

A multiplayer experience

created

by William Evans

Table of Contents

[Analysis 1](#_Toc29333547)

[Problem Definition 1](#_Toc29333548)

[Background Research 1](#_Toc29333549)

[Maze design 2](#_Toc29333550)

[Path finding 3](#_Toc29333551)

[Pac-Man VS. 4](#_Toc29333552)

[Identification of End Users 5](#_Toc29333553)

[Objectives 5](#_Toc29333554)

[1 - Start Screen 5](#_Toc29333555)

[2 - Sign-up Screen 6](#_Toc29333556)

[3 - Login Screen 6](#_Toc29333557)

[Models 10](#_Toc29333558)

[Basic Multiplayer Model 10](#_Toc29333559)

[Maze representation 11](#_Toc29333560)

[Networking 11](#_Toc29333561)

[A\* 13](#_Toc29333562)

[Design 14](#_Toc29333563)

[Project Technology 14](#_Toc29333564)

[High-level overview 15](#_Toc29333565)

[Mainloop 15](#_Toc29333566)

[Menu System 16](#_Toc29333567)

[Database 18](#_Toc29333568)

[High Level Description 18](#_Toc29333569)

[Main 18](#_Toc29333570)

[Splash Screens 19](#_Toc29333571)

[Tutorial 19](#_Toc29333572)

[Single Player 20](#_Toc29333573)

[Multiplayer 20](#_Toc29333574)

[Sprites 21](#_Toc29333575)

[Pathfinding 21](#_Toc29333576)

[Data Structures 21](#_Toc29333577)

[Multiplayer Sprites 22](#_Toc29333578)

[Networking 22](#_Toc29333579)

[GUI 22](#_Toc29333580)

[Local Database 23](#_Toc29333581)

[Local Settings 23](#_Toc29333582)

[User Interface 24](#_Toc29333583)

[Start Screen 24](#_Toc29333584)

[Settings 25](#_Toc29333585)

[Classic Mode 26](#_Toc29333586)

[Tutorial 27](#_Toc29333587)

[High Scores 28](#_Toc29333588)

[Sign Up 29](#_Toc29333589)

[Login 30](#_Toc29333590)

[Accounts 31](#_Toc29333591)

[Multiplayer 32](#_Toc29333592)

[Multiplayer – Host 33](#_Toc29333593)

[Multiplayer – Client 35](#_Toc29333594)

[Class Descriptions 37](#_Toc29333595)

[Class Diagram 37](#_Toc29333596)

[Class Description Table 40](#_Toc29333597)

[Function Description Table 58](#_Toc29333598)

[Key Algorithms 60](#_Toc29333599)

[A\* 60](#_Toc29333600)

[SQL 62](#_Toc29333601)

[Networking 62](#_Toc29333602)

[Data Structures 68](#_Toc29333603)

[Priority Queue 68](#_Toc29333604)

[Maze 69](#_Toc29333605)

[Technical solution 71](#_Toc29333606)

[Key Algorithms 71](#_Toc29333607)

[Code 71](#_Toc29333608)

[Main 71](#_Toc29333609)

[Splash Screens 73](#_Toc29333610)

[Tutorial 85](#_Toc29333611)

[Single Player 96](#_Toc29333612)

[Multiplayer 104](#_Toc29333613)

[Sprites 104](#_Toc29333614)

[Pathfinding 149](#_Toc29333615)

[Data structures 152](#_Toc29333616)

[Multiplayer Sprites 157](#_Toc29333617)

[Networking 167](#_Toc29333618)

[GUI 174](#_Toc29333619)

[Local Database 189](#_Toc29333620)

[Local Settings 196](#_Toc29333621)

[Testing 197](#_Toc29333622)

[Strategy 197](#_Toc29333623)

[Video 198](#_Toc29333624)

[Plan 198](#_Toc29333625)

[Key 198](#_Toc29333626)

[Table 199](#_Toc29333627)

[Evaluation 209](#_Toc29333628)

[Objective Analysis 209](#_Toc29333629)

[User Feedback 213](#_Toc29333630)

[Possible Extensions 214](#_Toc29333631)

# 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.

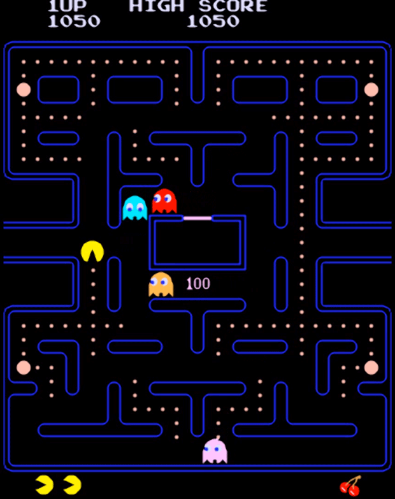
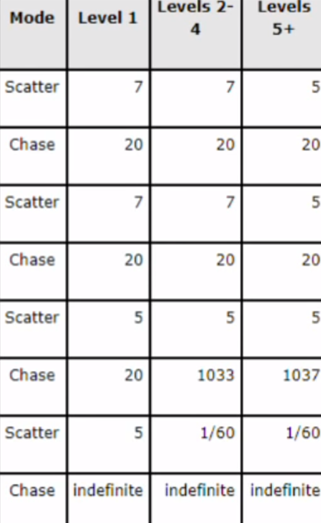
It is also important that my game has a built-in account system. This will allow the player to connect more easily with friends and encourage competition within the single-player modes through high scores. In order to accomplish this, I will have to build a local login and signup system, using a database 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 180o 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’.

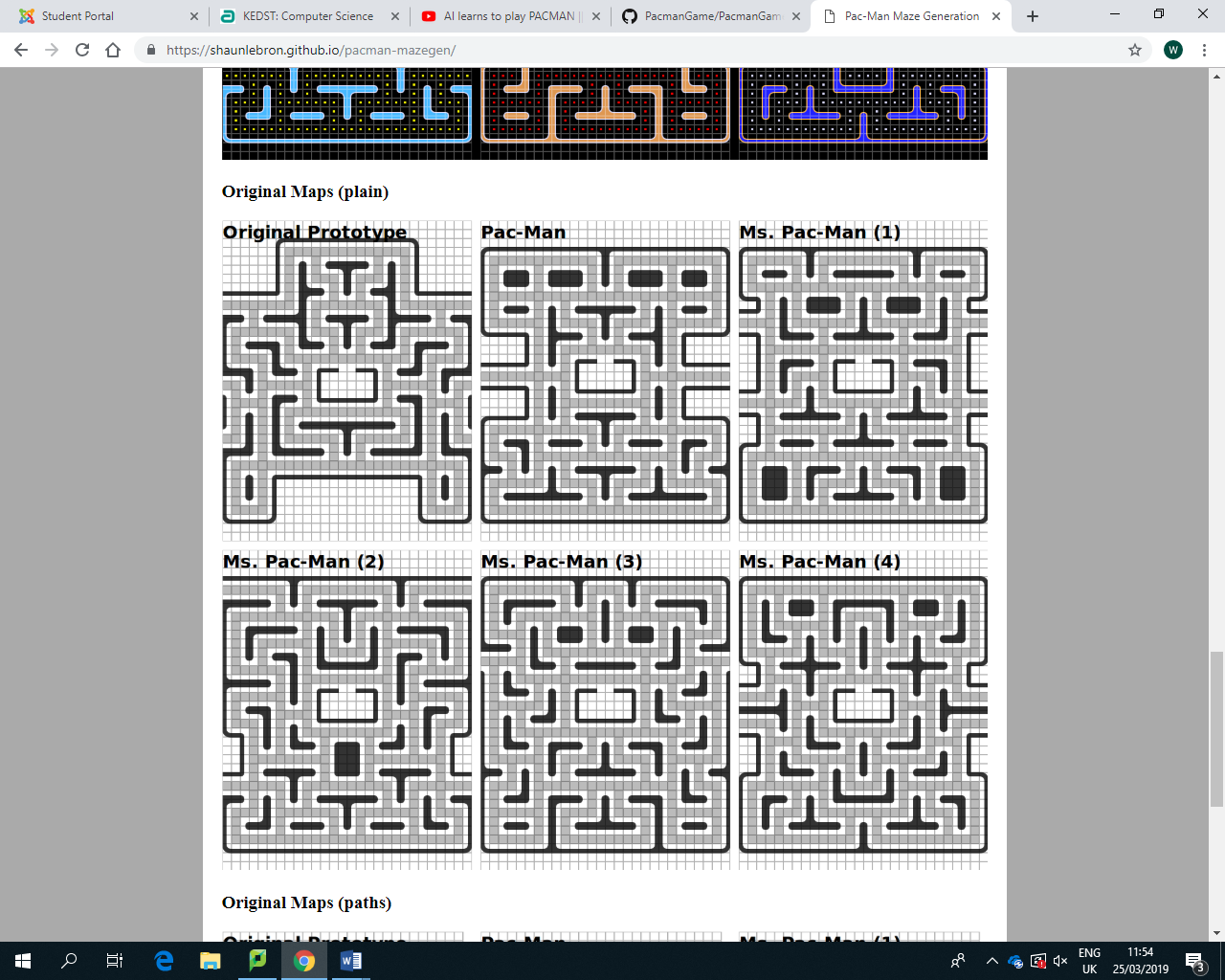
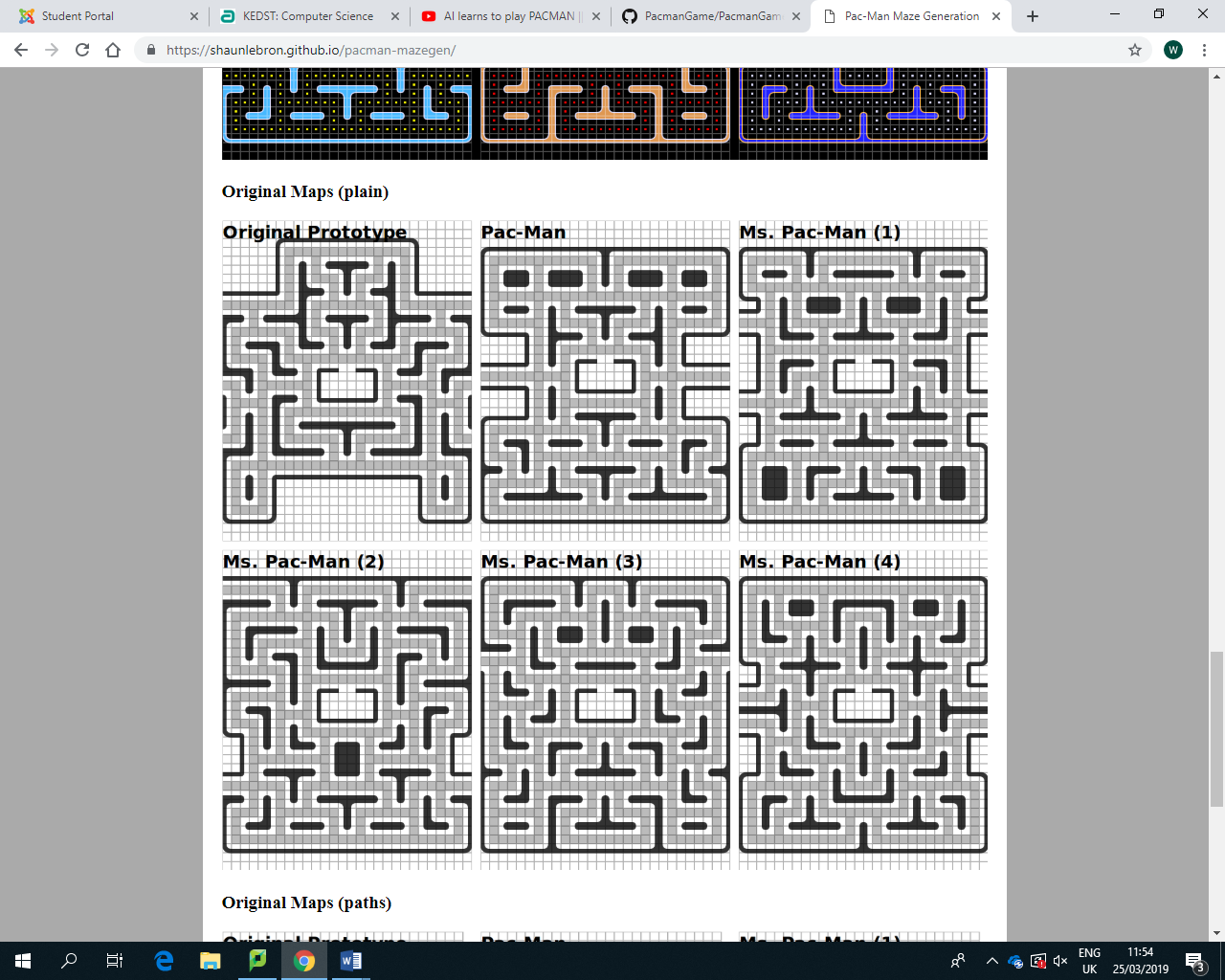
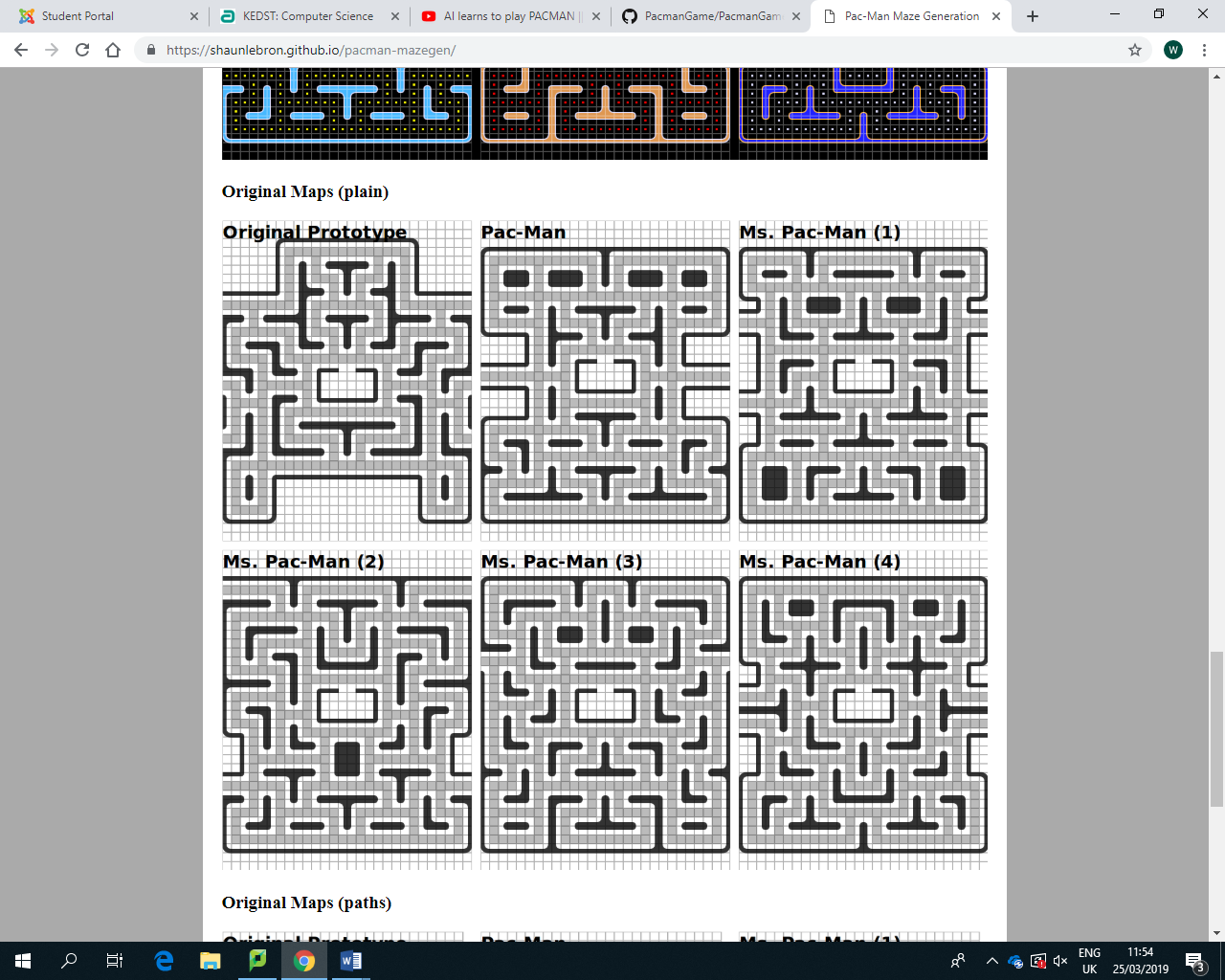
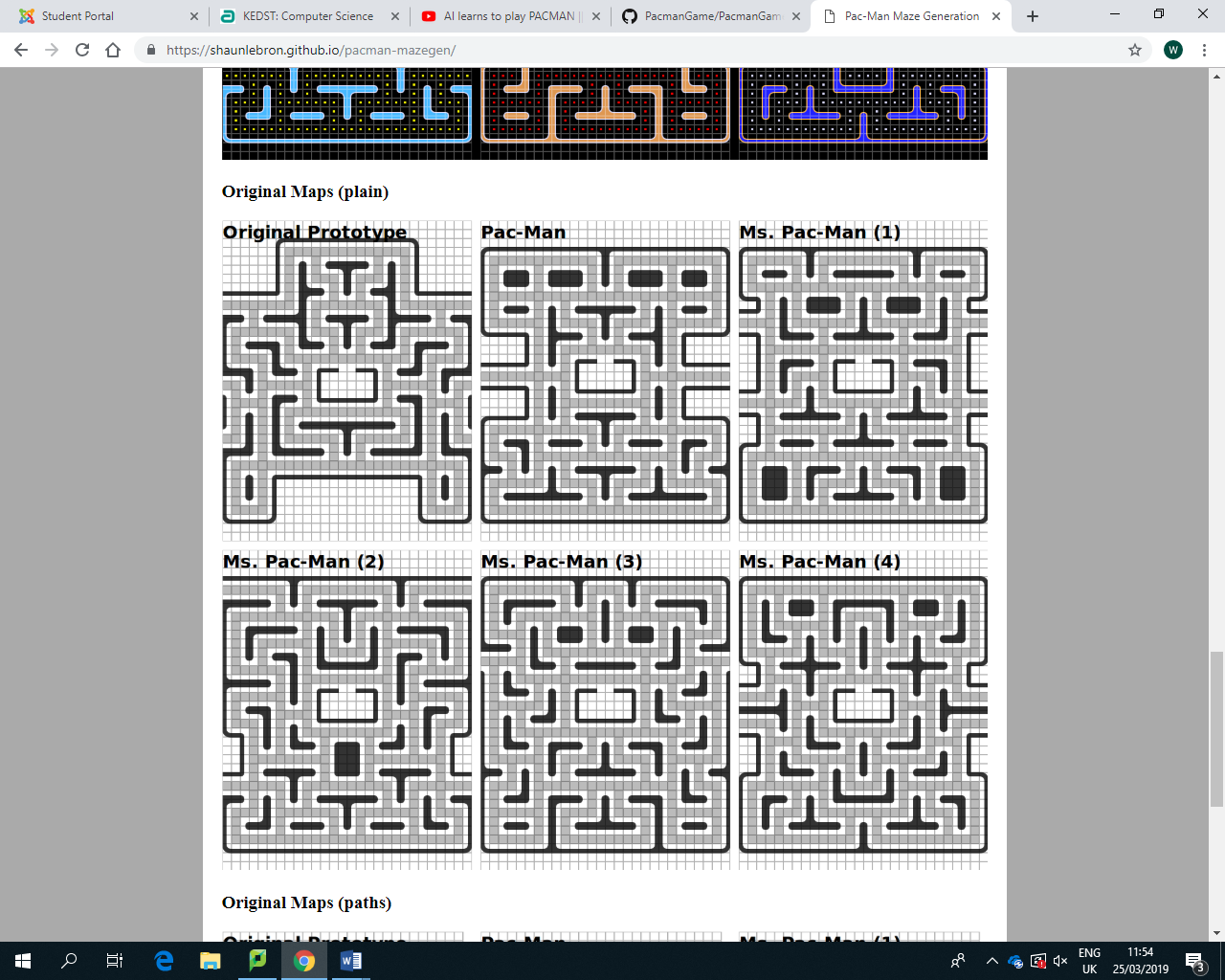
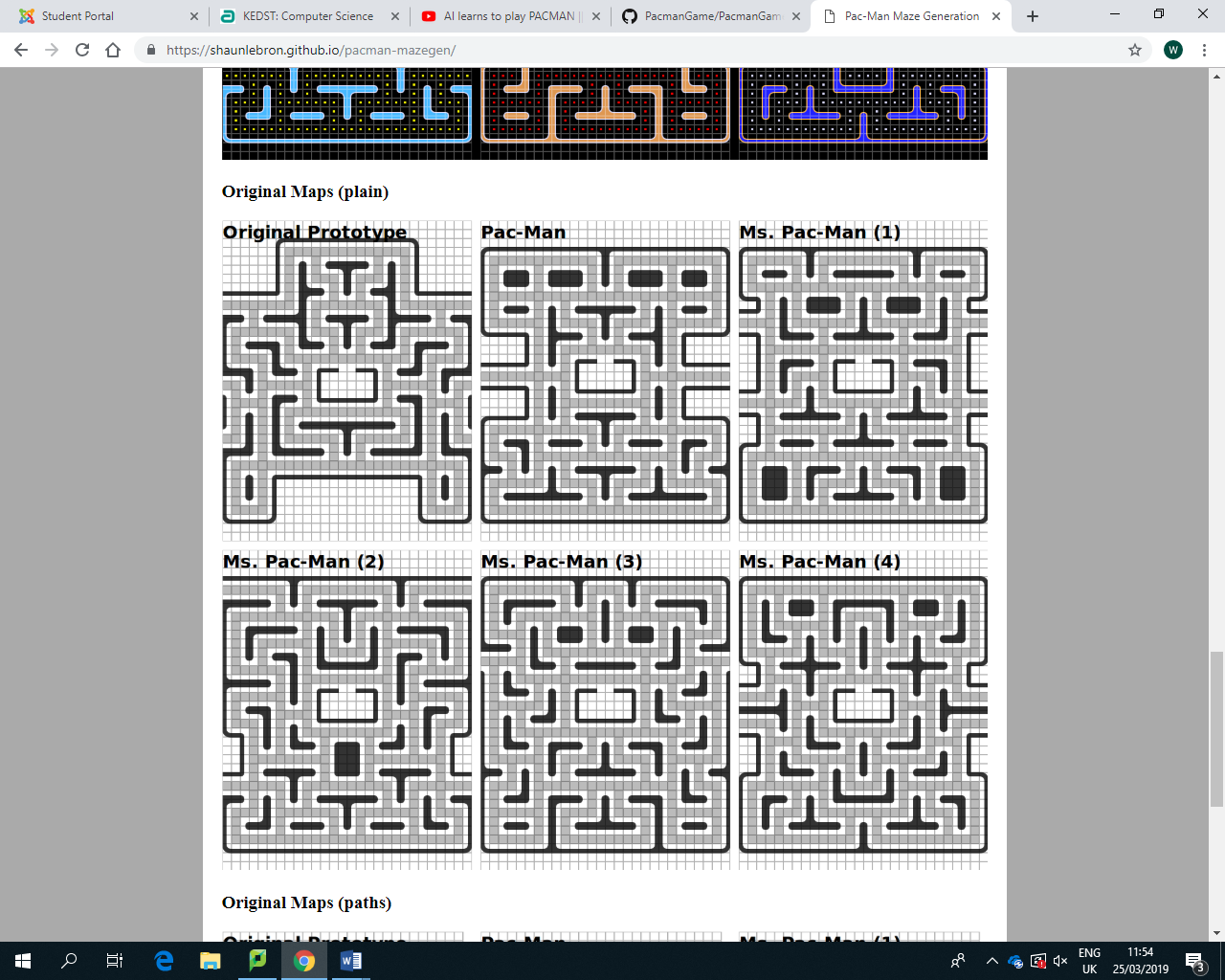
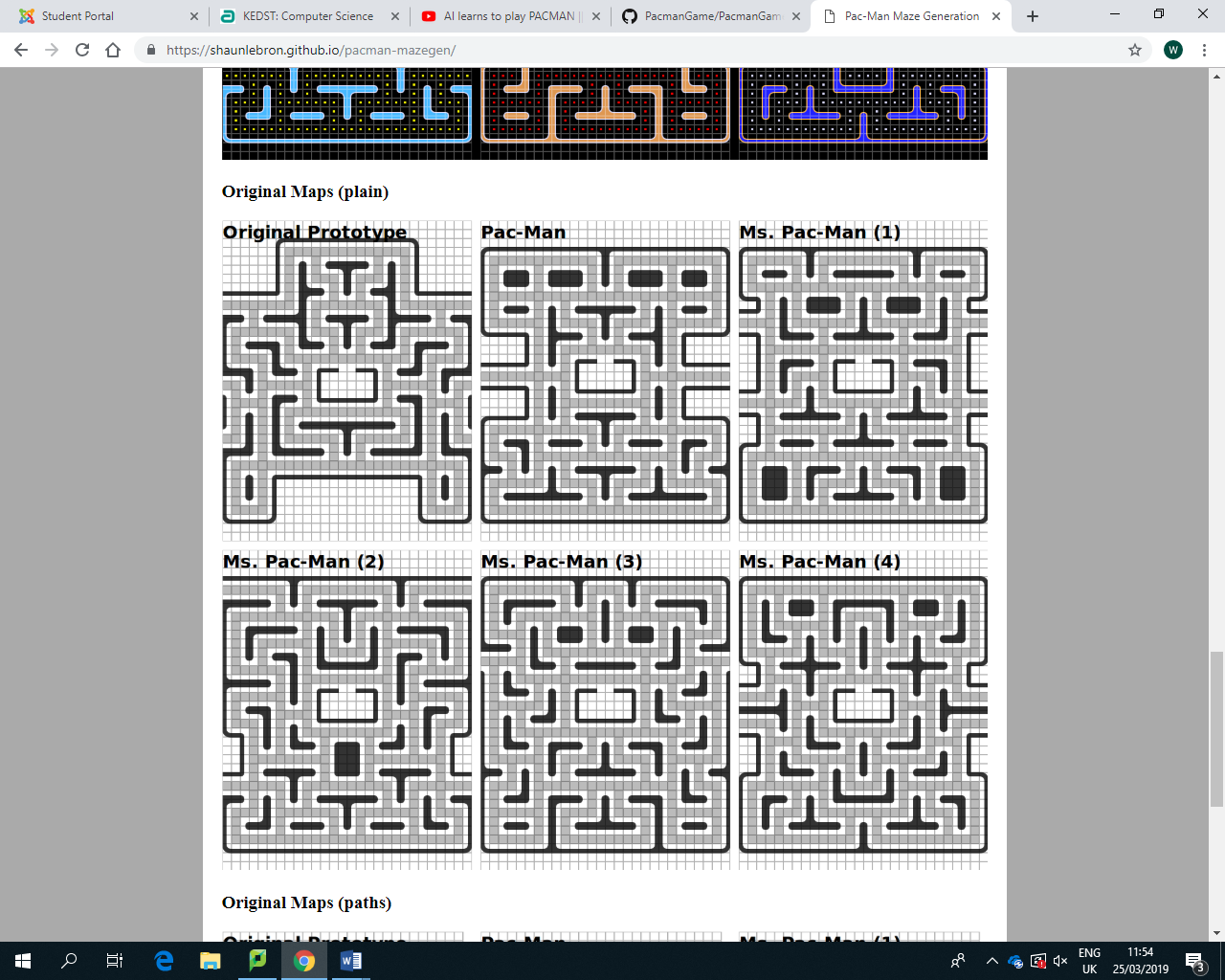
There are 3 modes: chase, scatter and frightened. The ghosts chase for a set time and then switch to scatter mode in which they no longer chase Pac-Man and instead head straight for one of the four corners each. When a power pellet is eaten, the ghosts change direction and enter scared mode whereby they 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.

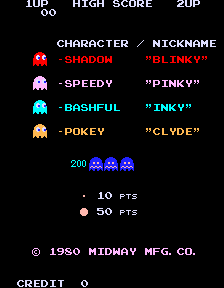
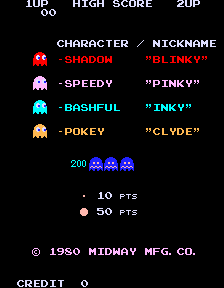
This idea sounds simple, but after doing some more research I do not believe it is feasible to create such an algorithm in my given timeframe. In fact, there have been many papers written on the subject, and the most elegant solution I found online is roughly 784 lines long and took the experienced programmer 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.

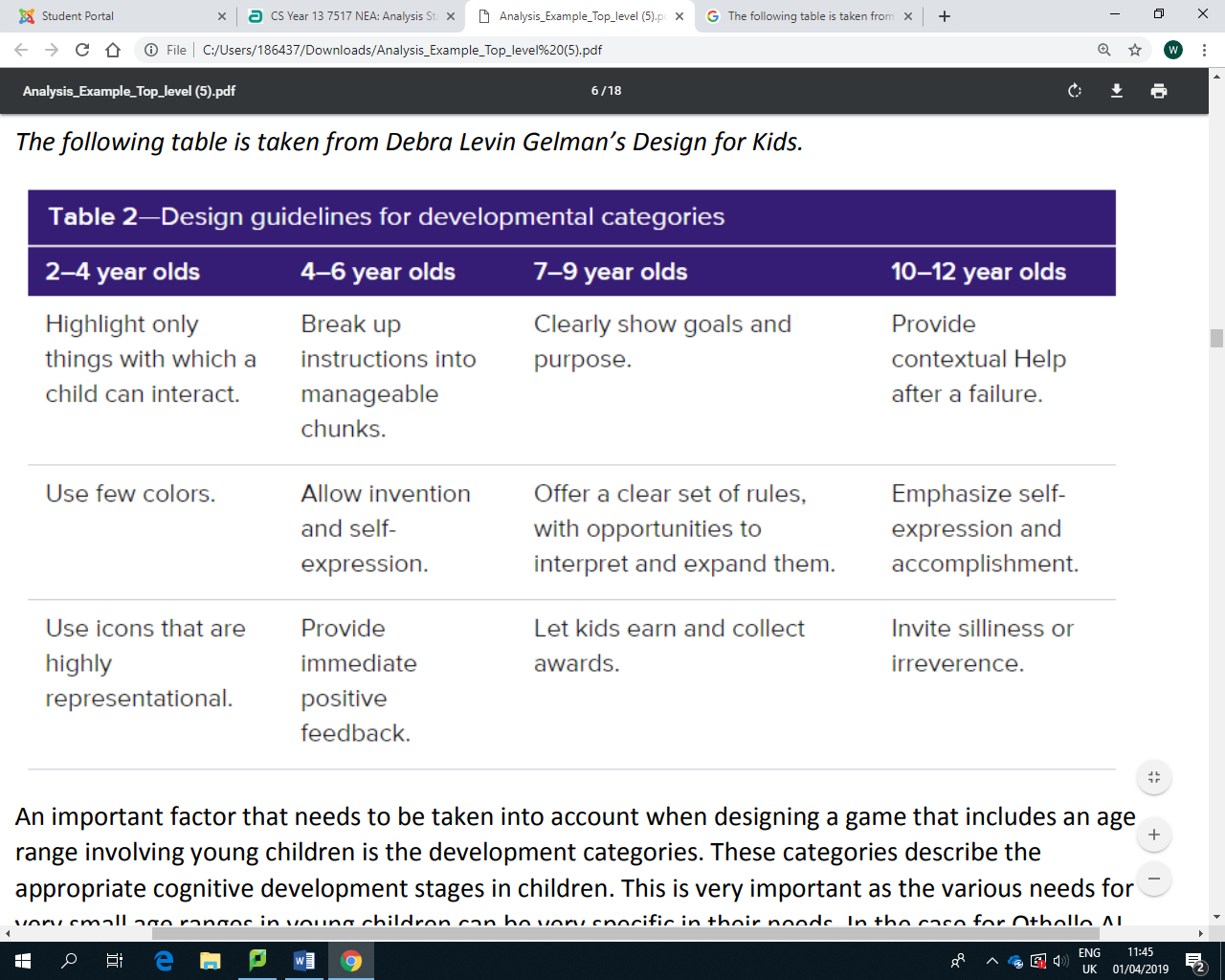
The winner of the game is the first to reach a predetermined number of points. You earn points as Pac-Man by eating ghosts, fruit, pellets and power pellets. Once Pac-Man is caught by the ghosts, whichever ghost caught Pac-Man will become Pac-Man in the following round. In order to balance the game,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

## 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. - Music that plays whenever the start screen is running.
   2. - A list of choices that become highlighted when a mouse hovers over them and can be clicked to choose the game mode.
   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
   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. - 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
   1. - Like the sign-up screen, this page should have input boxes, but this time only the ‘Username’ and ‘Password’ boxes are needed.
   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
   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.
   2. - Any changes made in the settings should be saved in a settings JSON file.
5. - Gameplay
   1. - The maze
      1. - The maze should be saved as a two-dimensional list in a text file or database.
      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.
      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.
      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. - When the user wins a level by eating all the pellets, the maze should flash white and blue.
   2. - Pac-Man
      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).
      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.
      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.
      4. - The classic ‘waka-waka’ sound should play only when Pac-Man is moving.
      5. - When moving, Pac-Man’s skin should also alternate between his slightly open and fully open mouth.
      6. - Pac-Man should also be able to travel through the tunnel at normal speed but not into the ghost hut.
   3. - Ghosts
      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.
      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.
      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.
      4. - The ghosts must only change direction when they are changing modes.
      5. - The ghosts must play a death sound when eaten.
   4. - Pellets
      1. - There should be two kinds of pellets: the normal pellet (small square skin), and the large power pellet (larger circular skin).
      2. - The small pellet should remain static, while the power pellets should flash.
      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.
      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
   1. - The tutorial mode must include a narrator that explains the game’s mechanics to the user.
      1. - This should be in the form of a scrolling message.
      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.
      3. - If the user presses space before the message has finished then the message should skip to the end and immediately fill the box.
   2. - The levels should progressively get harder and each should focus on a different aspect of the full game.
      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.
      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.
      3. - The third level should introduce Pinky.
      4. - The fourth level will include the most aggressive ghost: Blinky, and explain that when there are few pellets left, Blinky turns into Elroy.
      5. - Level five will introduce power pellets, as a way to counter the ghosts (specifically Blinky).
      6. - Level six will add the final ghost: ‘Inky’, who must be accompanied by Blinky due to his chase mechanic.
   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.

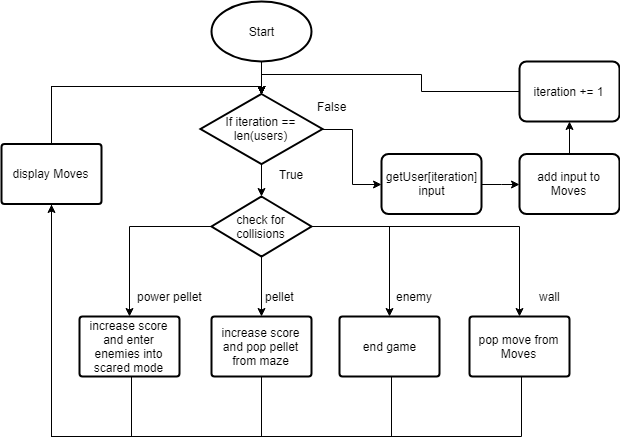
6.4 - After the scripted levels have finished the user should be able to play on randomly generated mazes.

1. - Classic Mode
   1. - The classic mode should have a high score at the top of the screen that is taken from the local database.
   2. - This should also see (like the original game) Pac-Man with just 3 lives (gaining a 4th at 10k pts).
   3. - The mode should feature just one maze (the original) in order to as closely resemble the original game as possible.
   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).
2. - High scores
   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).
   2. - This information should be formatted into 3 columns in the following order. Place (i.e. 1st, 2nd, 3rd), score then initials. This, again, is to stay true to the original game.
   3. - The top 3 scores should be coloured gold, silver and bronze.
   4. - When any key is pressed on this page, the user will be returned to the menu screen.

1. - Multiplayer
   1. - Multiplayer menu
      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.
      2. - There will be two options underneath this: ‘Create Game’ and ‘Join game’
   2. - Create Game menu
      1. - The central avatar should now be coloured in yellow (as Pac-Man).
      2. - The current user’s username should be displayed beneath the Pac-Man avatar, along with a score of 0.
      3. - This should instantiate a server object and thus allow other users to join the lobby.
      4. - There will be a box containing the game ID (host’s local IPV4 address) that other users can use to join the lobby.
      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).
      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.
   3. - Join Game menu
      1. - The avatars remain the same as the previous multiplayer menu.
      2. - There is an input box underneath the central avatar that the user can input a game ID into.
      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.
      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.
   4. - Gameplay
      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.
      2. - Each user should start as the avatar that was in their central box in the lobby.
      3. - Pac-Man should be able to gain points by collecting pellets and eating the other players.
      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.
      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.
      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.
      7. - 1st, 2nd and 3rd should be coloured gold, silver and bronze respectively.
2. - Database
   1. - The database should store information about every game that is played on a computer.
      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).
      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.
      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.
   2. - The database should store information about users.
      1. - When a user is created using the sign-up menu an entry into the ‘Users’ table should be added.
   3. - The database should be able to carry out complex queries.
      1. - We should be able to gather the top 10 scores (in order) in Classic games from the database.
      2. - The database should be able to take login details and verify them.
      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

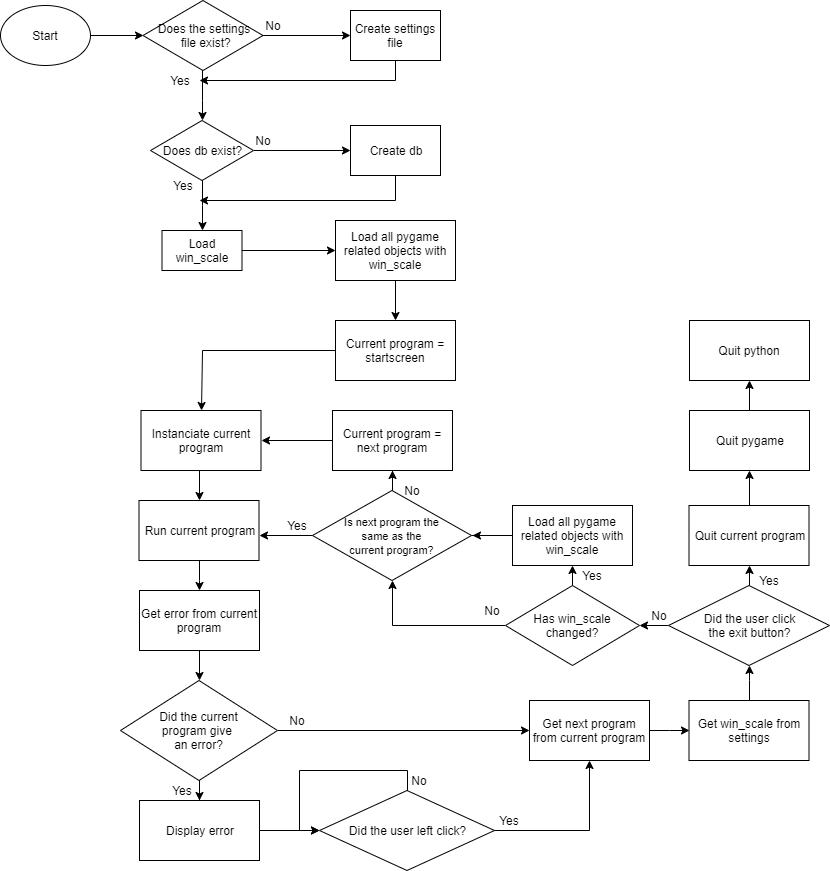
project. I will also be using the following libraries:

* 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.

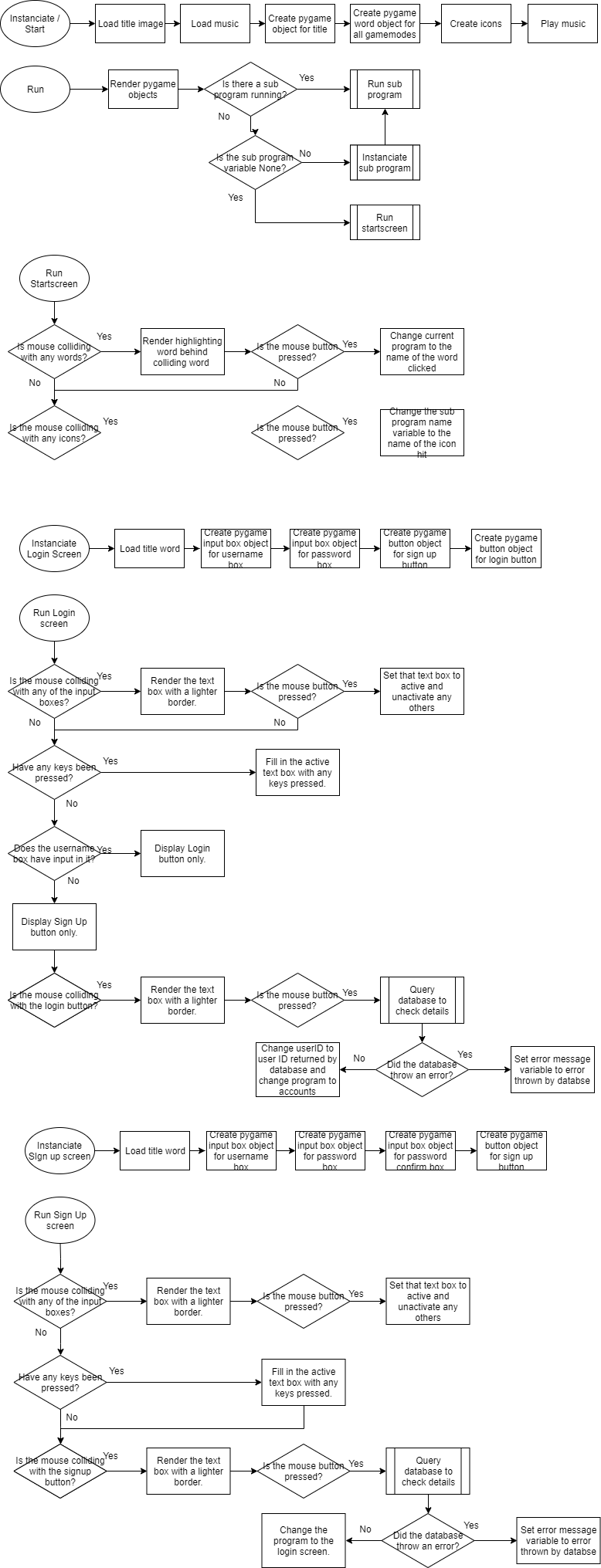
## High-level overview

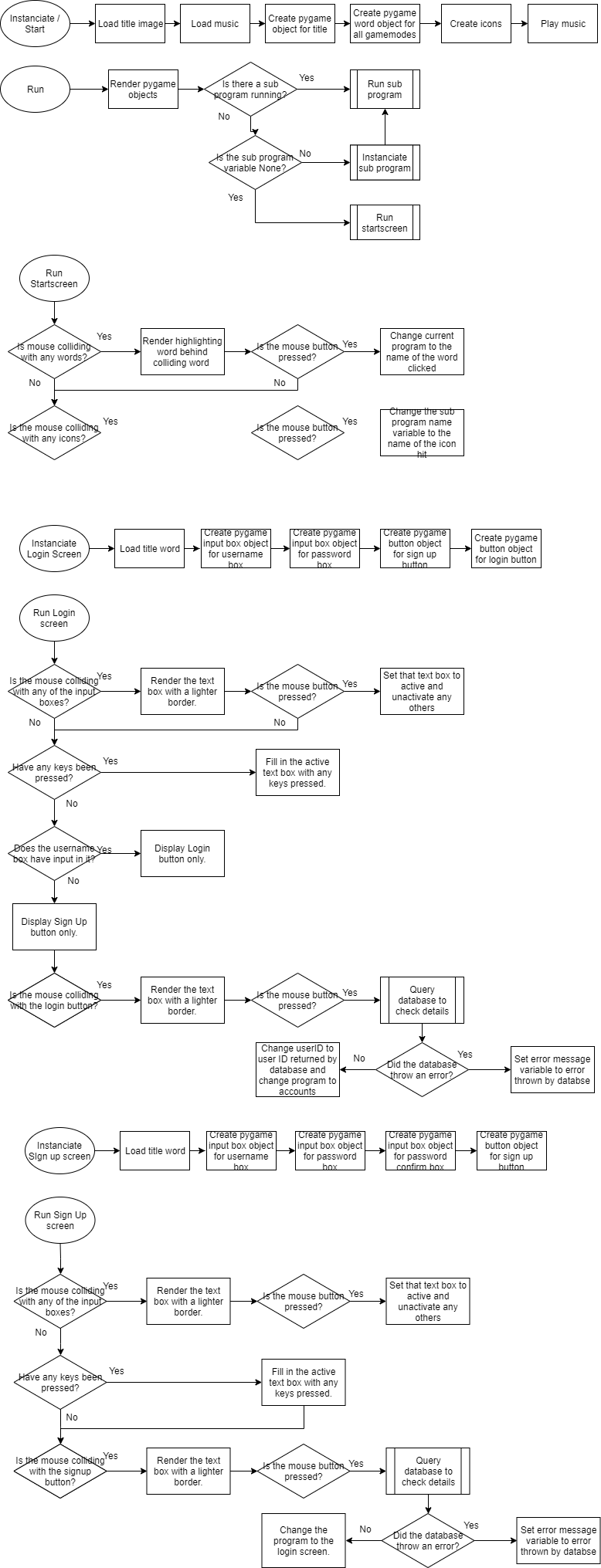
This section will contain flow charts outlining the key events and structure of my game. Due to the various menus and game modes, I separate each game mode and menu area into a separate chart.

### Mainloop

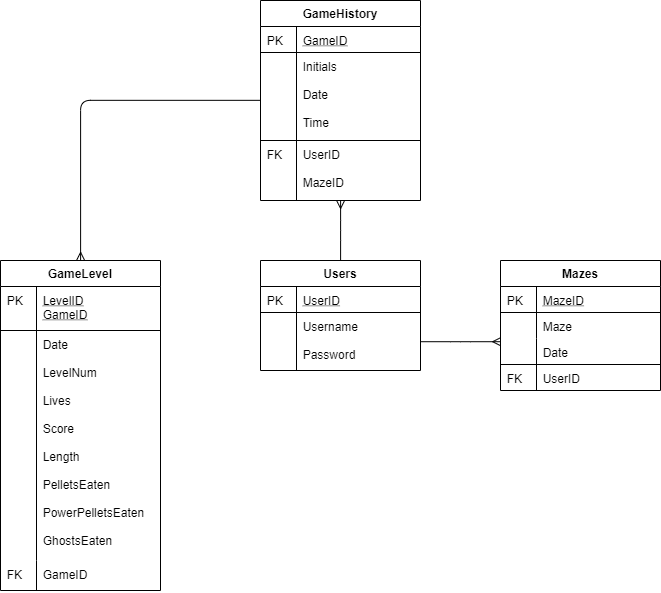


### Menu System





### 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.

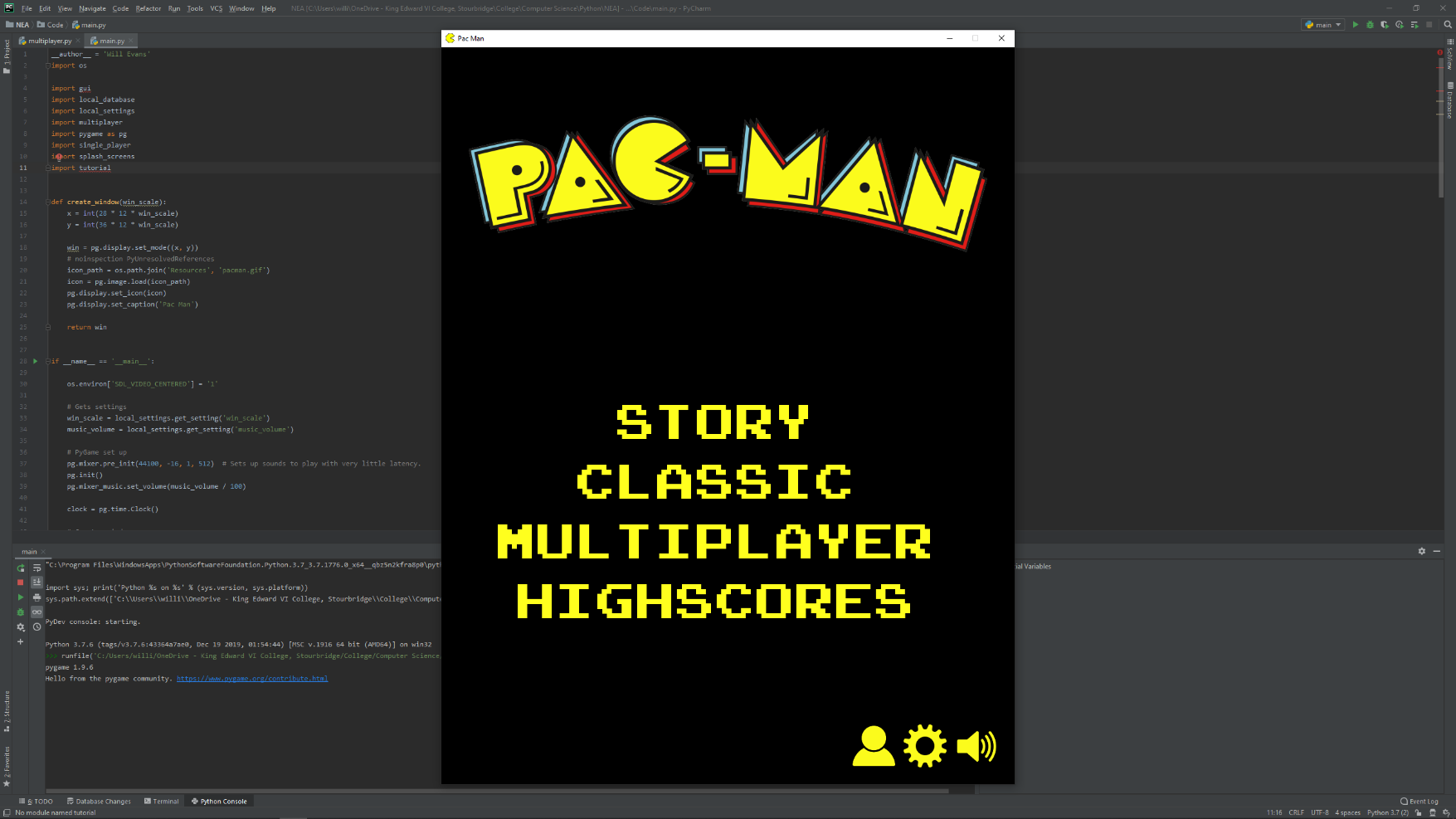
The final function is ‘get\_setting’, which simply retrieves a setting by again converting the file into a dictionary object and then returning the data corresponding to the settings requested.

## 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.



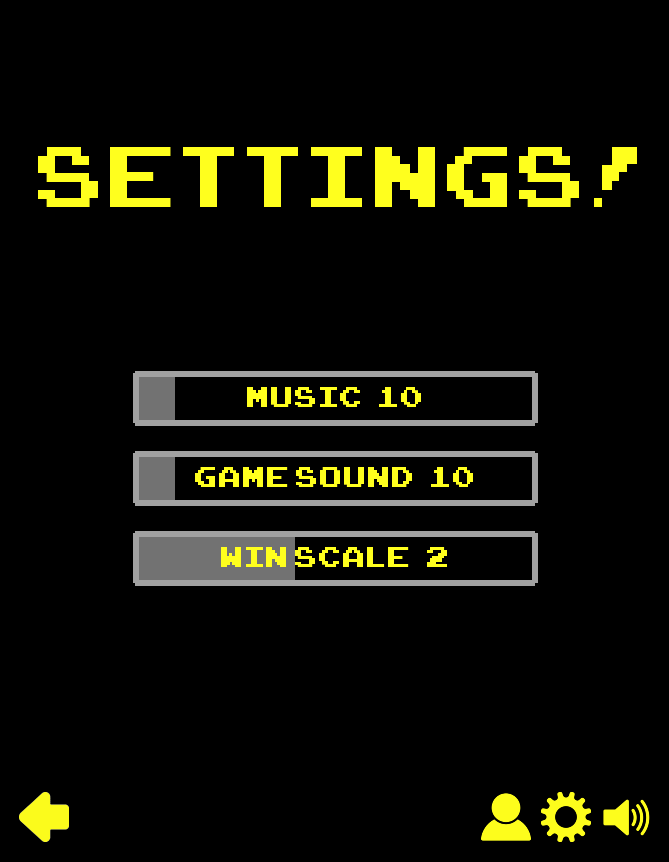
The options are interactive. When the user hovers over one with their mouse a highlighting colour will appear behind it. This alternates between blue and red as you go down in order to match the logo.

The 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 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.



High score.

Current score.

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

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.

### High Scores

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.



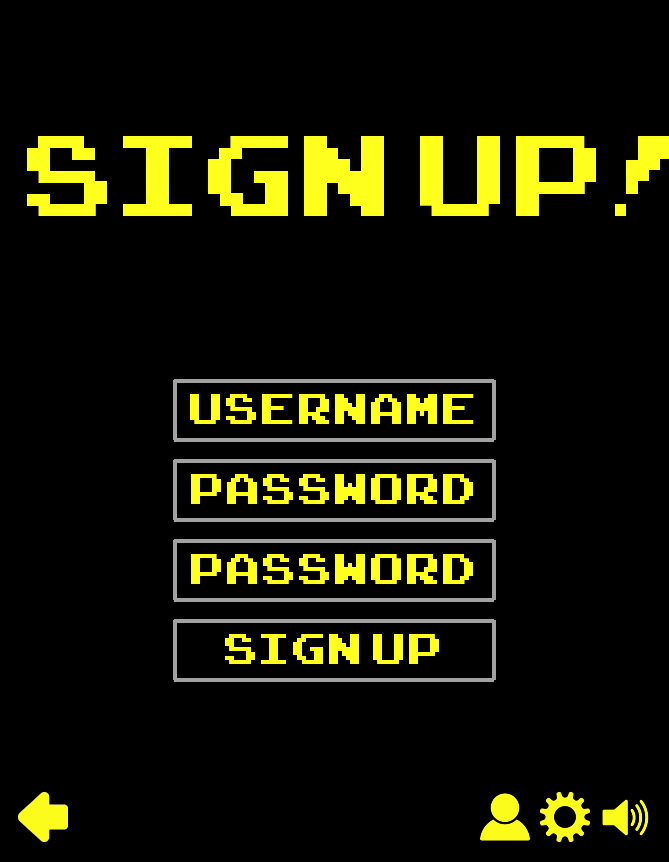
Large title that matches the majority of the other splash screens to maintain a continuity within my program.

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.

Back button to enable easy traversal of the menu system.

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.

### 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.

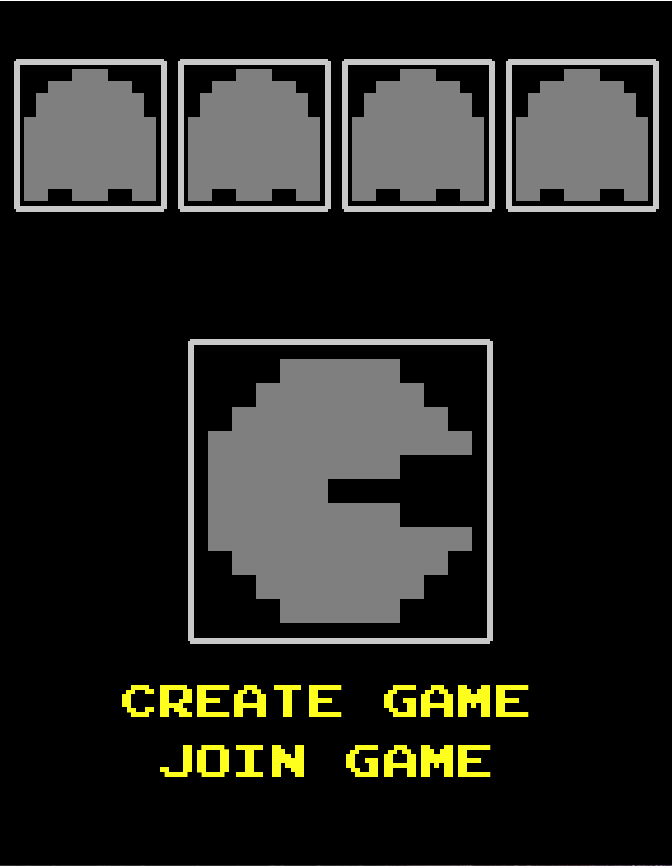
Back button to enable easy traversal of the menu system.

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.



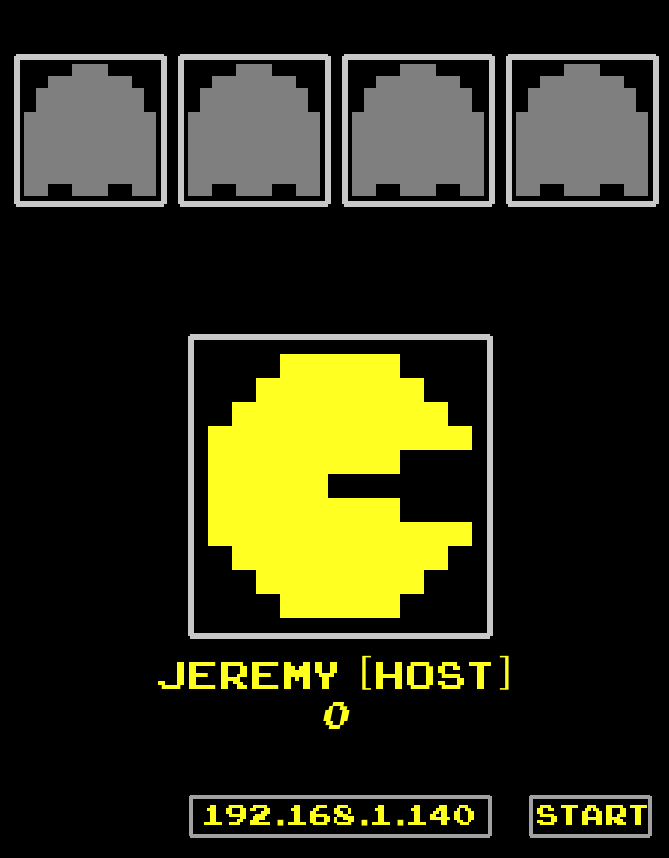
A single large box that will eventually contain the user’s character.

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

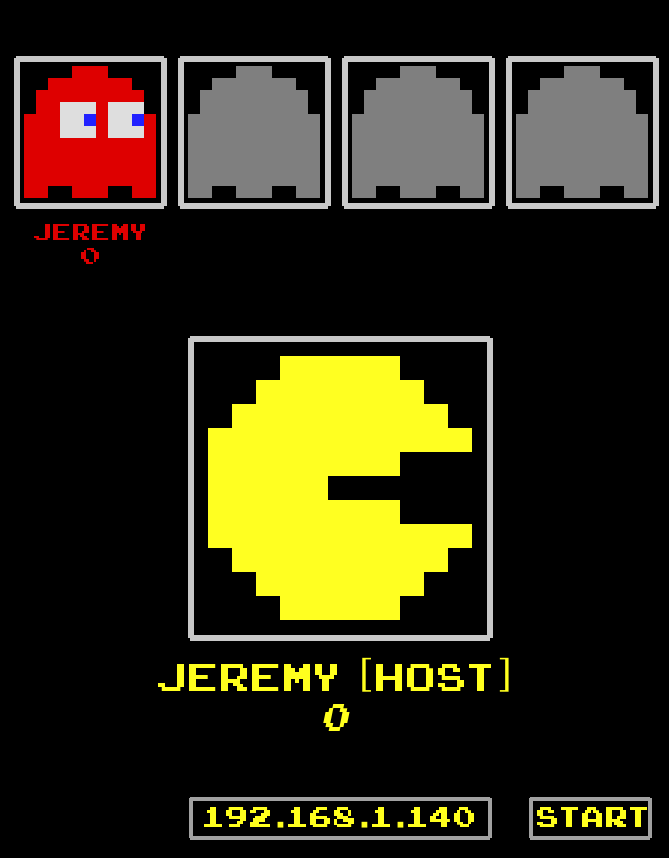
If the user selects the ‘Create Game’ option, they will be taken to their own lobby. Since they have entered the lobby their name appears under the central avatar, which has now become full colour. As they are the first person in the lobby they will get to play as Pac-Man. The ‘Start’ button has also 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.



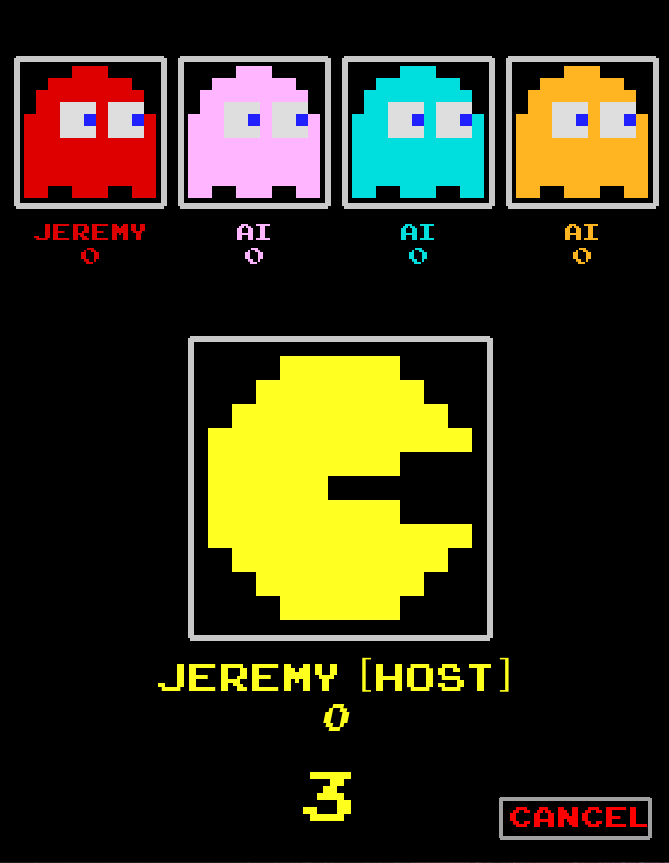
Start button that reacts to user’s mouse hovering to further convey that it is a clickable button.

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.

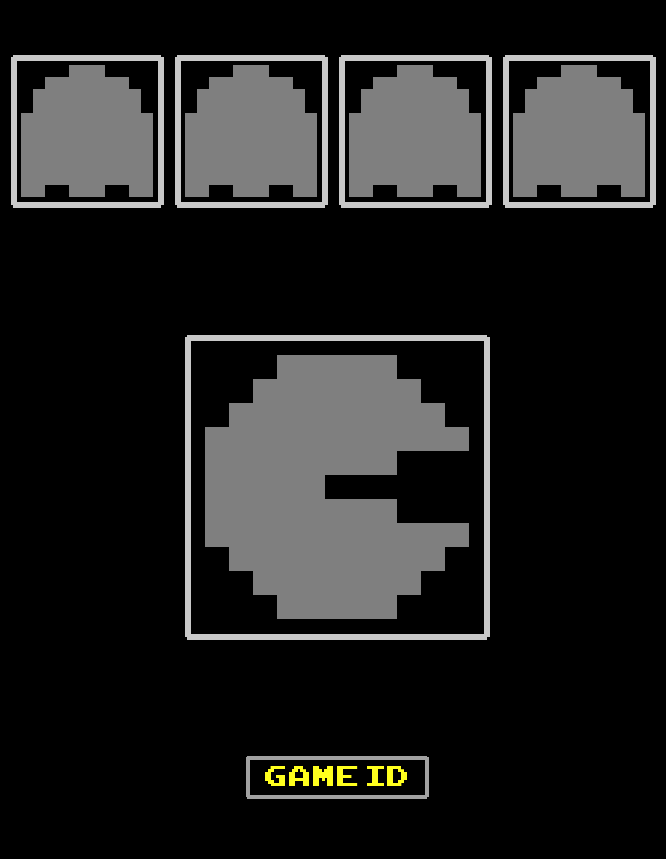


A countdown of 5 seconds is started which replaces the GameID box. This implies no other users can join as the GameID used to join is taken away- which is true at this point.

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

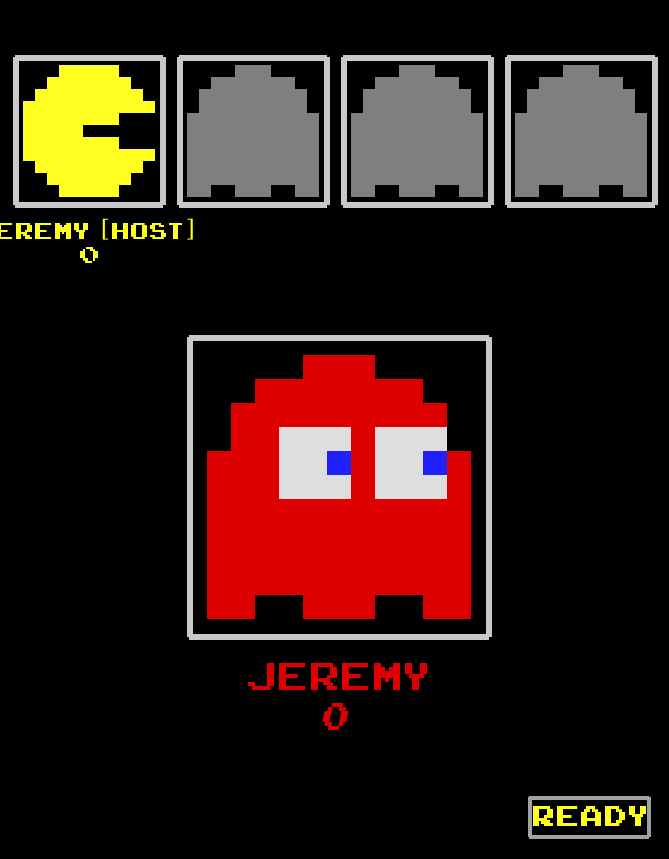
### Multiplayer – Client

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.



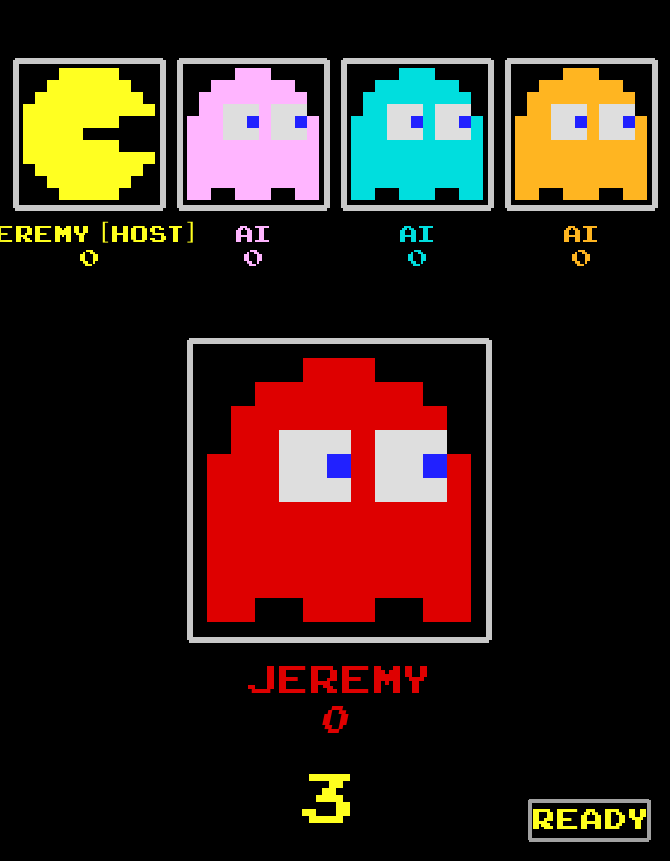
4 equally sized and space boxes containing greyed out ghost sprites. They will stay greyed out until someone else joins the lobby.

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.



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.

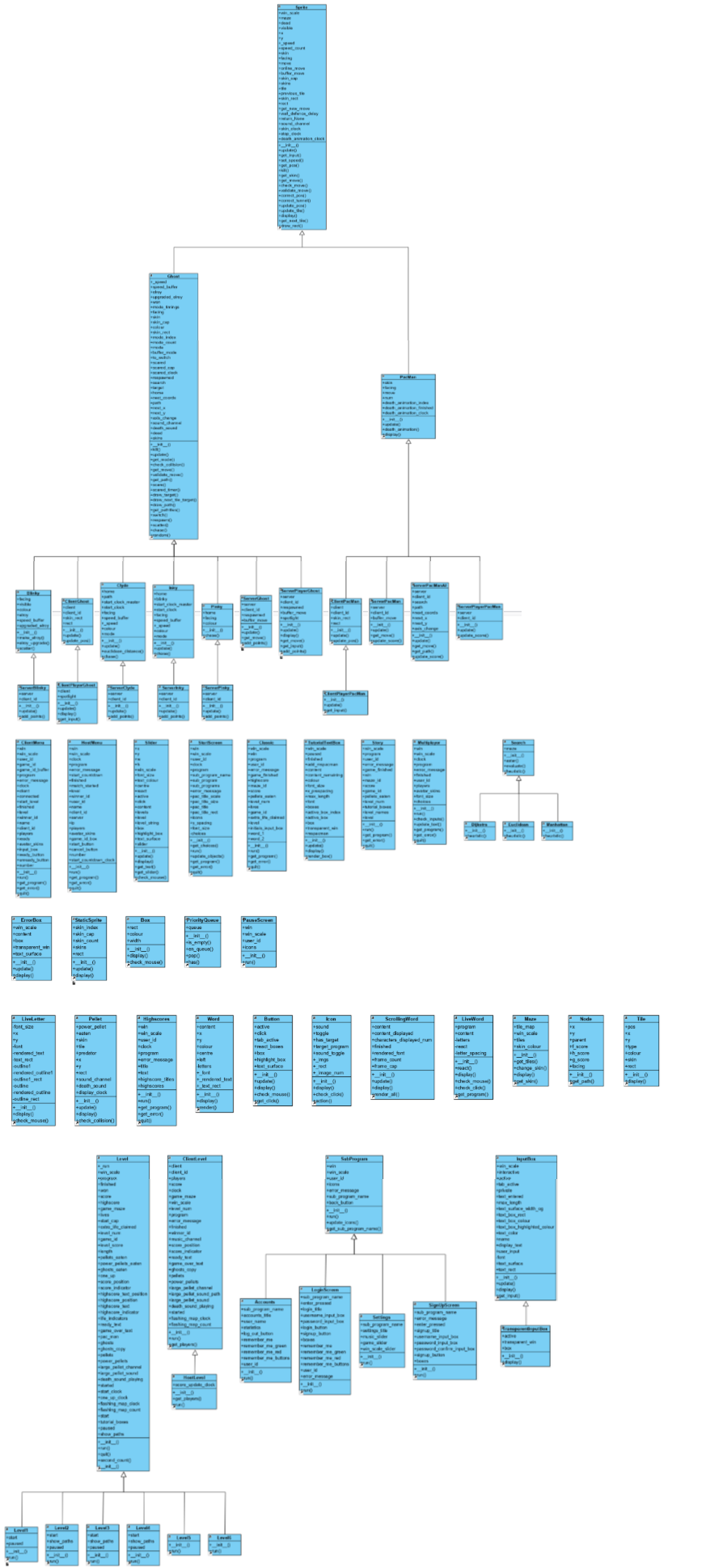
When the host starts the game, the other slots are filled up by AI and (to indicate extra players are in the game) the coloured sprites are revealed.

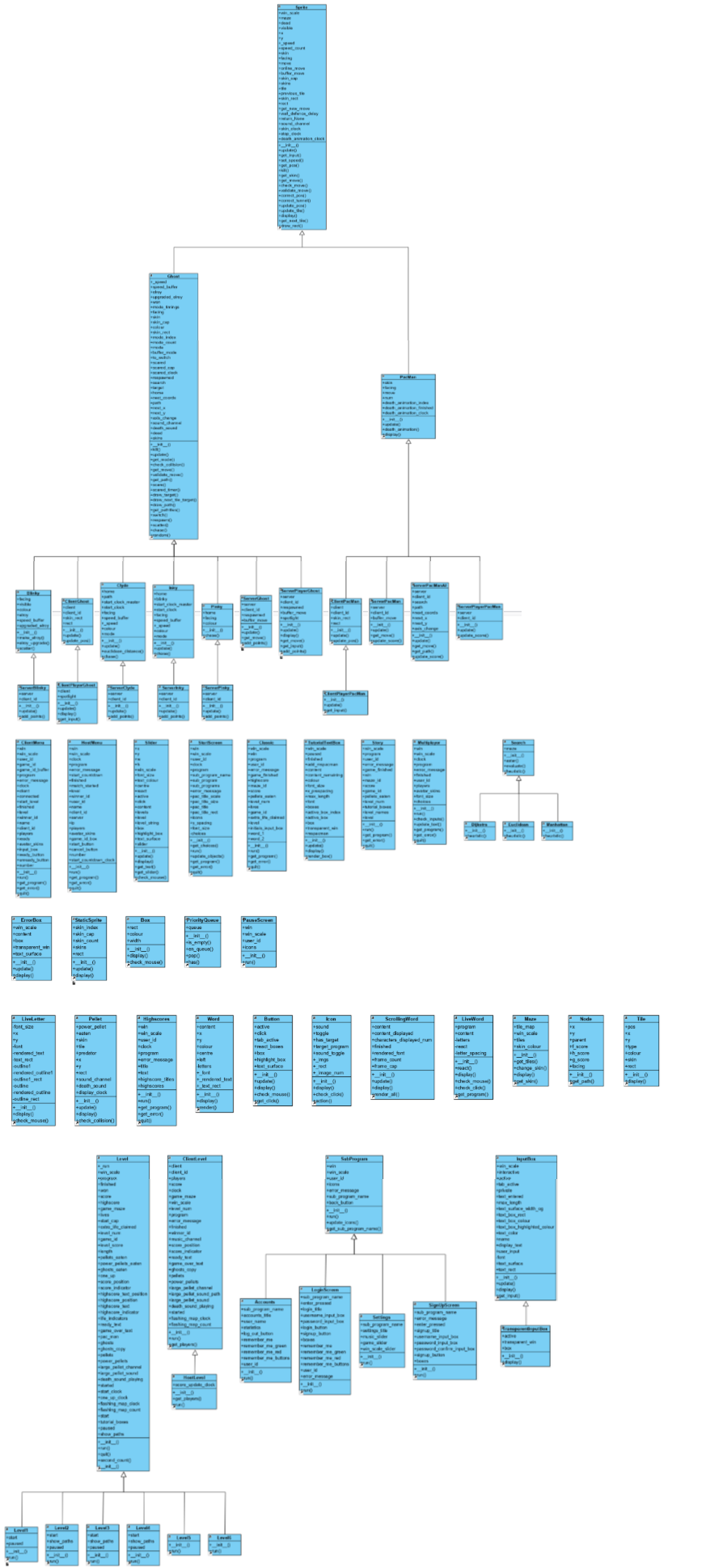
## Class Descriptions

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 180o 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:

1. **def** Manhattan(self, node, end):
2. x, y = end
3. **return** abs(node.x - x) + abs(node.y - y)

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

1. **def** get\_path(self, path):
2. **if** self.parent **is** **not** None:
3. path = self.parent.get\_path(path)
4. path.append((self.x, self.y))
5. **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

8. **class** Connection:
9. **def** \_\_init\_\_(self, user\_ip, conn, user\_id, players):
10. self.connected = True
11. self.\_\_player\_data = {
13. 'ready': None,
14. 'client\_move': None,
16. }
18. self.\_\_connected = False
19. self.\_\_IP = user\_ip
20. self.id = user\_id
21. self.\_\_PORT = 50007
22. self.\_\_conn = conn
24. self.send({
26. 'client\_id': user\_id,
27. 'players': players
29. })
31. self.\_\_player\_data['name'] = self.receive()['name']
33. threading.Thread(target=self.update).start()
35. **def** update(self):
36. **while** self.connected:
37. data = self.receive()
38. **if** data **is** **not** None:
39. **for** attribute, value **in** data.items():
40. self.\_\_player\_data[attribute] = value
42. **def** receive(self):
43. **try**:
44. data = self.\_\_conn.recv(4096)
45. **return** json.loads(data)
46. **except** Exception as e:
47. **print**("disconnected: {}".format(e))
49. **if** self.connected:
50. data = json.dumps(data)
51. data = bytes(data, 'utf-8')
52. **try**:
53. self.\_\_conn.sendall(data)
54. **except** ConnectionResetError:
55. **print**("disconnected")
56. self.connected = False
58. **def** get\_player\_data(self):
59. **return** self.\_\_player\_data
61. **def** get\_id(self):
62. **return** self.id
64. **def** close(self):
65. self.\_\_conn.close()
66. self.connected = False

69. **class** Server:
70. **def** \_\_init\_\_(self, name):
71. self.\_\_run = True
72. self.test\_count = 0
73. self.\_\_players\_template = {
75. 0:
76. {
77. 'name':     '{} [host]'.format(name),
78. 'skin':     'pac-man',
79. 'score':    0,
80. 'ready':    None,
81. 'pos':      [167, 318],
82. 'move':     None,
83. 'countdown': None,
84. 'start':    False,
85. 'end':      True,
86. 'place':    None,
87. 'finished': False
88. },
90. 1:
91. {
92. 'name':     None,
93. 'skin':     'blinky',
94. 'score':    0,
95. 'ready':    None,
96. 'pos':      [168, 176],
97. 'move':     None,
98. 'client\_move': None,
99. 'place': None
100. },
102. 2:
103. {
104. 'name':     None,
105. 'skin':     'pinky',
106. 'score':    0,
107. 'ready':    None,
108. 'pos':      [168, 214],
109. 'move':     None,
110. 'client\_move': None,
111. 'place': None
112. },
114. 3:
115. {
116. 'name':     None,
117. 'skin':     'inky',
118. 'score':    0,
119. 'ready':    None,
120. 'pos':      [144, 214],
121. 'move':     None,
122. 'client\_move': None,
123. 'place': None
124. },
125. 4:
126. {
127. 'name':     None,
128. 'skin':     'clyde',
129. 'score':    0,
130. 'ready':    None,
131. 'pos':      [192, 214],
132. 'move':     None,
133. 'client\_move': None,
134. 'place': None
135. }
136. }
138. self.\_\_players = copy.deepcopy(self.\_\_players\_template)
139. self.\_\_IP = self.get\_ip()
140. self.\_\_port = 50007
141. self.\_\_connections = []
142. self.\_\_s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)
143. self.\_\_s.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)
144. self.\_\_s.bind((self.\_\_IP, self.\_\_port))
146. self.has\_ai = False
148. 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()
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** None]
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))
166. **def** receive(self):
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)
178. **def** update\_data(self, client\_id, key, value):
179. self.\_\_players[client\_id][key] = value
181. **def** send\_data(self):
182. **for** connection **in** self.\_\_connections:
183. connection.send(self.\_\_players)
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[connection.get\_id()].copy()
191. self.\_\_connections.remove(connection)
192. time.sleep(1/120)
194. **def** get\_data(self, client\_id, type):
195. **return** self.\_\_players[client\_id][type]
197. **def** get\_players(self):
198. **return** self.\_\_players
200. **def** get\_ip(self):
201. **return** socket.gethostbyname(socket.gethostname())
203. **def** get\_client\_id(self):
204. **return** 0
206. **def** swap(self, client\_id):
207. pac\_man\_id = [id **for** id, data **in** self.\_\_players.items() **if** data['skin'] == 'pac-man'][0]
209. pac\_man\_skin = "{}".format(self.\_\_players[pac\_man\_id]['skin'])
210. client\_skin = "{}".format(self.\_\_players[client\_id]['skin'])
212. self.\_\_players[pac\_man\_id]['skin'] = client\_skin
213. self.\_\_players[client\_id]['skin'] = pac\_man\_skin
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]
221. client\_data['pos'] = og\_player\_data['pos'][::]
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
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
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()

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
250. self.\_\_player\_data = {
252. 'ready':        None,
253. 'client\_move':  None,
255. }
257. self.\_\_port = 50007
258. self.\_\_s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)
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
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})
273. self.connected = True
274. threading.Thread(target=self.update).start()
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")
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)
297. **def** update(self):
298. **while** self.connected:
299. data = self.receive()
300. **if** data **is** **not** None:
301. self.\_\_players = data
303. **def** update\_data(self, key, value):
304. self.\_\_player\_data[key] = value
306. **def** get\_data(self, client\_id, type):
307. **return** self.get\_players()[client\_id][type]
309. **def** send\_player\_data(self):
310. self.send(self.\_\_player\_data)
312. **def** get\_players(self):
313. **return** {int(k): v **for** k, v **in** self.\_\_players.items()}
315. **def** get\_client\_id(self):
316. **return** self.\_\_client\_id
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:
3. **def** \_\_init\_\_(self):
4. self.queue = []
6. **def** is\_empty(self):
7. **return** len(self.queue) == 0
9. **def** en\_queue(self, node):
10. self.queue.append(node)
12. **def** pop(self):
13. self.queue.sort(key=**lambda** x: x.f\_score)
14. **return** self.queue.pop(0)
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],

[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],

[1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1],

[1, 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, 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, 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, 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, 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, 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:

1. adjacent = {}
2. vectors = {'s': (0, 1),
3. 'e': (1, 0),
4. 'w': (-1, 0),
5. 'n': (0, -1),
6. 'ne': (1, -1),
7. 'se': (1, 1),
8. 'sw': (-1, 1),
9. 'nw': (-1, -1)
10. }
12. **for** key, value **in** vectors.items():
13. x, y = value
14. **try**:
15. new\_x = tile\_x + x
16. new\_y = tile\_y + y
18. **if** new\_x < 0 **or** new\_y < 0:
19. adjacent.update({key: None})
20. **continue**
22. adjacent.update({key: tile\_map[tile\_y + y][tile\_x + x]})
23. **except** IndexError:
24. adjacent.update({key: None})
26. values = tuple(list(adjacent.values())[:4])
27. values\_diag = tuple(adjacent.values())
29. # ghost edges
30. **if** values == (4, 3, 1, 4):
31. **return** 'left\_end\_ghost'
33. **if** values == (4, 1, 3, 4):
34. **return** 'right\_end\_ghost'
36. **if** values == (4, 1, 1, 4) **and** tile\_y == 12:
37. **return** 'lower\_boundary'
39. **if** values == (4, 1, 1, 4) **and** tile\_y == 16:
40. **return** 'upper\_boundary'
42. **if** values == (1, 4, 4, 1) **and** tile\_x == 10:
43. **return** 'right\_boundary'
45. **if** values == (1, 4, 4, 1) **and** tile\_x == 17:
46. **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

1. \_\_author\_\_ = 'Will Evans'
3. **import** pygame as pg
4. **import** splash\_screens
5. **import** tutorial
6. **import** single\_player
7. **import** multiplayer
8. **import** local\_database
9. **import** local\_settings
10. **import** text
11. **import** os

14. **def** create\_window(win\_scale):
15. x = int(28 \* 12 \* win\_scale)
16. y = int(36 \* 12 \* win\_scale)
18. win = pg.display.set\_mode((x, y))
19. # noinspection PyUnresolvedReferences
20. icon\_path = os.path.join('Resources', 'pacman.gif')
21. icon = pg.image.load(icon\_path)
22. pg.display.set\_icon(icon)
23. pg.display.set\_caption('Pac Man')
25. **return** win

28. **if** \_\_name\_\_ == '\_\_main\_\_':
30. os.environ['SDL\_VIDEO\_WINDOW\_POS'] = "{}, {}.".format(25, 50)
32. # Gets settings
33. win\_scale = local\_settings.get\_setting('win\_scale')
34. music\_volume = local\_settings.get\_setting('music\_volume')
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)
41. clock = pg.time.Clock()
43. # Creates window
44. win = create\_window(win\_scale)
46. # Creates database
47. local\_database.create\_db()
49. run = True
51. # List of programmes that can be run directly from this script
52. programmes = {
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
62. }
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)
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)
74. # Information on error messages (these must be stored in the main script).
75. error\_message = None
76. error\_box = None
77. buffer = program
79. # Mainloop
80. **while** run:
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)
91. # Hand control to another program
92. **if** buffer != program:
93. buffer = program
94. running\_program.quit()
95. pg.mixer.stop()
96. running\_program = program(win, win\_scale, user\_id)
98. # See if the user wants to close the application
99. events = pg.event.get()
100. **for** event **in** events:
101. **if** event.type == pg.QUIT:
102. run = False
103. running\_program.quit()
105. # Error message
106. **if** error\_message **is** **not** None **and** error\_box **is** None:
107. error\_box = text.ErrorBox(error\_message, win\_scale)
108. error\_box.display(win)
110. **if** error\_box **is** None:
111. running\_program.run(win, events)
112. **else**:
113. **if** **not** error\_box.update(events):
114. error\_box = None
115. error\_message = None
116. running\_program.error\_message = None
118. # Get information for next cycle
119. **try**:
120. program = programmes[running\_program.get\_program()]
121. error\_message = running\_program.get\_error()
122. user\_id = running\_program.user\_id
123. **except** KeyError:
124. error\_message = 'Not found'
125. program = programmes['StartScreen']
126. buffer = None
128. # Pygame Essential
129. clock.tick(60)
130. pg.display.update()
132. **if** error\_box **is** None **and** error\_message **is** None:
133. win.fill((0, 0, 0))
135. pg.quit()
136. quit()

### Splash Screens

1. \_\_author\_\_ = 'Will Evans'
3. **import** pygame as pg
4. **import** text
5. **import** icons
6. **import** local\_settings
7. **import** local\_database
8. **import** os

11. **class** StartScreen:
12. **def** \_\_init\_\_(self, win, win\_scale, user\_id):
13. """
14. Displays the StartScreen and keeps track of all the objects on the screen and user interactions with them.
15. :param win: The current window, all objects must be blitted to this window to be displayed.
16. :param win\_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
17. :param user\_id: UserID if the user has signed in (they don't have to be signed in for the startscreen).
18. """
20. # Essential Information
21. self.win = win
22. self.win\_scale = win\_scale
23. self.user\_id = user\_id
24. self.clock = pg.time.Clock()
25. self.program = 'StartScreen'
26. self.sub\_program\_name = None
27. self.sub\_program = None
28. self.sub\_programs = {
30. 'loginscreen':  LoginScreen,
31. 'signupscreen': SignUpScreen,
32. 'settings':     Settings,
33. 'accounts':     Accounts
35. }
37. self.error\_message = None
39. # Pac-Man logo
40. # Change this into a class
41. pac\_man\_logo\_path = os.path.join('resources', 'pac\_man\_logo.png')
42. self.pac\_title\_scale = 80 \* win\_scale
43. self.pac\_title\_size = (int(self.pac\_title\_scale \* 3.8), self.pac\_title\_scale)
44. self.pac\_title = pg.transform.smoothscale(pg.image.load(pac\_man\_logo\_path), self.pac\_title\_size)
45. self.pac\_title\_rect = self.pac\_title.get\_rect(center=(168 \* win\_scale, 80 \* win\_scale))
47. # Music
48. theme\_music\_path = os.path.join('resources', 'sounds', 'startscreen', 'theme.mp3')
49. pg.mixer.music.load(theme\_music\_path)
51. # Icons
52. self.icons = []
54. #   Sound Icon
55. sound\_img\_paths = [
56. os.path.join('resources', 'icons', 'unmute.png'),
57. os.path.join('resources', 'icons', 'mute.png')
58. ]
59. self.icons.append(icons.Icon((326, 422), sound\_img\_paths, win\_scale, sound=True, toggle=True))
61. #   Settings Icon
62. settings\_img\_path = [os.path.join('resources', 'icons', 'settings.png')]
63. self.icons.append(icons.Icon((296, 422), settings\_img\_path, win\_scale, target\_program='settings'))
65. #   Accounts Icon
66. accounts\_img\_path = [os.path.join('resources', 'icons', 'accounts.png')]
67. self.icons.append(icons.Icon((266, 422), accounts\_img\_path, win\_scale, target\_program='loginscreen'))
69. self.y\_spacing = 30
70. self.font\_size = 40
72. # Choices
73. self.\_\_choices = self.get\_choices(['Story', 'Classic', 'Multiplayer', 'Creator', 'Highscores'], win\_scale)
75. # Actions
76. pg.mixer.music.play()
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 that the user can
82. :param string\_choices: A list of strings with the names of the possible choices/options that the user can
83. choose.
84. :param win\_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
85. :return: List of options.
86. """
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
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. """
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)
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()
116. **if** program\_name **is** None:
117. self.sub\_program\_name = None
118. self.sub\_program = None
120. **elif** program\_name != self.sub\_program\_name:
121. self.sub\_program\_name = program\_name
122. self.sub\_program = None
124. **else**:
125. **for** text **in** self.\_\_choices:
126. **if** text.check\_mouse(\*pg.mouse.get\_pos()):
127. text.react()
129. self.update\_objects(events)
131. win.blit(self.pac\_title, self.pac\_title\_rect)
133. **for** word **in** self.\_\_choices:
134. word.display(win)
136. **for** icon **in** self.icons:
137. icon.display(win)
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. """
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
155. **for** word **in** self.\_\_choices:
156. **if** word.check\_click(\*pos):
157. self.program = word.get\_program()
159. **def** get\_program(self):
160. **return** self.program
162. **def** get\_error(self):
163. **return** self.error\_message
165. **def** quit(self):
166. pg.mixer.music.stop()

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 / update 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 displayed.
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 to play Classic).
177. :param icons\_list: List of icon objects from the StartScreen.
178. """
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
187. # Defined in subclasses
188. self.sub\_program\_name = None
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\_program=None)
194. **def** run(self, win, events):
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
202. **for** icon **in** self.icons:
203. icon.display(win)
204. self.back\_button.display(win)
205. self.update\_icons(events)
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
219. **def** get\_sub\_program\_name(self):
220. **return** self.sub\_program\_name

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 boxes 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 displayed.
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 to play Classic).
231. :param icons: List of icon objects from the StartScreen.
232. """
234. super().\_\_init\_\_(win, win\_scale, user\_id, icons)
236. self.sub\_program\_name = 'signupscreen'
237. self.error\_message = None
239. self.enter\_pressed = False
241. # Sign Up Title
242. self.signup\_title = text.Word('Sign Up!', (188, 90), (255, 255, 30), 80, win\_scale, centre=True)
244. # Username Input Box
245. self.username\_input\_box = text.InputBox(168, 206, 160, 30, 30, win\_scale, 'Username', centre=True)
247. # Password Input Box
248. self.password\_input\_box = text.InputBox(168, 246, 160, 30, 30, win\_scale, 'Password', centre=True, private=True)
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)
257. # Sign Up Button
258. self.signup\_button = text.Button('Sign Up',
259. (168, 326),
260. (160, 30),
261. 30,
262. (255, 255, 30),
263. 2,
264. win\_scale,
265. centre=True)
267. self.boxes = [
268. self.username\_input\_box,
269. self.password\_input\_box,
270. self.password\_confirm\_input\_box,
271. self.signup\_button
272. ]
274. **def** run(self, win, events):
275. super().run(win, events)
277. # Updates
278. **for** box **in** self.boxes:
279. box.update(events)
281. #   Sign up
282. self.signup\_button.update(events)
284. # Display
285. self.signup\_title.display(win)
287. **for** box **in** self.boxes:
288. box.display(win)
290. self.signup\_button.display(win)
292. **if** self.signup\_button.get\_click() **or** self.enter\_pressed:
293. self.enter\_pressed = False
294. approved, self.error\_message = local\_database.check\_sign\_up(
295. self.username\_input\_box.user\_input.lower(),
296. self.password\_input\_box.user\_input,
297. self.password\_confirm\_input\_box.user\_input
298. )
299. **if** approved:
300. self.sub\_program\_name = 'loginscreen'
301. local\_database.save\_user(self.username\_input\_box.user\_input.lower(), self.password\_input\_box.user\_input)
303. # Events
304. **for** event **in** events:
305. **if** event.type == pg.KEYDOWN:
306. # Allows the user to cycle through input boxes using tab
307. **if** event.key == pg.K\_TAB:
308. **try**:
309. active\_box = [box **for** box **in** self.boxes **if** box.tab\_active][0]
310. active\_box\_index = self.boxes.index(active\_box)
311. active\_box.tab\_active = False
312. active\_box.active = False
313. self.boxes[(active\_box\_index + 1) % len(self.boxes)].tab\_active = True
314. **except** IndexError:
315. self.boxes[0].tab\_active = True
316. **elif** event.key == pg.K\_RETURN:
317. self.enter\_pressed = True

320. **class** LoginScreen(SubProgram):
321. **def** \_\_init\_\_(self, win, win\_scale, user\_id, icons):
322. """
323. Class for the login screen.
324. :param win: The current window, all objects must be blitted to this window to be displayed.
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 to play Classic).
327. :param icons: List of icon objects from the StartScreen.
328. """
330. # Essential
331. super().\_\_init\_\_(win, win\_scale, user\_id, icons)
332. self.sub\_program\_name = 'loginscreen' **if** user\_id **is** None **else** 'accounts'
334. self.enter\_pressed = False
336. # Login Title
337. self.login\_title = text.Word('Login!', (188, 90), (255, 255, 30), 80, win\_scale, centre=True)
339. # Username Input Box
340. self.username\_input\_box = text.InputBox(168, 206, 150, 30, 30, win\_scale, 'Username', centre=True)
342. # Password Input Box
343. self.password\_input\_box = text.InputBox(168, 246, 150, 30, 30, win\_scale, 'Password', centre=True, private=True)
345. # Login Button
346. self.login\_button = text.Button('Login', (168, 286), (160, 30), 30, (255, 255, 30), 2, win\_scale, centre=True)
348. # SignUp Button
349. self.signup\_button = text.Button('SignUp', (168, 286), (160, 30), 30, (255, 255, 30), 2, win\_scale, centre=True)
351. self.boxes = [self.username\_input\_box, self.password\_input\_box, self.signup\_button]
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)
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)
373. self.remember\_me\_buttons = [self.remember\_me\_red, self.remember\_me\_green]
375. **def** run(self, win, events):
376. super().run(win, events)
378. # Updates
379. **for** box **in** self.boxes:
380. box.update(events)
382. **if** self.password\_input\_box.text\_entered:
383. self.boxes = [self.username\_input\_box, self.password\_input\_box, self.login\_button]
385. **else**:
386. self.boxes = [self.username\_input\_box, self.password\_input\_box, self.signup\_button]
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'
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'
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()
413. # Display
414. self.login\_title.display(win)
416. **for** box **in** self.boxes:
417. box.display(win)
419. self.remember\_me\_buttons[0].display(win)
421. # Events
422. **for** event **in** events:
423. **if** event.type == pg.KEYDOWN:
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.active)][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 = True
433. **except** IndexError:
434. self.boxes[0].tab\_active = True
435. **elif** event.key == pg.K\_RETURN:
436. self.enter\_pressed = True

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 reloads 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 to play Classic).
447. :param icons: List of icon objects from the StartScreen.
448. """
450. # Essential
451. super().\_\_init\_\_(win, win\_scale, user\_id, icons)
452. self.sub\_program\_name = 'settings'
454. # Settings Title
455. self.settings\_title = text.Word('Settings!', (178, 90), (255, 255, 30), 60, win\_scale, centre=True)
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))
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))
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\_scale')),
477. centre=True,
478. levels=5))
480. **def** run(self, win, events):
481. super().run(win, events)
483. # Events
484. **for** event **in** events:
485. **if** event.type == pg.KEYDOWN:
486. **if** event.key == pg.K\_ESCAPE:
487. **return** None
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)
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)
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\_string:
503. local\_settings.save\_setting('win\_scale', self.win\_scale\_slider.level\_string)
505. # Display
506. self.settings\_title.display(win)
508. **for** icon **in** self.icons:
509. icon.display(win)
511. self.music\_slider.display(win)
512. self.game\_slider.display(win)
513. self.win\_scale\_slider.display(win)

516. **class** Accounts(SubProgram):
517. **def** \_\_init\_\_(self, win, win\_scale, user\_id, icons):
519. # Essential
520. super().\_\_init\_\_(win, win\_scale, user\_id, icons)
521. self.sub\_program\_name = 'accounts'
523. # Login Title
524. self.accounts\_title = text.Word('Accounts!', (180, 90), (255, 255, 30), 60, win\_scale, centre=True)
526. # Logged in as
527. self.user\_name = text.Word(
528. 'Logged in as:       {}'.format(local\_database.get\_username(user\_id)),
529. (310, 120),
530. (255, 255, 30),
531. 15,
532. win\_scale,
534. )
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\_scale, centre=True))
544. self.statistics.append(text.Word(string\_2, (336 / 2, 270), colour, 25, win\_scale, 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))
552. self.statistics.append(text.Word(str(stat), (311, 240 + 25 \* num), colour, 20, win\_scale))
554. # Log out button
555. self.log\_out\_button = text.Button('Log Out', (47, 400), (75, 20), 20, (255, 255, 30), 2, win\_scale)
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)
565. **def** run(self, win, events):
566. super().run(win, events)
568. # Updates
569. self.log\_out\_button.update(events)
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)
578. # Display
579. self.accounts\_title.display(win)
580. self.user\_name.display(win)
581. self.remember\_me\_buttons[0].display(win)
583. **for** stat **in** self.statistics:
584. stat.display(win)
585. self.log\_out\_button.display(win)
587. # Events
588. **if** self.log\_out\_button.get\_click():
589. self.user\_id = None
590. self.sub\_program\_name = 'loginscreen'

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 displayed.
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 to play Classic).
600. :param icons: List of icon objects from the StartScreen.
601. """
603. # Essential
604. self.win = win
605. self.win\_scale = win\_scale
606. self.user\_id = user\_id
607. self.icons = icons
609. **def** run(self, win, events):
611. # Events
612. **for** event **in** events:
613. **if** event.type == pg.KEYDOWN:
614. **if** event.key == pg.K\_ESCAPE:
615. **return** None

618. **def** get\_mouse\_input(events):
619. **for** event **in** events:
620. **if** event.type == pg.MOUSEBUTTONUP:
621. **return** event.pos

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 displayed.
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 to play Classic).
631. """
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
641. # Title
642. self.title = text.Word('Highscores!', (175, 90), (255, 255, 30), 45, win\_scale, centre=True)
643. self.text = text.Word('Press any key to return...', (326, 417), (255, 255, 30), 15, win\_scale)
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]
652. self.highscores = []
653. place\_details = {
654. 0:
655. {
656. 'string': '1st',
657. 'colour': (255, 215, 0)
658. },
660. 1:
661. {
662. 'string': '2nd',
663. 'colour': (220, 220, 220)
664. },
666. 2:
667. {
668. 'string': '3rd',
669. 'colour': (205, 127, 50)
670. }
671. }
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)
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\_scale))
685. **if** num **is** 9:
686. **break**
688. **def** run(self, win, events):
690. # Events
691. **for** event **in** events:
692. **if** event.type **in** [pg.KEYDOWN, pg.MOUSEBUTTONUP]:
693. self.program = 'StartScreen'
695. # Display
696. self.title.display(win)
698. **for** title **in** self.highscore\_titles:
699. title.display(win)
701. **for** word **in** self.highscores:
702. word.display(win)
704. self.text.display(win)
706. **def** get\_program(self):
707. **return** self.program
709. **def** get\_error(self):
710. **return** self.error\_message
712. **def** quit(self):
713. **pass**

716. **if** \_\_name\_\_ == '\_\_main\_\_':
717. **pass**

### Tutorial

1. \_\_author\_\_ = 'Will Evans'
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

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 related variables).
21. :param user\_id: UserID if the user has signed in (they don't have to be signed in to play Classic).
22. """
24. # Essential
25. self.win\_scale = win\_scale
26. self.program = 'Story'
27. self.user\_id = user\_id
28. self.error\_message = None
29. self.game\_finished = False
30. self.win = win
32. # Maze
33. self.maze\_id = 1
35. # Variables
36. self.score = 0
38. #   Database
39. self.game\_id = local\_database.get\_game\_id(self.user\_id, self.maze\_id)
40. self.pellets\_eaten = 0
42. # Level Info
43. self.level\_num = 3
45. # Tutorial Messages
47. tutorial\_prompts\_file\_path = os.path.join('data', 'tutorial.json')
48. with open(tutorial\_prompts\_file\_path, 'r') as file:
49. self.tutorial\_prompts = json.load(file)
51. # Boxes for the first level
52. self.tutorial\_boxes = []
53. **for** prompt **in** self.tutorial\_prompts[str(self.level\_num)]:
54. self.tutorial\_boxes.append(text.TutorialTextBox(prompt, (255, 255, 30), win\_scale, add\_mspacman=True))
55. # Level
56. self.level\_names = {
58. 1:    Level1,
59. 2:    Level2,
60. 3:    Level3,
61. 4:    Level4,
62. 5:    Level5,
63. 6:    Level6,
65. }
67. self.level = self.level\_names[self.level\_num](
68. self.win\_scale,
69. self.game\_id, self.level\_num,
70. self.maze\_id,
71. [],
72. [],
73. self.score,
74. self.tutorial\_boxes,
75. )
77. **def** run(self, win, events):
78. """
79. This method is run directly from the main script.
80. :param win: The current window, all objects must be blitted to this window to be displayed.
81. :param events: Contains events from the pg.event.get() call containing all keyboard events.
82. :return: None
83. """
85. # Essential
86. self.level.run(win, events)
88. # Variables
89. self.score = self.level.score
90. self.pellets\_eaten = self.level.pellets\_eaten
92. # Events
93. **for** event **in** events:
94. **if** event.type == pg.KEYDOWN:
95. **if** event.key == pg.K\_ESCAPE:
96. self.program = 'StartScreen'
98. # If all the pellets have been eaten
99. **if** self.level.finished **and** self.level.won:
100. self.level\_num += 1
101. self.level.quit()
103. # Tutorial Messages
104. self.tutorial\_boxes = []
105. **for** prompt **in** self.tutorial\_prompts[str(self.level\_num)]:
106. self.tutorial\_boxes.append(text.TutorialTextBox(prompt,
107. (255, 255, 30),
108. self.win\_scale,
109. add\_mspacman=True)
110. )
112. self.level = self.level\_names[self.level\_num](
113. self.win\_scale,
114. self.game\_id,
115. self.level\_num,
116. self.maze\_id,
117. [],
118. [],
119. self.score,
120. self.tutorial\_boxes,
121. )
123. # If Pac-Man has died
124. **elif** self.level.finished:
125. self.score = self.level.score
126. self.tutorial\_boxes = self.level.tutorial\_boxes
127. self.level.quit()
129. self.level = self.level\_names[self.level\_num](
130. self.win\_scale,
131. self.game\_id,
132. self.level\_num,
133. self.maze\_id,
134. self.level.pellets,
135. self.level.power\_pellets,
136. self.score,
137. self.tutorial\_boxes,
138. )
140. **def** get\_program(self):
141. **return** self.program
143. **def** get\_error(self):
144. **return** self.error\_message
146. **def** quit(self):
147. self.level.quit()
148. pg.mixer.music.stop()

151. **class** Level:
152. **def** \_\_init\_\_(self, win\_scale, game\_id, level\_num, maze\_id, pellets, power\_pellets, score, tutorial\_boxes):
153. """
154. Responsible for running each level by calling sprite objects and handling their updates. The class
155. also controls other things, such as score, displaying maze etc.
156. :param win\_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
157. :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 number, 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 the 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 classic class.
171. :type power\_pellets: List.
172. :param score: Current score that should be displayed at the top.
173. :type score: Integer.
174. """
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
185. # Variables
186. #   Level
187. self.score = score
189. self.game\_maze = maze.Maze(maze\_id, win\_scale)
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
200. # Tutorial Variables
201. #   Messages
202. self.tutorial\_boxes = tutorial\_boxes
203. self.paused = False
205. #   Paths
206. self.show\_paths = False
208. # Length counting thread
209. threading.Thread(target=self.second\_count).start()
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)
215. # Start and ready text
216. self.ready\_text = text.Word('ready!', (17.5 \* 12, 20.5 \* 12), (255, 255, 30), 23, win\_scale, italic=True)
217. self.game\_over\_text = text.Word('game over', (18 \* 12, 20.5 \* 12), (255, 0, 0), 21, win\_scale)
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)))})
227. # Pac-Man
228. self.pac\_man = sprites.PacMan('pac-man', self.game\_maze, win\_scale)
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\_num)
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, blinky)
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)
258. # Pellets
259. self.pellets = pellets
260. self.power\_pellets = power\_pellets
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\_scale, 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. )
270. pellet\_sound\_channel = pg.mixer.Channel(2)
271. pellet\_sound\_channel.set\_volume(0.5 \* (local\_settings.get\_setting('game\_volume')/100))
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)
276. **for** row **in** self.game\_maze.tiles:
277. **for** tile **in** row:
278. **if** tile.type == 'pellet':
279. self.pellets.append(sprites.Pellet(pellet\_skin,
280. tile,
281. self.pac\_man,
282. win\_scale,
283. pellet\_death\_sound,
284. pellet\_sound\_channel)
285. )
287. **elif** tile.type == 'power\_pellet':
288. **if** level\_num >= 5:
289. self.power\_pellets.append(sprites.Pellet(power\_pellet\_skin,
290. tile,
291. self.pac\_man,
292. win\_scale,
293. pellet\_death\_sound,
294. pellet\_sound\_channel,
295. power\_pellet=True)
296. )
298. **else**:
299. self.pellets.append(sprites.Pellet(pellet\_skin,
300. tile,
301. self.pac\_man,
302. win\_scale,
303. pellet\_death\_sound,
304. pellet\_sound\_channel)
305. )
307. **else**:
308. **for** pellet **in** self.pellets:
309. pellet.predator = self.pac\_man
310. **for** power\_pellet **in** self.power\_pellets:
311. power\_pellet.predator = self.pac\_man
313. # Sound
314. self.large\_pellet\_channel = pg.mixer.Channel(3)
315. self.large\_pellet\_channel.set\_volume(0.5 \* (local\_settings.get\_setting('game\_volume')/100))
316. large\_pellet\_sound\_path = os.path.join('Resources', 'sounds', 'large\_pellet\_loop.wav')
317. self.large\_pellet\_sound = pg.mixer.Sound(large\_pellet\_sound\_path)
319. self.death\_sound\_playing = False
320. self.started = False
322. # Clocks and counts
323. self.start\_clock = 0
324. self.one\_up\_clock = 0
325. self.flashing\_map\_clock = 0
326. self.flashing\_map\_count = 0
328. **def** run(self, win, events):
329. """
330. This method is run from the Tutorial class and updates and displays by one frame.
331. :param win: The current window, all objects must be blitted to this window to be displayed.
332. :type win: Surface.
333. :param events: Contains events from the pg.event.get() call containing all keyboard events.
334. :type events: Tuple.
335. :return: Boolean returns True if level is won, False if level is lost, None otherwise
336. """
338. # Change to make sure there is a delay when there are no text boxes.
339. **if** **not** self.start:
341. # Score Indicators
342. self.score\_indicator.display(win)
344. # Maze
345. self.game\_maze.display(win)
347. self.ready\_text.display(win)
349. # Pellets
350. **for** pellet **in** self.pellets:
351. pellet.display(win)
353. #  Power pellets
354. **for** power\_pellet **in** self.power\_pellets:
355. power\_pellet.display(win)
357. # Ghosts
358. **for** ghost **in** self.ghosts:
359. ghost.display(win)
361. # Pac-Man
362. self.pac\_man.display(win)
364. **elif** self.paused:
365. self.score\_indicator.display(win)
366. self.game\_maze.display(win)
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)
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
387. self.game\_maze.display(win)
389. self.score\_indicator.display(win)
390. self.pac\_man.display(win)
392. # Mainloop of the level
393. **else**:
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. )
403. self.score\_indicator.display(win)
405. # Maze
406. self.game\_maze.display(win)
408. # Ghost Paths
409. **if** self.show\_paths:
410. **for** ghost **in** self.ghosts:
411. ghost.draw\_path(win)
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)
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.dead]
436. self.power\_pellets.remove(power\_pellet)
438. # Pac-Man
439. self.pac\_man.update(events)
440. self.pac\_man.display(win)
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
448. **if** self.pac\_man.death\_animation\_finished **and** **not** self.finished:
449. self.finished = True
451. # Ghosts
452. **if** all([**not** ghost.scared **for** ghost **in** self.ghosts]):
453. self.large\_pellet\_channel.stop()
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
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)
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()
478. ghost.update(events)
479. ghost.display(win)
481. **def** quit(self):
482. """
483. Quits the level. Stops music playing and saves that level to the databse.
484. :return: None
485. """
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. )
498. self.\_run = False
499. pg.mixer.stop()
501. **def** second\_count(self):
502. """
503. Counts seconds, so that the time can be recorded in the database.
504. :return: None
505. """
507. **while** self.\_run:
508. sleep(1)
509. self.length += 1

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 ghosts have and the
516. behaviour of clyde. This is shown through the paths that are displayed.
517. """
519. **def** \_\_init\_\_(self, win\_scale, game\_id, level\_num, maze\_id, pellets, power\_pellets, score, start\_cap=2):
520. super().\_\_init\_\_(win\_scale, game\_id, level\_num, maze\_id, pellets, power\_pellets, score, start\_cap)
522. **def** run(self, win, events):
523. super().run(win, events)
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)
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

542. **class** Level2(Level):
543. """
545. """
547. **def** \_\_init\_\_(self, win\_scale, game\_id, level\_num, maze\_id, pellets, power\_pellets, score, tutorial\_prompts):
548. super().\_\_init\_\_(win\_scale, game\_id, level\_num, maze\_id, pellets, power\_pellets, score, tutorial\_prompts)
550. **def** run(self, win, events):
552. super().run(win, events)
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)
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
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

579. **class** Level3(Level):
580. """
582. """
584. **def** \_\_init\_\_(self, win\_scale, game\_id, level\_num, maze\_id, pellets, power\_pellets, score, tutorial\_prompts):
585. super().\_\_init\_\_(win\_scale, game\_id, level\_num, maze\_id, pellets, power\_pellets, score, tutorial\_prompts)
587. **def** run(self, win, events):
588. super().run(win, events)
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)
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
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

615. **class** Level4(Level):
616. """
618. """
620. **def** \_\_init\_\_(self, win\_scale, game\_id, level\_num, maze\_id, pellets, power\_pellets, score, tutorial\_prompts):
621. super().\_\_init\_\_(win\_scale, game\_id, level\_num, maze\_id, pellets, power\_pellets, score, tutorial\_prompts)
623. **def** run(self, win, events):
624. super().run(win, events)
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)
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
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

651. **class** Level5(Level):
652. """
654. """
656. **def** \_\_init\_\_(self, win\_scale, game\_id, level\_num, maze\_id, pellets, power\_pellets, score, tutorial\_prompts):
657. super().\_\_init\_\_(win\_scale, game\_id, level\_num, maze\_id, pellets, power\_pellets, score, tutorial\_prompts)
659. **def** run(self, win, events):
660. super().run(win, events)

663. **class** Level6(Level):
664. """
666. """
668. **def** \_\_init\_\_(self, win\_scale, game\_id, level\_num, maze\_id, pellets, power\_pellets, score, tutorial\_prompts):
669. super().\_\_init\_\_(win\_scale, game\_id, level\_num, maze\_id, pellets, power\_pellets, score, tutorial\_prompts)
671. **def** run(self, win, events):
672. super().run(win, events)

### Single Player

1. \_\_author\_\_ = 'Will Evans'
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

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 related variables).
20. :param user\_id: UserID if the user has signed in (they don't have to be signed in to play Classic).
21. """
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
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'))
36. # Highscore
37. self.highscore = local\_database.get\_highscore()
38. **if** self.highscore **is** None:
39. self.highscore = 0
41. # Maze
42. self.maze\_id = 1
44. # Variables
45. self.score = 0
46. self.pellets\_eaten = 0
47. self.level\_num = 1
48. self.lives = 3
50. #   Database
51. self.game\_id = local\_database.get\_game\_id(self.user\_id, self.maze\_id)
53. # Level Info
54. self.level\_num = 1
55. self.extra\_life\_claimed = False
57. # Level
58. self.level = Level(
59. self.win\_scale,
60. self.game\_id, 1,
61. self.maze\_id,
62. [],
63. [],
64. self.lives,
65. self.score,
66. self.highscore,
67. start\_cap=4
68. )
69. # Initials Input Box
70. self.initials\_input\_box = text.TransparentInputBox(100, 30, 20, win\_scale, 'Initials')
71. self.word\_1 = text.Word('Enter 3 initials', (168, 180), (255, 255, 30), 20, win\_scale, centre=True)
72. self.word\_2 = text.Word('to save highscore', (168, 195), (255, 255, 30), 20, win\_scale, centre=True)
74. **def** run(self, win, events):
75. """
76. This method is run directly from the main script.
77. :param win: The current window, all objects must be blitted to this window to be displayed.
78. :param events: Contains events from the pg.event.get() call containing all keyboard events.
79. :return: None
80. """
82. # Essential
83. self.level.run(win, events)  # bool -: True if level won, false if pac-man dead and None if neither
85. # Variables
86. self.score = self.level.score
87. self.pellets\_eaten = self.level.pellets\_eaten
88. self.lives = self.level.lives
90. self.extra\_life\_claimed = self.level.extra\_life\_claimed
92. # Events
93. **for** event **in** events:
94. **if** event.type == pg.KEYDOWN:
95. **if** event.key == pg.K\_ESCAPE:
96. self.program = 'StartScreen'
98. # If all the pellets have been eaten
99. **if** self.level.finished **and** self.level.won:
100. self.level\_num += 1
101. self.level.quit()
102. self.level = Level(
103. self.win\_scale,
104. self.game\_id,
105. self.level\_num,
106. self.maze\_id,
107. [],
108. [],
109. self.lives,
110. self.score,
111. self.highscore,
112. extra\_life\_claimed=self.extra\_life\_claimed
113. )
115. # 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. )
133. **elif** self.lives == 0 **and** self.level.finished **and** **not** self.game\_finished:
134. self.level.quit()
135. self.game\_finished = True
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'
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)
149. **def** get\_program(self):
150. **return** self.program
152. **def** get\_error(self):
153. **return** self.error\_message
155. **def** quit(self):
156. pg.mixer.music.stop()
157. self.level.quit()

160. **class** Level:
161. **def** \_\_init\_\_(self, win\_scale, game\_id, level\_num, maze\_id, pellets, power\_pellets, lives, score, highscore,
162. start\_cap=2, extra\_life\_claimed=False):
163. """
164. Responsible for running each level by calling sprite objects and handling their updates. 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 number, 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 the 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 classic 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 player.
191. :type extra\_life\_claimed: Boolean.
192. """
194. # Essential
195. self.\_run = True
196. self.win\_scale = win\_scale
197. self.program = 'Classic'
198. self.finished = False
199. self.won = False
201. # Variables
202. #   Level
203. self.score = score
204. self.highscore = highscore
206. self.game\_maze = Maze(maze\_id, win\_scale)
208. self.lives = lives
210. self.start\_cap = start\_cap
212. self.extra\_life\_claimed = extra\_life\_claimed
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
223. # Length counting thread
224. threading.Thread(target=self.second\_count).start()
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)
232. self.highscore\_text\_position = (19 \* 12, 0.8 \* 12)
233. self.highscore\_position = (17 \* 12, 2 \* 12)
235. self.highscore\_text = text.Word('high score', self.highscore\_text\_position, (234, 234, 234), 24, win\_scale)
237. **if** self.score > self.highscore:
238. self.highscore\_indicator = text.Word('{}'.format(self.score),
239. self.highscore\_position,
240. (234, 234, 234),
241. 24,
242. win\_scale
243. )
244. **else**:
245. self.highscore\_indicator = text.Word('{}'.format(self.highscore),
246. self.highscore\_position,
247. (234, 234, 234),
248. 24,
249. win\_scale
250. )
252. self.life\_indicators = []
253. skin = pg.transform.scale(pg.image.load('Resources\\sprites\\pac-man\\w\_0.png'),
254. (22 \* win\_scale, 22 \* win\_scale))
256. **for** num **in** range(lives - 1):
257. rect = skin.get\_rect(center=((num \* 24 \* win\_scale + 18 \* win\_scale), (35 \* 12 \* win\_scale)))
258. self.life\_indicators.append(StaticSprite([skin], rect))
260. # Start and ready text
261. self.ready\_text = text.Word('ready!', (17.5 \* 12, 20.5 \* 12), (255, 255, 30), 23, win\_scale, italic=True)
262. self.game\_over\_text = text.Word('game over', (18 \* 12, 20.5 \* 12), (255, 0, 0), 21, win\_scale)
264. # Points (displayed when Pac-Man eats a ghost).
265. self.points\_text = {}
266. **for** text **in** os.listdir('Resources\\sprites\\{}'.format('points')):
267. # noinspection PyUnresolvedReferences
268. self.points\_text.update({text: pg.transform.scale(
269. pg.image.load('Resources\\sprites\\{}\\{}'.format('points', text)),
270. ((24 \* win\_scale), (10 \* win\_scale)))})
272. # Pac-Man
273. self.pac\_man = PacMan('pac-man', self.game\_maze, win\_scale)
275. # Ghosts
276. self.ghosts = []
277. self.ghosts\_copy = self.ghosts[::]
278. blinky = Blinky('blinky', self.pac\_man, self.game\_maze, win\_scale, level\_num)
279. self.ghosts.append(blinky)
280. self.ghosts.append(Pinky('pinky', self.pac\_man, self.game\_maze, win\_scale, level\_num))
281. self.ghosts.append(Clyde('clyde', self.pac\_man, self.game\_maze, win\_scale, level\_num))
282. self.ghosts.append(Inky('inky', self.pac\_man, self.game\_maze, win\_scale, level\_num, blinky))
284. # Pellets
285. self.pellets = pellets
286. self.power\_pellets = power\_pellets
288. **if** pellets == [] **and** power\_pellets == []:
289. pellet\_skin\_path = os.path.join('Resources', 'sprites', 'pellets', 'pellet.png')
290. pellet\_skin = pg.transform.scale(pg.image.load(pellet\_skin\_path), (4 \* win\_scale, 4 \* win\_scale))
292. power\_pellet\_skin\_path = os.path.join('Resources', 'sprites', 'pellets', 'power\_pellet.png')
293. power\_pellet\_skin = pg.transform.scale(pg.image.load(power\_pellet\_skin\_path),
294. (12 \* win\_scale, 12 \* win\_scale))
296. pellet\_sound\_channel = pg.mixer.Channel(2)
297. pellet\_sound\_channel.set\_volume(0.5 \* (local\_settings.get\_setting('game\_volume') / 100))
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)
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. )
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
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))
332. large\_pellet\_sound\_path = os.path.join('Resources', 'sounds', 'large\_pellet\_loop.wav')
333. self.large\_pellet\_sound = pg.mixer.Sound(large\_pellet\_sound\_path)
335. self.death\_sound\_playing = False
336. self.started = False
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
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 displayed.
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 otherwise
352. """
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)
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)
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
385. self.game\_maze.display(win)
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)
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
405. self.score\_indicator = text.Word('{}'.format(self.score),
406. self.score\_position,
407. (234, 234, 234),
408. 24,
409. self.win\_scale
410. )
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)
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))
425. **for** num **in** range(self.lives - 1):
426. rect = skin.get\_rect(center=(
427. (num \* 24 \* self.win\_scale + 18 \* self.win\_scale),
428. (35 \* 12 \* self.win\_scale)
429. )
430. )
432. self.life\_indicators.append(StaticSprite([skin], rect))
433. self.extra\_life\_claimed = True
435. self.highscore\_text.display(win)
436. self.highscore\_indicator.display(win)
438. #   Life indicators
439. **for** life\_indicator **in** self.life\_indicators:
440. life\_indicator.display(win)
442. # Maze
443. self.game\_maze.display(win)
444. self.game\_maze.display(win)
446. # Pellets
447. **for** pellet **in** self.pellets:
448. pellet.update()
449. pellet.display(win)
450. **if** pellet.eaten:
451. self.score += 10
452. self.level\_score += 10
453. self.pellets\_eaten += 1
454. self.pellets.remove(pellet)
456. #  Power pellets
457. **for** power\_pellet **in** self.power\_pellets:
458. power\_pellet.update()
459. power\_pellet.display(win)
460. **if** power\_pellet.eaten:
461. self.score += 50
462. self.level\_score += 50
463. self.power\_pellets\_eaten += 1
464. **if** **not** self.large\_pellet\_channel.get\_busy():
465. self.large\_pellet\_channel.play(self.large\_pellet\_sound, loops=-1)
466. **for** ghost **in** self.ghosts:
467. ghost.scare()
468. self.ghosts\_copy = [ghost **for** ghost **in** self.ghosts **if** **not** ghost.dead]
469. self.power\_pellets.remove(power\_pellet)
471. # Pac-Man
472. self.pac\_man.update(events)
473. self.pac\_man.display(win)
475. # Checks whether the level has ended
476. **if** self.pac\_man.dead:
477. **if** self.death\_sound\_playing:
478. **pass**
479. **else**:
480. pg.mixer.stop()
481. self.death\_sound\_playing = True
483. **if** self.pac\_man.death\_animation\_finished **and** **not** self.finished:
484. self.finished = True
485. **if** self.lives == 1:
486. self.lives -= 1
487. self.game\_over\_text.display(win)
488. pg.display.update()
489. sleep(2)
491. # Ghosts
492. **if** all([**not** ghost.scared **for** ghost **in** self.ghosts]):
493. self.large\_pellet\_channel.stop()
495. **for** ghost **in** self.ghosts\_copy:
497. **if** ghost.dead:
498. self.ghosts\_eaten += 1
499. self.ghosts\_copy.remove(ghost)
500. **for** ghost\_ **in** self.ghosts\_copy:
501. ghost\_.display(win)
502. points = 200 \* 2 \*\* ((len(self.ghosts) - 1) - len(self.ghosts\_copy))
503. self.score += points
504. self.level\_score += points
506. win.blit(self.points\_text['{}.png'.format(str(points))],
507. (self.pac\_man.x, self.pac\_man.y - 8 \* self.win\_scale))
508. pg.display.update()
509. sleep(1)
511. **if** **not** self.pac\_man.dead:
512. **for** ghost **in** self.ghosts:
513. **if** ghost.\_\_class\_\_.\_\_name\_\_ == 'Blinky':
514. **if** len(self.pellets) < 20 + 2 \* self.level\_num:
515. ghost.make\_elroy()
516. **if** len(self.pellets) < 10 + 2 \* self.level\_num:
517. ghost.elroy\_upgrade()
519. ghost.update(events)
520. ghost.display(win)
522. **def** quit(self):
523. """
524. Quits the level. Stops music playing and saves that level to the databse.
525. :return: None
526. """
528. local\_database.save\_level(
529. self.level\_num,
530. self.game\_id,
531. self.lives,
532. self.level\_score,
533. self.length,
534. self.pellets\_eaten,
535. self.power\_pellets\_eaten,
536. self.ghosts\_eaten
537. )
539. self.\_run = False
540. pg.mixer.stop()
542. **def** second\_count(self):
543. """
544. Counts seconds, so that the time can be recorded in the database.
545. :return: None
546. """
548. **while** self.\_run:
549. sleep(1)
550. self.length += 1

### Multiplayer

1. \_\_\_author\_\_ = 'Will Evans'
3. **import** os
4. **import** pygame as pg
5. **import** networking
6. **import** local\_database
8. **from** sprites **import** \*
9. **from** multiplayer\_sprites **import** \*
10. **from** datastructures **import** Maze
11. **from** gui **import** \*

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 'join 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 related variables).
22. :param user\_id: Unique to each user, required to play online.
23. """
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
34. # Data
35. self.players = None
37. # Player Boxes
38. self.boxes, self.large\_box = get\_boxes(win\_scale)
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\_skins,
46. self.boxes,
47. self.large\_box,
48. win\_scale
49. )
51. # Choices
52. self.font\_size = 32
53. self.choices = []
54. self.choices.append(LiveWord('Create Game', 350, self.font\_size, win\_scale))
56. self.choices.append(LiveWord('Join Game', 380, self.font\_size, win\_scale))
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
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. """
73. # Events
74. **for** event **in** events:
75. **if** event.type == pg.KEYDOWN:
76. **if** event.key == pg.K\_ESCAPE:
77. self.program = 'StartScreen'
79. # Updating text reactions (i.e. highlighting)
80. self.check\_inputs(events)
81. self.update\_text()
83. # Displaying all objects
84. **for** choice **in** self.choices:
85. choice.display(win)
87. **for** box **in** self.boxes:
88. box.display(win)
89. self.large\_box.display(win)
91. **for** avatar **in** self.avatars:
92. avatar.display(win)
94. pg.display.update()
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. """
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()
109. **def** update\_text(self):
110. """
111. Updates the choices (highlights when mouse passes over).
112. :return: None
113. """
115. **for** choice **in** self.choices:
116. **if** choice.check\_mouse(\*pg.mouse.get\_pos()):
117. choice.react()
119. **def** get\_program(self):
120. **return** self.program
122. **def** get\_error(self):
123. **return** self.error\_message
125. **def** quit(self):
126. **pass**

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. """
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
152. # Host Data
153. self.name = local\_database.get\_username(user\_id)
155. self.client\_id = 0
157. # Instantiate Server
158. self.server = networking.Server(self.name)
159. self.ip = self.server.get\_ip()
161. self.players = self.server.get\_players()
163. # Thread that handles connecting clients
164. self.server.searching\_for\_clients = True
166. # Player Boxes
167. self.boxes, self.large\_box = get\_boxes(win\_scale)
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\_avatars(self.players,
173. self.finished,
174. self.avatar\_skins,
175. self.boxes,
176. self.large\_box,
177. win\_scale,
178. self.client\_id
179. )
181. # Input Box
182. self.game\_id\_box = InputBox(x=120,
183. y=400,
184. w=100,
185. h=20,
186. font\_size=20,
187. win\_scale=win\_scale,
188. name=self.ip,
189. interactive=False
190. )
192. # Start Button
193. self.start\_button = Button(content='Start',
194. pos=(265, 400),
195. dimensions=(60, 20),
196. font\_size=20,
197. text\_colour=(255, 255, 30),
198. width=2,
199. win\_scale=win\_scale
200. )
202. # Cancel Button
203. self.cancel\_button = Button(content='Cancel',
204. pos=(250, 400),
205. dimensions=(75, 20),
206. font\_size=20,
207. text\_colour=(255, 0, 0),
208. width=2,
209. win\_scale=win\_scale
210. )
212. # Number
213. self.number = Word(None, (180, 400), (255, 255, 30), 48, win\_scale)
215. # Clock
216. self.start\_countdown\_clock = 0
218. **def** run(self, win, events):
219. """
220. This method is run directly from the main script. It updates and displays all of the components on the screen.
221. It also calls the server object to ensure all the data is up to date.
222. :param win: The current window, all objects must be blitted to this window to be displayed.
223. :type win: Surface.
224. :param events: Contains events from the pg.event.get() call containing all keyboard events.
225. :type events: list.
226. :return: None
227. """
229. # Events
230. **for** event **in** events:
231. **if** event.type == pg.KEYDOWN:
232. **if** event.key == pg.K\_ESCAPE:
233. self.program = 'Multiplayer'
234. self.server.quit()
236. **if** self.level **is** None:
237. pg.mixer.stop()
239. players = self.server.get\_players()
241. # Check Score
242. **for** player\_id, player\_data **in** players.items():
243. **if** player\_data['score'] > 20000:
244. self.finished = True
245. **if** self.finished:
246. places = []
247. **for** id, data **in** players.items():
248. places.append({'id': id, 'score': data['score']})
249. places = sorted(places, key=**lambda** k: k['score'], reverse=True)
250. **for** place, data **in** enumerate(places):
251. self.server.update\_data(data['id'], 'place', place)
253. # Updating avatars as per how many people are connected
254. self.avatars, self.names, self.scores, self.ready\_indicators, self.places = get\_avatars(
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. )
265. # Displaying boxes and avatars
266. **for** box **in** self.boxes:
267. box.display(win)
268. self.large\_box.display(win)
270. **for** avatar **in** self.avatars:
271. avatar.display(win)
273. **for** name **in** self.names:
274. name.display(win)
276. **for** score **in** self.scores:
277. score.display(win)
279. **for** ready **in** self.ready\_indicators:
280. ready.display(win)
282. **for** place **in** self.places:
283. place.display(win)
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()
294. **if** self.start\_countdown\_clock > 3:
295. **if** self.winner\_id **is** **not** None:
296. self.server.swap(self.winner\_id)
298. **if** self.start\_countdown\_clock > 4.8:
299. self.start\_countdown\_clock = 0
300. self.server.reset()
301. self.server.send\_data()
303. game\_maze = Maze(1, self.win\_scale)
305. score = [data['score'] **for** data **in** self.server.get\_players().values() **if** data['skin'] == 'pac-man'][0]
307. self.level = HostLevel(self.win\_scale, 5, game\_maze, score, self.server)
308. self.match\_started = True
310. self.start\_countdown = False
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
320. **if** self.server.has\_ai **and** **not** self.match\_started:
321. self.server.remove\_ai()
323. self.start\_button.update(events)
324. self.start\_countdown = self.start\_button.get\_click()
325. self.start\_button.display(win)
327. **if** **not** self.match\_started:
328. self.game\_id\_box.update(events)
329. self.game\_id\_box.display(win)
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
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()
344. **def** get\_program(self):
345. **return** self.program
347. **def** get\_error(self):
348. **return** self.error\_message
350. **def** quit(self):
351. self.server.quit()
352. pg.mixer.stop()

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 players 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 displayed.
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. """
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
385. # Client Data
387. self.name = local\_database.get\_username(user\_id)
388. self.client\_id = None
389. self.players = None
390. self.ready = False
392. # Player Boxes
393. self.boxes, self.large\_box = get\_boxes(win\_scale)
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\_avatars(
399. self.players,
400. self.finished,
401. self.avatar\_skins,
402. self.boxes,
403. self.large\_box,
404. self.win\_scale,
405. )
407. # Input Box
408. self.input\_box = InputBox(120, 380, 100, 20, 20, win\_scale, 'Game ID')
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. )
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. )
429. # Countdown number
430. self.number = Word(None, (180, 400), (255, 255, 30), 48, win\_scale)
432. **def** run(self, win, events):
433. """
434. This method is run directly from the main script. It updates and displays all of the 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 displayed.
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. """
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'
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. )
464. **for** box **in** self.boxes:
465. box.display(win)
466. self.large\_box.display(win)
468. **for** avatar **in** self.avatars:
469. avatar.display(win)
471. **for** name **in** self.names:
472. name.display(win)
474. **for** score **in** self.scores:
475. score.display(win)
477. **for** ready **in** self.ready\_indicators:
478. ready.display(win)
480. **for** place **in** self.places:
481. place.display(win)
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
489. self.client = networking.Client(self.game\_id, self.name)
490. self.connected = self.client.connected
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)
499. **else**:
500. **if** **not** self.client.connected:
501. self.client.end()
502. self.program = 'Multiplayer'
503. self.error\_message = 'Connection Lost'
505. **if** **not** self.finished:
506. **if** **not** self.start\_level:
507. self.level = None
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)
519. self.client.update\_data('ready', self.ready)
520. self.client.send\_player\_data()
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)
530. self.level.run(win, events)
531. **else**:
532. **if** **not** self.start\_level:
533. self.level = None
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()
540. **def** get\_program(self):
541. **return** self.program
543. **def** get\_error(self):
544. **return** self.error\_message
546. **def** quit(self):
547. **try**:
548. self.client.end()
549. **except** AttributeError:
550. **pass**
551. **except** Exception as e:
552. **print**(e)
554. pg.mixer.stop()

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 updates. 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 number, 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. """
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
589. # Music
590. self.music\_channel = pg.mixer.Channel(5)
591. self.music\_channel.set\_volume(0.5 \* (local\_settings.get\_setting('game\_volume') / 100))
592. intro\_music\_path = os.path.join('Resources', 'sounds', 'intro\_music.wav')
593. self.music\_channel.play(pg.mixer.Sound(intro\_music\_path))
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)
599. self.ready\_text = Word('ready!', (17.5 \* 12, 20.5 \* 12), (255, 255, 30), 23, win\_scale, italic=True)
600. self.game\_over\_text = Word('game over', (18 \* 12, 20.5 \* 12), (255, 0, 0), 21, win\_scale)
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)))})
609. # Sprites
610. self.pac\_man, self.ghosts = self.get\_players(self.players, self.game\_maze, win\_scale, self.client)
611. self.ghosts\_copy = self.ghosts[::]
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\_pellet.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))
623. power\_pellet\_death\_sound\_path = os.path.join('Resources', 'sounds', 'pellet', 'death.wav')
624. pellet\_death\_sound = pg.mixer.Sound(power\_pellet\_death\_sound\_path)
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. )
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. )
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\_loop.wav')
652. self.large\_pellet\_sound = pg.mixer.Sound(self.large\_pellet\_sound\_path)
654. # Misc
655. self.death\_sound\_playing = False
656. self.started = False
658. # Clocks and counts
659. self.flashing\_map\_clock = 0
660. self.flashing\_map\_count = 0
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 data form level object to
665. level object which can't be done when running directly from main. It controls the updates and displaying of all
666. objects.
667. :param win: The current window, all objects must be blitted to this window to be displayed.
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. """
674. # Intro Music
675. **if** self.music\_channel.get\_busy():
677. # Display (only)
678. self.game\_maze.display(win)
680. **for** pellet **in** self.pellets:
681. pellet.display(win)
682. **for** power\_pellet **in** self.power\_pellets:
683. power\_pellet.display(win)
685. self.pac\_man.display(win)
687. **for** ghost **in** self.ghosts:
688. ghost.display(win)
690. self.score\_indicator.display(win)
692. self.ready\_text.display(win)
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
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()
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
709. # Display (only and selective)
710. self.game\_maze.display(win)
711. self.score\_indicator.display(win)
712. self.pac\_man.display(win)
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)
725. **for** power\_pellet **in** self.power\_pellets:
726. power\_pellet.update()
727. power\_pellet.display(win)
729. # When power pellet eaten
730. **if** power\_pellet.eaten:
731. self.score += 50
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)
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.dead]
741. self.power\_pellets.remove(power\_pellet)
743. # If all ghosts are not scared (i.e all pac\_man\_dead or scared timer ended) stop playing sound
744. **if** all([**not** ghost.scared **for** ghost **in** self.ghosts]):
745. self.large\_pellet\_channel.stop()
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
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)
761. # Update and display Pac-Man
762. self.pac\_man.display(win)
763. self.pac\_man.update(events)
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
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
781. # If Pac-Man is alive
782. **if** **not** self.pac\_man.dead:
784. # Update and display ghosts
785. **for** ghost **in** self.ghosts:
786. ghost.update(events)
787. ghost.display(win)
789. self.score\_indicator = Word('{}'.format(self.client.get\_data(self.client.get\_client\_id(), 'score')),
790. self.score\_position, (234, 234, 234), 24, self.win\_scale)
791. self.score\_indicator.display(win)
793. **return** True, None
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 to send and receive player
806. data.
807. :type client: Server.
808. :return: Pac-Man object and list of ghost objects.
809. """
811. # Sprites
812. ghosts = []
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)
826. # Blinky
827. **for** client\_id, player **in** players.items():
828. skin = player['skin']
830. **if** skin == 'blinky':
831. **continue**
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. )
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)
866. # Ghosts
867. **for** client\_id, player **in** players.items():
868. skin = player['skin']
869. **if** skin != 'pac-man':
870. # Ghosts
871. **if** client\_id == self.client\_id:
872. ghosts.insert(0,
873. ClientPlayerGhost(skin,
874. player['pos'],
875. pac\_man,
876. game\_maze,
877. win\_scale,
878. self.level\_num,
879. client,
880. client\_id)
881. )
883. **else**:
884. ghosts.append(
885. ClientGhost(skin,
886. player['pos'],
887. pac\_man,
888. game\_maze,
889. win\_scale,
890. self.level\_num,
891. client,
892. client\_id
893. )
894. )
896. **return** pac\_man, ghosts

899. **class** HostLevel(ClientLevel):
900. **def** \_\_init\_\_(self, win\_scale, level\_num, game\_maze, score, server):
901. """
902. Responsible for running each level by calling sprite objects and handling their updates. The class also
903. controls other things, such as score, displaying maze etc.
904. :param win\_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
905. :type win\_scale: Integer.
906. :param level\_num: This controls difficulty of the ghosts. The larger the level number, the more levels Pac-Man
907. has completed and the harder the ghosts become.
908. :type level\_num: Integer.
909. :param game\_maze: 2D list of maze.
910. :type game\_maze: 2D list.
911. :param score: Current score that should be displayed at the top.
912. :type score: Integer.
913. :param server: Object used to send and receive data.
914. :type server: Server.
915. """
917. super().\_\_init\_\_(win\_scale, level\_num, game\_maze, score, server)
919. # Clock
920. self.score\_update\_clock = 0
922. **def** get\_players(self, players, game\_maze, win\_scale, server):
923. """
924. This takes the list of players and assigns each of them the appropriate multiplayer sprite based on whether they
925. are Pac-Man or a ghost.
926. :param players: Dictionary of players and their various attributes.
927. :type players: 2D dictionary.
928. :param game\_maze: 2D list of maze.
929. :type game\_maze: 2D list.
930. :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 to send and receive player
933. data.
934. :type server: Server.
935. :return: Pac-Man object and list of ghost objects.
936. """
938. ghost\_sprites = {
940. 'pinky': ServerPinky,
941. 'clyde': ServerClyde
943. }
944. # Sprites
945. ghosts = []
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)
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\_num, 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)
974. # Ghosts
975. **for** client\_id, player **in** players.items():
976. skin = player['skin']
978. **if** skin == 'blinky':
979. **continue**
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**:
994. ghosts.append(
995. ServerGhost(skin,
996. player['pos'],
997. pac\_man,
998. game\_maze,
999. win\_scale,
1000. self.level\_num,
1001. server,
1002. client\_id)
1003. )
1005. **elif** player['name'] == 'AI' **and** skin != 'pac-man':
1006. **if** skin == 'inky':
1007. ghosts.append(ServerInky(player['skin'],
1008. pac\_man,
1009. game\_maze,
1010. win\_scale,
1011. self.level\_num,
1012. blinky,
1013. server,
1014. client\_id)
1015. )
1016. **else**:
1017. ghosts.append(ghost\_sprites[skin](player['skin'],
1018. pac\_man,
1019. game\_maze,
1020. win\_scale,
1021. self.level\_num,
1022. server,
1023. client\_id)
1024. )
1026. **return** pac\_man, ghosts
1028. **def** run(self, win, events):
1029. """
1030. Does the same as client level except it also keeps track of each players score.
1031. :param win: The current window, all objects must be blitted to this window to be displayed.
1032. :type win: Surface.
1033. :param events: Contains events from the pg.event.get() call containing all keyboard events.
1034. :type events: Tuple.
1035. :return: None
1036. """
1038. # Scores
1039. self.pac\_man.update\_score(self.score)
1040. **if** self.score\_update\_clock > 1:
1041. self.score\_update\_clock = 0
1042. **for** ghost **in** self.ghosts:
1043. points = get\_distance\_points(ghost, self.pac\_man)
1044. ghost.add\_points(points)
1045. **else**:
1046. self.score\_update\_clock += 1 / 60
1047. super().run(win, events)

1050. **def** get\_avatar\_skins():
1051. """
1052. Gets skins for each avatar from resources folder. Chooses between grey and coloured depending on whether the player is
1053. connected.
1054. :return: Dictionary of skins.
1055. """
1057. 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))})
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))})
1069. **return** avatar\_skins

1072. **def** get\_boxes(win\_scale):
1073. """
1074. Returns box objects for the menu screens. These are just the 4 empty boxes across the top 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. """
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))
1091. large\_box\_rect = ((95, 170), (150, 150))
1092. large\_box = Box(\*large\_box\_rect, colour, line\_width, win\_scale)
1094. **return** boxes, large\_box

1097. **def** get\_avatars(players, finished, avatar\_skins, boxes, large\_box, win\_scale, client\_id=None):
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 (when 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 = []
1117. colours = {
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),
1125. }
1127. place\_details = {
1128. 0:
1129. {
1130. 'string': '1st',
1131. 'colour': (255, 215, 0)
1132. },
1134. 1:
1135. {
1136. 'string': '2nd',
1137. 'colour': (220, 220, 220)
1138. },
1140. 2:
1141. {
1142. 'string': '3rd',
1143. 'colour': (205, 127, 50)
1144. },
1146. 3:
1147. {
1148. 'string': '4th',
1149. 'colour': (169, 169, 169)
1150. },
1152. 4:
1153. {
1154. 'string': '5th',
1155. 'colour': (169, 169, 169)
1156. }
1158. }
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):
1165. x, y = large\_box.rect.center
1166. x /= win\_scale
1167. y /= win\_scale
1168. colour = colours[players[client\_id]['skin']]
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))
1176. # Large Name
1177. name = players[client\_id]['name']
1178. names.append(Word(name, (x, y + 95), colour, 26, win\_scale, centre=True))
1180. # Score
1181. **if** players[num]['score'] **is** **not** None:
1182. score = str(players[client\_id]['score'])
1184. scores.append(Word(content=score, pos=(x, y + 115), colour=colour,
1185. font\_size=26, win\_scale=win\_scale, italic=True, centre=True))
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))
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']]['colour'],
1198. font\_size=26,
1199. win\_scale=win\_scale,
1200. italic=True,
1201. centre=True)
1202. )
1204. large\_box\_boolean = True
1206. **if** large\_box\_boolean:
1207. num += 1
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))
1214. **else**:
1215. skin = pg.transform.scale(avatar\_skins[players[num]['skin']],
1216. (66 \* win\_scale, 66 \* win\_scale))
1218. x, y = box.rect.center
1219. x /= win\_scale
1220. y /= win\_scale
1222. colour = colours[players[num]['skin']]
1224. # Name
1225. name = players[num]['name']
1226. names.append(Word(name, (x, y + 50), colour, 16, win\_scale, centre=True))
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))
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))
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)
1248. )
1250. rect = skin.get\_rect(center=box.rect.center)
1251. avatars.append(StaticSprite([skin], rect))
1253. **else**:
1254. skin = pg.transform.scale(avatar\_skins['grey\_{}'.format('blinky')],
1255. (66 \* win\_scale, 66 \* win\_scale))
1256. **for** box **in** boxes:
1257. rect = skin.get\_rect(center=box.rect.center)
1258. avatars.append(StaticSprite([skin], rect))
1260. skin = pg.transform.scale(avatar\_skins['grey\_pac-man'],
1261. (132 \* win\_scale, 132 \* win\_scale))
1262. rect = skin.get\_rect(center=large\_box.rect.center)
1263. avatars.append(StaticSprite([skin], rect))
1265. **return** avatars, names, scores, ready\_indicators, places

1268. **def** get\_distance\_points(ghost, pac\_man):
1269. """
1270. Works out the euclidean distance between the ghost and Pac-Man objects and returns an amount of points to give the
1271. ghost based on how far it is.
1272. :param ghost: Ghost object.
1273. :param pac\_man: Pac-Man object.
1274. :return: Points.
1275. """
1277. # Gets manhattan distance from pac\_man to ghost
1278. distance = abs(ghost.tile.x - pac\_man.tile.x) + abs(ghost.tile.y - pac\_man.tile.y)
1280. # Subtracts distance from 13 so the closer to pac\_man the more points you get
1281. points = (13 - distance) \* 4
1283. # Ensures points are not negative
1284. **return** max(0, points)

1287. **def** get\_mouse\_input(events):
1288. **for** event **in** events:
1289. **if** event.type == pg.MOUSEBUTTONDOWN:
1290. **return** event.pos

### Sprites

1. \_\_author\_\_ = 'Will Evans'
3. **import** os
4. **from** pathfinding **import** Manhattan as Search
5. **import** pygame as pg
6. **import** random
7. **import** local\_settings
8. **from** threading **import** Thread
9. **from** time **import** sleep

12. **class** Sprite:
13. **def** \_\_init\_\_(self, resource\_pack, position, maze, win\_scale):
14. """
15. Template for sub-classes: 'Pac-Man' and 'Ghost'.
16. :param resource\_pack: Contains the path to the folder containing the skins for the sprite.
17. :param position: x, y co-ords for the position of the sprite on the screen.
18. :param maze: Two dimensional list representation of the maze.
19. :param win\_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
20. """
22. # Essential
23. self.win\_scale = win\_scale
24. self.maze = maze
26. #
27. self.dead = False
28. self.visible = False
30. # Position
31. x, y = position
32. self.x = x \* win\_scale
33. self.y = y \* win\_scale
35. # Speed
36. self.\_speed = 1.5
37. self.speed\_count = 0
39. self.skin = '0'
40. self.facing = 'e'
41. self.move = 'e'
42. self.online\_move = 'e'
43. self.buffer\_move = 'e'
45. # Skins
46. self.skin = '0'
47. self.skin\_cap = 4
49. #   Normal Skins
50. self.normal\_skins = {}
51. **for** skin **in** os.listdir('resources\\sprites\\{}'.format(resource\_pack)):
52. # noinspection PyUnresolvedReferences
53. self.normal\_skins.update({skin: pg.transform.scale(
54. pg.image.load('resources\\sprites\\{}\\{}'.format(resource\_pack, skin)),
55. ((22 \* win\_scale), (22 \* win\_scale)))})
57. self.skins = self.normal\_skins
59. # Tiles
60. tilex = int(self.x / (12 \* self.win\_scale))
61. tiley = int(self.y / (12 \* self.win\_scale))
63. self.tile = self.maze.tiles[tiley - 3][tilex]
64. self.previous\_tile = self.tile
66. # Rects
67. self.skin\_rect = self.skins['e\_0.png'].get\_rect(center=(self.x, self.y))
68. self.rect = pg.Rect(self.x - int(8 \* win\_scale),
69. self.y - int(8 \* win\_scale),
70. 17 \* win\_scale,
71. 17 \* win\_scale)
73. # Movement booleans
74. self.get\_new\_move = True
75. self.wall\_defence\_delay = True
76. self.return\_None = False
78. # Sound
79. self.sound\_channel = pg.mixer.Channel(0)
80. self.sound\_channel.set\_volume(0.5 \* local\_settings.get\_setting('game\_volume') / 100)
82. # Clocks
83. self.skin\_clock = 0
84. self.stop\_clock = 0
85. self.death\_animation\_clock = 0
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. """
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)
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. """
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'
119. **return** move
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. """
128. self.\_speed = speed
130. **def** get\_pos(self):
131. """
132. Returns position of the sprite.
133. :return: Position of sprite (x,y).
134. """
136. **return** self.x, self.y
138. **def** kill(self):
139. """
140. Kills the sprite by setting the dead attribute to True.
141. :return: None.
142. """
144. self.dead = True
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. """
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 skin 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)
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. """
169. move = self.get\_input(events)
170. **return** self.check\_move(move)
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. """
179. self.move = move  # Saves the move
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
186. **if** self.return\_None:
187. **return** None
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)
194. # If return\_none is false then self.facing is still a valid move
195. **if** move == self.facing:
196. **return** self.facing
198. # If move is None it is because the sprite has collided with a wall
199. **elif** move **is** None:
200. **return** None
202. **else**:
203. self.get\_new\_move = False
204. **return** move
206. # Move in the current direction until the tile has changed
207. **else**:
208. **return** self.validate\_move(self.facing)
210. **def** validate\_move(self, move):
211. """
212. Checks specifically whether the move will cause the player to collide with a wall or 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. """
217. tile\_facing = self.get\_next\_tile(self.facing)
218. tile\_move = self.get\_next\_tile(move)
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
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) co-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. """
240. pac\_x, pac\_y = self.tile.pos
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)
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 following 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
259. self.x -= difference\_x
260. self.y -= difference\_y
262. **else**:
263. self.wall\_defence\_delay = True
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. """
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
275. **if** self.x > 29 \* 12 \* self.win\_scale **and** self.facing == 'e':
276. self.x = 0 \* 12 \* self.win\_scale
278. **def** update\_pos(self, move):
279. """
280. Updates sprite's current position, according to the move, current speed and win\_scale.
281. :param move: Move that has now been checked can be used to move the sprite.
282. :return: None
283. """
285. # Dictionary keeping track of what direction the moves will move the sprite and with 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. }
294. x, y = moves[move]
296. # This records what move has been used to move in the direction. Usually sent to the server in a
297. # multiplayer game to correctly show the direction of a sprite on clients
298. self.online\_move = move
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
305. # sprite position updated as per the above move and multiplied by win\_scale to allow different sized windows
306. self.x += x \* self.win\_scale
307. self.y += y \* self.win\_scale
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(center=(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)
317. **def** update\_tile(self):
318. """
319. Updates what tile the sprite is on. This is used by many methods to determine whether the sprite is going to
320. collide with walls in the future.
321. :return: None
322. """
324. # Gets tile x,y coords as opposed to pixel x,y coords based on the sprites pixel position
325. rect\_tile\_x, rect\_tile\_y = self.rect.centerx / (12 \* self.win\_scale), self.rect.centery / (12 \* self.win\_scale)
327. # Gets tile x,y coords for the sprites current tile
328. tile\_x, tile\_y = self.tile.pos
330. # If we do the following when the sprite is off the screen (in the tunnel) we get many errors
331. **if** rect\_tile\_x > 0:
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
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 displayed.
349. :return: None
350. """
352. win.blit(self.skins['{}\_{}.png'.format(self.facing, self.skin)], self.skin\_rect)
354. **def** get\_next\_tile(self, move):
355. """
356. This is used by the move validating methods by returning the next tile the sprite will 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. """
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)}
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 top of the window to allow
377. # space for indicators such as score and highscore
378. tile\_y -= 3
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]
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 displayed
391. :return: None
392. """
393. pg.draw.rect(win, (255, 0, 0), self.rect)

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. """
405. # Essential
406. position = (167, 318)
407. super().\_\_init\_\_(resource\_pack, position, maze, win\_scale)
409. # Skins
410. self.skin = '0'
411. self.facing = 'e'
412. self.move = 'e'
413. self.num = 0
415. #   Death animation
416. self.death\_animation\_index = 0
417. self.death\_animation\_skins = {}
418. self.death\_animation\_finished = False
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)))})
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('sounds', resource\_pack, sound))})
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. """
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()
449. **def** death\_animation(self):
450. """
451. Cycles through a series of death animation skins and plays the death animation sound
452. :return: True if death animation has finished
453. """
455. # Plays the sound once the first skin has been displayed (after 1/8 of a second) unless 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
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 displayed
470. :return: None
471. """
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)

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 modes: chase (length increases
489. over levels) to scatter (length decreases over levels) and how long between becoming scared to returning to
490. normal (lowers over levels).
491. """
493. super().\_\_init\_\_(resource\_pack, position, maze, win\_scale)
495. # Speed
496. self.\_speed = 4 / 3
497. self.speed\_buffer = self.\_speed
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
505. # This is used to keep track of which ghost caught Pac-Man
506. self.won = False
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]
516. # Skins
517. self.facing = 'e'
518. self.skin = '0'
520. self.skin\_cap = 10
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. )
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. )
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\_flashing', skin)
540. self.scared\_flashing\_skins.update(
541. {skin: pg.transform.scale(pg.image.load(scared\_flashing\_skin\_path), skin\_size)}
542. )
544. self.colour = (255, 255, 255)
546. # Rect
547. self.skin\_rect = self.skins['e\_0.png'].get\_rect(center=(self.x, self.y))
549. self.mode\_index = 0
550. self.mode\_count = 0
552. # Modes
553. self.mode = self.scatter
554. self.buffer\_mode = self.scatter
556. self.to\_switch = False
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
564. self.respawned = False
566. # Path finding
567. self.search = Search(maze.tile\_map)
569. self.target = target
571. self.home = (26, 4)
573. self.next\_coords = self.scatter()[1]
575. self.path = self.get\_path(self.mode)
577. self.next\_x = 0
578. self.next\_y = 0
580. self.axis\_change = 'x'
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')
587. **def** kill(self):
588. """
589. When a ghost is killed this is run.
590. :return: None
591. """
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)
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)
612. super().update(move)
613. self.check\_collision()
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. """
621. # Keeps track of how long a mode has been active for
622. self.mode\_count += 1/60
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
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 'scatter'). Switch is also set
632. # to True. This is a boolean and not a return as it is only used if the ghost is 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
645. # Slows down the ghost when they are passing over a ghost barrier (like the original 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
652. # When a ghost reaches either of these coords (outside the ghost area) their mode is 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
657. # If a ghost is dead the active mode is 'respawn' which will direct them to the ghost 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
670. **elif** self.respawned:
671. **return** self.respawn
673. **elif** self.scared:
674. **return** self.random
676. **return** self.buffer\_mode
678. **def** check\_collision(self):
679. """
680. Checks whether the ghost is colliding with the target (Pac-Man)
681. :return: None
682. """
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
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. """
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
705. path = self.path
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
721. self.path = path
722. self.next\_coords = self.path[1]
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
728. **if** x == tile\_x **and** y == tile\_y - 3:
729. path = self.get\_path(self.mode)
731. **if** path **is** None:
732. path = self.chase()
734. **if** path **is** None:
735. **return** self.facing
737. self.path = path
738. self.next\_coords = self.path[1]
740. # Works out which direction the next coordinates are
741. x, y = self.next\_coords
742. y += 3
744. x \*= 12 \* self.win\_scale
745. x += 6 \* self.win\_scale
747. y \*= 12 \* self.win\_scale
748. y += 6 \* self.win\_scale
750. self.next\_x = x
751. self.next\_y = y
753. # If we just had self.x < x here then when we have a larger a screen the pos will jump 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'
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'
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. """
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
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
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)
809. # Works out which direction the next coordinates are
810. x, y = self.next\_coords
811. y += 3
813. x \*= 12 \* self.win\_scale
814. x += 6 \* self.win\_scale
816. y \*= 12 \* self.win\_scale
817. y += 6 \* self.win\_scale
819. self.next\_x = x
820. self.next\_y = y
822. # If we just had self.x < x here then when we have a larger a screen the pos will jump 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'
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'
841. **def** validate\_move(self, move):
842. """
843. Checks to see if a move is valid by checking whether it is opposite the current direction (facing). And by
844. using the Sprite's validate\_move above. This is not needed in classic mode as the path finding algorithm does
845. not produce paths that require a 180 degree change in direction, however this is needed for online.
846. :param move: Sprite's unchecked next move.
847. :return: Validated move.
848. """
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)
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 unreachable (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. """
864. path = mode()
865. **if** path **is** None:
866. path = self.chase()
867. **return** path
869. **def** scare(self):
870. """
871. Sets the ghost into scared mode when called.
872. :return: None.
873. """
875. **if** self.dead:
876. # If the ghost is dead then they cannot become scared
877. **pass**
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 is scared
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
893. **def** scared\_timer(self):
894. """
895. Keeps track of how long the ghost is scared and adjust attributes accordingly.
896. :return: None.
897. """
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 how 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
907. **if** int(self.scared\_clock) == 8:
908. self.scared = False
909. self.\_speed = 4/3
910. self.skins = self.normal\_skins
911. **break**
913. sleep(0.25)
914. self.scared\_clock += 0.25
916. **def** draw\_target(self, win):
917. """
918. At the moment used for debugging ghost paths and ensuring they are working correctly. Displays ghost's path
919. :param win: The current window, all objects must be blitted to this window to be displayed.
920. :return: None.
921. """
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)
929. **def** draw\_next\_tile\_target(self, win):
930. """
931. Used for debugging ghost paths and ensuring they are working correctly. Displays ghost's target tile.
932. :param win: The current window, all objects must be blitted to this window to be displayed.
933. :return: None.
934. """
936. pg.draw.rect(win, (0, 255, 0), pg.Rect(self.next\_x, self.next\_y, 4, 4))
938. **def** draw\_path(self, win):
939. """
940. Used for the storymode to teach the use how the AI works. Simply draws the path that 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 displayed.
943. :return:
944. """
946. **try**:
947. tile\_path = [self.maze.tiles[tile\_y][tile\_x] **for** tile\_x, tile\_y **in** self.path]
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')
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. """
961. **if** len(path) == 1:
962. **return** pathtiles
964. previous\_tile = path[0]
965. previous\_x, previous\_y = previous\_tile.rect.center
967. current\_tile = path[1]
968. current\_x, current\_y = current\_tile.rect.center
970. rect\_long = 14 \* self.win\_scale
971. rect\_short = 2 \* self.win\_scale
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)
986. pathtiles.append(rect)
988. **return** self.get\_pathtiles(path[1:], pathtiles)
990. **def** switch(self):
991. """
992. Changes the direction of the ghost.
993. :return: Returns the opposite of the current move
994. """
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
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. """
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)
1017. **def** scatter(self):
1018. """
1019. Pathfinding mode: It targets the specific ghost's home tile.
1020. :return: The next path.
1021. """
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)
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. """
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)
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. """
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)
1054. **return** self.search.astar(start\_tile, target\_tile, self.facing)

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 modes: chase (length increases
1066. over levels) to scatter (length decreases over levels) and how long between becoming scared to returning to
1067. normal (lowers over levels).
1068. """
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)
1076. **def** make\_elroy(self):
1077. """
1078. Turns Blinky into elroy (faster).
1079. :return: None
1080. """
1082. self.elroy = True
1083. self.speed\_buffer = 13/9
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
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. """
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
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)

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 modes: chase (length increases
1120. over levels) to scatter (length decreases over levels) and how long between becoming scared to returning to
1121. normal (lowers over levels).
1122. """
1124. position = (168, 214)
1125. super().\_\_init\_\_(resource\_pack, position, target, maze, win\_scale, level)
1127. self.home = (1, 4)
1128. self.facing = 's'
1129. self.colour = (255, 181, 255)
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. """
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)
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]
1148. **if** x == 0:
1149. change = 'y'
1150. tile\_x\_temp = tile\_x
1151. **else**:
1152. change = 'x'
1153. tile\_y\_temp = tile\_y
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)
1167. **try**:
1168. **if** self.maze.tiles[abs(tile\_y\_temp)][tile\_x\_temp].type **in** ['pellet', 'empty\_tile']:
1169. **break**
1171. **except** IndexError:
1172. **continue**
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

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 modes: chase (length increases
1189. over levels) to scatter (length decreases over levels) and how long between becoming scared to returning to
1190. normal (lowers over levels).
1191. """
1193. position = (192, 214)
1194. super().\_\_init\_\_(resource\_pack, position, target, maze, win\_scale, level)
1195. self.home = (1, 32)
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
1204. self.colour = (255, 181, 33)
1206. **def** update(self, events):
1207. """
1208. Run once a frame, this is the method that controls the start (when Clyde is still inside the centre).
1209. :param events: Contains events from the pg.event.get() call containing all keyboard events.
1210. :return: None
1211. """
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
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
1239. **return** ((tile\_x - target\_tile\_x)\*\*2 + (tile\_y - target\_tile\_y)\*\*2)\*\*0.5
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
1254. target\_tile = (target\_tile[0], target\_tile[1] - 3)
1256. path = self.search.astar(start\_tile, target\_tile, self.facing)
1257. **if** path **is** None:
1258. path = super().chase()
1259. **return** path

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 modes: chase (length increases
1271. over levels) to scatter (length decreases over levels) and how long between becoming scared to returning to
1272. normal (lowers over levels).
1273. """
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
1285. self.colour = (0, 222, 222)
1287. **def** update(self, events):
1288. """
1289. Run once a frame, this is the method that controls the start (when Inky is still inside the centre).
1290. :param events: Contains events from the pg.event.get() call containing all keyboard events.
1291. :return: None
1292. """
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
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. """
1318. path = None
1319. start\_tile = self.tile.pos
1320. start\_tile = (start\_tile[0], start\_tile[1] - 3)
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)
1326. x, y = vector
1328. target\_x, target\_y = (pac\_x + x, pac\_y + y - 3)
1330. found = False
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
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
1343. **if** target\_y > 29:
1344. target\_y = 29
1345. **elif** target\_y < 1:
1346. target\_y = y
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**
1360. path = self.search.astar(start\_tile, (tile\_x\_temp, abs(tile\_y\_temp)), self.facing)
1362. **if** path **is** None:
1363. path = super().chase()
1364. **return** path

1367. **class** Pellet:
1368. **def** \_\_init\_\_(self, skin, tile, predator, win\_scale, death\_sound, sound\_channel, power\_pellet=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. """
1380. self.power\_pellet = power\_pellet
1381. self.eaten = False
1383. self.skin = skin
1384. self.tile = tile
1385. self.predator = predator
1387. self.x = self.tile.pos[0]
1388. self.y = self.tile.pos[1]
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. )
1396. self.sound\_channel = sound\_channel
1397. self.death\_sound = death\_sound
1399. self.display\_clock = 0
1401. **def** update(self):
1402. """
1403. Runs every frame, just checks whether the pellets is colliding.
1404. :return: None
1405. """
1407. self.check\_collision()
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 displayed
1413. :return: None
1414. """
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)
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. """
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
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 displayed.
1440. :return: None
1441. """
1443. win.blit(self.skin, self.rect)

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. """
1454. self.skin\_index = 0
1455. self.skin\_cap = 8
1456. self.skin\_count = 0
1458. self.skins = skins
1459. self.rect = rect
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
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 displayed.
1471. :return: None
1472. """
1474. win.blit(self.skins[self.skin\_index], self.rect)

1477. **if** \_\_name\_\_ == '\_\_main\_\_':
1478. **pass**

### Pathfinding

1. \_\_author\_\_ = 'Will Evans'
3. **from** datastructures **import** \*

6. **class** Node:
7. **def** \_\_init\_\_(self, x, y, facing, parent=None):
8. """
9. Every tile in a path (or possible path) is called a node. It has tilex and tiley values and also stores the
10. the node object of the tile that came before it in the path.
11. :param x: Tile x
12. :param y: Tile y
13. :param facing: The direction the sprite is currently facing (stops ghosts from being able to turn around).
14. :param parent: The node object of the tile before in the path.
15. """
17. self.x = x
18. self.y = y
20. self.parent = parent
22. self.f\_score = 0
23. self.h\_score = 0
24. self.g\_score = 0
26. self.facing = facing
28. **def** get\_path(self, path):
29. """
30. Recursive algorithm to get the path once the target node has been reached.
31. :param path: Path so far
32. :return: Path with self added.
33. """
35. **if** self.parent **is** **not** None:
36. path = self.parent.get\_path(path)
37. path.append((self.x, self.y))
38. **return** path

41. **class** Search:
42. **def** \_\_init\_\_(self, maze):
43. """
44. Search is an object so that we can save the maze.
45. :param maze: 2D list of the maze.
46. """
48. self.maze = maze
50. **def** astar(self, start, end, facing):
51. """
52. Mainloop of the search algorithm.
53. :param start: Start tile (tilex, tiley).
54. :param end: End tile (tilex, tiley).
55. :param facing: Direction sprite is currently facing.
56. :return: Path in (x, y) format.
57. """
59. open\_queue = PriorityQueue()
60. closed\_set = []
61. start\_node = Node(\*start, facing)
62. start\_node.h\_score = self.heuristic(start\_node, end)
63. start\_node.g\_score = start\_node.h\_score + start\_node.f\_score
64. open\_queue.en\_queue(start\_node)
65. flag = False
67. **while** **not** open\_queue.is\_empty():
69. # Getting the next node (closest to the goal) to evaluate
70. current\_node = open\_queue.pop()
71. closed\_set.append(current\_node)
73. # Checking whether the goal has been reached
74. **if** (current\_node.x, current\_node.y) == end **and** flag:
75. **return** current\_node.get\_path([])
77. flag = True
79. # Getting adjacent nodes
80. children = get\_children(current\_node, self.maze)
82. # Adding newly evaluated nodes to the open\_queue if not already evaluated
83. **for** child **in** children:
84. self.evaluate(child, end)
85. **if** **not** open\_queue.has(child):
86. **if** **not** in\_closed(child, closed\_set):
87. open\_queue.en\_queue(child)
88. **return** None
90. **def** evaluate(self, child, end):
91. """
92. Assigns each child an h score and a g score, then combines these for the f score. These determine the fitness of
93. the child, by taking into account how many nodes there have been before the child and how close the child is to
94. the end node. This score is then used to choose the next child to expand.
95. :param child: Node object.
96. :param end: End tile (tilex, tiley)
97. :return: None
98. """
100. child.h\_score = self.heuristic(child, end)
101. child.g\_score = child.parent.g\_score + 1
102. child.f\_score = child.g\_score + child.h\_score
104. **def** heuristic(self, node, end):
105. **return** 0

108. **class** Dijkstra(Search):
109. # Checks all paths
110. **def** \_\_init\_\_(self, maze):
111. super().\_\_init\_\_(maze)
113. # noinspection PyMethodMayBeStatic
114. **def** heuristic(self, node, end):
115. """
116. Gives a score based on how close the tile is to the end child. In this case it returns 0, as the default
117. heuristic is Dijkstra's which checks every path.
118. :param node: Node to be evaluated.
119. :param end: End tile (tilex, tiley)
120. :return: Score.
121. """
123. **return** 0

126. **class** Manhattan(Search):
127. # Uses Manhattan distance as heuristic (most efficient)
128. **def** \_\_init\_\_(self, maze):
129. super().\_\_init\_\_(maze)
131. # noinspection PyMethodMayBeStatic
132. **def** heuristic(self, node, end):
133. x, y = end
134. **return** abs(node.x - x) + abs(node.y - y)

137. **class** Euclidean(Search):
138. # Not as efficient as Manhattan as cost of diagonal is the same as east and the north move in Pac-Man
139. **def** \_\_init\_\_(self, maze):
140. super().\_\_init\_\_(maze)
142. # noinspection PyMethodMayBeStatic
143. **def** heuristic(self, node, end):
144. x, y = end
145. **return** ((node.x - x)\*\*2 + (node.y - y)\*\*2)\*\*0.5

148. **def** get\_children(node, maze):
149. """
150. Returns next available tiles from current tile. This can then be added to the list of children.
151. :param node: Current tile.
152. :param maze: 2D list of maze.
153. :return: List of children of current tile.
154. """
156. vectors = {'s': (0, 1), 'e': (1, 0), 'w': (-1, 0), 'n': (0, -1)}
157. children = []
158. possible\_moves = ['n', 'e', 's', 'w', 'n', 'e']
159. **del** vectors[possible\_moves[possible\_moves.index(node.facing) + 2]]
160. **for** key, value **in** vectors.items():
161. x, y = value
162. **try**:
163. # Checking surrounding tiles for walls
164. **if** maze[node.y + y][node.x + x] != 1:
165. children.append(Node(node.x + x, node.y + y, key, node))
166. **except** IndexError:
167. **continue**
169. **return** children

172. **def** in\_closed(child, closed):
173. """
174. Checks if a child is already in the closed set (already been evaluated).
175. :param child: Node object of child to be tested.
176. :param closed: List of nodes in closed set.
177. :return: Boolean as to whether or not the child is in the closed set.
178. """
180. **for** node **in** closed:
181. **if** node.x == child.x **and** node.y == child.y:
182. **return** True
183. **return** False

### Data structures

1. \_\_author\_\_ = 'Will Evans'
3. **import** json
4. **import** local\_database
5. **import** pygame as pg
6. **import** os

9. **class** PriorityQueue:
11. **def** \_\_init\_\_(self):
12. self.queue = []
14. **def** is\_empty(self):
15. **return** len(self.queue) == 0
17. **def** en\_queue(self, node):
18. self.queue.append(node)
20. **def** pop(self):
21. self.queue.sort(key=**lambda** x: x.f\_score)
22. **return** self.queue.pop(0)
24. **def** has(self, child):
25. **for** node **in** self.queue:
26. **if** node.x == child.x **and** node.y == child.y:
27. **return** True
28. **return** False

31. **def** get\_maze(maze\_id):
32. """
33. Temporary function to retrieve the maze from a json file.
34. :param maze\_id: Each maze in the file has a maze ID
35. :return: 2D list of the required maze.
36. """
38. **return** json.loads(local\_database.get\_maze(maze\_id))

41. **class** Tile:
42. **def** \_\_init\_\_(self, tile\_x, tile\_y, \_type, win\_scale, skin):
43. """
44. Class for a tile, containing the type of tile, position, skin and how to blit it.
45. :param tile\_x: Tile x-coord (not pixel)
46. :param tile\_y: Tile y-coord (not pixel)
47. :param \_type: The tile type stored as a number (directly from maze json file).
48. :param win\_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
49. :param skin: Tile's image.
50. """
52. self.pos = (tile\_x, tile\_y)
53. self.x = tile\_x
54. self.y = tile\_y
56. **if** \_type == 0:
57. self.type = 'pellet'
58. self.colour = (0, 0, 0)
60. **elif** \_type == 1:
61. self.type = 'wall'
62. self.colour = (0, 0, 255)
64. **elif** \_type == 2:
65. self.type = 'power\_pellet'
66. self.colour = (0, 0, 0)
68. **elif** \_type == 3:
69. self.type = 'ghost\_barrier'
70. self.colour = (0, 0, 0)
72. **elif** \_type == 4:
73. self.type = 'empty\_tile'
74. self.colour = (0, 0, 0)
76. **elif** \_type == 5:
77. self.type = 'out\_of\_bounds'
78. self.colour = (0, 0, 0)
80. **elif** \_type == 6:
81. self.type = 'inside'
82. self.colour = (0, 0, 0)
84. self.skin = skin
86. self.rect = pg.Rect(tile\_x \* 12 \* win\_scale, tile\_y \* 12 \* win\_scale, 12 \* win\_scale, 12 \* win\_scale)
88. **def** display(self, win):
89. """
90. Displays tile.
91. :param win: The current window, all objects must be blitted to this window to be displayed.
92. :return: None
93. """
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)

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. """
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'
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. """
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\_colour, skin\_name)
128. skin = pg.transform.scale(pg.image.load(skin\_address), (12 \* self.win\_scale, 12 \* self.win\_scale))
130. **elif** data == 3:
131. skin\_address = os.path.join('Resources', 'sprites', 'walls', skin\_colour, 'ghost\_barrier.png')
132. skin = pg.transform.scale(pg.image.load(skin\_address), (12 \* self.win\_scale, 12 \* self.win\_scale))
133. **else**:
134. skin = None
135. tiles[y].append(Tile(x, y + 3, data, self.win\_scale, skin))
137. **return** tiles
139. **def** change\_skin(self):
140. """
141. Changes the colour of all tiles in the 2D list.
142. :return: None
143. """
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)
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 displayed.
155. :return: None
156. """
158. **for** row **in** self.tiles:
159. **for** tile **in** row:
160. tile.display(win)
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 surrounding 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. """
172. tile\_map = [[0 **if** x == 2 **else** x **for** x **in** row] **for** row **in** tile\_map]
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. }
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
192. **if** new\_x < 0 **or** new\_y < 0:
193. adjacent.update({key: None})
194. **continue**
196. adjacent.update({key: tile\_map[tile\_y + y][tile\_x + x]})
197. **except** IndexError:
198. adjacent.update({key: None})
200. values = tuple(list(adjacent.values())[:4])
201. values\_diag = tuple(adjacent.values())
203. # ghost edges
204. **if** values == (4, 3, 1, 4):
205. **return** 'left\_end\_ghost'
207. **if** values == (4, 1, 3, 4):
208. **return** 'right\_end\_ghost'
210. **if** values == (4, 1, 1, 4) **and** tile\_y == 12:
211. **return** 'lower\_boundary'
213. **if** values == (4, 1, 1, 4) **and** tile\_y == 16:
214. **return** 'upper\_boundary'
216. **if** values == (1, 4, 4, 1) **and** tile\_x == 10:
217. **return** 'right\_boundary'
219. **if** values == (1, 4, 4, 1) **and** tile\_x == 17:
220. **return** 'left\_boundary'
222. # ghost corners
223. **if** values\_diag == (1, 1, 4, 4, 4, 4, 4, 4):
224. **return** 'left\_upper\_corner\_ghost'
226. **if** values\_diag == (1, 4, 1, 4, 4, 4, 4, 4):
227. **return** 'right\_upper\_corner\_ghost'
229. **if** values\_diag == (4, 4, 1, 1, 4, 4, 4, 4):
230. **return** 'right\_lower\_corner\_ghost'
232. **if** values\_diag == (4, 1, 4, 1, 4, 4, 4, 4):
233. **return** 'left\_lower\_corner\_ghost'
235. # boundaries edges
236. values\_temp = tuple([1 **if** x **is** None **else** x **for** x **in** values])
238. **if** values == (0, 1, 1, None) **or** values == (0, 1, 1, 5) **or** values\_temp == (4, 1, 1, 5):
239. **return** 'upper\_boundary'
241. **if** values == (None, 1, 1, 0) **or** values == (5, 1, 1, 0) **or** values\_temp == (5, 1, 1, 4):
242. **return** 'lower\_boundary'
244. **if** values == (1, 0, None, 1) **or** values == (1, 0, 4, 1) **or** values == (1, 0, 5, 1):
245. **return** 'left\_boundary'
247. **if** values == (1, None, 0, 1) **or** values == (1, 4, 0, 1) **or** values == (1, 5, 0, 1):
248. **return** 'right\_boundary'
250. # boundary spits
251. **if** values\_diag == (1, 1, 1, None, None, 1, 0, None):
252. **return** 'upper\_left\_spit\_boundary'
254. **if** values\_diag == (1, 1, 1, None, None, 0, 1, None):
255. **return** 'upper\_right\_spit\_boundary'
257. **if** values\_diag == (None, 1, 1, 1, 1, None, None, 0):
258. **return** 'lower\_left\_spit\_boundary'
260. **if** values\_diag == (None, 1, 1, 1, 0, None, None, 1):
261. **return** 'lower\_right\_spit\_boundary'
263. **if** values\_diag == (1, None, 1, 1, None, None, 1, 0):
264. **return** 'right\_upper\_spit\_boundary'
266. **if** values\_diag == (1, None, 1, 1, None, None, 0, 1):
267. **return** 'right\_lower\_spit\_boundary'
269. **if** values\_diag == (1, 1, None, 1, 1, 0, None, None):
270. **return** 'left\_lower\_spit\_boundary'
272. **if** values\_diag == (1, 1, None, 1, 0, 1, None, None):
273. **return** 'left\_upper\_spit\_boundary'
275. # boundary corners
276. **if** values == (1, 1, None, None) **or** values == (1, 1, None, 5):
277. **return** 'left\_upper\_boundary'
279. **if** values == (1, None, 1, None) **or** values == (1, None, 1, 5):
280. **return** 'right\_upper\_boundary'
282. **if** values == (None, None, 1, 1) **or** values == (5, None, 1, 1):
283. **return** 'right\_lower\_boundary'
285. **if** values == (None, 1, None, 1) **or** values == (5, 1, None, 1):
286. **return** 'left\_lower\_boundary'
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])
292. **if** values\_temp == (1, 1, 0, 0):
293. **return** 'left\_upper\_corner'
295. **if** values\_temp == (1, 0, 1, 0):
296. **return** 'right\_upper\_corner'
298. **if** values\_temp == (0, 1, 0, 1):
299. **return** 'left\_lower\_corner'
301. **if** values\_temp == (0, 0, 1, 1):
302. **return** 'right\_lower\_corner'
304. **if** values\_diag\_temp == (1, 1, 1, 1, 1, 0, 1, 1):
305. **return** 'left\_upper\_inside\_corner'
307. **if** values\_diag\_temp == (1, 1, 1, 1, 1, 1, 0, 1):
308. **return** 'right\_upper\_inside\_corner'
310. **if** values\_diag\_temp == (1, 1, 1, 1, 0, 1, 1, 1):
311. **return** 'left\_lower\_inside\_corner'
313. **if** values\_diag\_temp == (1, 1, 1, 1, 1, 1, 1, 0):
314. **return** 'right\_lower\_inside\_corner'
316. # edges
317. values\_temp = tuple([1 **if** x == 6 **else** x **for** x **in** [0 **if** x == 4 **else** x **for** x **in** values]])
318. **if** values\_temp == (1, 1, 1, 0):
319. **return** 'upper\_edge'
321. **if** values\_temp == (1, 1, 0, 1):
322. **return** 'right\_edge'
324. **if** values\_temp == (0, 1, 1, 1):
325. **return** 'lower\_edge'
327. **if** values\_temp == (1, 0, 1, 1):
328. **return** 'left\_edge'
330. **return** 'temp'

### Multiplayer Sprites

1. \_\_author\_\_ = 'Will Evans'
3. **import** sprites
4. **import** pygame as pg
5. **import** random
6. **from** pathfinding **import** Manhattan as Search

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.
15. :param win\_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
16. :param client: Client object from networking module (controls the sending and receiving of player data.
17. :param client\_id: ClientID (comes client object, but is stored in multiplayer section).
18. """
20. super().\_\_init\_\_(resource\_pack, maze, win\_scale)
21. self.client = client
22. self.client\_id = client\_id
24. **def** update(self, events):
25. """
26. Gets input from Client object.
27. :param events: Needed to keep the same signature, but not used.
28. :return: None.
29. """
31. **if** **not** self.dead:
33. move = self.client.get\_data(self.client\_id, 'move')
34. pos = self.client.get\_data(self.client\_id, 'pos')
35. self.facing, self.skin = self.get\_skin(move)
36. **if** move **is** **not** None:
37. self.skin\_clock += 1
38. **else**:
39. **if** **not** self.sound\_channel.get\_busy():
40. self.sound\_channel.play(self.sounds['siren.wav'])
41. self.update\_pos(pos)
42. **else**:
43. **if** self.death\_animation():
44. **return** None
45. **return** self.dead
47. **def** update\_pos(self, pos):
48. self.update\_tile()
50. self.x, self.y = pos
52. self.x \*= self.win\_scale
53. self.y \*= self.win\_scale
55. self.skin\_rect = self.skins['{}\_{}.png'.format(self.facing, self.skin)].get\_rect(center=(self.x, self.y))
56. self.rect = pg.Rect(self.x - int(6 \* self.win\_scale),
57. self.y - int(6 \* self.win\_scale),
58. 12 \* self.win\_scale,
59. 12 \* self.win\_scale)

62. **class** ClientPlayerPacMan(ClientPacMan):
63. **def** \_\_init\_\_(self, resource\_pack, maze, win\_scale, client, client\_id):
64. """
65. Client side Pac-Man sprite that is controlled by keyboard inputs.
66. :param resource\_pack: Contains the path to the folder containing the skins for the sprite.
67. :param maze: Two dimensional list representation of the maze.
68. :param win\_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
69. :param client: Client object from networking module (controls the sending and receiving of player data.
70. :param client\_id: ClientID (comes client object, but is stored in multiplayer section).
71. """
73. super().\_\_init\_\_(resource\_pack, maze, win\_scale, client, client\_id)
75. **def** update(self, events):
76. """
77. Sends client move to server.
78. :param events: Used to get keyboard inputs.
79. :return: None.
80. """
82. self.client.update\_data('client\_move', self.get\_input(events))
83. self.client.send\_player\_data()
85. **return** super().update(events)
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 returned.
92. """
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'
105. **return** self.buffer\_move

108. **class** ClientGhost(sprites.Ghost):
109. **def** \_\_init\_\_(self, resource\_pack, position, target, maze, win\_scale, level, client, client\_id):
110. """
111. Client side ghost sprite that takes input from another client or possibly the server (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 different 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 receiving of player data.
120. :param client\_id: ClientID (comes client object, but is stored in multiplayer section).
121. """
123. super().\_\_init\_\_(resource\_pack, position, target, maze, win\_scale, level)
125. self.client = client
126. self.client\_id = client\_id
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. """
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()
143. **def** update\_pos(self, pos):
144. self.x, self.y = pos
145. self.x \*= self.win\_scale
146. self.y \*= self.win\_scale
148. self.skin\_rect = self.skins['{}\_{}.png'.format(self.facing, self.skin)].get\_rect(center=(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)

155. **class** ClientPlayerGhost(ClientGhost):
156. **def** \_\_init\_\_(self, resource\_pack, position, target, maze, win\_scale, level, client, client\_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 different 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 receiving of player data.
167. :param client\_id: ClientID (comes client object, but is stored in multiplayer section).
168. """
170. super().\_\_init\_\_(resource\_pack, position, target, maze, win\_scale, level, client, client\_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)
177. **def** update(self, events):
178. """
179. Sends client move to server.
180. :param events: Used to get keyboard inputs.
181. :return: None.
182. """
184. self.client.update\_data('client\_move', self.get\_input(events))
185. self.client.send\_player\_data()
186. super().update(events)
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 displayed.
193. :return: None
194. """
196. win.blit(self.skins['{}\_{}.png'.format(self.facing, self.skin)], self.skin\_rect)
197. win.blit(self.spotlight, self.spotlight.get\_rect(center=self.skin\_rect.center))
199. **def** get\_input(self, events):
200. """
201. Gets inputs from keyboard events.
202. :param events: Used to get keyboard inputs.
203. :return: Buffer move, so that if there isn't a keyboard input the last input will be returned.
204. """
206. **for** event **in** events:
207. **if** event.type == pg.KEYDOWN:
208. **if** event.key == pg.K\_UP:
209. self.buffer\_move = 'n'
210. **elif** event.key == pg.K\_RIGHT:
211. self.buffer\_move = 'e'
212. **elif** event.key == pg.K\_DOWN:
213. self.buffer\_move = 's'
214. **elif** event.key == pg.K\_LEFT:
215. self.buffer\_move = 'w'
217. **return** self.buffer\_move

220. # Server side Pac-Man
221. **class** ServerPacMan(sprites.PacMan):
222. **def** \_\_init\_\_(self, resource\_pack, maze, win\_scale, server, client\_id):
223. """
224. Server side Pac-Man sprite controlled by another client.
225. :param resource\_pack: Contains the path to the folder containing the skins for the sprite.
226. :param maze: Two dimensional list representation of the maze.
227. :param win\_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
228. :param client: Client object from networking module (controls the sending and receiving of player data.
229. :param client\_id: ClientID (comes client object, but is stored in multiplayer section).
230. """
232. super().\_\_init\_\_(resource\_pack, maze, win\_scale)
233. self.server = server
234. self.client\_id = client\_id
236. **def** update(self, events):
237. """
238. Gets input from Server object, as the get\_move method has been overridden below.
239. :param events: Needed to keep the same signature, but not used.
240. :return: None.
241. """
243. self.dead = super().update(events)
245. pos = [coord / self.win\_scale **for** coord **in** [self.x, self.y]]
246. self.server.update\_data(self.client\_id, 'pos', pos)
247. self.server.update\_data(self.client\_id, 'move', self.online\_move)
248. **return** self.dead
250. **def** get\_move(self, events):
251. move = self.server.get\_data(self.client\_id, 'client\_move')
253. self.buffer\_move = move
255. **return** self.check\_move(self.buffer\_move)
257. **def** update\_score(self, score):
258. self.server.update\_data(self.client\_id, 'score', score)

261. # Server side ghost
262. **class** ServerGhost(sprites.Ghost):
263. **def** \_\_init\_\_(self, resource\_pack, position, target, maze, win\_scale, level, server, client\_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 different 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 receiving of player data.
274. :param client\_id: ClientID (comes client object, but is stored in multiplayer section).
275. """
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
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. """
290. super().update(events)
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)
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. """
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
311. **return** self.check\_move(self.buffer\_move)
313. **def** add\_points(self, points):
314. """
315. Updates points for that particular player based on interactions the sprite has server side.
316. :param points: Points to be added to player's total.
317. :return: None.
318. """
320. score = self.server.get\_data(self.client\_id, 'score')
321. self.server.update\_data(self.client\_id, 'score', score + points)

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 receiving of player data.
332. """
334. super().\_\_init\_\_(resource\_pack, maze, win\_scale)
335. self.server = server
336. self.client\_id = 0
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. """
345. self.dead = super().update(events)
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)
351. **return** self.dead
353. **def** update\_score(self, score):
354. self.server.update\_data(self.client\_id, 'score', score)

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 receiving of player data.
365. :param client\_id: ClientID (comes client object, but is stored in multiplayer section).
366. """
368. super().\_\_init\_\_(resource\_pack, maze, win\_scale)
369. self.server = server
370. self.client\_id = client\_id
372. # Search
373. self.search = Search(maze.tile\_map)
375. # Path finding
376. self.path = self.get\_path()
377. self.next\_coords = self.path[1]
379. self.next\_x = 0
380. self.next\_y = 0
382. self.axis\_change = 'x'
384. **def** update(self, events):
385. """
386. Updates server with move.
387. :param events: Used to get keyboard events.
388. :return: None.
389. """
391. self.dead = super().update(events)
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)
397. **return** self.dead
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. """
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
413. tile\_x, tile\_y = self.tile.pos
414. x, y = self.next\_coords
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
424. x \*= 12 \* self.win\_scale
425. x += 6 \* self.win\_scale
427. y \*= 12 \* self.win\_scale
428. y += 6 \* self.win\_scale
430. self.next\_x = x
431. self.next\_y = y
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'
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'
449. **def** get\_path(self):
450. """
451. Gets path to random pellet tile.
452. :return: Path
453. """
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)
463. **def** update\_score(self, score):
464. self.server.update\_data(self.client\_id, 'score', score)

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 different 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 receiving of player data.
480. """
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))
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)
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)
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))
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. """
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
521. **return** self.check\_move(self.buffer\_move)
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'
535. **return** self.buffer\_move
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)

542. **class** ServerBlinky(sprites.Blinky):
543. **def** \_\_init\_\_(self, resource\_pack, target, maze, win\_scale, level, server, client\_id):
545. super().\_\_init\_\_(resource\_pack, target, maze, win\_scale, level)
546. self.server = server
547. self.client\_id = client\_id
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)
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)

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
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)
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)

577. **class** ServerInky(sprites.Inky):
578. **def** \_\_init\_\_(self, resource\_pack, target, maze, win\_scale, level, blinky, server, client\_id):
579. super().\_\_init\_\_(resource\_pack, target, maze, win\_scale, level, blinky)
580. self.server = server
581. self.client\_id = client\_id
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)
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)

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
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)
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

1. \_\_author\_\_ = 'Will Evans'
3. **import** threading
4. **import** socket
5. **import** json
6. **import** copy
7. **import** time

10. **class** Connection:
11. **def** \_\_init\_\_(self, user\_ip, conn, user\_id, players):
12. """
13. Class for each connection the server has with a client. Controls all information going from server to client.
14. :param user\_ip: IP for user.
15. :param conn: Socket connection used to send and receive data from client.
16. :param user\_id: A number from 1-4, used to match connection with avatar.
17. :param players: List of all players and their information, used to update client's screen.
18. """
20. self.connected = True
21. self.\_\_player\_data = {
23. 'ready': None,
24. 'client\_move': None,
26. }
28. self.\_\_connected = False
29. self.\_\_IP = user\_ip
30. self.id = user\_id
31. self.\_\_PORT = 50007
32. self.\_\_conn = conn
34. # Essential trade of info
35. self.send({
37. 'client\_id': user\_id,
38. 'players': players
40. })
42. self.\_\_player\_data['name'] = self.receive()['name']
44. threading.Thread(target=self.update).start()
46. **def** update(self):
47. """
48. Runs once every frame, updates each item in player\_data dictionary. Only used at the beginning of the connection
49. to share basic information between the client and the host.
50. :return: None
51. """
53. **while** self.connected:
54. data = self.receive()
55. **if** data **is** **not** None:
56. **for** attribute, value **in** data.items():
57. self.\_\_player\_data[attribute] = value
59. **def** receive(self):
60. """
61. Receives data from client (player\_data).
62. :return: Data in string form (not bytes).
63. """
65. **try**:
66. data = self.\_\_conn.recv(4096)
67. **return** json.loads(data)
68. **except** Exception as e:
69. **print**("disconnected: {}".format(e))
71. **def** send(self, data):
72. """
73. Converts data into bytes and sends to client. This is usually the 2D dictionary of player info.
74. :param data: Data that is to be sent (player\_data).
75. :return: None
76. """
78. **if** self.connected:
79. data = json.dumps(data)
80. data = bytes(data, 'utf-8')
81. **try**:
82. self.\_\_conn.sendall(data)
83. **except** ConnectionResetError:
84. **print**("disconnected")
85. self.connected = False
87. **def** get\_player\_data(self):
88. **return** self.\_\_player\_data
90. **def** get\_id(self):
91. **return** self.id
93. **def** close(self):
94. """
95. When a client quits the connection must be closed. Closes the socket connection and marks the object as not
96. connected.
97. :return: None
98. """
100. self.\_\_conn.close()
101. self.connected = False

104. **class** Server:  # Instantiates whenever a user clicks create game.
105. **def** \_\_init\_\_(self, name):
106. """
107. Class for server that controls the sending and receiving of game data for each player between the host and
108. all clients connected.
109. :param name: Host's name. Comes form database based on user's sign in details.
110. """
112. self.\_\_run = True
113. self.test\_count = 0
114. self.\_\_players\_template = {
116. 0:
117. {
118. 'name':     '{} [host]'.format(name),
119. 'skin':     'pac-man',
120. 'score':    0,
121. 'ready':    None,
122. 'pos':      [167, 318],
123. 'move':     None,
124. 'countdown': None,
125. 'start':    False,
126. 'end':      True,
127. 'place':    None,
128. 'finished': False
129. },
131. 1:
132. {
133. 'name':     None,
134. 'skin':     'blinky',
135. 'score':    0,
136. 'ready':    None,
137. 'pos':      [168, 176],
138. 'move':     None,
139. 'client\_move': None,
140. 'place': None
141. },
143. 2:
144. {
145. 'name':     None,
146. 'skin':     'pinky',
147. 'score':    0,
148. 'ready':    None,
149. 'pos':      [168, 214],
150. 'move':     None,
151. 'client\_move': None,
152. 'place': None
153. },
155. 3:
156. {
157. 'name':     None,
158. 'skin':     'inky',
159. 'score':    0,
160. 'ready':    None,
161. 'pos':      [144, 214],
162. 'move':     None,
163. 'client\_move': None,
164. 'place': None
165. },
166. 4:
167. {
168. 'name':     None,
169. 'skin':     'clyde',
170. 'score':    0,
171. 'ready':    None,
172. 'pos':      [192, 214],
173. 'move':     None,
174. 'client\_move': None,
175. 'place': None
176. }
177. }
179. self.\_\_players = copy.deepcopy(self.\_\_players\_template)
180. self.\_\_IP = self.get\_ip()
181. self.\_\_port = 50007
182. self.\_\_connections = []
183. self.\_\_s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)
184. self.\_\_s.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)
185. self.\_\_s.bind((self.\_\_IP, self.\_\_port))
187. self.has\_ai = False
189. self.searching\_for\_clients = True
190. threading.Thread(target=self.connect).start()
191. threading.Thread(target=self.receive).start()
192. threading.Thread(target=self.check\_connections).start()
194. **def** connect(self):
195. """
196. Before the game starts the this method will listen for new connections, and then create a connection when one
197. is received.
198. :return: None
199. """
201. **while** self.searching\_for\_clients **and** self.\_\_run:
202. self.\_\_s.listen(1)
203. **if** self.searching\_for\_clients:
204. **try**:
205. conn, addr = self.\_\_s.accept()
206. available\_ids = [k **for** k, v **in** self.\_\_players.items() **if** v['name'] **is** None]
207. client\_id = available\_ids[0]
208. connection = Connection(addr[0], conn, client\_id, self.\_\_players)
209. self.\_\_connections.append(connection)
210. **except** Exception as e:
211. **print**("Connect: {}".format(e))
213. **def** receive(self):
214. """
215. This method updates the player data based on information sent my each client to each of the connections. Updates
216. twice a second to account for any syncing issues between clients and server.
217. :return: None
218. """
220. **while** self.\_\_run:
221. **try**:
222. **for** connection **in** self.\_\_connections:
223. data = connection.get\_player\_data()
224. **for** attribute, value **in** data.items():
225. self.\_\_players[connection.get\_id()][attribute] = value
226. **except** Exception as e:
227. **print**(e)
228. time.sleep(1/120)
230. **def** update\_data(self, client\_id, key, value):
231. self.\_\_players[client\_id][key] = value
233. **def** send\_data(self):
234. **for** connection **in** self.\_\_connections:
235. connection.send(self.\_\_players)
237. **def** check\_connections(self):
238. """
239. Runs while the server is running. Checks every connection to see if it is still there, prevents errors with
240. connections.
241. :return: None
242. """
244. **while** self.\_\_run:
245. **for** connection **in** self.\_\_connections:
246. **if** **not** connection.connected:
247. connection.close()
248. self.\_\_players[connection.get\_id()] = self.\_\_players\_template[connection.get\_id()].copy()
249. self.\_\_connections.remove(connection)
251. # Limits the number of times the thread can run to save power
252. # 1/120 has been chosen to ensure it is run at least once between frames
253. time.sleep(1/120)
255. **def** get\_data(self, client\_id, type):
256. **return** self.\_\_players[client\_id][type]
258. **def** get\_players(self):
259. **return** self.\_\_players
261. **def** get\_ip(self):
262. **return** socket.gethostbyname(socket.gethostname())
264. **def** get\_client\_id(self):
265. **return** 0
267. **def** swap(self, client\_id):
268. """
269. Whichever client\_id is passed into this method will become the next Pac-Man. This happens when a ghost catches
270. Pac-Man and is needed to swap the skins and start position of players around when they become Pac-Man.
271. :param client\_id: ClientID of player that is becoming Pac-Man
272. :return: None
273. """
275. pac\_man\_id = [id **for** id, data **in** self.\_\_players.items() **if** data['skin'] == 'pac-man'][0]
277. pac\_man\_skin = "{}".format(self.\_\_players[pac\_man\_id]['skin'])
278. client\_skin = "{}".format(self.\_\_players[client\_id]['skin'])
280. self.\_\_players[pac\_man\_id]['skin'] = client\_skin
281. self.\_\_players[client\_id]['skin'] = pac\_man\_skin
283. **def** reset(self):
284. """
285. At the end of each round the positions of each sprite need to be reset according to the player\_template in order
286. for the sprites to spawn in the correct place in the following round.
287. :return: None
288. """
290. **for** client\_id, client\_data **in** self.\_\_players.items():
291. og\_player\_data = [og\_client\_data
292. **for** og\_client\_data **in** self.\_\_players\_template.values()
293. **if** og\_client\_data['skin'] == client\_data['skin']][0]
295. client\_data['pos'] = og\_player\_data['pos'][::]
297. **def** add\_ai(self):
298. """
299. Changes name of all available ghosts into 'AI', which will mean they become controlled by AI in the game. This
300. method is run when the game countdown is started.
301. :return: None
302. """
304. **for** client\_data **in** self.\_\_players.values():
305. **if** client\_data['name'] **is** None:
306. client\_data['name'] = 'AI'
307. self.has\_ai = True
309. **def** remove\_ai(self):
310. """
311. This changes all AI ghosts back to None when the game countdown is stopped (to allow for more players to join).
312. :return:
313. """
315. **for** client\_data **in** self.\_\_players.values():
316. **if** client\_data['name'] == 'AI':
317. client\_data['name'] = None
318. self.has\_ai = False
320. **def** quit(self):
321. """
322. Correctly adjusts attributes so that all threads and connections terminate when the server is no longer needed.
323. :return: None
324. """
326. self.searching\_for\_clients = False
327. self.\_\_run = False
328. **for** connection **in** self.\_\_connections:
329. connection.close()
330. self.\_\_s.close()

333. **class** Client:
334. **def** \_\_init\_\_(self, host\_ip, name):
335. """
336. Runs on clients when the user selects join game and enters a GameID.
337. :param host\_ip: GameID that the user enters. It is actually the host's local IPV4 address.
338. :param name: Name of client according to the Users table in the database.
339. """
341. self.\_\_host\_ip = host\_ip
342. self.\_\_name = name
343. self.connected = False
344. self.connection\_failed = None
346. self.\_\_player\_data = {
348. 'ready':        None,
349. 'client\_move':  None,
351. }
353. self.\_\_port = 50007
354. self.\_\_s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)
356. **try**:
357. self.\_\_s.connect((self.\_\_host\_ip, self.\_\_port))
358. self.connection\_failed = False
359. **except** (ConnectionRefusedError, socket.gaierror, TypeError):
360. self.connection\_failed = True
362. **if** **not** self.connection\_failed:
363. # first share of essential data
364. init\_data = self.receive()
365. **if** init\_data **is** **not** None:
366. self.\_\_client\_id = init\_data['client\_id']
367. self.\_\_players = init\_data['players']
368. self.send({'name': self.\_\_name})
370. self.connected = True
371. threading.Thread(target=self.update).start()
373. **def** send(self, data):
374. data = json.dumps(data)
375. data = bytes(data, 'utf-8')
376. **try**:
377. self.\_\_s.sendall(data)
378. **except** OSError:
379. **print**("disconnected")
381. **def** receive(self):
382. **try**:
383. data = self.\_\_s.recv(1024)
384. **return** json.loads(data)
385. **except** ConnectionResetError as e:
386. self.connected = False
387. **print**(e)
388. **except** WindowsError as e:
389. self.connected = False
390. **print**(e)
391. **except** Exception as e:
392. **print**(e)
394. **def** update(self):
395. """
396. Receives data from server and sets equal to players.
397. :return: None
398. """
400. **while** self.connected:
401. data = self.receive()
402. **if** data **is** **not** None:
403. self.\_\_players = data
405. **def** update\_data(self, key, value):
406. self.\_\_player\_data[key] = value
408. **def** get\_data(self, client\_id, type):
409. **return** self.get\_players()[client\_id][type]
411. **def** send\_player\_data(self):
412. self.send(self.\_\_player\_data)
414. **def** get\_players(self):
415. """
416. Corrects integer keys in players (as they are saved as strings when converted to and from bytes) and returns.
417. :return: Dictionary: players
418. """
419. **return** {int(k): v **for** k, v **in** self.\_\_players.items()}
421. **def** get\_client\_id(self):
422. **return** self.\_\_client\_id  # Ask here if client id is None
424. **def** end(self):
425. self.connected = False
426. self.\_\_s.close()

429. **if** \_\_name\_\_ == "\_\_main\_\_":
430. **pass**

### GUI

1. \_\_author\_\_ = 'Will Evans'
2. **import** pygame as pg
3. **import** os.path
4. **import** sprites

7. **class** Word:
8. **def** \_\_init\_\_(self, content, pos, colour, font\_size, win\_scale, italic=False, bold=False, centre=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 related 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. """
22. self.content = content
24. x, y = pos
25. self.x = x \* win\_scale
26. self.y = y \* win\_scale
28. self.colour = colour
29. self.centre = centre
30. self.left = left
31. self.\_\_letters = []
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)
38. **if** self.content **is** **not** None:
39. self.render()
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
46. """
48. **if** self.content **is** **not** None:
49. win.blit(self.\_rendered\_text, self.\_text\_rect)
51. **def** render(self):
52. """
53. Renders font. Takes the content and colour and converts this into a font object. Then a rect object is created
54. based on the position and alignment instructions.
55. :return: None
56. """
58. self.\_rendered\_text = self.\_font.render(self.content, True, self.colour)
60. **if** self.centre:
61. self.\_text\_rect = self.\_rendered\_text.get\_rect(center=(self.x, self.y))
62. **elif** self.left:
63. self.\_text\_rect = self.\_rendered\_text.get\_rect(midleft=(self.x, self.y))
64. **else**:
65. self.\_text\_rect = self.\_rendered\_text.get\_rect(midright=(self.x, self.y))

68. **class** LiveWord:
69. **def** \_\_init\_\_(self, content, y, font\_size, win\_scale, highlight\_colour=(255, 255, 255)):
70. """
71. Live words have the ability to highlight when selected by the user. Seperate to word class as live
72. words must be rendered letter by letter so that when the highlighted layer is shown, it correctly covers each
73. individual letter.
74. :param content: String: content of the word.
75. :param y: y coordinate of word (do not require x as they are automatically centred).
76. :param font\_size: Integer: Font size
77. :param win\_scale: Integer: Window Scale (How large the window is - must be multiplied by all size related
78. variables).
79. :param highlight\_colour: (r, g, b): default is set to white but can be any colour.
80. """
82. self.program = content
83. self.content = list(content)
84. self.\_\_letters = []
85. self.\_\_react = False
86. self.\_\_letter\_spacing = font\_size \* 7/12
88. **for** num, letter **in** enumerate(content):
89. x = 173 + ((- len(content) / 2) + num) \* self.\_\_letter\_spacing
90. self.\_\_letters.append(LiveLetter(letter, x, y, font\_size, win\_scale, highlight\_colour))
92. **def** react(self):
93. self.\_\_react = True
95. **def** display(self, win):
96. """
97. Displays each letter in the word, sets react to False so that if the mouse stops colliding with the word,
98. then the word will stop displaying as highlighted.
99. :param win: The current window, all objects must be blitted to this window to be displayed.
100. :type win: Surface.
101. :return: None.
102. """
104. **for** letter **in** self.\_\_letters:
105. letter.display(self.\_\_react, win)
106. self.\_\_react = False
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. """
118. **for** letter **in** self.\_\_letters:
119. **if** letter.check\_mouse(x, y):
120. **return** True
121. **return** False
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. """
133. **for** letter **in** self.\_\_letters:
134. **if** letter.check\_mouse(x, y):
135. **return** True
137. **def** get\_program(self):
138. **return** self.program

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. """
159. self.\_\_font\_size = font\_size \* win\_scale
160. self.x = x \* win\_scale
161. self.y = y \* win\_scale
163. font\_path = os.path.join('resources', 'fonts', 'ARCADECLASSIC.TTF')
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))
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))
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))
177. **def** display(self, react, win):
178. """
179. Blits the letter object ot the screen. If react is True then a black outline and another 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 displayed.
184. :type win: Surface.
185. :return: None.
186. """
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)
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. """
205. **return** self.\_\_text\_rect.collidepoint(x, y)

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 appearance 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. """
220. self.content = content
221. self.content\_displayed = ''
222. self.characters\_displayed\_num = 0
223. self.finished = False
225. self.rendered\_font = Word(self.content\_displayed, pos, colour, font\_size, win\_scale, False, False, False, True)
227. self.frame\_count = 0
228. self.frame\_cap = frame\_cap
230. **def** update(self, events):
231. self.frame\_count += 1
232. **if** self.frame\_count == self.frame\_cap:
233. self.frame\_count = 0
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()
243. **def** display(self, win):
244. self.rendered\_font.display(win)
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()

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 onto 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 mspacman.
261. :param win\_scale: Window Scale (How large the window is - must be multiplied by all size related variables).
262. """
264. # Essential
265. self.win\_scale = win\_scale
266. self.paused = False
267. self.finished = False
269. self.add\_mspacman = add\_mspacman
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
280. # This is a lovely trick to calculate how long the text will be (so that we know when 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)
285. self.boxes = []
287. **while** len(self.content\_remaining) > 0:
288. self.render\_box()
290. self.active\_box\_index = 0
291. self.active\_box = self.boxes[self.active\_box\_index]
293. # Box
294. self.box = Box((7, 360), (321, 60), (230, 230, 230), 2, win\_scale)
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))
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)
307. **def** update(self, events):
309. space\_pressed = False
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**
319. self.paused = all([line.finished **for** line **in** self.active\_box])
321. **for** event **in** events:
322. **if** event.type == pg.KEYDOWN:
323. **if** event.key == pg.K\_SPACE:
324. space\_pressed = True
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()
336. **if** **not** self.paused **and** self.add\_mspacman:
337. self.mspacman.update(events)
339. **def** display(self, win):
341. # Transparent Win
342. win.blit(self.transparent\_win, (7 \* self.win\_scale, 360 \* self.win\_scale))
344. # Words
346. **for** line **in** self.active\_box:
347. line.display(win)
349. # Box
350. self.box.display(win)
352. # MsPacMan
353. **if** self.add\_mspacman:
354. self.mspacman.display(win)
356. **def** render\_box(self):
357. content\_buffer = ''
358. line\_num = 0
359. lines = []
361. **for** word **in** self.content\_remaining:
362. content\_buffer += word
363. self.content\_remaining = self.content\_remaining[1:]
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))
370. self.boxes.append(lines)
371. **break**
373. current\_length = self.font.size(content\_buffer)[0] + self.font.size(self.content\_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. )
386. line\_num += 1
387. content\_buffer = ''
389. **if** line\_num == 3 **or** ellipsis\_exit:
390. self.boxes.append(lines)
391. **break**

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 aggregated. 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. """
415. x, y = pos
416. w, h = dimensions
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
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 displayed.
429. :type win: Surface.
430. :return: None.
431. """
433. pg.draw.rect(win, self.colour, self.rect, self.width)
435. **def** check\_mouse(self, x, y):
436. **return** self.rect.collidepoint(x, y)

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 time.
458. :type name: String.
459. :param interactive: Whether or not the box should respond to any user input. If false 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 password input boxes.
466. :type private: Boolean.
467. :param max\_length: The maximum amount of characters the box can hold before it no longer accepts any more
468. characters from the user.
469. :type max\_length: Integer.
470. """
472. # Essential
473. self.win\_scale = win\_scale
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
482. self.text\_surface\_width\_og = w
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 \* win\_scale)
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 = ''
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)
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. """
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
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
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
544. # Re-render the text.
545. **if** len(self.user\_input) == 0:
546. self.display\_text = self.name
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
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)
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 displayed.
564. :type win: Surface.
565. :return: None.
566. """
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.win\_scale)
575. **def** get\_input(self):
576. **return** self.user\_input

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 user 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. """
589. # Essential
590. self.win\_scale = win\_scale
592. self.content = content
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. )
606. **def** update(self, events):
607. """
608. The only check that the error box performs is to see if a mouse button has been lifted 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. """
615. **for** event **in** events:
616. **if** event.type **in** [pg.MOUSEBUTTONUP, pg.KEYDOWN]:
617. **return** False
618. **return** True
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 displayed.
624. :type win: Surface.
625. :return: None.
626. """
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)

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 use 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 become the top left point of the
652. button.
653. :type centre: Boolean.
654. """
656. x, y = pos
657. w, h = dimensions
659. self.active = False
660. self.click = False
662. self.tab\_active = False
664. self.react\_boxes = []
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)
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, centre=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, centre=centre)
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 program using the button must
680. implement the button's function and can use the click attribute to decide when to execute 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. """
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
696. self.click = collision **and** click
698. **if** **not** self.tab\_active:
699. self.active = collision
700. **elif** collision:
701. self.tab\_active = False
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 displayed.
707. :type win: Surface.
708. :return: None.
709. """
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)
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. """
725. **return** self.box.check\_mouse(x, y)
727. **def** get\_click(self):
728. **return** self.click

731. **class** Slider:
732. **def** \_\_init\_\_(self, content, x, y, w, h, font\_size, text\_colour, box\_width, win\_scale, centre=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. """
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
770. self.win\_scale = win\_scale
772. self.font\_size = font\_size
773. self.text\_colour = text\_colour
774. self.centre = centre
776. self.react = False
777. self.active = False
778. self.click = False
780. self.content = content
782. self.levels = levels
784. self.level = (level/levels) \* w \* win\_scale
786. self.level\_string = level
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, centre=centre)
792. # Text
793. self.text\_surface = self.get\_text()
795. # Transparent slider
796. self.slider = self.get\_slider()
798. **def** update(self, events):
799. """
800. Checks where on the slider the mouse has been clicked and adjusts the slider and word 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. """
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
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.levels)
819. self.slider = self.get\_slider()
820. self.text\_surface = self.get\_text()
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 displayed.
827. :return: None
828. """
830. win.blit(self.slider, (self.x \* self.win\_scale, self.y \* self.win\_scale))
831. self.text\_surface.display(win)
833. **if** self.react:
834. self.highlight\_box.display(win)
835. **else**:
836. self.box.display(win)
838. **def** get\_text(self):
839. """
840. Instanciates a text object. This must be run every time the level is changed in order to update the number.
841. :return: The word object.
842. """
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
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. """
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
865. **def** check\_mouse(self, x, y):
866. **return** self.box.check\_mouse(x, y)

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)
875. self.active = True
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))
881. self.box = Box((168, 210), (200, 100), (230, 230, 230), 2, win\_scale, centre=True)
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)

889. **class** Icon:
890. **def** \_\_init\_\_(self, pos, imgs, win\_scale, sound=False, toggle=False, target\_program=False):
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 more 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 True.
898. :param target\_program: If the icon should run a sub-program when clicked then the name should be passed here.
899. """
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
908. # Sound
909. **if** self.sound:
910. pg.mixer.music.play()
912. x, y = pos
913. self.\_imgs = [pg.transform.smoothscale(pg.image.load(img), (25 \* win\_scale, 25 \* win\_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
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 displayed.
921. :return: None
922. """
924. win.blit(self.\_imgs[self.\_image\_num], self.\_rect)
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. """
934. **return** self.\_rect.collidepoint(x, y)
936. **def** action(self):
937. """
938. This usually runs when the check\_click function returns True, and completes the icons desired action.
939. :return: None
940. """
942. **if** self.toggle:
943. self.\_image\_num = abs(self.\_image\_num - 1)
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

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. """
16. sql = """CREATE TABLE Users
17. (UserID INTEGER,
18. UserName TEXT,
19. Password TEXT,
20. PRIMARY KEY(UserID))
21. """
22. cursor.execute(sql)

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. """
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)

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. """
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))
67. """
68. cursor.execute(sql)

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. """
78. sql = """CREATE TABLE MultiplayerGameHistory
79. (MultiPlayerGameID INTEGER,
80. GameID INTEGER,
81. PRIMARY KEY(MultiPlayerGameID),
82. FOREIGN KEY(GameID) REFERENCES GameHistory(GameID))
84. """
85. cursor.execute(sql)

88. **def** create\_multiplayer\_player\_history(cursor):
89. """
90. Creates table 'MultiplayerPlayerHistory'
91. :param cursor: Object used to execute SQL within the database.
92. :return: None
93. """
95. sql = """"""
96. cursor.execute(sql)

99. **def** create\_mazes(cursor):
100. """
101. Creates table 'Mazes'
102. :param cursor: Object used to execute SQL within the database.
103. :return: None
104. """
106. sql = """CREATE TABLE Mazes
107. (MazeID INTEGER,
108. UserID INTEGER,
109. Maze TEXT,
110. Date TEXT,
111. PRIMARY KEY(MazeID),
112. FOREIGN KEY(UserID) REFERENCES Users(UserID))
113. """
114. cursor.execute(sql)
116. sql = """
117. INSERT INTO Mazes
118. (Maze)
119. VALUES (?)
121. """
123. **for** maze\_id **in** range(1):
124. mazes = ['Level1', 'Level2']
125. default\_mazes\_path = os.path.join('Resources', 'Levels.txt')
126. with open(default\_mazes\_path, 'r') as json\_file:
127. maze = json.load(json\_file)[mazes[maze\_id]]
129. maze = json.dumps(maze)
130. query(sql, (maze,))

133. **def** create\_db():
134. """
135. Checks whether the database has already been created and if not creates it.
136. :return: None
137. """
139. **if** **not** os.path.exists('data\\database.db'):
140. with sqlite3.connect('data\\database.db') as db:
141. cursor = db.cursor()
143. create\_users(cursor)
144. create\_game\_level(cursor)
145. create\_game\_history(cursor)
146. create\_multiplayer\_game\_history(cursor)
147. create\_mazes(cursor)

150. **def** query(sql, data=None):
151. """
152. Used to execute SQL statments, can also return data.
153. :param sql: The SQL that will be executed.
154. :param data: Data can be added when an SQL statement inputs data into the database.
155. :return: Results if there are any.
156. """
158. 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

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 just created.
173. :param user\_id: Users personal ID
174. :param maze\_id: MazeID for the maze being played on.
175. :return: GameID
176. """
178. sql = """INSERT INTO GameHistory (UserID, MazeID, Date, Time) VALUES (?, ?, ?, ?)"""
179. data = (user\_id, maze\_id, get\_date(), get\_time())
180. query(sql, data)
182. sql = """SELECT Max(GameID) FROM GameHistory"""
183. **return** query(sql)[0][0]

186. **def** get\_highscore():
187. """
188. Returns the current highscore! In one line! One complex query!
189. :return: Highscore
190. """
192. # This is one complex query!
193. **return** query("""SELECT Max(Scores) FROM(SELECT Sum(Score) as Scores from GameLevel Group BY GameID)""")[0][0]

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. """
210. sql = """
211. INSERT INTO GameLevel
212. (GameID, LevelNum, Lives, Score, Length, PelletsEaten, PowerPelletsEaten, GhostsEaten)
213. VALUES (?, ?, ?, ?, ?, ?, ?, ?)
214. """
216. query(sql, (game\_id, level\_num, lives, score, length, pellets\_eaten, power\_pellets\_eaten, ghosts\_eaten))

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. """
227. sql = """
228. UPDATE GameHistory
229. SET Initials=?
230. WHERE GameID=?
231. """
233. **if** len(initials) == 3:
234. query(sql, (initials, game\_id))

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. """
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()))

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 here.
256. :param username: The user's chosen username.
257. :param password: The user's chosen password.
258. :return: None
259. """
261. sql = """
262. INSERT INTO Users (Username, Password)
263. VALUES (?, ?)
264. """
266. query(sql, (username, password))

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. """
276. sql = """
277. SELECT Maze
278. FROM Mazes
279. WHERE MazeID=?
280. """
282. **return** query(sql, (maze\_id,))[0][0]

285. **def** login(username, password):
286. """
287. Checks user provided details against database.
288. :return: UserID
289. """
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

304. **def** check\_sign\_up(user\_name, password, password\_confirm):
305. """
306. Performs the following checks in order: Username length, username availability, password 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 checks failed. (appropriate for
312. given any error).
313. """
315. # Username Length
316. **if** len(user\_name) < 5:
317. **return** False, "Username must be 5 characters"
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"
324. # Password Length
325. **if** len(password) < 7:
326. **return** False, "Password must be 7 characters"
328. # Password Strength
329. **if** **not** re.search("[a-z]", password):
330. **return** False, "Password must contain lower"
332. **if** **not** re.search("[A-Z]", password):
333. **return** False, "Password must contain upper"
335. **if** **not** re.search("[0-9]", password):
336. **return** False, "Password must contain number"
338. # Passwords match
339. **if** password != password\_confirm:
340. **return** False, "Passwords must match"
342. **return** True, None

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. """
352. sql = """SELECT Username FROM Users WHERE UserID=?"""
353. **return** query(sql, (user\_id,))[0][0]

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. """
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()
376. **if** len(stats\_tuple) **is** 0:
377. **return** None
378. **else**:
379. **return** dict(stats\_tuple[0])

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. """
391. **return** query(sql)

394. **def** get\_date():
395. """
396. Gets date.
397. :return: Date
398. """
400. **return** datetime.datetime.now().strftime("%d/%m/%Y")

403. **def** get\_time():
404. """
405. Gets time.
406. :return: Time.
407. """
409. **return** datetime.datetime.now().strftime("%H:%M:%S")

412. **if** \_\_name\_\_ == '\_\_main\_\_':
413. sql = """
414. INSERT INTO Mazes
415. (Maze)
416. VALUES (?)
418. """
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]]
425. maze = json.dumps(maze)
426. query(sql, (maze,))

### Local Settings

1. \_\_author\_\_ = 'Will Evans'
3. **import** json
4. **import** os.path

7. **def** write\_settings():
8. """
9. If there is not a settings file, one will be created.
10. :return: None
11. """
13. file\_path = os.path.join('data', 'settings.json')
15. with open(file\_path, 'w+') as file:
16. json.dump(
18. {
20. 'win\_scale':         2,
21. 'music\_volume':     50,
22. 'game\_volume':      50,
23. 'user\_name':        None,
24. 'password':         None,
26. },
28. file)

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. """
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()
44. with open(file\_path, 'r') as file:
45. data = json.load(file)
46. **return** data[setting]
48. **except** (PermissionError, OSError):
49. **print**("settings fetch error")

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. """
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
67. with open(file\_path, 'w') as file:
68. json.dump(data, file)
69. **break**
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 letter prevents account creation. | Username:  James  Password: cheese123 | Error: ‘Password must contain upper’. | The expected. | None. | 08:32 |
| 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 obstacle in the way. | A keyboard input. (in this case right arrow key) | Pac-Man should not change direction. | The expected. | None. | 16:34 |
| 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 the most efficient. | None. | We can observe the on-screen path, manually work out the shortest path and the two should be the same. | The expected. | None. | 19:55 |
| 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 screen indicating Pac-Man has 3 lives. | The expected. | None. | 28:12 |
| 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 gold, silver and bronze for 1st through 3rd. | None. | The top three high scores should be coloured gold, silver and bronze for 1st through 3rd. | The expected. | None. | 33:15 |
| 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 the menu that the user can interact with, that ultimately enhance the user experience.** | 1 |
| 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 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, | 5.1-5.2 |
| 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 1st through 5th 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.