2019

Name: David Terence-Abanulo Candidate Number: 3651 Centre ID: 12290

Woodhouse College

10/14/2019

NEA ANALYSIS FOR UNO PROJECT

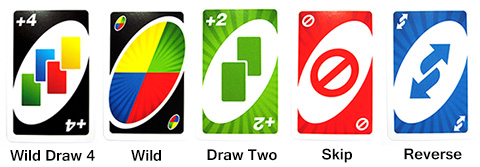


**Analysis**

**1.1-Introduction**

As a student, I find myself often looking for a quick game to pass the time when taking a break from revising and usually that game ends up being UNO. Therefore, my project for my NEA is to create a fully functional 2D shedding card type game that can be accessed as an executable file and run like an application on windows. This application would contain a single player function where the user can play against an AI and a multiplayer function with which up to 4 players can play locally (connected to the same server) or play against each other with the addition of an AI to fill empty spaces (if less than 4 players join the server).

The game itself works with up to 4 players, each player is dealt a random set of cards from the UNO deck, players then take turns placing a card from their hand into to central pile, a card a player places on the pile must be the same number or colour as the card on the top of the pile, however players can always use action cards in their hand (including skip player, +2 cards to consecutive player, +4 to a consecutive player etc.)

[[1]](#endnote-2)

Action cards in UNO

**1.2 – Outline of tasks (both programming and documentation)**

|  |  |  |
| --- | --- | --- |
| **Task** | **Priority (out of 5)** | **Due date** |
| Introduction | 1 | 15/09/19 |
| Analysis of similar systems | 2 | 20/09/19 |
| Analysis of existing system | 3 | 22/09/19 |
| MVC implementation analysis | 3 | 1/09/19 |
| Questionnaire | 3 | 6/10/19 |
| Prototype (deck of cards) with annotations | 4 | 8/10/19 |
| Researching AI algorithms to implement in python | 4 | 10/10/19 |
| Proposed Solution (include multiplayer solution) | 4 | 12/10/19 |
| **Analysis Completed & Printed** | **5** | **14/10/19** |
| Complete “deck of cards” prototype in OOP |  |  |
| Create a start-up menu in pygame |  |  |
| Write an overview for documented design |  |  |
| Create flowcharts for main functions of project |  |  |
|  |  |  |
|  |  |  |

**1.3 -Description of Existing Application**

UNO is a lot more established and well-known in comparison to other shedding type card games with many variations of the game available on Android, IOS, Xbox, PS4 and PC (windows). I will be focusing on analysing the windows version of UNO which is an application available on the mainstream digital application store, Steam. This is version which is developed by Ubisoft is the most popular online rendition of UNO to date with an estimated 500,000-1,000,000 owners of the game and the highest number of players online at once totalling up to 2,309 and up to 115,000 viewers on live streaming platforms such as Twitch.

[[2]](#endnote-3)

Line graph depicting the fluctuation of Players and Viewers of the game since its release in January 2017 till date

This version of UNO works essentially the same way as the original does, with each player given an equal number of cards from the deck and the remainder placed as a draw pile. Players must match the colour or number of a card placed down in the discard pile or use action cards randomly distributed amongst them (including, skip, reverse, +2, +4, and wildcards) to reduce the cards in their hand to 0 first. Once a player has one card remaining, they must call ‘UNO!’ when placing their final card in their hand, if this is not done then the player must draw another card from the draw pile.



2v2 option (multiplayer)

User account name and profile picture (there is a database that holds this information

Displays what card has just been played

In game Voice Chat

**[[3]](#endnote-4)**

Themed Cards

Colourful GUI to draw the user in-aesthetically pleasing

Number of cards each player has left



Score

**[[4]](#endnote-5)**

**Strengths (according to User reviews on steam)**

* Colourful and exciting GUI really enhances the user’s game experience, arguably better than playing in real life
* Good game to play online with friends through multiplayer
* Themed cards add an interesting twist to the gameplay
* Offers controller support
* Available in multiple languages

**Weaknesses (according to User reviews on Steam)**

* Game is not worth $10
* A max of only 4 players allowed to play in one game
* No support for rebinding keys to user’s preference
* Noticeable latency, poor online connection during multiplayer
* If you don’t have friends that play the game, then your experience with single player AI would be very dull, and searching for an online match takes too long
* Game frequently crashes
* Poorly optimised for users on different

This version of UNO works exactly as a normal game of UNO should and attracts users with its vibrant colours and easy to use GUI however the multiplayer aspect (which seems to be the most sought-after feature from users) is very poor due to latency severe latency issues and game crashes. This leads me to believe that this version of UNO is poorly optimised for networking. Furthermore, this game is only available on Windows OS and its minimum requirement to run is Intel Core i3 530 2.93GHZ which leaves out a considerable number of PCs on the market.

Based on this current version I plan to create a similar GUI game display, with players situated in the corners of the screen and only being able to view their own cards and the last placed card in the middle with the addition of a score counter next to each player that will increase or decrease based on cards played and whoever wins the game. I aim for my game in multiplayer to have a room/lobby that players can join and launch a game together across the same network server. Since a common issue is latency, I will endeavour to reduce latency in my game. I also will ensure my game can be run on majority of modern computers and laptops which includes devices running windows and MacOS. This version of UNO also requires you to have the Steam Engine installed on your desktop to play whereas my version will simply require python.

**1.4-Analysis of Two Similar Card Games**

As UNO is a shedding type card game, I have found another two games that are similar to it:

**Boom-O (created in 2001):** In Boom-O each player is dealt 7 cards and 3-time bomb cards that represent ‘lives. Players are given cards that can either increase or decrease the timer. The main objective is for the player to put down one card per turn until they have no cards in their hand while trying to keep the timer total below 60 seconds. If a player can’t play the correct card, they must turn over one of their bomb cards, losing a life. Once a player clears their hand, all other players must turn over their bomb cards and the survivor wins.

****

Bombs card (3 per player)

Discard pile

Draw pile

[[5]](#endnote-6)

**[[6]](#endnote-7)**

**Strengths**

* The element of a timer and ‘lives’ implemented into the card game adds a sense of intensity and urgency when playing the game, this allows the game to not last too long but be a fun game to play with friends
* The game is heavily reliant on the aspect of luck which means everyone has an equal opportunity of winning, no matter how experienced player(s) are
* Unique action cards such as ‘trade hands’ and ‘double play’ can completely change the course of gameplay in an unexpected way which makes playing the game more fun

**Weaknesses**

* The game is heavily reliant on the aspect of luck which means there’s no real strategy to learn or create.
* There is no score counter in the game which could serve as another method of winning, e.g. if a player score reaches 100, they automatically win. Scores could be based on cards added to the deck.
* The rules can get quite confusing as players have to keep track of both the timer and the number of cards, they have left in their hand

**Craits (created in the 1970s):** In Craits a standard card of 52 cards is used instead of a specialised pack in UNO and Boom-O. There are 15 hands and in the first hand 8 cards are dealt per player, this amount decreases by 1 after every hand till 1 card is dealt to each player and then incremented by 1 until once again 8 cards are dealt to each player. Like UNO, each player takes turns in placing a card from their hand on to the pile with according matching suit or rank (this excludes an eight or nine which are wild cards). Furthermore, if a two is placed on the pile at any point ‘the count’ begins in which each player must place an Ace or a two which increments the count by 1, once a player fails to place an Ace or a two they must draw a number of cards equal to the count e.g. if the count = 10 then the player must draw 10 cards from the deck.

**[[7]](#endnote-8)**

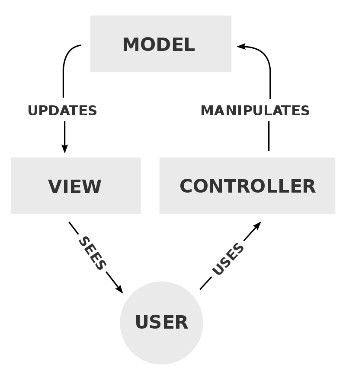
**Strengths**

* The fact that this game utilises a standard deck of 52 cards means that it is more convenient and cheaper to play as one does not need to buy a specialised deck

**Weaknesses**

* The game can get repetitive and boring quite quickly as there is no scoring system for each player
* Players may find it difficult to remember the special attributes of each card as they are just normal playing cards, this may end up confusing gameplay

**Conclusion:** Boom-O and Craits are very similar to UNO in the fundamental fact that the main objective of the game is for the player to reduce their hand to zero cards before their opponents. Boom-O is faster paced and this may appeal more to children as games do not last very long and are more intense, on the other hand Craits is a much more slow and steady game that doesn’t really have any unique aspects to it as a shedding card game but this may allow for a more mathematical, strategic approach when being played.

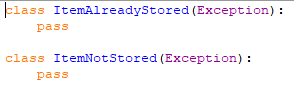
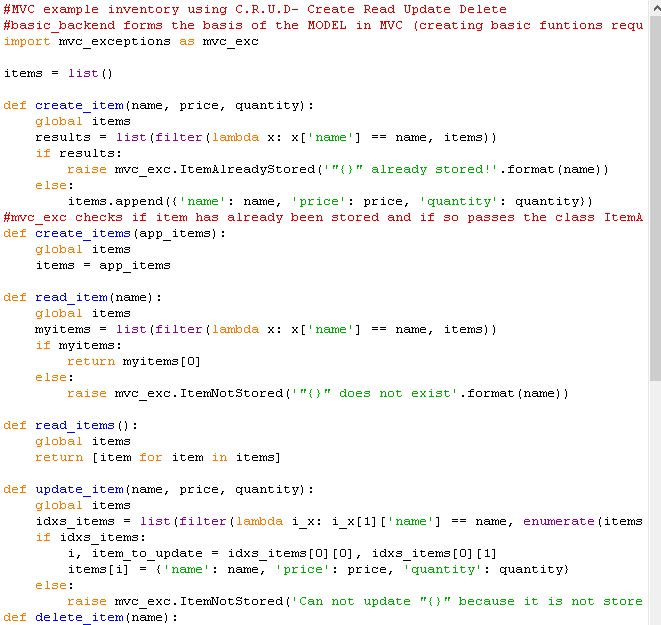
**1.5 Implementation Research**

**Model-View-Controller**

The fundamental structure of my UNO game will utilise an MVC which will allow me to manage the internal coding mechanics of the game and the design of the game in a more separate and effective manner. MVC stands for Model-View-Controller which is a software design pattern used for creating a user interface. The View is what the user sees and interacts with on the screen, the Controller is core code that runs in the background, processing inputs from the user and the Model essentially keeps a record of all the changes performed by the controller and updates the View accordingly for the user. [[8]](#endnote-9)

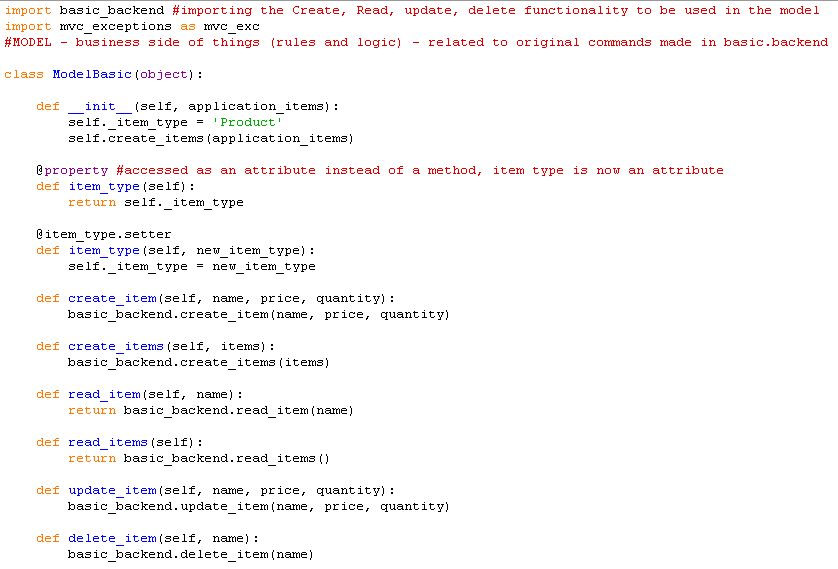
For instance, in my UNO game, the Controller would be the fundamental source code that controls how the cards are shuffled and evenly distributed amongst players, deciding which cards have special attributes and the effect on other players, keeping scores for each player, keeping track of which cards have been discarded and how many cards each player has left. This information would be passed on to a Model and then displayed visually to the user using Pygame.

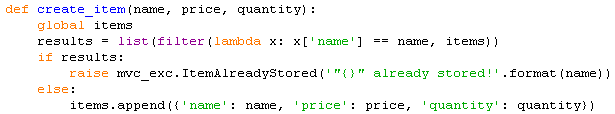
To learn how an MVC works in python and as a result how to implement my own one from scratch I have followed an online tutorial to create a mock inventory of a supermarket, in which the user can view what items are in stock, add or remove an item from stock and update the price or quantity of an item in stock. First, I had to create basic commands that could be done on a supermarket inventory (such as create, read, update and delete).

**** [[9]](#endnote-10)

In addition to creating the basic functions I implemented error-handling using try and exception: methods to be used if a user tries performing an invalid action such as trying to update an item that doesn’t exist

After making these basic functions I imported them into the main MVC program, uses the functions made is basic\_backend.py to formulate a model of the inventory, and a controller that takes user input from the view and refers to the model to make changes to the view.



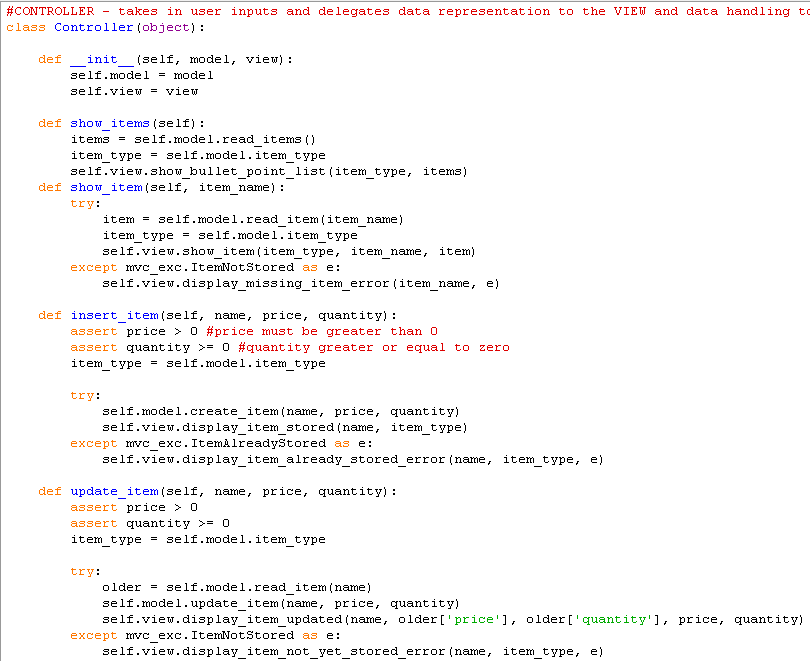
****

Above is a snippet of the model aspect of my MVC implementation. As you can see the basic functions defined in the basic\_backend file are used here

****

Here is the view aspect of an MVC model, as you can see, I created text-based images that would appear on the screen if the user attempted to perform certain commands e.g. creating an item that already exists

bvcccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

****

And finally, here is the ‘controller’ aspect of an MVC model. This is arguably the most essential part as it enables the main program to function by taking inputs from the view module, processing them based off the basic functions used in the model module and getting a result which is then outputted back to the view and the model is updated for the next input, this is done by inheritance and polymorphism. The controller essentially links the model and the view together.

bvcccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

**AI (Maths & Algorithms)**

**Maths and Complexity behind UNO**

**Maths & Monte Carlo** - Uno is essentially a game of luck and involves little to know strategy or maths. Which would make it difficult to create an AI. The AI in my game should be able to analyse playing patterns of each player and play cards accordingly to win (which may include predicting what cards a player has in their hand based on what cards they already played).

Assuming action cards such as reverse, skip, +2, +4 and change colour are not considered. In single player mode against an AI, you and the AI are given an equal number of cards. There are 52 cards in a deck, which means each player get 13 cards. Out of the total 52 cards, there are 4 suits (red, green, yellow and blue) and each suit has 1 set of 0-9 cards and. This means there is a 10/13 chance of getting any 0-9 card of any suit and a 1/13 chance of getting a +2 or reverse or skip card. In terms of analysing which card each player has in their hand, this means that players are more likely to have numbered cards as oppose to action cards and if there are a total of 4 players then it is likely that one, or even two players may not have actions cards at all. This would be helpful if one was taking a more mathematical approach to creating an AI like the Monte Carlo method. The Monte Carlo method uses probability distribution to evaluate the success and failure of a model. I would use this in my UNO game by inputting a range of probability values for the chances of each type of card being placed on the pile after every move in the game until a user empties their hand. The Monte Carlo method would repeat this for random ranges of probability values (from most extreme to most likely) until it has an 'accurate' outcome for each situation. For instance, if the AI and the user were trying to win after a Blue 7 was placed on the pile, the AI would go through a bunch of different scenarios such as, playing a wildcard on to the pile or a Blue 0, or Red 7 etc and keep playing till it or the player empties their hand. If the AI empties their hand first (thus winning) the AI would record how many moves it took to win and whichever method took the least number of moves would be assigned a high value of probability. Eventually after going through every outcome, my AI would have a list of efficient method to win against a user. Each method could be put into a tree that my AI can traverse and based on the situation my AI would pick a tree and traverse it. This method is laborious but it suitable for a game such as UNO that doesn’t necessarily have a distinct probability attached to each move (as the game is more luck based than strategy).

**Minimax Algorithm** - As an UNO AI would essentially be searching through a set of valid moves, using the optimal moves to get to the goal which in this case is getting rid of all the cards in its hand first. This means that I can represent this as a tree algorithm, where the start node is when the first card is placed on the pile and the AI has x number of cards in their hand and the final node is when the AI has no cards in their hand. Now I must start to think about how an AI would traverse this tree algorithm from node to node in the fastest route possible.

This is done by using the Minimax algorithm which finds the optimal move for a player. It is typically used in two player turn based games such as Chess and Tic-tac-toe this means it should work well for my UNO game. In a Minimax algorithm the two players are defined as either the maximiser and a minimizer where the maximiser attempts to get the highest value possible and the minimiser attempts to get the lowest value possible and they traverse a tree from state to state with each state corresponding to a defined value, these values are calculated using a heuristic function. In terms of tic-tac-toe, using the heuristic function one could calculate the value of the board based on the certain state of the game, e.g. if the maximiser wins the board then the value is set to 10 and if the minimiser wins the board then the value is set to -10. This idea is used to form the basis of a tic-tac-toe game AI by creating a findbestmove function, a minimax function which considers all the moves the AI can make and returns the best value for that move. For this algorithm to work, the AI would be the maximiser and the other player (in this case the user) would be the minimiser.

An example (which I will be following) of minimax being used to find the optimal move for an AI:

**function** minimax (board, depth, isMaximizingPlayer):

**if** current board state is a terminal state:

**return** value of the board

[[10]](#endnote-11)

**if** isMaximizingPlayer :

bestVal = -INFINITY

**for each** move in board :

value = minimax(board, depth+1, false)

bestVal = max( bestVal, value)

**return** bestVal

**else** :

bestVal = +INFINITY

**for each** move in board :

value = minimax(board, depth+1, true)

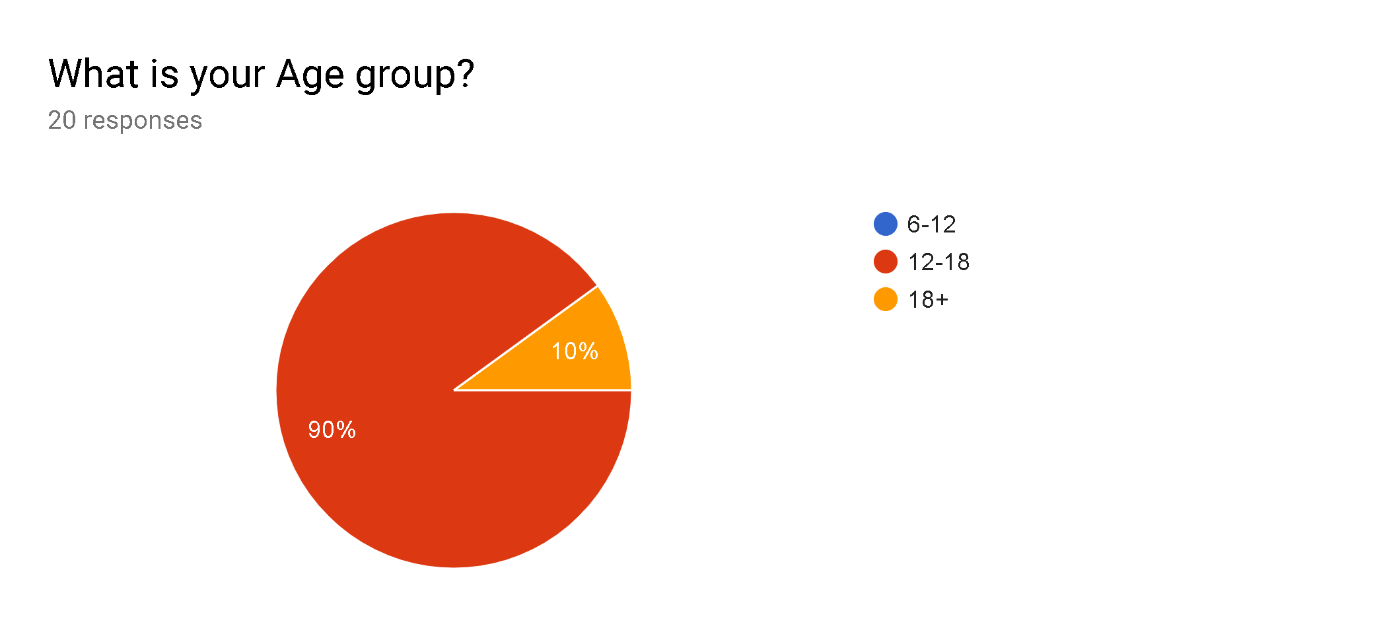
bestVal = min( bestVal, value)

**return** bestVal

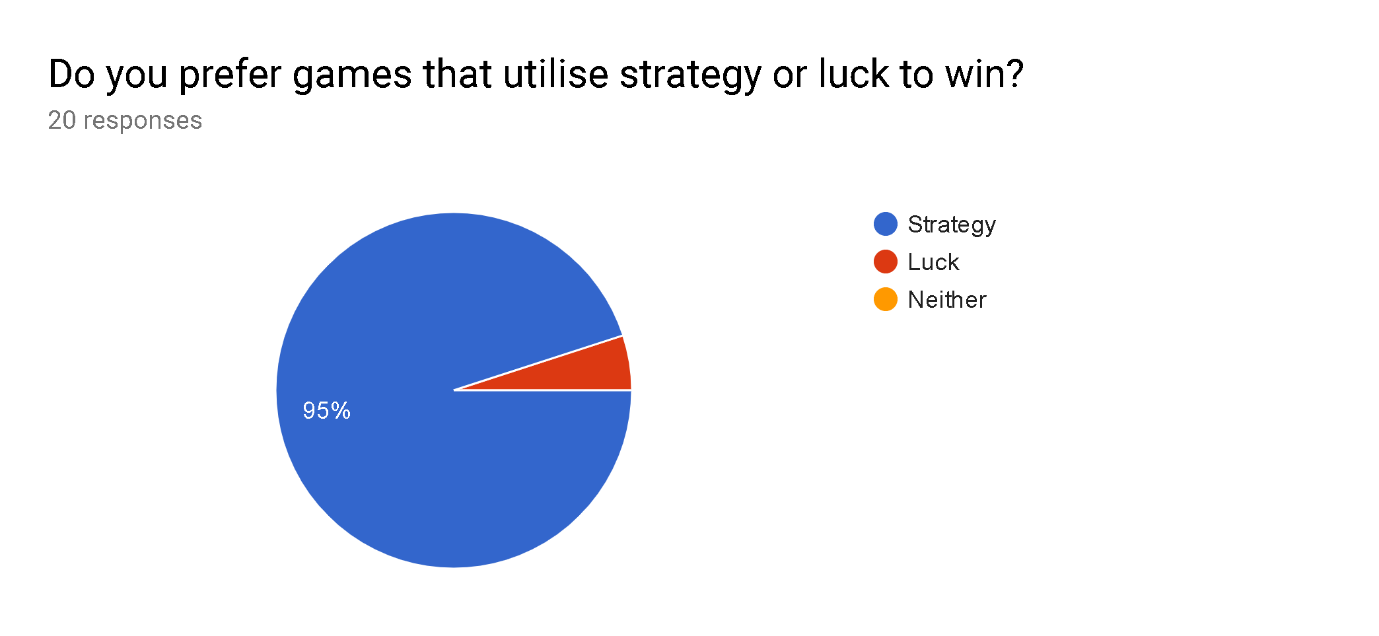
However, this algorithm may prove too difficult and time constraining to implement, if this is the case then I will use the built-in random function in python to create a basic AI, that randomly picks cards from it’s hand to play at each turn.

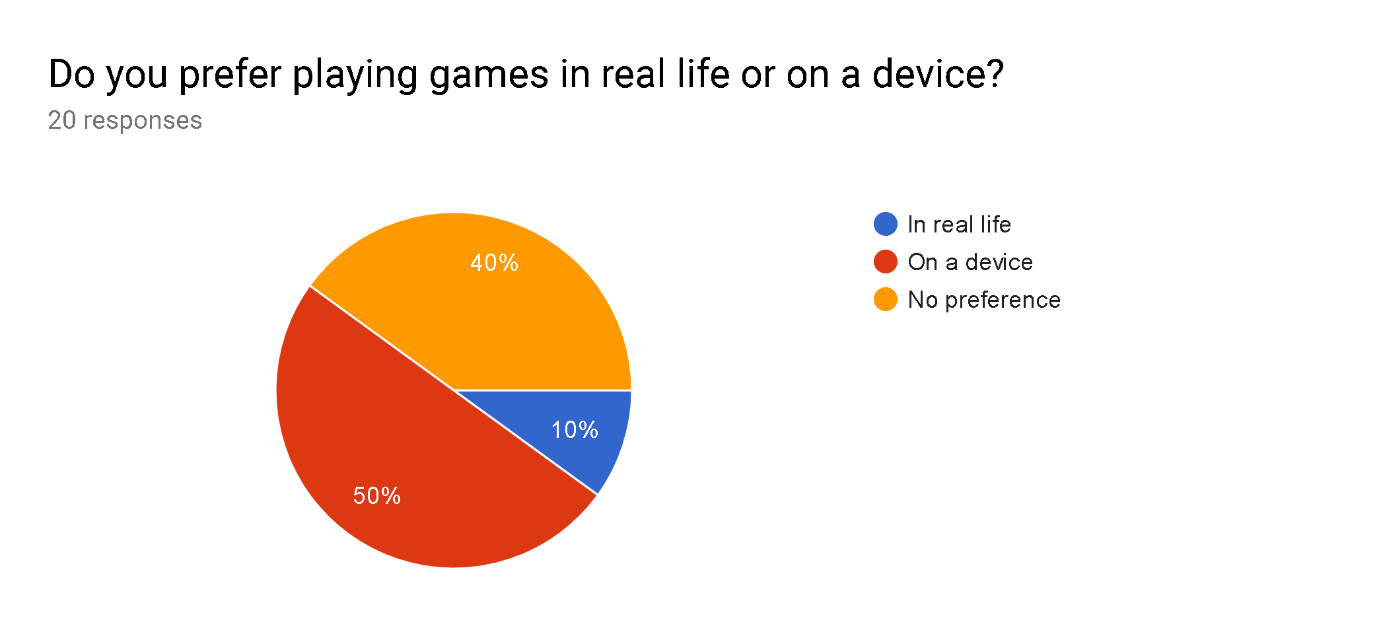
**1.6 Analysis of Responses from Questionnaire**

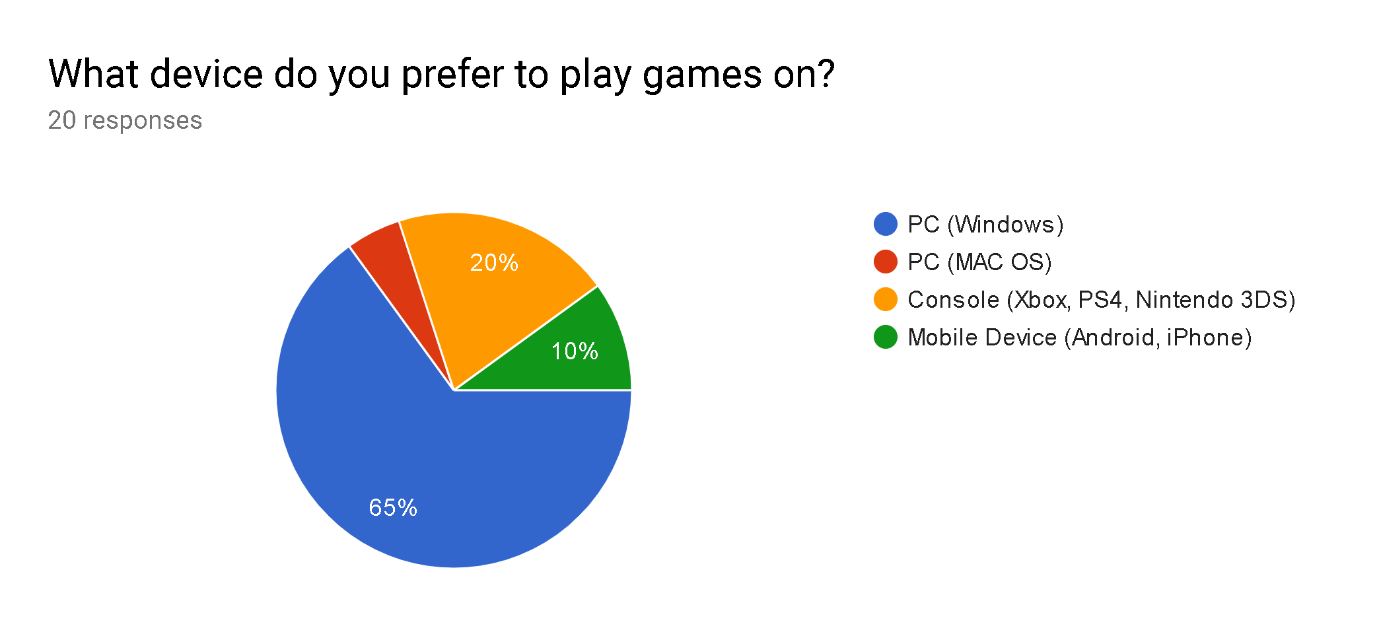
On the 24th September I sent out a gaming-based questionnaire to learn more about my potential users and their views on certain aspects of gaming. These were their responses:



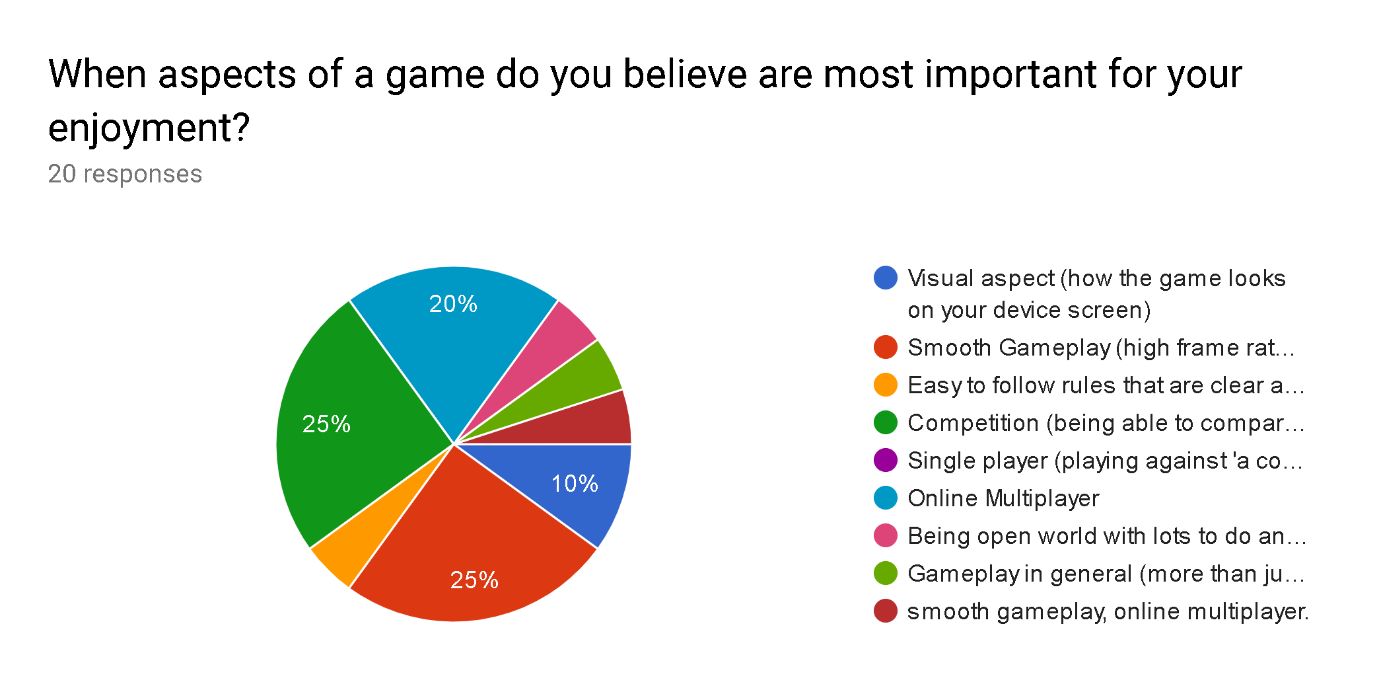
A significant of my potential users will be teenagers between the age of 12-18, this means I can assume that my users will be familiar with the technology used (in this case a computer or laptop) and as majority of my users are in their adolescence they should be able to catch on to the rules of UNO at a much faster rate than someone older which means I have to won’t focus on explicitly detailing the rules of the game.

A resounding 95% of responses said they prefer a game that involves strategy. This is good because strategy in a UNO helps make the game more interesting and addictive because users will constantly be thinking of new ways to play certain cards at certain times to ensure they win more games. To motivate users, I intend to create a function that gives users extra points based on how quickly they manage to win a game, urging them to think of fast UNO strategies and giving them an incentive to keep playing to try beat their best time. Furthermore, with the implementation of an AI users will be able to see how their strategies stack up against a ‘computer’s’.



This question received mixed responses, but majority said they prefer playing games on a device. This is probably due to the added convenience playing on a device has, as users can play anywhere at any time and with any of their friends. My UNO game will be limited to being played on a desktop but will allow users to play with their friends across a LAN network (to reduce potential latency issues)

Based on the responses from this questions, majority of my potential users prefer to use PC as oppose to a console or mobile device. In addition to this 93% of those who chose PC as their preference, favour Windows OS over MAC OS for gaming. This means that I will focus on making sure my game can run on windows OS. However, a secondary focus in the future would be trying to make my UNO game compatible with MAC OS users as this was the

second most popular choice. 

The responses for this question were quite evenly spread with the two most popular choices being ‘Competition (being able to compare you score with others, win: lose ratio etc)’ and ‘Smooth gameplay (high frame rate and reduced latency). This means that for my UNO project, I must focus on ensuring each user has a score that remains on their profile and potentially allows users to compare scores with their friends as well as making sure the game runs smoothly on python, which may mean implementing a time.sleep module in python and the clock module from pygame to make my blocks of code run in a more sequential manner.



- ‘too many rules’ – this means that when I add a ‘help’ function for users I will make sure it’s not too forced so those who are experienced in gaming or already know the rules of UNO can quickly avoid it and get straight into the game

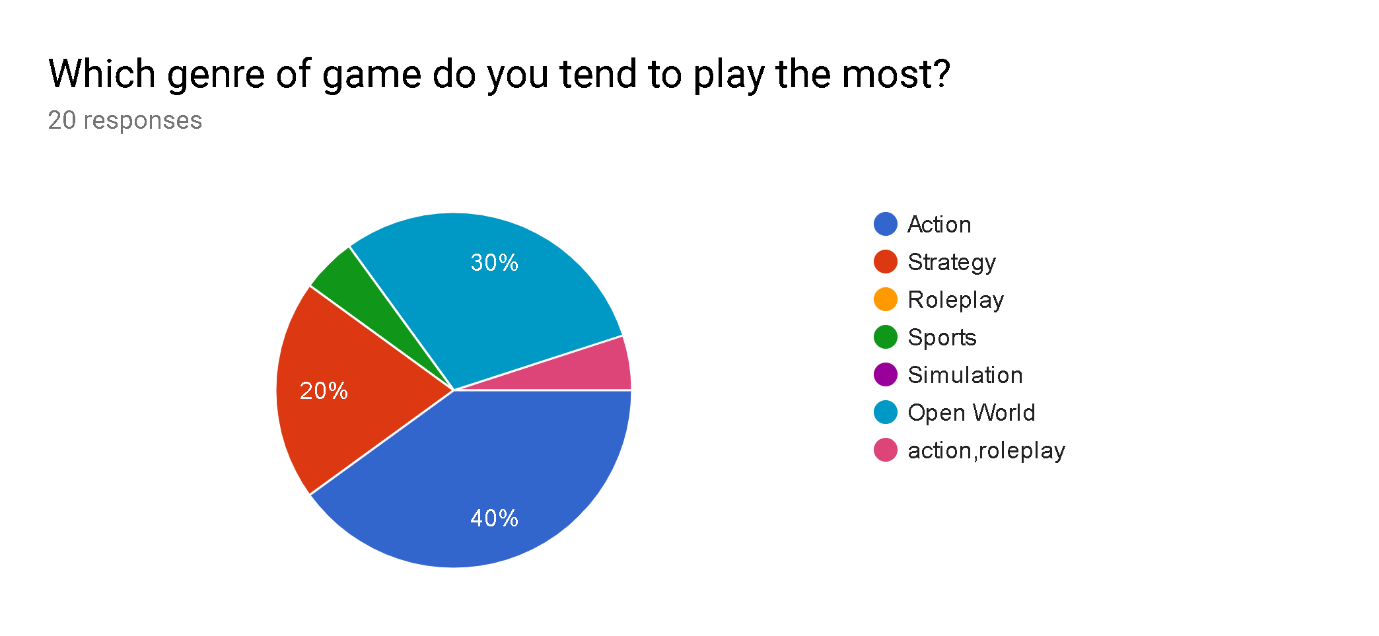
- ‘turn based, top down, being like Dota gameplay wise’ – this is good because UNO is already a turn-based game, but UNO is not a top down game like Dota

- ‘games that don’t have an in-game text chat or voice chat’ – this isn’t a priority for my project however if given more time on the project I could implement a ‘say’ function where players can type ‘Say’ following by what they wish to communicate.

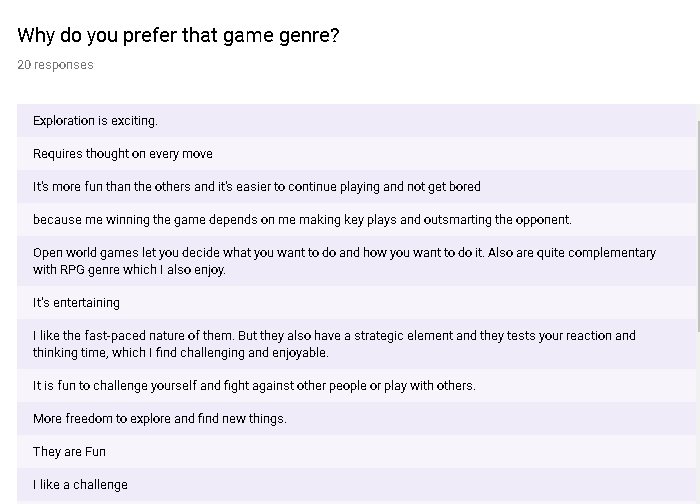
- ‘when there is a lack of choice of how to play the game and progress’ – UNO is a very versatile game in that there are no set ways to play it, if the user clears all their cards first the method they choose to do so is up to the user. However, there is a lack of progress in UNO, I could attempt to implement an advancement to the basic scoring system, in which a certain score equates to a milestone and with each milestone users unlock more features, such as card themes etc.

- ‘frame drops and lag’ – my game will likely not have any lags or frame drops as It will be running on python which is optimised for most windows devices.

- ‘pay-to-win’ – My rendition of UNO will not include any pay to win aspects.

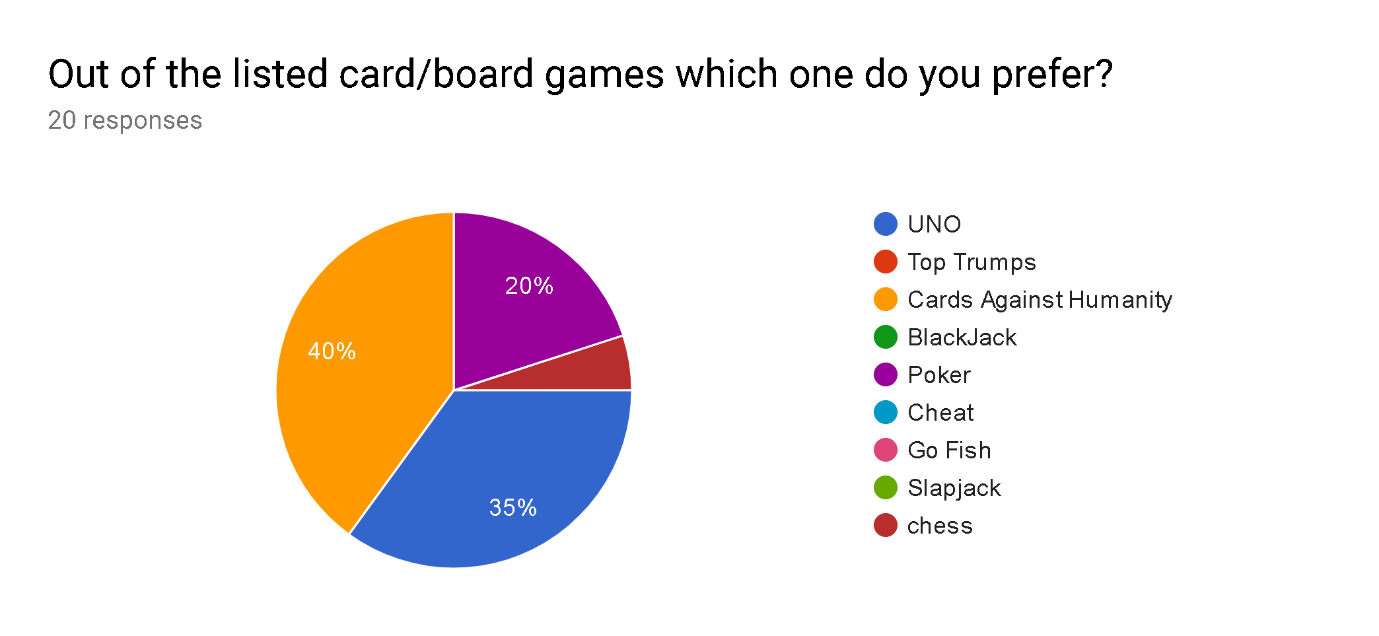


The most popular choices were ‘Action’, ‘Strategy’ and ‘Open world’. While ‘open world’ doesn’t apply to my application at all. I will aim to make my version of UNO fast paced to give users a sense of action and users will be able to discover and implement their own winning strategy

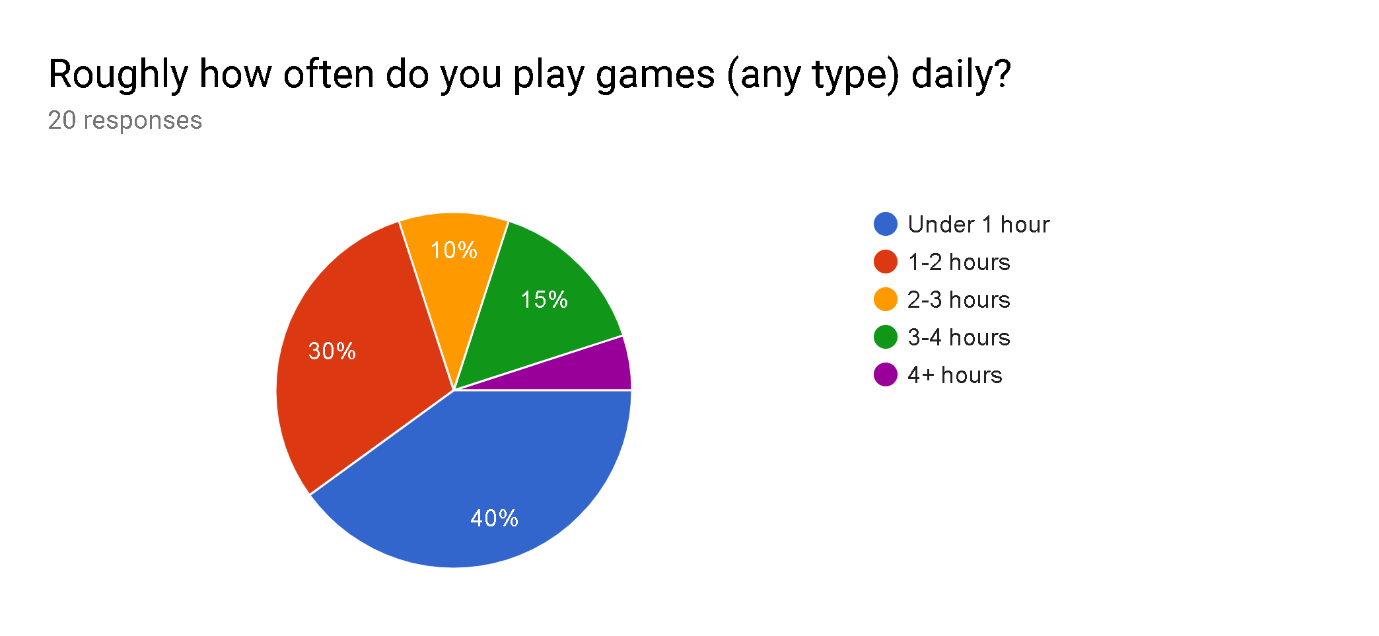


- ‘requires thought on every move’ & ‘because me winning the game depends on me making key plays and outsmarting the opponent’ – this is true with UNO as you must think about every move to enable to get rid of all your cards first

‘it is fun to challenge yourself and fight against other people or play with others’ – this means I will have to focus on making sure the AI I create for single player mode is challenging enough for the user to stay engaged during gameplay, furthermore a multiplayer option against other users (whether on the same computer or across a network) is key to my project’s success



UNO is the second most popular card behind cards of humanity this means that my rendition of UNO has the potential to become popular with a lot of users as the game is already well established and liked.



A significant number of my potential clients only played games from ‘under an hour’ up to ‘2 hours’ per day. This means that most of my users may not have that much experience in gaming which means that a ‘help’ screen with a list of instructions may be necessary, so all users understand how to play UNO. However, this low average of gameplay per day may be due to the fact that it is school term time so gameplay overall will be reduced due to school priorities, because of this fact it will be better if the games are fast-paced and short so users don’t spend too long gaming and have enough time to allocate to other daily activities.

**1.7 Proposed Solution & Objectives**

As this project is to be done over several months below, based on responses from my questionnaire and my own research I have made a rough outline of the objectives I’ve set for myself to complete over the duration of my NEA, this will form my proposed solution to creating a rendition of UNO using python and pygame.

Programming aspect (strongly based around MVC):

**MODEL + CONTROLLER**

Module – Singleplayer

-register account (SQL)

-enter username, email and password

-log in (SQL)

-retrieve user data and display profile (e.g name, date account was created, wins and losses, highest score)

-load **AI**

**START GAME**

Module – Multiplayer

-log in (SQL)

-retrieve user data and display profile (e.g name, date account was created, wins and losses, highest score)

-join or create a lobby

-wait till every player in lobby is ‘ready’ if all players ready, start game

**START GAME**

Module – **start game**

-initialise deck

-distribute cards to players

-when card clicked card placed on pile and update model (event management in pygame)

-if player places card on pile that doesn’t match colour or number, card is returned to player hand and they draw a card

-draw a card from pile

-for every correctly placed card score is incremented by 10

-for every incorrectly placed card/draw from pile, score is decreased by 5

-if player(s) has one card left, press enter to call uno, if uno not called draw card from pile + decrease score by 5

-if player has no cards left, they win. Score +100

-if score is above user highscore, score updated on their profile

-user pressed Q to leave game

Module – **AI**

-define constant rules for ai to follow

-implement Monte Carlo to calculate values and create a tree based on these values that the AI will traverse to win

**VIEW (PYGAME)**

Image list

-individual uno cards

-player icons

Modules/functions

-addscreen function

* My game will start with a main menu screen that allows the user to navigate to different regions of my game including, singleplayer, multiplayer, help, etc
* Cards will be dealt to individual users when the game is run, users will be able to select the amount of cards in play prior to the start of the game
* Users will be keybinds to play the card game
* I will attempt to add a login system that allows the user to create an account and allow the user to view their profile attributes.
* Login system will be checked, if a user creates an account with already used username it will return invalid
* Add music to my game which can be muted if the user wants too
* Ensure that my game is correct Uno card image, and an animation that accompanies user functions, e.g. discarded a key will see the card visually placed in the centre pile
* User’s will be able to play using keyboard input
* Each user account will store total time points, total games played and a win to lose ratio
* There will be a ranking system against other registered users that corresponds to a user’s score and will allow them to compare their ranking against friends.
* When my project is completed, I will test it amongst my family and peers to collect a range of responses with varying ages as a factor to determine the general success of my project as a fully functioning game application.
* For my project's multiplayer I will attempt to implement a client server relationship in which users connected to the same network can play against each other on separate devices, this will be done using sockets. If I am unable to make an online multiplayer function, I will limit multiplayer to one computer, meaning users will have to physically take turns or control their avatar using different sections of a keyboard (e.g. one user playing on arrow keys and the other on WASD).
* I will implement an AI possibly using a pseudo minimax method, which based on a list of available moves I create in UNO, I will give each move a value based on each situation and create a tree (a compilation of moves) for each common situation e.g., if a +2 is placed on the pile, the AI will place a +2 or +4 to avoid having to draw 2 more cards, this move will be set to the highest value as it is the best move in that situation. (Prototypes will be included in documentation after further research). If I am unable to do this I may attempt to hardcode the moves for an AI from scratch using my knowledge of UNO.
* Create a help function that gives the user a rundown of the Uno game and how it works but I will make sure this function is only called if the user needs it to not hinder experienced players
* For the view aspect of my project I will be focusing on using pygame to create screens such as the start up screen and main game screen. I will import default avatar images for each player added to the lobby. I may also add a profile screen, in which the user can view theirs and other players wins, loses and high score as well as total games played (scores will also be used in creating a ranking system for players to add to the competitive nature of my game)
* I will implement an MVC to link my pygame view and python model and controller together

**1.8 OOP Paradigm Prototype v1**

Since UNO is a very complicated game to code, I have decided to format my code in OOP, so it is easier for me to step through my code when I encounter an error and easier for my NEA Supervisor to analyse my code if necessary. I have started off by creating the fundamentals of an UNO game in which my code allows the programmer to create a deck, shuffle the deck, draw cards from the deck and initialise players. I have also created an SQL database in which users can create an account or log into an existing account. Currently I am attempting to connect the SQL database to class Player: so, once the user logs in, they are initialised as a player when they start playing UNO, I will include this in my documented design.

Below is my deck of cards code in OOP:

import random

class Card:

def \_\_init\_\_(self, suit, number):

self.suit = suit

self.number = number

def \_\_unicode\_\_(self):

return self.showcard()

def \_\_eq\_\_(self, other):

if self.suit == other.suit and self.number == other.number:

return True

return False

def \_\_repr\_\_(self):

return self.showcard()

def showcard(self):

if self.number == 10:

number = "Reverse"

elif self.number == 11:

number = "Skip"

elif self.number == 12:

number = "+2"

else:

number = self.number

print("{} {}".format(self.suit, number))

class Deck:

def \_\_init\_\_(self):

self.cards = []

self.builddeck()

def builddeck(self):

for s in ["Yellow","Red","Blue","Green"]:

for n in range(0,13):

self.cards.append(Card(s, n))

def showdeck(self):

for card in self.cards:

card.showcard()

def shuffle(self):

for i in range(len(self.cards) - 1, 0, -1):

random.shuffle(self.cards)

def drawcard(self):

return self.cards.pop()

def discard(self):

return self.hand.pop()

class Player:

def \_\_init\_\_(self, u\_name\_l):

self.hand = []

self.pile = []

self.name = u\_name\_l

def draw(self, deck, x):

for i in range(x):

self.hand.append(deck.drawcard())

#return self #allows player draw multiple cards

def showhand(self):

print("\nYour hand is")

for card in self.hand:

card.showcard()

def showpile (self):

print("'\nCards on the pile are")

for card in self.pile:

card.showcard()

def discard(self): #remove random card - could be used in 'AI'

discard\_card = input("What card do you want to discard? ").split(' ')

suit = discard\_card[0]

if discard\_card[1] == 'Reverse':

discard\_card[1] = 10

number = int(discard\_card[1])

discard\_card = Card(suit, number)

elif discard\_card[1] == 'Skip':

discard\_card[1] = 11

number = int(discard\_card[1])

discard\_card = Card(suit, number)

elif discard\_card[1] == '+2':

discard\_card[1] = 12

number = int(discard\_card[1])

discard\_card = Card

if discard\_card in self.hand:

self.hand.remove(discard\_card)

self.pile.append(discard\_card)

deck = Deck()

#deck.shuffle()

Player1 = Player('David')

#Player1.draw(deck, 5)

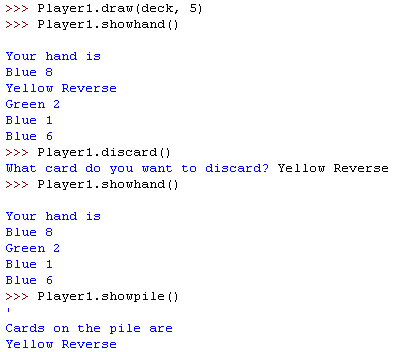
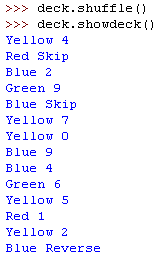
#Player1.showhand()

#Player1.discard()

#Player1.showhand()

#Player1.showpile()

Below is the output of my code, I tested all the necessary functions to ensure it works correctly. So far, my code allows a created player to X number of cards from the main deck (which can be shuffled or unshuffled), see what cards are in their hand, discard a card in their hand onto a pile. However, I still must create an addplayer function which takes user that can logged into an SQL database and add them to the lobby/game. I also have to make the pile accessible to all players before I can continue with making the rest of my UNO game.



**Documented Design**

This section of my write up is an explicit outlining of everything I aim to create for my UNO project based on research from my analysis, it consists of snippets of code, pseudocode and flowcharts. Below are bullet points that explain exactly what I plan to design for each section of my code.

2.1 The Overview

**Login system, start-up menu (and other additional functions)\***

* A Login system that will allow the user to either create a new account or log into an existing one with a username and password. The entered usernames and password will be stored in a SQL database table
* Once a user has logged in there will be a Main Menu screen displayed, on this main menu screen you will be able to select ‘Singleplayer’ button, ‘Multiplayer’ button and ‘Quit’ button
* A help page that will be display a list of commands and corresponding keybinds whenever called by the user
* The user(s) will also be able to pause their game at any point, this will halt the game until the user resumes gameplay
* A user’s total wins will be stored on their profile and this will correspond to a ranking system, the lowest being ‘noob’ and highest being ‘Master’. E.g. one win will equal 10points and a lose will equal minus 5points
* The program will terminate whenever the ‘Quit’ button is pressed using the left mouse click button (this button will only be present on the start up menu screen however the user will be able to quit by pressing <Q> on their keyboard at any time
* The program will progress to the singleplayer function using the left mouse click button
* The user will not be able to see the AI’s hand as to not influence their playing style/strategy

\***the additional functions are highlighted in red**, these functions aren’t necessary essential to the fundamental purpose of my UNO program but will make the game more enjoyable users, hence they will be implemented if enough time is available

**Main Game (Singleplayer)**

* The single player screen will be split into two sides, one side for the user (labelled Player1) and the other for the computer. In the centre of the screen there will be an UNO card.png image displayed that will visually represent the deck of UNO cards that is currently yet to be distributed amongst Player1 and Computer.
* Player1 will initiate the game first by pressing the <Return> Key on their keyboard. This will prompt my code to create a deck of cards, shuffle them and deal them amongst the players (in this case the user and the ‘AI’)
* The dealt cards will be placed in Player1’s hand and the AI’s hand accordingly
* Player1 will take their turn first by pressing the <Space> key on their keyboard. This will prompt a function that will allow the player to select which card they wish to discard, by typing the name when prompted. My program will then run a discard function on this card, visually removing it from the Player’s hand and placing it onto a pile in the middle of the screen
* The ‘AI’ will then take it’s turn by randomly selecting a card from its hand to discard using a ‘random-discard’ function. This card will also be placed on to the pile, on top of the pile in the centre (each player may have their own individual pile as oppose to one central pile that all player’s can access but it’s functionality will remain the same if this is the case)
* The action cards that I have included in my selection of UNO cards are +2, Reverse, and Skip. Each action card will have their own special function that performs exactly what the card intends and visually outputs the result to the screen
* The user (Player1) and the AI will continue discarding cards to the pile. Once the user has one card left in their hand they must ‘call UNO’, this will be done by pressing the <U> key on their keyboard, however this option will have a 5 second timer set to it, if the user calls UNO after this time has elapsed or at any other point during the game, they will have to draw one card and it will be added to their hand.
* The winner of the game is decided by who gets rid of all their cards first. This means that my program will have to check how many cards are remaining, in the user and the AI’s hand after each discard and whoever reaches 0 cards first will be declared the winner. This will stop the game

**Main Game (Multiplayer)**

* The multiplayer screen will have two further buttons, ‘multiplayer local’ and ‘multiplayer online’
* ‘Multiplayer Local’ if the button is clicked on. A new screen will pop up, similar to the singleplayer screen with it being split into two parts.
* ‘Multiplayer Local’ will work the exact same way as singleplayer, however instead of the user playing against an AI, the user will play against another user sharing the same computer/keyboard. This means the two users will have to take it in turns accordingly to play cards in their hand.
* ‘Multiplayer Online’ will use sockets in python to allow two players connect to a server (if they are using the same wifi connection) and play against each other.
* Users will be able

**Quit Game**

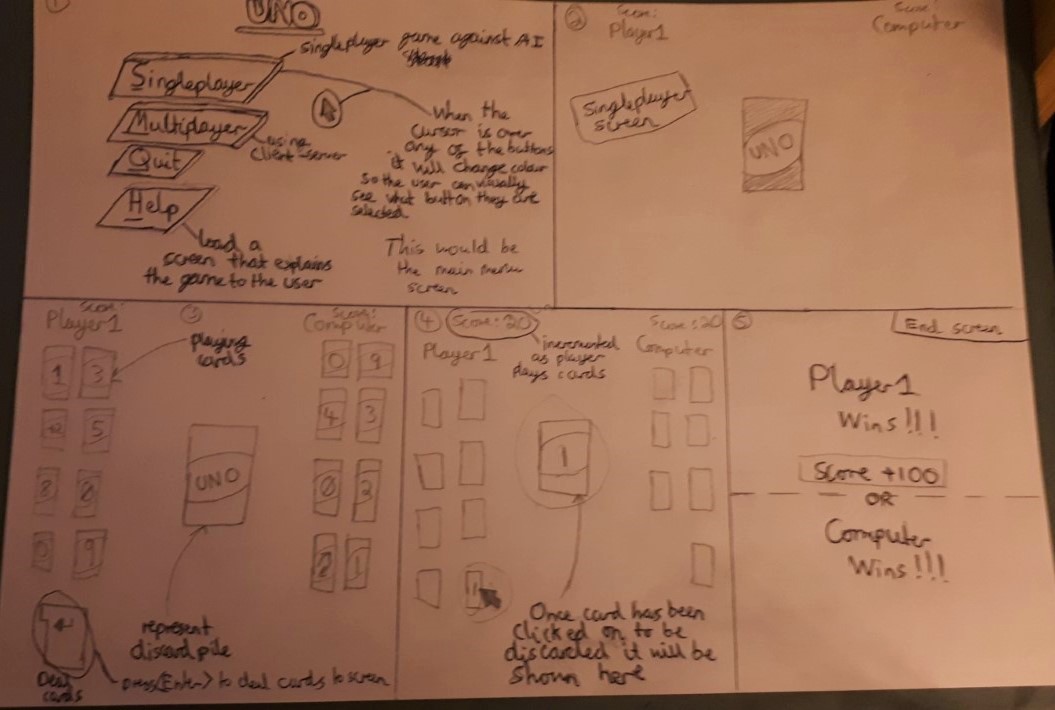
* The user will be able to quit the game by clicking on the ‘Quit’ button on the start up menu screen or pressing the <Q> at any point whilst the program is running.
* If a user quits in the middle of a multiplayer online game, then the opponent automatically wins.

**Help**

* The user will be able to see a list of available commands and their associated keys or clicks, as well as a brief description of how to the play the game

**Graphical Design Outline**

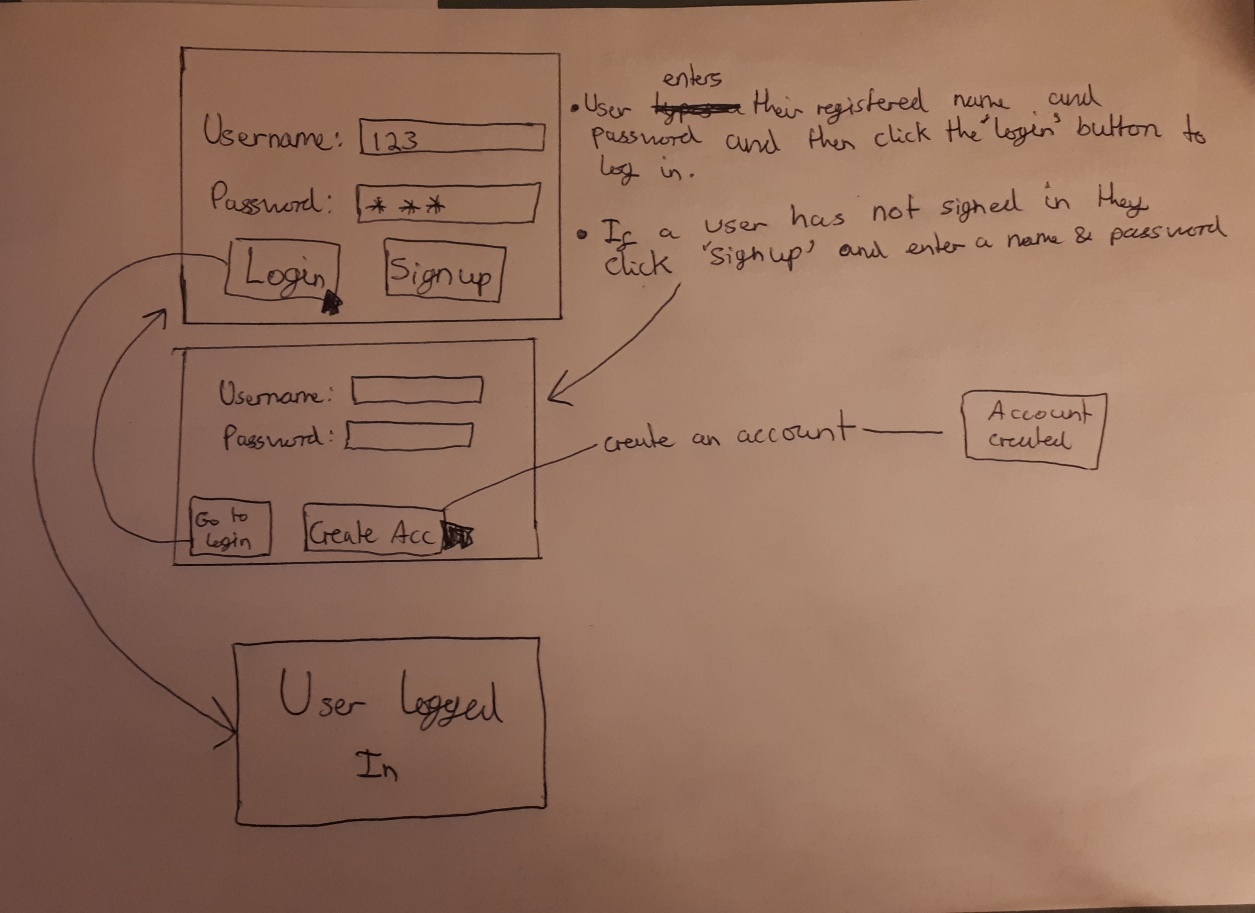
Below are annotated sketches of each major function which visually shows what the user will see on their screen at each main stage of my game



The above image is split into 5 main sections.

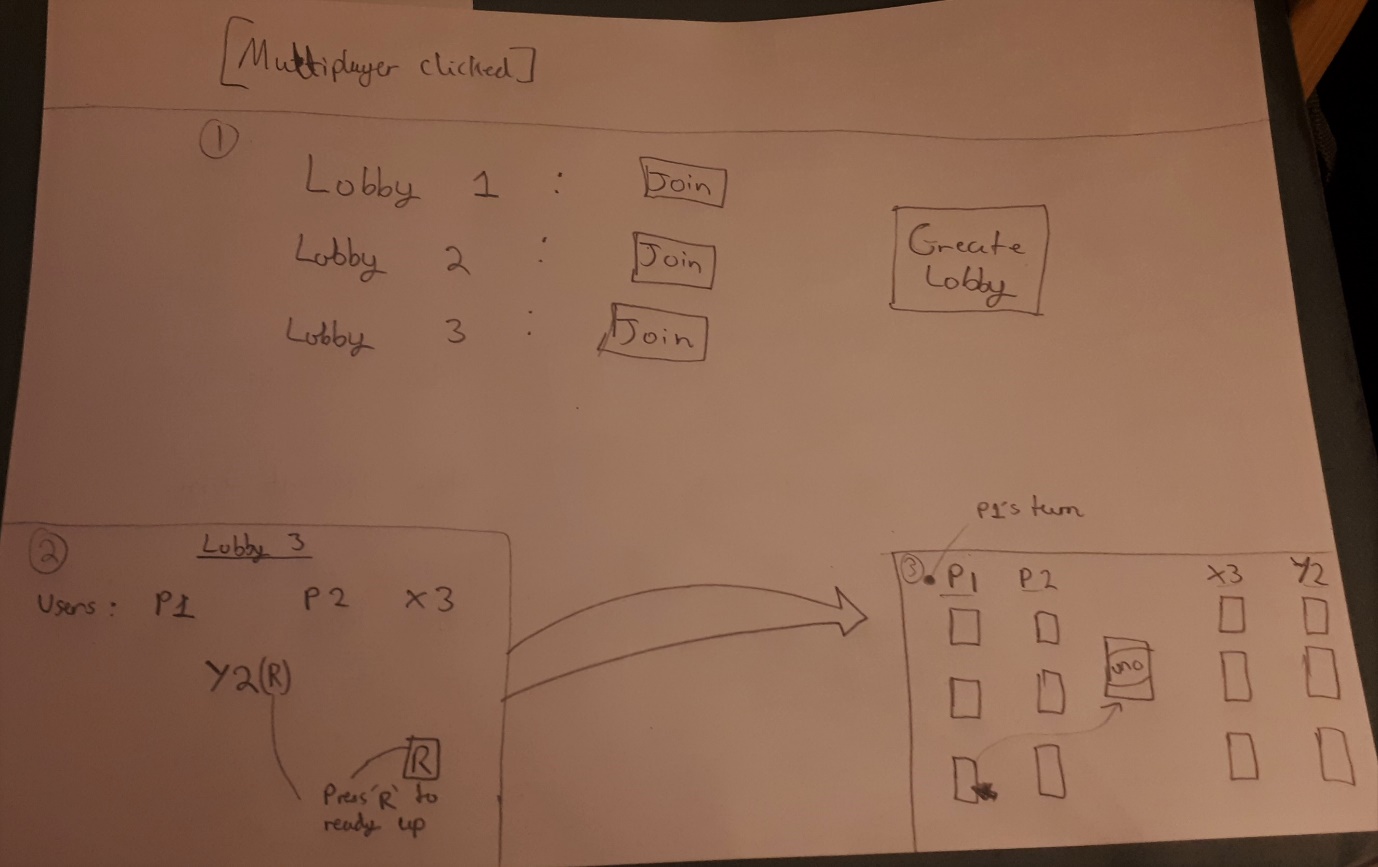
* The first section represents the main menu screen of my game that will load up when the user first runs the game. This screen consists of 4 main overview points listed above (singleplayer, multiplayer, quit and help). The user will be able to click these buttons and the according functions will be carried out
* The second section represents a rough sketch of the singleplayer screen that would be displayed when that button has been clicked. This screen will have a flipped over UNO card in the centre of the screen, to represent an undealt deck of cards. The screen will also be split into Player1 and Computer.
* The third section shows the screen displayed to the player once cards are dealt (cards will be dealt by pressing a key e.g. the ENTER key)
* The fourth section shows the screen once a player plays a card by clicking the card they want to discard from their hand. Once this card is selected it will be discarded and shown on the pile in the centre of the screen. The computer will then discard their own card and it will be shown on the pile. The player’s score will also be incremented by 20 points for discarding a card and this score will be stored on the user’s profile
* The fifth section shows the end screen. Once the user gets rid of all their cards before the computer, they will be presented with a win screen to signify the end of the game. A win will also add 100 points to a user’s score

**Logging In (GUI)**



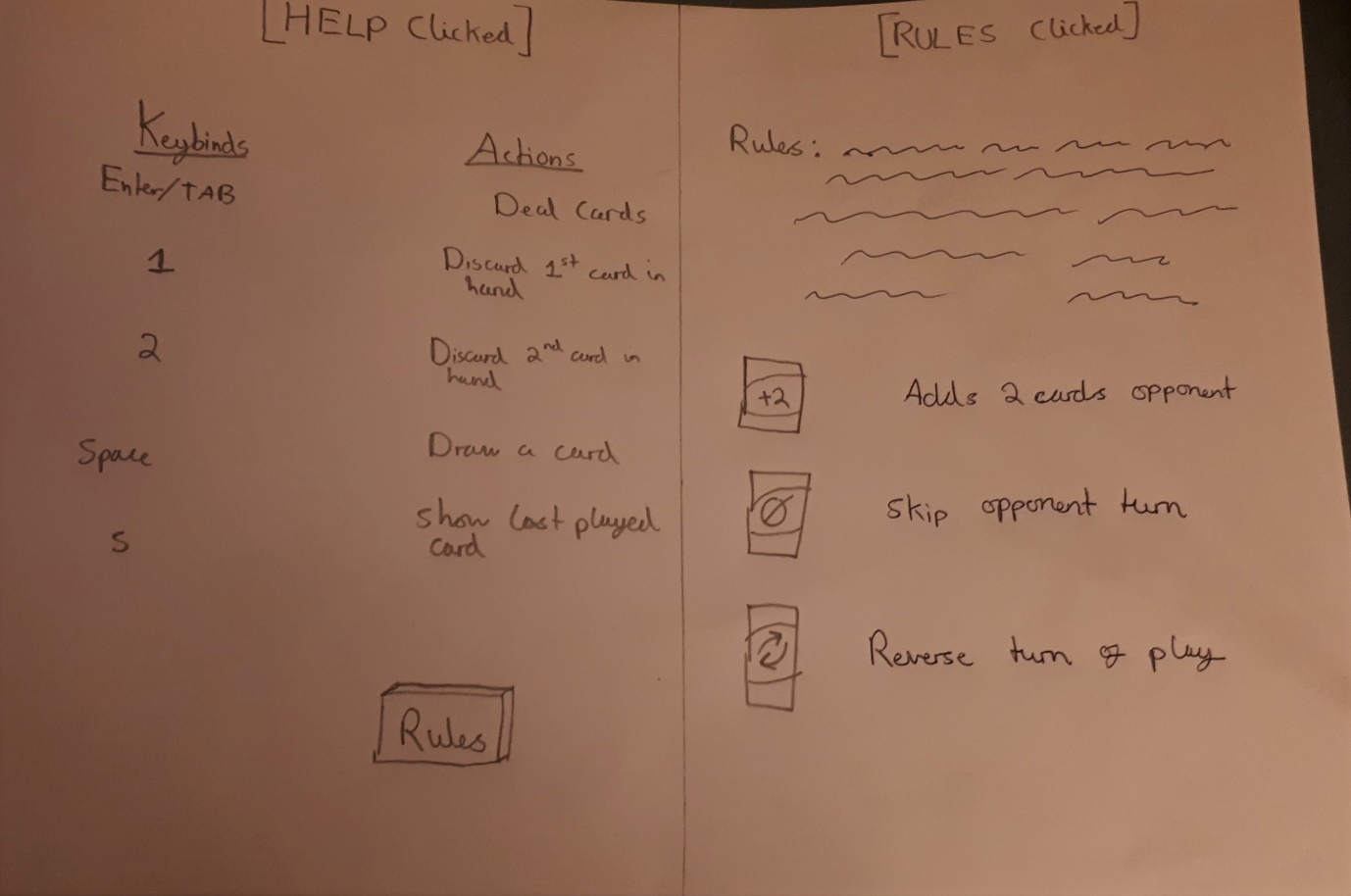
Above is a sketch of my log in/ register system. When the user runs my program, this is what will first be loaded up. A window allowing the user to log in with a registered username and password (the password will be hidden from the screen using ‘\*’ for security). The user will then click the login button to log in and a ‘User Logged In’ window will be displayed to notify the user that their log in was successful. They will then be taken to the UNO Main menu. If the user has not registered they will click the ‘signup’ button and be taken to the sign-up screen in which they will enter a username and password and click ‘Create Acc’ to register. However, if another user has already registered using a similar username an error message will pop saying ‘a user has already registered with that info’.

**Lobby Multiplayer (gui)**



Above is a sketch of the prototype I envisioned for the multiplayer screens. From the main menu (referenced earlier) if the ‘multiplayer’ button is clicked a screen will be displayed with a list of created lobbies by other users with a ‘join’ button next to each lobby and a ‘create lobby’ button on the side. If ‘join’ is clicked the user will be sent to another screen that displays any other user’s in the lobby and each user will have to press the ‘R’ key on their keyboard to be put in a ‘ready’ state. If all users are in a ready state then the game will automatically start and the user will be shown the game screen (referenced to above). **#Multiplayer will be done using a client-server method, using sockets in python (as of 6/12/19 it is yet to be completed but will be worked on before the final submission is due)**

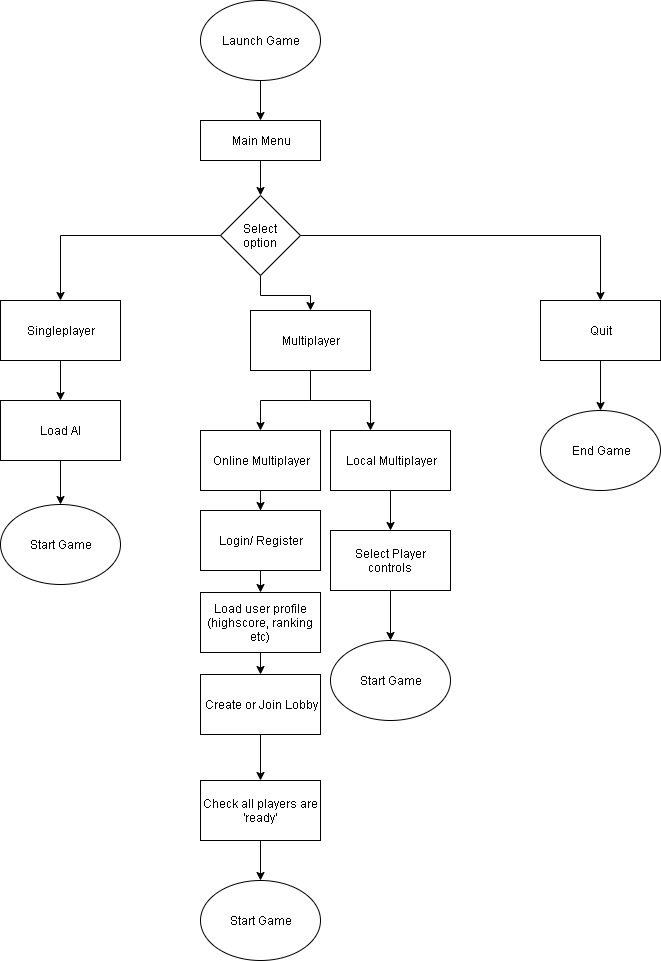
**Help (gui)**



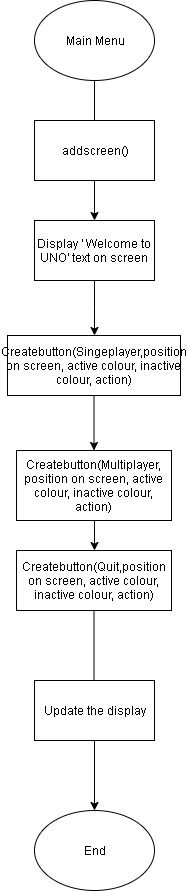
If a user clicks the ‘help’ button the above sketch will be displayed. The left-hand side depicts the help screen which shows the user a list of all the available keybinds in the game and what each keybind does. There will also be a ‘rules’ button at the bottom of the screen, if this is clicked a rules screen will be displayed which explains how UNO works and shows a list of all the action cards and what each one does.

**2.2 Flowcharts**

Below are various flowcharts that depict how I plan to code each significant stage of project.



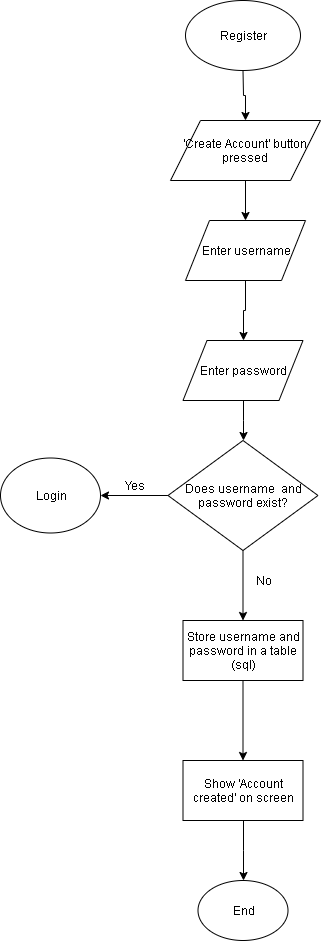
-The main menu aspect (and similarly most of the other GUI driven sections of my project) of the above flow diagram will be created in Pygame. This is done by initialising the pygame built-in methods using pygame.init followed by setting the initial values of the resolution of the window in which my UNO game would be executed (I did this using the arbitrary variables x and y to represent width and height, e.g. x = 800 and y = 800 to create a 800x800 window). A main menu (along with any GUI section) requires different colours to make the game more attractive to the user so I initialised colours in pygame using their RBG values. The Main Menu itself, consists of 3 buttons: ‘Singleplayer’, ‘Multiplayer’ and ‘Quit’ however pygame doesn’t have a built-in method to create buttons so I had to create my own ‘createbutton’ method as well as an ‘addscreen’ function. Below is a flowchart diagram that explains this in more detail.

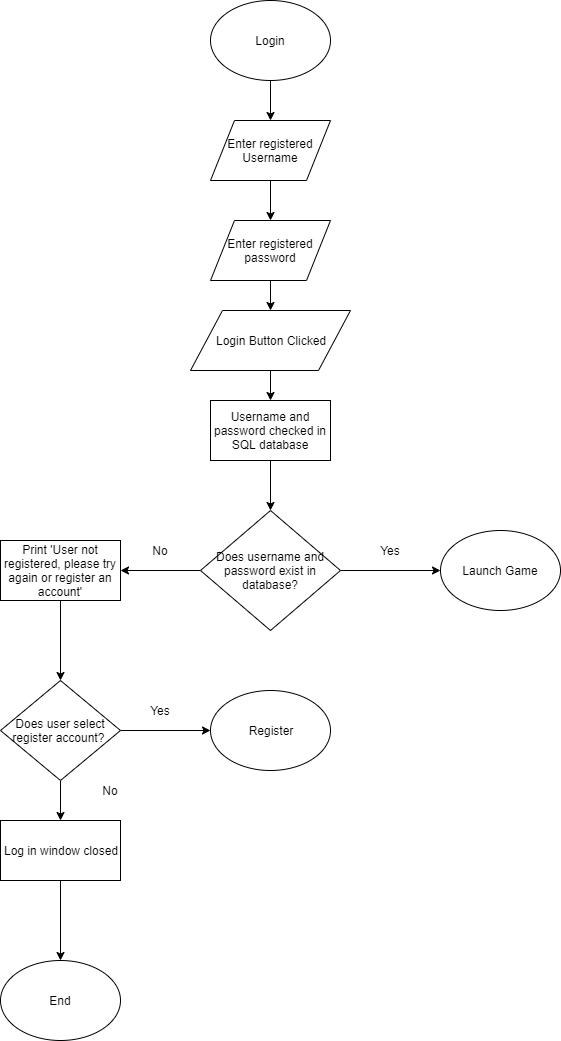


Pygame.display.update() – always has to be run after any changes are made graphically

This createbutton takes the name of the button, the coordinates of each corner of the button on the screen (to create a rectangle that the user can click on), an inactive colour of the button and an active colour of the button so the user knows when their mouse is over the button. This function also recorded the cursor position using pygame.mouse.get\_pos() and mouse input using pygame.mouse.get\_pressed() to see if the user has clicked on the button. If the user does click the button, then an ‘action’ is run (which is simply a call to whatever function you wish to run once the button is clicked)

This function is simply used to create a blank window of a predefined resolution with a white background



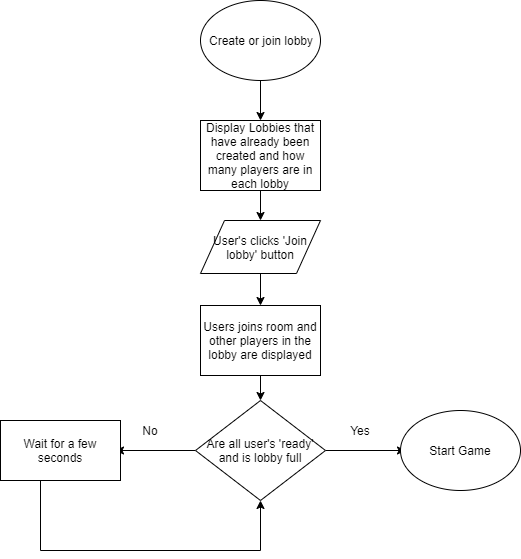
1. ****

   **Validation** - Username and password are for the login and the register function are validated by selecting all stored usernames and passwords in the database table and checking if they match against the newly entered username and password when a user logs in.

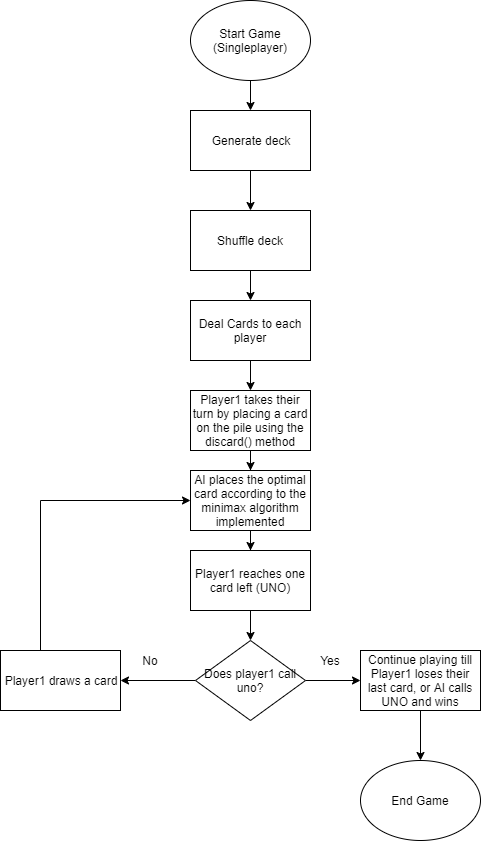
   In the above flowchart I store user data in a table using SQL, this data is stored long term and will need to be accessed at any time by my program to retrieve data.

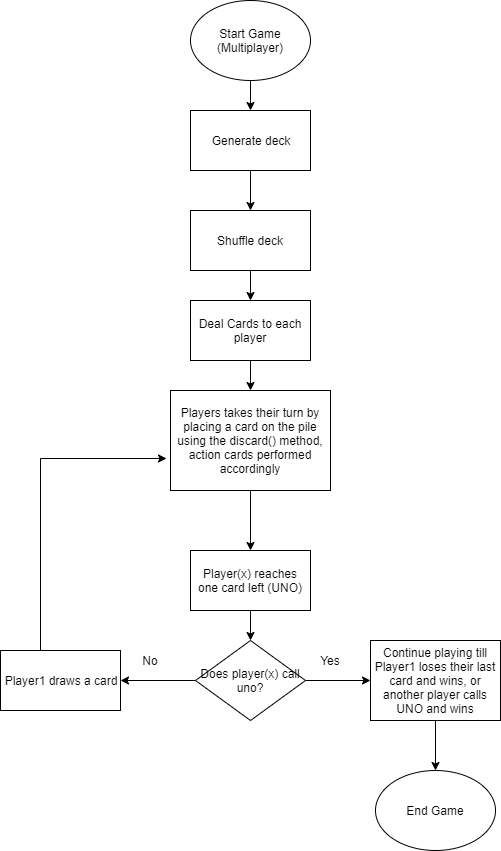
   User Table:

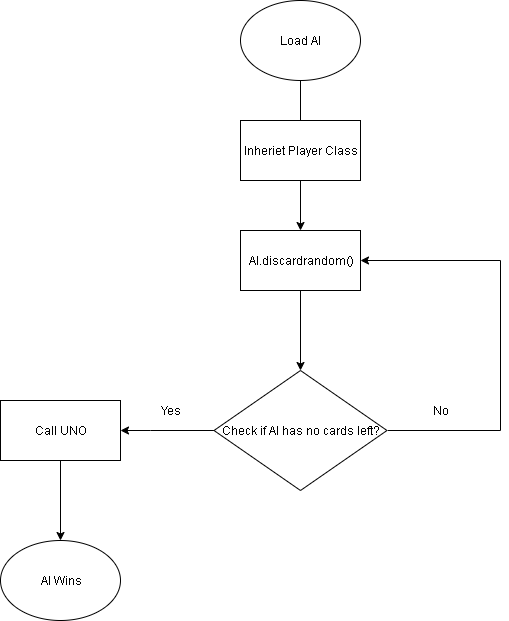
   |  |  |  |  |  |
   | --- | --- | --- | --- | --- |
   | Field Name | Field Type | Purpose | Example | Validation |
   | user\_id | INT | To uniquely identify each user and serve as reference point to fetch a specific users’ data if necessary | 3 | is an integer |
   | username | VARCHAR | Store the username of a user that registers | ‘bob’ | Is not empty |
   | password | VARCHAR | Store the password of a user that registers | ‘bob123’ | Is not empty |
   | high score | INT | Store the value of a user’s high score, initially set to 0 when a user logs in for the first time | 100 | Is an integer |
   | Total Games Played | INT | Stores the value  Of total games played by the user | 4 | Is an integer |
   | Ranking | VARCHAR | Stores the rank of each individual user based on their score. | ‘100IQ’ | Is not empty |

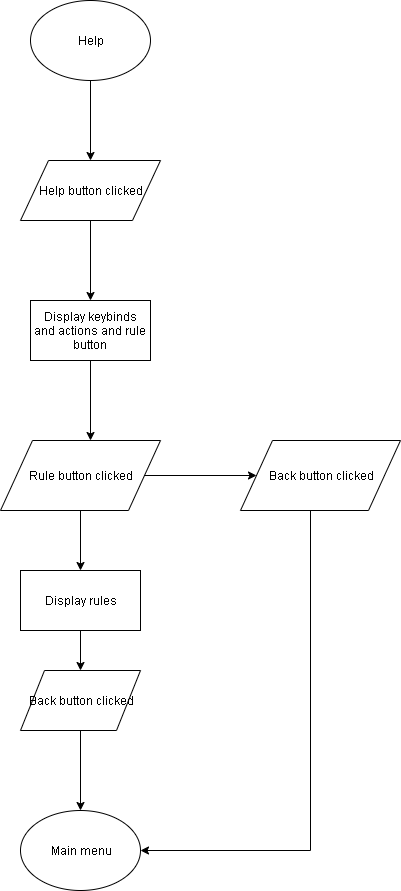
   ****

   Once a user logs in and clicks the multiplayer button. They will be presented with the option of clicking the multiplayer online, or multiplayer offline. ‘Multiplayer Online’ will consist of the above flowchart (creating and joining lobbies). Once the user clicks on ‘multiplayer online’ a list of lobbies will be displayed on the screen. The user will click on a lobby name to join it or click the create lobby button at the button of the screen to create a lobby. Once the max number of users join the lobby (which will be monitored by a counter), the game will start as normal.

   ****

   ****

   **2.3 Core Functionality**

   In this chapter of my documented design I will discuss exactly how each main function of my program will work and highlight any unique programming paradigms or new skills used. To aid with my explanations I will attach small sections of my code (the full code will be present at the end of my documentation)

   I started off by creating the fundamental part of the UNO board game which are the cards. I decided to do this using Object-Oriented Programming as it would make it easier for me to step through my code when I ran into issues & it is a lot easier to edit or update certain aspects. I split the game of UNO into three main sections, a Card, the Deck and the Player(s) and created separate classes for each of them.

   Below is a class diagram and the original code for my Card Class. A single card in UNO (similar to most playing cards) can be split into two fundamental attributes, a suit and a number. In UNO the suit is typically the colour of the card (Red, Yellow, Green or Blue) and the number is the number on the face of the UNO card (0, 1,2,3,4 etc.). So, I used the def \_\_init\_\_ functions to initialise a Card class with each card (self) having the attributes of a suit and a number. My UNO contains 3 action cards in addition to the normal set of playing cards. To keep the number aspect of my UNO cards as integers (so it would be easier to call upon them later) I created a dictionary called vals and gave numbers 10 to 12 the names of action cards. This means when I build my deck of cards later on I can loop from numbers 1 to 12 and the number 10,11 and 12 will be assigned action card titles.

   **Highlighted Code(to create a Card object):** Class Card:

   def \_\_ init\_\_ (self, suit, number):

   self.suit = suit

   vals = {10: “reverse”, 11: “skip”, 12: “+2”}

   if number < 10:

   self.number = number

   else:

   self.number = vals[number]

   self.image = pygame.image.load(str(self) + “.png”)

   def getImage():

   return self.image

   def showcard(self):

   return "{} {}".format(self.suit, self.number)

   Now that I had a means to create a Card, I had to create a function that displayed the card to the screen. The function showcard, simply prints the card to the screen in the format ‘suit\_number’, e.g. If I created a card called Yellow 9 using Card(Yellow, 9), I could then do Card.showcard() to return the card as ‘Yellow\_9’. I also used pygame’s inbuilt function to load a corresponding image for each card I create. This means whenever the showcard function would be called in my program It would also preload an image of the card it was performing the function on. This loaded image could then be used later in various aspects of my program (especially when creating the GUI for my UNO project).

   To distinguish the action cards from the normal numbered cards, I implemented a block of code within the class Card that checks what number the created card has and if that number is a 10 then the number will be changed to a ‘reverse’ action card, if that number is a 11, then it will be changed to a ‘skip’ action card and if that number is a 12, then it will be changed to a ‘+2’ action card. This was done using a dictionary as it’s easier to update the values if necessary.

   The pygame.image.load function is used to search for the card title given by it’s suits and number in my file directory and then load the image into the program. Unfortunately, this method means that only a limited amount of UNO cards can be played with at one time, as loading in too many has caused my program to crash occasionally which is why singleplayer will only be limited to 10 cards per player.

   The next stage of my code is initialising the Deck of cards for my UNO game. The attributes in this class are an array of cards and a builddeck function. The builddeck function works by scanning through an created array of suits and for each suit using the Card class to create a card with the suit and numbers from 0 to 12 (this is where the value dictionary from the Card class comes in and if the number is 10 or more it, uses that number as a key and changes the value accordingly). These created Cards are then appended to an empty cards array and the deck is built.

   class Deck:

   def \_\_init\_\_(self):

   self.cards = []

   self.builddeck()

   def builddeck(self):

   for s in ["yellow","red","blue","green"]:

   for n in range(0,13):

   self.cards.append(Card(s, n))

   The deck is simply shown by calling a showdeck function which prints each card in the deck to the console window. E.g. deck.showdeck() will print the deck

   def showdeck(self):

   for card in self.cards:

   print(card)

   The deck must also be able to be shuffled before they are dealt to players. As each card is now in an array I can import random and using random.shuffle to shuffle the cards in the array. E.g. deck.shuffle() will shuffle the array of cards and calling the showdeck function after will show how they have been shuffled.

   def shuffle(self):

   random.shuffle(self.cards)

   Cards must be drawn from the deck when they are being dealt to players in the game. This is simply done by creating a drawcard function that when calls will pop a card from the array of cards created. Pop is used as it removes and returns the value indicated for future reference.

   def drawcard(self):

   return self.cards.pop()

   Now that I have a functioning deck of cards, I need focus on initialising a player and their attributes. In UNO a player has three main functions available to them, they can draw several cards from the deck, discard cards from their hand back onto the deck/pile and view the current cards left in their hand. To do this I created another Class labelled Player with attributes of the Player’s name and their hand.

   class Player:

   def \_\_init\_\_(self, name):

   self.hand = []

   self.name = name

   As I stated the main functions of a player are to draw, discard and view the current state of their hand while playing. The draw functionality must allow the player to true more than just one card at a time and each card must be placed in the player’s hand. The code below works by drawing cards using the drawcard function created in the Deck class and it will repeat this function until it draws x number of cards from the deck and then place these cards in the player’s hand. E.g. Player.draw(deck, 10) will draw 10 cards from the cards array (deck) and append each card into their hand (stored as an array). Users will be able to select how many cards they wish to have in play at the start of the game through the pseudocode below.

   Numofcardstostart🡪int(input(“How many cards do you want to start with ?: “)

   Player.draw(deck, numofcardstostart)

   This pseudocode will then refer to the code below and carry out the function.

   def draw(self, deck, x):

   for i in range(x):

   self.hand.append(deck.drawcard())

   #allows player draw multiple cards

   The discard function should allow a player to remove a selected card from their hand and play it on to the pile. If the card has any special actions, it should be played once it’s been discarded. For the gui aspect of my UNO when a user presses a certain key it will select a certain card and discard it. E.g. if a user presses number 1 it will discard the first card in their hand array.

   Since my UNO takes inputs from user’s keypresses (using an inbuilt function in pygame that allows you to check if certain keys have been pressed). For my game users will select the card they wish to discard by pressing the number corresponding to the placement of that card in their hand. For instance, if a user wishes to play their 9th card they will press number 9 on their keyboard.

   **To get a user’s keyboard input:**

   In pygame key inputs are registered as events. These events are placed in a list and you can extract an event from this list and check if it’s occurred.

   for event in pygame.event.get():

   if event.type == pygame.QUIT:

   pygame.quit()

   quit()

   elif event.type == pygame.KEYDOWN and event.key == pygame.K\_1:

   down = 0

   discard\_card\_selected()\*

   Above shows the method for how I will check for keyboard input from a user. If a user has pressed the number 1 on their keyboard (which pygame stores at pygame.K\_1). This checks if the event that has occurred is a KEYDOWN event (if a key has been pressed) and if that key is number 1 on the keyboard then it will jump to the discard function with this information. This means that the user will use number keys to play my version of UNO as oppose to using their mouse when selected cards, I chose this method as it would be easier for me to understand when a certain card has been selected and what to do as a result whereas with using mouse click inputs, this can be more tricky.

   \*The discard\_card\_selected() function simply runs the discard function below.

   Below is my discard function. This function first checks if the user has any cards left by checking if the length of the array self.hand (user’s hand) is not 0 (meaning the user has no cards). If the user has cards in their hand then the user can discard a card using a method I created named ‘throwAway’. This ‘throwAway’ method is designed to the follow the standard rules of UNO which means a user can’t be able to discard a card if it doesn’t match the suit or number of the card already placed in the game pile at the centre, if this is the case then the card will be discarded and the game will continue as normal, however if this is not the case then the user will be penalized by having to draw a card from the deck. This was done in my code by initially creating a maingamepile (maingamepile🡪[]) and then through the throwaway algorithm, it will check if there is a card in the maingamepile, if there is then it will compare that card to the card the player has selected to discard (acknowledging any special features of the card if necessary – action cards such as +2).

   In a game of UNO when a user plays a card in their hand, if it is an action card (like a +2, skip or reverse) then the attribute of that card is performed. I implemented this feature in my own game by checking if the card that has been selected to be removed has the number +2, skip or reverse. If this is the case then the necessary action will be carried out in the main game section of my code.

   def discard(self):

         global down

         global skipturn

         global reverseturn

         if len(self.hand) != 0:

           discard\_card = self.hand[down]

           self.throwAway(discard\_card)

       def throwAway(self, discard):

          global maingamepile

          if len(maingamepile) > 1:

           lastcardplaced = maingamepile[-1]

           for card in self.hand:

               if card == discard:

                   if card.number == lastcardplaced.number or card.suit == lastcardplaced.suit:

                       if card.number == "+2":

                           gametext\_display("Computer Draws 2 more cards", 2, 4,15)

                           pygame.display.update()

                           Computer.draw(deck, 2)

                       elif card.number == "skip":

                           skipturn = True

                           gametext\_display("Computer's Turn Will Be Skipped Next Round", 2, 4,15)

                           pygame.display.update()

                       elif card.number == "reverse":

                           reverseturn = True

                           gametext\_display("Computer's Turn Will Be Reversed Next Round", 2,4,15)

                           pygame.display.update()

                       self.hand.remove(card)

                       maingamepile.append(card)

                       break

                   else:

                       print("got no cards")

                       gametext\_display("Invalid move. 1 card added to hand.",2,5,15)

                       pygame.display.update()

                       Player1.draw(deck, 1)

                       break

          else:

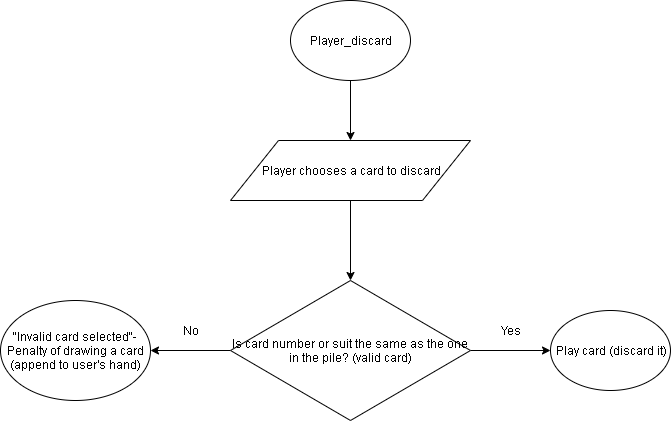
           for card in self.hand:

              if card == discard:

               self.hand.remove(card)

               maingamepile.append(card)

   **Player Discard Decision Tree**

   ****

   Finally, every user must be able to see their own hand displayed on the screen. To do this, the section of code below is created to loop through each card in a player’s hand (self.hand) and display it to the screen using a \***displayimage** method I created to make adding things to my UNO screen easier. The ‘h’ variable is used to systematically shift the position of the next card being displaying by a certain amount, this enables the user to view all the cards in their hand at once without any drastic overlapping between each card.

   Once a user has no cards left in their hand, then they’ve completed the game’s objective and have won the game. To check for this instance I added a section to my showhand function that checks if the length of created array (self.hand) is 0 and if this is the case then my game will display that the Player has won across the screen (using a \*\***gametext\_display** method I created) and play music to signify the victory.

   def showhand(self):

         print("{} Hand is: ".format(self.name))

         h=30

         for card in self.hand:

   **displayimage**(card.image,div\_iwidth,div\_iheight-h)

             pygame.display.update()

             h=h+25

         if len(self.hand) == 0:

           Player1wins = True

           add\_screen()

   **gametext\_display**("Player1 won. Game Over", 2, 2, 40)

           pygame.mixer.music.load("winnermusic.mp3")

           pygame.mixer.music.play(-1)

           pygame.display.update()

           time.sleep(5)

           pygame.quit()

           os.\_exit(1)

   \*def displayimage(image\_name,div\_iwidth, div\_iheight):

       iwidth = x-div\_iwidth

       iheight = y-div\_iheight

       uno\_window.blit(image\_name, (iwidth,iheight))

       time.sleep(0.3)

   Above is a method I created from scratch to display a preloaded image (using pygame inbuilt function pygame.image.load). This function takes the name of the image saved and the position you want to place the image according to the size of the window which is set to 800x600. This function will then calculate the position you want to play the image in and using pygame’s built in function x.blit(name, position) it will place the image of the surface of my pygame window.

   \*\* def gametext\_display(text,divby\_x,divby\_y,fontsize):

       gametext = pygame.font.Font('freesansbold.ttf', fontsize)

       TextSurf,TextRect = text\_objects(text, gametext)

       TextRect.center = ((x/divby\_x) ,(y/divby\_y))

       uno\_window.blit(TextSurf, TextRect)

   The above method is similar to the displayimage method but takes the extra parameter of fontsize. This function takes the text I wish to place on the screen, make it into a rectangle object and places it in the specified position.

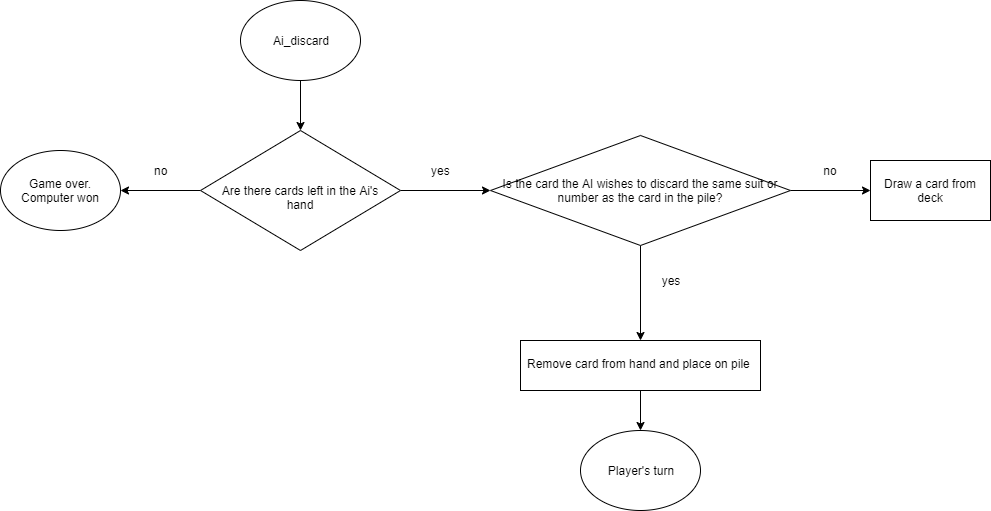
   I created these two functions to make programming my game more efficient and easier to customise as pygame does not have any built-in functions to display text or images.

   **2.4 AI**

   Using inheritance, I created a Class for my AI that linked directly to my Player class (as my AI would share most of the original functions a Player has). This means that my AI can also draw cards, discard cards and display their hand. This also means all the methods I made for the player will be used when I type AI instead of Player unless I override certain methods. The main aspect of the AI is seen when they are discarding cards as the AI must follow the rules of UNO using a decision tree algorithm that checks if the card it wishes to play is the same suit or number as the card that is already in play from the Player. If this is not the case, then the Computer will draw a card to add to its hand. I initially started with an 'AI' that randomly discards cards but later improved it to this.

   **AI Discard Decision Tree**

   Below is a decision tree that outlines my AI current method of ‘thinking’ when playing the game.

               for i in range (len(self.hand)):

               ai\_card = self.hand[i]

               if ai\_card.suit == lastcardplaced.suit or ai\_card.number == lastcardplaced.number: #checks if card the computer wants to discard is the same suit or number as the card the player first discarded

                   print("computer discarded", ai\_card)

                   self.aithrowAway(ai\_card)

                   if ai\_card.number == "+2":

                       gametext\_display("Player1 Draws 2 more cards", 2, 5,15)

                       pygame.display.update()

                       Player1.draw(deck, 2)

                   elif ai\_card.number == "skip":

                       gametext\_display("Player1's Turn Skipped", 2, 5,15)

                       pygame.display.update()

                       Computer.discard()

                   elif ai\_card.number == "reverse":

                       gametext\_display("Player1's Turn Reversed", 2,5,15)

                       pygame.display.update()

                       Computer.discard()

                   break

               else:

                   gametext\_display("Computer draws a card", 2, 5, 15)

                   print("oop")

                   Computer.draw(deck, 1)

                   break

               break

   def aithrowAway(self, discard):

           for card in self.hand:

               if card == discard:

                 self.hand.remove(card)

                 maingamepile.append(card)

   Originally, I was going to create an AI using a Minimax algorithm however as I did further research into the algorithm itself I realised it may not work well for a game like UNO as the moves for a player are not as set and rigid as they are for a game like Chess or tic tac toe where every move has a set rule too it. I don’t believe it is feasible to create an AI using this method under the current time constraints (as I wish to focus on the Multiplayer aspect of my game as this was deemed more important by my targeted users -evidenced in my Analysis Questionnaire page 15) however as I stated in my Analysis if this method was not possible I would attempt to hardcode an AI using the rules of UNO and popular strategies which is what the above code mainly does. Currently this is the only AI I have however I still aim to improve this AI and if possible implement a self learning aspect that means as the AI continues playing it will increase in difficulty, this will be manually hardcoding into the game.

   **2.5 GUI Aspect**

   As seen in the research from my analysis (in the user questionnaire). GUI is quite an important aspect of an UNO game, or any game for that matter. This means I had to focus on the GUI aspect as well as the core functionality. I did this by using pygame to create the initial sketches and rough outlines of screens that I made (referenced to on pages 27-30).

   **Main menu:**

   As depicted in my sketch my main menu must have 4 buttons, singleplayer, multiplayer, quit and help. Clicking on each button will result in the appropriate action being performed, e.g. clicking singleplayer will take the user to the singleplayer screen against the ‘computer’.

   def startup\_menu():

   **add\_screen()**

       pygame.mixer.music.load("Menumusic.mp3")

       pygame.mixer.music.play(-1)

       startup = True

       while startup:

           for event in pygame.event.get():

               if event.type == pygame.QUIT:

                   pygame.quit()

                   quit()

           gametext\_display('Welcome to UNO',2,2,50)

           singeplayer\_button = **createbutton**('SINGLEPLAYER',50,450,160,50,green,orange,singleplayer)

           mutliplayer\_button = **createbutton**('MULTILPLAYER',250,450,160,50,blue,orange,multiplayer)

           quit\_button = createbutton('QUIT',450,450,100,50,red,orange,quitgame)

           help\_button = createbutton('HELP', 600,450,100, 50, white, orange,help\_screen)

           pygame.display.update()

   Here is the code dedicated to the main menu of my UNO game. Using a **add\_screen()** method I made it simply loads a screen of specified size and loads the background image on to the screen.

   Pseudocode for add\_ screen():

   white🡪(255,255,255) #initialises the colour white to be used later

   Gamewindow 🡪 pygame.display.set\_mode((X , y) #x and y are the resolution of the screen

   Gamewindow.fill(white) #fills the background with white

   Backgroundimage🡪Pygame.image.load('background.png’) #loads the image into the program to do accessed at any point later

   Gamewindow.blit(Backgroundimage, X, y) #attaches the image to the white background this making it the new background image

   Pygame.display.set\_caption('Game Name’)

   Pygame.display.update()

   It then starts playing music to add to the user enjoyment of the game (using pygame’s inbuilt mixer function) and begins to display text and buttons to the screen to welcome my user to the game. As pygame does not have an inbuilt function to create buttons like tkinter I used a tutorial online (linked here <https://pythonprogramming.net/pygame-button-function/>) to learn how to make a function for clickable buttons. The function allows me to define the size of the button, the original colour of the button and the colour of the button when it’s being hovered over along with an action to be performed if the button is clicked.

   def **createbutton**(button\_name,x1,y2,w1,h2,inactive\_colour,active\_colour,action=None):

       mouse = pygame.mouse.get\_pos()

       click = pygame.mouse.get\_pressed()

       if x1+w1 > mouse[0] > x1 and y2+h2 > mouse[1] > y2:

           pygame.draw.rect(uno\_window, active\_colour,(x1,y2,w1,h2))

           if click[0] == 1 and action != None:

               action()

       else:

           pygame.draw.rect(uno\_window, inactive\_colour,(x1,y2,w1,h2))

       button\_font\_size = pygame.font.Font("freesansbold.ttf",20)

       textSurf, textRect = text\_objects(button\_name, button\_font\_size)

       textRect.center = ( (x1+(w1/2)), (y2+(h2/2)) )

       uno\_window.blit(textSurf, textRect)

   **Singleplayer**

   If the singleplayer button is clicked, then it goes the function named singleplayer and execute code from there. The singleplayer function should display a similar screen to the sketch I referred to earlier (with an UNO card image in the centre and Players on either side). Below is the section of my code that carries out this task. The below code also records the keyboard input from the user and runs functions accordingly (using pygame.events). This means only certain keys will work when playing the game and this also allows me and anyone else who navigates my code to easily understand what each game key does and what exactly the output should be.

   Below are all the recorded inputs that a user will use within my game and the corresponding outputs that the user will see on the screen (these inputs will also be present in the help window).

   |  |  |  |
   | --- | --- | --- |
   | **Input (storage if any)** | **Process/Function** | **Output (storage if any)** |
   | <TAB > Key Press | Dealing cards from deck to players in the game | Displays an image of each card as they are dealt to each individual player on their side of the screen |
   | <H> Key Press |  | Displays a list of key binds and commands as a separate window to the user |
   | <SPACEBAR> Key Press | Draw a card from the deck | Visually deck a card from the deckand places it in the user’s hand |
   | <D> Key Press | Draw a card from the pile | Visually removes the card from the pile and places it back in the player’s hand that presses this key |
   | <Left Mouse Click> Key Press | Clicking a button which progresses to the next screen, e.g. clicking ‘single player’ | Displays the next screen when a specific button is clicked |
   | <1> key press | Discards the first card in the player’s hand and places it on the pile | Card will be visually removing from hand and shown on the pile |
   | <P> Key Press | Pauses the current state of the game | Display ‘pause screen’ on the screen |
   | <S> Keypress |  | Displays last card placed on the pile |
   | <2> key press | Discards the second card in the player’s hand and places it on the pile | Card will be visually removing from hand and shown on the pile |
   | <3> key press | Discards the third card in the player’s hand and places it on the pile | Card will be visually removing from hand and shown on the pile |
   | <x number> key press | Discards the x card in the player’s hand and places it on the pile | Card will be visually removing from hand and shown on the pile |

   The GUI for my singleplayer game will be similar to my sketch outlined on page 28. With the screen being split into the Player’s side and the Computer’s side and a card in the middle this will be done using the gametext\_display and displayimage functions I created (referenced to on pages 44).

   **Multiplayer**

   As I have not completed the code for multiplayer yet I will add to this section later, however my aim is that my multiplayer GUI will be the similar to my singleplayer GUI above. When a user clicks the multiplayer button they will be presented with 3 lobbies to connect to (for two users to play together they will have to connect to the same lobby). Once a user clicks on a lobby they will be placed inside the lobby and will have to press R on their keyboard to ready up. Once all players are ready the game will start and the user will see a similar screen to the singleplayer screen only the 'Computer' will now be 'Player2'.

   Def multiplayer\_clicked():

   Createbutton('Lobby 1’, position, joinlobby)

   Createbutton ('Lobby 2’, position, joinlobby)

   Createbutton ('Lobby 3’, position, joinlobby)

   Class Lobby (self, Players, lobbynumber)

   Self.players = Players

   Self.lobbynumber = lobbynumber

   Def joinlobby():

   add\_screen()

   Numofplayersinlobby 🡪 0

   Numofplayersinlobby🡪 numofplayersinlobby +1

   gametext\_display('Press R to ready up', position)

   For event in pygame.events:

   Pygame.events.get()

   If event.type in pygame.events == pygame.KEYDOWN and event.key == pygame.K\_r:

   Playerstatus == (‘Ready')

   Numofplayersrdy🡪numofplayersrdy+1

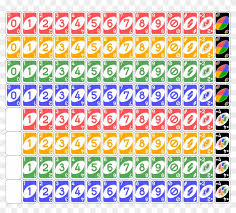
   If numofplayersrdy 🡪numofplayersinlobby:

   Start\_multiplayergame()

   **To Be Completed**

   **2.6 Creating card images**

   My UNO game also required UNO specific card images. To do this I downloaded a sprite image of all UNO cards from google images.

   I then used Photoshop CS6 to separate the UNO sprite image into the individual UNO cards I needed. Finally I renamed each image to the name of the UNO card, this meant I could reference the image in my code (as long as they are in the same directory) and I could load images and display them to the screen.

   For example, this would be the separated UNO card image for the card RED 1. Once separated it would be saved as red 1.png and this means when I initialised my cards in my code (in Class Card and Class Deck, pages 40-41) whenever I called upon the Card (red, 1) it would be associated with this image. This also meant I could easily load the image on to the screen using pygame.load.image(‘red 1.png’)

   **2.7 Logging in Function**

   **Storing user data**

   As mentioned earlier in my analysis one of my objective was to allow users to log in to their account and access their ranking, high score and total games played. Users will also be able to sign up if they haven’t made an account before in before playing UNO. To do this I used SQL as I am more comfortable with the language in comparison to CSV. I used the following commands:

   **With sqlite3.connect(databasename.db) as db:**

   **c 🡪 db.cursor**

   This is just used to initialise the connection between my program and SQL

   **CREATE TABLE:** Used to create a table name users to store user's name and password and ID

   c.execute(‘CREATE TABLE IF NOT EXISTS users (user-id INTEGER PRIMARY KEY, username TEXT NOT NULL, password TEXT NOT NULL;)')

   **SELECT \* FROM USERS:** Used to select all the data stored in the users table

   Once a user enters a username and password my code will have to check if it exists in the user table, if it does they will be logged in but if not they should receive an error and be exited from the game. This is done by using SELECT \* FROM USERS and checking the inputted data against the selected data. A pseudocode example of this is below.

   def login():

   find\_user 🡪 ('SELECT \* FROM users WHERE username = ? and password = ?'

   )

   If username and password in find\_user:

   Result 🡪 True

   if result:

   login 🡪True

   uno\_gui()

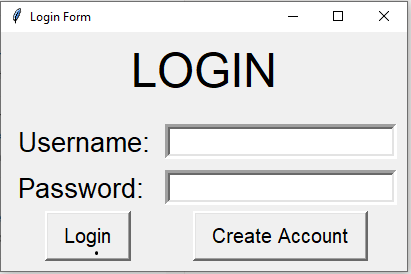
   else:

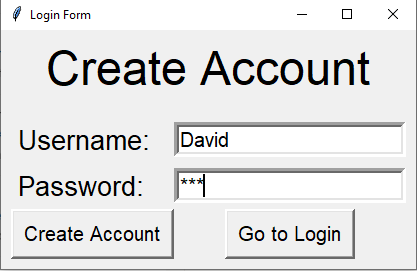
   Print('Username Not Found.')

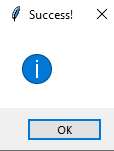
   The same function will be used to check if a username has already been registered on the database when a user signs in.

   **GUI for log in system**

   The GUI for my log in system will be handled using tkinter as they have supported use of widget and it’s easier to add buttons and error messages using tkinter in comparison to pygame. Below is the output of my log in system for each scenario.

   ****This is the initial login window

   Example of entered data. Username: David. Password: lol

   Once the user clicks create account this window will be displayed. Then the user will log in with the same data they entered to register and once they click log in, the game will load as normal.

   **2.9 Technical Solution**

   Below is a table that states key algorithms used in my code, what band they are in and the highlighted code as evidence.

   |  |  |  |
   | --- | --- | --- |
   | **Algorithm/Model** | **Group** | **Highlighted Code** |
   | List Operations | A | for s in ["yellow","red","blue","green"]:              for n in range(0,13):                  self.cards.append(Card(s, n))  for i in range(x):              self.hand.append(deck.drawcard())              #allows player draw multiple cards |
   | Complex user-defined use of OOP e.g. (classes, inheritance etc.) | A | class Card:      …  class AI(Player):      …  #login class for uno  class Unologin:      def \_\_init\_\_(self,master):          # Window          self.master = master          # Some Usefull variables          self.username = StringVar()          self.password = StringVar()          self.n\_username = StringVar()          self.n\_password = StringVar()          #Create Widgets          self.widgets() (**PAGE 66**) |
   |  |  |  |
   |  |  |  |
   |  |  |  |
   |  |  |  |
   |  |  |  |

   Below is a copy of my current full code with comments to explain all major functions

   import random

   import pygame

   import os

   import time

   from tkinter import \*

   from tkinter import messagebox as ms

   import sqlite3

   from sys import exit

   #imported modules necessary for game's function

   pygame.mixer.pre\_init(44100,16,2,4096) #initialises pygame mixer for music

   pygame.init() #initialises pygame

   skipturn = False #variable used to check if a skip action card has been played

   reverseturn = False #variable used to check if a skip reverse card has been played

   Player1wins = False  #variable used to check if Player has won

   Computerwins = False   #variable used to check if Computer has won

   Score = 0 #plcaeholder for score value for users (yet to be added t table)

   x = 800 #screen resolution width

   y = 600 #screen resolution height

   div\_iwidth = 750 #adjustment to width for dispalying images

   div\_iheight = 550 #adjustment to height for dispalying images

   white = (255,255,255)

   black = (0,0,0)

   green = (0,255,0)

   red = (255,0,0)

   blue = (0,0,255)

   orange = (255,165,0)  #predefined colours

   mouseposition = pygame.mouse.get\_pos() #gets mouse position

   class Card:

       def \_\_init\_\_(self, suit, number): #Card given the attribute Suit and Number

           self.suit = suit

           vals = {10: "reverse", 11:"skip", 12:"+2"} #dictionary to store action card values paired with number keys

           if type(number) is int:

               if number < 10:

                   self.number = str(number)

               else:

                   self.number = vals[number] #used when building the deck of cards, to create action cards using above dictionary

           self.image = pygame.image.load(str(self) + ".png") #load each creating image into the program according to the card name

       def \_\_eq\_\_(self, other):

           if self.suit == other.suit and self.number == other.number:

               return True

           return False

       def \_\_repr\_\_(self):

           return self.showcard()

       def getImage():

           return self.image

       def showcard(self):

           return "{} {}".format(self.suit, self.number) #returns value of a created Card in a printed format

   class Deck:

       def \_\_init\_\_(self):

           self.cards = [] #an array stored all the card values

           self.builddeck()

       def builddeck(self):

           for s in ["yellow","red","blue","green"]:

               for n in range(0,13):

                   self.cards.append(Card(s, n))   #builds a deck by pairing each colour with numbers from 0 to 12, then converting them into a Card object and adding them to an array of card

       def showdeck(self):

           for card in self.cards:

               print(card)  #when showdeck is called, it will display all cards in the self.cards array

       def shuffle(self):

           random.shuffle(self.cards) #uses random to shuffle order of cards array

       def drawcard(self):

           return self.cards.pop()  #removes last card in cards array and returns the value

   class Player:

       def \_\_init\_\_(self, name):

           self.hand = [] #array for player's hand

           self.name = name

       def draw(self, deck, x):

           for i in range(x):

               self.hand.append(deck.drawcard())

               #allows player draw multiple cards and appends them into hand array

       def showhand(self):

         print("{} Hand is: ".format(self.name))

         h=30

         for card in self.hand:

             print(card)

             displayimage(card.image,div\_iwidth,div\_iheight-h) #display's each card in a player's hand on to the game screen

             pygame.display.update()

             h=h+25 #moves each following card down by 25 to ensure the user can see all the cards

         #if len(self.hand) == 1:

         #   global UNO\_called

         #   while UNO\_called == False:

         #      gamtext\_display("Press U to call UNO", 2,4,15)

          #     print("call uno thing")

          #     #pygame.display.update()

          #     if UNO\_called == True:

          #         gametext\_display("UNO CALLED", 2,5, 15)

          #         break

          #  pygame.display.update()   UNO CALLED FUNCTION NOT WORKING, WILL FIX

         if len(self.hand) == 0:

           Player1wins = True

           add\_screen()

           gametext\_display("Player1 won. Game Over", 2, 2, 40)

           gametext\_display("Score: +100 ", 2,4,40)

           pygame.mixer.music.load("winnermusic.mp3")

           pygame.mixer.music.play(-1)

           pygame.display.update()

           time.sleep(5)

           pygame.quit()

           os.\_exit(1)

           #once a player has no cards in their hand, Player has won, displays win screen and plays winner music

       def discard(self):

         global down

         global skipturn

         global reverseturn

         if len(self.hand) != 0:

           discard\_card = self.hand[down]

           self.throwAway(discard\_card) #if the player has cards left in their hand, it will discard the card based on their keyboard input using throwAway

       def throwAway(self, discard):

          global maingamepile

          if len(maingamepile) > 1: #checks if there is a card or cards in play already

           lastcardplaced = maingamepile[-1] #if there is then the last card on the pile = the last card placed

           for card in self.hand:

               if card == discard:

                   if card.number == lastcardplaced.number or card.suit == lastcardplaced.suit: #checks if card selected to be discarded has the same suit or number as the last card in play on the pile

                       if card.number == "+2":

                           gametext\_display("Computer Draws 2 more cards", 2, 4,15)

                           pygame.display.update()

                           Computer.draw(deck, 2) #if card is a +2, computer gets 2 more cards

                       elif card.number == "skip":

                           skipturn = True

                           gametext\_display("Computer's Turn Will Be Skipped Next Round", 2, 4,15)

                           pygame.display.update() #if card is a skip, computer's turn will be skipped next in the main game loop

                       elif card.number == "reverse":

                           reverseturn = True

                           gametext\_display("Computer's Turn Will Be Reversed Next Round", 2,4,15)

                           pygame.display.update() #if card is a reverse, computer's turn will be reversed next in the main game loop (effectively player gets another free turn)

                       self.hand.remove(card) #card is removed from player's hand

                       maingamepile.append(card) #card is added on to main game pile

                       break

                   else:

                       invalidturn = True

                       gametext\_display("Invalid move. 1 card added to hand.",2,5,15)

                       pygame.display.update()

                       Player1.draw(deck, 1)

                       break  #if the user selects a card that isn't the same suit or number the card is invalidly played and they will face a draw card penalty

          else:

           for card in self.hand:

              if card == discard:

               self.hand.remove(card)

               maingamepile.append(card) #if there are no cards in play on the main game pile then the card the user selects will be discarded as normal (as this means the user is starting the game first)

   class AI(Player):

       def discard(self):

           global maingamepile

           lastcardplaced = maingamepile[-1]

           print("lastplayed card is", lastcardplaced)

           for i in range (len(self.hand)):

               ai\_card = self.hand[i]

               if ai\_card.suit == lastcardplaced.suit or ai\_card.number == lastcardplaced.number: #checks if card the computer wants to discard is the same suit or number as the card the player first discarded

                   print("computer discarded", ai\_card)

                   self.aithrowAway(ai\_card)

                   if ai\_card.number == "+2":

                       gametext\_display("Player1 Draws 2 more cards", 2, 5,15)

                       pygame.display.update()

                       Player1.draw(deck, 2)

                   elif ai\_card.number == "skip":

                       gametext\_display("Player1's Turn Skipped", 2, 5,15)

                       Computerskip = True

                       pygame.display.update()

                       Computer.discard()

                   elif ai\_card.number == "reverse":

                       gametext\_display("Player1's Turn Reversed", 2,5,15)

                       Computerreverse = True

                       pygame.display.update()

                       Computer.discard()

               else:

                   gametext\_display("Computer draws a card", 2, 5, 15)

                   Computer.draw(deck, 1)

                   break

               break

               #works similarly to the Player discard function. Check card suit, number and follows UNO rules accordingly

       def aithrowAway(self, discard):

           for card in self.hand:

               if card == discard:

                 self.hand.remove(card)

                 maingamepile.append(card) #removes card from hand and places in main game pile

       def showhand(self):

           print("{}'s Hand is: ".format(self.name))

           h=30

           for card in self.hand:

               print(card)

               displayimage(deckImg,div\_iwidth-600,div\_iheight-h) #parameters altered slightly to images displayed on the right hand side of the screen

               pygame.display.update()

               h=h+25

           if len(self.hand) == 0:

               Computerwins = True

               add\_screen()

               gametext\_display("Computer won. Game Over", 2, 2, 40)

               gametext\_display("Score: +100 ", 2,4,40)

               pygame.display.update()

               time.sleep(5)

               pygame.quit()

               os.\_exit(1)

               #similar to Player showhand

   deck = Deck() #initialises deck

   maingamepile = [] #initialises maingamepile

   Player1 = Player('Player1')

   Computer = AI('Computer')

   #GUI + EXTRA FUNCTION STUFF

   def displayimage(image\_name,div\_iwidth, div\_iheight): #displaying imaages (UNO cards) on the screen

       iwidth = x-div\_iwidth

       iheight = y-div\_iheight

       uno\_window.blit(image\_name, (iwidth,iheight))

       time.sleep(0.3)

   def text\_objects(text, font): #this function takes the rectangle and puts it over the whole of the text so it can be moved as one

       textSurface = font.render(text, True, black)

       return textSurface, textSurface.get\_rect()

   def gametext\_display(text,divby\_x,divby\_y,fontsize): #displaying text on the screen

       gametext = pygame.font.Font('freesansbold.ttf', fontsize)

       TextSurf,TextRect = text\_objects(text, gametext)

       TextRect.center = ((x/divby\_x) ,(y/divby\_y))

       uno\_window.blit(TextSurf, TextRect)

   def deal\_deck\_selected():

           numofcards = int(input("How many cards to you want dealt to each player? (max 9): "))

           deck.shuffle()

           Player1.draw(deck, numofcards)

           Player1.showhand()

           Computer.draw(deck, numofcards)

           Computer.showhand()

           gametext\_display('Player1 starts first, use the number keys to select a card',2,12,15)

           #when the TAB key is pressed my program jumps to this function which deals cards to each Player and displays their hand to the screen

   def display\_last\_discarded():

       global maingamepile

       lastcardplaced = maingamepile[-1]

       displayimage(lastcardplaced.image, div\_iwidth-300, div\_iheight-150)

       gametext\_display("Last placed card is:", 2, 3.5, 15)

       pygame.display.update()

       #an image of the last card in play will be displayed to the screen

   def discard\_card\_selected(): #turn controller

       global skipturn

       global reverseturn

       global computerreverse

       global computerskip

       global invalidturn

       if Player1wins == False and Computerwins == False and skipturn == False and reverseturn == False:

           Player1.discard()

           Computer.discard()

           empty\_singleplayer\_screen()

           Player1.showhand()

           Computer.showhand()

           display\_last\_discarded()

           time.sleep(2.5)

           #If no one has won and no action cards have been played (excluded +2), gameplay will run as normal.

       elif skipturn == True:

           Player1.discard()

           skipturn = False

           #if a player plays a skip card then skipturn will become True, when the user selects a card to play, they will be able to discard another one and the computer won't

       elif reverseturn == True:

           Player1.discard()

           reverseturn = False #works the same as the skip section above

       elif computerskip == True:

           Computer.discard()

           computerskip = False

       elif computerreverse == True:

           computer.discard()

           computerreverse = False

       elif invalidturn == True:

           Player1.discard()

           Computer.discard()

           #once the user is punished for making an invalid move, both will be able to play cards as normal

   def deck\_image(width,height):

       global deckImg

       deckImg = pygame.image.load('deck\_image.png')

       uno\_window.blit(deckImg, (width,height))

       #blank uno card image to visually represent the 'pile'

   def add\_screen():

       global uno\_window

       uno\_window = pygame.display.set\_mode((x, y)) #creates a window with specified resolution (x,y)

       uno\_window.fill(white)

       pygame.display.set\_caption('Python UNO') #sets window title

       fps = pygame.time.Clock() #creates a clock that counts fps

       fps.tick(20)

       backgroundImg = pygame.image.load('background\_image.png')

       displayimage(backgroundImg, x, y)

       pygame.display.update()

       #used to completely update a screen (as images can't be removed from a screen in pygame)

       #acts as a blank template

   def createbutton(button\_name,x1,y2,w1,h2,inactive\_colour,active\_colour,action=None):

       mouse = pygame.mouse.get\_pos()

       click = pygame.mouse.get\_pressed()

       if x1+w1 > mouse[0] > x1 and y2+h2 > mouse[1] > y2:

           pygame.draw.rect(uno\_window, active\_colour,(x1,y2,w1,h2))

           if click[0] == 1 and action != None:

               action()

       else:

           pygame.draw.rect(uno\_window, inactive\_colour,(x1,y2,w1,h2))

       #checks if mouse position is within button's defined region and if so, colour will change, if the button is clicked then the defined action will be performed

       button\_font\_size = pygame.font.Font("freesansbold.ttf",20)

       textSurf, textRect = text\_objects(button\_name, button\_font\_size)

       textRect.center = ( (x1+(w1/2)), (y2+(h2/2)) )

       uno\_window.blit(textSurf, textRect)

       #defines button text font and size and adds it the surface of the screen

   def startup\_menu():

       add\_screen()

       pygame.mixer.music.load("Menumusic.mp3")

       pygame.mixer.music.play(-1)

       startup = True

       while startup:

           for event in pygame.event.get():

               if event.type == pygame.QUIT:

                   pygame.quit()

                   quit()

                   #if the user clicks the x on the window the window will close and the game will quit

           gametext\_display('Welcome to UNO',2,2,50)

           singeplayer\_button = createbutton('SINGLEPLAYER',50,450,160,50,green,orange,singleplayer)

           mutliplayer\_button = createbutton('MULTILPLAYER',250,450,160,50,blue,orange,multiplayer)

           quit\_button = createbutton('QUIT',450,450,100,50,red,orange,quitgame)

           help\_button = createbutton('HELP', 600,450,100, 50, white, orange,help\_screen)

           pygame.display.update()

       #main menu - plays main menu music and displays singleplayer,multiplaer,quit and help button

   def quitgame():

       pygame.quit()

       quit()

       #action for quit button to exit game

   def help\_screen():

       add\_screen()

       startup = True

       while startup:

           for event in pygame.event.get():

               if event.type == pygame.QUIT:

                   pygame.quit()

                   quit()

           back\_button = createbutton('BACK',50,450,100,40,white, orange,startup\_menu)

           rulescreen\_button = createbutton('RULES',500,450,100,40,white,orange,rulescreen)

           gametext\_display('KEYBINDS                             ACTIONS',2,12,25)

           gametext\_display('Tab                                       Deal Cards', 2,6,18)

           gametext\_display('x                     Discard Card x in Hand', 2,4,18)

           gametext\_display('Space                                    Draw a card',2,3,18)

           gametext\_display('s                           Show last played card',2,2.3,18)

           pygame.display.update()

           #action for help button, displays back button to main menu and rule button to rule screen and keybinds

   def rulescreen():

       add\_screen()

       startup = True

       while startup:

           for event in pygame.event.get():

               if event.type == pygame.QUIT:

                   pygame.quit()

                   quit()

           back\_button = createbutton('BACK',50,450,100,40,white, orange,help\_screen)

           gametext\_display("Rules: The rules of uno are simple.", 3.5,12,20)

           gametext\_display("Your aim is to get rid of your cards before your opponent.", 2.3,9,20)

           gametext\_display("Start the game by pressing the TAB button",3.3,7,20)

           gametext\_display("Press the numbers keys to discard a card, e.g. 3 will discard your third card", 2,6,20)

           gametext\_display("If you can't match the colour or number of the card on the pile, press SPACE",2,5,20)

           gametext\_display("to draw another card",4.5,4.5,20)

           gametext\_display("if you play a wrong card, a draw will be drawn for you", 3,4,20)

           gametext\_display("Action Cards: Reverse- Reverses the turn order",2.5,3.5,20)

           gametext\_display("Skip - skips the opponents turn", 3.5,3,20)

           gametext\_display("+2- your opponent gets two extra cards", 2.5,2.5,20)

           pygame.display.update()

           #action for rule button, explains the rules of UNO

   def singleplayer():

       pygame.mixer.music.load("Menumusic\_2.mp3")

       pygame.mixer.music.play(-1)

       add\_screen()

       play = True

       while play == True:

           for event in pygame.event.get():

               if event.type == pygame.QUIT:

                   pygame.quit()

                   quit()

               elif event.type == pygame.KEYDOWN and event.key == pygame.K\_TAB:

                   deal\_deck\_selected()

               elif event.type == pygame.KEYDOWN and event.key == pygame.K\_1:

                   global down

                   down = 0

                   discard\_card\_selected()

               elif event.type == pygame.KEYDOWN and event.key == pygame.K\_2:

                   down = 1

                   discard\_card\_selected()

               elif event.type == pygame.KEYDOWN and event.key == pygame.K\_3:

                   down = 2

                   discard\_card\_selected()

               elif event.type == pygame.KEYDOWN and event.key == pygame.K\_4:

                   down = 3

                   discard\_card\_selected()

               elif event.type == pygame.KEYDOWN and event.key == pygame.K\_5:

                   down = 4

                   discard\_card\_selected()

               elif event.type == pygame.KEYDOWN and event.key == pygame.K\_6:

                   down = 5

                   discard\_card\_selected()

               elif event.type == pygame.KEYDOWN and event.key == pygame.K\_7:

                   down = 6

                   discard\_card\_selected()

               elif event.type == pygame.KEYDOWN and event.key == pygame.K\_8:

                   down = 7

                   discard\_card\_selected()

               elif event.type == pygame.KEYDOWN and event.key == pygame.K\_9:

                   down = 8

                   discard\_card\_selected()

               elif event.type == pygame.KEYDOWN and event.key == pygame.K\_0:

                   down = 9

                   discard\_card\_selected()

               elif event.type == pygame.KEYDOWN and event.key == pygame.K\_s:

                   display\_last\_discarded()

                   time.sleep(4)

               elif event.type == pygame.KEYDOWN and event.key == pygame.K\_SPACE:

                   Player1.draw(deck, 1)

                   Player1.showhand()

                   gametext\_display("You've drawn a card from the pile",2,5,15)

               #checks what number in the list of cards the user has selected to remove, and removes that card from their hand

               #pressing space draws a card for the user at any point in the game

               #pressing s displays the last card in play for 4 seconds as a reminder

               '''elif event.type == pygame.KEYDOWN and event.key == pygame.K\_u:

                   global Uno\_called

                   UNO\_called = True'''

           gametext\_display('Player1', 12, 12, 15)

           gametext\_display('Computer', 1.2, 12, 15)

           back\_button = createbutton('BACK',300,500,200,40,white, orange,startup\_menu) #back button to main menu

           width = (x/2.3) #location on screen

           height = (y/3) #location on screen

           deck\_image(width,height) #blank UNO CARD image to represent pile

           pygame.display.update()

   def empty\_singleplayer\_screen():

       add\_screen()

       gametext\_display('Player1', 12, 12, 15)

       gametext\_display('Computer', 1.2, 12, 15)

       width = (x/2.3) #location on screen

       height = (y/3) #location on screen

       deck\_image(width,height)

       #blank template for singleplayer screen, used after cards have been discarded to refresh view

   def multiplayer():

       add\_screen()

       startup = True

       while startup:

           for event in pygame.event.get():

               if event.type == pygame.QUIT:

                   pygame.quit()

                   quit()

           back\_button = createbutton('BACK',50,450,100,40,white, orange,startup\_menu)

           #multiplayer\_local\_button = createbutton('Multiplayer Local',250,450,100,50,blue,orange,multiplayer\_local)

           multiplayer\_online\_button = createbutton('Multiplayer Online',375,450,250,50,blue,orange,multiplayer\_online)

           pygame.display.update()

   def multiplayer\_online():

       pass

   def uno\_gui():

       if login == True:

           startup\_menu()

       else:

           print("User not logged in")

       #user must log in to play first

   # make database and users (if not exists already) table at programme start up

   with sqlite3.connect('uno\_user\_database.db') as db:

       c = db.cursor()

   c.execute('CREATE TABLE IF NOT EXISTS users (user\_id INTEGER PRIMARY KEY, username TEXT NOT NULL ,password TEXT NOT NULL, userscore INTEGER);')

   db.commit()

   db.close()

   #creates a table with 4 columns, userid, username, password and score

   #login class for uno

   class Unologin:

       def \_\_init\_\_(self,master):

           # Window

           self.master = master

           # Some Usefull variables

           self.username = StringVar()

           self.password = StringVar()

           self.userscore = IntVar()

           self.n\_username = StringVar()

           self.n\_password = StringVar()

           #Create Widgets

           self.widgets()

       #Login Function

       def login(self):

           #Establish Connection

           with sqlite3.connect('uno\_user\_database.db') as db:

               c = db.cursor()

           #Find user if there is any, if username and password exist, log in else username not found

           find\_user = ('SELECT \* FROM users WHERE username = ? and password = ?')

           c.execute(find\_user,[(self.username.get()),(self.password.get())])

           result = c.fetchall()

           if result:

               self.logf.pack\_forget()

               self.head['text'] = self.username.get() + '\n Logged In'

               self.head['pady'] = 100

               self.head['padx'] = 100

               global login

               login = True

               uno\_gui()

           else:

               ms.showerror('Username Not Found.')

       def new\_user(self):

           #Establish Connection

           with sqlite3.connect('uno\_user\_database.db') as db:

               c = db.cursor()

           #Find Existing username if user enters name that already exists, they must try another one else success

           find\_user = ('SELECT \* FROM users WHERE username = ?')

           c.execute(find\_user,[(self.username.get())])

           if c.fetchall():

               ms.showerror('Error!','Username Taken Try a Diffrent One.')

           else:

               ms.showinfo('Success!')

               self.log()

           #Create New Account

           insert = 'INSERT INTO users(username,password) VALUES(?,?)'

           c.execute(insert,[(self.n\_username.get()),(self.n\_password.get())])

           db.commit()

           #Frame Packing Methords

       def log(self):

           self.username.set('')

           self.password.set('')

           self.crf.pack\_forget()

           self.head['text'] = 'LOGIN'

           self.logf.pack()

       def cr(self):

           self.n\_username.set('')

           self.n\_password.set('')

           self.logf.pack\_forget()

           self.head['text'] = 'Create Account'

           self.crf.pack()

       #Draw Widgets

       def widgets(self):

           self.head = Label(self.master,text = 'LOGIN',font = ('',35),pady = 10)

           self.head.pack()

           self.logf = Frame(self.master,padx =10,pady = 10)

           Label(self.logf,text = 'Username: ',font = ('',20),pady=5,padx=5).grid(sticky = W)

           Entry(self.logf,textvariable = self.username,bd = 5,font = ('',15)).grid(row=0,column=1)

           Label(self.logf,text = 'Password: ',font = ('',20),pady=5,padx=5).grid(sticky = W)

           Entry(self.logf,textvariable = self.password,bd = 5,font = ('',15),show = '\*').grid(row=1,column=1)

           Button(self.logf,text = ' Login ',bd = 3 ,font = ('',15),padx=5,pady=5,command=self.login).grid()

           Button(self.logf,text = ' Create Account ',bd = 3 ,font = ('',15),padx=5,pady=5,command=self.cr).grid(row=2,column=1)

           self.logf.pack()

           self.crf = Frame(self.master,padx =10,pady = 10)

           Label(self.crf,text = 'Username: ',font = ('',20),pady=5,padx=5).grid(sticky = W)

           Entry(self.crf,textvariable = self.n\_username,bd = 5,font = ('',15)).grid(row=0,column=1)

           Label(self.crf,text = 'Password: ',font = ('',20),pady=5,padx=5).grid(sticky = W)

           Entry(self.crf,textvariable = self.n\_password,bd = 5,font = ('',15),show = '\*').grid(row=1,column=1)

           Button(self.crf,text = 'Create Account',bd = 3 ,font = ('',15),padx=5,pady=5,command=self.new\_user).grid()

           Button(self.crf,text = 'Go to Login',bd = 3 ,font = ('',15),padx=5,pady=5,command=self.log).grid(row=2,column=1)

   #create log in window and application object

   root = Tk()

   root.title("Login Form")

   Unologin(root)

   root.mainloop()

   **Server (used for multiplayer…to be completed)**

   import socket

   from \_thread import \*

   import sys

   server = "192.168.1.72" #ipv4 address here

   port = 5555

   s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

   try:

   s.bind((server, port))

   except socket.error as e:

   str(e)

   s.listen(2)

   print("Waiting for a connection, Server Started")

   def threaded\_client(conn):

   conn.send(str.encode("Connected"))

   reply = ""

   while True:

   try:

   data = conn.recv(2048)

   reply = data.decode("utf-8")

   if not data:

   print("Disconnected")

   break

   else:

   print("Received: ", reply)

   print("Sending : ", reply)

   conn.sendall(str.encode(reply))

   except:

   break

   print("Lost connection")

   conn.close()

   while True:

   conn, addr = s.accept()

   print("Connected to:", addr)

   start\_new\_thread(threaded\_client, (conn,))

   **List of References**

   Below is an ordered list of citation that link to where I retrieved any image or information used in this analysis.

   <https://sea.mashable.com/entertainment/2138/its-confirmed-you-can-indeed-end-an-uno-game-with-an-action-card> [↑](#endnote-ref-2)
2. <https://steamdb.info/app/470220/graphs/> [↑](#endnote-ref-3)
3. <https://store.steampowered.com/app/470220/UNO/> [↑](#endnote-ref-4)
4. <https://store.steampowered.com/app/470220/UNO/> [↑](#endnote-ref-5)
5. <https://tesera.ru/images/items/855783/f00fc229-565a-400e-b9cf-9a9152ffd067.pdf> [↑](#endnote-ref-6)
6. <https://tesera.ru/images/items/855783/f00fc229-565a-400e-b9cf-9a9152ffd067.pdf> [↑](#endnote-ref-7)
7. <https://en.wikipedia.org/wiki/Craits> [↑](#endnote-ref-8)
8. <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller> [↑](#endnote-ref-9)
9. <https://www.giacomodebidda.com/mvc-pattern-in-python-introduction-and-basicmodel/> [↑](#endnote-ref-10)
10. <https://www.tutorialspoint.com/artificial_intelligence_with_python/artificial_intelligence_with_python_gaming.htm> [↑](#endnote-ref-11)