long as it's a
    pellet.
1044.                :return: The next path.
1045.                """
1046.
1047.                start_tile = self.tile.pos
1048.                chosen_row = random.choice(self.maze.tiles[1:-1])
1049.                pellet_tiles = [tile for tile in chosen_row if tile.type == 'pellet']
1050.                target_tile = random.choice(pellet_tiles).pos
1051.                start_tile = (start_tile[0], start_tile[1] - 3)
1052.                target_tile = (target_tile[0], target_tile[1] - 3)
1053.
1054.                return self.search.astar(start_tile, target_tile, self.facing)
1055.
1056.
1057.        class Blinky(Ghost):
1058.            def __init__(self, resource_pack, target, maze, win_scale, level):
1059.                """
1060.                Class for Blinky (contains home tile, starting position and can become elroy).
1061.                :param resource_pack: Contains the path to the folder containing the skins for the
    sprite.
1062.                :param target: Pac-Man object. Required to receive updates on which tile Pac-
    Man is currently on.
1063.                :param maze: Two dimensional list representation of the maze.
1064.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
1065.                :param level: Level number, used to determine how long to wait between the two mode
    s: chase (length increases
1066.                over levels) to scatter (length decreases over levels) and how long between becomin
    g scared to returning to
```

```
1067.                normal (lowers over levels).
1068.                """
1069.
1070.                position = (168, 176)
1071.                super().__init__(resource_pack, position, target, maze, win_scale, level)
1072.                self.facing = 'e'
1073.                self.visible = True
1074.                self.colour = (222, 0, 0)
1075.
1076.            def make_elroy(self):
1077.                """
1078.                Turns Blinky into elroy (faster).
1079.                :return: None
1080.                """
1081.
1082.                self.elroy = True
1083.                self.speed_buffer = 13/9
1084.
1085.            def elroy_upgrade(self):
1086.                """
1087.                Turns Blinky into upgraded elroy (faster, and still targets Pac-
    Man in scatter mode).
1088.                :return: None
1089.                """
1090.                self.upgraded_elroy = True
1091.                self.speed_buffer = 5/3
1092.
1093.            def scatter(self):
1094.                """
1095.                Pathfinding mode: It targets the Blinky's home tile unless Blinky is in elroy mode,
     in which case this will
1096.                function in the same way as the chase mode.
1097.                :return: The next path.
1098.                """
1099.
1100.                start_tile = self.tile.pos
1101.                if not self.elroy:
1102.                    target_tile = self.home
1103.                else:
1104.                    target_tile = self.target.tile.pos
1105.
1106.                start_tile = (start_tile[0], start_tile[1] - 3)
1107.                target_tile = (target_tile[0], target_tile[1] - 3)
1108.                return self.search.astar(start_tile, target_tile, self.facing)
1109.
1110.
1111.        class Pinky(Ghost):
1112.            def __init__(self, resource_pack, target, maze, win_scale, level):
1113.                """
1114.                Class for Pinky (contains home tile, starting position).
1115.                :param resource_pack: Contains the path to the folder containing the skins for the
    sprite.
1116.                :param target: Pac-Man object. Required to receive updates on which tile Pac-
    Man is currently on.
1117.                :param maze: Two-dimensional list representation of the maze.
1118.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
1119.                :param level: Level number, used to determine how long to wait between the two mode
    s: chase (length increases
1120.                over levels) to scatter (length decreases over levels) and how long between becomin
    g scared to returning to
1121.                normal (lowers over levels).
1122.                """
1123.
1124.                position = (168, 214)
1125.                super().__init__(resource_pack, position, target, maze, win_scale, level)
1126.
1127.                self.home = (1, 4)
```

```
1128.                 self.facing = 's'
1129.                 self.colour = (255, 181, 255)
1130.
1131.             def chase(self):
1132.                 """
1133.                 Pathfinding mode: Uses the tile 4 spaces ahead of Pac-
    Man to get the path. This decreases by one until the
1134.                 target reaches Pac-Man if the tiles in front are not reachable.
1135.                 :return: The next path.
1136.                 """
1137.
1138.                 start_tile = self.tile.pos
1139.                 target_tile = self.target.tile.pos
1140.                 start_tile = (start_tile[0], start_tile[1] - 3)
1141.                 target_tile = (target_tile[0], target_tile[1] - 3)
1142.
1143.                 targets = {'n': (0, -4), 'e': (4, 0), 's': (0, 4), 'w': (-4, 0)}
1144.                 tile_x, tile_y = target_tile
1145.                 facing = self.target.facing
1146.                 x, y = targets[facing]
1147.
1148.                 if x == 0:
1149.                     change = 'y'
1150.                     tile_x_temp = tile_x
1151.                 else:
1152.                     change = 'x'
1153.                     tile_y_temp = tile_y
1154.
1155.                 for i in range(5):
1156.                     if change == 'x':
1157.                         if x > 0:
1158.                             tile_x_temp = tile_x + (x - i)
1159.                         else:
1160.                             tile_x_temp = tile_x + (x + i)
1161.                     else:
1162.                         if y > 0:
1163.                             tile_y_temp = tile_y + (y - i)
1164.                         else:
1165.                             tile_y_temp = tile_y + (y + i)
1166.
1167.                     try:
1168.                         if self.maze.tiles[abs(tile_y_temp)][tile_x_temp].type in ['pellet', 'empty
    _tile']:
1169.                             break
1170.
1171.                     except IndexError:
1172.                         continue
1173.
1174.                 path = self.search.astar(start_tile, (tile_x_temp, abs(tile_y_temp)), self.facing)

1175.                 if path is None:
1176.                     path = super().chase()
1177.                 return path
1178.
1179.
1180.         class Clyde(Ghost):
1181.             def __init__(self, resource_pack, target, maze, win_scale, level):
1182.                 """
1183.                 Class for Clyde (contains home tile, starting position, start clock (Clyde doesn't
    leave centre straight away).
1184.                 :param resource_pack: Contains the path to the folder containing the skins for the
    sprite.
1185.                 :param target: Pac-Man object. Required to receive updates on which tile Pac-
    Man is currently on.
1186.                 :param maze: Two dimensional list representation of the maze.
1187.                 :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
```

```
1188.             :param level: Level number, used to determine how long to wait between the two mode
    s: chase (length increases
1189.             over levels) to scatter (length decreases over levels) and how long between becomin
    g scared to returning to
1190.             normal (lowers over levels).
1191.             """
1192.
1193.             position = (192, 214)
1194.             super().__init__(resource_pack, position, target, maze, win_scale, level)
1195.             self.home = (1, 32)
1196.
1197.             self.path = [(16, 15), (16, 14), (16, 13)]
1198.             self.start_clock_master = 0
1199.             self.start_clock = 0
1200.             self.facing = 'n'
1201.             self.speed_buffer = self._speed
1202.             self._speed = 1
1203.
1204.             self.colour = (255, 181, 33)
1205.
1206.         def update(self, events):
1207.             """
1208.             Run once a frame, this is the method that controls the start (when Clyde is still i
    nside the centre).
1209.             :param events: Contains events from the pg.event.get() call containing all keyboard
     events.
1210.             :return: None
1211.             """
1212.
1213.             if self.start_clock_master > 18:
1214.                 if not self.scared:
1215.                     self._speed = self.speed_buffer
1216.                 super().update(events)
1217.             else:
1218.                 self.mode = self.get_mode()
1219.                 self.start_clock += 1/60
1220.                 self.start_clock_master += 1/60
1221.                 if self.start_clock <= 8/60:
1222.                     self.facing = 'n'
1223.                     self.update_pos('n')
1224.                 if 8/60 < self.start_clock <= 16/60:
1225.                     self.facing = 's'
1226.                     self.update_pos('s')
1227.                 elif self.start_clock > 16/60:
1228.                     self.start_clock = 0
1229.
1230.         def euclidean_distance(self, target):
1231.             """
1232.             Needs this to work out how far from Pac-Man Clyde is. (Used in chase method).
1233.             :param target: Object you want to measure the distance to (Pac-Man).
1234.             :return: Distance.
1235.             """
1236.             tile_x, tile_y = self.tile.pos
1237.             target_tile_x, target_tile_y = target.tile.pos
1238.
1239.             return ((tile_x - target_tile_x)**2 + (tile_y - target_tile_y)**2)**0.5
1240.
1241.         def chase(self):
1242.             """
1243.             Pathfinding mode: Targets Pac-
    Man's tile until the distance to him is less than 8 tiles, when Clyde, instead,
1244.             targets his home corner.
1245.             :return: The next path.
1246.             """
1247.             start_tile = self.tile.pos
1248.             start_tile = (start_tile[0], start_tile[1] - 3)
1249.             if self.euclidean_distance(self.target) > 8:
1250.                 target_tile = self.target.tile.pos
```

```
1251.                 else:
1252.                     target_tile = self.home
1253.
1254.                 target_tile = (target_tile[0], target_tile[1] - 3)
1255.
1256.                 path = self.search.astar(start_tile, target_tile, self.facing)
1257.                 if path is None:
1258.                     path = super().chase()
1259.                 return path
1260.
1261.
1262.         class Inky(Ghost):
1263.             def __init__(self, resource_pack, target, maze, win_scale, level, blinky):
1264.                 """
1265.                 Class for Inky (contains home tile, starting position).
1266.                 :param resource_pack: Contains the path to the folder containing the skins for the
    sprite.
1267.                 :param target: Pac-Man object. Required to receive updates on which tile Pac-
    Man is currently on.
1268.                 :param maze: Two dimensional list representation of the maze.
1269.                 :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
1270.                 :param level: Level number, used to determine how long to wait between the two mode
    s: chase (length increases
1271.                 over levels) to scatter (length decreases over levels) and how long between becomin
    g scared to returning to
1272.                 normal (lowers over levels).
1273.                 """
1274.
1275.                 position = (144, 214)
1276.                 super().__init__(resource_pack, position, target, maze, win_scale, level)
1277.                 self.home = (26, 32)
1278.                 self.blinky = blinky
1279.                 self.start_clock_master = 0
1280.                 self.start_clock = 0
1281.                 self.facing = 'n'
1282.                 self.speed_buffer = self._speed
1283.                 self._speed = 1
1284.
1285.                 self.colour = (0, 222, 222)
1286.
1287.             def update(self, events):
1288.                 """
1289.                 Run once a frame, this is the method that controls the start (when Inky is still in
    side the centre).
1290.                 :param events: Contains events from the pg.event.get() call containing all keyboard
     events.
1291.                 :return: None
1292.                 """
1293.
1294.                 if self.start_clock_master > 4:
1295.                     if not self.scared:
1296.                         self._speed = self.speed_buffer
1297.                     super().update(events)
1298.                 else:
1299.                     self.mode = self.get_mode()
1300.                     self.start_clock += 1/60
1301.                     self.start_clock_master += 1/60
1302.                     if self.start_clock <= 8/60:
1303.                         self.facing = 'n'
1304.                         self.update_pos('n')
1305.                     if 8/60 < self.start_clock <= 16/60:
1306.                         self.facing = 's'
1307.                         self.update_pos('s')
1308.                     elif self.start_clock > 16/60:
1309.                         self.start_clock = 0
1310.
1311.             def chase(self):
```

```
1312.                """
1313.                Pathfinding mode: Takes the vector between Blinky and Pac-
    Man and doubles it. Adds this vector to Blinky's
1314.                position and target that tile.
1315.                :return: THe next path.
1316.                """
1317.
1318.                path = None
1319.                start_tile = self.tile.pos
1320.                start_tile = (start_tile[0], start_tile[1] - 3)
1321.
1322.                pac_x, pac_y = self.target.tile.pos
1323.                blinky_x, blinky_y = self.blinky.tile.pos
1324.                vector = (pac_x - blinky_x, pac_y - blinky_y)
1325.
1326.                x, y = vector
1327.
1328.                target_x, target_y = (pac_x + x, pac_y + y - 3)
1329.
1330.                found = False
1331.
1332.                try:
1333.                    if self.maze.tiles[abs(target_y)][target_x].type == 'pellet':
1334.                        path = self.search.astar(start_tile, (target_x, abs(target_y)), self.facing
    )
1335.                        found = True
1336.
1337.                except IndexError:
1338.                    if target_x > 26:
1339.                        target_x = 26  # check these numbers to make sure they are the correct ones

1340.                    elif target_x < 1:
1341.                        target_x = 1
1342.
1343.                    if target_y > 29:
1344.                        target_y = 29
1345.                    elif target_y < 1:
1346.                        target_y = y
1347.
1348.                if not found:
1349.                    for x, y in [(0, 0), (-1, 0), (1, 0), (0, 1), (0, -1)]:
1350.                        tile_x_temp = target_x + x
1351.                        tile_y_temp = target_y + y
1352.                        try:
1353.                            if self.maze.tiles[abs(tile_y_temp)][tile_x_temp].type == 'pellet':
1354.                                break
1355.                            else:
1356.                                continue
1357.                        except IndexError:
1358.                            continue
1359.
1360.                    path = self.search.astar(start_tile, (tile_x_temp, abs(tile_y_temp)), self.faci
    ng)
1361.
1362.                if path is None:
1363.                    path = super().chase()
1364.                return path
1365.
1366.
1367.      class Pellet:
1368.          def __init__(self, skin, tile, predator, win_scale, death_sound, sound_channel, power_p
    ellet=False):
1369.                """
1370.                Class for every pellet in the game.
1371.                :param skin: Contains the picture that is blitted to the screen.
1372.                :param tile: The tile that the pellet is on.
1373.                :param predator: The sprite that can collide with the pellet.
```

```
1374.             :param win_scale: Window Scale (How large the window is - must be multiplied by all
    size related variables).
1375.             :param death_sound: Sound file that plays when a pellet is eaten.
1376.             :param sound_channel: Pellet sound channel (same for all pellets, different one for
    all power pellets).
1377.             :param power_pellet: Boolean - decides whether or not
1378.             """
1379.
1380.             self.power_pellet = power_pellet
1381.             self.eaten = False
1382.
1383.             self.skin = skin
1384.             self.tile = tile
1385.             self.predator = predator
1386.
1387.             self.x = self.tile.pos[0]
1388.             self.y = self.tile.pos[1]
1389.
1390.             self.rect = self.skin.get_rect(center=(
1391.                                                    (self.x * 12 * win_scale) + 6 * win_scale,

1392.                                                    (self.y * 12 * win_scale) + 6 * win_scale
1393.                                                    )
1394.                                                )
1395.
1396.             self.sound_channel = sound_channel
1397.             self.death_sound = death_sound
1398.
1399.             self.display_clock = 0
1400.
1401.         def update(self):
1402.             """
1403.             Runs every frame, just checks whether the pellets is colliding.
1404.             :return: None
1405.             """
1406.
1407.             self.check_collision()
1408.
1409.         def display(self, win):
1410.             """
1411.             Displays the pellet using the skin. If it's a power pellet it will flash.
1412.             :param win: the current window, all objects must be blitted to this window to be di
    splayed
1413.             :return: None
1414.             """
1415.
1416.             if self.power_pellet:
1417.                 self.display_clock += 1/60
1418.                 if 0.3 > self.display_clock > 0.15:
1419.                     self.draw(win)
1420.                 elif self.display_clock > 0.3:
1421.                     self.display_clock = 0
1422.             else:
1423.                 self.draw(win)
1424.
1425.         def check_collision(self):
1426.             """
1427.             If pellet's and Pac-
    Man's  rectangles are colliding, kill pellet and play death sound.
1428.             :return: None
1429.             """
1430.
1431.             if self.rect.colliderect(self.predator.tile.rect):
1432.                 if not self.sound_channel.get_busy():
1433.                     self.sound_channel.play(self.death_sound)
1434.                 self.eaten = True
1435.
1436.         def draw(self, win):
```

```
1437.             """
1438.             Blits the skin to the window.
1439.             :param win: The current window, all objects must be blitted to this window to be di
    splayed.
1440.             :return: None
1441.             """
1442.
1443.             win.blit(self.skin, self.rect)
1444.
1445.
1446.     class StaticSprite:
1447.         def __init__(self, skins, rect):
1448.             """
1449.             Essentially just a picture (used to display how many lives the player has left).
1450.             :param skins: The skins that are blitted to the screen (can be one skin).
1451.             :param rect: The sprite's rect (decides where the skin is blitted).
1452.             """
1453.
1454.             self.skin_index = 0
1455.             self.skin_cap = 8
1456.             self.skin_count = 0
1457.
1458.             self.skins = skins
1459.             self.rect = rect
1460.
1461.         def update(self, events):
1462.             self.skin_count += 1
1463.             if self.skin_count == self.skin_cap:
1464.                 self.skin_index = abs(self.skin_index-1)
1465.                 self.skin_count = 0
1466.
1467.         def display(self, win):
1468.             """
1469.             Blits the skin to the window.
1470.             :param win: The current window, all objects must be blitted to this window to be di
    splayed.
1471.             :return: None
1472.             """
1473.
1474.             win.blit(self.skins[self.skin_index], self.rect)
1475.
1476.
1477.     if __name__ == '__main__':
1478.         pass
```

PATHFINDING

```
12. __author__ = 'Will Evans'
13.
14. from datastructures import *
15.
16.
17. class Node:
18.     def __init__(self, x, y, facing, parent=None):
19.         """
20.         Every tile in a path (or possible path) is called a node. It has tilex and tiley values an
    d also stores the
21.         the node object of the tile that came before it in the path.
22.         :param x: Tile x
23.         :param y: Tile y
24.         :param facing: The direction the sprite is currently facing (stops ghosts from being able
    to turn around).
25.         :param parent: The node object of the tile before in the path.
26.         """
27.
```

```
28.         self.x = x
29.         self.y = y
30.
31.         self.parent = parent
32.
33.         self.f_score = 0
34.         self.h_score = 0
35.         self.g_score = 0
36.
37.         self.facing = facing
38.
39.     def get_path(self, path):
40.         """
41.         Recursive algorithm to get the path once the target node has been reached.
42.         :param path: Path so far
43.         :return: Path with self added.
44.         """
45.
46.         if self.parent is not None:
47.             path = self.parent.get_path(path)
48.         path.append((self.x, self.y))
49.         return path
50.
51.
52. class Search:
53.     def __init__(self, maze):
54.         """
55.         Search is an object so that we can save the maze.
56.         :param maze: 2D list of the maze.
57.         """
58.
59.         self.maze = maze
60.
61.     def astar(self, start, end, facing):
62.         """
63.         Mainloop of the search algorithm.
64.         :param start: Start tile (tilex, tiley).
65.         :param end: End tile (tilex, tiley).
66.         :param facing: Direction sprite is currently facing.
67.         :return: Path in (x, y) format.
68.         """
69.
70.         open_queue = PriorityQueue()
71.         closed_set = []
72.         start_node = Node(*start, facing)
73.         start_node.h_score = self.heuristic(start_node, end)
74.         start_node.g_score = start_node.h_score + start_node.f_score
75.         open_queue.en_queue(start_node)
76.         flag = False
77.
78.         while not open_queue.is_empty():
79.
80.             # Getting the next node (closest to the goal) to evaluate
81.             current_node = open_queue.pop()
82.             closed_set.append(current_node)
83.
84.             # Checking whether the goal has been reached
85.             if (current_node.x, current_node.y) == end and flag:
86.                 return current_node.get_path([])
87.
88.             flag = True
89.
90.             # Getting adjacent nodes
91.             children = get_children(current_node, self.maze)
92.
93.             # Adding newly evaluated nodes to the open_queue if not already evaluated
94.             for child in children:
95.                 self.evaluate(child, end)
```

```
                if not open_queue.has(child):
                    if not in_closed(child, closed_set):
                        open_queue.en_queue(child)
        return None

            def evaluate(self, child, end):
                """
                Assigns each child an h score and a g score, then combines these for the f score. These determine the fitness of
                the child, by taking into account how many nodes there have been before the child and how close the child is to
                the end node. This score is then used to choose the next child to expand.
                :param child: Node object.
                :param end: End tile (tilex, tiley)
                :return: None
                """

                child.h_score = self.heuristic(child, end)
                child.g_score = child.parent.g_score + 1
                child.f_score = child.g_score + child.h_score

            def heuristic(self, node, end):
                return 0


        class Dijkstra(Search):
            # Checks all paths
            def __init__(self, maze):
                super().__init__(maze)

            # noinspection PyMethodMayBeStatic
            def heuristic(self, node, end):
                """
                Gives a score based on how close the tile is to the end child. In this case it returns 0, as the default
                heuristic is Dijkstra's which checks every path.
                :param node: Node to be evaluated.
                :param end: End tile (tilex, tiley)
                :return: Score.
                """

                return 0


        class Manhattan(Search):
            # Uses Manhattan distance as heuristic (most efficient)
            def __init__(self, maze):
                super().__init__(maze)

            # noinspection PyMethodMayBeStatic
            def heuristic(self, node, end):
                x, y = end
                return abs(node.x - x) + abs(node.y - y)


        class Euclidean(Search):
            # Not as efficient as Manhattan as cost of diagonal is the same as east and the north move in Pac-Man
            def __init__(self, maze):
                super().__init__(maze)

            # noinspection PyMethodMayBeStatic
            def heuristic(self, node, end):
                x, y = end
                return ((node.x - x)**2 + (node.y - y)**2)**0.5


        def get_children(node, maze):
```

```
            """
            Returns next available tiles from current tile. This can then be added to the list of c
    hildren.
            :param node: Current tile.
            :param maze: 2D list of maze.
            :return: List of children of current tile.
            """

            vectors = {'s': (0, 1), 'e': (1, 0), 'w': (-1, 0), 'n': (0, -1)}
            children = []
            possible_moves = ['n', 'e', 's', 'w', 'n', 'e']
            del vectors[possible_moves[possible_moves.index(node.facing) + 2]]
            for key, value in vectors.items():
                x, y = value
                try:
                    # Checking surrounding tiles for walls
                    if maze[node.y + y][node.x + x] != 1:
                        children.append(Node(node.x + x, node.y + y, key, node))
                except IndexError:
                    continue

            return children


        def in_closed(child, closed):
            """
            Checks if a child is already in the closed set (already been evaluated).
            :param child: Node object of child to be tested.
            :param closed: List of nodes in closed set.
            :return: Boolean as to whether or not the child is in the closed set.
            """

            for node in closed:
                if node.x == child.x and node.y == child.y:
                    return True
            return False
```

DATA STRUCTURES

```
__author__ = 'Will Evans'

import json
import local_database
import pygame as pg
import os


class PriorityQueue:

    def __init__(self):
        self.queue = []

    def is_empty(self):
        return len(self.queue) == 0

    def en_queue(self, node):
        self.queue.append(node)

    def pop(self):
        self.queue.sort(key=lambda x: x.f_score)
        return self.queue.pop(0)

    def has(self, child):
        for node in self.queue:
            if node.x == child.x and node.y == child.y:
```

```
                return True
        return False


def get_maze(maze_id):
    """
    Temporary function to retrieve the maze from a json file.
    :param maze_id: Each maze in the file has a maze ID
    :return: 2D list of the required maze.
    """

    return json.loads(local_database.get_maze(maze_id))


class Tile:
    def __init__(self, tile_x, tile_y, _type, win_scale, skin):
        """
        Class for a tile, containing the type of tile, position, skin and how to blit it.
        :param tile_x: Tile x-coord (not pixel)
        :param tile_y: Tile y-coord (not pixel)
        :param _type: The tile type stored as a number (directly from maze json file).
        :param win_scale: Window Scale (How large the window is - must be multiplied by all size r
    elated variables).
        :param skin: Tile's image.
        """

        self.pos = (tile_x, tile_y)
        self.x = tile_x
        self.y = tile_y

        if _type == 0:
            self.type = 'pellet'
            self.colour = (0, 0, 0)

        elif _type == 1:
            self.type = 'wall'
            self.colour = (0, 0, 255)

        elif _type == 2:
            self.type = 'power_pellet'
            self.colour = (0, 0, 0)

        elif _type == 3:
            self.type = 'ghost_barrier'
            self.colour = (0, 0, 0)

        elif _type == 4:
            self.type = 'empty_tile'
            self.colour = (0, 0, 0)

        elif _type == 5:
            self.type = 'out_of_bounds'
            self.colour = (0, 0, 0)

        elif _type == 6:
            self.type = 'inside'
            self.colour = (0, 0, 0)

        self.skin = skin

        self.rect = pg.Rect(tile_x * 12 * win_scale, tile_y * 12 * win_scale, 12 * win_scale, 12 *
     win_scale)

    def display(self, win):
        """
        Displays tile.
        :param win: The current window, all objects must be blitted to this window to be displayed
    .
```

```
92.         :return: None
93.         """
94.
95.         if self.type in ['wall', 'ghost_barrier']:
96.             win.blit(self.skin, self.rect)
97.         else:
98.             pg.draw.rect(win, self.colour, self.rect)
99.
100.
101.        class Maze:
102.            def __init__(self, maze_id, win_scale):
103.                """
104.                Class for the whole maze, contains details about all the tiles.
105.                :param maze: 2D list of all tiles (straight from json file).
106.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
    size related variables).
107.                """
108.
109.                self.tile_map = get_maze(maze_id)
110.                self.win_scale = win_scale
111.                self.tiles = self.get_tiles('blue')
112.                self.skin_colour = 'blue'
113.
114.            def get_tiles(self, skin_colour):
115.                """
116.                Gets list off tile objects based on the tile_map.
117.                :param skin_colour: This will decide which colour (blue or white) each tile is.
118.                :return: A 2D list of tile objects.
119.                """
120.
121.                tiles = []
122.                for y, row in enumerate(self.tile_map):
123.                    tiles.append([])
124.                    for x, data in enumerate(row):
125.                        if data == 1:
126.                            skin_name = f'{self.get_skin(x, y, self.tile_map)}.png'
127.                            skin_address = os.path.join('Resources', 'sprites', 'walls', skin_colou
    r, skin_name)
128.                            skin = pg.transform.scale(pg.image.load(skin_address), (12 * self.win_s
    cale, 12 * self.win_scale))
129.
130.                        elif data == 3:
131.                            skin_address = os.path.join('Resources', 'sprites', 'walls', skin_colou
    r, 'ghost_barrier.png')
132.                            skin = pg.transform.scale(pg.image.load(skin_address), (12 * self.win_s
    cale, 12 * self.win_scale))
133.                        else:
134.                            skin = None
135.                        tiles[y].append(Tile(x, y + 3, data, self.win_scale, skin))
136.
137.                return tiles
138.
139.            def change_skin(self):
140.                """
141.                Changes the colour of all tiles in the 2D list.
142.                :return: None
143.                """
144.
145.                if self.skin_colour == 'blue':
146.                    self.skin_colour = 'white'
147.                else:
148.                    self.skin_colour = 'blue'
149.                self.tiles = self.get_tiles(self.skin_colour)
150.
151.            def display(self, win):
152.                """
153.                Displays the maze, by displaying each individual Tile.
```

```
154.                :param win: The current window, all objects must be blitted to this window to be di
    splayed.
155.                :return: None
156.                """
157.
158.                for row in self.tiles:
159.                    for tile in row:
160.                        tile.display(win)
161.
162.            # noinspection PyMethodMayBeStatic
163.            def get_skin(self, tile_x, tile_y, tile_map):
164.                """
165.                Works out what skin each tile should have based on the type of tile in the surround
    ing 8 spaces.
166.                :param tile_x: x coord of tile.
167.                :param tile_y: y coord of tile.
168.                :param tile_map: The 2D list of tiles from the maze json file.
169.                :return: String name of skin for the tile specified by the x, y position.
170.                """
171.
172.                tile_map = [[0 if x == 2 else x for x in row] for row in tile_map]
173.
174.                # get adjacency list
175.                adjacent = {}
176.                vectors = {'s': (0, 1),
177.                           'e': (1, 0),
178.                           'w': (-1, 0),
179.                           'n': (0, -1),
180.                           'ne': (1, -1),
181.                           'se': (1, 1),
182.                           'sw': (-1, 1),
183.                           'nw': (-1, -1)
184.                           }
185.
186.                for key, value in vectors.items():
187.                    x, y = value
188.                    try:
189.                        new_x = tile_x + x
190.                        new_y = tile_y + y
191.
192.                        if new_x < 0 or new_y < 0:
193.                            adjacent.update({key: None})
194.                            continue
195.
196.                        adjacent.update({key: tile_map[tile_y + y][tile_x + x]})
197.                    except IndexError:
198.                        adjacent.update({key: None})
199.
200.                values = tuple(list(adjacent.values())[:4])
201.                values_diag = tuple(adjacent.values())
202.
203.                # ghost edges
204.                if values == (4, 3, 1, 4):
205.                    return 'left_end_ghost'
206.
207.                if values == (4, 1, 3, 4):
208.                    return 'right_end_ghost'
209.
210.                if values == (4, 1, 1, 4) and tile_y == 12:
211.                    return 'lower_boundary'
212.
213.                if values == (4, 1, 1, 4) and tile_y == 16:
214.                    return 'upper_boundary'
215.
216.                if values == (1, 4, 4, 1) and tile_x == 10:
217.                    return 'right_boundary'
218.
219.                if values == (1, 4, 4, 1) and tile_x == 17:
```

```
                return 'left_boundary'

            # ghost corners
            if values_diag == (1, 1, 4, 4, 4, 4, 4, 4):
                return 'left_upper_corner_ghost'

            if values_diag == (1, 4, 1, 4, 4, 4, 4, 4):
                return 'right_upper_corner_ghost'

            if values_diag == (4, 4, 1, 1, 4, 4, 4, 4):
                return 'right_lower_corner_ghost'

            if values_diag == (4, 1, 4, 1, 4, 4, 4, 4):
                return 'left_lower_corner_ghost'

            # boundaries edges
            values_temp = tuple([1 if x is None else x for x in values])

            if values == (0, 1, 1, None) or values == (0, 1, 1, 5) or values_temp == (4, 1, 1,
    5):
                return 'upper_boundary'

            if values == (None, 1, 1, 0) or values == (5, 1, 1, 0) or values_temp == (5, 1, 1,
    4):
                return 'lower_boundary'

            if values == (1, 0, None, 1) or values == (1, 0, 4, 1) or values == (1, 0, 5, 1):
                return 'left_boundary'

            if values == (1, None, 0, 1) or values == (1, 4, 0, 1) or values == (1, 5, 0, 1):
                return 'right_boundary'

            # boundary spits
            if values_diag == (1, 1, 1, None, None, 1, 0, None):
                return 'upper_left_spit_boundary'

            if values_diag == (1, 1, 1, None, None, 0, 1, None):
                return 'upper_right_spit_boundary'

            if values_diag == (None, 1, 1, 1, 1, None, None, 0):
                return 'lower_left_spit_boundary'

            if values_diag == (None, 1, 1, 1, 0, None, None, 1):
                return 'lower_right_spit_boundary'

            if values_diag == (1, None, 1, 1, None, None, 1, 0):
                return 'right_upper_spit_boundary'

            if values_diag == (1, None, 1, 1, None, None, 0, 1):
                return 'right_lower_spit_boundary'

            if values_diag == (1, 1, None, 1, 1, 0, None, None):
                return 'left_lower_spit_boundary'

            if values_diag == (1, 1, None, 1, 0, 1, None, None):
                return 'left_upper_spit_boundary'

            # boundary corners
            if values == (1, 1, None, None) or values == (1, 1, None, 5):
                return 'left_upper_boundary'

            if values == (1, None, 1, None) or values == (1, None, 1, 5):
                return 'right_upper_boundary'

            if values == (None, None, 1, 1) or values == (5, None, 1, 1):
                return 'right_lower_boundary'

            if values == (None, 1, None, 1) or values == (5, 1, None, 1):
```

```
286.                 return 'left_lower_boundary'
287.
288.             # corners
289.             values_temp = tuple([0 if x == 4 else x for x in values])
290.             values_diag_temp = tuple([0 if x == 4 else x for x in values_diag])
291.
292.             if values_temp == (1, 1, 0, 0):
293.                 return 'left_upper_corner'
294.
295.             if values_temp == (1, 0, 1, 0):
296.                 return 'right_upper_corner'
297.
298.             if values_temp == (0, 1, 0, 1):
299.                 return 'left_lower_corner'
300.
301.             if values_temp == (0, 0, 1, 1):
302.                 return 'right_lower_corner'
303.
304.             if values_diag_temp == (1, 1, 1, 1, 1, 0, 1, 1):
305.                 return 'left_upper_inside_corner'
306.
307.             if values_diag_temp == (1, 1, 1, 1, 1, 1, 0, 1):
308.                 return 'right_upper_inside_corner'
309.
310.             if values_diag_temp == (1, 1, 1, 1, 0, 1, 1, 1):
311.                 return 'left_lower_inside_corner'
312.
313.             if values_diag_temp == (1, 1, 1, 1, 1, 1, 1, 0):
314.                 return 'right_lower_inside_corner'
315.
316.             # edges
317.             values_temp = tuple([1 if x == 6 else x for x in [0 if x == 4 else x for x in value
    s]])
318.             if values_temp == (1, 1, 1, 0):
319.                 return 'upper_edge'
320.
321.             if values_temp == (1, 1, 0, 1):
322.                 return 'right_edge'
323.
324.             if values_temp == (0, 1, 1, 1):
325.                 return 'lower_edge'
326.
327.             if values_temp == (1, 0, 1, 1):
328.                 return 'left_edge'
329.
330.             return 'temp'
```

MULTIPLAYERSPRITES

```
1.  __author__ = 'Will Evans'
2.
3.  import sprites
4.  import pygame as pg
5.  import random
6.  from pathfinding import Manhattan as Search
7.
8.
9.  class ClientPacMan(sprites.PacMan):
10.     def __init__(self, resource_pack, maze, win_scale, client, client_id):
11.         """
12.         Client side Pac-
    Man sprite that takes input from another client or possibly the server (not from keyboard).
13.         :param resource_pack: Contains the path to the folder containing the skins for the sprite.

14.         :param maze: Two dimensional list representation of the maze.
```

```
        :param win_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
        :param client: Client object from networking module (controls the sending and receiving of player data.
        :param client_id: ClientID (comes client object, but is stored in multiplayer section).
        """

        super().__init__(resource_pack, maze, win_scale)
        self.client = client
        self.client_id = client_id

    def update(self, events):
        """
        Gets input from Client object.
        :param events: Needed to keep the same signature, but not used.
        :return: None.
        """

        if not self.dead:

            move = self.client.get_data(self.client_id, 'move')
            pos = self.client.get_data(self.client_id, 'pos')
            self.facing, self.skin = self.get_skin(move)
            if move is not None:
                self.skin_clock += 1
            else:
                if not self.sound_channel.get_busy():
                    self.sound_channel.play(self.sounds['siren.wav'])
            self.update_pos(pos)
        else:
            if self.death_animation():
                return None
        return self.dead

    def update_pos(self, pos):
        self.update_tile()

        self.x, self.y = pos

        self.x *= self.win_scale
        self.y *= self.win_scale

        self.skin_rect = self.skins['{}_{}.png'.format(self.facing, self.skin)].get_rect(center=(self.x, self.y))
        self.rect = pg.Rect(self.x - int(6 * self.win_scale),
                            self.y - int(6 * self.win_scale),
                            12 * self.win_scale,
                            12 * self.win_scale)


class ClientPlayerPacMan(ClientPacMan):
    def __init__(self, resource_pack, maze, win_scale, client, client_id):
        """
        Client side Pac-Man sprite that is controlled by keyboard inputs.
        :param resource_pack: Contains the path to the folder containing the skins for the sprite.
        :param maze: Two dimensional list representation of the maze.
        :param win_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
        :param client: Client object from networking module (controls the sending and receiving of player data.
        :param client_id: ClientID (comes client object, but is stored in multiplayer section).
        """

        super().__init__(resource_pack, maze, win_scale, client, client_id)

    def update(self, events):
        """
```

```
77.         Sends client move to server.
78.         :param events: Used to get keyboard inputs.
79.         :return: None.
80.         """
81.
82.         self.client.update_data('client_move', self.get_input(events))
83.         self.client.send_player_data()
84.
85.         return super().update(events)
86.
87.     def get_input(self, events):
88.         """
89.         Gets inputs from keyboard events.
90.         :param events: Used to get keyboard inputs.
91.         :return: Buffer move, so that if there isn't a keyboard input the last input will be retur
    ned.
92.         """
93.
94.         for event in events:
95.             if event.type == pg.KEYDOWN:
96.                 if event.key == pg.K_UP:
97.                     self.buffer_move = 'n'
98.                 elif event.key == pg.K_RIGHT:
99.                     self.buffer_move = 'e'
100.                    elif event.key == pg.K_DOWN:
101.                        self.buffer_move = 's'
102.                    elif event.key == pg.K_LEFT:
103.                        self.buffer_move = 'w'
104.
105.                return self.buffer_move
106.
107.
108.        class ClientGhost(sprites.Ghost):
109.            def __init__(self, resource_pack, position, target, maze, win_scale, level, client, cli
    ent_id):
110.                """
111.                Client side ghost sprite that takes input from another client or possibly the serve
    r (not from keyboard).
112.                :param resource_pack: Contains the path to the folder containing the skins for the
    sprite.
113.                :param position: Position that sprite should spawn in (needed as ghosts can have di
    fferent starting positions
114.                depending on skin.
115.                :param target: Pac-Man object
116.                :param maze: Two dimensional list representation of the maze.
117.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
118.                :param level: Level number (used if ghost is AI to determine difficulty).
119.                :param client: Client object from networking module (controls the sending and recei
    ving of player data.
120.                :param client_id: ClientID (comes client object, but is stored in multiplayer secti
    on).
121.                """
122.
123.                super().__init__(resource_pack, position, target, maze, win_scale, level)
124.
125.                self.client = client
126.                self.client_id = client_id
127.
128.            def update(self, events):
129.                """
130.                Gets input from Client object.
131.                :param events: Needed to keep the same signature, but not used.
132.                :return: None.
133.                """
134.
135.                move = self.client.get_data(self.client_id, 'move')
136.                pos = self.client.get_data(self.client_id, 'pos')
```

```
137.                self.check_collision()
138.                self.facing, self.skin = self.get_skin(move)
139.                self.update_tile()
140.                self.update_pos(pos)
141.                self.get_mode()
142.
143.            def update_pos(self, pos):
144.                self.x, self.y = pos
145.                self.x *= self.win_scale
146.                self.y *= self.win_scale
147.
148.                self.skin_rect = self.skins['{}_{}.png'.format(self.facing, self.skin)].get_rect(ce
    nter=(self.x, self.y))
149.                self.rect = pg.Rect(self.x - int(6 * self.win_scale),
150.                                    self.y - int(6 * self.win_scale),
151.                                    12 * self.win_scale,
152.                                    12 * self.win_scale)
153.
154.
155.        class ClientPlayerGhost(ClientGhost):
156.            def __init__(self, resource_pack, position, target, maze, win_scale, level, client, cli
    ent_id):
157.                """
158.                Client side Ghost sprite that is controlled by keyboard inputs.
159.                :param resource_pack: Contains the path to the folder containing the skins for the
    sprite.
160.                :param position: Position that sprite should spawn in (needed as ghosts can have di
    fferent starting positions
161.                depending on skin.
162.                :param target: Pac-Man object
163.                :param maze: Two dimensional list representation of the maze.
164.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
165.                :param level: Level number (used if ghost is AI to determine difficulty).
166.                :param client: Client object from networking module (controls the sending and recei
    ving of player data.
167.                :param client_id: ClientID (comes client object, but is stored in multiplayer secti
    on).
168.                """
169.
170.                super().__init__(resource_pack, position, target, maze, win_scale, level, client, c
    lient_id)
171.                self.client = client
172.                # change to spotlights
173.                spotlight_size = (864 * win_scale, 864 * win_scale)
174.                spotlight_path = os.path.join('resources', 'sprites', 'spotlights', '12x12.png')
175.                self.spotlight = pg.transform.scale(pg.image.load(spotlight_path), spotlight_size)
176.
177.            def update(self, events):
178.                """
179.                Sends client move to server.
180.                :param events: Used to get keyboard inputs.
181.                :return: None.
182.                """
183.
184.                self.client.update_data('client_move', self.get_input(events))
185.                self.client.send_player_data()
186.                super().update(events)
187.
188.            def display(self, win):
189.                """
190.                Blits sprite skin to window, and also a spotlight (after the maze, pellets and Pac-
    Man), but before the other
191.                ghosts and score.
192.                :param win: The current window, all objects must be blitted to this window to be di
    splayed.
193.                :return: None
```

```
                """

                win.blit(self.skins['{}_{}.png'.format(self.facing, self.skin)], self.skin_rect)
                win.blit(self.spotlight, self.spotlight.get_rect(center=self.skin_rect.center))

            def get_input(self, events):
                """
                Gets inputs from keyboard events.
                :param events: Used to get keyboard inputs.
                :return: Buffer move, so that if there isn't a keyboard input the last input will b
    e returned.
                """

                for event in events:
                    if event.type == pg.KEYDOWN:
                        if event.key == pg.K_UP:
                            self.buffer_move = 'n'
                        elif event.key == pg.K_RIGHT:
                            self.buffer_move = 'e'
                        elif event.key == pg.K_DOWN:
                            self.buffer_move = 's'
                        elif event.key == pg.K_LEFT:
                            self.buffer_move = 'w'

                return self.buffer_move


        # Server side Pac-Man
        class ServerPacMan(sprites.PacMan):
            def __init__(self, resource_pack, maze, win_scale, server, client_id):
                """
                Server side Pac-Man sprite controlled by another client.
                :param resource_pack: Contains the path to the folder containing the skins for the
    sprite.
                :param maze: Two dimensional list representation of the maze.
                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
                :param client: Client object from networking module (controls the sending and recei
    ving of player data.
                :param client_id: ClientID (comes client object, but is stored in multiplayer secti
    on).
                """

                super().__init__(resource_pack, maze, win_scale)
                self.server = server
                self.client_id = client_id

            def update(self, events):
                """
                Gets input from Server object, as the get_move method has been overridden below.
                :param events: Needed to keep the same signature, but not used.
                :return: None.
                """

                self.dead = super().update(events)

                pos = [coord / self.win_scale for coord in [self.x, self.y]]
                self.server.update_data(self.client_id, 'pos', pos)
                self.server.update_data(self.client_id, 'move', self.online_move)
                return self.dead

            def get_move(self, events):
                move = self.server.get_data(self.client_id, 'client_move')

                self.buffer_move = move

                return self.check_move(self.buffer_move)

```

```
257.            def update_score(self, score):
258.                self.server.update_data(self.client_id, 'score', score)
259.
260.
261.        # Server side ghost
262.        class ServerGhost(sprites.Ghost):
263.            def __init__(self, resource_pack, position, target, maze, win_scale, level, server, cli
    ent_id):
264.                """
265.                Server side ghost that is controlled by clients.
266.                :param resource_pack: Contains the path to the folder containing the skins for the
    sprite.
267.                :param position: Position that sprite should spawn in (needed as ghosts can have di
    fferent starting positions
268.                depending on skin.
269.                :param target: Pac-Man object
270.                :param maze: Two dimensional list representation of the maze.
271.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
272.                :param level: Level number (used if ghost is AI to determine difficulty).
273.                :param server: Server object from networking module (controls the sending and recei
    ving of player data.
274.                :param client_id: ClientID (comes client object, but is stored in multiplayer secti
    on).
275.                """
276.
277.                super().__init__(resource_pack, position, target, maze, win_scale, level)
278.                self.server = server
279.                self.client_id = client_id
280.                self.respawned = True
281.                self.buffer_move = self.facing
282.
283.            def update(self, events):
284.                """
285.                Validates client input (from server object) and sends back a valid move.
286.                :param events: Used to get keyboard events.
287.                :return: None
288.                """
289.
290.                super().update(events)
291.
292.                pos = [coord / self.win_scale for coord in [self.x, self.y]]
293.                self.server.update_data(self.client_id, 'pos', pos)
294.                self.server.update_data(self.client_id, 'move', self.online_move)
295.
296.            def get_move(self, events):
297.                """
298.                Gets move from server object and returns a validated move.
299.                :param events: Used to get keyboard inputs.
300.                :return: Valid move.
301.                """
302.
303.                if self.dead or self.respawned:
304.                    self.buffer_move = super().get_move(events)
305.                    return self.buffer_move
306.                else:
307.                    move = self.server.get_data(self.client_id, 'client_move')
308.                    if move is not None:
309.                        self.buffer_move = move
310.
311.                return self.check_move(self.buffer_move)
312.
313.            def add_points(self, points):
314.                """
315.                Updates points for that particular player based on interactions the sprite has serv
    er side.
316.                :param points: Points to be added to player's total.
317.                :return: None.
```

```
318.                """
319.
320.                score = self.server.get_data(self.client_id, 'score')
321.                self.server.update_data(self.client_id, 'score', score + points)
322.
323.
324.        class ServerPlayerPacMan(sprites.PacMan):
325.            def __init__(self, resource_pack, maze, win_scale, server):
326.                """
327.                Server side Pac-Man sprite controlled by keyboard inputs.
328.                :param resource_pack: Contains the path to the folder containing the skins for the
    sprite.
329.                :param maze: Two dimensional list representation of the maze.
330.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
331.                :param server: Server object from networking module (controls the sending and recei
    ving of player data.
332.                """
333.
334.                super().__init__(resource_pack, maze, win_scale)
335.                self.server = server
336.                self.client_id = 0
337.
338.            def update(self, events):
339.                """
340.                Sends move and pos to server object.
341.                :param events: Used to get keyboard events.
342.                :return: None.
343.                """
344.
345.                self.dead = super().update(events)
346.
347.                pos = [coord / self.win_scale for coord in [self.x, self.y]]
348.                self.server.update_data(self.client_id, 'move', self.online_move)
349.                self.server.update_data(self.client_id, 'pos', pos)
350.
351.                return self.dead
352.
353.            def update_score(self, score):
354.                self.server.update_data(self.client_id, 'score', score)
355.
356.
357.        class ServerPacManAI(sprites.PacMan):
358.            def __init__(self, resource_pack, maze, win_scale, server, client_id):
359.                """
360.                Server side Pac-Man AI. Makes random moves and sends to server object.
361.                :param resource_pack: Contains the path to the folder containing the skins for the
    sprite.
362.                :param maze: Two dimensional list representation of the maze.
363.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
364.                :param server: Server object from networking module (controls the sending and recei
    ving of player data.
365.                :param client_id: ClientID (comes client object, but is stored in multiplayer secti
    on).
366.                """
367.
368.                super().__init__(resource_pack, maze, win_scale)
369.                self.server = server
370.                self.client_id = client_id
371.
372.                # Search
373.                self.search = Search(maze.tile_map)
374.
375.                # Path finding
376.                self.path = self.get_path()
377.                self.next_coords = self.path[1]
378.
```

```
379.                self.next_x = 0
380.                self.next_y = 0
381.
382.                self.axis_change = 'x'
383.
384.            def update(self, events):
385.                """
386.                Updates server with move.
387.                :param events: Used to get keyboard events.
388.                :return: None.
389.                """
390.
391.                self.dead = super().update(events)
392.
393.                pos = [coord / self.win_scale for coord in [self.x, self.y]]
394.                self.server.update_data(self.client_id, 'move', self.online_move)
395.                self.server.update_data(self.client_id, 'pos', pos)
396.
397.                return self.dead
398.
399.            def get_move(self, events):
400.                """
401.                Gets move by working out the move needed to reach the next tile in the path that is worked out by randomly
402.                selecting a tile before hand.
403.                :param events: Used to get keyboard events.
404.                :return: Next move.
405.                """
406.
407.                if self.tile.pos[1] == 17:
408.                    if self.tile.pos[0] <= 5 or self.tile.pos[0] >= 22:
409.                        if self.tile.pos[0] == 5 or self.tile.pos[0] == 22:
410.                            self.next_coords = self.get_path()[1]
411.                        return self.facing
412.
413.                tile_x, tile_y = self.tile.pos
414.                x, y = self.next_coords
415.
416.                if x == tile_x and y == tile_y - 3:
417.                    try:
418.                        self.next_coords = self.get_path()[1]
419.                    except TypeError as e:
420.                        print(e)
421.                x, y = self.next_coords
422.                y += 3
423.
424.                x *= 12 * self.win_scale
425.                x += 6 * self.win_scale
426.
427.                y *= 12 * self.win_scale
428.                y += 6 * self.win_scale
429.
430.                self.next_x = x
431.                self.next_y = y
432.
433.                if self.axis_change == 'x':
434.                    if self.x < x and self.x - x < -self.win_scale:
435.                        return 'e'
436.                    elif self.x > x and self.x - x > self.win_scale:
437.                        return 'w'
438.                    else:
439.                        self.axis_change = 'y'
440.
441.                if self.axis_change == 'y':
442.                    if self.y < y and self.y - y < -self.win_scale:
443.                        return 's'
444.                    elif self.y > y and self.y - y > self.win_scale:
445.                        return 'n'
```

```
446.                     else:
447.                         self.axis_change = 'x'
448.
449.             def get_path(self):
450.                 """
451.                 Gets path to random pellet tile.
452.                 :return: Path
453.                 """
454.
455.                 start_tile = self.tile.pos
456.                 chosen_row = random.choice(self.maze.tiles[1:-1])
457.                 pellet_tiles = [tile for tile in chosen_row if tile.type == 'pellet']
458.                 target_tile = random.choice(pellet_tiles).pos
459.                 start_tile = (start_tile[0], start_tile[1] - 3)
460.                 target_tile = (target_tile[0], target_tile[1] - 3)
461.                 return self.search.astar(start_tile, target_tile, self.facing)
462.
463.             def update_score(self, score):
464.                 self.server.update_data(self.client_id, 'score', score)
465.
466.
467.         # Server side ghost controlled by the server player
468.         class ServerPlayerGhost(sprites.Ghost):
469.             def __init__(self, resource_pack, position, target, maze, win_scale, level, server):
470.                 """
471.                 Server side ghost sprite controlled by keyboard inputs.
472.                 :param resource_pack: Contains the path to the folder containing the skins for the
    sprite.
473.                 :param position: Position that sprite should spawn in (needed as ghosts can have di
    fferent starting positions
474.                 depending on skin.
475.                 :param target: Pac-Man object
476.                 :param maze: Two dimensional list representation of the maze.
477.                 :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
478.                 :param level: Level number (used if ghost is AI to determine difficulty).
479.                 :param server: Server object from networking module (controls the sending and recei
    ving of player data.
480.                 """
481.
482.                 super().__init__(resource_pack, position, target, maze, win_scale, level)
483.                 self.server = server
484.                 self.client_id = 0
485.                 self.respawned = True
486.                 self.buffer_move = self.facing
487.                 self.spotlight = pg.transform.scale(pg.image.load('resources\\sprites\\spotlights\\
    12x12.png'),
488.                                                     (864 * win_scale, 864 * win_scale))
489.
490.             def update(self, events):
491.                 """
492.                 Gets move from keyboard inputs and then sends to server object.
493.                 :param events: Used to get keyboard events.
494.                 :return: None.
495.                 """
496.                 super().update(events)
497.
498.                 pos = [coord / self.win_scale for coord in [self.x, self.y]]
499.                 self.server.update_data(self.client_id, 'pos', pos)
500.                 self.server.update_data(self.client_id, 'move', self.online_move)
501.
502.             def display(self, win):
503.                 win.blit(self.skins['{}_{}.png'.format(self.facing, self.skin)], self.skin_rect)
504.                 win.blit(self.spotlight, self.spotlight.get_rect(center=self.skin_rect.center))
505.
506.             def get_move(self, events):
507.                 """
508.                 Gets move using keyboard events.
```

```
509.                :param events: Used to get keyboard events.
510.                :return: Validated move.
511.                """
512.
513.                if self.dead or self.respawned:
514.                    self.buffer_move = super().get_move(events)
515.                    return self.buffer_move
516.                else:
517.                    move = self.get_input(events)
518.                    if move is not None:
519.                        self.buffer_move = move
520.
521.                return self.check_move(self.buffer_move)
522.
523.            def get_input(self, events):
524.                for event in events:
525.                    if event.type == pg.KEYDOWN:
526.                        if event.key == pg.K_UP:
527.                            self.buffer_move = 'n'
528.                        elif event.key == pg.K_RIGHT:
529.                            self.buffer_move = 'e'
530.                        elif event.key == pg.K_DOWN:
531.                            self.buffer_move = 's'
532.                        elif event.key == pg.K_LEFT:
533.                            self.buffer_move = 'w'
534.
535.                return self.buffer_move
536.
537.            def add_points(self, points):
538.                score = self.server.get_data(self.client_id, 'score')
539.                self.server.update_data(self.client_id, 'score', score + points)
540.
541.
542.        class ServerBlinky(sprites.Blinky):
543.            def __init__(self, resource_pack, target, maze, win_scale, level, server, client_id):
544.
545.                super().__init__(resource_pack, target, maze, win_scale, level)
546.                self.server = server
547.                self.client_id = client_id
548.
549.            def update(self, events):
550.                super().update(events)
551.                self.server.update_data(self.client_id, 'move', self.facing)
552.                pos = [coord / self.win_scale for coord in [self.x, self.y]]
553.                self.server.update_data(self.client_id, 'pos', pos)
554.
555.            def add_points(self, points):
556.                score = self.server.get_data(self.client_id, 'score')
557.                self.server.update_data(self.client_id, 'score', score + points)
558.
559.
560.        class ServerPinky(sprites.Pinky):
561.            def __init__(self, resource_pack, target, maze, win_scale, level, server, client_id):
562.                super().__init__(resource_pack, target, maze, win_scale, level)
563.                self.server = server
564.                self.client_id = client_id
565.
566.            def update(self, events):
567.                super().update(events)
568.                self.server.update_data(self.client_id, 'move', self.facing)
569.                pos = [coord / self.win_scale for coord in [self.x, self.y]]
570.                self.server.update_data(self.client_id, 'pos', pos)
571.
572.            def add_points(self, points):
573.                score = self.server.get_data(self.client_id, 'score')
574.                self.server.update_data(self.client_id, 'score', score + points)
575.
576.
```

```
577.        class ServerInky(sprites.Inky):
578.            def __init__(self, resource_pack, target, maze, win_scale, level, blinky, server, clien
    t_id):
579.                super().__init__(resource_pack, target, maze, win_scale, level, blinky)
580.                self.server = server
581.                self.client_id = client_id
582.
583.            def update(self, events):
584.                super().update(events)
585.                self.server.update_data(self.client_id, 'move', self.facing)
586.                pos = [coord / self.win_scale for coord in [self.x, self.y]]
587.                self.server.update_data(self.client_id, 'pos', pos)
588.
589.            def add_points(self, points):
590.                score = self.server.get_data(self.client_id, 'score')
591.                self.server.update_data(self.client_id, 'score', score + points)
592.
593.
594.        class ServerClyde(sprites.Clyde):
595.            def __init__(self, resource_pack, target, maze, win_scale, level, server, client_id):
596.                super().__init__(resource_pack, target, maze, win_scale, level)
597.                self.server = server
598.                self.client_id = client_id
599.
600.            def update(self, events):
601.                super().update(events)
602.                self.server.update_data(self.client_id, 'move', self.facing)
603.                pos = [coord / self.win_scale for coord in [self.x, self.y]]
604.                self.server.update_data(self.client_id, 'pos', pos)
605.
606.            def add_points(self, points):
607.                score = self.server.get_data(self.client_id, 'score')
608.                self.server.update_data(self.client_id, 'score', score + points)
```

NETWORKING

```
321.        __author__ = 'Will Evans'
322.
323.        import threading
324.        import socket
325.        import json
326.        import copy
327.        import time
328.
329.
330.        class Connection:
331.            def __init__(self, user_ip, conn, user_id, players):
332.                """
333.                Class for each connection the server has with a client. Controls all information go
    ing from server to client.
334.                :param user_ip: IP for user.
335.                :param conn: Socket connection used to send and receive data from client.
336.                :param user_id: A number from 1-4, used to match connection with avatar.
337.                :param players: List of all players and their information, used to update client's
    screen.
338.                """
339.
340.                self.connected = True
341.                self.__player_data = {
342.
343.                    'ready': None,
344.                    'client_move': None,
345.
346.                }
347.
```

```
348.                self.__connected = False
349.                self.__IP = user_ip
350.                self.id = user_id
351.                self.__PORT = 50007
352.                self.__conn = conn
353.
354.                # Essential trade of info
355.                self.send({
356.
357.                            'client_id': user_id,
358.                            'players': players
359.
360.                           })
361.
362.                self.__player_data['name'] = self.receive()['name']
363.
364.                threading.Thread(target=self.update).start()
365.
366.            def update(self):
367.                """
368.                Runs once every frame, updates each item in player_data dictionary. Only used at th
    e beginning of the connection
369.                to share basic information between the client and the host.
370.                :return: None
371.                """
372.
373.                while self.connected:
374.                    data = self.receive()
375.                    if data is not None:
376.                        for attribute, value in data.items():
377.                            self.__player_data[attribute] = value
378.
379.            def receive(self):
380.                """
381.                Receives data from client (player_data).
382.                :return: Data in string form (not bytes).
383.                """
384.
385.                try:
386.                    data = self.__conn.recv(4096)
387.                    return json.loads(data)
388.                except Exception as e:
389.                    print("disconnected: {}".format(e))
390.
391.            def send(self, data):
392.                """
393.                Converts data into bytes and sends to client. This is usually the 2D dictionary of
    player info.
394.                :param data: Data that is to be sent (player_data).
395.                :return: None
396.                """
397.
398.                if self.connected:
399.                    data = json.dumps(data)
400.                    data = bytes(data, 'utf-8')
401.                    try:
402.                        self.__conn.sendall(data)
403.                    except ConnectionResetError:
404.                        print("disconnected")
405.                        self.connected = False
406.
407.            def get_player_data(self):
408.                return self.__player_data
409.
410.            def get_id(self):
411.                return self.id
412.
413.            def close(self):
```

```
414.                """
415.                When a client quits the connection must be closed. Closes the socket connection and
    marks the object as not
416.                connected.
417.                :return: None
418.                """
419.
420.                self.__conn.close()
421.                self.connected = False
422.
423.
424.        class Server:  # Instantiates whenever a user clicks create game.
425.            def __init__(self, name):
426.                """
427.                Class for server that controls the sending and receiving of game data for each play
    er between the host and
428.                all clients connected.
429.                :param name: Host's name. Comes form database based on user's sign in details.
430.                """
431.
432.                self.__run = True
433.                self.test_count = 0
434.                self.__players_template = {
435.
436.                    0:
437.                        {
438.                            'name':     '{} [host]'.format(name),
439.                            'skin':     'pac-man',
440.                            'score':    0,
441.                            'ready':    None,
442.                            'pos':      [167, 318],
443.                            'move':     None,
444.                            'countdown': None,
445.                            'start':    False,
446.                            'end':      True,
447.                            'place':    None,
448.                            'finished': False
449.                        },
450.
451.                    1:
452.                        {
453.                            'name':     None,
454.                            'skin':     'blinky',
455.                            'score':    0,
456.                            'ready':    None,
457.                            'pos':      [168, 176],
458.                            'move':     None,
459.                            'client_move': None,
460.                            'place': None
461.                        },
462.
463.                    2:
464.                        {
465.                            'name':     None,
466.                            'skin':     'pinky',
467.                            'score':    0,
468.                            'ready':    None,
469.                            'pos':      [168, 214],
470.                            'move':     None,
471.                            'client_move': None,
472.                            'place': None
473.                        },
474.
475.                    3:
476.                        {
477.                            'name':     None,
478.                            'skin':     'inky',
479.                            'score':    0,
```

```
                            'ready':    None,
                            'pos':      [144, 214],
                            'move':     None,
                            'client_move': None,
                            'place': None
                        },
                    4:
                        {
                            'name':     None,
                            'skin':     'clyde',
                            'score':    0,
                            'ready':    None,
                            'pos':      [192, 214],
                            'move':     None,
                            'client_move': None,
                            'place': None
                        }
                    }

                self.__players = copy.deepcopy(self.__players_template)
                self.__IP = self.get_ip()
                self.__port = 50007
                self.__connections = []
                self.__s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                self.__s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
                self.__s.bind((self.__IP, self.__port))

                self.has_ai = False

                self.searching_for_clients = True
                threading.Thread(target=self.connect).start()
                threading.Thread(target=self.receive).start()
                threading.Thread(target=self.check_connections).start()

            def connect(self):
                """
                Before the game starts the this method will listen for new connections, and then cr
    eate a connection when one
                is received.
                :return: None
                """

                while self.searching_for_clients and self.__run:
                    self.__s.listen(1)
                    if self.searching_for_clients:
                        try:
                            conn, addr = self.__s.accept()
                            available_ids = [k for k, v in self.__players.items() if v['name'] is N
    one]
                            client_id = available_ids[0]
                            connection = Connection(addr[0], conn, client_id, self.__players)
                            self.__connections.append(connection)
                        except Exception as e:
                            print("Connect: {}".format(e))

            def receive(self):
                """
                This method updates the player data based on information sent my each client to eac
    h of the connections. Updates
                twice a second to account for any syncing issues between clients and server.
                :return: None
                """

                while self.__run:
                    try:
                        for connection in self.__connections:
                            data = connection.get_player_data()
                            for attribute, value in data.items():
```

```
                                self.__players[connection.get_id()][attribute] = value
                    except Exception as e:
                        print(e)
                    time.sleep(1/120)

            def update_data(self, client_id, key, value):
                self.__players[client_id][key] = value

            def send_data(self):
                for connection in self.__connections:
                    connection.send(self.__players)

            def check_connections(self):
                """
                Runs while the server is running. Checks every connection to see if it is still the
    re, prevents errors with
                connections.
                :return: None
                """

                while self.__run:
                    for connection in self.__connections:
                        if not connection.connected:
                            connection.close()
                            self.__players[connection.get_id()] = self.__players_template[connectio
    n.get_id()].copy()
                            self.__connections.remove(connection)

                    # Limits the number of times the thread can run to save power
                    # 1/120 has been chosen to ensure it is run at least once between frames
                    time.sleep(1/120)

            def get_data(self, client_id, type):
                return self.__players[client_id][type]

            def get_players(self):
                return self.__players

            def get_ip(self):
                return socket.gethostbyname(socket.gethostname())

            def get_client_id(self):
                return 0

            def swap(self, client_id):
                """
                Whichever client_id is passed into this method will become the next Pac-
    Man. This happens when a ghost catches
                Pac-
    Man and is needed to swap the skins and start position of players around when they become Pac-
    Man.
                :param client_id: ClientID of player that is becoming Pac-Man
                :return: None
                """

                pac_man_id = [id for id, data in self.__players.items() if data['skin'] == 'pac-
    man'][0]

                pac_man_skin = "{}".format(self.__players[pac_man_id]['skin'])
                client_skin = "{}".format(self.__players[client_id]['skin'])

                self.__players[pac_man_id]['skin'] = client_skin
                self.__players[client_id]['skin'] = pac_man_skin

            def reset(self):
                """
                At the end of each round the positions of each sprite need to be reset according to
     the player_template in order
```

```
606.                for the sprites to spawn in the correct place in the following round.
607.                :return: None
608.                """
609.
610.                for client_id, client_data in self.__players.items():
611.                    og_player_data = [og_client_data
612.                                      for og_client_data in self.__players_template.values()
613.                                      if og_client_data['skin'] == client_data['skin']][0]
614.
615.                    client_data['pos'] = og_player_data['pos'][::]
616.
617.            def add_ai(self):
618.                """
619.                Changes name of all available ghosts into 'AI', which will mean they become control
    led by AI in the game. This
620.                method is run when the game countdown is started.
621.                :return: None
622.                """
623.
624.                for client_data in self.__players.values():
625.                    if client_data['name'] is None:
626.                        client_data['name'] = 'AI'
627.                self.has_ai = True
628.
629.            def remove_ai(self):
630.                """
631.                This changes all AI ghosts back to None when the game countdown is stopped (to allo
    w for more players to join).
632.                :return:
633.                """
634.
635.                for client_data in self.__players.values():
636.                    if client_data['name'] == 'AI':
637.                        client_data['name'] = None
638.                self.has_ai = False
639.
640.            def quit(self):
641.                """
642.                Correctly adjusts attributes so that all threads and connections terminate when the
     server is no longer needed.
643.                :return: None
644.                """
645.
646.                self.searching_for_clients = False
647.                self.__run = False
648.                for connection in self.__connections:
649.                    connection.close()
650.                self.__s.close()
651.
652.
653.        class Client:
654.            def __init__(self, host_ip, name):
655.                """
656.                Runs on clients when the user selects join game and enters a GameID.
657.                :param host_ip: GameID that the user enters. It is actually the host's local IPV4 a
    ddress.
658.                :param name: Name of client according to the Users table in the database.
659.                """
660.
661.                self.__host_ip = host_ip
662.                self.__name = name
663.                self.connected = False
664.                self.connection_failed = None
665.
666.                self.__player_data = {
667.
668.                                    'ready':        None,
669.                                    'client_move':  None,
```

```
                                }

            self.__port = 50007
            self.__s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

            try:
                self.__s.connect((self.__host_ip, self.__port))
                self.connection_failed = False
            except (ConnectionRefusedError, socket.gaierror, TypeError):
                self.connection_failed = True

            if not self.connection_failed:
                # first share of essential data
                init_data = self.receive()
                if init_data is not None:
                    self.__client_id = init_data['client_id']
                    self.__players = init_data['players']
                    self.send({'name': self.__name})

                    self.connected = True
                    threading.Thread(target=self.update).start()

        def send(self, data):
            data = json.dumps(data)
            data = bytes(data, 'utf-8')
            try:
                self.__s.sendall(data)
            except OSError:
                print("disconnected")

        def receive(self):
            try:
                data = self.__s.recv(1024)
                return json.loads(data)
            except ConnectionResetError as e:
                self.connected = False
                print(e)
            except WindowsError as e:
                self.connected = False
                print(e)
            except Exception as e:
                print(e)

        def update(self):
            """
            Receives data from server and sets equal to players.
            :return: None
            """

            while self.connected:
                data = self.receive()
                if data is not None:
                    self.__players = data

        def update_data(self, key, value):
            self.__player_data[key] = value

        def get_data(self, client_id, type):
            return self.get_players()[client_id][type]

        def send_player_data(self):
            self.send(self.__player_data)

        def get_players(self):
            """
            Corrects integer keys in players (as they are saved as strings when converted to an
    d from bytes) and returns.
```

```
737.                :return: Dictionary: players
738.                """
739.                return {int(k): v for k, v in self.__players.items()}
740.
741.            def get_client_id(self):
742.                return self.__client_id  # Ask here if client id is None
743.
744.            def end(self):
745.                self.connected = False
746.                self.__s.close()
747.
748.
749.        if __name__ == "__main__":
750.            pass
```

GUI

```
1.  __author__ = 'Will Evans'
2.  import pygame as pg
3.  import os.path
4.  import sprites
5.
6.
7.  class Word:
8.      def __init__(self, content, pos, colour, font_size, win_scale, italic=False, bold=False, centr
    e=False, left=False):
9.          """
10.         Class used to put a word on the screen.
11.         :param content: String: content of the word.
12.         :param pos: (x, y) position of word.
13.         :param colour: (r, g, b) value of colour.
14.         :param font_size: Font size.
15.         :param win_scale: Window Scale (How large the window is - must be multiplied by all size r
    elated variables).
16.         :param italic: Boolean: italics or not.
17.         :param bold: Boolean: bold or not.
18.         :param centre: Boolean: centre or not.
19.         :param left: Boolean: align left or not.
20.         """
21.
22.         self.content = content
23.
24.         x, y = pos
25.         self.x = x * win_scale
26.         self.y = y * win_scale
27.
28.         self.colour = colour
29.         self.centre = centre
30.         self.left = left
31.         self.__letters = []
32.
33.         font_path = os.path.join('resources', 'fonts', 'ARCADECLASSIC.TTF')
34.         self._font = pg.font.Font(font_path, font_size * win_scale)
35.         self._font.set_italic(italic)
36.         self._font.set_bold(bold)
37.
38.         if self.content is not None:
39.             self.render()
40.
41.     def display(self, win):
42.         """
43.         Blits the rendered font to the screen as per the rect.
44.         :param win: The current window, all objects must be blitted to this window to be displayed
    .
45.         :return: None
```

```
        """

        if self.content is not None:
            win.blit(self._rendered_text, self._text_rect)

    def render(self):
        """
        Renders font. Takes the content and colour and converts this into a font object. Then a rect object is created
        based on the position and alignment instructions.
        :return: None
        """

        self._rendered_text = self._font.render(self.content, True, self.colour)

        if self.centre:
            self._text_rect = self._rendered_text.get_rect(center=(self.x, self.y))
        elif self.left:
            self._text_rect = self._rendered_text.get_rect(midleft=(self.x, self.y))
        else:
            self._text_rect = self._rendered_text.get_rect(midright=(self.x, self.y))


class LiveWord:
    def __init__(self, content, y, font_size, win_scale, highlight_colour=(255, 255, 255)):
        """
        Live words have the ability to highlight when selected by the user. Seperate to word class as live
        words must be rendered letter by letter so that when the highlighted layer is shown, it correctly covers each
        individual letter.
        :param content: String: content of the word.
        :param y: y coordinate of word (do not require x as they are automatically centred).
        :param font_size: Integer: Font size
        :param win_scale: Integer: Window Scale (How large the window is - must be multiplied by all size related
        variables).
        :param highlight_colour: (r, g, b): default is set to white but can be any colour.
        """

        self.program = content
        self.content = list(content)
        self.__letters = []
        self.__react = False
        self.__letter_spacing = font_size * 7/12

        for num, letter in enumerate(content):
            x = 173 + ((- len(content) / 2) + num) * self.__letter_spacing
            self.__letters.append(LiveLetter(letter, x, y, font_size, win_scale, highlight_colour))

    def react(self):
        self.__react = True

    def display(self, win):
        """
        Displays each letter in the word, sets react to False so that if the mouse stops colliding with the word,
        then the word will stop displaying as highlighted.
        :param win: The current window, all objects must be blitted to this window to be displayed.
                :type win: Surface.
                :return: None.
                """

                for letter in self.__letters:
                    letter.display(self.__react, win)
                self.__react = False
```

```
107.
108.            def check_mouse(self, x, y):
109.                """
110.                Returns true if the x, y coordinates supplied are colliding with any of the letters
    ' rects.
111.                :param x: x coord of mouse.
112.                :type x: Integer/
113.                :param y: y coord of mouse.
114.                :type y: Integer/
115.                :return: Boolean: True if mouse is colliding with letters, false if mouse is not.
116.                """
117.
118.                for letter in self.__letters:
119.                    if letter.check_mouse(x, y):
120.                        return True
121.                return False
122.
123.            def check_click(self, x, y):
124.                """
125.                Checks each letter to see if any of them have been clicked by the mouse.
126.                :param x: x coord of mouse that has been clicked.
127.                :type x: Integer.
128.                :param y: y coord of mouse that has been clicked.
129.                :type y: Integer.
130.                :return: Returns true if any of the letters have been clicked on else false.
131.                """
132.
133.                for letter in self.__letters:
134.                    if letter.check_mouse(x, y):
135.                        return True
136.
137.            def get_program(self):
138.                return self.program
139.
140.
141.        class LiveLetter:
142.            def __init__(self, letter, x, y, font_size, win_scale, highlight_colour):
143.                """
144.                Live letters are used by the live words class. They control one letter each.
145.                :param letter: The letter that will be rendered.
146.                :type letter: String.
147.                :param x: x coord of letter position.
148.                :type x: Integer.
149.                :param y: y coord of letter position.
150.                :type y: Integer.
151.                :param font_size: Font size.
152.                :type font_size: Integer.
153.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
    size related variables).
154.                :type win_scale: Integer.
155.                :param highlight_colour: (r, g, b) colour of the highlighted part of the letter.
156.                :type highlight_colour: Tuple.
157.                """
158.
159.                self.__font_size = font_size * win_scale
160.                self.x = x * win_scale
161.                self.y = y * win_scale
162.
163.                font_path = os.path.join('resources', 'fonts', 'ARCADECLASSIC.TTF')
164.
165.                self.__font = pg.font.Font(font_path, self.__font_size)
166.                self.__rendered_text = self.__font.render(letter, True, (255, 255, 30))
167.                self.__text_rect = self.__rendered_text.get_rect(center=(self.x, self.y))
168.
169.                self.__outline1 = pg.font.Font(font_path, int(self.__font_size * 54 / 50))
170.                self.__rendered_outline1 = self.__outline1.render(letter, True, pg.Color('black'))

171.                self.__outline1_rect = self.__rendered_outline1.get_rect(center=(self.x, self.y))
```

```
172.
173.                 self.__outline = pg.font.Font(font_path, int(self.__font_size * 62 / 50))
174.                 self.__rendered_outline = self.__outline.render(letter, True, highlight_colour)
175.                 self.__outline_rect = self.__rendered_outline.get_rect(center=(self.x, self.y))
176.
177.             def display(self, react, win):
178.                 """
179.                 Blits the letter object ot the screen. If react is True then a black outline and an
    other outline will be blitted
180.                 behind to highlight the letter.
181.                 :param react: Whether or not the letter will be highlighted or not.
182.                 :type react: Boolean.
183.                 :param win: The current window, all objects must be blitted to this window to be di
    splayed.
184.                 :type win: Surface.
185.                 :return: None.
186.                 """
187.
188.                 if react:
189.                     win.blit(self.__rendered_outline, self.__outline_rect)
190.                     win.blit(self.__rendered_outline1, self.__outline1_rect)
191.                     win.blit(self.__rendered_text, self.__text_rect)
192.                 else:
193.                     win.blit(self.__rendered_text, self.__text_rect)
194.
195.             def check_mouse(self, x, y):
196.                 """
197.                 Checks whether the x, y coords are colliding with the letter.
198.                 :param x: x coord of mouse.
199.                 :type x: Integer.
200.                 :param y: y coord of mouse.
201.                 :type y: Integer.
202.                 :return: True if the x, y coordinates are colliding with the text rect.
203.                 """
204.
205.                 return self.__text_rect.collidepoint(x, y)
206.
207.
208.         class ScrollingWord:
209.             def __init__(self, content, pos, colour, font_size, win_scale, frame_cap=1):
210.                 """
211.                 Scrolling words uses the word class (constantly re renders it) to give the appearan
    ce of scrolling / revealing
212.                 text.
213.                 :param content: String: content of the word.
214.                 :param pos: (x, y) position of word.
215.                 :param colour: (r, g, b) value of colour.
216.                 :param font_size: Font size.
217.                 :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
218.                 """
219.
220.                 self.content = content
221.                 self.content_displayed = ''
222.                 self.characters_displayed_num = 0
223.                 self.finished = False
224.
225.                 self.rendered_font = Word(self.content_displayed, pos, colour, font_size, win_scale
    , False, False, False, True)
226.
227.                 self.frame_count = 0
228.                 self.frame_cap = frame_cap
229.
230.             def update(self, events):
231.                 self.frame_count += 1
232.                 if self.frame_count == self.frame_cap:
233.                     self.frame_count = 0
234.
```

```
235.                     if self.content == self.content_displayed:
236.                         self.finished = True
237.                     else:
238.                         self.content_displayed += self.content[self.characters_displayed_num]
239.                         self.characters_displayed_num += 1
240.                         self.rendered_font.content = self.content_displayed
241.                         self.rendered_font.render()
242.
243.             def display(self, win):
244.                 self.rendered_font.display(win)
245.
246.             def render_all(self):
247.                 self.content_displayed = self.content
248.                 self.characters_displayed_num = len(self.content)
249.                 self.rendered_font.content = self.content_displayed
250.                 self.rendered_font.render()
251.
252.
253.         class TutorialTextBox:
254.             def __init__(self, content, colour, win_scale, add_mspacman=False):
255.                 """
256.                 Uses the scrolling word, and box class to display scrolling text that can go over o
    nto many lines
257.                 all in a neat and tidy box.
258.                 :param content: The boxes text content.
259.                 :param colour: The colour of the text in the box.
260.                 :param add_mspacman: Decides whether the text box should also include a sprite of m
    spacman.
261.                 :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
262.                 """
263.
264.                 # Essential
265.                 self.win_scale = win_scale
266.                 self.paused = False
267.                 self.finished = False
268.
269.                 self.add_mspacman = add_mspacman
270.
271.                 # Words
272.                 self.content = [f'{word}  ' for word in content.split(' ')]
273.                 self.content_remaining = self.content
274.                 self.colour = colour
275.                 self.win_scale = win_scale
276.                 self.font_size = 20
277.                 self.x_prespacing = 15 if not self.add_mspacman else 80
278.                 self.max_length = 340 - self.x_prespacing
279.
280.                 # This is a lovely trick to calculate how long the text will be (so that we know wh
    en to use a newline). By
281.                 # rendering the font here we can use this object to get the width.
282.                 font_path = os.path.join('resources', 'fonts', 'ARCADECLASSIC.TTF')
283.                 self.font = pg.font.Font(font_path, self.font_size)
284.
285.                 self.boxes = []
286.
287.                 while len(self.content_remaining) > 0:
288.                     self.render_box()
289.
290.                 self.active_box_index = 0
291.                 self.active_box = self.boxes[self.active_box_index]
292.
293.                 # Box
294.                 self.box = Box((7, 360), (321, 60), (230, 230, 230), 2, win_scale)
295.
296.                 # Transparent Win
297.                 self.transparent_win = pg.Surface((321 * win_scale, 60 * win_scale))
298.                 self.transparent_win.set_alpha(225)
```

```
299.                self.transparent_win.fill((161, 161, 161))
300.
301.                mspacman_skin_paths = [os.path.join('resources', 'sprites', 'ms.pac-
    man', f'{num}.png') for num in (0, 1)]
302.                mspacman_skins = [pg.transform.scale(pg.image.load(path), ((56 * win_scale), (56 *
    win_scale)))
303.                                  for path in mspacman_skin_paths]
304.                mspacman_rect = mspacman_skins[0].get_rect(center=(40 * win_scale, 370 * win_scale)
    )
305.                self.mspacman = sprites.StaticSprite(mspacman_skins, mspacman_rect)
306.
307.            def update(self, events):
308.
309.                space_pressed = False
310.
311.                # Words
312.                if len(self.boxes) > 0:
313.                    for line in self.active_box:
314.                        if line.finished:
315.                            continue
316.                        line.update(events)
317.                        break
318.
319.                self.paused = all([line.finished for line in self.active_box])
320.
321.                for event in events:
322.                    if event.type == pg.KEYDOWN:
323.                        if event.key == pg.K_SPACE:
324.                            space_pressed = True
325.
326.                if self.paused and not self.finished and space_pressed:
327.                    self.active_box_index += 1
328.                    if self.active_box_index == len(self.boxes):
329.                        self.finished = True
330.                    else:
331.                        self.active_box = self.boxes[self.active_box_index]
332.                elif space_pressed:
333.                    for line in self.active_box:
334.                        line.render_all()
335.
336.                if not self.paused and self.add_mspacman:
337.                    self.mspacman.update(events)
338.
339.            def display(self, win):
340.
341.                # Transparent Win
342.                win.blit(self.transparent_win, (7 * self.win_scale, 360 * self.win_scale))
343.
344.                # Words
345.
346.                for line in self.active_box:
347.                    line.display(win)
348.
349.                # Box
350.                self.box.display(win)
351.
352.                # MsPacMan
353.                if self.add_mspacman:
354.                    self.mspacman.display(win)
355.
356.            def render_box(self):
357.                content_buffer = ''
358.                line_num = 0
359.                lines = []
360.
361.                for word in self.content_remaining:
362.                    content_buffer += word
363.                    self.content_remaining = self.content_remaining[1:]
```

```
364.
365.                     if len(self.content_remaining) == 0:
366.                         lines.append(
367.                             ScrollingWord(content_buffer, (self.x_prespacing, 375 + 15 * line_num),
    self.colour, self.font_size,
368.                                           self.win_scale, 2))
369.
370.                         self.boxes.append(lines)
371.                         break
372.
373.                     current_length = self.font.size(content_buffer)[0] + self.font.size(self.conten
    t_remaining[0])[0]
374.                     ellipsis_exit = word.replace(' ', '')[:-4:-1] == '...'
375.                     if current_length >= self.max_length or ellipsis_exit:
376.                         lines.append(
377.                             ScrollingWord(content_buffer,
378.                                           (self.x_prespacing,
379.                                            375 + 15 * line_num),
380.                                           self.colour,
381.                                           self.font_size,
382.                                           self.win_scale,
383.                                           2)
384.                                 )
385.
386.                         line_num += 1
387.                         content_buffer = ''
388.
389.                     if line_num == 3 or ellipsis_exit:
390.                         self.boxes.append(lines)
391.                         break
392.
393.
394.         class Box:
395.             def __init__(self, pos, dimensions, colour, width, win_scale, centre=False):
396.                 """
397.                 Boxes are simply rectangles. They are capable of changing colour when a colour when
    a mouse is colliding with it
398.                 however this requires external support form code at the moment. Boxes are often agg
    regated. For example they are
399.                 used in input boxes etc.
400.                 :param pos: (x, y) Contains the x and y coordinates of the box.
401.                 :type pos: Tuple.
402.                 :param dimensions: (w, h) Contains width and height of the box.
403.                 :type dimensions: Tuple.
404.                 :param colour: (r, g, b) colour of box.
405.                 :type colour: Tuple.
406.                 :param width: Width of the box line.
407.                 :type width: Integer.
408.                 :param win_scale: Window Scale (How large the window is - must be multiplied by all
    size related variables).
409.                 :type: Integer.
410.                 :param centre: If True the x, y values will become the centre of the box. Otherwise
    the top left is used by
411.                 default.
412.                 :type centre: Boolean.
413.                 """
414.
415.                 x, y = pos
416.                 w, h = dimensions
417.
418.                 if centre:
419.                     self.rect = pg.Rect((x-1/2*w) * win_scale, (y-
    1/2*h) * win_scale, w * win_scale, h * win_scale)
420.                 else:
421.                     self.rect = pg.Rect(x * win_scale, y * win_scale, w * win_scale, h * win_scale)

422.                 self.colour = colour
423.                 self.width = width * win_scale
```

```
424.
425.           def display(self, win):
426.               """
427.               Blits the box to the screen.
428.               :param win: The current window, all objects must be blitted to this window to be di
   splayed.
429.               :type win: Surface.
430.               :return: None.
431.               """
432.
433.               pg.draw.rect(win, self.colour, self.rect, self.width)
434.
435.           def check_mouse(self, x, y):
436.               return self.rect.collidepoint(x, y)
437.
438.
439.       class InputBox:
440.           def __init__(self, x, y, w, h, font_size, win_scale, name='', interactive=True, centre=
   False, private=False,
441.                        max_length=14):
442.               """
443.               Input boxes use a box object and a word object (or will do). They are used to allow
    the user to allow the input
444.               of words from the user.
445.               :param x: x coord of box.
446.               :type x: Integer.
447.               :param y: y coord of box.
448.               :type: y: Integer.
449.               :param w: Width of box in pixels.
450.               :type w: Integer.
451.               :param h: Height of box in pixels.
452.               :type: h: Integer.
453.               :param font_size: Font size.
454.               :type font_size: Integer.
455.               :param win_scale: Window Scale (How large the window is - must be multiplied by all
    size related variables).
456.               :type win_scale: Integer.
457.               :param name: This is the string that will display if the text box is empty at any t
   ime.
458.               :type name: String.
459.               :param interactive: Whether or not the box should respond to any user input. If fal
   se the input box becomes a
460.               text box.
461.               :type interactive: Boolean.
462.               :param centre: If True the x, y values will become the centre of the box. Otherwise
    the top left is used by
463.               default.
464.               :type centre: Boolean.
465.               :param private: If True the text in the box will be asterisks. This is useful for p
   assword input boxes.
466.               :type private: Boolean.
467.               :param max_length: The maximum amount of characters the box can hold before it no l
   onger accepts any more
468.               characters from the user.
469.               :type max_length: Integer.
470.               """
471.
472.               # Essential
473.               self.win_scale = win_scale
474.
475.               self.interactive = interactive
476.               self.active = False
477.               self.tab_active = False
478.               self.private = private
479.               self.text_entered = False
480.               self.max_length = max_length
481.
482.               self.text_surface_width_og = w
```

```
483.
484.                 if centre:
485.                     self.text_box_rect = pg.Rect((x - 1/2*w) * win_scale, (y - 1/2*h) * win_scale,
    w * win_scale, h * win_scale)
486.                 else:
487.                     self.text_box_rect = pg.Rect(x * win_scale, y * win_scale, w * win_scale, h * w
    in_scale)
488.
489.                 self.text_box_colour = (161, 161, 161)
490.                 self.text_box_highlighted_colour = (240, 240, 240)
491.                 self.text_color = (255, 255, 30)
492.                 self.name = name
493.                 self.display_text = name
494.                 self.user_input = ''
495.
496.                 font_path = os.path.join('resources', 'fonts', 'ARCADECLASSIC.TTF')
497.                 self.__font = pg.font.Font(font_path, font_size * win_scale)
498.                 self.text_surface = self.__font.render(self.display_text, True, self.text_color)
499.                 self.text_rect = self.text_surface.get_rect(center=self.text_box_rect.center)
500.
501.             def update(self, events):
502.                 """
503.                 Updates the textbox. This includes changing the size of the box if it gets to small
     for the text and handling
504.                 events.
505.                 :param events: Contains events from the pg.event.get() call containing all keyboard
     events.
506.                 :return: None.
507.                 """
508.
509.                 # Updates the textbox
510.                 # Resize the box if the text is too long.
511.                 text_width = self.text_surface.get_width()
512.                 if text_width > self.text_box_rect.w - 10 * self.win_scale:
513.                     self.text_box_rect.w += 10 * self.win_scale
514.                     self.text_box_rect.x -= 5 * self.win_scale
515.
516.                 elif self.text_surface_width_og < text_width < self.text_box_rect.w - 20 * self.win
    _scale:
517.                     self.text_box_rect.w -= 10 * self.win_scale
518.                     self.text_box_rect.x += 5 * self.win_scale
519.                 if self.interactive:
520.                     for event in events:
521.                         if event.type == pg.MOUSEBUTTONUP:
522.                             # If the user clicked on the input_box rect.
523.                             if self.text_box_rect.collidepoint(*event.pos):
524.                                 # Toggle the active variable.
525.                                 self.active = True
526.                                 self.user_input = ''
527.                             else:
528.                                 self.active = False
529.                                 self.tab_active = False
530.                                 self.text_entered = False
531.
532.                         if event.type == pg.KEYDOWN:
533.                             if self.active or self.tab_active:
534.                                 if event.key == pg.K_BACKSPACE:
535.                                     self.user_input = self.user_input[:-1]
536.                                 elif event.key == pg.K_RETURN:
537.                                     return self.user_input  # think about this yea
538.                                 else:
539.                                     # Only accepts characters between 46-123
540.                                     if event.unicode.lower() in [chr(x) for x in range(46, 123)]:
541.                                         if len(self.user_input) < self.max_length:
542.                                             self.user_input += event.unicode
543.
544.                 # Re-render the text.
545.                 if len(self.user_input) == 0:
```

```
546.                     self.display_text = self.name
547.
548.                 if len(self.user_input) > 0:
549.                     self.text_entered = True
550.                     if self.private:
551.                         self.display_text = '*' * len(self.user_input)
552.                     else:
553.                         self.display_text = self.user_input
554.                 else:
555.                     self.text_entered = False
556.
557.                 self.text_surface = self.__font.render(self.display_text, True, self.text_color)
558.                 self.text_rect = self.text_surface.get_rect(center=self.text_box_rect.center)
559.
560.             def display(self, win):
561.                 """
562.                 Blits text and box to the screen.
563.                 :param win: The current window, all objects must be blitted to this window to be di
    splayed.
564.                 :type win: Surface.
565.                 :return: None.
566.                 """
567.
568.                 # Blit the text.
569.                 win.blit(self.text_surface, self.text_rect)
570.                 # Blit the rect.
571.                 pg.draw.rect(win, self.text_box_colour
572.                              if not (self.active or self.tab_active)
573.                              else self.text_box_highlighted_colour, self.text_box_rect, 2 * self.wi
    n_scale)
574.
575.             def get_input(self):
576.                 return self.user_input
577.
578.
579.         class ErrorBox:
580.             def __init__(self, content, win_scale):
581.                 """
582.                 Error box are used for when there is an error that effects the game in a way the us
    er would not expect.
583.                 :param content: The text that will be outputted in the box.
584.                 :type content: String.
585.                 :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
586.                 :type win_scale: Integer.
587.                 """
588.
589.                 # Essential
590.                 self.win_scale = win_scale
591.
592.                 self.content = content
593.
594.                 self.box = Box((20, 196), (296, 40), (255, 255, 255), 2, win_scale)
595.                 self.transparent_win = pg.Surface((296 * win_scale, 40 * win_scale))
596.                 self.transparent_win.set_alpha(225)
597.                 self.transparent_win.fill((161, 161, 161))
598.                 self.text_surface = Word(content,
599.                                          [coord/win_scale for coord in self.box.rect.center],
600.                                          (255, 255, 130, 100),
601.                                          20,
602.                                          win_scale,
603.                                          centre=True
604.                                          )
605.
606.             def update(self, events):
607.                 """
608.                 The only check that the error box performs is to see if a mouse button has been lif
    ted up or a key has been
```

```
609.                pressed down. This is because when the user presses something it will disappear.
610.                :param events: Contains events from the pg.event.get() call containing all keyboard
    events.
611.                :type events: List.
612.                :return: Boolean: True if the user has pressed any key or mouse button else False.
613.                """
614.
615.                for event in events:
616.                    if event.type in [pg.MOUSEBUTTONUP, pg.KEYDOWN]:
617.                        return False
618.                return True
619.
620.            def display(self, win):
621.                """
622.                Blits the box and text to the surface.
623.                :param win: The current window, all objects must be blitted to this window to be di
    splayed.
624.                :type win: Surface.
625.                :return: None.
626.                """
627.
628.                win.blit(self.transparent_win, (20 * self.win_scale, 196 * self.win_scale))
629.                self.box.display(win)
630.                self.text_surface.display(win)
631.
632.
633.        class Button:
634.            def __init__(self, content, pos, dimensions, font_size, text_colour, width, win_scale,
    centre=False):
635.                """
636.                Buttons are used to allow the user to select certain outcomes or events. Buttons us
    e the word class and the
637.                box class.
638.                :param content: The text that will be blitted inside the button's box.
639.                :param pos: (x, y) Contains the x and y coordinates of the box.
640.                :type pos: Tuple.
641.                :param dimensions: (w, h) Contains width and height of the box.
642.                :type dimensions: Tuple.
643.                :param font_size: Font Size.
644.                :type font_size: Integer.
645.                :param text_colour: (r, g, b) colour of text.
646.                :type text_colour: Tuple.
647.                :param width: Width of box. (thickness or line)
648.                :type width: Integer.
649.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
650.                :type win_scale: Integer.
651.                :param centre: If True the button will be centred at x, y. Otherwise x, y will beco
    me the top left point of the
652.                button.
653.                :type centre: Boolean.
654.                """
655.
656.                x, y = pos
657.                w, h = dimensions
658.
659.                self.active = False
660.                self.click = False
661.
662.                self.tab_active = False
663.
664.                self.react_boxes = []
665.
666.                self.box = Box((x, y), (w, h), (161, 161, 161), width, win_scale, centre=centre)
667.                self.highlight_box = Box((x, y), (w, h), (255, 255, 255), width, win_scale, centre=
    centre)
668.
669.                if centre:
```

```
670.                 pos = [coord/win_scale for coord in self.box.rect.center]
671.                 self.text_surface = Word(content, pos, text_colour, font_size, win_scale, centr
    e=centre)
672.             else:
673.                 pos = [coord/win_scale for coord in self.box.rect.midright]
674.                 self.text_surface = Word(content, pos, text_colour, font_size, win_scale, centr
    e=centre)
675.
676.         def update(self, events):
677.             """
678.             Checks whether the mouse is 'colliding' with the button's box and if the mouse has
    been clicked. It becomes
679.             active if it has been clicked (meaning the colour of the box will change). The prog
    ram using the button must
680.             implement the button's function and can use the click attribute to decide when to e
    xecute the desired outcome.
681.             :param events: Contains events from the pg.event.get() call containing all keyboard
     events.
682.             :type events: List.
683.             :return: None
684.             """
685.
686.             pos = pg.mouse.get_pos()
687.             collision = self.check_mouse(*pos)
688.             click = False
689.             for event in events:
690.                 if event.type == pg.MOUSEBUTTONUP:
691.                     click = True
692.                 elif event.type == pg.KEYDOWN:
693.                     if event.key == pg.K_RETURN:
694.                         click = True
695.
696.             self.click = collision and click
697.
698.             if not self.tab_active:
699.                 self.active = collision
700.             elif collision:
701.                 self.tab_active = False
702.
703.         def display(self, win):
704.             """
705.             Blits the box, and text to the win surface.
706.             :param win: The current window, all objects must be blitted to this window to be di
    splayed.
707.             :type win: Surface.
708.             :return: None.
709.             """
710.
711.             if self.active or self.tab_active:
712.                 self.highlight_box.display(win)
713.             else:
714.                 self.box.display(win)
715.             self.text_surface.display(win)
716.
717.         def check_mouse(self, x, y):
718.             """
719.             Returns True if the (x, y) coord is within the box's rect.
720.             :param x: x coord of mouse.
721.             :param y: y coord of mouse.
722.             :return: True if (x, y) is colliding else False.
723.             """
724.
725.             return self.box.check_mouse(x, y)
726.
727.         def get_click(self):
728.             return self.click
729.
730.
```

```
731.        class Slider:
732.            def __init__(self, content, x, y, w, h, font_size, text_colour, box_width, win_scale, c
    entre=False, level=50,
733.                         levels=100):
734.                """
735.                Uses a box, a word and a transparent box to create a slider. It can be used by the
    user to select a level for
736.                something. The range is from 0 to the levels parameter. The default is 100 and this
     is mainly used for sound
737.                sliders, but the one for win scale, for example, only goes up to 5.
738.                :param content: The word that will be displayed in the slider (name of the slider).

739.                :param x: x coord of pos.
740.                :type x: Integer.
741.                :param y: y coord of pos.
742.                :type y: Integer.
743.                :param w: Width of box.
744.                :type w: Integer.
745.                :param h: Width of box.
746.                :type h: Integer.
747.                :param font_size: Font size.
748.                :type font_size: Integer.
749.                :param text_colour: (r, g, b) colour of the word / content.
750.                :type text_colour: Tuple.
751.                :param box_width: Width of box line.
752.                :type box_width: Integer.
753.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
754.                :type win_scale: Integer.
755.                :param centre: If True the (x, y) coords will become the boxes centre else the (x,
    y) coords will be the boxes
756.                top left.
757.                :type centre: Boolean.
758.                :param level: The current level of the slider.
759.                :type level: Integer.
760.                :param levels: The maximum level of the slider.
761.                :type levels: Integer.
762.                """
763.
764.                # Needed to move the slider
765.                self.x = x if not centre else x - 1/2 * w
766.                self.y = y if not centre else y - 1/2 * h
767.                self.w = w
768.                self.h = h
769.
770.                self.win_scale = win_scale
771.
772.                self.font_size = font_size
773.                self.text_colour = text_colour
774.                self.centre = centre
775.
776.                self.react = False
777.                self.active = False
778.                self.click = False
779.
780.                self.content = content
781.
782.                self.levels = levels
783.
784.                self.level = (level/levels) * w * win_scale
785.
786.                self.level_string = level
787.
788.                # Boxes (normal and highlighted)
789.                self.box = Box((x, y), (w, h), (161, 161, 161), box_width, win_scale, centre=centre
    )
790.                self.highlight_box = Box((x, y), (w, h), (255, 255, 255), box_width, win_scale, cen
    tre=centre)
```

```
791.
792.                # Text
793.                self.text_surface = self.get_text()
794.
795.                # Transparent slider
796.                self.slider = self.get_slider()
797.
798.            def update(self, events):
799.                """
800.                Checks where on the slider the mouse has been clicked and adjusts the slider and wo
    rd to match this value.
801.                :param events: Contains events from the pg.event.get() call containing all keyboard
     events.
802.                :type events: Tuple.
803.                :return: None.
804.                """
805.
806.                pos = pg.mouse.get_pos()
807.                collision = self.check_mouse(*pos)
808.                click = False
809.                if pg.mouse.get_pressed()[0]:
810.                    click = True
811.                self.click = collision and click
812.                self.react = collision
813.
814.                if self.click:
815.                    x, y = pos
816.                    self.level = x - self.x * self.win_scale
817.                    self.level_string = round((self.level / (self.win_scale * self.w)) * self.level
    s)
818.
819.                self.slider = self.get_slider()
820.                self.text_surface = self.get_text()
821.
822.            def display(self, win):
823.                """
824.                Blits the transparent slider (pygame surface object), the text, and the box to the
    screen. If react is True then
825.                the highlighted (white) box will be blitted instead.
826.                :param win: The current window, all objects must be blitted to this window to be di
    splayed.
827.                :return: None
828.                """
829.
830.                win.blit(self.slider, (self.x * self.win_scale, self.y * self.win_scale))
831.                self.text_surface.display(win)
832.
833.                if self.react:
834.                    self.highlight_box.display(win)
835.                else:
836.                    self.box.display(win)
837.
838.            def get_text(self):
839.                """
840.                Instanciates a text object. This must be run every time the level is changed in ord
    er to update the number.
841.                :return: The word object.
842.                """
843.
844.                text_surface = Word('{} {}'.format(self.content, self.level_string),
845.                                    [coord / self.win_scale for coord in self.box.rect.center],
846.                                    self.text_colour,
847.                                    self.font_size,
848.                                    self.win_scale,
849.                                    centre=True
850.                                    )
851.                return text_surface
852.
```

```
853.            def get_slider(self):
854.                """
855.                Gets the transparent surface that will become the sliding element. This must be run
     every time the level changes
856.                in order display that change graphically as well.
857.                :return: Slider object.
858.                """
859.
860.                slider = pg.Surface((self.level, self.h * self.win_scale))
861.                slider.set_alpha(225)
862.                slider.fill((130, 130, 130))
863.                return slider
864.
865.            def check_mouse(self, x, y):
866.                return self.box.check_mouse(x, y)
867.
868.
869.        class TransparentInputBox(InputBox):
870.            def __init__(self, w, h, font_size, win_scale, name=''):
871.                x = 168
872.                y = 226
873.                super().__init__(x, y, w, h, font_size, win_scale, name, True, True, False, 3)
874.
875.                self.active = True
876.
877.                self.transparent_win = pg.Surface((200 * win_scale, 100 * win_scale))
878.                self.transparent_win.set_alpha(225)
879.                self.transparent_win.fill((161, 161, 161))
880.
881.                self.box = Box((168, 210), (200, 100), (230, 230, 230), 2, win_scale, centre=True)

882.
883.            def display(self, win):
884.                win.blit(self.transparent_win, (68 * self.win_scale, 160 * self.win_scale))
885.                self.box.display(win)
886.                super().display(win)
887.
888.
889.        class Icon:
890.            def __init__(self, pos, imgs, win_scale, sound=False, toggle=False, target_program=Fals
    e):
891.                """
892.                Class for creating icons.
893.                :param pos: Position on the screen where the icon will be.
894.                :param imgs: Img(s). Either the icon image or if is is toggleable you will need mor
    e than one image.
895.                :param win_scale: Window Scale (How large the window is - must be multiplied by all
     size related variables).
896.                :param sound: If the icon controls sound then this is True.
897.                :param toggle: If this icon is a toggle (image changes after each press) this is Tr
    ue.
898.                :param target_program: If the icon should run a sub-
    program when clicked then the name should be passed here.
899.                """
900.
901.                # Essential
902.                self.sound = sound
903.                self.toggle = toggle
904.                self.has_target = True if target_program is not False else False
905.                self.target_program = target_program
906.                self.sound_toggle = sound
907.
908.                # Sound
909.                if self.sound:
910.                    pg.mixer.music.play()
911.
912.                x, y = pos
```

```
913.                self._imgs = [pg.transform.smoothscale(pg.image.load(img), (25 * win_scale, 25 * wi
    n_scale)) for img in imgs]
914.                self._rect = self._imgs[0].get_rect(bottomright=(x * win_scale, y * win_scale))
915.                self._image_num = 0
916.
917.            def display(self, win):
918.                """
919.                Blits the icon to the screen using the position and imgs.
920.                :param win: The current window, all objects must be blitted to this window to be di
    splayed.
921.                :return: None
922.                """
923.
924.                win.blit(self._imgs[self._image_num], self._rect)
925.
926.            def check_click(self, x, y):
927.                """
928.                Checks if the mouse coords are inside a rectangle.
929.                :param x: x-coord of mouse click
930.                :param y: y-coord of mouse click
931.                :return: True if mouse click is inside rect else False.
932.                """
933.
934.                return self._rect.collidepoint(x, y)
935.
936.            def action(self):
937.                """
938.                This usually runs when the check_click function returns True, and completes the ico
    ns desired action.
939.                :return: None
940.                """
941.
942.                if self.toggle:
943.                    self._image_num = abs(self._image_num - 1)
944.
945.                if self.sound:
946.                    if self.sound_toggle:
947.                        pg.mixer.music.pause()
948.                        self.sound_toggle = False
949.                    else:
950.                        pg.mixer.music.unpause()
951.                        self.sound_toggle = True
```

LOCAL DATABASE

```
1.  __author__ = 'Will Evans'
2.  import sqlite3
3.  import json
4.  import datetime
5.  import os.path
6.  import re
7.
8.
9.  def create_users(cursor):
10.     """
11.     Creates table 'Users'
12.     :param cursor: Object used to execute SQL within the database.
13.     :return: None
14.     """
15.
16.     sql = """CREATE TABLE Users
17.                           (UserID INTEGER,
18.                            UserName TEXT,
19.                            Password TEXT,
20.                            PRIMARY KEY(UserID))
21.              """
22.     cursor.execute(sql)
```

```
23.
24.
25. def create_game_history(cursor):
26.     """
27.         Creates table 'GameHistory'
28.         :param cursor: Object used to execute SQL within the database.
29.         :return: None
30.         """
31.
32.     sql = """CREATE TABLE GameHistory
33.                             (GameID INTEGER,
34.                              UserID INTEGER,
35.                              MazeID INTEGER,
36.                              Initials TEXT,
37.                              Date TEXT,
38.                              Time TEXT,
39.                              PRIMARY KEY(GameID),
40.                              FOREIGN KEY(UserID) REFERENCES  Users(UserID),
41.                              FOREIGN KEY(MazeID) REFERENCES Mazes(MazeID))
42.                           """
43.     cursor.execute(sql)
44.
45.
46. def create_game_level(cursor):
47.     """
48.         Creates table 'GameLevel'
49.         :param cursor: Object used to execute SQL within the database.
50.         :return: None
51.         """
52.
53.     sql = """
54.                 CREATE TABLE GameLevel
55.                             (LevelID INTEGER,
56.                              GameID INTEGER,
57.                              LevelNum INTEGER,
58.                              Lives INTEGER,
59.                              Score INTEGER,
60.                              Length INTEGER,
61.                              PelletsEaten INTEGER,
62.                              PowerPelletsEaten INTEGER,
63.                              GhostsEaten INTEGER,
64.                              PRIMARY KEY(LevelID),
65.                              FOREIGN KEY(GameID) REFERENCES GameHistory(GameID))
66.
67.                 """
68.     cursor.execute(sql)
69.
70.
71. def create_multiplayer_game_history(cursor):
72.     """
73.         Creates table 'MultiplayerGameHistory'
74.         :param cursor: Object used to execute SQL within the database.
75.         :return: None
76.         """
77.
78.     sql = """CREATE TABLE MultiplayerGameHistory
79.                             (MultiPlayerGameID INTEGER,
80.                             GameID INTEGER,
81.                             PRIMARY KEY(MultiPlayerGameID),
82.                             FOREIGN KEY(GameID) REFERENCES GameHistory(GameID))
83.
84.                             """
85.     cursor.execute(sql)
86.
87.
88. def create_multiplayer_player_history(cursor):
89.     """
90.         Creates table 'MultiplayerPlayerHistory'
```

```
        :param cursor: Object used to execute SQL within the database.
        :return: None
        """

    sql = """"""
    cursor.execute(sql)


def create_mazes(cursor):
            """
                Creates table 'Mazes'
                :param cursor: Object used to execute SQL within the database.
                :return: None
                """

            sql = """CREATE TABLE Mazes
                                    (MazeID INTEGER,
                                     UserID INTEGER,
                                     Maze TEXT,
                                     Date TEXT,
                                     PRIMARY KEY(MazeID),
                                     FOREIGN KEY(UserID) REFERENCES Users(UserID))
                    """
            cursor.execute(sql)

            sql = """
                      INSERT INTO Mazes
                      (Maze)
                      VALUES (?)

                      """

            for maze_id in range(1):
                mazes = ['Level1', 'Level2']
                default_mazes_path = os.path.join('Resources', 'Levels.txt')
                with open(default_mazes_path, 'r') as json_file:
                    maze = json.load(json_file)[mazes[maze_id]]

                maze = json.dumps(maze)
                query(sql, (maze,))


        def create_db():
            """
            Checks whether the database has already been created and if not creates it.
            :return: None
            """

            if not os.path.exists('data\\database.db'):
                with sqlite3.connect('data\\database.db') as db:
                    cursor = db.cursor()

                    create_users(cursor)
                    create_game_level(cursor)
                    create_game_history(cursor)
                    create_multiplayer_game_history(cursor)
                    create_mazes(cursor)


        def query(sql, data=None):
            """
            Used to execute SQL statments, can also return data.
            :param sql: The SQL that will be executed.
            :param data: Data can be added when an SQL statement inputs data into the database.
            :return: Results if there are any.
            """

            db_path = os.path.join('data', 'database.db')
```

```
159.            with sqlite3.connect(db_path) as db:
160.                cursor = db.cursor()
161.                if data is None:
162.                    cursor.execute(sql)
163.                else:
164.                    cursor.execute(sql, data)
165.                results = cursor.fetchall()
166.                db.commit()
167.            return results
168.
169.
170.        def get_game_id(user_id, maze_id):
171.            """
172.            Creates a game history entry using the arguments and date, time. Returns the GameID jus
    t created.
173.            :param user_id: Users personal ID
174.            :param maze_id: MazeID for the maze being played on.
175.            :return: GameID
176.            """
177.
178.            sql = """INSERT INTO GameHistory (UserID, MazeID, Date, Time) VALUES (?, ?, ?, ?)"""
179.            data = (user_id, maze_id, get_date(), get_time())
180.            query(sql, data)
181.
182.            sql = """SELECT Max(GameID) FROM GameHistory"""
183.            return query(sql)[0][0]
184.
185.
186.        def get_highscore():
187.            """
188.            Returns the current highscore! In one line! One complex query!
189.            :return: Highscore
190.            """
191.
192.            # This is one complex query!
193.            return query("""SELECT Max(Scores) FROM(SELECT Sum(Score) as Scores from GameLevel Grou
    p BY GameID)""")[0][0]
194.
195.
196.        def save_level(level_num, game_id, lives, score, length, pellets_eaten, power_pellets_eaten
    , ghosts_eaten):
197.            """
198.            Saves Individual level data to GameLevel.
199.            :param level_num: Level number.
200.            :param game_id: GameID.
201.            :param lives: How many lives at the end of the level.
202.            :param score: Score for that particular level.
203.            :param length: Length of level.
204.            :param pellets_eaten: Pellets eaten in level.
205.            :param power_pellets_eaten: Powerpellets eaten in level.
206.            :param ghosts_eaten: Ghosts eaten in level.
207.            :return: None
208.            """
209.
210.            sql = """
211.                    INSERT INTO GameLevel
212.                    (GameID, LevelNum, Lives, Score, Length, PelletsEaten, PowerPelletsEaten, Ghos
    tsEaten)
213.                    VALUES (?, ?, ?, ?, ?, ?, ?, ?)
214.                  """
215.
216.            query(sql, (game_id, level_num, lives, score, length, pellets_eaten, power_pellets_eate
    n, ghosts_eaten))
217.
218.
219.        def save_initials(game_id, initials):
220.            """
221.            Saves initials to game history after the game has finished.
```

```
222.            :param game_id: GameID that the initials will be assigned to.
223.            :param initials: 3 Character initials.
224.            :return: None
225.            """
226.
227.            sql = """
228.                  UPDATE GameHistory
229.                  SET Initials=?
230.                  WHERE GameID=?
231.                  """
232.
233.            if len(initials) == 3:
234.                query(sql, (initials, game_id))
235.
236.
237.        def save_maze(user_id, maze):
238.            """
239.            After a user has created a maze it can be saved to their account using this funtion.
240.            :param user_id: UserID from user who created level.
241.            :param maze: 2D list of maze.
242.            :return: None
243.            """
244.
245.            sql = """
246.                    INSERT INTO
247.                    Mazes (UserID, Maze, Date, Time)
248.                    VALUES (?, ?, ?, ?)
249.                  """
250.            query(sql, (user_id, json.loads(maze), get_date(), get_time()))
251.
252.
253.        def save_user(username, password):
254.            """
255.            When a user has completed the sign up form their information is used to create a user h
    ere.
256.            :param username: The user's chosen username.
257.            :param password: The user's chosen password.
258.            :return: None
259.            """
260.
261.            sql = """
262.                    INSERT INTO Users (Username, Password)
263.                    VALUES (?, ?)
264.                 """
265.
266.            query(sql, (username, password))
267.
268.
269.        def get_maze(maze_id):
270.            """
271.            Returns the 2D maze from the database that corresponds to the MazeID.
272.            :param maze_id:
273.            :return:
274.            """
275.
276.            sql = """
277.                  SELECT Maze
278.                  FROM Mazes
279.                  WHERE MazeID=?
280.                  """
281.
282.            return query(sql, (maze_id,))[0][0]
283.
284.
285.        def login(username, password):
286.            """
287.            Checks user provided details against database.
288.            :return: UserID
```

```
289.            """
290.
291.            sql = """
292.                        SELECT UserID
293.                        FROM Users
294.                        WHERE Username=?
295.                        AND Password=?
296.                    """
297.            try:
298.                user_id = query(sql, (username, password))[0][0]
299.            except IndexError:
300.                return None
301.            return user_id
302.
303.
304.        def check_sign_up(user_name, password, password_confirm):
305.            """
306.            Performs the following checks in order: Username length, username availability, passwor
    d length, password strength,
307.            password matches.
308.            :param user_name: User's chosen username
309.            :param password: User's chosen password.
310.            :param password_confirm: User's chosen password confirm.
311.            :return: Boolean: whether the details are valid, String: Error message if one of the ch
    ecks failed. (appropriate for
312.            given any error).
313.            """
314.
315.            # Username Length
316.            if len(user_name) < 5:
317.                return False, "Username must be 5 characters"
318.
319.            # Username availability
320.            sql = """SELECT EXISTS(SELECT 1 FROM Users WHERE Username=?)"""
321.            if query(sql, (user_name,))[0][0]:
322.                return False, "Username taken"
323.
324.            # Password Length
325.            if len(password) < 7:
326.                return False, "Password must be 7 characters"
327.
328.            # Password Strength
329.            if not re.search("[a-z]", password):
330.                return False, "Password must contain lower"
331.
332.            if not re.search("[A-Z]", password):
333.                return False, "Password must contain upper"
334.
335.            if not re.search("[0-9]", password):
336.                return False, "Password must contain number"
337.
338.            # Passwords match
339.            if password != password_confirm:
340.                return False, "Passwords must match"
341.
342.            return True, None
343.
344.
345.        def get_username(user_id):
346.            """
347.            Returns username from database, given the user_id
348.            :param user_id: UserID returned from login earlier.
349.            :return: Username corresponding to the UserID.
350.            """
351.
352.            sql = """SELECT Username FROM Users WHERE UserID=?"""
353.            return query(sql, (user_id,))[0][0]
354.
```

```
355.
356.        def get_statistics(user_id):
357.            sql = """
358.                SELECT
359.                SUM(Score) as Total_Score,
360.                SUM(PelletsEaten) as Pellets_Eaten,
361.                SUM(PowerPelletsEaten) as Power_Pellets_Eaten,
362.                SUM(Length) as Time_Played,
363.                SUM(GhostsEaten) as Ghosts_Eaten
364.                FROM GameLevel, GameHistory
365.                WHERE GameHistory.GameID = GameLevel.GameID AND UserId=?
366.                GROUP BY UserID
367.                """
368.
369.            database_path = os.path.join('data', 'database.db')
370.            with sqlite3.connect(database_path) as db:
371.                db.row_factory = sqlite3.Row
372.                cursor = db.cursor()
373.                cursor.execute(sql, (user_id,))
374.                stats_tuple = cursor.fetchall()
375.
376.            if len(stats_tuple) is 0:
377.                return None
378.            else:
379.                return dict(stats_tuple[0])
380.
381.
382.        def get_highscores():
383.            sql = """SELECT Sum(score) as SCORE, GameHistory.Initials
384.                     FROM GameLevel, GameHistory
385.                     WHERE GameLevel.GameID = GameHistory.GameID
386.                     AND GameHistory.Initials IS NOT NULL
387.                     GROUP BY GameHistory.GameID
388.                     ORDER BY SCORE DESC
389.                    """
390.
391.            return query(sql)
392.
393.
394.        def get_date():
395.            """
396.            Gets date.
397.            :return: Date
398.            """
399.
400.            return datetime.datetime.now().strftime("%d/%m/%Y")
401.
402.
403.        def get_time():
404.            """
405.            Gets time.
406.            :return: Time.
407.            """
408.
409.            return datetime.datetime.now().strftime("%H:%M:%S")
410.
411.
412.        if __name__ == '__main__':
413.            sql = """
414.                  INSERT INTO Mazes
415.                  (Maze)
416.                  VALUES (?)
417.
418.                  """
419.
420.            maze_id = 1
421.            mazes = ['Level1', 'Level2']
422.            with open('Resources\\Levels.txt', 'r') as json_file:
```

```
423.                 maze = json.load(json_file)[mazes[maze_id - 1]]
424.
425.             maze = json.dumps(maze)
426.             query(sql, (maze,))
```

LOCAL SETTINGS

```
1.  __author__ = 'Will Evans'
2.
3.  import json
4.  import os.path
5.
6.
7.  def write_settings():
8.      """
9.      If there is not a settings file, one will be created.
10.     :return: None
11.     """
12.
13.     file_path = os.path.join('data', 'settings.json')
14.
15.     with open(file_path, 'w+') as file:
16.         json.dump(
17.
18.                     {
19.
20.                         'win_scale':         2,
21.                         'music_volume':     50,
22.                         'game_volume':      50,
23.                         'user_name':        None,
24.                         'password':         None,
25.
26.                     },
27.
28.                     file)
29.
30.
31. def get_setting(setting):
32.     """
33.     Reads value from chosen setting. If there is no settings file, one will be created.
34.     :param setting: The setting that should be read from.
35.     :return: The value of the setting
36.     """
37.
38.     file_path = os.path.join('data', 'settings.json')
39.     while True:
40.         try:
41.             if not os.path.exists(file_path):
42.                 write_settings()
43.
44.             with open(file_path, 'r') as file:
45.                 data = json.load(file)
46.                 return data[setting]
47.
48.         except (PermissionError, OSError):
49.             print("settings fetch error")
50.
51.
52. def save_setting(setting, value):
53.     """
54.     Saves a given value to a given setting.
55.     :param setting: The setting that will be written to.
56.     :param value: The value that will be written into the setting.
57.     :return: None
58.     """
```

```
59.
60.     file_path = os.path.join('data', 'settings.json')
61.     while True:
62.         try:
63.             with open(file_path, 'r') as file:
64.                 data = json.load(file)
65.                 data[setting] = value
66.
67.             with open(file_path, 'w') as file:
68.                 json.dump(data, file)
69.             break
70.
71.         except PermissionError:
72.             print("save failed")
```

TESTING

STRATEGY

As with every project, a great deal of testing is needed to ensure it is fully working and complete for end-user use. Hopefully, by completing this test, I will be able to fix any areas where there are currently issues. I will be using video-based testing as I believe this will be the best way to not only evidence my tests, but also in order to fix the issues captured (if there are any). I will attach timestamps to the video, to make navigating through the video and to aid in the understanding the current test being carried out. Of course, I will do my best to verbally assist this as well.

To test my signup algorithm, I will test each field with valid and invalid inputs to make sure that only secure passwords are used. I will also showcase my systems ability to recognise duplicate usernames and also test my login system using the user I create. I will also further test my login system using slightly inaccurate information, such as a correct password but with the wrong username and vice versa. This should expose any flaws in the detail-checking logic and also show the database design illustrated in the design and technical solution working.

Testing my settings is also a fairly simple exercise. I will need to manipulate the sliders, to show they can change, then confirm they save in the JSON file and finally confirm they affect the necessary areas of my game correctly.

For my tutorial testing, I will focus on checking the gameplay aspects for the whole game. This can then be carried over to the other modes as they use the same logic and sprites. The specific areas I will checking for are the collisions between Pac-Man and the ghosts in different modes, the ghosts' behaviours of ghosts on different levels and in different modes (especially scared and dead mode). I will check all other areas of the gameplay simply by playing and observing the game. I will also need to check game mode specific things for the tutorial, which in this case will be the tutorial messages. This includes confirming the messages occur at the correct time, they are the correct messages (according to the JSON file containing the messages) and that they are displayed correctly using the 'scrolling word' objects.

After testing the gameplay specific aspects of my game, we can evaluate the features specific to the Classic mode. This includes making sure Pac-Man loses lives when killed, seeing whether he gains an extra life at 10,000 points, whether the high score changes if a game is played where the score becomes the high score in the game and more.

I will also work out some of the high scores from the database before comparing these with ones on the high scores page, to see if it is working correctly. I shall also make sure that high scores save after initials have been entered.

The majority of my testing will be on the multiplayer section of my project. This is both where most of my focus has been and where most of the bugs will likely appear. I must check the creating, joining and leaving of lobbies, ensuring continuity across all players connected and making sure no errors occur due to lost connections etc. I must then test whether my efforts to keep the game in sync are working adequately. This must include states not just positions of sprites. When a game is finished the players must correctly swap according to the rules and the match must end when a user reaches 20k points. Then, each player must be awarded a position 1st through 5th.

VIDEO

All of my tests so far and more can be seen in the following video

https://www.youtube.com/watch?v=N5uu_PKdjRk

PLAN

KEY

1 - Start Screen
2 - Login Screen
    2.4 - Login Errors
3 - Sign Up Screen
    3.2 - Sign Up Errors
4 - Settings
5 - Tutorial / Gameplay
    5.1 - Sprite movement
    5.2 - Pac-Man
        5.2.5 - Pac-Man eating ghosts
        5.2.6 - Pac-Man death
    5.3 - Ghosts
        5.3.8 - Ghost modes
    5.4 - Scrolling messages
    5.5 - Gameplay
        5.6.2 - Round ending
    5.6 - Maze
6 - Classic
7 - High scores
8 - Multiplayer
    8.1 - Creating a game
    8.2 - Joining a game
    8.3 - Gameplay
        8.3.3 - Points
        8.3.6 - Places

TABLE

Test No.	Purpose of the test (To test)	Test Data	Expected Outcome	Actual Outcome	Changes Needed	Timestamp
1.1	Whether the intro music starts playing on the start screen.	None.	The music should start playing even after just returning to the start screen.	The expected.	None.	00:11
1.2.1	Whether the 'LiveWord' class responds to mouse collisions.	Mouse input.	A highlighting colour should appear behind the word.	The expected.	None.	00:29
1.2.2	Whether the 'LiveWord' class responds to mouse clicks.	Mouse click.	The player should be taken to the appropriate screen.	The expected.	None.	00:53
1.3.1	That toggle icons respond to mouse clicks.	Mouse click.	The icon should toggle its skin and perform an action.	The expected.	None.	01:12
1.3.2	That navigation icons respond to mouse collisions.	Mouse click.	The should be taken to the appropriate screen.	The expected.	None.	01:36
2.1	That you can press tab to select the first and subsequent boxes.	Tab character.	The first box should be selected when the tab is hit, the selection should go down with every subsequent tab.	If the first box was selected by the mouse, the tab character just selects it again (and not the next one).	Allow the code to understand if a box is currently highlighted using the mouse and thus account for this when the tab is hit.	02:02
2.2	That the text box responds to mouse and keyboard input.	James. Mouse click. James.	The box should highlight when clicked and then (and only then) display any text written.	The expected.	None.	02:31
2.3	That the signup/log in button responds to clicking, mouse hovering and also username box interaction.	Username: James Mouse click.	The button should start off as a signup button then changes to login when the username box is clicked. A log in should be attempted when the button is clicked.	The expected.	None.	03:02

2.4.1	That invalid usernames prevent account sign in.	Username: James Password: Cheese123	Error showing incorrect details were entered.	The expected	None.	04:03
2.4.2	That invalid passwords prevent account sign in.	Username: Jeremy Password: Milk123	Error showing incorrect details were entered.	The expected	None.	04:36
2.4.3	That an invalid name and username prevents account sign in.	Username: James Password: Milk123	Error showing incorrect details were entered.	The expected	None.	06:38
2.4.4	That a user can log in when the surname and password are correct.	Username: Jeremy Password: Cheese123	The user is logged in and taken to the accounts screen	The expected	None.	06:57
2.5	That the button class can change its content based on input. (Remember me button in this case).	Mouse click.	The button should highlight when hovered over (as tested before) but then should change colour when clicked.	The expected.	None.	04:49
2.6	That clicking the remember me button saves the users information to the settings document.	Mouse click.	The user's username and password should be saved to settings and the user should be signed in next time the game loads.	The expected.	None.	05:11
3.1	That a taken username prevents account creation.	Username: Jeremy	Error: 'Username taken', regardless of string case.	The expected.	None.	07:10
3.2.1	That a password without 7 or more characters prevents account creation.	Username: James Password: Cheese	Error: 'Password must be 7 characters.	The expected.	None.	07:36
3.2.2	That a password without any lower-case letters prevents account creation.	Username: James Password: CHEESE123	Error: 'Password must contain lower'.	The expected.	None.	08:00
3.2.3	That a password without a capital	Username: James	Error: 'Password must contain upper'.	The expected.	None.	08:32

	letter prevents account creation.	Password: cheese123				
3.2.4	That a password without a number prevents account creation.	Username: James Password: Cheesey	Error: 'Password must contain number'.	The expected.	None.	08:54
3.2.5	That two passwords that do not match prevent account creation.	Username: Jamesy Password: Cheese123 Confirm Password: Cheese321	Error: 'Password must match'	The expected.	None.	09:16
3.3	That the sign-up button starts the details check.	None.	The button should lead to the database being queried about the validity of the details.	The expected.	None.	09:45
4.1	That sliders respond to mouse input.	Mouse hover and drag.	The user should be able to drag the slider which in turn changes the number in the slider.	The expected.	None.	11:27
4.2	That sliders alter in-game settings as per their function.	None.	When the value of a slider changes, this change should be reflected in the settings.	The expected.	None.	11:52
5.1.1	That the Pac-Man sprite takes keyboard input.	Any arrow keys.	When the up-arrow key is pressed Pac-Man travels up (where applicable) etc and continues in this direction.	The expected.	None.	15:36
5.1.2	Pac-Man plays a noise when moving.	None.	When Pac-Man is moving a siren noise should play on repeat, but this should stop as soon as it stops moving.	The expected.	None.	16:00
5.1.3	That the Pac-Man sprite stops when it hits an obstacle.	None.	Pac-Man should collide with the wall and then stop moving.	The expected.	None.	16:16
5.1.4	That Pac-Man does not execute a move command when there is an	A keyboard input. (in this case right arrow key)	Pac-Man should not change direction.	The expected.	None.	16:34

	obstacle in the way.					
5.1.5	Whether the tunnel works correctly.	None.	When any sprite moves of the screen they should instantly reappear on the other side.	The expected.	None.	17:08 / 30:30
5.2.1	Whether Pac-Man can collide with pellets.	None.	When Pac-Man collides with a pellet it should disappear.	The expected.	None.	17:20
5.2.2	Whether the users score increases when colliding with a pellet.	None.	When Pac-Man collides with a pellet the user's score should increase by 10pts or 50pts if it is a power-pellet.	The expected.	None.	17:34 / 22:57
5.2.3	Whether a sound is played when Pac-Man eats a pellet.	None.	When Pac-Man collides with a pellet one 'Pellet death sound' should play.	The expected.	None.	17:49
5.2.4	Whether Pac-Man eating a Power-pellet causes the ghosts to enter scared mode.	None.	When Pac-Man eats a power pellet the ghost' should change to scared mode.	The expected.	None.	22:57
5.2.5.1	Whether Pac-Man can eat ghosts in scared mode.	None.	When Pac-Man eats a ghost, they should enter 'dead' mode.	The expected.	None.	23:38
5.2.5.2	Whether the user's score increases after eating ghosts.	None.	The users' score should increase by 200 when Pac-Man eats the first ghost, then 400, 800 and 1600.	If a ghost is still in dead mode from a previous power pellet this will be reflected in the score as if they had been eaten again.	To be solved.	24:09
5.2.6.1	Whether colliding with a ghost that's 'alive' will kill Pac-Man	None.	All other players should stop being displayed leaving just Pac-man who can no longer receive keyboard input.	The expected.	None.	19:08

5.2.6.2	Whether the Pac-Man death animation plays when Pac-Man collides with an 'alive' ghost.	None.	Pac-Man should quickly switch between 8 special skins which make up his death animation.	The expected.	None.	19:08
5.2.6.3	Whether a new round starts when Pac-Man's death animation finishes.	None.	A new round should begin with all characters in their original position, however, the score and pellets remain unchanged.	The expected.	None.	19:08
5.3.1	Whether the ghosts are in the correct mode and for the correct lengths of time. (Including Elroy)	None.	The timings (that can be worked out by printing the mode in the console and using a stopwatch) should be equal to the timings outlined in the analysis and design.	The expected.	None.	24:50 / 30:22
5.3.2	Whether the individual ghosts' target the correct tile in scatter and chase mode.	None.	For example, Pinky should target the tile 4 ahead of Pac-Man in chase mode. Other cases and ghosts are outlined in the analysis and design.	The expected.	None.	26:06
5.3.3	Whether the ghosts' mode changes to scared mode when Pac-Man eats a power-pellet.	None.	Ghosts should change skin to the blue scared skin, become slower, target random tiles.	The expected.	None.	25:37
5.3.4	Whether the ghosts' paths display correctly in story mode.	None.	The paths drawn on the maze should cross through all the tiles that are in the ghost's path list.	The expected.	None.	19:39
5.3.5	Whether ghosts follow their paths correctly.	None.	In story mode, the ghosts' paths are displayed. They must simply follow this line.	The expected.	None.	19:49
5.3.6	Whether the ghosts' path is	None.	We can observe the on-screen path, manually work out	The expected.	None.	19:55

	the most efficient.		the shortest path and the two should be the same.			
5.3.7	Whether the ghosts target random tiles when in scared mode.	None.	We can print out the target tile every tick and then test to see if these tiles are random.	The expected.	None.	26:35
5.3.8.1	Whether ghosts' change to dead mode after being eaten.	None.	Ghosts should change skin to the dead skins and change speed to become much faster.	The expected.	None.	26:53
5.3.8.2	Whether dead ghosts return to the centre and change modes.	None.	Ghosts should return to the centre and then change to whatever mode they would have been on.	The expected.	None	27:10
5.3.8.3	Whether ghosts play the death sound when they are eaten.	None.	Ghosts should play the death sound when they are eaten.	The expected.	None.	27:27
5.4.1	That the scrolling text pulls messages from the 'tutorial' text file.	None.	The scrolling text should be pulled continually from the text file depending on which level.	The expected.	None.	13:46
5.4.2	That the scrolling text never overflows the text box.	None.	The scrolling text should work out whether the next word will collide with the text box and start a new line.	The expected.	None.	14:09
5.4.3	That the scrolling text stops when it reaches the third line.	None.	The scrolling box should detect when the next word will overflow and if it is on the third line. It should then print three dots.	The expected.	None.	14:33
5.4.4	That you can click the space bar to continue to print out a message.	Space bar.	Once three dots are printed, if the user presses the space bar the current contents of the box should be erased and the rest of the message printed.	The expected.	None	14:47

5.4.5	That you can skip the scrolling message by pressing space.	Space bar.	When a message is being printed, the user should be able to press the space bar to skip the scrolling.	The expected.	None.	15:11
5.4.6	That the Ms.Pac-Man sprite changes skin only when the text is scrolling.	None.	When a message is being printed, we should be able to see Ms.Pac-Man's mouth opening and closing, however, when the message is finished, the skin changing should finish.	The expected.	None.	15:21
5.5.1	That the round ends when Pac-Man wins.	None.	When Pac-Man has eaten all the pellets, the round should end and a new one should begin.	The expected.	None.	21:36
5.5.2.1	That the round ends when Pac-Man dies.	None.	When Pac-Man is eaten by a ghost the round should end.	The expected.	None.	22:04
5.5.2.2	That Pac-Man does not lose a life at the end of the round.	None.	When Pac-Man is eaten his lives should not decrease by one.	The expected.	None.	21:50
5.6.1	That the maze displays correctly.	None.	The maze should display correctly (with all the correct tile sprites) with reference to the JSON file containing the maze tilemap.	The expected.	None.	13:29
5.6.2	That the maze flashes when a round is won.	None.	When Pac-Man has eaten all the pellets the maze should flash (alternating between white and blue).	The expected.	None.	21:03
6.1	That there is a high score at the top of the screen.	None.	The game should fetch the high score from the database and display it at the top of the screen.	The expected.	None.	27:54
6.2.1	That Pac-Man starts with 3 lives.	None.	There should be 3 life indicators in the bottom left corner of	The expected.	None.	28:12

			the screen indicating Pac-Man has 3 lives.			
6.2.2	That Pac-Man loses a life when he dies.	None.	When Pac-Man is caught by a ghost, he should respawn if he has a life. If he does, this life should then be lost.	The expected.	None.	28:35
6.2.3	That the game ends if Pac-Man has no lives at the end of the round.	None.	When Pac-Man is eaten, if he has no lives then the game over text should appear and the game will end.	The expected.	None.	28:57
6.2.4	That Pac-Man gains another life at 10k points.	None.	When the user reaches 10k points, they should receive an extra life.	The expected.	None.	31:45
6.3.1	That the user is prompted to save a high score.	None.	After Pac-Man's death animation a box should pop up where the user can enter 3 initials to save the high score.	The expected.	None.	29:00
6.3.2	That the user's score is saved.	3 initials. Enter.	When the user enters 3 initials and hits enter, they should be taken to the high score screen where they can see their save high score (if it is in the top ten). If they do not enter 3 initials it should not save.	The expected.	None.	31:11
7.1	That the high score page displays the correct scores in the correct order.	None.	When taken to the high scores page, the high scores displayed should be highest in the database and in order.	The expected.	None.	32:38
7.2	That the order of the high score information is: place, score, initials.	None.	The high score information should have the order: place, score, initials.	The expected.	None.	32:58
7.3	That the top three high scores are coloured	None.	The top three high scores should be coloured gold, silver	The expected.	None.	33:15

	gold, silver and bronze for 1st through 3rd.		and bronze for 1st through 3rd.			
7.4	That the user is returned to the start screen if any button is pressed.	Any button.	If any button is pressed the user should be taken back to the start screen.	The expected.	None.	33:32
8.1.1	That players can create a game lobby.	None.	A lobby is created, and the player is automatically added to it	The expected.	None.	35:08
8.1.2	That all players can see when a player readies up.	None.	The ready text will appear under the player that readied up	The expected.	None.	35:33
8.1.3	That all players can see all other players' avatar.	None.	Each player's avatar should appear (for them) in the central box and any other players will be above.	The last person to join the lobby sees the 3rd ghosts' avatar, and not their own.	None.	37:01 /
8.1.4	That all players can see all other player's name.	None.	Each player's name should appear (for them) below the central box and any other players will be above.	The expected.	None.	37:07
8.1.5	That all players can see all other player's score.	None.	Each player's score should appear (for them) below the central box and any other players will be above.	The expected.	None.	37:11
8.1.6	That a countdown can be started by the host.	Mouse click.	The countdown should appear on all connected player's screen when the host slicks start button	The expected.	None.	37:18
8.2.1	The user should be able to enter a GameID that will allow them to join a game.	None.	After the user enters the GameID they should join a game and all other players in that game should appear along the top of the screen.	The expected.	None.	35:47

8.2.2	That players who join can ready up.	None.	Players that join a game (not hosts) should be able to ready up, using the ready up button in the bottom right corner.	The expected.	None.	36:56
8.3.1	That the user's score appears in the top left.	None.	Each user should see their own score in the top left.	Whilst technically the user's score does appear in the top left, this can sometimes be incorrect when a user has been caught by PAC-Man (causing them to inherit PAC-man's score).		47:33
8.3.2	That each user starts as the avatar they were shown in the lobby.	None	Each user should play as the avatar from the lobby.	Due to the error in test 8.1.3 this test failed the first time around.	Same as 8.1.3.	47:34
8.3.3.1	That the user playing as Pac-Man gains 10pts from a pellet.	None.	The user playing as Pac-Man should receive 10pts from eating a pellet.	The expected.	None.	50:55
8.3.3.2	That the user playing as Pac-Man gains 50pts from a power pellet.	None.	The user playing as Pac-Man should receive 50pts from eating a pellet.	The expected.	None.	50:55
8.3.3.3	That the user playing as Pac-Man gains points from eating ghosts in the same way as the classic game.	None.	The user playing as Pac-Man should receive points from eating ghosts in the same way as the classic game.	The expected.	None.	50:55

8.3.3.4	That the ghosts gain points for being close to Pac-Man.	None.	The users playing as the ghosts should gain points from being close to Pac-Man.	The expected.	None.	48:26
8.3.3.5	That the ghosts gain points for eating Pac-Man	None.	The ghost who eats Pac-Man should gain 1600pts.	The expected.	None.	48:38
8.3.4	That the ghost who catches Pac-Man becomes him in the next game.	None.	The ghost that catches Pac-Man should swap roles with the Pac-Man player in the following game.	The expected.	None.	48:45
8.3.5	That if the score reaches 20k in a game, the match will end at the end of that game.	None.	When a player reaches 20k points, the match continues until the game ends at which point each player are taken back to the lobby.	The expected.	None.	51:03
8.3.6.1	That each player is given a place from 1st to 5th depending on their scores at the end of the match	None.	When the match ends each player should be given a place 1st through 5th.	The expected.	None.	51:06
8.3.6.2	That the players (finishing 1st through 3rd)'s places are coloured gold, silver and bronze.	None.	The players' places (that came 1st, 2nd, and 3rd) should be coloured gold, silver and bronze.	The expected.	None.	51:06

EVALUATION

OBJECTIVE ANALYSIS

Objective Reference	To what extent was the objective met?	Test Reference
1 - Start Screen	I believe the following objectives were met very well. I have a fully interactive menu with 4 options: Tutorial, Classic, Multiplayer, and High scores, which are all accurate descriptions of the main features of the game. There are also three icons on	1

	the menu that the user can interact with, that ultimately enhance the user experience.	
1.1	As shown in the video whenever the Start Screen is running the music is also running. Whilst I would've liked the music to not restart when using a subprogram, the music does start again and play whenever the StartScreen is returned to.	1.1
1.2	Using the Live Word class from the GUI script, I was able to create a Start Screen with choices that react to mouse movement.	1.2
1.3	Similar to the previous, I created an icon class which allowed me to create icons, adding to the functionality and intuitiveness of the Start Screen.	1.3
2 - Sign-up Screen	**The objectives for the sign-up screen are very similar to the login screen and use the same classes, which have meant again the final product successfully meets the objectives.**	2
2.1	The input box class has allowed me to perfectly hit this objective, allowing the user to intuitively interact with the signup screen by clicking on boxes or pressing the tab button.	2.1 - 2.2
2.2	This sign-up button is far simpler than the logins. It simply initiates a check on the input data to ensure it is adequate. This check completes perfectly and works with all boundary data I have tested.	2.3
3 - Login Screen	**Whilst there are very few objectives for the login screen, I feel I have executed them very well, achieving a good looking and user-friendly login screen. I have also surpassed the objectives by adding a button that allows the user to stay logged in even if they close the program called 'Remember me'.**	3
3.1	The input box class has allowed me to perfectly hit this objective, allowing the user to intuitively interact with the login screen by clicking on boxes or pressing the tab button.	3.1 - 3.4
3.2	When text is entered into the username field the signup button on the login page turns into a sign-up button, which perfectly matches what I had outlined in my objectives.	3.5
4 - Settings Screen	**The settings section meets the objectives perfectly.**	4
4.1	All sliders change the desired settings and change inappropriate increments.	4.1
4.2	The settings also save correctly in the JSON file, meeting this objective perfectly.	4.2
5 - Gameplay	**The gameplay objectives have almost been completely met, save for the issue with pellets flickering sometimes.**	
5.1 - Maze	The maze is stored as a 2D list stored as a JSON object in a database (this is to allow data to be attached to different mazes - which supersedes the objectives). As per the objectives, this is turned into a 2D list of tile objects which store information regarding the look of the maze allowing faster rendering of the maze. When the user finishes a game, the maze is also able to flash blue and white.	5.6
5.2 - Pac-Man	Pac-Man is controlled using keyboard input, which is validated in order to stop Pac-Man from moving outside of the maze and to	5.1-5.2

	make sure he keeps moving when no input is received (until a collision with a wall or ghost). He is also able to collide with pellets and ghosts. When moving, he produces the 'Waka waka' sound,	
5.3 - Ghosts	As shown in my test video the ghosts correctly change to scared mode when a power pellet is eaten and switch correctly between chase and scatter mode according to the level timings. When dead they navigate quickly to the ghost hut and utilise A* correctly to achieve these things. The ghosts also do not change direction unless switching mode and make a death sound when eaten.	5.3
5.4 - Pellets	I believe this objective has been mostly met. Whilst the functionality regarding the pellets is there i.e. the correct amount of score is eaten for the two types and power pellets cause ghosts to enter scared mode. However, the pellets must remain static, only the power pellets should flash, and whilst they do flash, sometimes the normal pellets flicker. I feel it's likely an issue relating to the order of things displayed on the screen, but I have been unable to resolve it.	5.2
6 - Tutorial mode	I have met, fully all but one objective: random maze generation.	
6.1	The mode has a narrator (which I have chosen to be in the form of Ms.Pac-Man). I created a class that allows messages to scroll across the screen to mimic speech. The user is able to continue to the next bit of the message or skip the scrolling by pressing space.	5.4
6.2	The levels change according to the levels outlined in the high-level description, which make the levels progressively harder. This starts from simply introducing the user to moving Pac-Man and collecting pellets, all the way to all 4 ghosts.	6.2
6.3	Pac-Man instantly respawns when dying in tutorial mode, as opposed to having limited lives.	6.3
6.4	I have not met this objective since I wanted to make the maze truly random and (as discussed in the analysis) this would be very difficult given the timeframe.	
7 - Classic Mode	All objectives regarding classic mode have been met fully.	6
7.1	The correct high score is taken from the database and then displayed at the top of the screen.	6.1
7.2	Pac-Man starts with three lives and gains one when the user reaches 10k points.	6.2
7.3	The maze from the original Pac-Man is used.	
7.4	The user is prompted to select 3 initials to save with their score when the game finishes. This then appears straight away on the high score page that the user is taken to right after.	6.3
8 - High Scores	The objectives relating to the high scores have all been fully met.	7
8.1	The high score mode features the correct top ten scores (that have initials attached) achieved in classic mode.	7.1
8.2	The information is ordered: place, score, initials.	7.2

8.3	The top three scores are coloured gold, silver and bronze respectively.	7.3
8.4	If the user presses any key, they are taken to the Start Screen.	7.4
9 - Multiplayer	**Again, all objectives have been hit, but also superseded. I decided to add AI to the lobby as though they were actual players when the host clicks start (if there are any free spaces).**	
9.1 - Multiplayer Menu	As with the following menus, the multiplayer menu has 4 avatar boxes along the top of the screen, with a central large box in the middle. The four smaller ones feature a greyed-out ghost, while the central box contains a similarly grey Pac-Man. Under these boxes are two interactive options: 'Create Game' and 'Join Game'.	8
9.2 - Create Game Menu	At this stage, the central avatar turns to a coloured in Pac-Man, with the user's name displayed underneath. There is also their score of 0 under that, and further down: the GameID box. If a user joins at this stage, they are added to one of the 4 boxes above. At any time, the host is able to click the start button which starts the countdown on all connected players machines and also adds AI to any open slots.	8.1
9.3 - Join Game Menu	The avatars remain the same as in the create game menu. Instead of a number in the box beneath there is an input box. When a user puts a valid GameID into this box they are taken to the lobby and any players already in this lobby are added to the boxes above, while the central box is coloured in with the ghost the host assigned to the user. Like the other menu, the users' name and scores appear under the boxes. You can also ready up using the button on this page.	8.2
9.4 - Gameplay	Similar to the tutorial game mode, there are no lives here nor a high score indicator. Each user starts as the avatar they were given in the lobby. The player playing as Pac-Man earns points from pellets and eating ghosts, whereas the ghosts earn points from being close to or eating Pac-Man. At the end of a game, each user is awarded a place from 1^{st} through 5^{th} with the first 3 being coloured.	8.3
10 - Database	**My database has fully met all objectives set out in the analysis.**	
10.1	The database stores information including game mode, length, pellets eaten, ghosts eaten and more from each and every level played on a machine. The user that played these levels is added if signed in, which can be used in complex queries.	
10.2	The database should hold information about users, allowing them to log in and save their statistics.	
10.3	The database should facilitate complex queries, such as returning the high score, top 10 high scores and a list of a specific user's statistics. The database should also be able to indicate whether a set of login details is correct or whether sign-up details are valid.	

USER FEEDBACK

Below is a selection of questions and answers from a few different family and friends.

How did you find navigating the menu system?

"It was pretty easy. They [word options] are clearly laid out and well-spaced, the interactives of the words make it easy to know you can click them to go somewhere. The icons don't react in the same way, which is my only criticism, however, they are a good description of what menu they lead to. I like that the back button has been added, as it wasn't clear how to return to the main menu before."

How did you find creating and logging into your account?

"It was great, it felt just like a normal sign-up/login screen, it even has a remember me feature. I would say that a prompt to tell the user what an acceptable password is before they try multiple would be quite nice as well. I do like that you added the tab feature to skip to the next input box as I felt this was missing from the early concept you showed me."

Having played the tutorial mode, do you feel you understand the game more now?

"I don't think I've ever played Pac-Man myself but seeing all of the different things happening in the background is really interest and I suppose yes I understand the game far more now. I might even be pretty good at it."

After playing the classic mode, how do you feel I have captured the original game?

"To be fair, you might've just made it better. No bugs, more retro feel, I know you mentioned there is no fruit at the moment, but even as a fan of Pac-Man, I didn't miss them. Well done."

What do you all think of the multiplayer functionality?

"I've got to say, I love the game mode itself, as in how you win the game, very original. I love the fact that you are forced to work together as a team due to being visually impaired [ghosts] and then straight after your trying to catch the very person you were working with."

"I think my favourite bit is the lobby. That's not an insult to the game, but the seamlessness of joining a lobby and then being able to leave and re-join is really cool. Everyone's name is right there too! Even having ready and countdown stuff appear on everyone's screen is awesome."

"I'd say the whole thing is definitely polished, I love the fact that in the game you are working with your team, but at the same time trying to deceive them and catch Pac-Man for yourself. I'd say the Pac-Man AI needs work [Pac-Man makes random moves], but I suppose this only applies if you're not playing with the full 5 people like us and let's be honest it means the humans get to play as Pac-Man more. The way the AI work, however on the ghost's side is great."

To what extent would you agree the high score screen enhances the user experience?

"Yeah, it's laid out very well and combines the best of the original game and this retro style you've got going on, pretty neat. I do definitely feel as though it enhances the experience, specifically by encouraging yourself to try and try again to beat your high score and obviously easily being able to see how you're performing."

Analysis of user feedback.

It is clear from the interview that I hit all of the important objectives I set out in my analysis. The most important (especially for this section) being how balanced the multiplayer section was. Comments such as the mode being 'definitely polished' and 'very original' reinforce the fact that I achieved what I set out in my problem definition. There were a few areas for improvement, and I have included these below, along with some of my own suggestions for how I could improve my solution in the future.

POSSIBLE EXTENSIONS

Despite extensive research and the deduction that it is unfeasible given my time frame, I still would have loved to implement an algorithm that could randomly generate a Pac-Man maze (following all the rules set out in my analysis). I do believe I could add this feature to my project in the future and that it would add something to my game. Whilst it would not be used in the Classic mode (as this is designed to closely mimic the original game), I think it could be used subtly and effectively in an extension to the tutorial mode.

There is also one feature of the Classic mode that I have not added: fruit. Although this would certainly not be a difficult feature to implement, I chose to spend most of my time perfecting the networking aspect of my game (as this was the main focus of my interview feedback), and other key algorithms.

I would also like to add a few more features to my multiplayer experience, in the future. Whilst I feel, mechanically, it is fully-fledged in the current state, adding a few of the following features would certainly enhance the user experience and increase replayability:

Manual player swaps - This would allow the host to change which players start as which characters. I feel this is an important feature as it would allow players to choose which ghost to play with or allow players to start as Pac-Man which (despite many attempts to balance the gameplay) may be an advantage in some player's eyes.

Balancing - This isn't a feature per se, but I still would like to spend a substantial time reviewing gameplay to ensure it is balanced. Having such a small number of test games to watch gameplay, I can still not say for sure whether my game is truly fair. Unfortunately, this is not something I can work on at this stage as, like I said, I do not have many games to view.

True online - My game works on a local area network and could theoretically be used on the internet, however after briefly attempting this it is clearly essential to have some form of server that would simply, again, take too much time and effort to bring to light for little gain.

Pac-Man AI - I would have loved to include a better version of the Pac-Man AI. At the moment, Pac-Man makes random moves as creating an AI that could manage many different factors in decision making (unlike the ghosts) would have taken a long time and would've likely needed some form of machine learning that I have not yet had the time to learn effectively.

Furthering the idea of an online server, I would have loved to use this to share high scores with other users on the game. This would have been very simple (only requiring scripts to upload and download the high scores) as I already have the code needed to store high scores in a database, but, again, creating an online server is a project in its own right.

I also feel a 'creator mode' would have been a great addition. This would allow users to create their own mazes. Most of the code is already in place (such as an algorithm that can turn any Pac-Man maze design into a visual maze) as I did contemplate this feature in very early prototypes, however, the ability to decide what makes a Pac-Man maze valid held back this idea. Of course, it is certainly possible, but as with many

possible extensions the short time frame in which to complete my project made the addition less worthwhile.

As the distance between tiles is always one, I chose to use a tilemap instead of a graph in early concepts of my A* pathfinding algorithm. I have since come to realise my algorithm would have been better served with a graph connecting only the intersections. This would require far less processing power, as a path would only need to be found after reaching an intersection, and not at every new tile as it is now. Whilst my game does not have a profound issue with running smoothly, adding this feature would likely allow framerates to increase or allow it to be run on much less capable machines.

As I picked up PyGame with the start of my project, I had little time to fully explore all of its advanced features, instead of remaking some features I didn't know existed with lower-level PyGame functions. This is likely less efficient and also took up much of my development time. I would love to go back to my project and improve efficiency by using some of these advanced PyGame features, such as meshes. Meshes could not only improve speed but may also make my hitboxes more realistic and responsive.