# 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. Moreover, my game will allow up to 5 more spectators to join each game - a total of 10 users per match. It will also incorporate even more features, such as: maze generation (in keeping with the original map design); personalized local accounts and progress saves, and of course a story spinoff relating to my own interpretation of how the feud between mouth and ghost began.

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; 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 tile map 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.

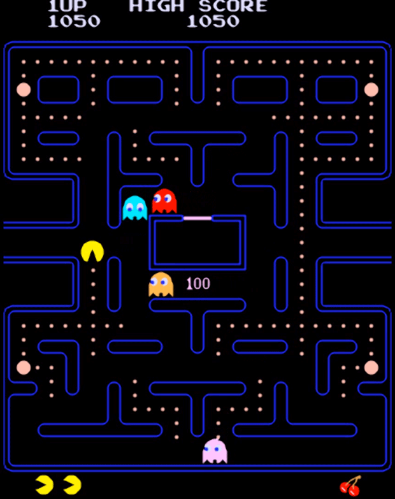
It is also important that my game has a built-in account system. This will allow the player to play online games more easily with friends and encourage competition within the single player modes through high scores. In order to accomplish this, I will have to build a central server, making use of a database that will hold accounts and high scores etc.

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 tile map and add sprites and sound effects to mimic the quality of the original game. The library is also makes moving and logging interactions between multiple objects across the screen many times a second. These features make ‘Pygame’ a very capable alternative to another (more basic) python GUI framework like ‘TkInter’.

## 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 pixels 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, 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 timings of these modes vary as each level progresses and from level to level.

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

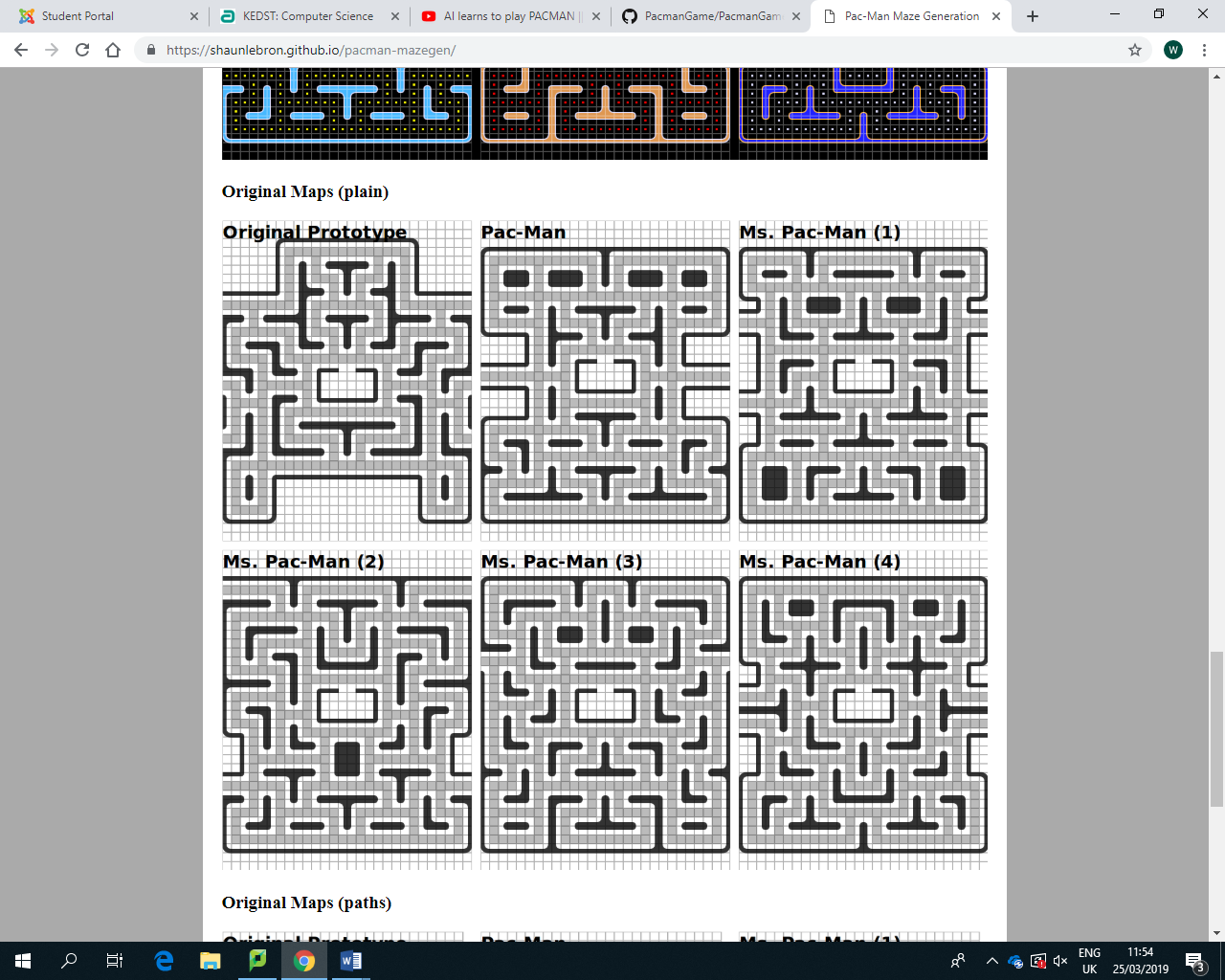
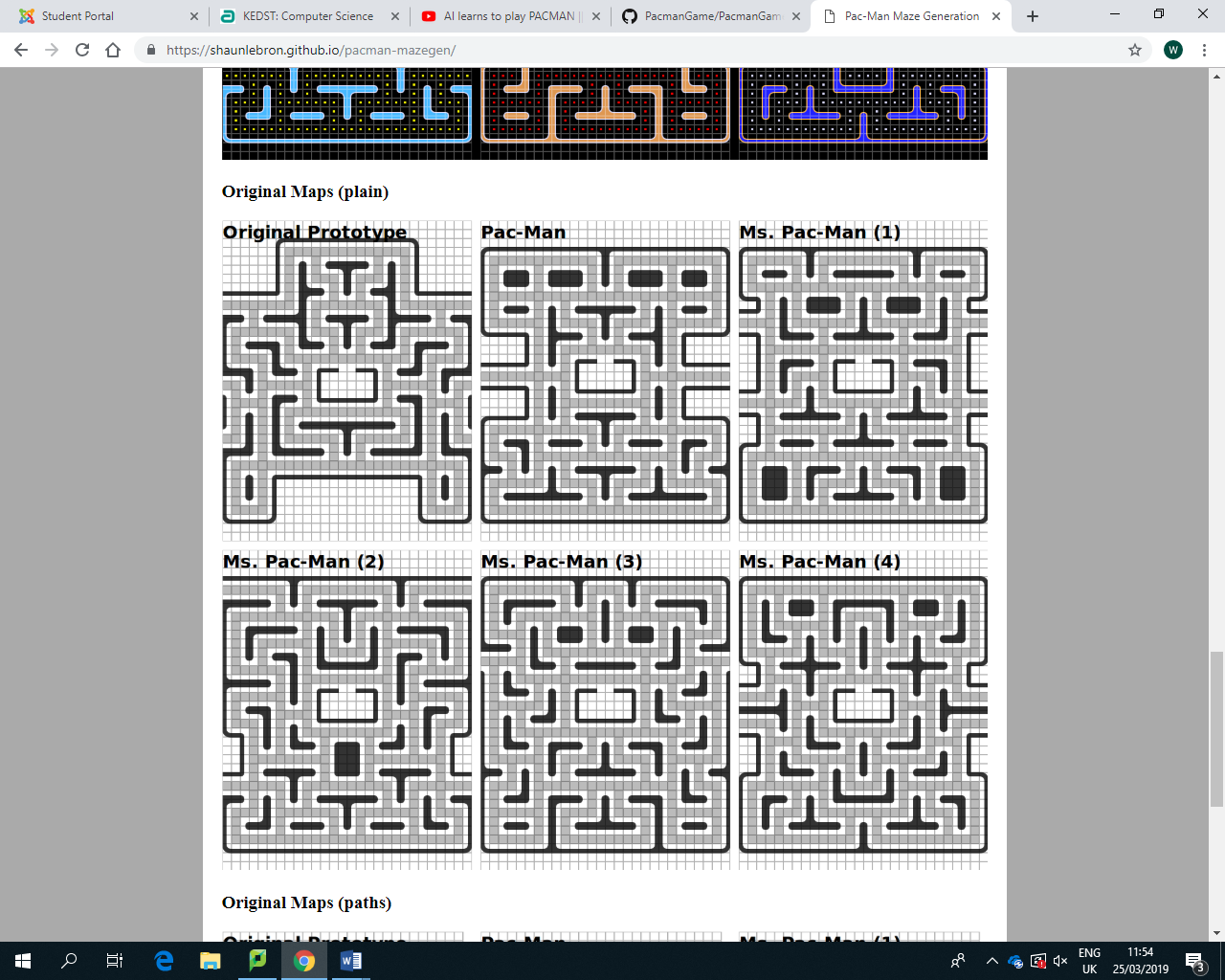
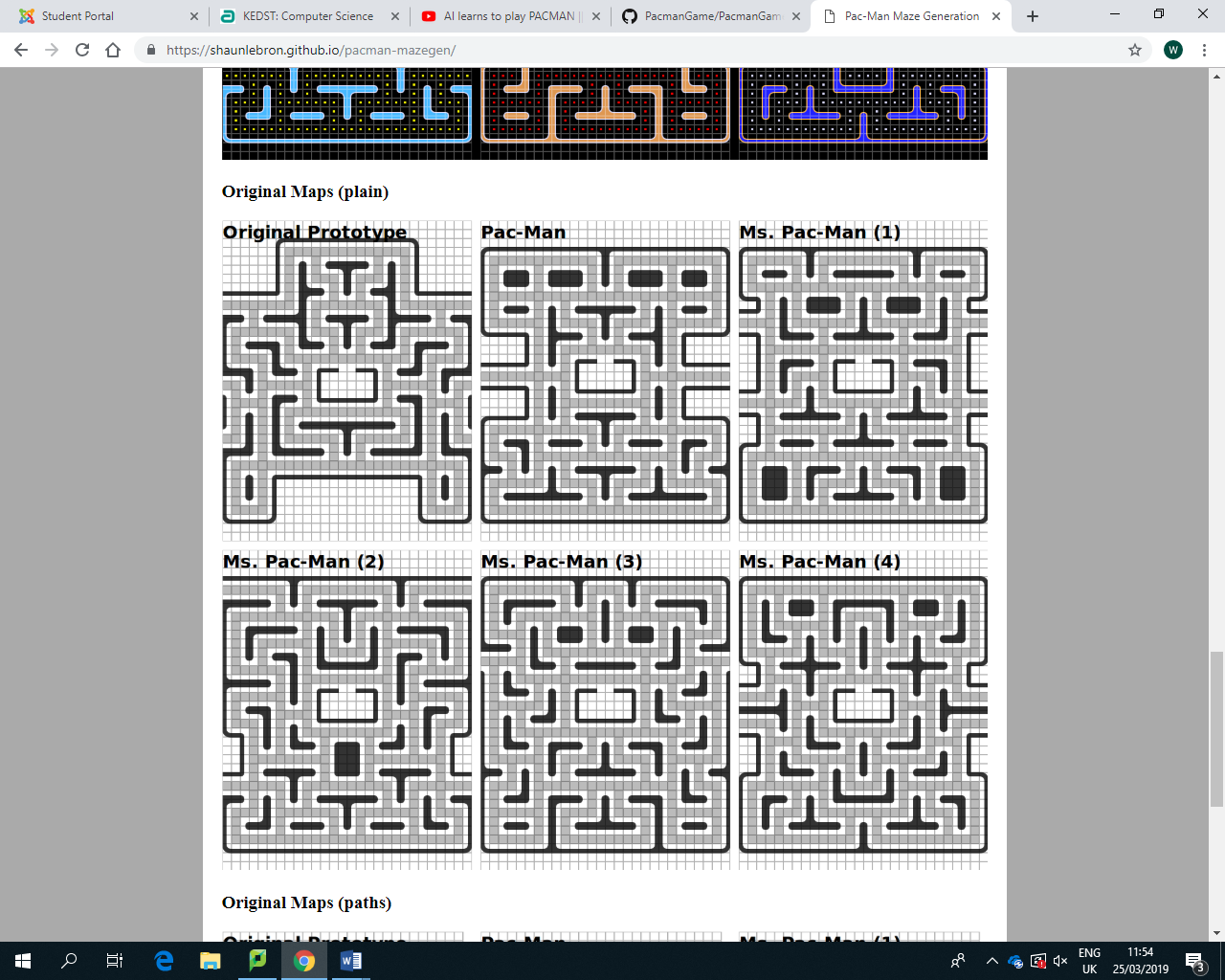
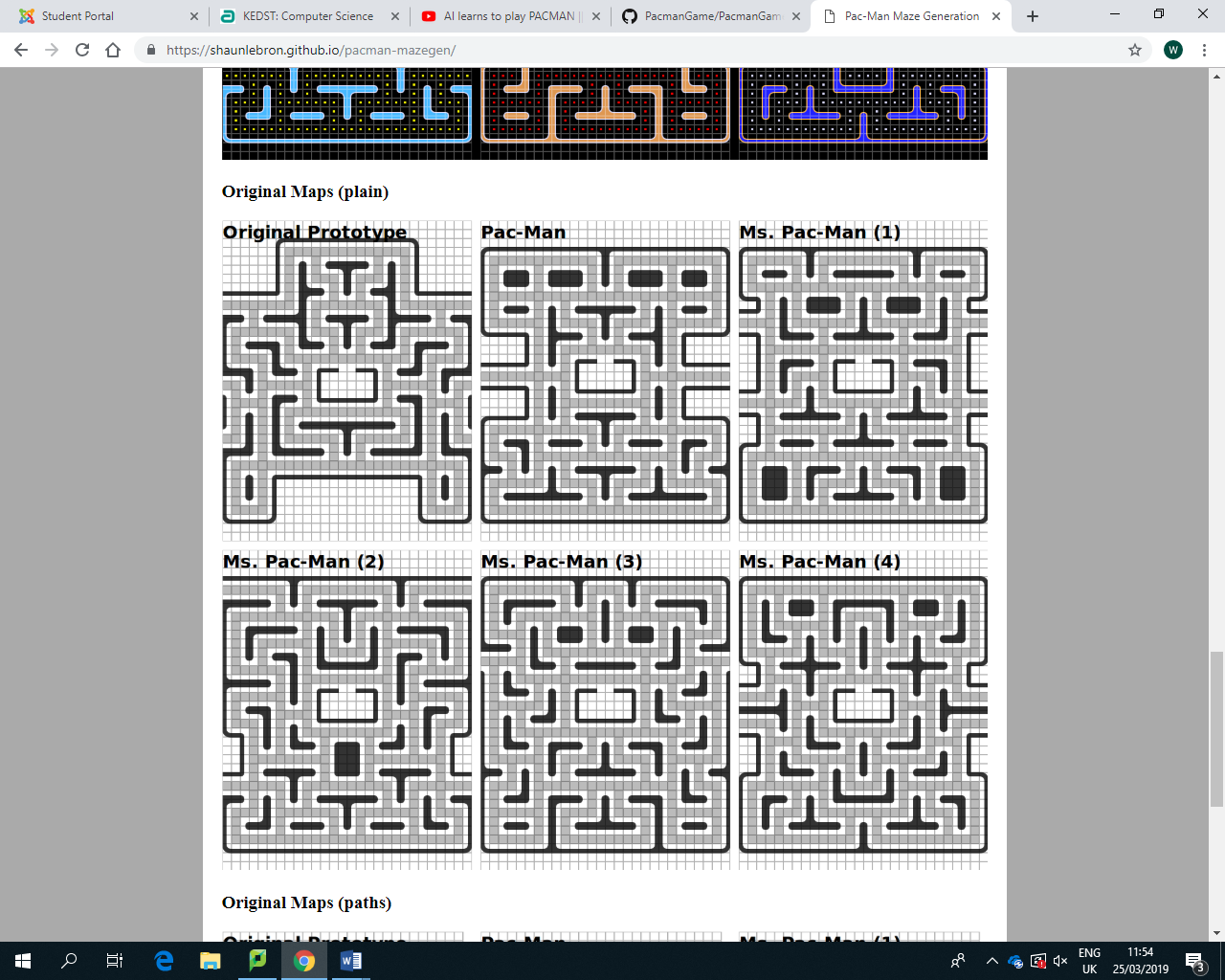
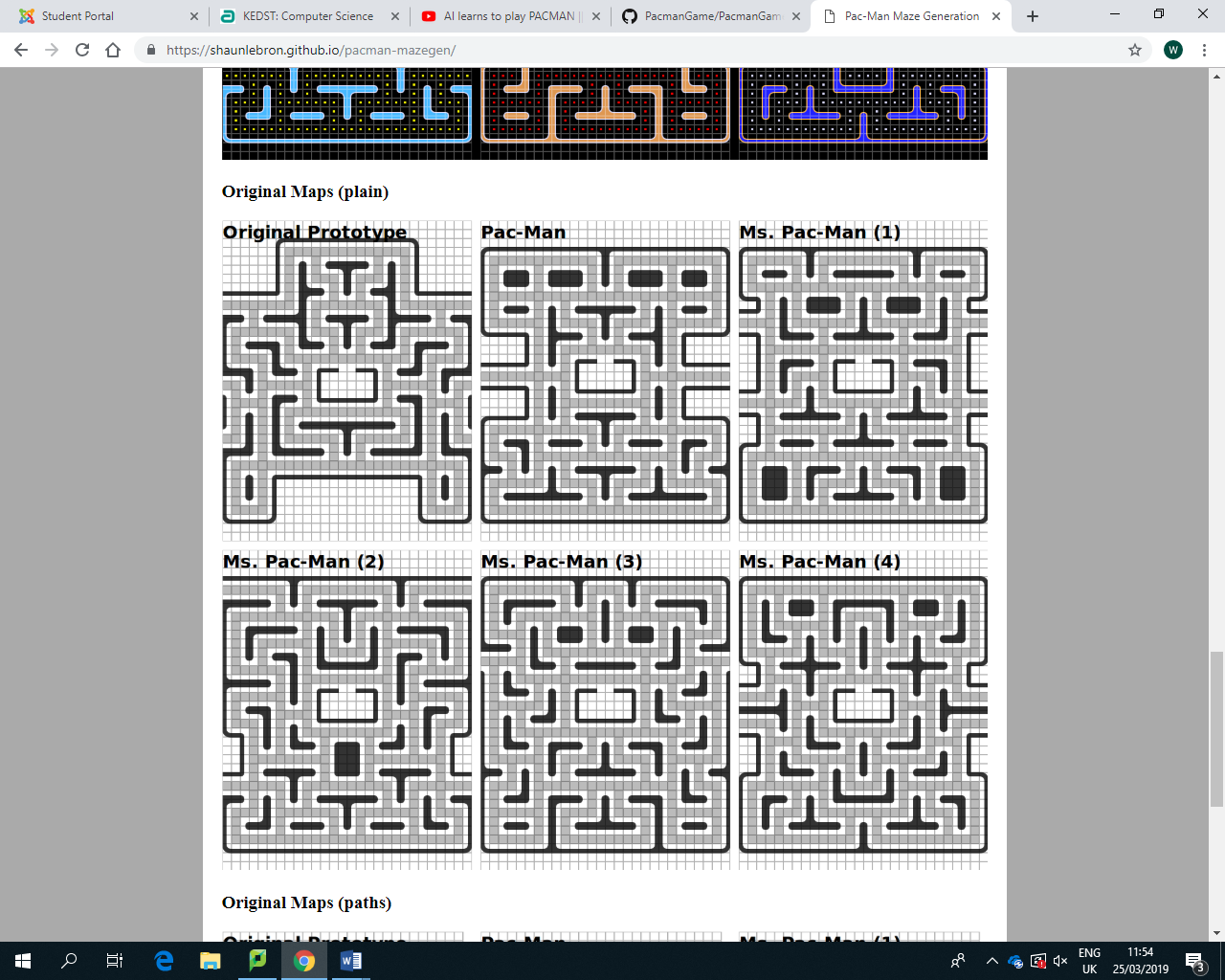
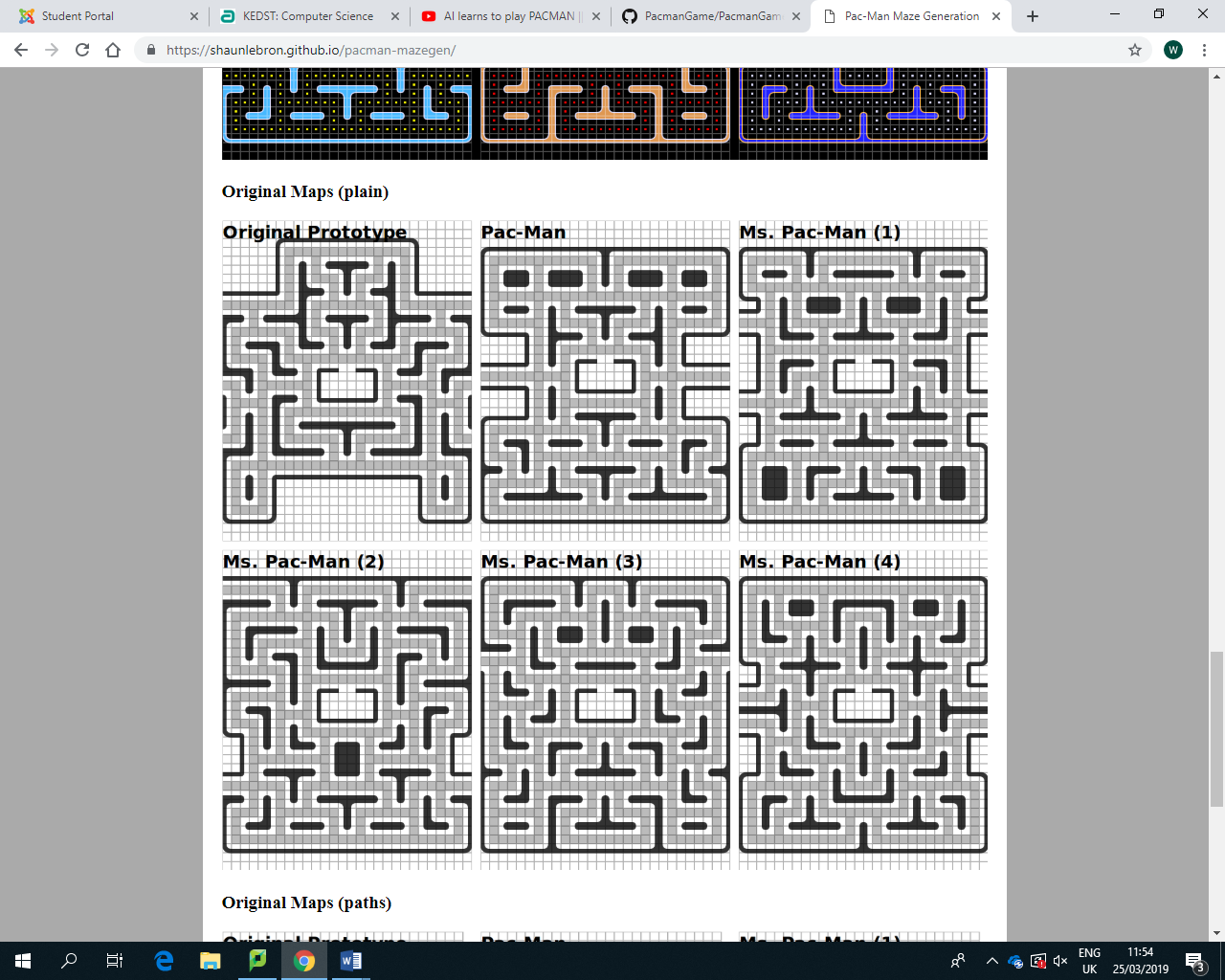
In Pac-Man the mazes are designed and hard-coded into the game with the following set of rules:

* 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.

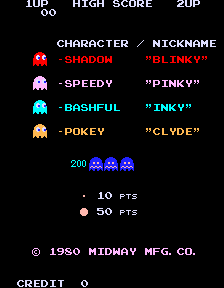
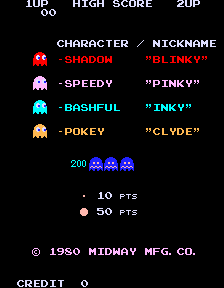
After looking into machine learning, it appears it is either extremely difficult (without the use of dedicated libraries) or too easy (with the addition of libraries) so instead I will create my own simplified version of the maze generation algorithm in which random pieces are chosen and added to a half 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 will still give a random maze every time, but there will be a smaller total number of possible mazes.



### Path finding

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 path finding 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. 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. In fact, this method is essential as it is the quickest function that will work, ensuring A\* is as efficient as possible.

### 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 higher quality, but also with slightly different aesthetics to the originals. The game mode is also different on Pac-Man Vs.

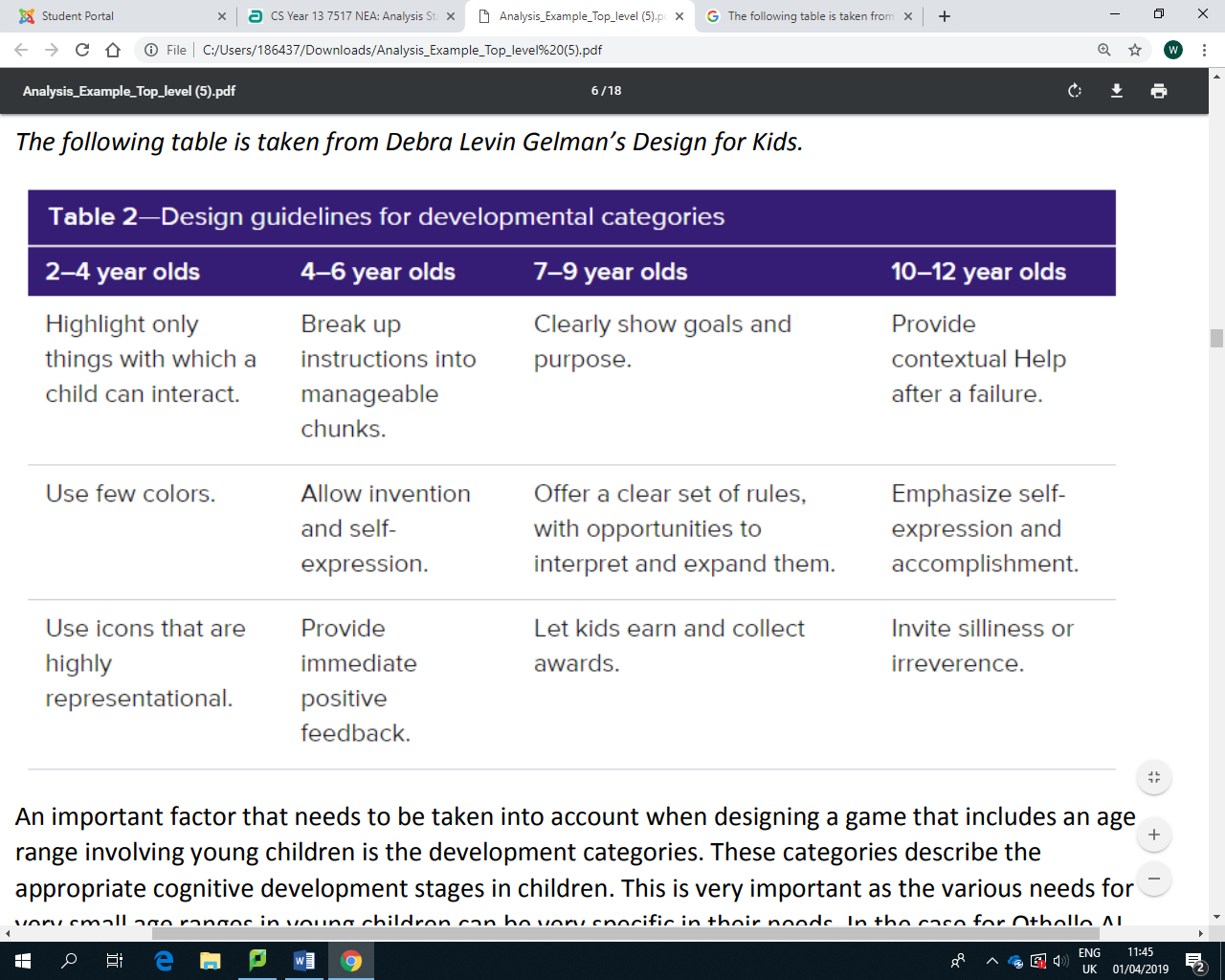
The winner of the game is the first to reach a predetermined number of points. You earn points as Pac-Man by eating ghosts, fruit, pellets and power pellets. Once Pac-Man is caught by the ghosts, whichever ghost caught Pac-Man will become Pac-Man in the following round. In order to balance the game, 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 rather than local 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 incorporated into the first game (with broken up instructions)
* A mode whereby the user can design their own maps (allowing self-expression)
* 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. Even still a high score system can be very easily implemented using a relational database that will encourage healthy competition in these higher skill/age bands.

## Objectives

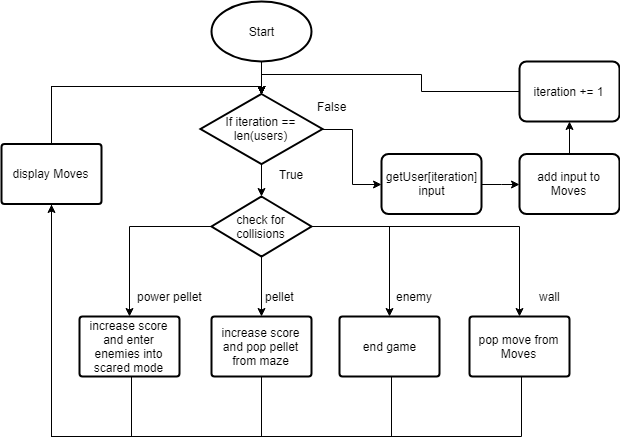
In this section I will outline and further expand on the main points and objectives I have for my project, starting with how the role of each of my pages, and then moving on to gameplay and game modes.

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. - 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’. I feel there should also be an extra toggle button, which will allow the user to remain signed in if they would like.
   2. - There should be a sign-up button that changes to a login button when the user enters something into the username field.
3. - Sign-up Screen
   1. - Have three input boxes that can be highlighted by either by clicking or pressing tab. These will have names: ‘Username’, ‘Password’, ‘Confirm Password’ which displays inside the box until the user enters something instead.
   2. - 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; contain a number.
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.
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 an image of one of the 36 different tile pieces, 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 of 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\* path finding 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 includer 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 of the pellets.
      2. - The second level should introduce the ghost Clyde and explain his behavior (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 easily avoid Blinky.
      6. - Level six will add the final ghost ‘Inky’, who must be accompanied by Blinky due to his chase mechanic.
      7. - Level seven (the final level). This level will be most similar to the Classic mode, but with a different (random) maze every time you complete it.
   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.
7. - Classic Mode
   1. - 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).
8. - 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 for 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 user’s 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 a user
      4. - 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.
      5. - 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 are no lives nor is there a high score text at the top of the screen. The only indicator is 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 user’s 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. - Creator
   1. – Something happens
3. - 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.
      2. -
   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 client has processed the maze it sends it off 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 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)