**Checkers Online: Documentation**

**Aims of the Program**

The aims of this program are to create a version of checkers which can be played over a network, and which also keeps moves and decisions open to the players who are playing. The game will also feature a single-player mode which can be played on one computer. By creating the program with these features in mind, it will allow the players of the game to play it on a network together or, alternatively, on one computer if they do not have a network which is accessible.  
The program should also be easy to pick up and play. This will be achieved by making an easy to comprehend, readable graphical user interface (GUI) which will tell the user key details such as who’s turn it is, how many checkers each player has left, and also allow them to exit or “forfeit” at any time.

**Analysis of Requirements: Functional Requirements**

Below is a list of functional requirements the program must have to work as intended and achieve functionality, if nothing else. Each one will be notated as ‘FR’ with a number, allowing it to be referenced as needed in this document, and has a brief description of what it means and why it is a functional requirement. These are

**FR#1 - Main Menu:** The program must have a main menu interface which allows navigation between three options: host game, join game or single player. This is needed so that the user can select which method they want to use to play the game in a gratifying, ease of access way. This requirement is only valid if networking (**FR#6**) is achieved.

**FR#2 – User Interface:** The game window must have a simple GUI which allows the user to play the game, if nothing else. This includes displaying the board and checkers, highlighting selected checkers, moving the checkers across the interface when necessary, showing when and who has won and who’s turn it is currently. This will allow core functionality, because it will enable players to play the game as expected.

**FR#4 – Game Rules:** The game must adhere to standard checker rules such as diagonal movement and ‘jumping’ over checkers that make the game what it is. This is needed so that it is simply a game of checkers and guarantees players will play the game properly. However, rules such as forcing players to ‘jump’ over checkers when possible from variations of the game will not be enforced.

**FR#5 – Win state:** The game must reach a state of one player winning when one player has no checkers remaining. This ensures the game doesn’t continue endlessly without end, and should trigger the GUI to display this state in a graphical manner.

**FR#6 – Networking / Singleplayer:** The main aim of the program is to create a version of checkers that is playable over a network and, therefore, this is a functional requirement. However, it is possible that networking could be deemed an impossible task. Should this be the case, it is possible that the program can function as a single player version only. However, this will then re-define the project itself and make it simply “Standalone Checkers.”

**FR#7 – Synchronization:** To achieve networking (**FR#6**) the two computers connected must be synchronised so that the two individual games are always at the same point in the game. This is necessary, because if it is not achieved, they will be essentially playing two different games which defeats the purpose of creating a networked version of the game.

**Further Details on Functional Requirements**

This section details how the functional requirements will tie together to create the full, functioning game. It will go through it in chronological order of how the functions are presented to the user.

Firstly, the menu (**FR#1**) will be presented to the user as soon as the game is launched. This will be a generic, button-based menu which presents the user with a choice of which method of playing the game they would like. This will then trigger the user interface (**FR#2**) to load which will allow ‘point-click-move’ controls with the mouse. If the player hovers over a checker whilst it is their turn, it will highlight the checker and if it is clicked, it will be in a ‘selected’ state and stay highlighted throughout the state. To exit the state, the user will have to click elsewhere on the interface, and the checker will be moved if the location they have chosen is valid according to the game rules (**FR#3**).

I will be enforcing the following checkers game rules (**FR#3)** using code throughout this ‘point-click-move’ control:  
1) The checkers can only move diagonally.  
2) Checkers can only move in the direction of the opponent’s side of the board, except in special cases. (See rules 5, 6 and 7.)  
3) The player is allowed one move per turn, except in special cases. (See rule 6.)  
4) Turns will alternate between the two teams after the player has made their move(s) for their turn.  
5) Checkers can ‘jump’ over the other team’s checkers if the diagonal space behind them is clear.  
6) If another ‘jump’ can be made after a first ‘jump’, the player can get another move in the same turn which will allow them to move a checker again.  
7) If a checker reaches the opposite side of the board to which their team started on, the checker will be granted ‘king status’ which allows the checker to move in any direction around the board.  
8) When one team has no checkers remaining, the opposing team has won the game. Alternatively, if a player forfeits the match, then the opposing team wins by default, regardless of the amount of checkers they have. (**FR#5)**9) The team that starts the game will be determined pseudo-randomly.

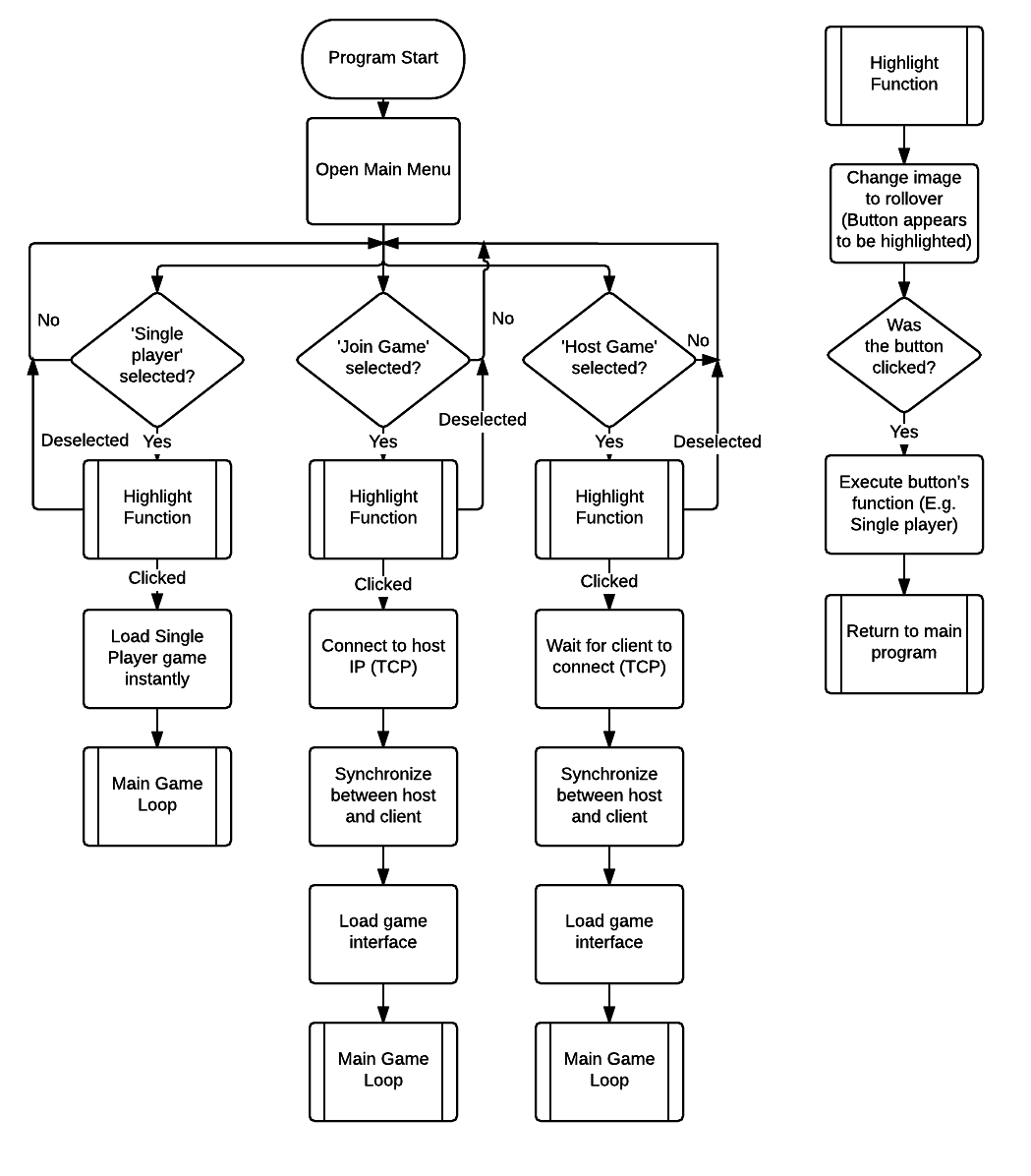
If a method of networking (**FR#6**) was chosen from the menu (**FR#1**), the player will either host or join a game, and the game will not begin until the connection has been established. The host will always be the white team, whilst the joining player will always be red. The host’s computer will pseudo-randomly choose the team which starts. This will allow synchronization from the very beginning of the game through to the end. (**FR#7**).

**Analysis of Requirements: Non-Functional Requirements**

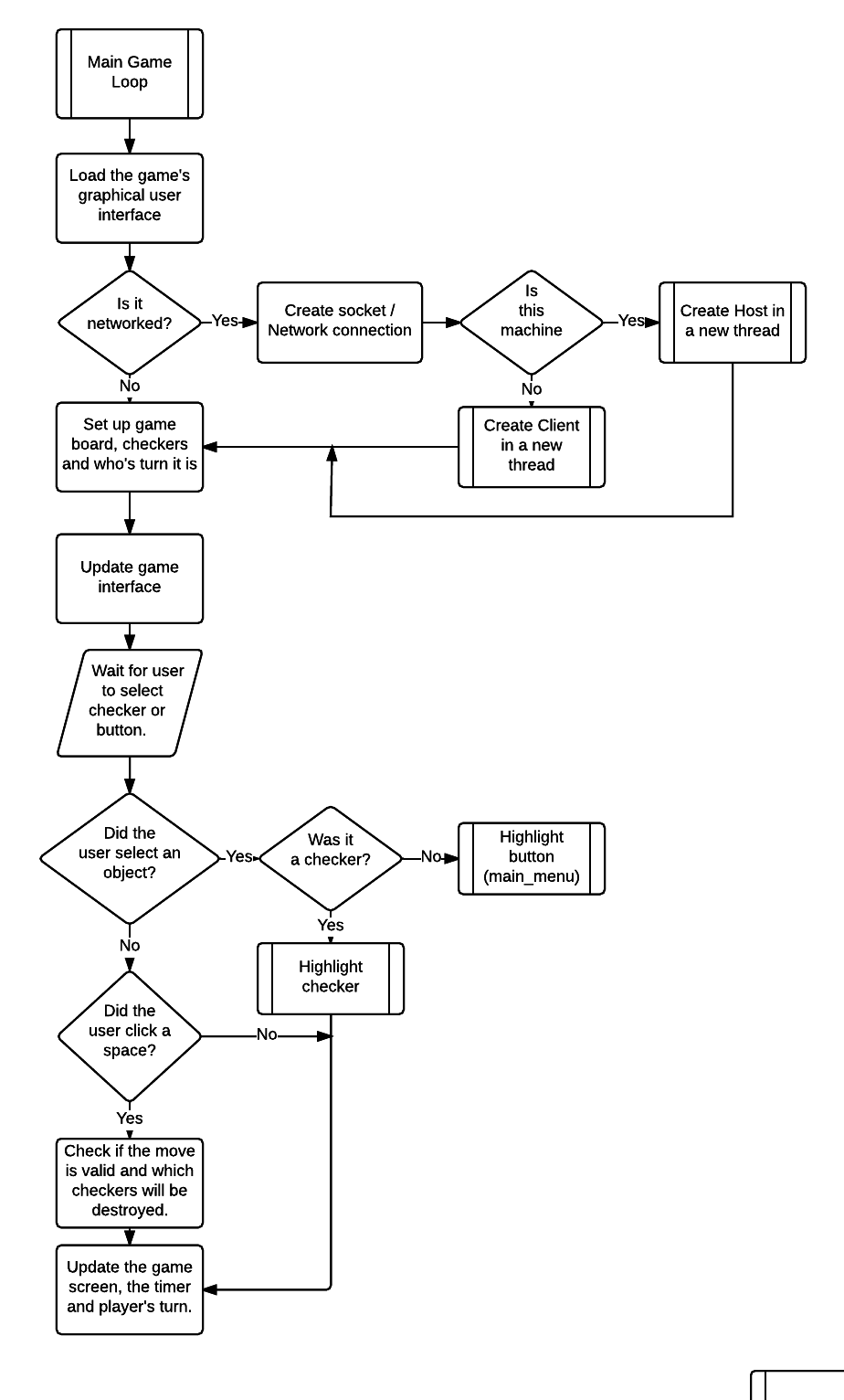
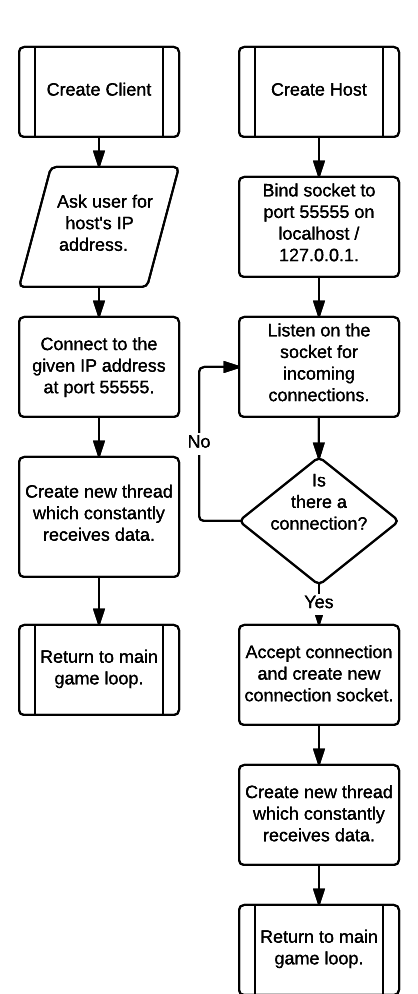
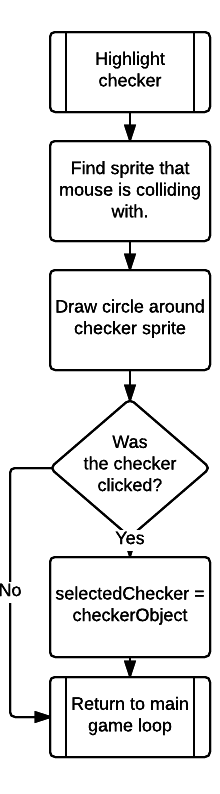
These are desirable requirements, which are not essential for the game to function, but are suitable for the game to have. They are denoted as ‘NFR#’ and a number, with a brief explanation of it.

**NFR#1 – Timer:** This will prevent each turn lasting endlessly. The timer will be 90 seconds so that the user has to make their move during that time, else they skip their turn.  
**NFR#2 – Additional User Interface Features**: It is not necessary that some items are displayed, such as the number of each team’s checkers remaining or the timer, if included (**NFR#1**). However, it would be very useful to display these items to guide the player.

**Game Design**



This flow chart displays how the program will begin from the main menu. It will open the main menu before doing anything else, and the only functionality required on the menu is the highlighting and clicking of buttons, which will then redirect the player to the game loop as required by the method of playing the game.



The above flow chart shows the main loop of the program. The user will enter this loop from any of the choices they have been given, but the button they choose will send the necessary parameters to the main loop to allow it to then either establish networking or not. This diagram leaves out intricate details such as how the game will be created, or how checkers are represented in code, and also how it will be logically recognised if a move is valid, for example.

Since these details are important and difficult to program, I have used other methods (UML class diagrams and pseudocode) to show more information about these aspects.

Below is a UML (Unified Modeling Language) class diagram for a checker. For the purpose of explanation of what the contents are, it will include a brief description of each of the attributes and methods in the diagram.

|  |
| --- |
| Checker (Sprite Object) |
| String:Image – The image of the checker. (Red or white, and king versions if this object is a king.)  Tuple:Rect – The rectangular boundaries of the sprite, allowing collision detection in pygame.  String:Team – A definition of whether the checker is on team “r” for red, or “w” for white.  Boolean:King – A Boolean variable which represents whether the object has king status or not.  Tuple:Location – A location of where the object is in the board. A 4x8 list will be used to represent the board, and this location variable will represent the index the object is in within this list. |
| moveChecker(location) – A function which will move the object from one tile to another on the board by printing it to the screen again. Must check if the move is valid before moving it.  updateLocation(listLocation) – This will be used to update the ‘Location’ attribute of this object so that the object knows where it is on the board. Useful for checking where the object can move to in *moveCheck().*  destroyChecker() – The function which will remove the checker object from the game when the checker is ‘jumped’ over.  makeKing() – A simple function that will set the king status to true and change the object’s image to the king image. |

This is the basic structure of the class which will be used to create checker objects. The sprite will be displayed on screen and contain the information necessary for the player to move them across the board, destroy the checkers and make them a king, as the game rules explained.

*Comment: In the finished version, the methods were edited slightly to accommodate for validation of checker movements and additional ‘jumps’ made in one move. It also includes a draw() function so that the checker can be re-drawn to the surface at any time by any of the other methods that may need this.*

Below is some pseudocode which I wrote just before beginning the project. It briefly outlines the logic for adding (x,y) tuples of locations on the board that the checker can move to a valid moves list. When the moveCheck() function is called, this list will be created and the code will check if the mouse position is at these points on the grid.

**Key of Variables:**

**Checker**: An instance of the ‘checker’ class which is not on the player’s team, and can therefore be destroyed.  
**Self**: Referring to the currently selected checker by the player.  
**x:** The x position of the currently selected checker (From the **tuple:Location** attribute) used for logic.  
**y:** The y position of the currently selected checker (From the **tuple:Location** attribute) used for logic.  
**.team**: The **string:Team** attribute of the checker instance.

// from old space to new space = old space + 1  
// therefore new space - old space = 1 (or -1)

// assuming object is on a tile already, done during board creation

if checker on tile:  
 if checker.team NOT self.team:  
 if (y MOD 2 == 0) and x NOT 0: /// new y is even and x is not at index 0  
 if (new x of checker) - (current x of checker) == 0: // last move was LEFT  
 x -= 1 // therefore move one to the LEFT again

// else it must have moved RIGHT  
 // x does not change, so RIGHT again

elif (y MOD 2 != 0) and x != 3: // new y is odd and x is not at index 3  
 if (new x of checker) - current x of checker) == 0: // last move was RIGHT x += 1 //move to the RIGHT again

// else must be x-1 and moved LEFT previously  
 // x does not change // moves LEFT again

// else it is an impossible move, not added to valid moves list

if (new y of checker) - (current y of checker) == 1: // checker moved DOWN board  
 y += 1 #therefore move DOWN board again

else: // checker has moved UP board  
 y -= 1 // so move up board again

add (x,y) to valid moves list

elif tile is empty:  
 add move to valid moves list

**Testing**

A table of tests has been recorded below. The table includes the following information:

• Functionality Tested: Which functionality of the program was tested.  
• Test Number: Tests will be numbered for each functionality, starting at test 1 on each. (For reference later in this document.)  
• Test Details: What the test consisted of.  
• Expected Results: The results that are expected before the test is performed.  
• End Test Results: The results of the test which was performed.

If the end result is what was expected, the table’s cell will be coloured green. If it is not what was expected, it will be coloured red.

A conclusion of the test results will be written below the table, along with comments on the performance of the program and what will need to be changed as a consequence of the results.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Functionality Tested | Test Number | Test Details | Expected Results | End Test Results |
| Main Menu | 1 | The mouse will be hovered over each button in turn. | Each button should appear to be ‘highlighted’ when the mouse hovers over it. | Each button ‘highlights’ correctly, as expected. |
| 2 | The mouse will be clicked on ‘Host game.’ | The button should trigger the connection screen to appear, showing ‘waiting for connection’ and their IP address. | This button functions correctly and loads a screen which displays ‘Waiting for connection…’ along with my IP. |
| 3 | The mouse will be clicked on ‘Join game.’ | This should trigger the connection screen to appear, showing ‘Waiting for connection…’ along with a windows dialogue. | As expected, this causes the correct screen to load and the windows dialogue to appear, prompting the user to enter an IP to connect to. |
| 4 | The mouse will be clicked on ‘Single player.’ | Clicking this should lead to the game starting instantly in single player mode, allowing the user to control both teams on one machine. | The single player version of the game loads correctly, without establishing a network and allowing full control over both teams from the one machine. |
| 5 | The red ‘X’ button will be clicked.  (Exit Button) | This should exit the program. | The program exits correctly. |
| 6 | The mouse will be clicked around the remaining screen space. | No button highlights or functions should be triggered. This should simply do nothing. | Nothing happened, as expected of the program. |
| Connection Screen (Host Game) | 1 | The game screen will be clicked. | Until a connection is established, clicking should do nothing. | As expected, the game screen does not do anything. |
| 2 | The red ‘X’ button will be clicked.  (Exit Button) | This should exit the program. | The program becomes unresponsive due to the socket blocking I/O. As a result, the program was modified by closing the socket and catching errors in try-except when closing the socket. |
| Connection Screen (Join Game) | 3 | The windows dialogue red ‘X’ button will be pressed. (Exit windows dialogue.) | The windows dialogue should close and cause the program to do nothing. | As expected, this closes the windows dialogue and the program does nothing. |
| 4 | The IP address ‘127.0.0.1’ will be typed into the text box on the windows dialogue and entered. | Since the host game will be on the same machine for this test, 127.0.0.1 should connect to the localhost game correctly. | As expected, the two games connect correctly and the game begins once they are connected. |
| 5 | The IP address in the text box will be left blank and entered. | The program should close, as the IP address cannot be blank. | The program closes, as expected. |
| Connection Screen (Both) | 6 | The game will be hosted on a virtual machine with a different IP address and main computer will attempt to connect. | The virtual machine and main machine should connect and the game should start normally. | The virtual machine and main computer connected successfully, resulting in the game starting correctly. |
| Game Screen | 1 | The forfeit button will be pressed. | This should exit the game and send a ‘forfeit’ signal if the game is networked. | This worked, but the opponent’s game became unresponsive. After some investigation, I found that it was because the socket was attempting to receive data after the connection had been closed. Therefore, the socket now closes when the game ends in this way. |
| 2 | The main menu button will be pressed. | This should make the game exit and return to the main menu. | The game exits and returns to the menu. |
| 3 | The mouse will be hovered over each checker. | Each checker will highlight in turn. | Each checker highlights in turn, as expected. |
| 4 | Each checker will be clicked, and then ‘unclicked.’ | When clicked, the checker will stay highlighted and will show the player assist, displaying where it can move to. | This worked as intended for all of the checkers selected.  (Tested with both teams.) |
| 5 | A full game will be played in singleplayer. | The game should allow movement of checkers, jumping over opponent checkers, and a win state. | The game worked correctly, however there was a glitch where you could jump over a checker at the top or bottom of the board. Fixed using logic. |
| 6 | A full game will be played in multiplayer. | The game should allow movement of checkers, jumping over opponent checkers, and a win state. | This worked correctly after applying the fix for the glitch mentioned above. |
| 7 | Two checkers will be set up so that an opponent’s checker can perform a ‘double jump.’ | The two checkers should be able to be ‘jumped over’ in the same turn. | This works correctly, and also resets the timer when the second jump can be made. (Gives the player a second turn, as intended.) |
| 8 | Wait for the timer to reach 0. | This should automatically switch turns to the other team after the timer hits 0. | The turn switches from red->white or white->red as intended. |

After some minor fixes, the program fully achieves the aims originally set out, with even some extra functionality that was added during the process of programming the game. For example, the player assist which shows where the checker can move to was not originally intended. The testing went well over all with only 3 unexpected results (Connection screen test 2, game screen tests 1 and 5) which were fairly easy to fix.

**Critique**

I think that the program is successful, as the program has reached all of the original aims set at the beginning of the project and functions very well. It is easy for a user to pick up and play with little need for instructions, and the graphical user interface looks professional and clear. However, the game could be improved by making the interface customizable to the user’s preference of colours or to support colour blindness.  
Furthermore, it has also been noticed that the timer can be exploited and purposely using it to not move checkers. This is an issue, because a player can purposely keep their checkers in, for example, the corner of the board and avoid moving them. This makes it impossible to jump over the checker, and as a result, impossible to win the game until they move. If more time was available for the project, this could be fixed by adding extra functionality to the timer and making it move a checker at random if the timer expires.  
Furthermore, there are very slight issues with the timer which do not affect the gameplay or timer at all, but make it look as if the timer is skipping seconds. This is due to the way that the pygame events work. When the mouse is idle for more than one second, the timer will tick down second by second, but when moving the mouse again, it will tick down the one second it missed instantly because the timer event is processed. I was unable to find a way to fix this irregularity, but it is not game-breaking and even over a network, the two games remain synchronised. It is unclear how it could be fixed, seeing as it is an issue with pygame.  
If I were to do the project again, I also think it would be beneficial to make the Host and Client classes inherit from a base Network class. This is because there are similarities between the two which could be inherited from a base class, instead of writing the same code twice. The same applies to the pyButton class, except in reverse. I have made a base pyButton class, but I think the different button functionalities such as ‘menu’, or ‘forfeit’ could be separate classes which inherit from the base class, and then add functionality to these buttons to make the class more reusable for future projects.