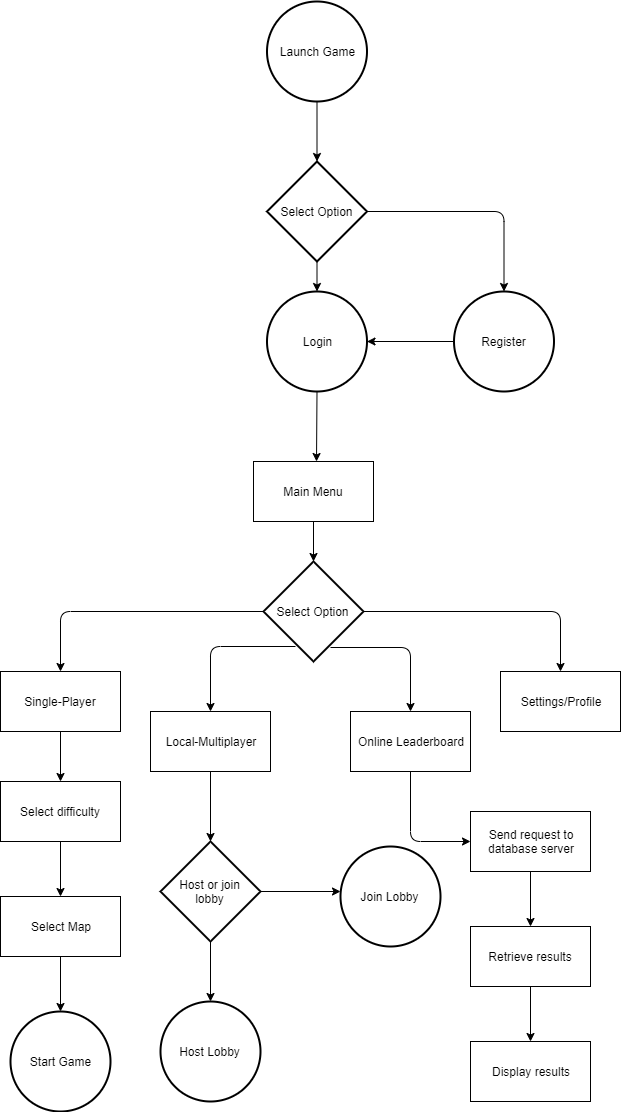
**Design**

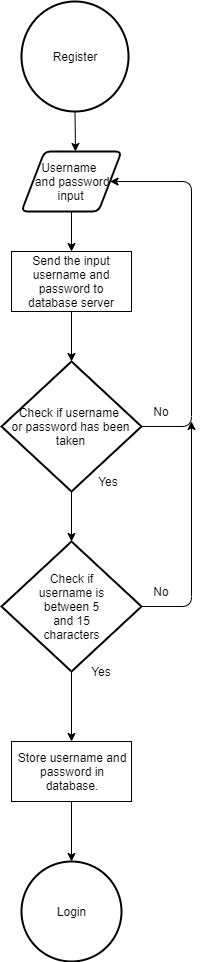
High level Overview

These flowcharts describe the overview of how my project will work. The flowcharts show the process in how users access different parts of the game and what happens when users click on certain buttons.

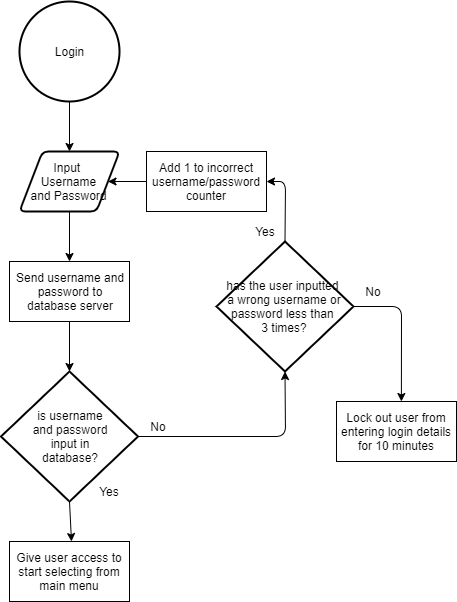
**Main Menu**



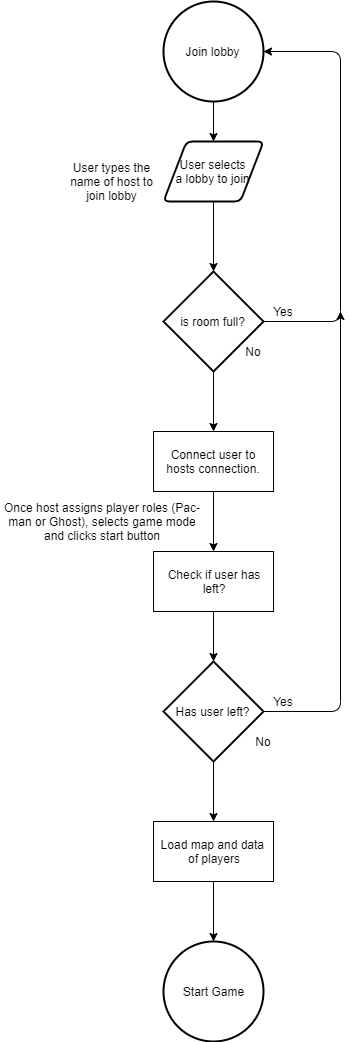
**Register**



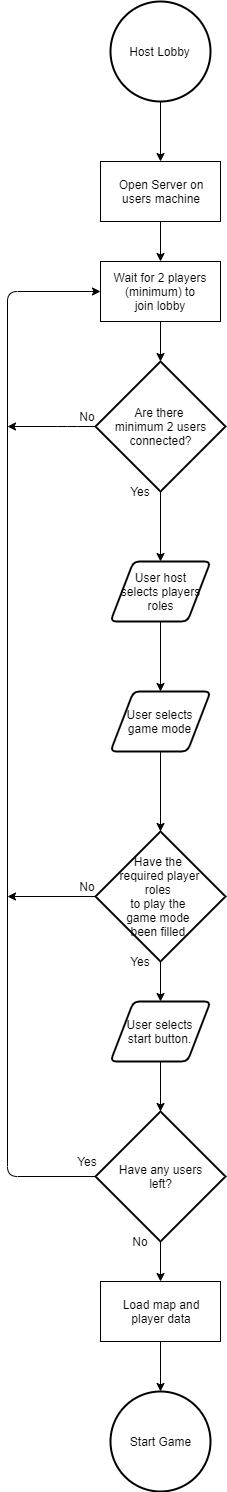
**Login**



**Join Lobby**



**Host Lobby**



Description of modular systems

When the game is launched, the user is shown a login/register screen. This is a diagram of the different screens that will be included:

Lobbies

Selected Lobby screen

Settings

Local-Multiplayer

Single-Player

Leaderboard

Register

Main Menu

Username, High scores and game mode displayed.

Login

The leaderboard will only show the local multiplayer high scores for each game mode individually. However, in the settings users will be able to see their single-player high score, and also in game.

When the user launches the game, the attribute “state”, of Class Board is initialised as “loginOrRegister”. Once the user registers then clicks login to enter the main menu, the state changes to “menu”. The user is then able to see, Single-Player, Local-Multiplayer, Leaderboard and Settings options on their screen. Once the user enters a game and it has started, the user state changes to “play”, and once the game ends the user is returned to the main menu and the state changes back to “menu”.

* This is an IPSO table for my project.

|  |  |  |  |
| --- | --- | --- | --- |
| **Inputs** | **Processes** | **Storage** | **Outputs** |
| -player input (keyboard presses and mouse clicks)  -user details, such as username, password | -increasing player level after every game by using calculations.  -collision detection between players and walls. This is done by distinguishing 1’s and 0’s. If the tile represents 0 it is a wall, if it represents 1, it is a free cell.  -calculate players new position in map using pixel position, based on the user’s keyboard input.  -calculate score and decide if it is higher than the current high score.  -calculate if user is in current position to pick up dots or power-ups.  -calculate if a player won or lost. | Database tables:  Users  GameMode  UserGameMode  Local storage:  Preferred Controls  Maps (if I create more than 1 map) | -Sorted list of users, high scores and game mode. (Displayed in Leaderboard section).  -Output game map to the screen.  -Output Pacman and ghost sprites to screen.  -Output power-up sprites to screen. |

Data Dictionary

I will be storing data long term and retrieving data from a MySQL database when queried. Here are the tables in my database in which I will be storing and retrieving data:

**Users table:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Field name** | **Field type** | **Field size** | **Purpose** | **Example data** | **Validation** |
| playerID | INT | 5 | Assigning a unique identifier to each player which acts as a Primary Key. Allows me to normalise database | 00001 | Is numeric |
| Username | VARCHAR | 20 | Stores username of each player | FotiosD | Must be between 5 and 15 characters. |
| Password | VARCHAR |  | Stores player’s password | fotios\_dimi1 | Must contain an integer and letters. Must have a minimum length of 5 and maximum length of 15 |
| player\_level | FLOAT |  | Stores player’s level. |  | Is numeric |
| highscore | INT |  | Stores the player’s highest score on single player. |  | Is numeric |

**GameMode table:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Field name** | **Field type** | **Field size** | **Purpose** | **Example data** | **Validation** |
| gameID | INT | 3 | Assigns a unique identifier to each game mode created by the project developer (myself). This acts as a Primary Key. | 001 | Is numerical |
| game\_mode | VARCHAR | 20 | Stores the name of each game mode created. | Regular pacman | n/a |

**UserGameMode:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Field name** | **Field type** | **Field size** | **Purpose** | **Example data** | **Validation** |
| playerID | INT | 5 | Stores the ID of a player related to a specific game mode. Foreign Key | 00001 | Is numeric |
| gameID | INT | 3 | Stores the ID of the specific game mode. Foreign Key | 001 | Is numeric |
| player\_wins | INT |  | Stores the totals wins of a player in a specific game mode | 10 | Is numeric |
| player\_losses | INT |  | Stores the totals losses of a player in a specific game mode | 5 | Is numeric |

The table UserGameMode allows the user to see their totals wins and losses per game mode.

Data structures

* Variables most often used in my game:
  + pos: tuple list which includes x and y position
  + ghost\_players: All players which are ghosts (only for multiplayer).
  + pacman\_players: All players which are Pacman.
  + time: This will only be used for the second game mode.
  + Grid: The array that represents my map.
* I will be using a queue for respawning power-ups onto the map. Initially when the game starts various random power-ups will be spawned. In addition, there will be a full queue of power-ups when the game starts. As players pick up the power-ups they are added onto the queue and the first power-up is popped off and spawned onto the map. Therefore, a queue is perfect for designing this. Furthermore, once the power-up is popped off there is a time delay before it spawns.
* Data structures used in Python:
  + SQL
  + Timer
  + Queues
  + Tuple lists
* Data structures in Pygame:
  + Surfarray (Allows efficient per pixel effects)
  + display (Controls display window and screen)
  + draw (Draws simple shapes)
  + mixer (Loads and plays sounds)
  + time (manages time and framerate)
  + image (Loads, saves and transfers surfaces)
  + key (manages the keyboard device and controls inputs)
  + event (Manages the incoming events from input devices and the windowing platform)

Validation

I will be validating various user inputs to ensure guidelines are met so that the game will not crash or break.

|  |  |  |  |
| --- | --- | --- | --- |
| **Field name** | **Validation checks** | **Description** | **Error message** |
| Username | Min length is 5 characters.  Max length is 15 characters. | Make sure that the username length meets the requirement. | “Username must be a length between 5 and 15 characters” |
| Password | Min length is 5 characters.  Max length is 15 characters.  Must contain numerical and alphabetical characters. | Make sure that the password length meets the requirement.  Make sure that the password contains numerical and alphabetical characters. | “Password must bet within 5 and 15 characters”  “Make sure your password contains numerical and alphabetical characters” |

Database design

“Users” table stores information on the player, such as Username, Password, player level and high score. This is very useful as it allows me to develop a leaderboard system so players can compete with each other, by uniquely identifying each user. It also allows more than one user to access their profile on the same PC, and it allows users to access their profile on multiple devices.

“GameMode” table stores the information on each game mode. If future game modes are created for my game, they can be easily added to the database. This is also useful for displaying which game mode users got a high score in. It allows me to store each user wins and losses for each individual game mode.

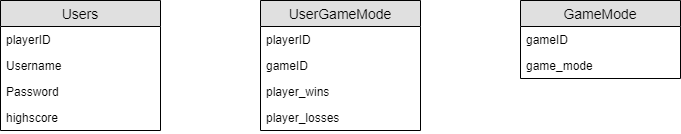
Entity relationship diagram:



Each user can play many gamemodes, and a gamemode can have (contains) many users/many players can play one game mode.

To normalise this a join table called “UserGameMode” is introduced between the two tables. This table contains a composite key (from two foreign keys), consisting of playerID and gameID.

The new diagram is represented like this:



**Normalisation**

**1st Normal Form**

If I did not include the join table (“UserGameMode”), I would have to expand the ”Users” table, so it looks like this: Users (playerID, Username, Password, gamemode1\_wins, gamemode1\_losses, gamemode2\_wins, gamemode2\_losses, highscore)

The issue with this, is that some users may not want to play local multiplayer, so there will be a lot of wasted space and memory in the database. In addition, if my game obtains additional game modes, I will have to add more columns to the Users table. To fix this issue, we need to get rid of the repeating attributes gamemode\_wins and gamemode\_losses. Moreover, querying a specific players wins and losses in a certain game mode will be more difficult.

**2nd/3rd Normal Form**

A table is in second normal form if it is in first normal form and also contains no partial dependencies. A table is in third normal form if it is in second normal form and all of the attributes depend only on the whole primary key and nothing but the whole primary key.

My table meets these requirements as player\_wins and player\_wins, both depend on nothing but the whole primary key. In addition, there are no partial dependencies because only the join table “UserGameMode” contains a composite primary key and no other table does.

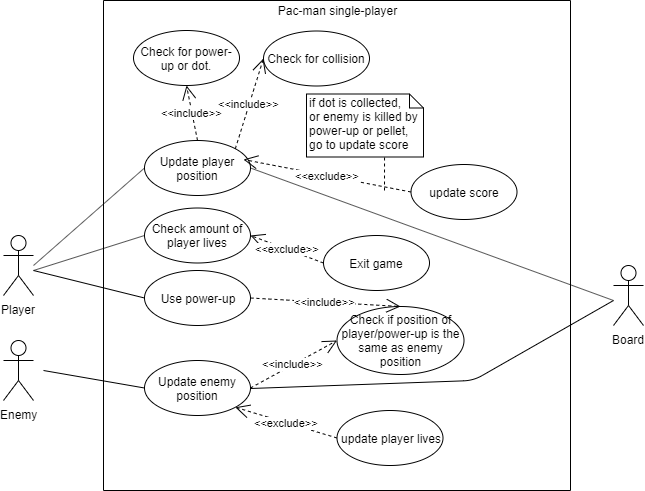
There are many players on 1 board and many enemies on 1 board

Problem statement

* My game is based around Pac-man. Pac-man is a game where a single user tries to obtain as many pellets/dots around a maze as possible while avoiding the enemy ghosts who try to hunt Pac-man down.
* My implementation can be played via single-player or multi-player (locally)
* My implementation involves implementing, not only my own version of Pacman with different algorithms, but also a game mode where players have to avoid lava tiles (dying) and try obtain a certain amount of points to win. This game mode has not been seen before.
* My project introduces a never before seen game mode in Pac-Man. Local multiplayer contains the second game mode and single-player covers the first game mode.

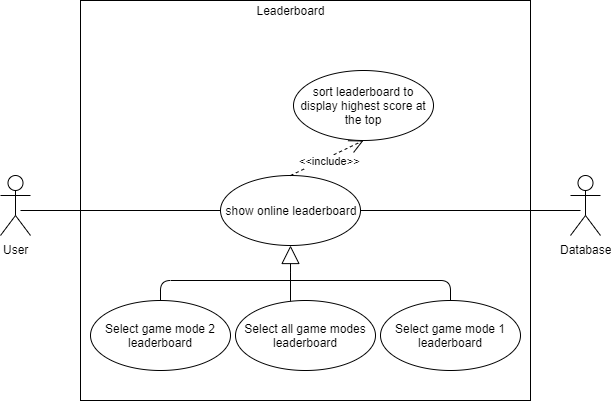
Use Case diagrams

**Single-player gameplay**

****

* This diagram shows:
  + Enemy and player movement
  + Runs all checks for player to see if new position is valid and if there are any items or enemies in that new position.
  + Game values update accordingly (e.g. score or player lives)
  + Shows implementation of power-up usage

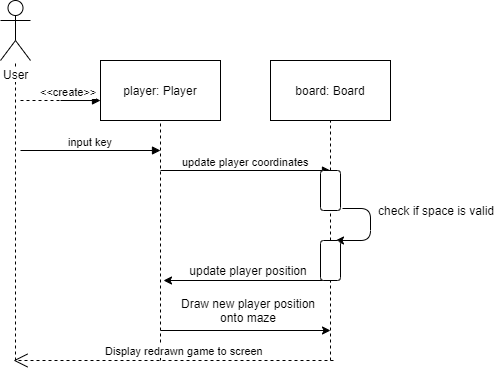
**Leaderboard**



* This diagram shows:
  + User access to online leaderboard
  + Leaderboard options/selection
  + Database query and leaderboard sorting.

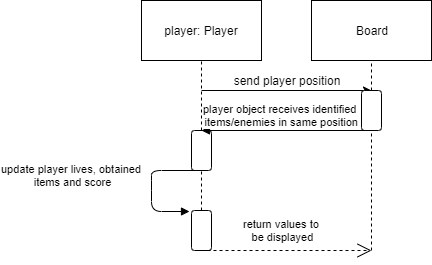
Sequence diagrams

**Player movement**



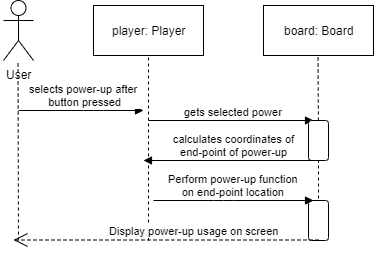
* This diagram shows what happens when a user presses a movement key to move their character. Once the user presses the key, the object
* Player retrieves those coordinates and updates them.
* The object board then checks if that position is valid and the player is allowed to move into that space, if not then the player stays in the same position.
* However, if the new position is valid the board will place the new position as the current position now.
* The object player, then retrieves the position and draws the new player position onto the screen. The board now updates the screen

**Checking items (power-ups or dots) or enemy in same position as player**

****

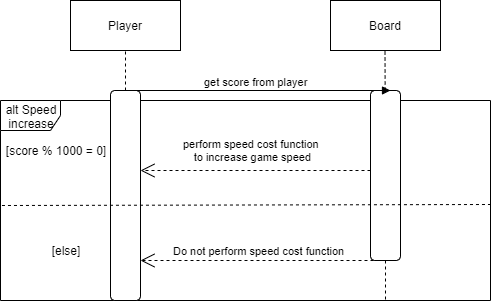
* This diagram shows how my program checks if an item or enemy is in the same location as a player.
* Class Board retrieves player position from the object player.
* It then checks that position for enemies (from an enemy position list) and checks if any power-ups or dots are currently in that location.
* If an enemy is in that position, object player updates the player lives. If a power-up is in that position, the power-up gets added to the players inventory. If a dot is in that position the score gets updates.
* Class Board then retrieves these values and displays to screen.

**Shows how power-up is used**



* This diagram shows how my program determines and uses the players obtained power-up
* The user first presses the key to user a power-up.
* The object player then determines what power-up is in the players inventory and passes it on to the object board.
* Object board then calculates at which coordinates (in the map) the power-up will act upon (this can vary between power-ups)
* The object player then performs the function of that power-up on that location. The board then retrieves that location and updates the display onto the screen.

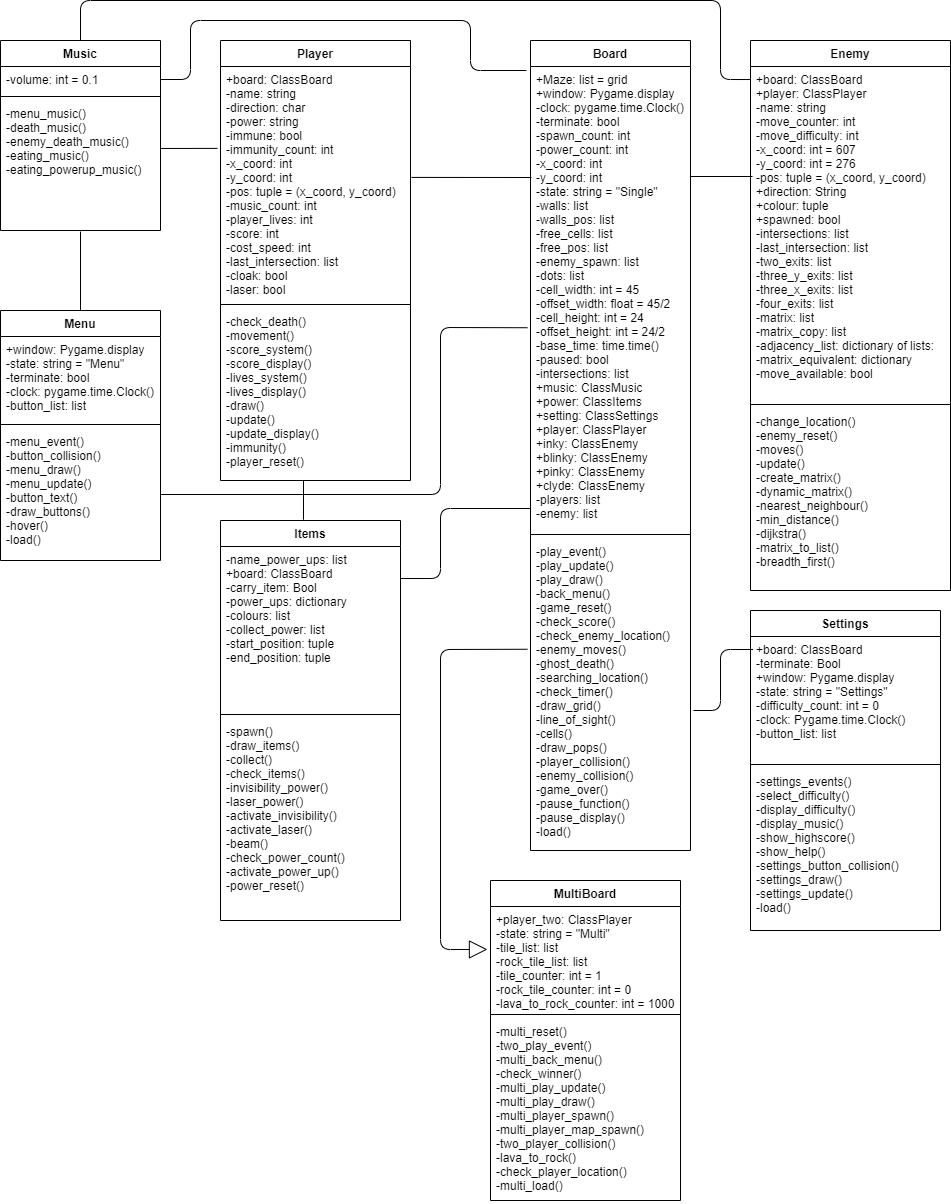
**Gameplay pace (speed gameplay increase)**



* This diagram shows how the speed of the game increases over time by using a cost function
* The Class Board retrieves the score from the Class Player.
* If the score hits a multiple of 1000, the cost function is performed to increase the speed of all characters in the game. If the score has not hit a multiple of 1000, score function is not performed.

Class diagrams

**System overview**

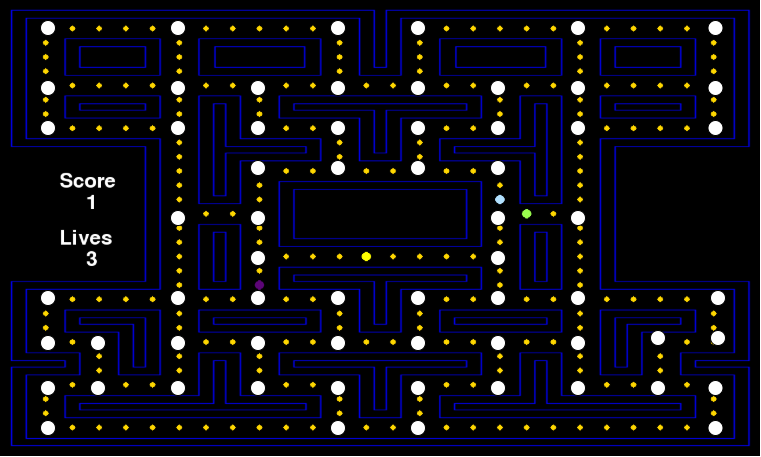


AI algorithms

* Here I will be giving an insight into the algorithms I am using for each of the four enemies. Each enemy will have a calculated distance from their position to the players position. The distance will be calculated using Pythagoras. It will take the x and y distance (in pixels) from the enemy position to the users’ position and find the hypotenuse from those values.
* The four enemies are called, Pinky, Blinky, Inky and Clyde.
* The algorithms I will be implementing include:
  + Dijkstra
  + Breadth-First Search
  + Line of sight/ray casting
  + Dynamic Matrix
  + Enemy Detection
  + Also other small algorithms will be included that are talked about in the Analysis, Design, and Technical Solution.

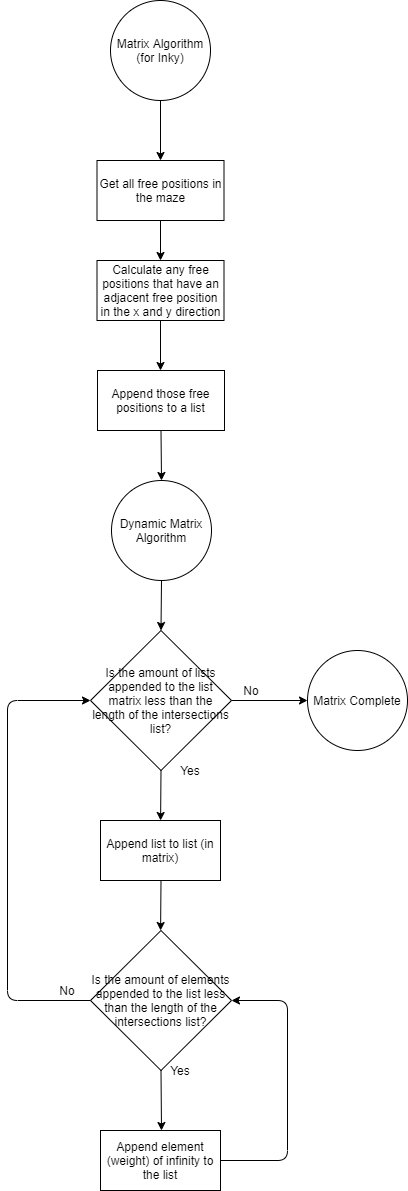
**Inky Algorithm**

The ghost Inky will use a Dijkstra algorithm to find Pacman (the player). Dijkstra works by finding the optimal shortest-path algorithm by calculating the total weights (distances) between each node. The image below shows where each node would be (the white dots). The nodes are placed at intersections to allow the enemy to change direction at these points. Between the nodes, there are vertices which are weighted based on the distance (using coordinates)

****

An adjacency matrix is used to store all vertices between two nodes. An adjacency matrix is a list of lists, in which every list initially stores values of infinity before weights are calculated. Each index also has a unique set of coordinates, which are stored as a key-value pair in an intersection dictionary. The dictionary is used to manoeuvre Pinky on the maze once Dijkstra calculates the shortest path.

All the ghosts besides Inky will have their last intersection stored. For all the adjacent intersections to the ghosts last intersection, the vertices’ weights will be changed to a very high number or infinity. By doing this, the ghosts will not follow each other when path finding for Pacman, instead they will try reroute and trap Pacman. Furthermore, it stops the ghost Inky from following other ghosts and Inky will try trap Pacman as he won’t take the same route as the other ghosts. To do this, I need to create a dynamic matrix algorithm.

****

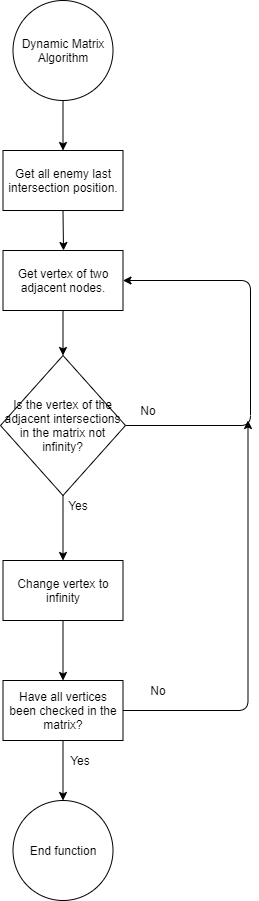
The values on the first row and column (0, 1, 2…)

Here is where the weights of the vertices will be stored. If there is a weight other than infinity there is an adjacent node. If weight is inf (infinity), the two nodes are not adjacent.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 | 1 | 2 |
| 0 |  |  |  |
| 1 |  |  |  |
| 2 |  |  |  |

**Dynamic Matrix Algorithm**

This algorithm is used to constantly change the matrix depending on the other ghosts’ positions, besides Inky. The matrix is only change for Inky as Pinky does not use weights when calculating the path to Pacman using Breadth-First Search. It will only work for Dijkstra.



**Pinky Algorithm**

The ghost Pinky will use breadth first search to find Pacman. It will start at the tree root (starting node) and explores all of the neighbour nodes at the present depth before moving on to the nodes at the next depth level. Like the nodes I used for Dijkstra, I will be using the same nodes for Breadth-First search. Those nodes will be the intersections which I displayed previously. I will be representing the tree-graph by converting the adjacency matrix I created for the previous algorithms into an adjacency list. The adjacency list created stores each node and what nodes are adjacent to it; however, the adjacency list does not hold the weights of the vertices, unlike the adjacency matrix. This means that the path is not weighted, so Breadth-First Search won’t necessarily find the shortest path, but instead Pinky will use the first possible path found my Breadth-First Search.

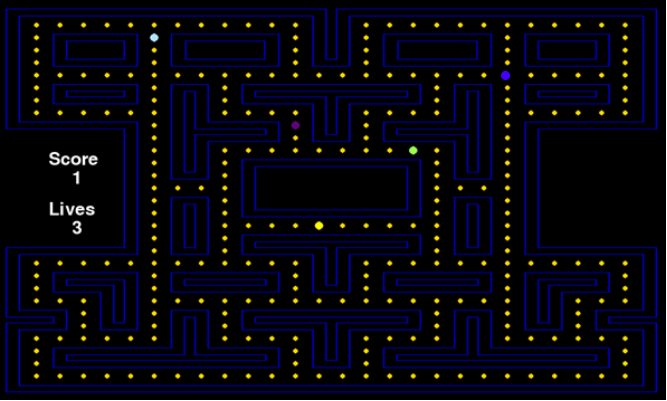
**Blinky Algorithm**

The ghost Blinky will move randomly throughout the whole game until he sees Pacman in his line of sight. Once he sees Pacman, Blinky will move straight towards Pacman (using ray casting/line of sight), until Pacman leaves his line of sight.

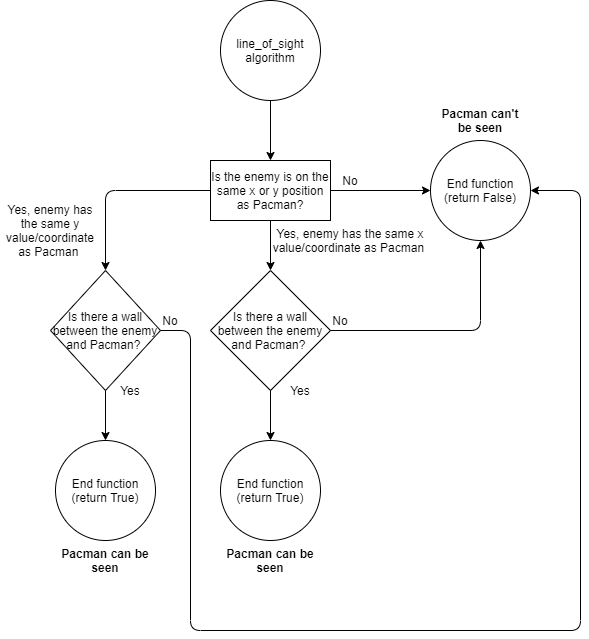
**Clyde Algorithm**

The ghost Clyde, will move intersection to intersection without try to search for Pacman (like Blinky). Clyde also acts as a blind ghost, therefore, even if Pacman is in Clyde’s line of sight, Clyde will not spot Pacman.

**Line of Sight Algorithm (Used for Inky, Pinky and Blinky):**

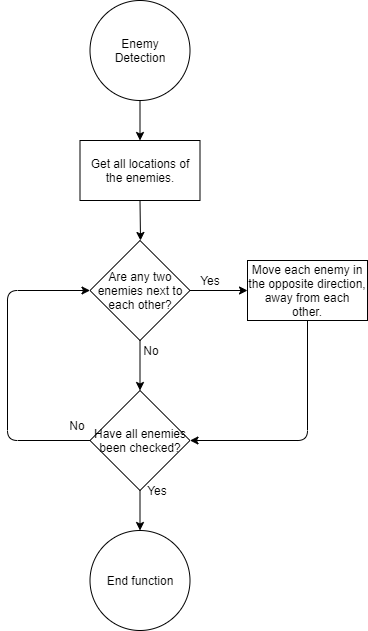


This algorithm is a form of ray casting. The destination for every searching algorithm (Dijkstra and Breadth-First Search) is the last intersection (node) Pacman was located at. If either Inky or Pinky reach this last intersection, they will stop or an error would occur, therefore a line of sight algorithm would be needed. Whenever Inky or Pinky reach the last intersection, they will then move towards Pacman, who will always be in their line of sight if they have reached the last intersection. Blinky on the other hand does not use a searching algorithm, and just uses line of sight. If a user just stumbles into the line of sight of Blinky, Blinky will move straight towards Pacman until Pacman moves away from its line of sight.



**Enemy Detection Algorithm**

This algorithm is used so enemies can detect each other and avoid moving over each other (overlapping). This algorithm checks other enemies’ locations and if the ghosts are next to each other, they will move one tile away from each other in opposite directions.



Local-Multiplayer/Networking

Users will have an option to join or host a lobby. If a user hosts a lobby, they will open a server and port on their computer, and all other users that join will send and receive messages through the port on that server. The server will send messages to all users connected, and the users will receive and send messages back.

* Messages being sent to server:
  + X and Y coordinates of player
  + When a user collects a power-up
  + When player uses a power-up
* Messages being sent back to all clients:
  + Co-ordinates of all players gets sent to all clients so the maze (board) can be updated on the client side, as the maze will be stored individually for every client. The maze will not be server side.
  + Server will randomize where power-ups will spawn, and sends the locations to all clients.
  + When a player collects a power-up, so it is updated on everyone’s board
  + Location of where power-ups are used.

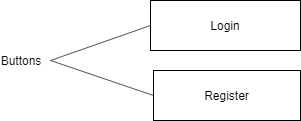
UI Design/HCI rationale

My project aims to cater to all target audiences, younger and older audiences. To make it accessible to all users I need create a UI that is easy to read and understand. To create a very accessible UI, I need to create a clear layout of the buttons and make sure they are organised on the screen in a simple, clear way.

The background of the screen (before user enters a game) will be colourful and have a strong contrast to colour of the buttons, so the buttons stand out visibly to all users. The background will have dark colours, while the buttons will project bright visible colours.

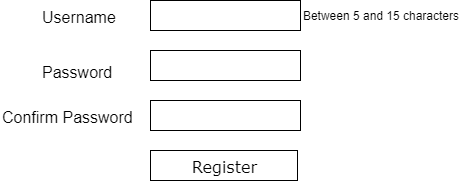
According to this website “<https://www.vistaprint.com/hub/design-decoded-top-12-easy-read-fonts?couponAutoload=1&GP=11%2f09%2f2019+13%3a07%3a16&GPS=5527287966&GNF=0>”, Open Sans is one of the most readable fonts for user interfaces, so I will be using it when creating text for the screen.

**Start Menu**

****This is the first screen that will appear when users enter the game. Both buttons Login and Register only send users to different screens.

The login button sends the user to the login screen and the register button sends the user to the register screen.

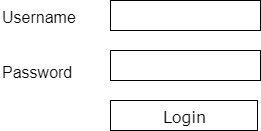
**Register Menu**

If users click “Register” from the Start Menu, this screen will appear. Here users type in a Username, Password and a confirmation of that same Password.

When the button Register is clicked, the program determines if “Password” and “Confirm Password”. If they are not an error message is sent, saying “Passwords do not match”. If the passwords match, the program will then check if the Username is between 5 and 15 characters. If everything is successful up to now, the program will send a query to the database to check if both the username and password have not been taken.

If everything is successful, the User will be taken to the Login Menu.

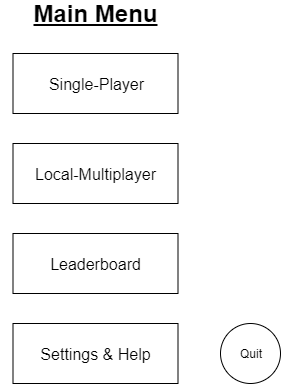
**Login Menu**

If users click “Login” from the Start Menu, this screen will appear. Here users type in their Username and Password to access the Main Menu.

When the button Login is clicked, a query is sent to the database to check if the Username and Password are correct. If they are not, the user will receive an error message saying “type login details again”. If login details have been typed in wrong 3 times, the user will receive a ten-minute time out.

If login is successful, the User will be shown the Main Menu screen.

**Main Menu**

Once the User Logs in successfully, this screen will appear. There will be an arcade-style background. I will aim to create an animated background. Furthermore, four buttons will appear o Once the User Logs in successfully, this screen will appear. There will be an arcade-style background. I will aim to create an animated background. Furthermore, five buttons will appear on the screen.

The Single-Player button sends the User to the Start-Game Menu, where the user can select the settings (such as difficulty) for the game about to be played by the user.

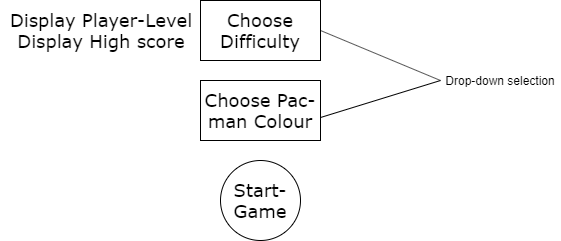
The Local-Multiplayer button sends the User to the Lobby menu where users can select to either Join or Host a lobby. After this, the users will enter the gamemode chosen and the game will end with a decided winner.

The Leaderboard button sends the User to the Leaderboard Menu where the User can see the online rankings of all Users displayed (from the database).

The Settings & Help button sends the User to the settings where they can adjust the volume for the game, see their high score for single-player and their player-level, and more. Players unfamiliar with the game can also receive help, which will instruct the player how to play the game.

The quit button has the most basic function, where the user exits the program once it has been pressed.

**Start-Game Menu**

****

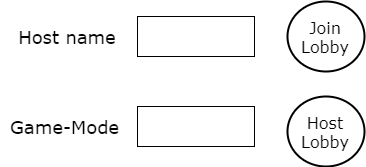
Once the User selects “Single-Player” they will be directed to this screen/menu. Both “Choose Difficulty” and “Choose Pac-man Colour” are drop-down selections.

The difficulty changes the cost function which affects how much the speed of the Ghosts/enemies and Pac-man increase throughout the game. The User may also change the colour of their Pac-man. Users can unlock cool colours and colour schemes by levelling-up.

The Start-Game Menu also displays the Users current high score and player level.

Once the User wants to start the game, they can select the button “Start-Game”. It will then take the user to the actual game where the gameplay starts.

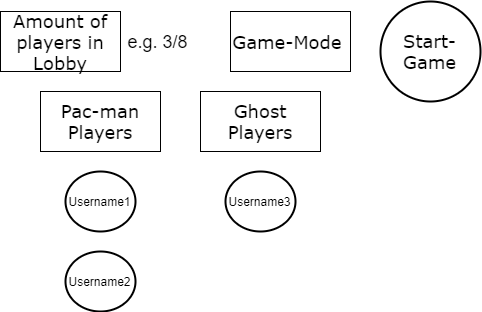
**Lobby Menu**

****

To join a lobby, the user must type in the hosts username then select the button “Join Lobby”. A request will be sent to join that lobby. If a lobby is not found with the username typed in, the game will send a message back saying “Lobby not found”. If lobby is found, user will be sent to the Multiplayer-Game Lobby.

To Host a Lobby, the user needs to select the game-mode, which will be a drop-down selection, and then select “Host Lobby”. The lobby name will be the user’s username. The user will now be sent to the Multiplayer-Game Lobby. Other users can now join this lobby.

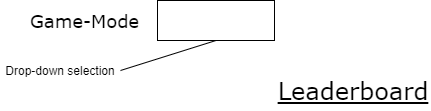
**Multiplayer-Game Lobby**

****

Whenever Users enter the lobby, the Host selects what role they play. When there are a sufficient number of players in a role, the Host may start the game.

Depending on what game-mode, there is a minimum number of players in each role. For game-mode 3, there is a minimum of 1 Pac-man and 1 Ghost. For game-mode 2, there is a minimum of 2 Pac-man players and 0 ghost players. Gamemode 3 however, will most likely not be implemented.

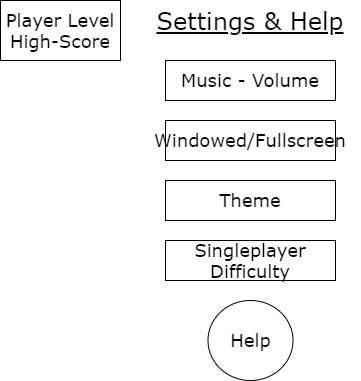
The UI also shows the number of players in the lobby, and displays each player underneath the title of their role.

**Leaderboard**

Below the leaderboard title will be displayed all players high scores and what game-mode they achieved it in. The user can sort by game-mode using the drop down above. My program also sorts the players to displayed on the leaderboard from highest score to lowest score.

I will be using a Merge Sort, to sort the leaderboard from highest to lowest high scores because the amount of data may be massive if many players play the game. Bubble sort would be much slower, especially when data grows.

**Settings & Help**



The settings provide you with information on your Single-Player high score and your player-level.

Furthermore, the “Music – Volume” option is an input box, where the user types the volume (between 0 and 100). The “Windowed/Fullscreen” option allows the user to choose between having their game Fullscreen (max screen size according to resolution), or Windowed, which makes the program appear in a smaller window and not cover the whole screen. The theme option allows the user to choose the background of their game during gameplay and out of gameplay (in the menu screen). When the user selects the “Help” option a text box appears on the screen with instructions on how to play the game. The user can close it by clicking the close button at the bottom of the text box.

*Rather than choosing the “Singleplayer difficulty” after the user has selected the Singleplayer gamemode, the user may decide to select the difficulty in the settings*