GCE Computer Science

H446 - 03

Programming Project: Chess

Name: Oliver Butler

Candidate Number: 3036

Orleans Park School

1. Analysis

The Problem

Currently, chess has been having a downfall of players due the challenging nature of the game as well as more recent generations being attuned to separate forms of entertainment such as: video games, television and sports. This decrease in population of chess players means there is a lack of players to play against decreasing the skill of players, despite there being the most information available to any one available now. Chess, also, usually requires a second player, which means that a portion of the people who have no one to play with will have an exceptional inability to increase their skills due to the lack of a skilled player, or even just a normal player, being present. Furthermore, only online chess games will allow the feature of analysing the moves as well as showing the history of games compared to having to write them down by hand. This then means most players, with a computer, without an internet connection will have no resources to be able to fully understand how to improve. Therefore, due to the decreasing player count, as well as the possibility of lack of connection to the online world, most players will not be able to have an equal ground compared to some players, meaning there is an increased likelihood they will quit playing due to the skill gap of certain players, causing chess as a whole to dwindle in numbers.

Most chess players will play on a computer or physical chess board. For physical chess boards, you must at least have a second player to play against, meaning it is impossible to practise (unless you are doing chess puzzles) against other players, therefore decreasing the likelihood of getting better at the game. You also have less access to resources that may better help you such as the ability to analyse your moves unless you write down every move by hand, which may be time consuming. Furthermore, on a computer, you will usually have to have an internet connection, meaning if you have no connection to the internet you will likely have no way of playing either against another player or access certain resources, which also makes progression into getting better at the game harder.

Suitability to a Computational Solution

Many of these problems are solvable using computational methods. By allowing a user who has a computer to use a program which allows the user to play either against an AI or against a player on the same device, where giving the option to write the game history to a file or online database if needed to those who have a connection to the internet. An AI can be used to allow users with no available partner to play against and increase their skills, as well as an option to play against other players on the same computer to allow those who do not have more than one computer to play against a person in the same household. Those who have access to the internet may also utilize an online database which stores information on their games played. A suitable GUI that includes information about the

statistics of players may allow users to see how their ability is increasing over time to therefore encourage them more to increase their skill in the game.

The Proposed Solution

My idea for a chess game will allow players who have access to a computer to play chess either against an AI or two players in an offline environment. This will allow users who do not have access to the internet to play against someone to increase their skills whilst still allowing players who have access to the internet to create an account and record their progression on a database.

Offline playing -
This option of playing will allow any user to play against another user on the same device, this is to allow people who do no have access to multiple devices and an internet connection to play with each other. This is to allow all ranges of users to use this application as you all users will be able to access this version.

Online playing -
This option of playing will allow any user to play against another user on the same device however each user can sign-in and use an account where the results of the game can be stored online. This is to allow users who can gain access to an online database to better record their games and history such that they can more easily improve.

AI playing -
If the user does not have an opponent to play against on hand, they can play against an AI. This will allow users who might not be able to play against anyone either in an offline or online environment to at least be able to use the program for their needs. This still allows users to improve in chess despite not being able to play against another human user.

Menu Options -
The user will have the option between three options “Offline”, “Online” or “AI” which can be accessed through the menu. Furthermore, there will be information on how to use the program as well as in depth information on basic strategy, rules and theory on chess such that if the user has little knowledge of how to play, they will be able to understand how to. This is to allow users with even the most basic knowledge of using a computer to be able to use the application easily, and to be at a similar advantage to other players.

Other Options -
All games will be recorded onto text files, each with the move history of each game. This allows playback of the game to allow the user to analyse the moves played in the game to see where they went wrong. Furthermore, recording of games on the database will require an account to be made which will record data on the game, allowing a ranking system on

players. This will allow the player to see the progress they are making and see how well they are compared to other players in the database.

All these features will allow players who have a computer to have the ability to play against anyone, or even no one, either on an online or offline basis, which allows them to increase their skills overtime despite possibly having a lack of resources to do so.

Limitations

As there is no multiplayer option available, it is likely that the user will not be able to play against a wide range of players. This is due to only being able to play against players on the same device as adding a multiplayer option is quite time consuming and complex. Furthermore, you must decide whether it will run on a dedicated server, or a client will host, which would either cost money if you were to have a dedicated server, or be complex to implement correctly. You must have someone who takes care of the server in case any problems arise.

Furthermore, unlike many high-tier chess programs, the AI will likely not use machine learning meaning it will likely be easy to play against. This is due to the complexity of training the AI as well as having the resources and ability to code it in the first place. This will drastically decrease the depth of the AI meaning it will be much easier to beat due to the amount of moves being predicted is cut. However, it will decrease the time it takes to calculate an AI move, as the more moves you predict, it will exponentially increase the time it takes to calculate the next move. This will decrease computational power.

The data stored from the server will likely not be secure as most companies meaning it is important to make an account which a password used is not important to the user. While the data may be encrypted or hashed, this does not mean it will not be impossible for someone to maliciously get access to this data. Furthermore, due to the project being free, there will likely be no dedicated server that the data is being stored on, meaning if there is a large amount of users, it will become quite slow.

Lastly, due to the complexity of creating a good anti-hack, it is likely users may be able to manipulate the game in ways that were not designed using external software to do so. This, however, is not a huge problem, as all games will be on the same device, meaning if a user were to cheat, it will be easily detectable by the other user, and as all games are recorded, there will be proof they cheated.

Stakeholders

To be able to successfully reach a large range of users and their demands, it is important to have a wide range of stakeholders that have ranging abilities, either with their ability to use technology, or their skill in chess. This is why I believe that the stakeholders that best fit these demands are two stakeholders who range from competing in tournaments, to a beginner with little knowledge of how to play chess. This is because we can get a fair examination of the project, as we have opinions which covers all aspects of chess players.

Stakeholder 1: Oscar Butler, an avid chess player with a rated elo of 2027. He has played in multiple ranked tournaments and has a wide knowledge of chess, due to this it is important to get an idea of what a chess enthusiast wants in the program as they have lots of experience in chess and know what can help benefit new players.

Questions about the program	**Answer**
“How can this help me get ready for my chess tournaments?”	When choosing the separate playing options, you are able to select upon multiple settings “Offline”, “AI” or “Online”. If you have a person to play against on the same device, you can play against them as usual. However you are able to play against an AI that uses algorithms to determine the best moves for itself. Furthermore, there is no need for an internet connection, meaning if you are there at the tournament but have no access to the internet but have a device that can run the program, you can play any time.
“Can I review past games?”	All games will be recorded to a text file in the game file where it is stored. This can be read and is written in the algebraic chess notation. This allows users to read the past game and try to understand where they went wrong.
“Can I play against other users?”	You can either play against users on the same device or an AI. An online multiplayer may come but the base program will only support an AI or offline game.
“Are there different chess game modes?”	You can only play either against a user and an AI however you can play bullet chess, a form of chess where both users have a very low time to play.

Questions to the stakeholder	**Answer**	**Response**
“What features do you believe are important for you in a chess program?”	“Being able to play against different people or a computer with randomised opening moves in order to improve my opening knowledge. It	To adapt to these needs, a list of opening moves will be accessible in a menu which gives information about how to play a game.

	is also useful if the chess program has a list of commonly played opening moves."	
"How familiar are you with chess and what would you look for in a program to help you improve from there?"	"I have played chess for a year, but I have already won prize money in many tournaments. It is important to me to be able to review my mistakes from games I played in tournaments, preferably through computer analysis."	All games will be recorded onto a file written in the algebraic chess notation. This allows users to analyse their mistakes and see where they went wrong by replaying the match.
"How difficult of an AI would you prefer?"	"A computer with a high depth would be useful to play against, however not too hard where it is impossible to beat."	An AI that compromises between time to calculate moves as well as an AI which is difficult for ordinary players to play against will be used. This will allow players that either have little knowledge to be challenged whilst also not being impossible.

Stakeholder 2: Jake Elliot, a chess player with an unrated elo of 500, with little knowledge of the game who has only recently picked up the game. As he is the target consumer, it is important to know what he wants in a program to help him progress as a new player to become better at the game.

Questions about the program	**Answer**
"Is there a way I can play against people online"	Sadly, an online multiplayer will likely not be used but you can play against other players on the same device and have everything recorded onto an online database which can be used to help show progression.

Questions to the stakeholder	**Answer**	**Response**
"What features do you believe are important for you in a chess program?"	"I'm quite new to chess so I don't know much about the rules and how pieces would move so as long as I can't do any moves which I shouldn't it should be fine."	All rules and help guidance will be given in a menu which shows how to play.
"How familiar are you with chess and what would you look for in a program to	"I've only played a few games with people, and only know the basics. I however don't have anyone to play	An AI can be used in the absence of not having anyone to play with.

help you improve from there?”	with so hopefully I can play against some other people”	
“How difficult of an AI would you prefer?”	“As long as I have some way of beating it I would hope it isn't too difficult”	An AI that compromises between time to calculate moves as well as an AI which is difficult for ordinary players to play against will be used. This will allow players that either have little knowledge to be challenged whilst also not being impossible.

User Access Table

User	Access to:
Player (Without Internet Connection)	• The Option to play in separate modes: - Offline (against themselves or someone on the same device) - AI (against an AI) • A guide on the simple rules of chess • An option to record the information of games onto a text file
Player (With Internet Connection)	• Gain access to everything listed in “Player (Without Internet Connection) • The Option to play in an additional mode “Online” where the results of the match are stored on a database • Create an account which records information onto an online database of the player’s “Wins” “Losses” “Draws” onto their account

Research about similar programs

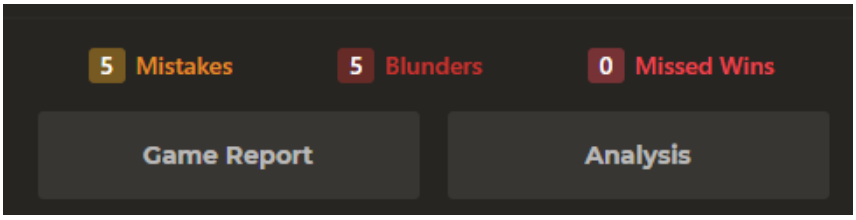
Chess.com:

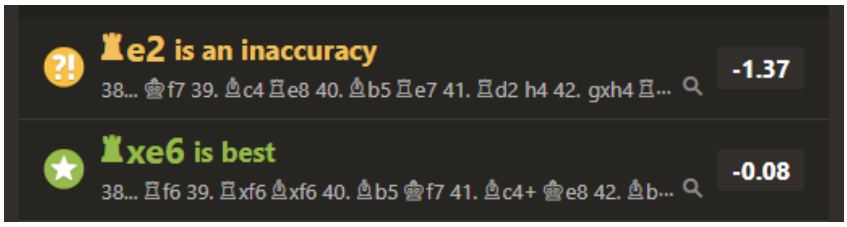
Summary:

Chess.com is a chess website, built on the stockfish engine, which offers a variety of chess content such as: puzzles, multiplayer matches and AI analysis. It is a great website to use for a wide variety of players, from beginners to advanced.

Likes:

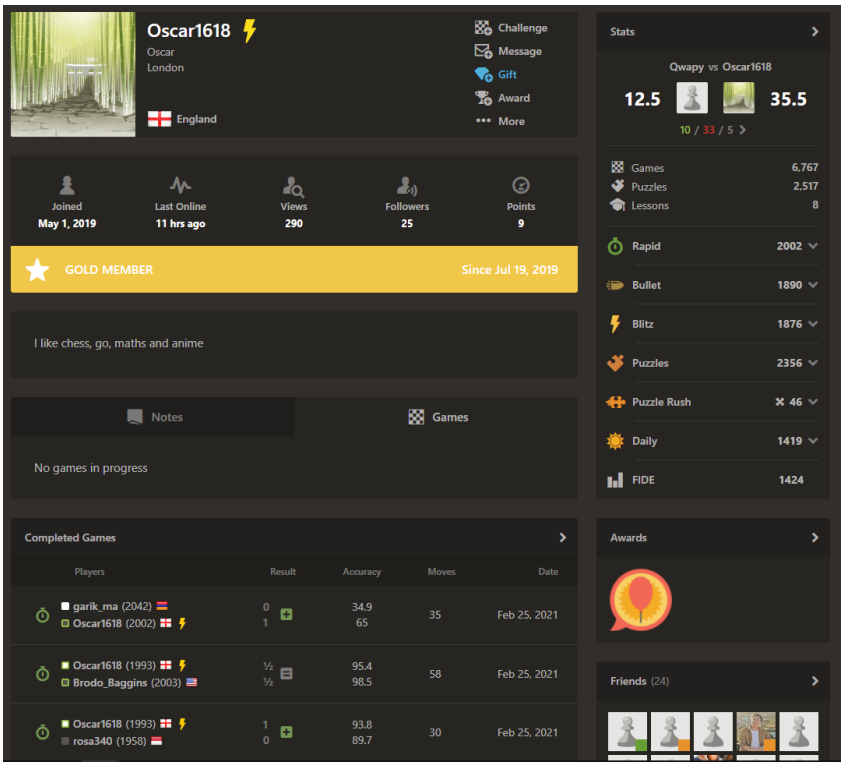
Chess Engine - The chess engine used for analysis of moves as well as AI is the stockfish engine. This is one of the most powerful chess engines allowing it to have a very powerful AI. It allows users to check their mistakes they have made during games, check which side has a better position as well as the chances of winning a match. This is great for many beginner users to see where they have gone wrong, and to see what moves would be the best in each situation.





AI - You are given a wide range of AI to play against ranging between different elo rankings from beginner to grandmaster level, this is great to have in a program so that almost any player at any level has a challenging opponent, or easy opponent, that they can play against.

Account Information - If you choose to create an account, each user can be given an elo rating, this is used to rank users skills, as well as create fair and equal matches between users. They store a wide range of information on users, allowing them to analyze their past games, their multiple rankings in separate variants of chess and challenge users.



Puzzles - Ranked puzzles are given to the user to complete to improve their skills. These range from small games which last only a few moves, where your objective is to win material, have a better position or to win the game, starting from a certain position.

Dislikes:

Membership - Certain aspects of Chess.com are locked behind paywalls, which while

cheap, sets a disadvantage to less fortunate players. Such examples include: not being allowed to analyse games or do puzzles.

Online - Every aspect of Chess.com is locked behind having an internet connection, on most devices. This means if you do not have an internet connection at the time, you will not be able to use their services. This is inefficient as if you are travelling often, it is unlikely you will always be able to connect to the internet.

Adaptations:

Every feature offered by my program will be accessible to everyone, or atleast have an option which is equivalent to those who cannot, such as in the case where a user does not have a connection to the internet. This means that the whole application will be free to use and have no requirement to pay for any features. Furthermore, whilst there is a large amount of user stored data which can be useful to the user, only a small subset of it is essential to store and use for player development.

Lichess:

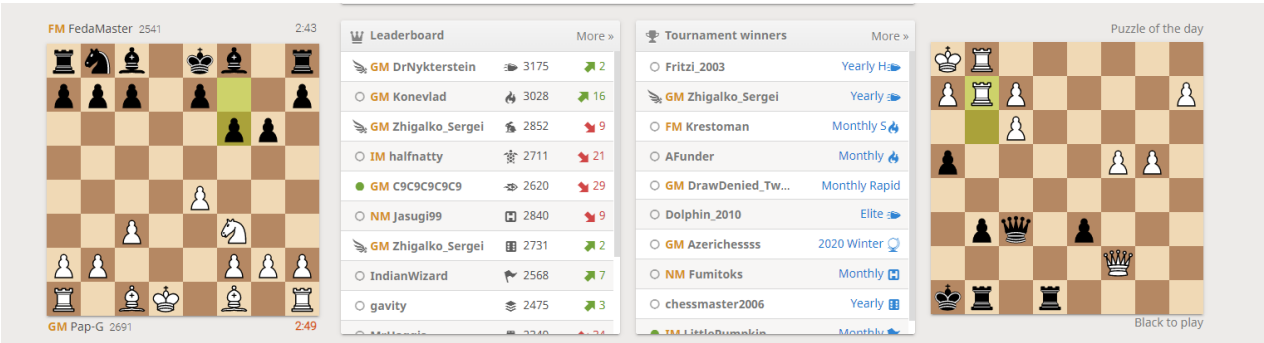
Summary:

Lichess is a chess website specializing in hosting and creating tournaments, whilst also offering users simple chess games. This website is great towards players who have an understanding of chess at a simple level and beyond.

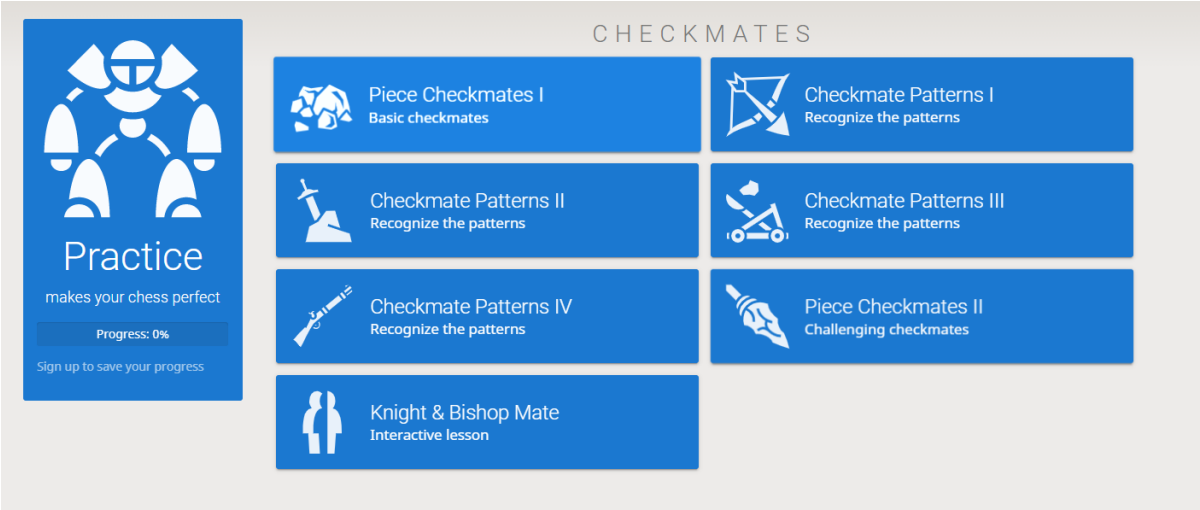
Likes:

Membership - All features are free to use, and remain so, meaning you can play as many games as you want to, this is due to the fact that the website is non-profit and open source.

Tournaments - Lichess offers a range of tournaments which can be viewed by all users, even on the front page live. This is great for players who want to see how higher rank players can play, and learn from them, or to create miniature tournaments for either school or against friends.



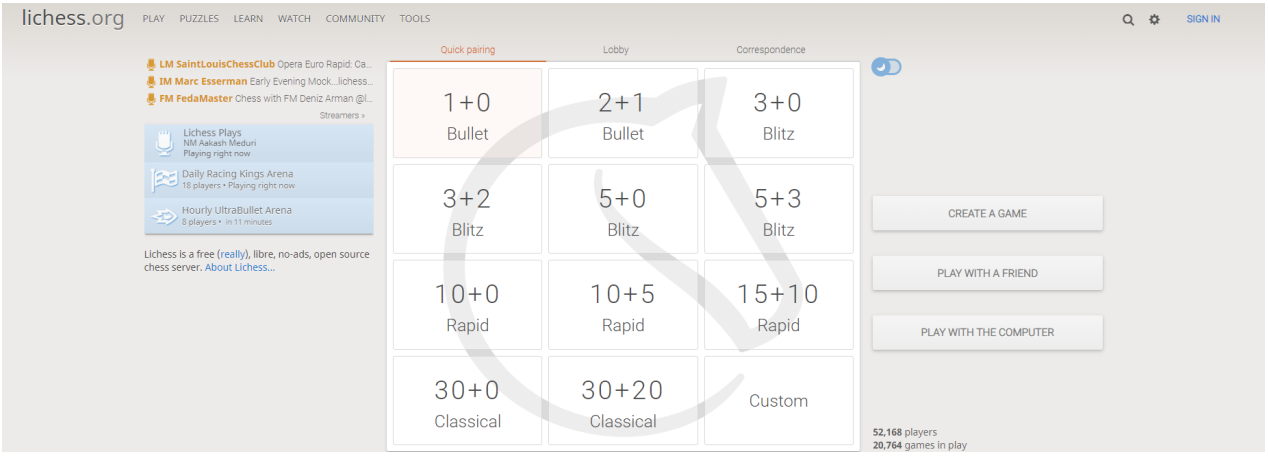
Learning - A large range of lessons and puzzles are available for free to all users, allowing beginners to advanced players to learn.



Dislikes:

Online - Every aspect is locked behind having an internet connection, on most devices. This means if you do not have an internet connection at the time, you will not be able to use their services. This is inefficient as if you are travelling often, it is unlikely you will always be able to connect to the internet.

UI - Whilst the UI is very simple, it is quite ugly to most users. Allowing users to change certain UI elements, or having a darker theme, in my opinion at least, would likely be favourable to most users.



Adaptations:

From this, I will try to create a UI which is more appealing to users and easier to navigate by eliminating unnecessary information or options. Furthermore, allowing an option for users to play in offline environments to further progress, similar to the adaptations to chess.com.

Chess24:

Summary:

Chess24 is a chess website which hosts tournaments, games and news on chess. This website is primarily built for users who are familiar with, and interested in higher skill chess tournaments or chess news.

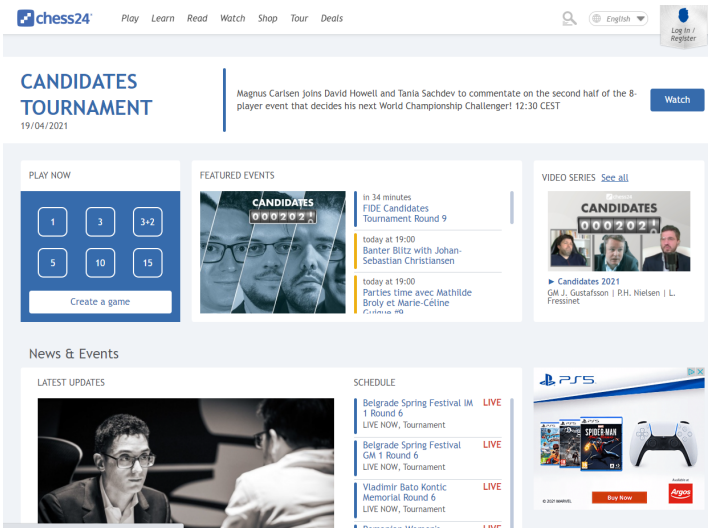
Likes:

Tournaments - Due to the website sponsoring many high skill players such as Magnus Carlson, there are a wide range of tournaments and games available to play against these high skilled grandmasters, making it great for players with a high skill rank.

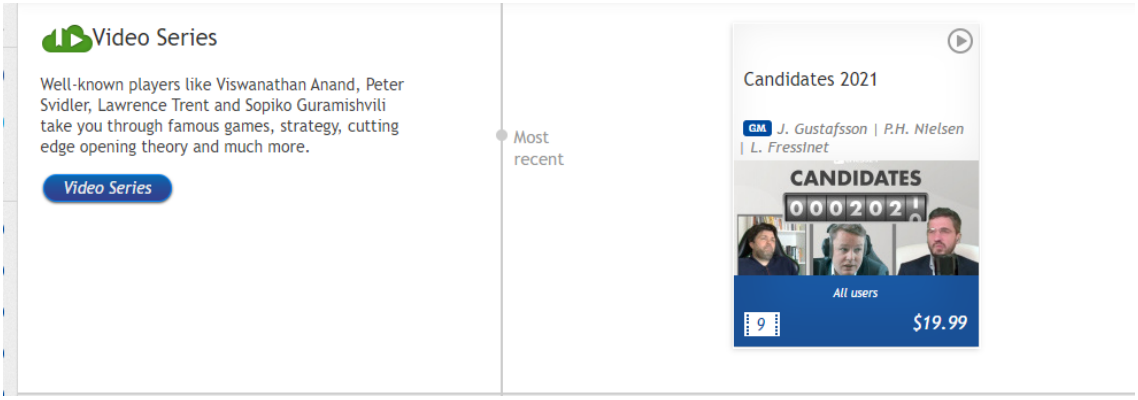
Learning - There are courses available for players to take to learn the basics of chess, however, do not develop into more complicated techniques. This is great for newer players but not for more advanced players.

Dislikes:

UI - Many ads are on the screen as well as a large amount of information at once making the website harder to navigate. This makes it harder to find information that the user might want like how to just play a basic game.



Pay-to-win - A large amount of learning content is behind a paywall as either books or video series, making it unfair for some users who cannot afford or access this.



Adaptations:

From this, I will ensure all content is free to access such that it is fair to all users. Furthermore, create a UI which does not bombarde the user with a large amount of information at once making it easier to navigate.

Research Summary:

From this, I will likely use a free-to-play approach where there are no features locked behind a paywall. This is to allow all users to have an equal level field and to be able to access as much content as I can provide, such that any range of users will have something they can do. Furthemore, when developing the AI, I must compromise between time to calculate as well as power will be used, as an AI which takes 20 seconds per turn would not be the best to use. Most likely, a simple AI, which is easy to run on all devices, will be used, so that most players will have a challenge to play against, whilst still maintaining a fair but equal game. Most features will be accessible to everyone except if you don't have an internet connection, where if you do, you will be able to access a database which stores your previous plays. This is to create a fair application that primarily focuses on allowing all users to progress equally. Lastly, there will be no monetization via ads or any sponsored sections. This is to simplify the UI making it look more appealing, as well as to not annoy any users in meaningless information.

Hardware and Software Requirements

Requirements	Why it is needed
- Mouse	- To be able to select between different options and to be able to select and move pieces why are only possible with a mouse
- Screen / Monitor	- To have a visual representation of all options to the user and to be able to easily use the program
- An updated version of MAC, Windows 7-10 or Ubuntu Operating System	- Such that the device that is running the program is able to compile the information correctly for that device.
- 100MB Free Storage Space	- To be able to store the program on the device
- 2GB RAM	- This is so you can run the game smoothly without the use of VRAM. This will mainly be used for the AI when calculating each move, else only a small amount is needed.
- Keyboard (optional)	- To input data needed by the program such as when accessing your information on the online database.
- Internet Connection (optional)	- To be able to connect to the server / database which stores the information needed to sign-in to online options.

Target Platform

Due to simplicity, the project will primarily run on most desktop PC's running an OS that can run Unity Applications, such as Windows 7+, Mac OS 10.12+ or Ubuntu 16.04+. This will allow us to have a larger reach in players due to a larger proportion of chess players will have access to a PC. Furthermore, this will simplify UI such that we do not need to scale the resolution and aspect ratio for all platforms, as well as input devices such as a touch screen or controller.

Project Objectives & Success Criteria

Objective 1	**Order of Completion**	
Create a GUI to allow users to select seperate game modes (e.g. ai, online, offline etc.)		
Success Criteria		
- Online, AI, Offline selections, each navigating to their corresponding gamemodes - Main Menu screen		
Possible solutions / How to achieve success criteria		
1. Allow users to select either “settings” / how to play or “Play” i) The ”settings” option will display options available for the user as well as a tutorial on how to play / the rules of the game ii) The “Play” option will route into separate options such as “Online”, “Offline”, “AI”		
Possible problems		
- A good and easily usable GUI should be used or else many users may be turned off by either the complexity or look of the GUI, this means a good balance of usability and design must be used.		
1. Knowledge into creating appealing GUI in Unity		

Objective 2	**Order of Completion**	
Create basic rules for chess when players play against other users (against themselves / players using the same device) with making sure all moves and rules are valid		
Success Criteria		
- A GUI that allows users to move pieces around the board - Rules which correspond to the correct pieces such as movement and taking - Having a check for either check, checkmate or stalemate - Implement more GUI which shows the players current score (collected pieces) as well as timers. - The whole history of moves will be written into a file		
Possible solutions / How to achieve success criteria		
1. Firstly, using prior knowledge as to Objective 1 on creating GUIs, a board will be created (8x8), each piece will have its own separate and correct corresponding image. This can either be drawn in an editor or taken from google etc. 2. Then, each piece will be coded into having correct movement corresponding to the correct piece. Each piece will obtain the ability to take when possible including checks for each piece, if the move is possible, the next player's turn will change over, else, it will continue. This will also include checkmate, check and stalemate.		

3. As all the rules are implemented, a more in depth GUI will be built. This includes a timer which alternates between each player, the time being able to be changed before the match to best suit the users. It will also use a point system corresponding to the worth of each piece (pawn -> 1, knight + bishop -> 3, rook -> 5, queen -> 9)

Possible problems

- Certain checks for checkmate, check and stalemate may be harder to code, especially for stalemate, this might cause problems with the code later on. Furthermore, there are more complicated rules which might be much harder to design later on such as you cannot castle in check, repeated moves rules as well etc.

Coding / Technical work required

1. GUI knowledge
2. An in depth knowledge on the rules of chess

Objective 3	**Order of Completion**	
Implement more difficult win and draw rules including timers which decrease over time		

Success Criteria

- A timer which counts down and alternate per person
- Implement 50 move rule (when 50 moves are done without a piece being taken or a pawn moving)
- En passant (a pawn can take another pawn if it has just moved 2 places forward and lands next to it)
- Castling (neither the king or castle has moved and all pieces between have moved)
- Threefold repetition (the same position has been repeated 3 times)

Possible solutions / How to achieve success criteria

1. A timer will update each frame and decrease over time, the timer will alternate per person. If the counter ends the person loses
2. The 50 move rule will have a counter which increases over time per move and if no pawn moves or no piece are captured for 50 moves the game ends
3. When a pawn passes another by 2, it will be able to en passant
4. Check if certain places are free to move to and a way of checking it has not moved yet
5. Store the positions of all pieces and checks them and if they repeat 3 times the game draws

Possible problems

- The threefold repetition may use a lot of memory over time as it must store many positions, a more efficient way of doing this may be to delete / clear the list once it is impossible for any previous moves to be replicated (e.g. due to a piece being taken etc.)

Coding / Technical work required

3. GUI knowledge
4. An in depth knowledge on the rules of chess

Objective 4	**Order of Completion**	
Create an AI which a user can play against, this using a similar interface to Objective 2.		
Success Criteria		
- Using either a API or prior knowledge to allow the AI to make moves which are legal as well as make logical sense. - Implementing this with a GUI to allow a user to play against an AI		
Possible solutions / How to achieve success criteria		
1. Using either the API from resources, such as Chess.com or other programs, or implementing and coding an AI from scratch i) Firstly, the AI will have to make at least legal moves, these might not be great but must be doable in a game ii) Secondly, each piece will be assigned values (seen prior to the last Objective), allowing the AI to evaluate when it is possible to take a piece iii) Thirdly, using Minimax algorithm to have a more logic move as well as alpha-beta pruning iv) Lastly using the positional analysis on where is is the best for a piece to be at a time this can help the AI to understand where to go and where more pieces may be worth than others 2. Integrate this to where a player can play against an AI with similar GUI to as the Offline option		
Possible problems		
- Due to the difficulty of coding a good AI, most use machine learning or very difficult solutions, this means my AI will likely not be as good compared to most AI available if I do code it.		
Coding / Technical work required		
GUI AI implementation for Chess		

Objective 5	**Order of Completion**	
Create a secure login option for the “Online” option		
Success Criteria		
- GUI which allows user to enter username/email and password - Database which stores encrypted version of the password - User redirected to correct GUI when they pass authentication - Pop up to say when the user has entered incorrect details - Options to change account passwords / details when a username is forgotten (using questions)		
Possible solutions / How to achieve success criteria		

1. Create a database which includes a table to hold usernames and passwords, this can b e hosted on many different websites but will require an online connection
2. Create a GUI which allows the users to input their information when either creating or logging into an account. This is checked for valid information when on login and creation of an account to make sure emails and usernames are not repeated.
3. When creating an account, the email is encrypted as well as password will be hashed, this is then stored into a database. The database will store the users login information as well as other fields such as: ("Games Played", "Games Won", "Games Lost", "Games Tied", "Current Ranking")
4. The user will have to input questions as if the user forgets their password / login information, then they can use this to gain access to their account

Possible problems

- If a commonly known encryption is used or hashing method, it may likely decrease the security of the users

Coding / Technical work required

1. SQL
2. Validate input of username and password
3. Research into encryption as well hashing

Objective 6	**Order of Completion**	

Manipulate data in a database to allow a ranking system as well as previous stats. This will then be presented to the user on the application.

Success Criteria

- When a user has the correct credentials, they will be able to choose to see their account information. This includes the most recent game's history, wins, loss, draws etc.
- Manipulate data to add a ranking system

Possible solutions / How to achieve success criteria

1. Once the users correct information is given, they can choose to see their data. Using the database which stores their information, we can use it to receive their information on previous games, etc.
2. Using elo ranking formulae, we can calculate the elo of the player to calculate their performance

Possible problems

- If a lack of players are in the system, it may become harder to calculate performance as they will not accurately show the full potential of each user or might over assume their elo / skill.

Coding / Technical work required

SQL / Databases

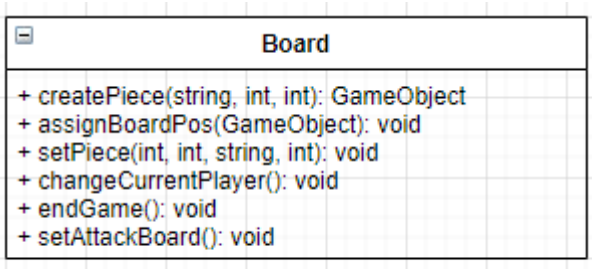
2. Design

Algorithm

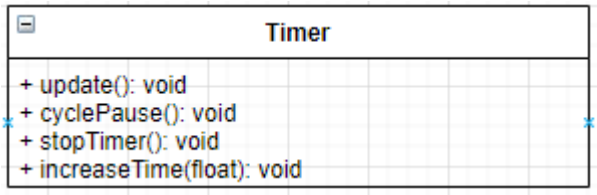
Classes -

There will be 7 main classes used in the game, this is the Board class, Timer class, Pieces class, MoveTile class, AI class, Database class and Menu class.

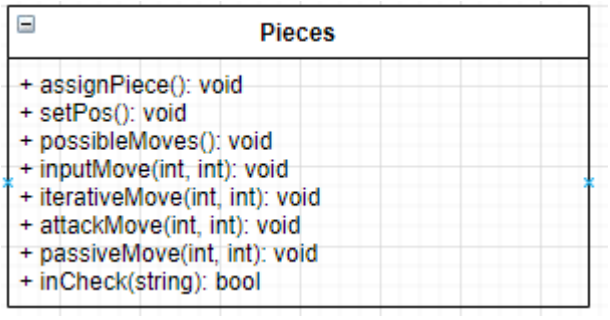
Board class - The main class which stores the board, attack board, game checks such as drawing, stalemate etc., and the pieces on the board. It instantiates the location of a piece on the board and will handle the rules of chess which are to do with winning, losing or drawing. It stores the history of all moves, which will then be appended to a text file at the end of the game.



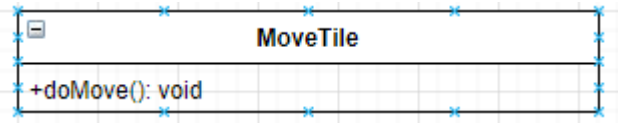
Timer class - Controls a decrementing timer for each user. The timer will start at 15 minutes and decrease for the user's turn by calculating the time between each frame. Once the timer reaches zero, then the user will lose. This is used to make sure games do not last for a long period of time.



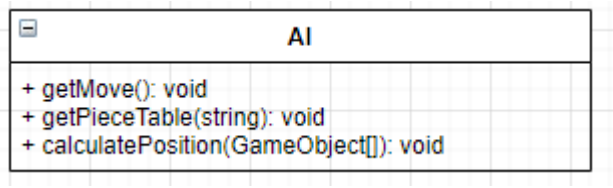
Pieces class - Controls the piece logic and being able to control the pieces. Each piece is given a sprite value corresponding to the name which is used to visually represent the piece on the screen. All moves for all types of pieces are located in this class.



Movetile class - Allows the user to move the piece by clicking on a tile which is created. Each tile is created in the pieces class when a piece is clicked on and creates a new tile. The tile then when clicked on will then destroy pieces and move them corresponding to which type of tile it is. It will handle the logic for the selected move.

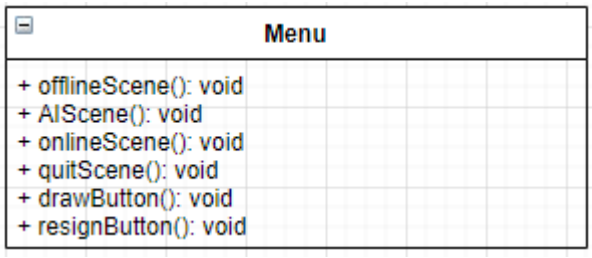


AI class - Used when the AI option is selected in the main menu, this using the minimax algorithm as well as alpha-beta pruning will select a location for a piece to move to which best fits where to go. It will control the logic for what move the AI will select.



Database class - Used when the online option is given, to allow users to communicate to each to create an account or sign-in, where the result of matches will then be stored into a database, where a summary of a users data can be shown to show the progression of the player.

Menu class - Used to give each button in the menu a control, this will then be used to control scenes and allow users to select options to navigate through the program.



Downloaded Assets -

When choosing the sprite assets, you can either create your own or download free ones online. I'm choosing to download pre-made ones, because drawing is not my strong suit and it is likely there are better assets out there that I can use that would make my game better than trying to make them myself.
Using sprites - https://devilsworkshop.itch.io/pixel-art-chess-asset-pack

Needed Variables -

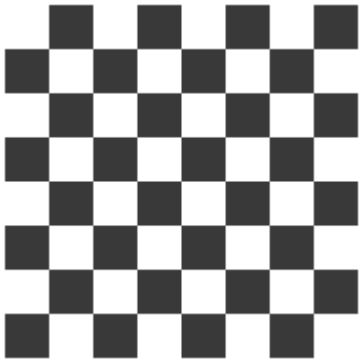
Variable Name	Class	Type	Use
Board	Main / Board	8x8 Array	To store the pieces on the board to be accessed when taking or moving a piece
Pieces (may need to separate by colour to two variables)	Main / Board	16-32 Array	Stores all pieces to easily access when taking a piece or moving a piece
White Attack Board	Main / Board	8x8 Boolean Array	States whether or not a position is under attack from the white side. This is used for checking if it is possible to checkmate
Black Attack Board	Main / Board	8x8 Boolean Array	States whether or not a position is under attack from the black side. This is used for checking if it is possible to checkmate
Current Player	Main / Board	String	Keeps track of current player colour to make sure the other user cannot move when it is not their turn
X Position	Pieces + Move Tile	Integer	Stores the X position of the piece / move tile so it knows the position on the board
Y Position	Pieces + Move Tile	Integer	Stores the Y position of the piece / move tile so it knows the position on the board

Piece Value	Pieces	Integer	Stores the piece value for an AI which knows which piece is more important to take
Piece Colour	Pieces	String	Stores the piece colour to make sure you cannot take the same colour as your own
Has Moved	Pieces	Boolean	Stores if the piece has moved or not. This is used for pieces which have special moves which can only be done when it has not moved yet
In Check	Pieces	Boolean	Stores to see if the piece is in check (used for the king), this is to make sure you cannot move a piece while your king is in check
En Passant Attack	Pieces	Boolean	States whether a pawn can en passant as it is a move which is only possible in certain positions.
White Time	Timer	Float	Store the current time of the white side as once the timer stops, the game will end to make sure both users do not take too long between taking turns.
Black Time	Timer	Float	Store the current time of the black side as once the timer stops, the game will end to make sure both users do not take too long between taking turns.
Is Attack	Move Tile	Boolean	States whether the tile is a move tile or an attack tile as to make it easier to distinguish if the move is taking a piece or not
Is Castle	Move Tile	Boolean	States if the tile is a castle move as to distinguish if this is a specific type of move to calculate.
Is En Passant	Move Tile	Boolean	States if the tile is an en passant move as to distinguish if this is a specific type of move to calculate.
Current Time	Timer	Float	Stores the current time of the timer

			as once this reaches 0, the game ends
Is Paused	Timer	Boolean	States whether the timer is paused or not as so the timer does not decrease when it is not their turn
Starting Time	Timer	Float	Stores the time which the timer is counting down from as this can be manipulated to allow any time to start with

Board -

A chess board is an 8x8 board which each cell stores can only store one chess piece at a time. Each cell will store a reference to a GameObject in either the black pieces or white pieces array as to easily access either side's pieces instead of iterating through the board to find the location of each piece.



To allow a piece to be visually shown on the board, we must translate its position onto each cell. Each piece has its location stored in an array which we use to translate that board position into coordinates onto the screen. This will then center each piece onto the board.

We must also make sure no values parsed are outside the boundaries of the array, meaning only values between 0-7 will work (we use 0-7 as an array starts at 0 not 1 so it will still represent an 8x8 array).

We must make sure also to rotate turns each round after a piece has moved:

Def rotateTurn():

```
Switch(currentPlayerColour):
        Case ‘white’: #if the current player white
                currentPlayerColour = ‘black’
        Case ‘black’: #if the current player black
                currentPlayerColour = ‘white’
```

We must also create two boolean arrays which are similar to the 8x8 board, but stores if the location on the board is being attacked or not. This is used to check if a piece moves to a location, to see if it will cause a check, which is where the king is under attack so therefore must be stopped.

At the end of each turn, we iterate through all possible moves each side can do and then store the locations of the positions of the board into the boolean array as true. This means this location is currently able to be attacked.

```
Def getAttackPos():

blackAttackPos = null
whiteAttackPos = null
For currentIndex in range len(whitePieces): #iterates through all pieces
        getMoves(whitePieces[currentIndex]) #gets all possible moves and sets each
        getMoves(blackPieces[currentIndex]) #spot to true if it can be attack
```

Timer -

Each side has an instance of the Timer class, this has all logical operations for a timer which counts down from a value. On start, each timer will be given a time in seconds, it will then count down from that value when it is that users turn, else it will be paused. This means during rotating players, we must pause each timer, or resume each timer. Once the timer is 0, which is checked every frame, if so we end the game with a loss due to lack of time.

```
Def rotateTurn():

Switch(currentPlayerColour):
        Case ‘white’: #if current player is white
                whiteTimer.pause #pauses white timer
                blackTimer.resume #resumes black timer
                currentPlayerColour = ‘black’
        Case ‘black’: #if current player is black
                blackTimer.pause
```

```
            whiteTimer.resume
            currentPlayerColour = 'white'
```

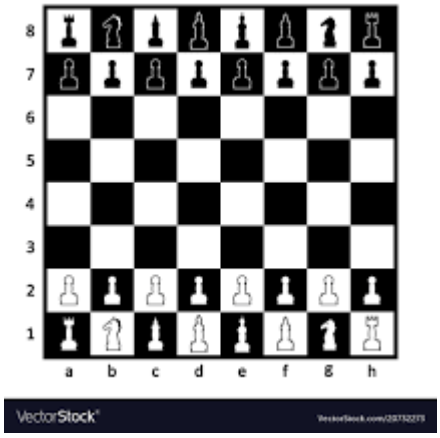
This rotates players turns

```
Def Update(): #this is called every frame

If paused != true: #if the timer is not paused it will change time
        currentTime -= time.deltaTime #the time between each frame is subtracted
        If currentTime <= 0: #if the timer runs out then end the game
            gameEnd()
```

This calculates the current time and will end the game if it reaches 0.

Chess Pieces -

Each chess piece type has its own moving pattern as well as starting positions. Firstly each piece must be instantiated at the correct position on the board as seen on this diagram:



We must then give each piece its own logic. Each piece can only move to a location if the location is Null (meaning no piece is stored in the location) or if it is the opposite colour of the piece, which then will in turn destroy the piece.

We must also, after every turn, append a text document which stores the movement of each piece. This is so a player can read this document to recreate the game by seeing the moves done in the game by both sides. We represent a piece with a shortened string and once a piece is moved, we then say the piece which moved and the location it moved to on the board. I will be using a variation of the Algebraic notation for this.

The notation is below for each piece and special move types:

Squares:

- x-axis = a-h - y-axis = 1-8 -example = b4

Pieces (pawns are not included):

-King = K -Knight = N -Queen = Q -Bishop = B -Rook = R -Example: Qb3b4

Captures:

-Insert an “x” e.g. Bishop takes e5 from d4 would be Bd4xe5

Special Moves:

-Pawn Promotion = pieceLocation + Promoting material. E.g. e8Q promotes the pawn at e8 to a Queen

-Castling = 0-0 (kingside castle) 0-0-0 (queenside castle)

Win/Draw:

-Draw offer = (=)

-Check/Checkmate = ++ (this is appended to the end of a move)

-Winner = 1-0, 0-1 ½-½ (white-black, 1 meaning winner, 0 meaning loser and ½ meaning draw)

Moving the Pieces -

If a piece moves to a location which is null, we must create a “move plate”, this is what allows a user to move a piece to a location. This can be done by creating a place on the board which the user can click on to move the piece to that location. However, if it the location is not null, and the piece it is attack is not its own colour, we create an “attack plate”, this will move to piece to that location, while also destroying the other

Def movePlate(xpos,ypos):

Instantiate(movePlate, xpos, ypos);#creates the attack plate from a prefab
movePlate.attack = false #setting the position on the board and if it can attack
movePlate.xpos = xpos
movePlaye.ypos= ypos

Def attackPlate(xpos,ypos):

Instantiate(attackPlate,xpos,ypos) #creates the attack plate from a prefab
movePlate.attack = true #setting the position on the board and if it can attack
movePlate.xpos = xpos
movePlaye.ypos= ypos

This creates a prefab at the location given. A prefab is a GameObject which can be reused and instantiated to have the same components.

Types of Moves -

We have two different types of moves which are possible, either input moves (meaning the piece moves to a specific coordinate depending on the location of the piece) or iterative move (meaning the piece moves in a straight line).

```
Def InputMove(xpos,ypos):

If (isValidPos( xpos, ypos) == true): #checks if values are between 0-8
        If (board(xpos,ypos) == null): #checks if spot is empty
                movePlate(xpos,ypos) #creates a move plate at x y
        Else if (board(xpos,ypos).playerColour != playerColour): #if is another player
                attackPlate(xpos,ypos)  #creates a move plate at x y
```

This means if the location is a valid spot, it will check either if it is null (no position there) or if the piece there is not of the same colour. This will then create a move plate, or an attack plate respectively

```
Def IterativeMove(xpos, ypos, changeX, changeY):

while (isValidPos(xpos,ypos) == true and board(xpos,ypos) == null):
        movePlate(xpos,ypos)
        xpos += changeX
        ypos += changeY
If (isValidPos(xpos,ypos) == true  and board(xpos,ypos).playercolour != playercolour)
        attackPlate(xpos,ypos)
```

This means while the location is valid and there isn't a piece there, it will create a move plate, however once it means a piece or it isn't valid anymore it will stop. Lastly, it will then check if the piece met is not the same colour, which it will then create an attack plate.

We must also check that once a piece moves, it does not cause a check. This is done by seeing the selected pieces moves in advance and calculating if this will cause a check. There is either the piece moves a piece which causes a check on the opponent's side, you move your own piece and causes a check on your side or you take a piece which currently was causing a check or would cause a check if you were to just move the piece.

After each possible legal move, if it is a move where no pieces are taken, we iterate through all pieces, updating a new attack board, and if the king's location is in the attack board, we do not allow the piece to move there and move it back. If the piece can take a piece, we do the same however remove the piece from the array and check if the king is in check, then put the piece back into the array after the check is done. It can be done like so:

```
moveCheck(newXPos,newYPos):
```

```
If piece!= “king”:
        If (board[newXPos,newYPos] != null: #checks if attack spot
                attackPiece = board[newXPos,newYPos]
                indexVal = board[newXPos,newYPos].getArray(pieces)
                board[newXPos,newYPos] = null #deletes the piece there
                pieces[indexVal] = null #deletes the piece there
        Board[xpos,ypos] = null
        board[newXPos,newYPos] = piece #sets current piece to that spot
        If kingInCheck == true: #if it moves and is in check
                board[newXPos,newYPos] = attackPiece
                pieces[indexVal] = attackPiece
                Board[xpos,ypos] = piece
                Return true #meaning cannot do move
        Else: #if not in check after moving
                board[newXPos,newYPos] = attackPiece
                pieces[indexVal] = attackPiece
                Board[xpos,ypos] = piece
                Return false #can do move

Return false
```

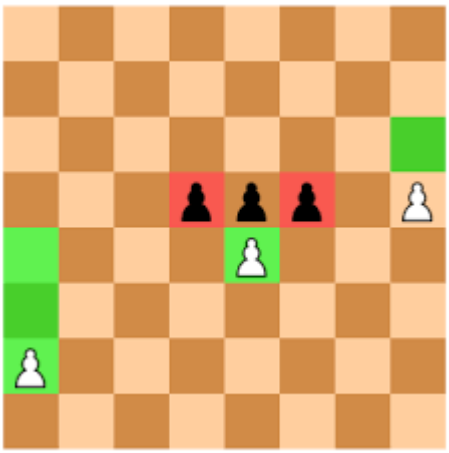
Types of Pieces -

Pawn - 

A pawn can move in different ways depending on the colour of the pawn. At the start, if the pawn hasn't moved yet, it has the option to move 2 places forward. This can be checked using a ‘HasMoved’ variable which keeps track if the piece has moved yet:

```
If playerColour == ‘black’: #a black pawn
        inputMove(xpos, ypos-1)#moves 1 place down the board
        inputMove(xpos + 1, ypos-1) #checks if it can attack
        inputMove(xpos - 1, ypos-1) #checks for attack
        If ‘hasMoved’ = false: #if it hasn't moved yet
                InputMove(xpos,ypos-2): #move 2 spaces forward
Else if playerColour == ‘white’: #if white pawn
        inputMove(xpos, ypos+1) #move 1 place up board
        inputMove(xpos + 1, ypos+1) #checks if it can attack
        inputMove(xpos - 1, ypos+1) #checks if it can attack
        If ‘hasMoved’ = false: #if it hasn't moved yet
```

InputMove(xpos,ypos+2): #move 2 place up board



This will then allow the pawn to move forward twice if it hasn't moved yet, else only once, then take either piece on the left or right

Lastly, a special move called “en Passant” can be used where if a pawn moves 2 places in advance, if there is a pawn of opposite colour either to the left or right of the pawn, it will allow the pawn to take the piece as if it only moved 1 slot.

When a pawn moves two spaces, we must check if there is a pawn next to it, we then assign it the variable ‘enPassant’ to true, this will allow the pawn to take it as if it only moved one space. However, it must take the pawn in the next turn else it forfeits the right to en Passant.

Def checkEnPassant(xpos,ypos):

If board(xpos + 1,ypos).name == pawn and board(xpos + 1,ypos).playercolour != playercolour: #if it is a white pawn and different colour

board(xpos + 1,ypos).enPassant = true #allow it to enpassent

board(xpos - 1,ypos).enPassantXPos = xpos + 1# sets coords where it can

```
        board(xpos - 1,ypos).enPassantYPos = ypos
Else if board(xpos - 1,ypos).name == pawn and board(xpos - 1,ypos).playercolour != playercolour: #if it is black pawn and different colour
        board(xpos - 1,ypos).enPassant = true
        board(xpos - 1,ypos).enPassantXPos = xpos - 1
        board(xpos - 1,ypos).enPassantYPos = ypos
```

This sets the variable enPassant to true and will set to the location it is true at to allow the pawn to en Passant

```
Def doEnPassant(enPassantXPos,enPassantYPos):

If enPassant == true: #if it can en passant
        attackPlate(enPassantXPos,enPassantYPos)
```

This checks if enPassant is true and if so will allow the piece to move there

```
Def resetEnPassant():

If currentPlayer = 'white': #iterates through all current sides piece and resets values
        For i in range(len(whitePieces)):
                If whitePieces[i].name = pawn and whitePieces[i].enPassant = true:
                        whitePieces[i].enPassant = false
Else:
        For i in range(len(blackPieces)):
                Else if blackPieces[i].name = pawn and blackPieces[i].enPassant = true:
                        blackPieces[i].enPassant = false
```

This resets enPassant every turn to make sure that if a piece did not en Passant then it cannot do it the next turn

Lastly, once a pawn reaches the end of the board, it can be promoted. The player can choose to choose any piece to promote it to (besides a pawn) and once this happens the pawn is then changed to the selected piece. This can be then checked every time the pawn is moved.

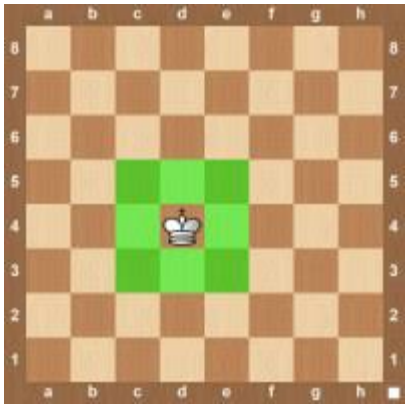
```
Def checkPromote():

If ypos = 0 or ypos = 7: #checks to see if reaches the end of the board
        If colour = "black":
```

```
            indexVal = findIndex(gameObject)# finds index of piece
            blackPieces[indexVal] = newPiece#assigns new piece
    Else if colour = “white”:
            indexVal = findIndex(gameObject) # finds index of piece
            whitePieces[indexVal] = newPiece #assigns new piece
    destroy(gameObject) # destroy the gameobject since we do not need it anymore
```

King: 

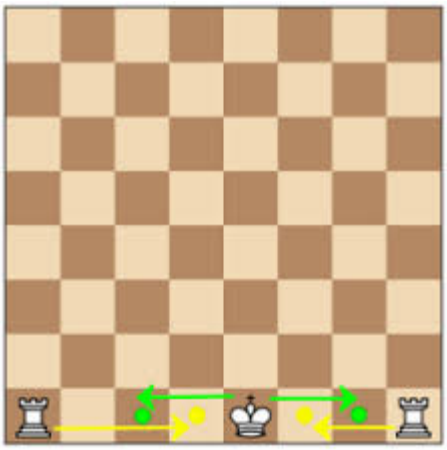
The king has 10 separate moves and uses the ‘hasMoved’ variable to keep track if the king has moved. The king has 8 input moves and 2 ways it can castle, king side or queen side.

```
Def kingMove(xpos, ypos):

inputMove(xpos + 1,ypos + 1):
inputMove(xpos,ypos + 1):
inputMove(xpos + 1,ypos - 1):
inputMove(xpos + 1,ypos):
inputMove(xpos - 1,ypos):
inputMove(xpos - 1,ypos + 1):
inputMove(xpos ,ypos - 1):
inputMove(xpos - 1,ypos - 1):
```



This creates a box around the king in which it can move.
Next we must check if the king can castle, a move which involves a castle and king which both move to a location depending on the side it is. You can only castle if you are not in check and the side next to you is empty, there is a castle at the end square, and if you castle, you cannot castle into check.

```
Def castleMove(xpos,ypos):

If hasMoved = false & inCheck = false:
        If board[7,7].name == “rook” and board[6,7] == null and board[5,7]== null and
blackAttackboard[6,7] != true:
                castle()  #black king side castle
        If board[0,7].name == “rook” and board[1,7] == null and board[2,7]== null and
blackAttackBoard[3,7] == null and blackAttackboard[2,7] != true:
                castle()  #black queen side castle
        If board[0,0].name == “rook” and board[1,0] == null and board[2,0]== null and
board[3,0] == null and  whiteAttackboard[2,0] != true:
                castle() #white king side castle
        If board[7,0].name == “rook” and board[6,0] == null and board[5,0]== null and
whiteAttackboard[6,0] != true:
                castle() #white queen side castle
```



Lastly, we must make sure that the king cannot move into check, this is done by checking if the location is in the opposite side's attack positions.

```
Def kingCheckMove():

If colour = “white”:
        If blackAttackPos[newXPos,newYPos] != true:
                doMove()
Else:
        If whiteAttackPos[newXPos,newYPos] != true:
                doMove()
```

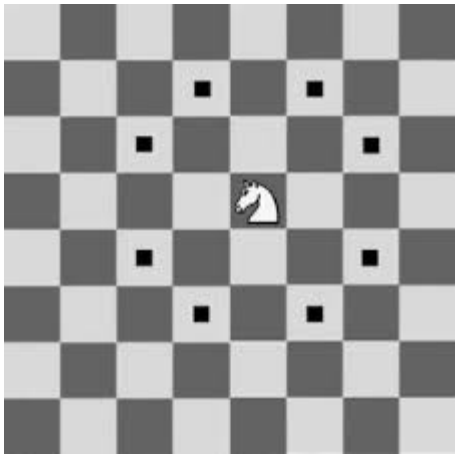
Knight: 

The knight has 8 moves, all input moves.

Def knightMove():

inputMove(xpos + 2,ypos - 1):
inputMove(xpos + 2,ypos + 1):
inputMove(xpos - 2,ypos + 1):
inputMove(xpos - 2,ypos - 1):
inputMove(xpos + 1,ypos + 2):
inputMove(xpos - 1,ypos + 2):
inputMove(xpos + 1,ypos - 2):
inputMove(xpos - 1,ypos - 2):

This creates where the knight can move like so:

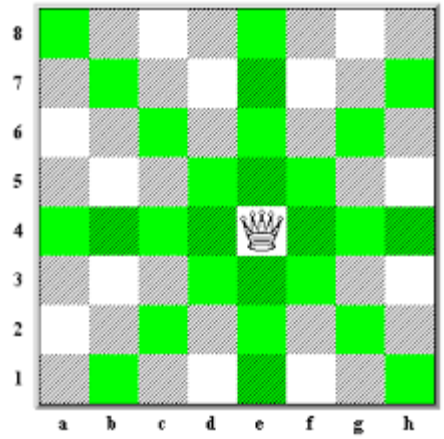


Queen: 
The queen has 8 moves, all being iterative (meaning it moves in a straight line or a diagonal). It can be done like so:

Def queenMove():

iterativeMove(xPos,yPos, 1,0)
iterativeMove(xPos,yPos, 0,1)
iterativeMove(xPos,yPos, -1,0)
iterativeMove(xPos,yPos, 0,-1)
iterativeMove(xPos,yPos, 1, 1)
iterativeMove(xPos,yPos, -1, -1)
iterativeMove(xPos,yPos, -1, 1)
iterativeMove(xPos,yPos, 1, -1)

This allows the piece to move like so:



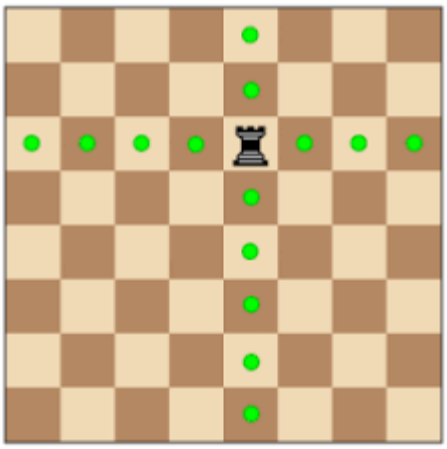
Rook: ♖♜

A rook has 4 moves, all where they move in a straight line. It is done like so:

Def rookMove():

iterativeMove(xpos,ypos,1,0)
iterativeMove(xpos,ypos,-1,0)
iterativeMove(xpos,ypos,0,1)
iterativeMove(xpos,ypos,0,-1)

This allows the piece to move like so:



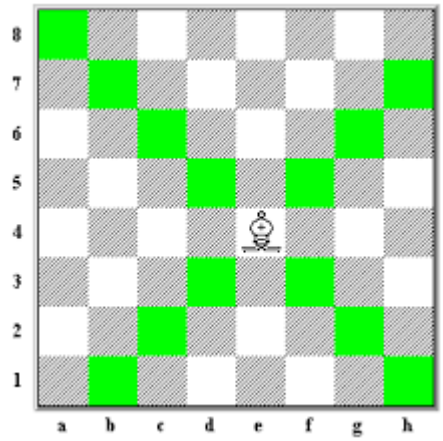
Bishop: ♗♝

A bishop has 4 moves, where all are diagonal. It is done like so:

Def bishopMove():

iterativeMove(xpos,ypos,1,1)
iterativeMove(xpos,ypos,1,-1)
iterativeMove(xpos,ypos,-1,1)
iterativeMove(xpos,ypos,-1,-1)

This allows the piece to move like so:



This concludes the majority of the basic piece logic / legal moves that a piece can do.

Win/Draw/Loss Logic -

There are multiple ways to win or draw. To win, the king must be in checkmate, this is where there are no possible ways a piece can move to where it will then not be able to be taken. This is done like so:

Def checkmateCheck():

```
If inCheck == true:
        possibleMoves = 0
        For i in range len(pieces):
                possibleMoves += getPossibleMoves(): #returns all possible moves a
```

```
piece can do
        If possibleMoves = 0: #if it can't move anymore
                gameEnd()  #checkmate
```

This will iterate through all pieces and find if there are any legal moves, if there are none the game will end. Similarly, stalemate is a possible ending where there are no legal moves a piece can do however the king is not in check.

```
Def stalemateCheck():

If inCheck == false:
        possibleMoves = 0
        For i in range len(pieces):
                possibleMoves += getPossibleMoves(): #returns all possible moves a
piece can do
        If possibleMoves = 0: #if it can't move anymore
                gameEnd() #stalemate
```

Next, a player can either agree to a tie, or to a loss. This can be done by giving a button where if it clicks, calls the gameEnd() function.
Lastly, when winning, there is a lack of time, where both players are giving equal timings, where they decrease over time during their turn. If the timer goes down to 0, then the person with no time left loses.

Insufficient material is when either side has pieces which cannot force a check meaning the game will never end, this includes:

- **king vs king**
- **knight, king vs king**
- **bishop, king vs king**
- **rook, king vs rook, king**
- **bishop, king vs bishop, king** (where both bishops are on the same square)
- **rook, king vs bishop, king**
- **rook, king vs knight, king**
- **knight, knight, king vs king**

This can be checked where at the end of each turn, check to see if either side pieces and if they correlate to that combination.

Next is the 50 move rule, where either side has yet to move a pawn, or take a piece for 50 moves (a move meaning both white and black do their turn). This can be kept in a counter, called 50Moves where it is set to 0, once a piece is moved or taken in the MoveTile class, we check if it is a pawn or an attackTile, if it is then we reset the counter, however if it isn't, we then reset the counter. We then continue this for until the counter reaches 100, to

where it is then a draw and calls the gameEnd() function.

Lastly, the trifold repetitions is where the positions on a board are repeated three times, meaning all pieces are at the same position. We can keep it in an array which stores the 8x8 locations of each piece on a board. We reset this once a pawn or piece is taken, and can store over 100 board positions as at that point the 50 move rule would occur. After each move, we check to see if the location is repeated three times, if so it will call the gameEnd function.

AI Logic -

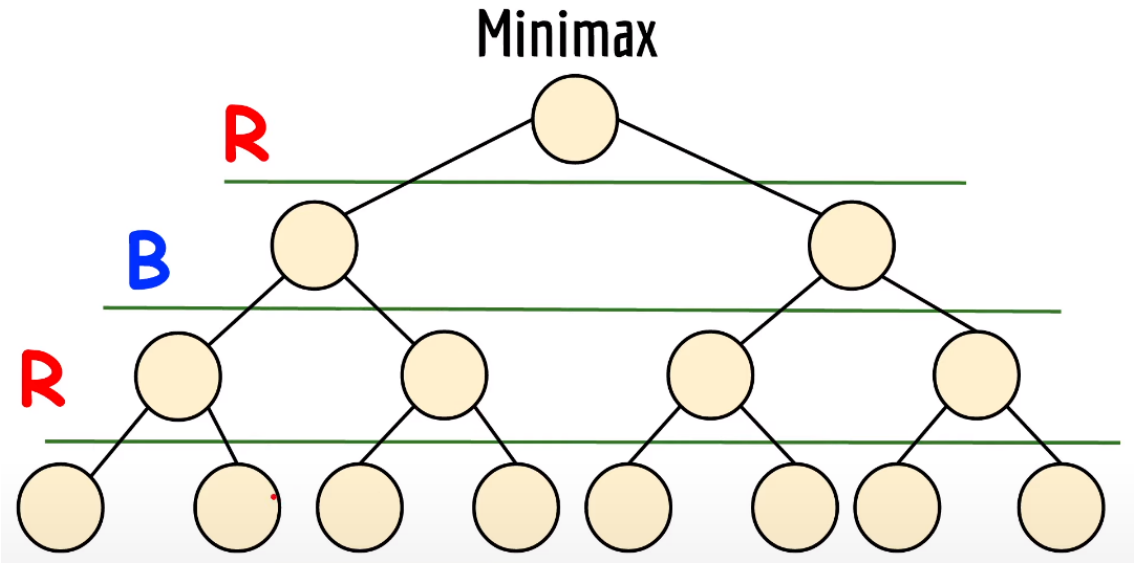
There are two ways for a chess AI, you can either use an API or code it yourself. An API can be used as it is likely a better or more efficient algorithm has already been made meaning not only would it make the AI stronger, it would use less resources. However, this approach is much easier and takes less time compared to making one yourself.

An approach to make a simple chess AI could be of two ways. A simple way would be to randomly choose a piece, then to randomly move the piece to a legal position. This can be done like so:

```
Def randomMove():
While true:
        ranVal = random.randint(0,len(pieces)-1)#a random number between 0 and
length of array
        Moves =pieces[ranVal].getPossibleMoves() #gets all moves
        If Moves !=0:
                Break
ranVal = random.randint(0,len(Moves)-1) #gets a random move
doMove(ranVal)#does the move
```

While this technically isn't an AI, it can be used as an easy bot as it will have no knowledge of what to do and will randomly do a move.

However an easy approach which gives an AI a sense of intelligence is the MiniMax algorithm. The algorithm works like so:



So the starting node has two branches meaning you have two possible moves, in reality, it could be 80 moves, however during the start of the game there are 20. The moves will then have an outcome if it moves to a location stored as integer values, these are determined by the location of the piece as well as if a piece takes another piece, how much it gains. Then the opposite player has multiple moves it can take from those locations as so on. The depth of the AI is determined by how many moves it looks at in advance but must calculate all positions all moves and positions for them meaning the bigger the depth the longer it takes.

So the red nodes are trying to find the best outcome out of the possible leaf nodes it can compare to, each node compares to what it is branched off from, meaning in this case there are two nodes it compares to and finds the highest possible number.

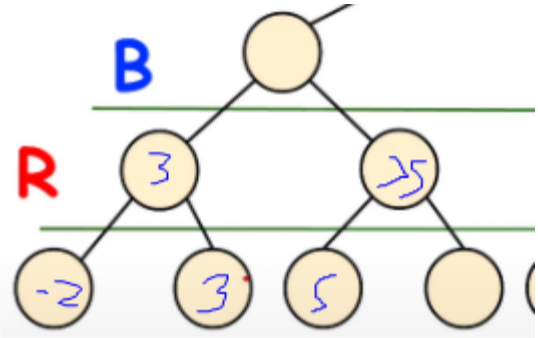
The blue nodes are trying to find the worst outcome, this is to minimize the damage of the moves that blue takes. This means it must find the smallest number.

This repeats till we reach the top node, this will then be the move the AI does as it will be the best calculated move possible for the AI.

During the start of a game, there are 40 possible combinations that can be done (where both white and black move) this then continues to 400 for the second, 8902 for the third and then finally 197,281 moves for the third. This means there are a total of 197,281 games it must calculate to just find the best positions it can go to. This means anywhere even near a depth of 3 can take seconds, even minutes to calculate.

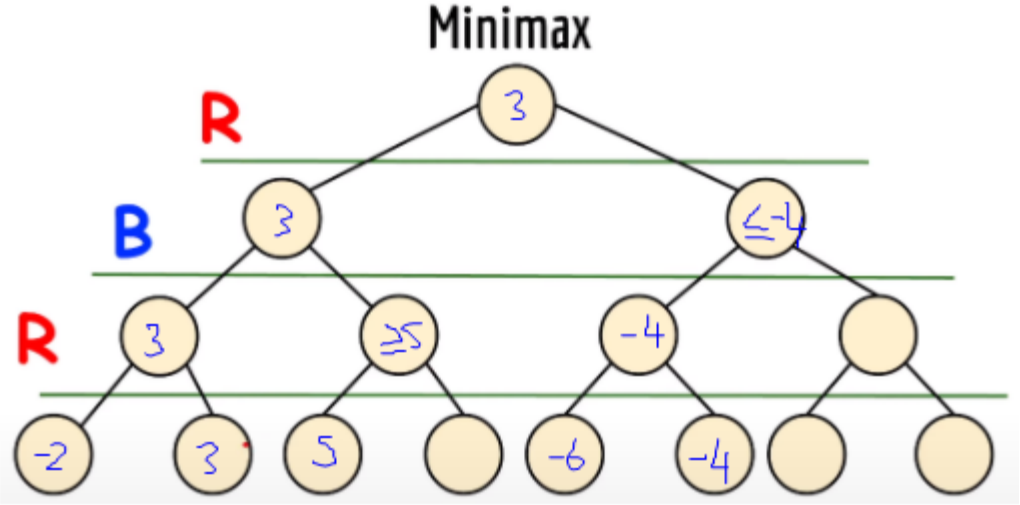
This is where we use a process known as Alpha-Beta pruning. When calculating each leaf

node, we can instead use this algorithm to determine whether or not it is worth calculating some other leaf nodes. Let's say we have these values:



We already know the first red node is 3, and since the blue node tries to minimise the values, the other node must be smaller. We calculate that one node is already 5, meaning that node will be >=5. This means there is no reason to calculate the other leaf node as it will already choose 3. This can significantly reduce the time to calculate each possible move as we cut down many possible calculations which were already necessary.

Finished example:



This can be coded like so:

Def minimax(positions, depth, maxPlayer):

If depth == 0 or checkIfGameEnd(positions) == true: #if in the state the game ends
Return calculate() #evaluates the position of board

```
If maxPlayer == true: #blacks turn
        maxEval = +∞ #positive infinity
        Foreach possibleMoves(positions): #gets all possible moves and
                Evaluate = minimax(positions, depth - 1, false)
                maxEval = max(maxEval, evaluate) #returns whats higher
        Return maxEval
Else: #whites turn
        maxEval = -∞ #negative infinity
        Foreach possibleMoves(positions): #gets all possible moves
                Evaluate = minimax(positions, depth - 1, true)
                maxEval = max(maxEval, evaluate) #returns whats higher
        Return maxEval
```

This will allow for any value of depth assuming it is positive, however the higher the depth the longer the time to calculate as this does not include the alpha-beta optimisation.

We can calculate the positions of each player on the board by checking how much they cost and adding it to their total. Them subtracting the opposite colours side. If it is positive it is better for the AI else if it is negative its winning for the other side.

```
Def calculate():

whiteWorth = 0
blackWorth = 0
For i in range(len(pieces)): #goes through all pieces
        If piece.colour = “white”: #if its a white piece
                whiteWorth += pieces[i].pieceWorth #gets the piece worth and adds
        Else: #black piece
                blackWorth += pieces[i].pieceWorth #gets the piece worth and adds
Return blackWorth - whiteWorth
```

Database -

There are many different online databases with different APIs or we can host our own database. In this case, I will use an SQL server which I host myself. This can be achieved with the SQL server 2016 developer on the microsoft website. The design for it is as so:

PlayerLoginData(userEmail, userAccountName userHashedPassword, userQuestion1)

PlayerMatchData(playerPlace, playerWins, playerLosses, playerDraws, *userAccountName*)

When the user selects the online option, he will be able to create or sign-in with an account. The user will input their credentials and will store them in the database. Using a hashing algorithm, the users password will be stored, this can be in combination with their email address.

After a user wins, loses or draws a game, their count will be increased as long as a connection to the server can be made.

Entity relationship Model -

Entity Description -

PlayerLoginData(userEmail, userAccountName userHashedPassword, userQuestion1)

PlayerMatchData(playerPlace, playerWins, playerLosses, playerDraws, *userAccountName*)

Entity Data Description -

PlayerLoginData

Attribute	Type	Purpose
userEmail	PrimaryKey, Varchar(30), Not Null	stores the users email address during signup

userAccountName	Foreign Key, Varchar(20), Not Null	stores the user's accountName
userHashedPassword	Varchar(50), Not Null	stores the user's hashed password

PlayerMatchData

Attribute	Type	Purpose
userAccountName	Primary Key, Varchar(20), Not Null	stores the user's accountName to link the two entities together
playerWins	Integer, Not Null, Default 0	stores the user's win count
playerLosses	Integer, Not Null, Default 0	stores the user's loss count
playerDraws	Integer, Not Null, Default 0	stores the user's draw count

SQL -

When creating an account:
Insert into PlayerLoginData(userEmail, userAccount, userHashedPassword)
Values _____

Checking signing into a game
Select userAccountName
From PlayerLoginData
Where userAccountName = _____ and userHashedPassword = _____

Checking if an account name / email is used:
Select userAccountName, userEmail
From PlayerLoginData
Where userEmail = _____ or userAccountName = _____

When a player wins a game:
Update PlayerMatchData
Set playerWins = playerWins + 1
Where PlayerLoginData.userAccountName = PlayerMatchData.userAccountName and PlayerLoginData.userAccountName = _____

When a player loses a game:
Update PlayerMatchData
Set playerWins = playerLosses + 1

Where PlayerLoginData.userAccountName = PlayerMatchData.userAccountName and PlayerLoginData.userAccountName = _____

When a player draws a game:
Update PlayerMatchData
Set playerWins = playerLosses + 1
Where PlayerLoginData.userAccountName = PlayerMatchData.userAccountName and PlayerLoginData.userAccountName = _____

Screen design

Main Menu -

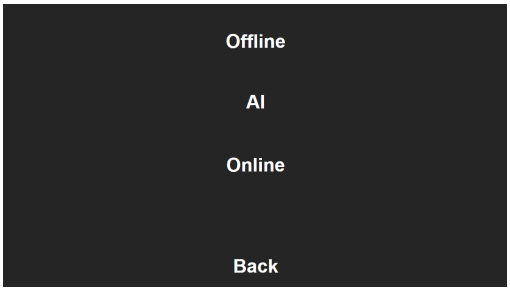
Once the program is run, the main menu is shown where three options are given, “Play”, “How To Play” and “Exit”. If the user selects “Exit” the program is terminated.

How to Play Menu -

Text and images are given to show the user a very basic way of using the software as well as how to play chess. They may also select “Back” to go back to the “Main Menu”. This is to allow users with little knowledge of how the application works as well as how check works will be able to know how to play.

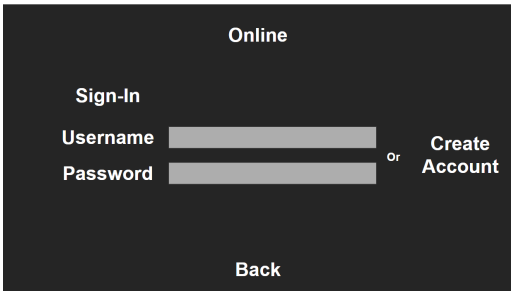
Play Menu -

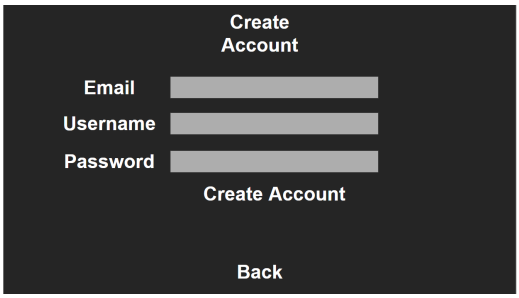
Four options are given in the Play menu, if “Back” is selected it returns back to the “Main Menu”. Both “Offline” and “AI” will place a user into a game whilst the “Online” option will go to the “Online Menu”. This is used so users can easily select between game modes.



Online Menu -

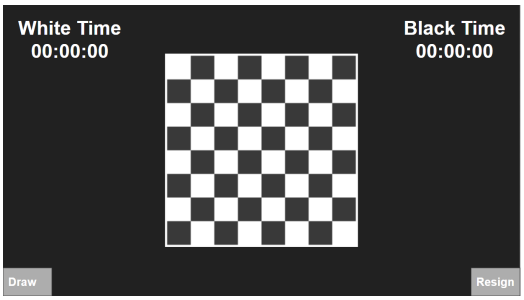
This menu is used to allow users to sign in to an account or select to make an account. The users will then be placed into a match against each other where the ending of each game is recorded onto an online database.





Board / Game Menu -

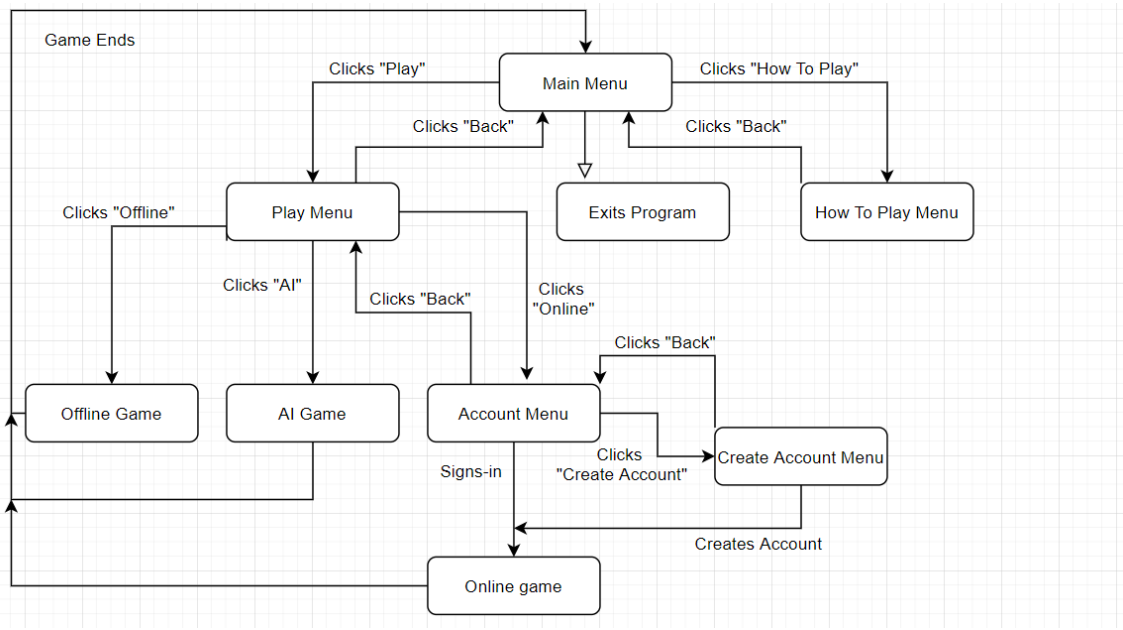
Lastly, the Game menu is where each game is played, where depending on the setting selected, you can play either against another player, or an AI. Two buttons can be selected “Draw” and “Resign” where either offer a draw to the other player or resign the current player respectively.



Systems diagram

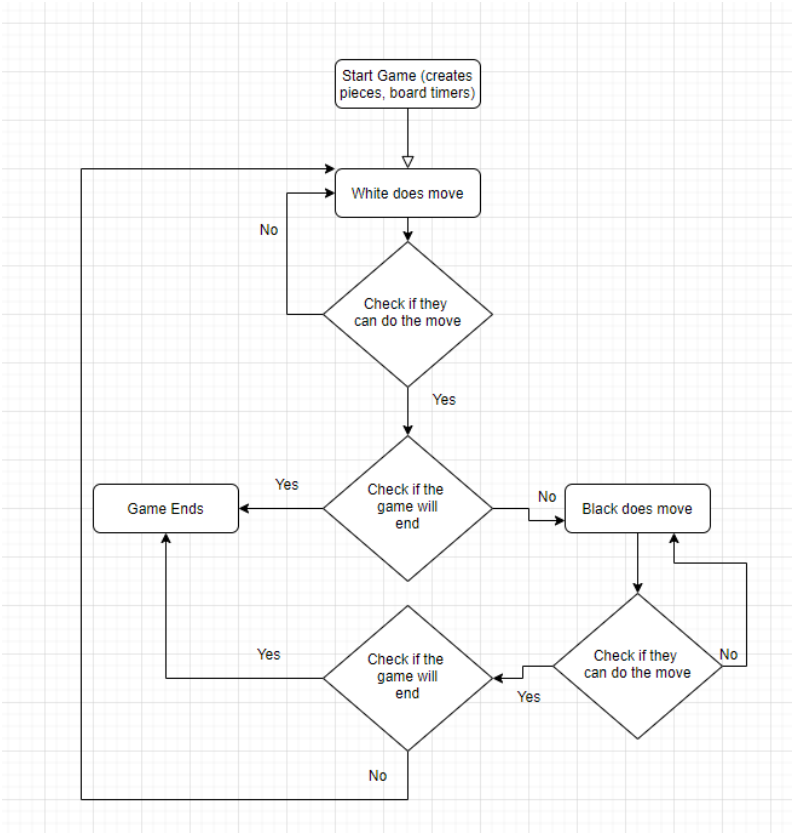
Menu Flowchart-

This is the diagram for the main menu and all the different paths it can take, where if you click a button or input data, where it will go



Game Flowchart-

This is a diagram of how the basic rules of chess works and what happens when you create a game.



Test Table

Menu / GUI Tests -

Used to test if all buttons go to the correct place in the main menu

No	Test	Data	Expected Outcome
1ai	Clicking on “Play” in the Main Menu		Opens the “Play Menu”
ii	Clicking on “How To Play” in the Main Menu		Opens the “How To Play Menu”
iii	Clicking on “Exit” in the Main Menu		Closes the application
b	Clicking “Back” on any menu screen		Takes you to the previous menu screen you last were on
ci	Clicking on “Offline” in the Play Menu		Creates a new game in an offline environment
ii	Clicking on “AI” in the Play Menu		Creates a new game where your opponent is an AI
iii	Clicking on “Online” in the Play menu		Opens the “Login Menu”
di	Clicking on “Create Account” in the Login menu		Open the “Create Account Menu”
ii	Clicking on “Sign-in” in the Login menu		Creates a new game in an Online environment
2ai	Input correctly formatted information into each field when creating a new account	Email = “Email@address.com” Password = “Password”	Creates the account and signs in to the account
ii	Input already used email or account names when creating a new account	Email = “Email@address.com” Password = “OtherPassword”	Does not create the account and will ask to redo with a separate username / email.
bi	Inputting incorrect passwords	Email =	Does not sign into the account

	with an email address when logging in	“Email@address.com” Password = “WrongPassword”	and will ask to redo
ii	Inputting correct fields into the login screen	Email = “Email@address.com” Password = “Password”	Login to the account and create a new game in an online environment

General Board Tests -

Used to test all board gui as well as creates pieces, allows pieces to move, as well as basic rules of chess

No	Test	Data	Expected Outcome
3a	Creating a new game		A board containing 16 piece on each side in the format below:  Also, creates two timers which decrease on the turn of the player which it is attached to
3bi	Clicking on “Resign” button		Opens the “Resign Menu”
ii	Click “Yes” in the Resign Menu		Ends the game and the player who resigned loses. The “Play Again Menu” will also be opened
iii	Clicking “No” in the Resign Menu		Returns to the game
3ci	Clicking on “Draw” button		Opens the “Draw Menu”
ii	Click “Yes” in the Draw Menu		Ends the game and both players draw. The “Play Again Menu” will also be opened

iii	Clicking “No” in the Draw Menu		Returns to the game
3di	Click “Yes” in the “Play Again Menu”		Creates a new chess game
ii	Click “No” in the “Play Again Menu”		Opens the Main Menu
ei	Moving a white piece		Stop the white timer and resume the black timer. Updates the new position of the piece onto the board.
ii	Moving a black piece		Stop the black timer and resume the white timer. Updates the new position of the piece onto the board
fi	Once a timer reaches 0		Ends the game and the player whose time ran out will lose. The “Play Again Menu” will also be opened
ii	Once checkmate is reached		Ends the game and the current player wins. The “Play Again Menu” will also be opened.
iii	Once stalemate is reached		Ends the game and both players draw. The “Play Again Menu” will also be opened.
iv	No pawn is moved or piece is taken after 50 moves (meaning black and white moves)		Ends the game and both players draw. The “Play Again Menu” will also be opened.
v	The exact position has been repeated 3 times		Ends the game and both players draw. The “Play Again Menu” will also be opened.
vi	Last remaining piece are king vs king		Ends the game and both players draw. The “Play Again Menu” will also be opened.
vii	Last remaining piece are knight, king vs king		Ends the game and both players draw. The “Play Again Menu” will also be opened.
viii	Last remaining piece are		Ends the game and both players

	bishop king vs king		draw. The “Play Again Menu” will also be opened.
ix	Last remaining piece are rook, king vs rook, king		Ends the game and both players draw. The “Play Again Menu” will also be opened.
x	Last remaining piece are bishop, king vs bishop, king (where both bishops are on the same square)		Ends the game and both players draw. The “Play Again Menu” will also be opened.
xi	Last remaining piece are rook, king vs bishop, king		Ends the game and both players draw. The “Play Again Menu” will also be opened.
xii	Last remaining piece are rook, king vs knight, king		Ends the game and both players draw. The “Play Again Menu” will also be opened.
xiii	Last remaining piece are knight, knight, king vs king		Ends the game and both players draw. The “Play Again Menu” will also be opened.

Piece Tests -

Used to test if all pieces will move correctly

No	Test	Data	Expected Outcome
4ai	Clicking on a piece of the same colour to the turn it is		Shows all available moves on the board the piece can do
ii	Clicking on a piece of the opposite colour to the turn it is		The piece shouldnt be able to move
b	Taking a piece with another piece of an occupied spot of different colour		Deletes the piece and adds the amount of points the piece was worth
c	Move the piece onto the location of an occupied spot of same colour		The piece should not move and should stay where it is
d	Moving a piece into check of your own king / while you king		The piece should only be able to move if it will stop the check

	is in check occurs		
5ai	Clicking a pawn once it has not moved yet and is not being blocked		Two plates are created, if it is white both above the piece, else two below the piece
ii	Clicking a pawn once it has already moved and is not being block		Only being able to move forward one space
iii	Moving a pawn two spaces forward next to a pawn of opposite colour		The pawn should be able to en passant but for only one turn
bi	The king or castle has moved and you try to castle		The piece will not be able to move
ii	The king or castle hasn't moved and you try to castle		The piece should move
iii	Try to castle while under check		The piece will not be able to move
iv	Moving the king into checkmate		The piece will not be able to move
v	Castling through an attacking square		The piece will not be able to move
6a	Clicking a pawn		Shows correct move pattern
b	Clicking a castle / rook		Shows correct move pattern
c	Clicking a bishop		Shows correct move pattern
d	Clicking a queen		Shows correct move pattern
e	Clicking a king		Shows correct move pattern
f	Clicking a knight		Shows correct move pattern

AI Tests -

Used to test if an AI can move pieces correctly

No	Test	Data	Expected Outcome

7	Making the AI move a piece		AI does a legal move and rotates to other players turn

Online Tests -

Used to test if the online option is used and once after the game ends, the game will correctly store the results of the game in the database per person.

No	Test	Data	Expected Outcome
8a	Player 1 wins, Player 2 loses		Player one's win count will be increased by one and player two's loss count will increase by one
b	Player 1 loses, Player 2 wins		Player two's win count will be increased by one and player one's loss count will increase by one
c	Both players draw		Both players draw counts are increased by one

Data Requirement -

As my chess program already has the data needed and does not use any data from any external program, user or API, there is no data processing required. This is because all the data needed for the program to work is already defined in code. While the program could be improved using external data given to the program, such as using an API for the chess AI, which tells where the AI should move, this is not the approach that I will use as it may increase file size or require an internet connection which I do not want to be necessary feature

3. Developing the Coded Solution

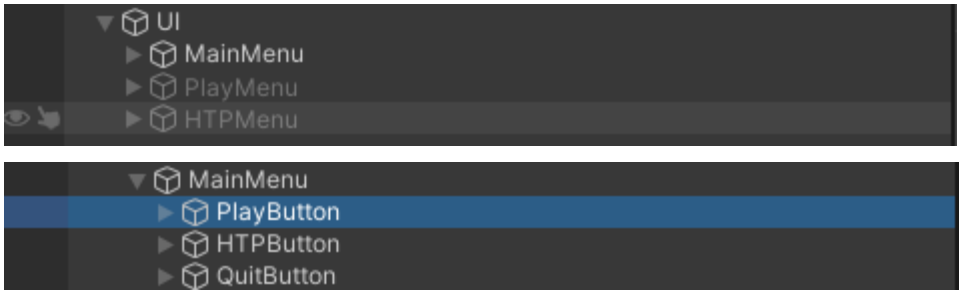
Stage One Of Development:

In this stage, I will begin the project by creating the base UI as well as simple chess rules and moves.

Main Menu:

Firstly, I created a basic scene that contains some buttons that can change between different menu options. While this may not be the final UI layout, it is basic enough to allow the user to navigate the program easily, where images and more can easily be added without disrupting the base game.

Each menu can be separated into different game objects so once we click on the button, it will go to a new screen of different buttons. This will also have buttons and images which will help navigate the user around the program, allowing them to select options that suit their needs.



We assign each button a script which we write in our UI Class, or a default selection option in Unity settings.

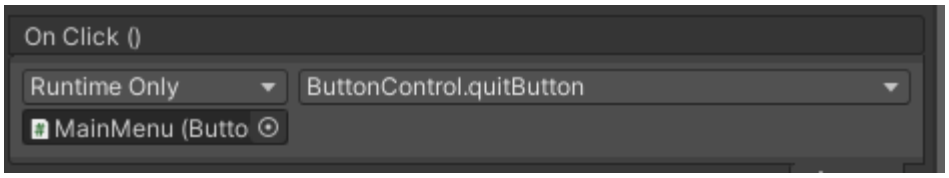
Play Button:

This will change the current menu and set the current active menu to the “PlayMenu”.

HTP Button:

This will change the current menu and set the current active menu to the “HTPMenu”.

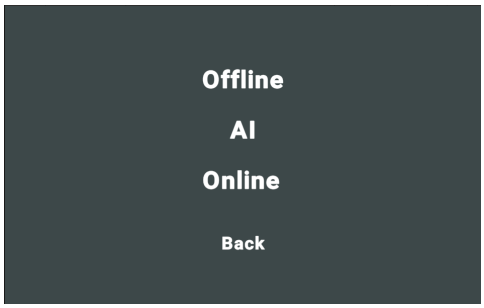
Quit Button:



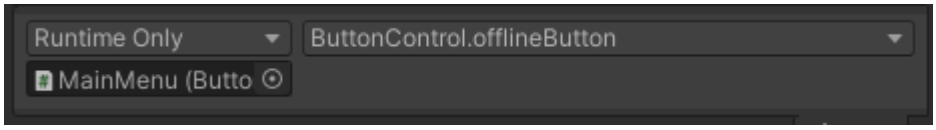

```
public void quitButton()
{
    //Quits the application
    Application.Quit();
}
```

This will quit the application and close it.

Play Menu:



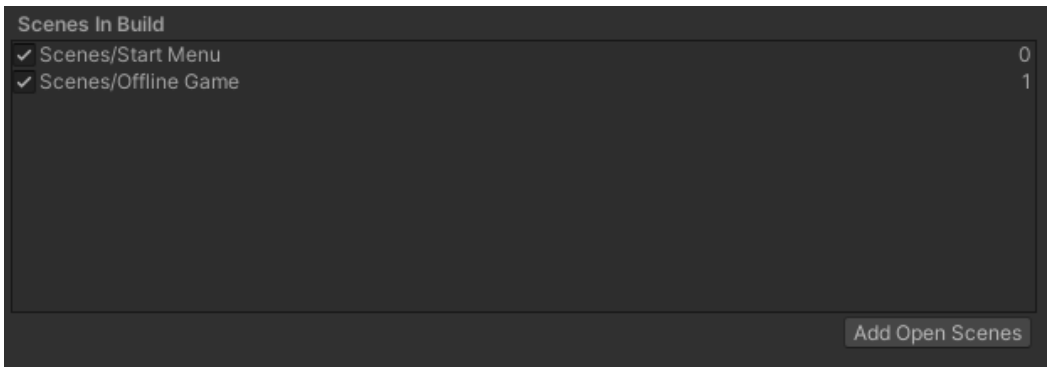
Both the “Offline”, “AI” and “Online” button, when clicked will change the current scene to the game, however introduce different settings.



It will call the current script below:

```
public void offlineButton()
{
    //loads the next scene in the build manager section (in this case offline)
    SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
}
```

This will change the current scene to the offline scene, this can be changed in the build settings if more modes are added.



Back Button:

This will go back to the original menu “Main Menu”.

How To Play Menu:

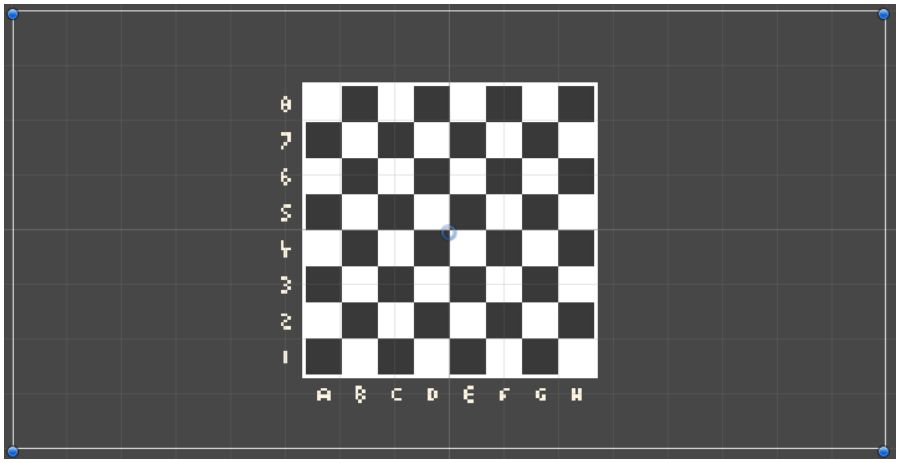
Text and images will be displayed on how to use the program, where next will scroll between text, and back going back to the original menu.

Back Button:

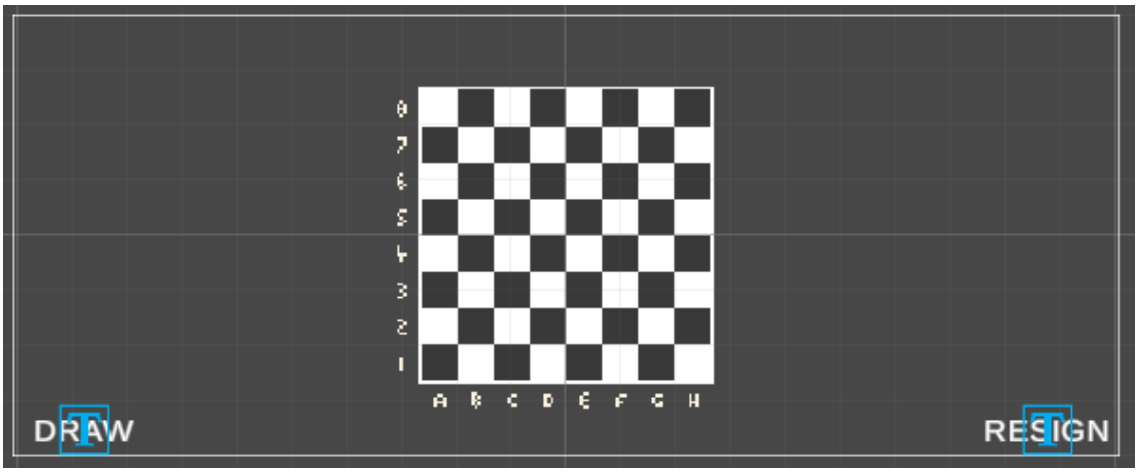
Board UI:

Once the user selects the corresponding settings in the “Play Menu”, we load a new scene which holds the code and assets for the main chess game. We first create a UI for the board, timer and optional buttons to offer a draw or to surrender a game.

Firstly, I created a board which is centered to the middle of the screen, with allocated board location notation, the camera will be fixed to the board and will sit in the middle of the screen.

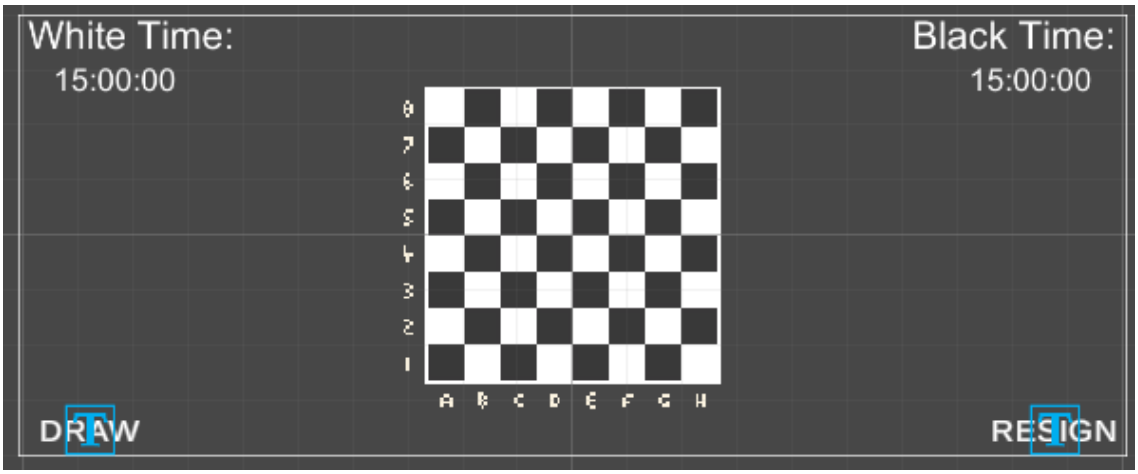


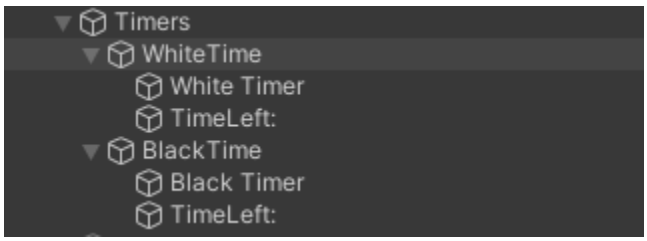
Next I added buttons which currently are not assigned to anything, but will allow the user, once implemented, to resign or draw the game, assuming either player agrees to it. When clicked on, the user will be given a menu where they can be made sure that they agree to resigning or drawing.



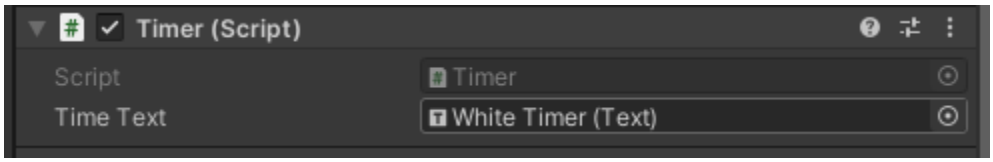
Both buttons currently are not assigned, but later on when the ability to lose or draw a game, they will be implemented.

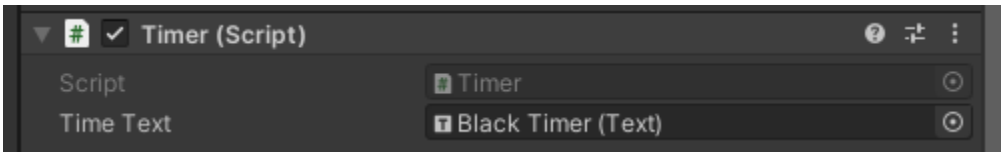
Lastly, I added the Timers, both will need a script which updates the time for each user and will cycle for each user's turn. Unity luckily has an update function which is called every frame, and allows us to calculate the change in timer per frame, meaning we can create a timer.





Both the white timer and the black timer are made of separate blocks of text, one which is static saying “White Time:” or “Black Time:”, and then the other which can change with the script. Both timers are given the script so their text can change with time:

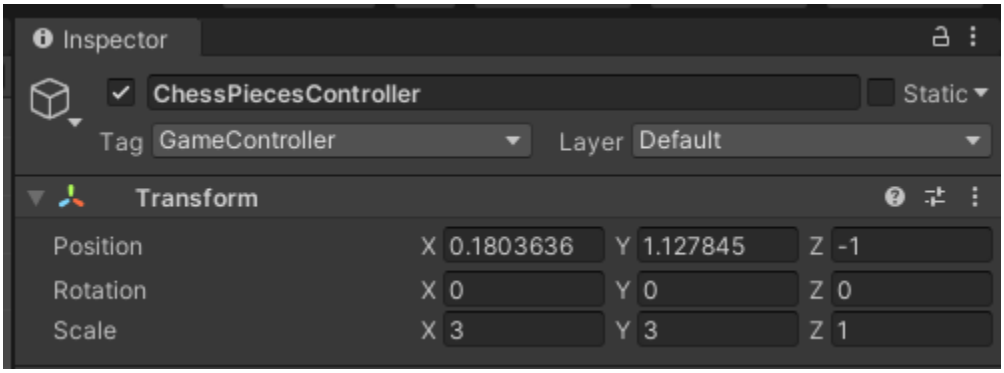




Board, Pieces & Tiles Class:

Main Class:

The board class, called “Main” in this case, is used to store the current board positions, controls the board rules and user rules as well as instantiates all pieces for the user to use. We must firstly have a way to create our instance of the board, this is done by assigning it to an empty game object, which we can call “ChessPiecesController”, which we also give a tag to of “GameController” such that we can easily access it between scripts.



We then create a new script which controls all the board functions, which we can call main and assign it to the controller. We must have a way to store the positions of different pieces on the board, this is so we can know where each piece is on the board. We also must know which is the current player, how many points each side has as well as settings to know if a game will be online or offline.

```
public GameObject ChessPieces; //used to reference ChessPiece prefab

//Creating GameObject arrays
//stores the locations of the gameobjects on an 8x8 board on a 2D array (xPos,yPos)
public GameObject[,] boardPositions = new GameObject[8, 8];
//used to store the current instatiated pieces
public GameObject[] whitePieces = new GameObject[16];
public GameObject[] blackPieces = new GameObject[16];
//used to store the points gained from each
public int blackPoints;
public int whitePoints;
//stores current amount of pieces on board
public int blackPiecesLength = 16;
public int whitePiecesLength = 16;
//used to store the current player and if the game is over and if playing against AI
public string currentPlayer = "white";
public bool canMove = true;

public bool againstAI = false;
public bool isOnlineGame = false;
```

We must then create pieces to be stored in each array. Whilst we don't have a pieces class, we can still create each piece, but it won't be correctly assigned yet. We must assign this to the start of the program so that each piece is assigned at the start without having a user to click any buttons

```
void Start()
{
    //this assigns each piece and places them into the associated array
    whitePieces = new GameObject[] {
        createPiece("WR",0,0), createPiece("WKn",1,0), createPiece("WB",2,0), createPiece("WQ",3,0), createPiece("WKi",4,0),
        createPiece("WB",5,0), createPiece("WKn",6,0), createPiece("WR",7,0),
        createPiece("WP",0,1), createPiece("WP",1,1), createPiece("WP",2,1), createPiece("WP",3,1), createPiece("WP",4,1),
        createPiece("WP",5,1), createPiece("WP",6,1), createPiece("WP",7,1) };
    blackPieces = new GameObject[] {
        createPiece("BR", 0, 7), createPiece("BKn", 1, 7), createPiece("BB", 2, 7), createPiece("BQ", 3, 7), createPiece("BKi", 4, 7),
        createPiece("BB", 5, 7), createPiece("BKn", 6, 7), createPiece("BR", 7, 7),
        createPiece("BP", 0, 6), createPiece("BP", 1, 6), createPiece("BP", 2, 6), createPiece("BP", 3, 6), createPiece("BP", 4, 6),
        createPiece("BP", 5, 6), createPiece("BP", 6, 6), createPiece("BP", 7, 6) };

    for (int currentPiece = 0; currentPiece < 16; currentPiece++)
    {
        assignBoardPos(whitePieces[currentPiece]);
        assignBoardPos(blackPieces[currentPiece]);
    }
```

Each piece will have a board location of an x and y coordinate, as well as the name of each type of piece so we can differentiate them later on.

```
public GameObject createPiece(string name, int xPos, int yPos)
{   //Creates a piece, assigning the name, xpos and ypos to it
    GameObject currentGO = Instantiate(ChessPieces, new Vector3(0, 0, -1), Quaternion.identity);
    Pieces cp = currentGO.GetComponent<Pieces>();
    cp.name = name;
    cp.xPos = xPos;
    cp.yPos = yPos;
    cp.assignPiece();
    return currentGO;
}
17 references
public void assignBoardPos(GameObject currentPiece)
{
    //gets xpos and ypos of the piece
    Pieces cp = currentPiece.GetComponent<Pieces>();
    boardPositions[cp.xPos, cp.yPos] = currentPiece;
}
```

Basic chess Rules:

Before we create the pieces class, we must create a few rules such that the game does not break easily. Firstly, to make sure that a move is valid, we must make sure that the piece cannot move to a location which is outside the 8x8 board:

```
public bool validBoardPos(int x, int y)
{
    //used to see if the positions which are used are valid in the board (must be between 0-7)
    if (x < 0 || y < 0 || x > 7 || y > 7)
    {
        return false;
    }
    return true;
}
```

Next we want to be able to swap between players for each turn, so that after each move, the current player variable is swapped, this in turn must also stop both timers:

```
public void changeCurrentPlayer()
{
    //used to change the current player
    switch (currentPlayer)
    {
        case ("white"):
            currentPlayer = "black";
            break;
        case ("black"):
            currentPlayer = "white";
            break;
    }
    //rotates timers
    GameObject whiteTimer = GameObject.FindGameObjectWithTag("WhiteTimer");
    whiteTimer.GetComponent<Timer>().cyclePause();
    GameObject blackTimer = GameObject.FindGameObjectWithTag("BlackTimer");
    blackTimer.GetComponent<Timer>().cyclePause();
}
```

UI and Timers:

Timer Class:

The timer class allows us to have a sense of time on how long each user can move for, rotating between turns. We need to be able to store the beginning time for each user, know whether or not the timer is paused, and reference the text that will be edited:

```
private GameObject controller;
private bool paused = true;
private float currentTime = 900; //15 minutes in seconds
public Text timeText;
```

We then must update the time to change, we can do this by calculating the change in time between frames and updating the timer with this. We, however, only want to do this if the timer is currently not paused such that it doesn't continue.

```
if (!paused && controller.GetComponent<Main>().canMove)
{
    if (currentTime > 0)
    {
        //calculate change in time per frame
        currentTime -= Time.deltaTime;
        convertToText();
    }
```

Next, if the timer reaches zero, we then want to end the game as that user has lost:

```
else
{
    //user runs out of time
    currentTime = 0f;
    convertToText();
    //ends the game
    Main endGame = controller.GetComponent<Main>();
    endGame.gameEnd(endGame.currentPlayer);
}
```

We then need to convert the text, so it updates on the users screen, this is done by formatting the text as seen below:

```
public void convertToText()
{
    //converting float to text in time format of minute : second : milliseconds
    timeText.text = string.Format("{0:00}:{1:00}:{2:000}", Mathf.FloorToInt(currentTime / 60), Mathf.FloorToInt(currentTime % 60),
        Mathf.FloorToInt(currentTime % 1 * 1000));
}
```

The timer class is now implemented, however we need to make sure that the timers rotate between players every time a move is done:

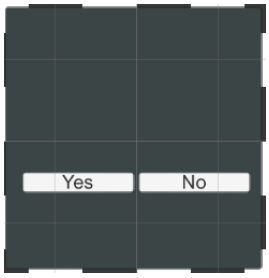
```
public void changeCurrentPlayer()
{
    //used to change the current player
    switch (currentPlayer)
    {
        case ("white"):
            currentPlayer = "black";
            break;
        case ("black"):
            currentPlayer = "white";
            break;
    }
    //rotates timers
    GameObject whiteTimer = GameObject.FindGameObjectWithTag("WhiteTimer");
    whiteTimer.GetComponent<Timer>().cyclePause();
    GameObject blackTimer = GameObject.FindGameObjectWithTag("BlackTimer");
    blackTimer.GetComponent<Timer>().cyclePause();
}
```

Furthermore, at the start of the program, we must start only the white timer:

```
//Starts white timer
GameObject whiteTimer = GameObject.FindGameObjectWithTag("WhiteTimer");
whiteTimer.GetComponent<Timer>().cyclePause();
```

Endgame UI:

At the end of every game, a box should be displayed such that the user can select whether or not they want to play again, who won and by how many points. As drawing and winning are different things, we need separate functions for these too. Firstly, we create a canvas which holds text to edit. We however need to disable this until the end of the game so that the text does not show up before the game ends. We then need two buttons which allow the users to play again.



The yes and no buttons can be mapped to these functions, where we either load the scene again, or go back to the main menu.

```
public void noButton()
{
    //assigned to the game object "NoButton" in the "PlayAgain" menu to load the scene "Start Menu"
    SceneManager.LoadScene("Start Menu");
}

0 references
public void yesButton()
{
    //assigned to the game object "YesButton" in the "PlayAgain" menu to load the scene "Offline Game"
    //which loads the current scene you are on (plays again)
    SceneManager.LoadScene("Offline Game");
}
```

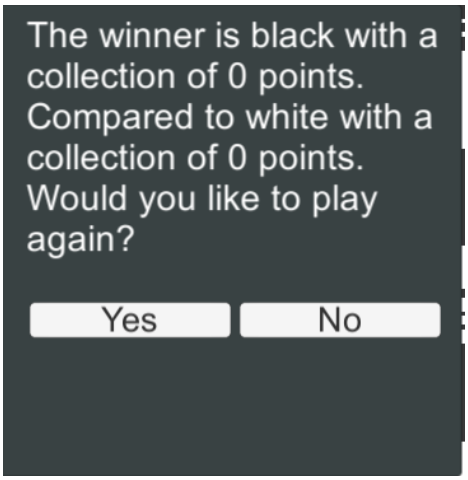
Next, once the game ends, we want to change the text appearing on the screen such that it shows who wins. In the case of winning or losing, we need to show the user who won and by how much.

```
public void gameEnd(string takePiece)
{
    stopTimer();
    //called once the game ends and assigns values for the winner text
    gameOver = true;
    int winnerPoints = 0;
    string Winner = "white";
    int loserPoints = 0;
    string Loser = "black";
    switch (takePiece)
    {
        case ("WKi"): //black wins
        case ("white"):
            Winner = "black";
            Loser = "white";
            winnerPoints = blackPoints;
            loserPoints = whitePoints;
            break;
        case ("BKi"): //white wins
        case ("black"):
            winnerPoints = whitePoints;
            loserPoints = blackPoints;
            break;
    }
```

We first find out the variables needed for who won the game and by how much. Next we need to update the UI:

```
//Finds certain gameobjects and changes the properties of them either disabling them, making them
//visible or changing the text
GameObject.FindGameObjectWithTag("ResignButton").SetActive(false);
GameObject.FindGameObjectWithTag("DrawButton").SetActive(false);
GameObject.FindGameObjectWithTag("PlayAgainUI").transform.localScale = new Vector3(1, 1, 1);
GameObject.FindGameObjectWithTag("WinnerText").GetComponent<Text>().text =
    "The winner is " + Winner + " with a collection of " + winnerPoints.ToString() + " points. Compared to " + Loser +
    " with a collection of " + loserPoints.ToString() + " points. Would you like to play again?";
```

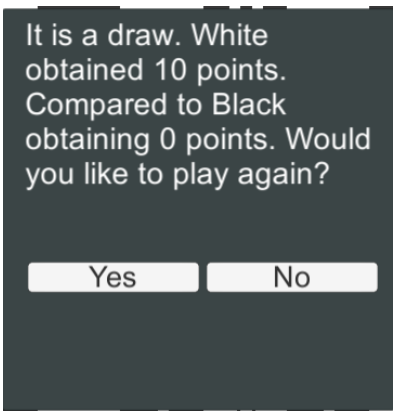
This appears like so:



Next, if a players draw, we do not need to show who won, and only show how much each user had:

```
public void drawEnd()
{
    stopTimer();
    //Used when there is a draw. Finds different game objects and assigns properties similar in
    //gameEnd() function
    GameObject.FindGameObjectWithTag("ResignButton").SetActive(false);
    GameObject.FindGameObjectWithTag("DrawButton").SetActive(false);
    GameObject.FindGameObjectWithTag("PlayAgainUI").transform.localScale = new Vector3(1, 1, 1);
    GameObject.FindGameObjectWithTag("WinnerText").GetComponent<Text>().text =
        "It is a draw. White obtained " + whitePoints.ToString() + " points. Compared to Black obtaining " + blackPoints.ToString() + " points. Would you like to play again?";
}
```

This can be seen like below:



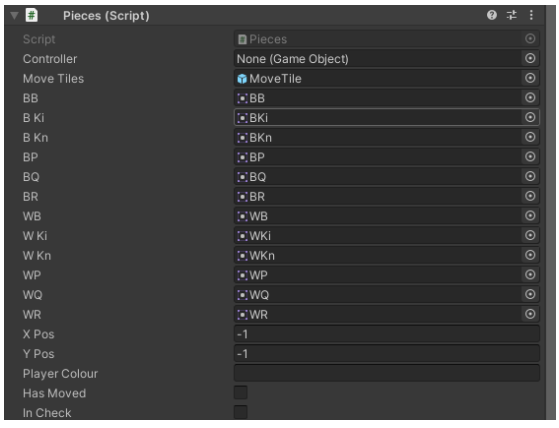
Pieces Class:

The pieces class stores all information about how a piece can move, what it looks like and selecting which move to do. This includes the position of the piece, the sprite image, whether it has moved or not, the colour of the piece, whether the piece is in check (which is used for the king) and the piece's worth. We also need to be able to access the instance of the controller class, as well as access the move tiles which we will use later on.

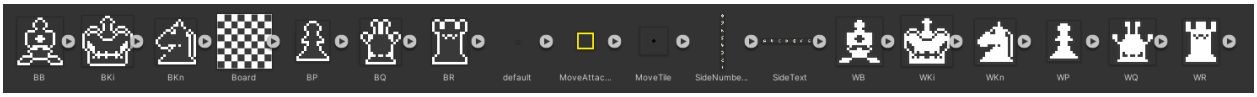
```
public class Pieces : MonoBehaviour
{

    // References
    public GameObject controller; //used to access instance of main class
    //Creating the references for sprites, each assigned to a piece (used in assignPiece())
    public Sprite BB, BKi, BKn, BP, BQ, BR, WB, WKi, WKn, WP, WQ, WR;

    //Creating Board positions (x and y), both assigned -1 as they are not on the board yet. They range between (0-7) created an
    //8x8 board in total. They are assigned later on in the "MainGame" class.
    public int xPos = -1;
    public int yPos = -1;

    //The colour of the piece
    public string playerColour;
    //Private as I do not want code to be able to change the pieceWorth of pieces after they are declared
    private int pieceWorth = 0;
    //If the pawn has moved or not (while it is quite unneccessary for it to be a variable stored in non-pawn pieces, this makes it much easier
    //and does not affect any other pieces anyways
    public bool hasMoved = false;

    public bool inCheck = false;
```

Creating the pieces:

Firstly, we create a prefab which we use to create each instance of the piece, which we do in the Main class. This will allow us to assign variables that the piece can use such as sprites.



Each sprite can be viewed in the sprite folder, and will use one of these assets.



Once we instantiate the piece in the Main class, we reference the chess piece prefab each time to create new pieces:

```
public GameObject createPiece(string name, int xPos, int yPos)
{   //Creates a piece, assigning the name, xpos and ypos to it
    GameObject currentGO = Instantiate(ChessPieces, new Vector3(0, 0, -1), Quaternion.identity);
    Pieces cp = currentGO.GetComponent<Pieces>();
    cp.name = name;
    cp.xPos = xPos;
    cp.yPos = yPos;
    cp.assignPiece();
    return currentGO;
}
```

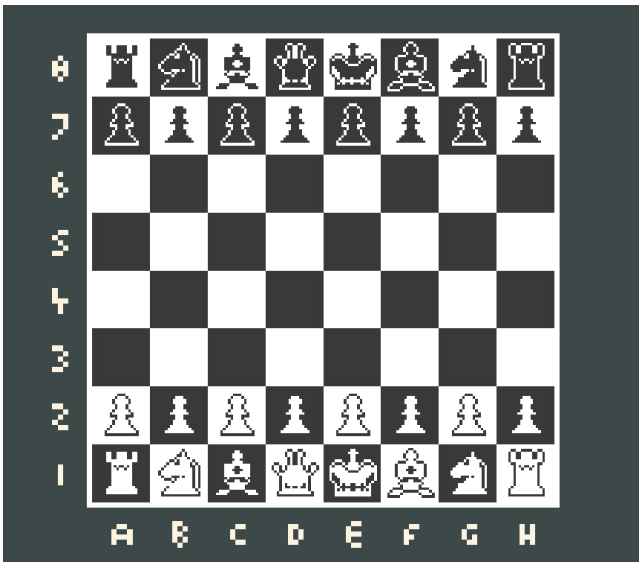
Once the piece is made, we must assign it a sprite value, the piece worth and the player colour:

```
//accesses main instance of class
controller = GameObject.FindGameObjectWithTag("GameController");
switch (name)
{
    //Assigns the sprite value, piece worth and colour of the piece
    case "BB":
        playerColour = "B";
        pieceWorth = 3;
        GetComponent<SpriteRenderer>().sprite = BB;
        break;
    case "BKi":
        playerColour = "B";
        pieceWorth = 90;
        GetComponent<SpriteRenderer>().sprite = BKi;
        break;
```

This repeats for all 12 piece combinations. Lastly, we assign the position to the board on the screen, this was done by calculating the distance between the middle of each block, and finding an algorithm which would place each object in the correct position according to its position on the board. This took some estimation and time but found to be around these values:

```
public void setCoordPos()
//This function transforms the position of the GameObjects, where we must translate the positions which
//are on the board (0-7) to co-ordinates on the screen so they are placed correctly.
{
    transform.position = new Vector3((xPos * 0.66f) - 2.3f, (yPos * 0.66f) - 2.3f);
}
```

Now once we run the program, each piece is assigned to the board and placed, however we cannot move any pieces yet on the board:



Giving pieces movement:

Most pieces can be separated into their types of moves, a straight line move / iterative move, and an input move. An iterative move is where a piece can move in a straight line in different directions, but must stop once either meeting another piece or the end of the board.

```
public void straightLineMove(int changeInX, int changeInY)
{
    int possibleXPos = xPos + changeInX;
    int possibleYPos = yPos + changeInY;
    Main board = controller.GetComponent<Main>();
    //use a while loops as we will create a moveTile every time a space is fre to move until it is not.
    while (board.validBoardPos(possibleXPos, possibleYPos) == true && board.boardPositions[possibleXPos, possibleYPos] == null)
    {
        //creates a move tile
        moveTile(possibleXPos, possibleYPos);
        possibleXPos += changeInX; possibleYPos += changeInY;
    }
    if (board.validBoardPos(possibleXPos, possibleYPos) == true && board.boardPositions[possibleXPos, possibleYPos].GetComponent<Pieces>().playerColour != playerColour)
    {
        //creates an attack tile
        attackTile(possibleXPos, possibleYPos);
    }
}
```

This will create “move tiles”, which are tiles which do not involve attacking the piece, when a position is valid and is currently not occupied. This will repeat and check the next position on the board. Once this ends, either due to the location being off the board or the position is not free, it does one final check to see if the position is occupied by a piece which is not the same colour. This creates an “attack tile”, which are tiles which will cause a piece to be removed from play if taken.

Next, implementing the input move will allow pieces such as the king and knight to move to specific locations. This is done like so, creating a passive tile if location is empty, or an attack tile if the position is an opposite colour.

```
public void inputMove(int newXPos, int newYPos)
{
    //Used to check if a certain location is empty and none other (used for king and knight)
    Main sc = controller.GetComponent<Main>();
    if (sc.validBoardPos(newXPos,newYPos) == true)
    {
        GameObject cp = sc.boardPositions[newXPos, newYPos];
        if (cp == null){
            passiveMove(newXPos, newYPos);
        } else if (cp.GetComponent<Pieces>().playerColour != playerColour)
        {
            attackTile(newXPos, newYPos);
        }
    }
}
```

Lastly, pawns have a specific move type, where they can move forward one spot, but also move diagonal one spot if a piece is right next to it, where it will then attack it.

```
public void pawnMove(int newXPos, int newYPos)
{
    Main board = controller.GetComponent<Main>();
    if (board.validBoardPos(newXPos, newYPos) == true && board.boardPositions[newXPos, newYPos] == null)
    {
        moveTile(newXPos, newYPos);
    } if (board.validBoardPos(newXPos + 1, newYPos) == true && board.boardPositions[newXPos + 1, newYPos] != null && board.boardPositions[newXPos + 1, newYPos].GetComponent<Pieces>().playerColour != playerColour)
    {
        attackTile(newXPos + 1, newYPos);
    } if (board.validBoardPos(newXPos - 1, newYPos) == true && board.boardPositions[newXPos - 1, newYPos] != null && board.boardPositions[newXPos - 1, newYPos].GetComponent<Pieces>().playerColour != playerColour)
    {
        attackTile(newXPos - 1, newYPos);
    }
}
```

Move Tiles:

To allow pieces to move, we must make sure that the user selects which move they want the piece to do, this is done by using what are known as “Move Tiles”, which if you select a piece, will allow you to move the piece to a given location. We have two main variants of the move tile to begin with, this is the “attack tile” and “passive tile”, being that it either will destroy a piece, or move to an empty spot respectively.

```
3 references
public void moveTile(int x,int y)
{
    //Used to instantiate a move tile
    GameObject mt = Instantiate(moveTiles, new Vector3((x * 0.66f) - 2.3f, (y * 0.66f) - 2.3f, -3.0f), Quaternion.identity);
    mt.gameObject.tag = "MovePlate";
    MoveTiles mtscript = mt.GetComponent<MoveTiles>();
    mtscript.setCurrentPiece(gameObject);
    mtscript.setBoardPos(x, y);
}
```

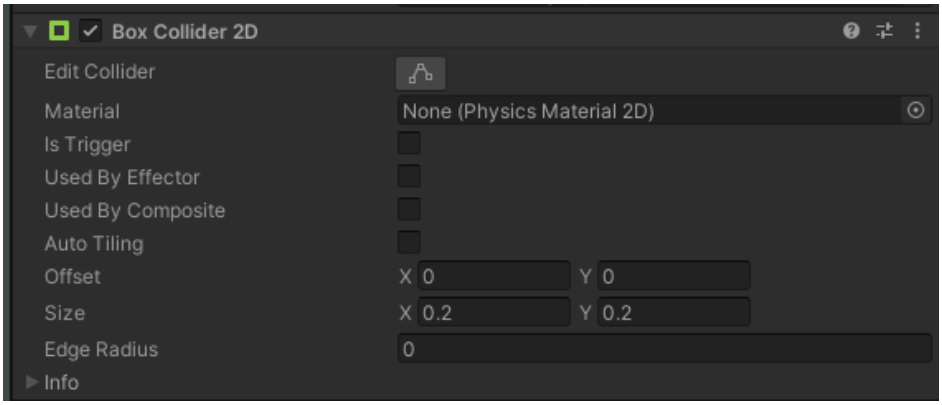
```
3 references
public void attackTile(int x, int y)
{
    //Used to instantiate an attack tile
    GameObject at = Instantiate(moveTiles, new Vector3((x * 0.66f) - 2.3f, (y * 0.66f) - 2.3f, -3.0f), Quaternion.identity);
    at.gameObject.tag = "MovePlate";
    MoveTiles mtscript = at.GetComponent<MoveTiles>();
    mtscript.isAttack = true;
    mtscript.setCurrentPiece(gameObject);
    mtscript.setBoardPos(x, y);
}
```

Both tiles instantiate a new instance of the MoveTile class which we must implement, to allow these pieces to move.

MoveTile Class:

Firstly, we must have a prefab that the piece can use to create a move tile. We assign it the move tile script which we will create fully, but this will allow us to create tiles that the pieces can move to.

We then give the chess pieces prefab and move tiles prefab the “Box Collider” component, this will allow us to gain access to the “OnMouseUp()” function, which when you click on the prefab in game, can call a function.



We first must be able to know what location the piece is at, the piece which must be moved and whether or not it is attacking a piece, which we store in variables.

```
//Used to reference game objects
public GameObject controller; // references the controller
public GameObject currentPiece; //references the piece you are moving

//Used to store where the curent piece wants to move to
public int xBoard;
public int yBoard;

//Used to check to create an attack or move tile
public bool isAttack = false;
```

When a tile is placed on a place, we must allow the piece to be able to move to that location if the player has clicked there. We also need to delete any pieces which will get taken.

We first must check which side the piece is on so we can add the piece's worth to each side, and remove how many pieces are in play, then destroy it. If the piece however is a king, we must end the game.

```
if (isAttack == true)
{

    if (controller.GetComponent<Main>().boardPositions[xBoard,yBoard].name == "WKi" || controller.GetComponent<Main>().boardPositions[xBoard, yBoard].name == "BKi")
    {
        //used to check if the piece is a king and if so it will call the function gameEnd to end the game
        currentPiece.GetComponent<Pieces>().DestroyTiles();
        board.gameEnd(controller.GetComponent<Main>().boardPositions[xBoard, yBoard].name);
    } if (controller.GetComponent<Main>().boardPositions[xBoard,yBoard].GetComponent<Pieces>().playerColour == "W")
    {
        board.blackPoints += board.boardPositions[xBoard, yBoard].GetComponent<Pieces>().getPieceWorth();
        board.whitePiecesLength -= 1;
    } else if(controller.GetComponent<Main>().boardPositions[xBoard, yBoard].GetComponent<Pieces>().playerColour == "B")
    {
        board.whitePoints += board.boardPositions[xBoard, yBoard].GetComponent<Pieces>().getPieceWorth();
        board.blackPiecesLength -= 1;
    }
    GameObject piece = controller.GetComponent<Main>().boardPositions[xBoard, yBoard];
    Destroy(piece);
}
```

We then update the hasMoved variable, meaning that the piece has now moved. This is later on used in allowing more complicated pieces to do specific moves such as castling. Then we move the piece to the new location on the board, delete the remaining tiles so that you cannot move the piece anymore, and then change players.

```
} if (currentPiece.GetComponent<Pieces>().hasMoved == false)
{
    //changes the piece that is moving to true
    currentPiece.GetComponent<Pieces>().hasMoved = true;
}

//changes the location of the current piece to the wanted location
controller.GetComponent<Main>().boardPositions[currentPiece.GetComponent<Pieces>().xPos, currentPiece.GetComponent<Pieces>().yPos] = null;

currentPiece.GetComponent<Pieces>().xPos = xBoard;
currentPiece.GetComponent<Pieces>().yPos = yBoard;
currentPiece.GetComponent<Pieces>().setCoordPos();

controller.GetComponent<Main>().assignBoardPos(currentPiece);

//destroys all tiles
currentPiece.GetComponent<Pieces>().DestroyTiles();
Main player = controller.GetComponent<Main>();

//changes the current player
player.changeCurrentPlayer();

player.checkDraw();
```

```
public void DestroyTiles()
{
    //Finds all the gameobjects withb the tag "MovePlate" and destroys them
    GameObject[] movePlates = GameObject.FindGameObjectsWithTag("MovePlate");
    for (int i=0; i < movePlates.Length; i++)
    {
        Destroy(movePlates[i]);
    }
}
```

Allowing Pieces to move:

Using what we have already written, we can now assign pieces movement. Each simple type of move can be broken down into using the “straightLineMove” function, and “inputMove” function. Some movements have yet to be implemented, but this will allow us to create simple movements for each piece.

We then must be able to know which move type to do for each piece, so by creating functions for each type of piece, we can allow pieces to move as intended by using their name.

```
public void possibleMoves()
{
    //Gets the possible moves for a piece
    Main player = controller.GetComponent<Main>();
    if (player.currentPlayer == "white")
    {
        switch (name)
        {
            case "WB":
                //Diagonals
                bishopMove();
                break;
            case "WKi":
                kingMove();
                break;
            case "WKn":
                knightMove();
                break;
            case "WP":
                pawnMove(xPos, yPos + 1);
                break;
            case "WQ":
                queenMove();
                break;
            case "WR":
                rookMove();
                break;
        }
    }
    else if(player.currentPlayer == "black")
    {
        switch (name)
        {
            case "BB":
                bishopMove();
                break;
            case "BKi":
                kingMove();
                break;
            case "BKn":
                knightMove();
                break;
            case "BP":
                pawnMove(xPos, yPos - 1);
                break;
            case "BQ":
                queenMove();
                break;
            case "BR":
                rookMove();
                break;
        }
    }
}
```

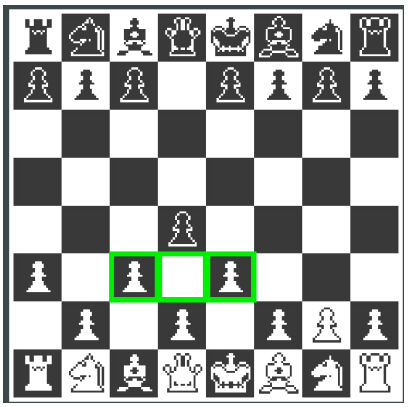
Lastly, we need a way to click on a piece and then be able to get all possible moves for that piece, as well as removing any remaining tiles that are there, so it does not show up on the screen once you choose a different piece.

```
private void OnMouseUp()
{
    //Clicking on the piece prefab will show all moves
    controller = GameObject.FindGameObjectWithTag("GameController");
    Main ai = controller.GetComponent<Main>();

    DestroyTiles();
    possibleMoves();
```

Pawn:

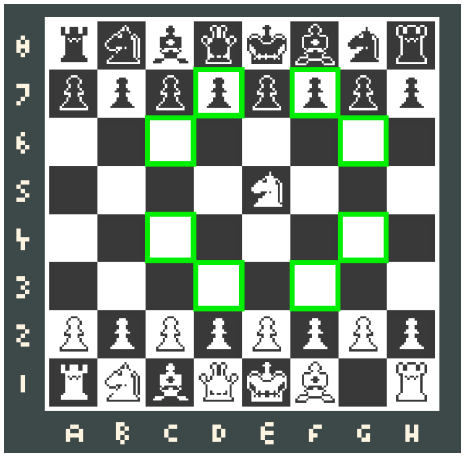
```
//white
pawnMove(xPos, yPos + 1);
```

```
//black
pawnMove(xPos, yPos - 1);
```



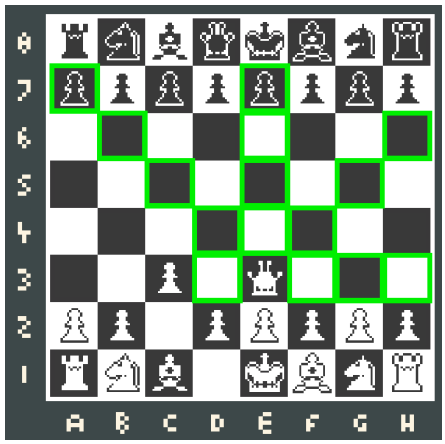
Knight:

```
public void knightMove()
{
    inputMove(xPos + 2, yPos + 1);
    inputMove(xPos + 2, yPos - 1);
    inputMove(xPos - 2, yPos + 1);
    inputMove(xPos - 2, yPos - 1);
    inputMove(xPos + 1, yPos + 2);
    inputMove(xPos - 1, yPos + 2);
    inputMove(xPos + 1, yPos - 2);
    inputMove(xPos - 1, yPos - 2);
}
```



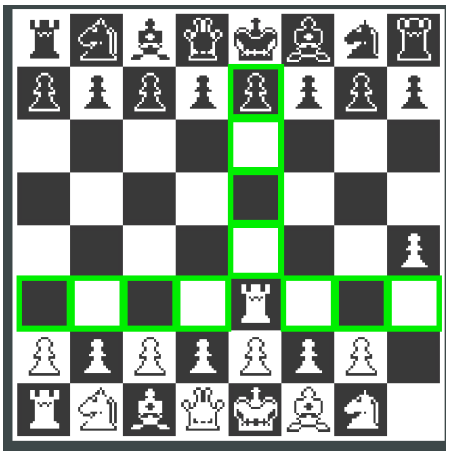
Queen:

```
public void queenMove()
{
    //Straight Lines
    straightLineMove(1, 0);
    straightLineMove(0, 1);
    straightLineMove(-1, 0);
    straightLineMove(0, -1);
    //Diagonal Lines
    straightLineMove(1, 1);
    straightLineMove(-1, -1);
    straightLineMove(-1, 1);
    straightLineMove(1, -1);
}
```



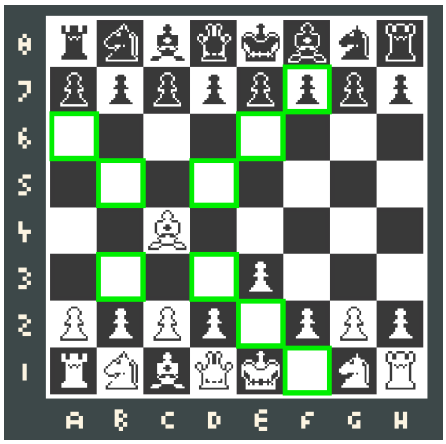
Rook:

```
public void rookMove()
{
    //Straight Lines
    straightLineMove(1, 0);
    straightLineMove(0, 1);
    straightLineMove(-1, 0);
    straightLineMove(0, -1);
}
```



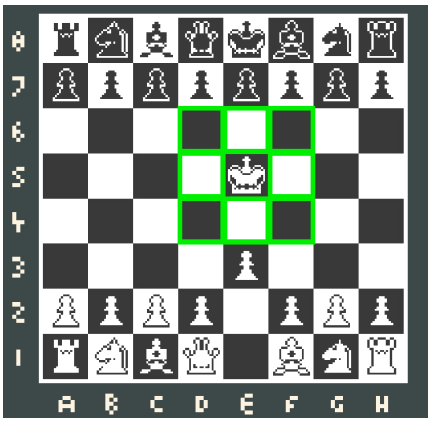
Bishop:

```
public void bishopMove()
{
    //Diagonals
    straightLineMove(1, 1);
    straightLineMove(-1, -1);
    straightLineMove(-1, 1);
    straightLineMove(1, -1);
}
```



King:

```
public void kingMove()
{
    inputMove(xPos + 1, yPos + 1);
    inputMove(xPos, yPos + 1);
    inputMove(xPos + 1, yPos - 1);
    inputMove(xPos + 1, yPos);
    inputMove(xPos - 1, yPos);
    inputMove(xPos - 1, yPos + 1);
    inputMove(xPos, yPos - 1);
    inputMove(xPos - 1, yPos - 1);
```



This completes stage one of the project, where I implemented basic movement of pieces, some basic rules for the game, as well as the main menu options.

Feedback From Client:

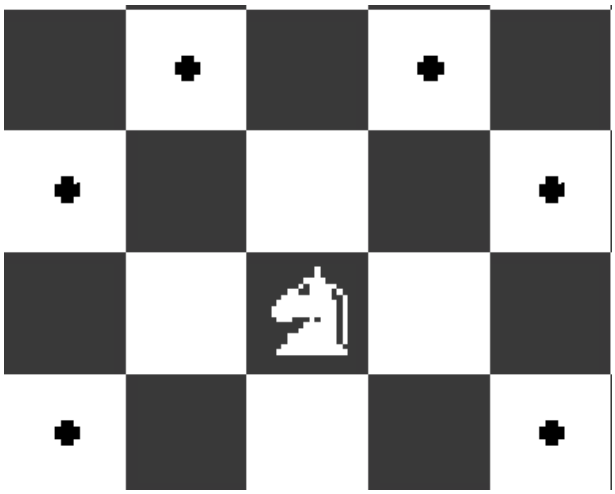
Client Feedback:

Client	Feedback
Oscar Butler	Currently, the main menu is quite basic and barebone. Whilst you are given all the needed options, there is just a monocolor background with no images or enhancing content. The chess game itself is also very basic and is lacking many different moves and rules, which is not realistic of an actual chess game.
Jake Elliot	When selecting between moves, the move tiles are hard to see and quite ugly in my opinion, making it hard to differentiate from a normal move from an attacking move. It would be nice if there was a separate colour for this.

Response:

Currently, most of the options are currently in the beginning phase of development and will change at a later date, especially all the missing moves and rules will be implemented in the next development stage. However, I do agree that the move tiles are quite ugly and can be changed.

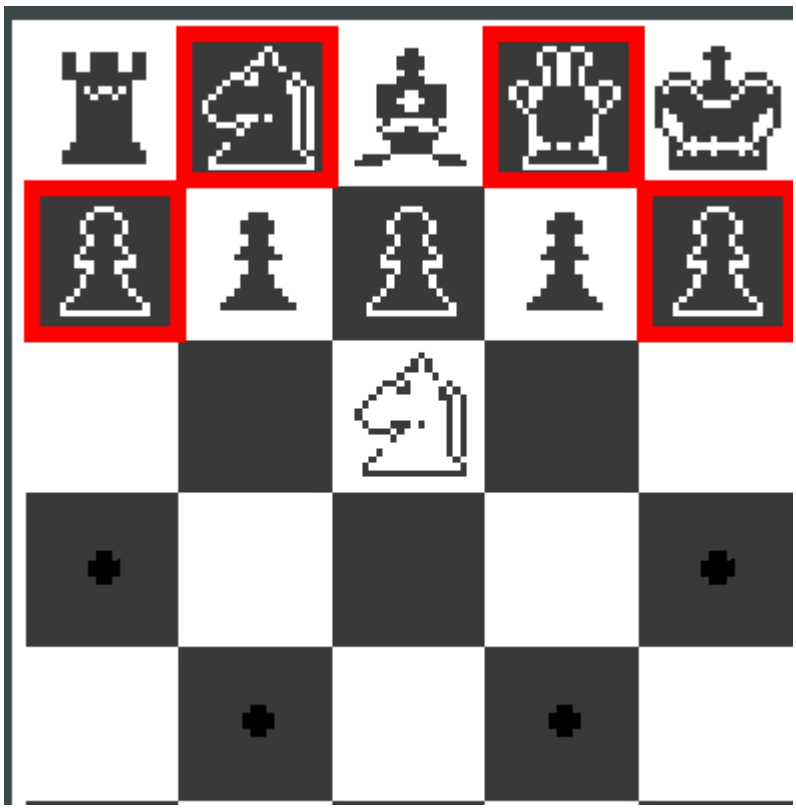
Firstly, I changed the move tile image of the basic move set, such that it is easier to see on the board. They have been changed to use a separate move tile asset:



Next, we check whether or not the move is attacking, which if so, we change the colour of the move tile and what the move tile looks like:

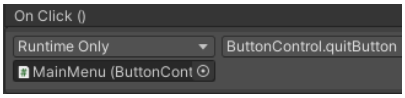
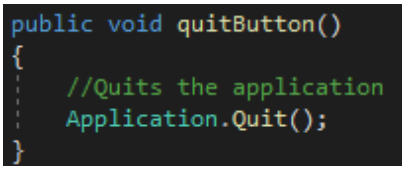
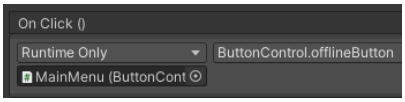
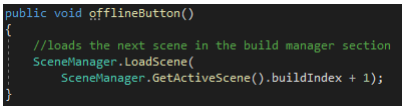
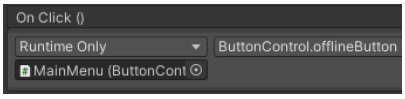
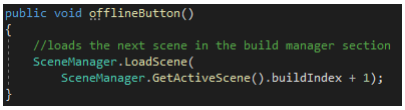
```
public void Start()
{
    if (isAttack)
    {
        gameObject.GetComponent<SpriteRenderer>().color = new Color(1, 0, 0, 1);
        gameObject.GetComponent<SpriteRenderer>().sprite = attackTile;
    }
}
```

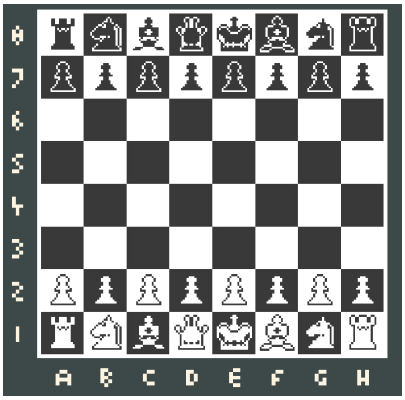
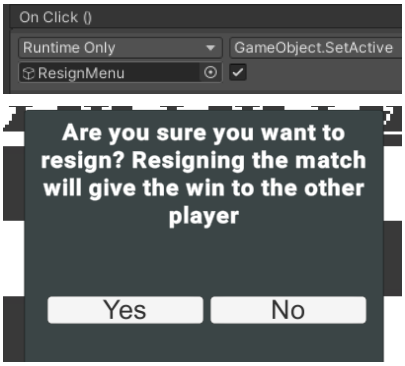
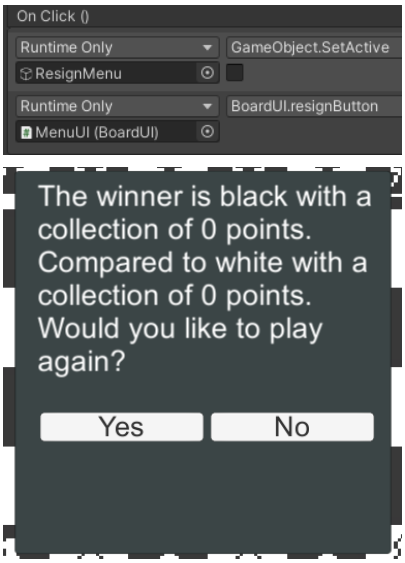
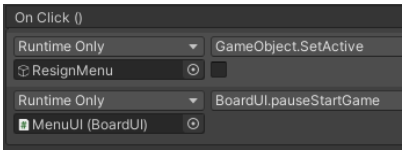
This looks like so:

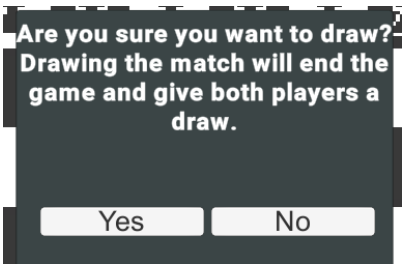
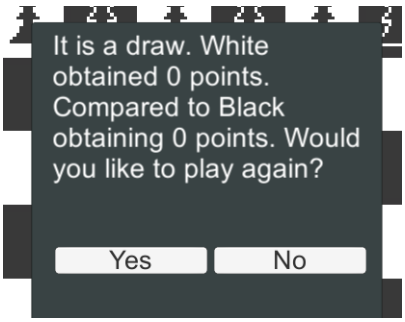
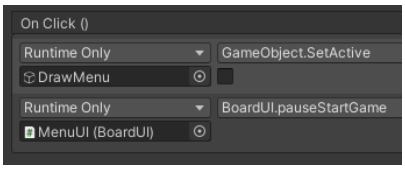
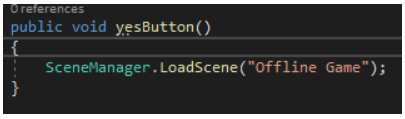
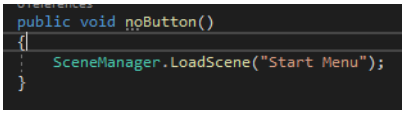
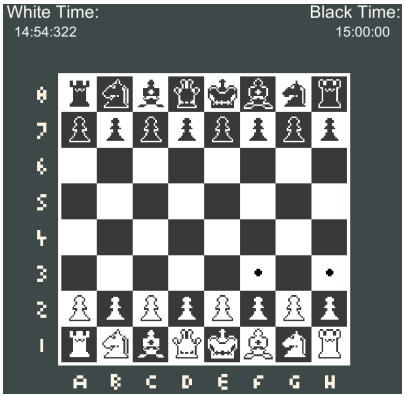


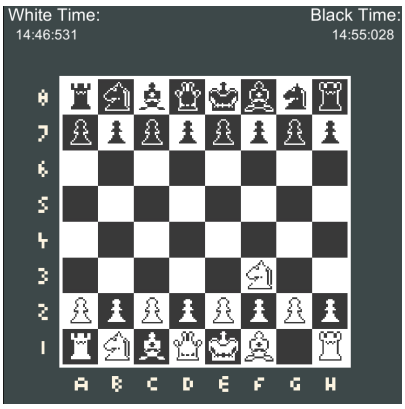
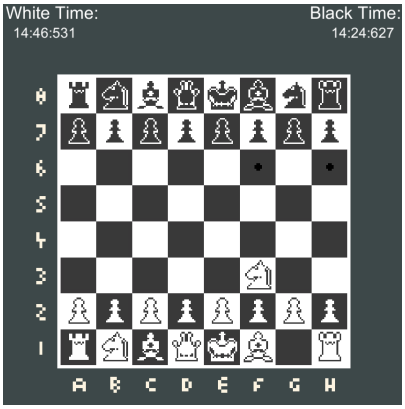
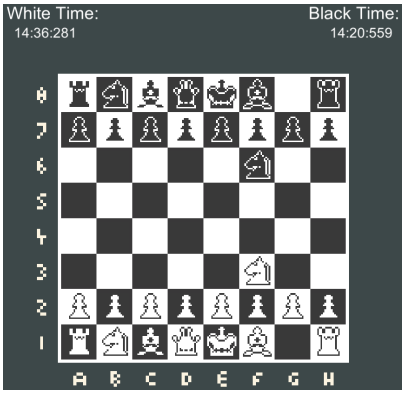
Tests:

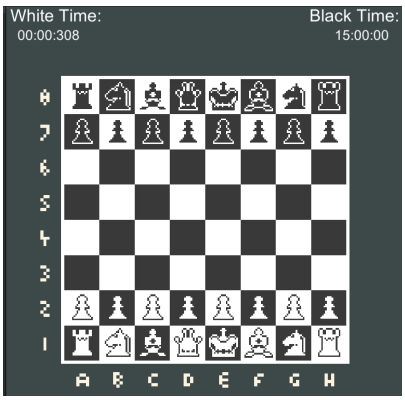
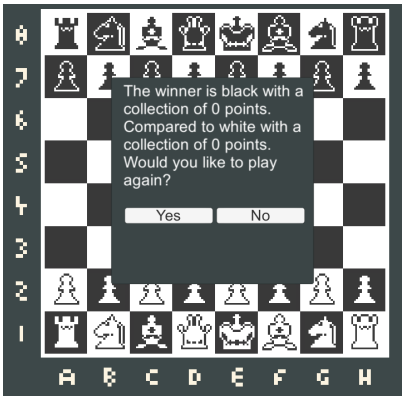
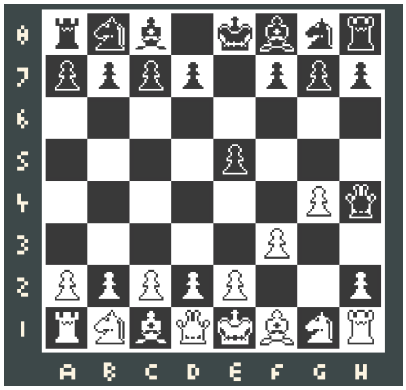
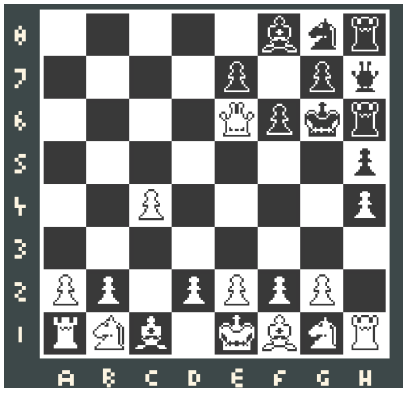
Note: Further Evidence can be found in the “UI video”.

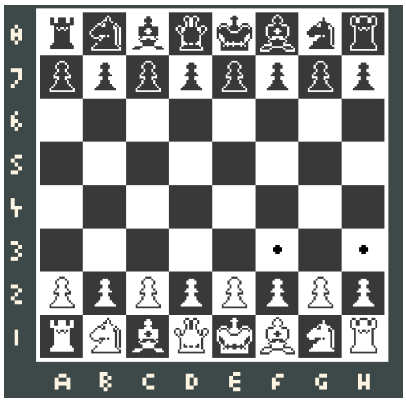
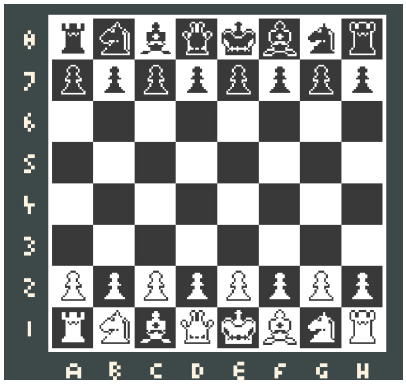
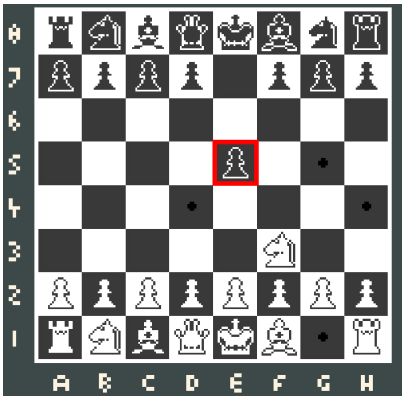
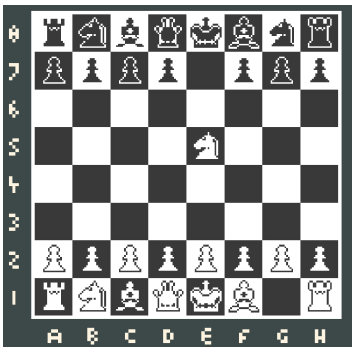
No	Test	Outcome	Evidence	Pass or Fail and actions taken
1ai	Clicking on “Play” in the Main Menu	Opens the “Play Menu”		**Pass**
ii	Clicking on “How To Play” in the Main Menu	Opens the “How To Play Menu”		**Pass**
iii	Clicking on “Exit” in the Main Menu	Closes the application	 	**Pass**
b	Clicking “Back” on any menu screen	Takes you to the main menu		**Pass**
ci	Clicking on “Offline” in the Play Menu	Creates a new game in an offline environment	 	**Pass**
ii	Clicking on “AI” in the Play Menu	Creates a new game where your opponent is an AI	 	**Pass**

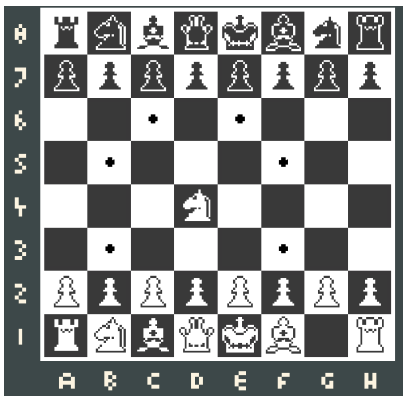
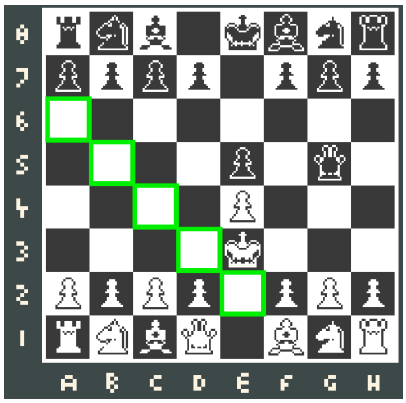
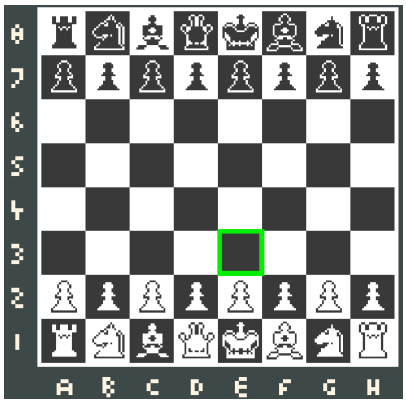
3a	Creating a new game	Creates the board		**Pass**
3bi	Clicking on “Resign” button	Opens the “Resign Menu”		**Pass**
ii	Click “Yes” in the Resign Menu	Ends the game and the player who resigned loses. The “Play Again Menu” will also be opened		**Pass**
iii	Clicking “No” in the Resign Menu	Returns to the game		**Pass**

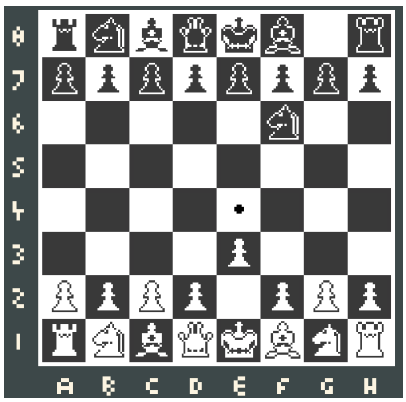
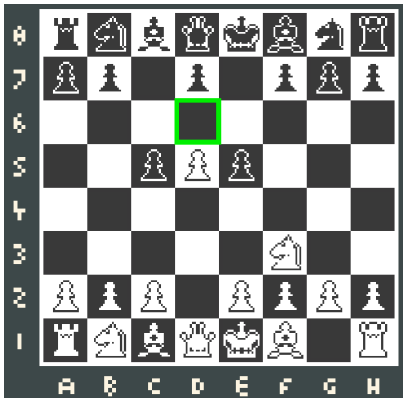
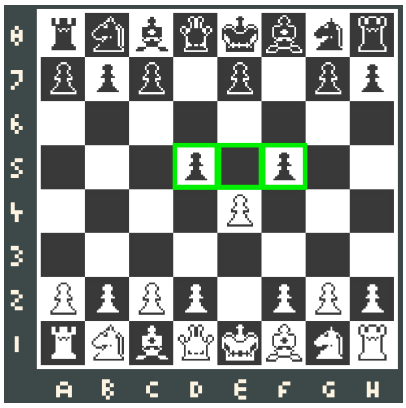
3ci	Clicking on “Draw” button	Opens the “Draw Menu”		**Pass**
ii	Click “Yes” in the Draw Menu	Ends the game and both players draw. The “Play Again Menu” will also be opened		**Pass**
iii	Clicking “No” in the Draw Menu	Returns to the game		**Pass**
3di	Click “Yes” in the “Play Again Menu”	Creates a new chess game		**Pass**
ii	Click “No” in the “Play Again Menu”	Opens the Main Menu		**Pass**
ei	Moving a white piece	Stop the white timer and resume the black timer. Updates the new position of the piece onto the board.	**Before:**  **After:**	**Pass**

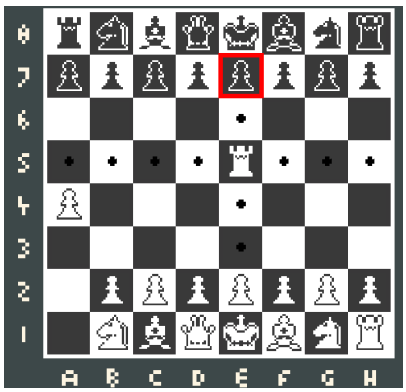
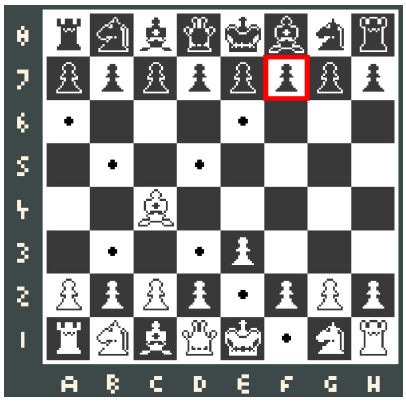
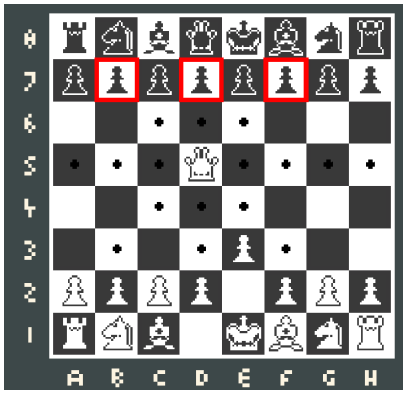
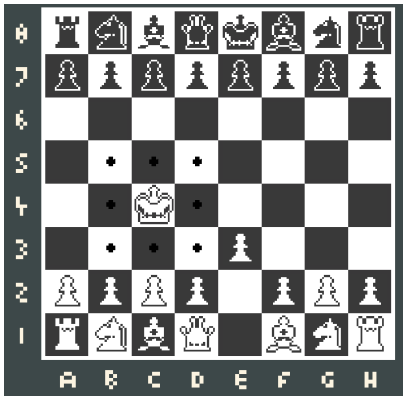
				
ii	Moving a black piece	Stop the black timer and resume the white timer. Updates the new position of the piece onto the board	**Before:**  **After:** 	**Pass**
fi	Once a timer reaches 0	Ends the game and the player whose time ran out will lose. The “Play Again Menu” will also be opened	**Before:**	**Pass**

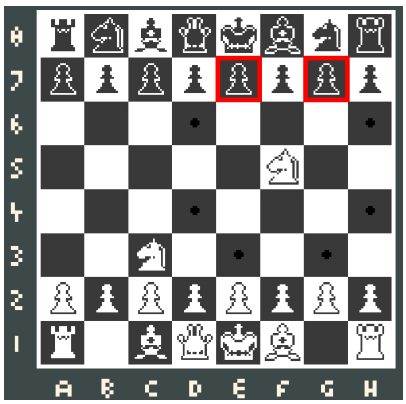
			 **After:** 	
ii	Once checkmate is reached	Nothing		**Fail** The game should end, checkmate will be a rule added at a later development stage.
iii	Once stalemate is reached	Nothing		Fail The game should end in a tie, stalemate will be a rule added at a later development stage.

4ai	Clicking on a piece of the same colour to the turn it is	Shows all available moves on the board the piece can do		**Pass**
ii	Clicking on a piece of the opposite colour to the turn it is	The piece is unable to move		**Pass**
b	Taking a piece with another piece of an occupied spot of different colour	Deletes the piece and adds the amount of points the piece was worth	**Before:**  **After:** 	**Pass**

c	Move the piece onto the location of an occupied spot of same colour	The move does not appear on a spot which is located by the same colour		**Pass**
d	Moving a piece into check of your own king / while you king is in check occurs	The piece is able to move to a position which does not block it		**Fail** The piece shouldn't be able to move, this will be implemented correctly once check is implemented at a later stage.
5ai	Clicking a pawn once it has not moved yet and is not being blocked	The piece moves forward once		**Fail** The Pawn should be able to move two places, this will be a rule added at a later development stage.

ii	Clicking a pawn once it has already moved and is not being block	Only being able to move forward one space		**Pass**
iii	Moving a pawn two spaces forward next to a pawn of opposite colour	The piece can only move forward		**Fail** Enpassant will be a rule added at a later development stage.
6a	Clicking a pawn	Shows correct move pattern		**Pass**

b	Clicking a castle / rook	Shows correct move pattern		**Pass**
c	Clicking a bishop	Shows correct move pattern		**Pass**
d	Clicking a queen	Shows correct move pattern		**Pass**
e	Clicking a king	Shows correct move pattern		**Pass**

f	Clicking a knight	Shows correct move pattern		**Pass**

Stage Two Of Development:

In this stage, I will begin to implement more complicated moves, rules, more UI functionality, as well as some minor improvements to other functions.

Chess Rules:

Draw by Lack of Material:

A rule of chess is known as “draw by lack of material” where both sides have a lack of material to end the game, so it is a draw. A list of combinations of materials can be found in the previous design section. We first must check whether or not there is a piece at a certain location on the board corresponding to a type of piece, and if that is the case, we then draw the game.

```
public void checkDraw()
{
    //used to check if there is a draw due to insufficient material
    if (whitePiecesLength == 1 && blackPiecesLength == 1)//king vs king
    {
        drawEnd();
    } //everything below includes king
    else if ((whitePiecesLength == 2 && blackPiecesLength == 1) || (whitePiecesLength == 1 && blackPiecesLength == 2))
        //if one player has one piece and the other has two (this is to optimise the amount of checks)
    {
        if (((whitePieces[2] != null || whitePieces[5] != null) && blackPiecesLength == 1) || ((blackPieces[2] != null || blackPieces[5] != null) && whitePiecesLength == 1))
            //knight vs king
        {
            drawEnd();
        } else if (((whitePieces[1] != null || whitePieces[6] != null) && blackPiecesLength == 1) || ((blackPieces[1] != null || blackPieces[6] != null) && whitePiecesLength == 1))
            //bishop vs king
        {
            drawEnd();
        }
    } else if (whitePiecesLength == 2 && blackPiecesLength == 2)
    { //boths players have two pieces
        if ((whitePieces[0] != null || whitePieces[7] != null) && (blackPieces[0] != null || blackPieces[7] != null))
            // rook vs rook
        {
            drawEnd();
        }
        else if ((whitePieces[2] != null && blackPieces[5] != null) || (whitePieces[5] != null && blackPieces[2] != null))
            // bishop vs bishop (both being on the same square colour)
        {
            drawEnd();
        }
        else if ((whitePieces[0] != null || whitePieces[7] != null) && (blackPieces[2] != null || blackPieces[1] != null || blackPieces[5] != null || blackPieces[6] != null))
            //rook vs bishop
        {
            drawEnd();
        }
        else if ((blackPieces[0] != null || blackPieces[7] != null) && (whitePieces[2] != null || whitePieces[1] != null || whitePieces[5] != null || whitePieces[6] != null))
            // rook vs knight
        {
            drawEnd();
        }
    } else if ((whitePiecesLength == 3 && blackPiecesLength ==1) || (blackPiecesLength == 3 && whitePiecesLength == 1))
    { //three pieces vs one
        if ((whitePieces[1] != null && whitePieces != null && blackPiecesLength == 1) || (blackPieces[1] != null && blackPieces[6] != null && whitePiecesLength == 1))
            //two knights vs king
        {
            drawEnd();
        }

    }
}
```

Threefold Repetition:

Threefold repetition is the rule where if the same position is repeated 3 times in a game, the game ends in a draw. To solve this problem, we must store the positions of all pieces in the last ~50 moves, as if there are 50 moves where no pieces are taken, the game will end, so we do not need to store more than 50 moves. Firstly, to efficiently store the boards each time, we convert it into easier to work with data, in this case, an array which stores arrays of each board, being 64 spots:

```
//used for threefold repition
public int[,] threeFoldHistory = new int[101,64];
public int historyLength = 0;
```

After each move, the position will be updated to the array. To distinguish between pieces, we use their position in the array, as well as their colour, making sure each piece has its own value.

```
public bool convertBoardToInt()
{ //converts the current board positions to an integer format
    int[] intBoard = new int[64];
    for(int i =0; i < whitePieces.Length; i++)
    { //to distinguish from each piece, the array position is used, and if it is black, it is +17 to distinguish sides
        if(whitePieces[i] != null)
        {
            intBoard[whitePieces[i].GetComponent<Pieces>().xPos + whitePieces[i].GetComponent<Pieces>().yPos *8] = i + 1;
        } if (blackPieces[i] != null)
        {
            intBoard[blackPieces[i].GetComponent<Pieces>().xPos + blackPieces[i].GetComponent<Pieces>().yPos * 8] = i + 17;
        }
    }
    return checkRepition(intBoard);
}
```

Lastly, we then check if it has been repeated multiple times by checking each board with separate positions, and if all squares are equal, we have a counter which increments, which if it is equal to three, will end the game. If it doesn't, we append the current board position to the end of the array. This will repeat for every single move that the user does.

```
1 reference
public bool checkRepition(int[] boardPositions)
{//used for threefold repition
    int checkSum = 0;
    for (int currentCheck = 0; currentCheck < historyLength; currentCheck++)
    { //iterates through all stored moves
        int checkCurrent = 0;
        for(int valueCheck = 0; valueCheck < boardPositions.Length; valueCheck++)
        {
            if (threeFoldHistory[currentCheck,valueCheck] == boardPositions[valueCheck])
            { //if a square is equal to the same square on a seperate board
                checkCurrent++;
            }
        }
        if (checkCurrent == 64)
        { //if all 64 spots are the same
            checkSum++;
            if (checkSum == 2)
            { //if the positon is repeated three times, end the game
                return true;
            }
        }
    }
    for (int i = 0; i < 64; i++)
    {//adds the move to history
        threeFoldHistory[historyLength, i] = boardPositions[i];
    }
    historyLength++;
    return false;
}
```

Once we then choose which move to do in the MoveTile class, we can check for threefold repetition.

```
//checks for threefold repition
if (board.convertBoardToInt())
{
    board.drawEnd();
}
```

50 move rule:

The 50 move rule is where if a pawn does not move, castle does not occur or a piece is not taken for 50 moves. To implement this, we just need a counter which keeps track of when each time a piece moves, and if it reaches 50 we draw the game. Else we reset the counter back to zero. This can be stored in just a variable and can be changed throughout the program.

```
//used to count for 50 move rule
public float moves = 0.0f;
```

Once we then choose the move in the MoveTile class, we then update it depending on the movetype, where we either update by +0.5, or reset it back to zero. In this case, if it is an attacking move, or if you are moving a pawn, since these positions can never be repeated again. We can also reset the history length since you cannot have any of the previous moves played again.

```
//50 moves rule
if (currentPiece.GetComponent<Pieces>().name == "WP" || currentPiece.GetComponent<Pieces>().name == "BP" || attackEnPassant || isAttack)
{
    board.moves = 0.0f;
    board.historyLength = 0;
}
else
{
    board.moves += 0.5f;
    if (board.moves == 50f)
    {
        board.drawEnd();
    }
}
```

Complicated Moves:

There are some moves I have yet to implement, including moving a pawn two places, if it has yet to move, en passant and castling.

Moving a Pawn two places:

Pawns can move two places, assuming that it has yet to move, or that there are no pieces between the pawn and the location it is going to move to. This means that there will only be a need to create a passive move tile since we are not attacking any pieces.

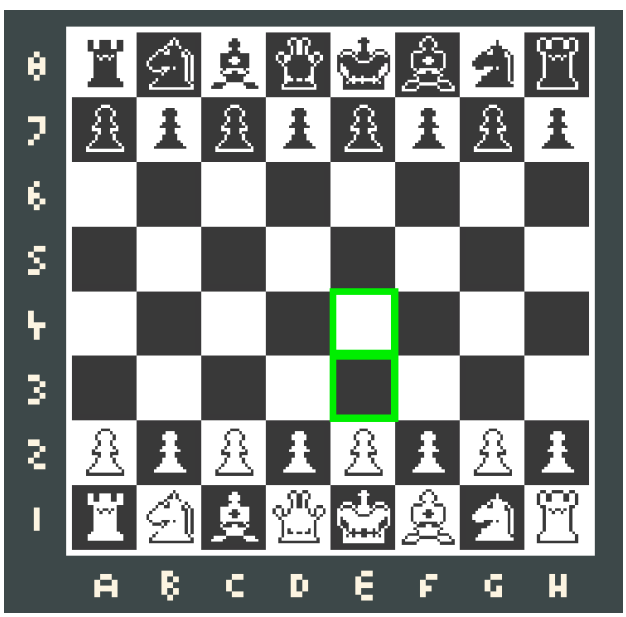
```
public void pawnNotMove(int changeInX, int changeInY)
{
    //Used to allow a pawn to move two spaces if the piece has not moved yet
    Main board = controller.GetComponent<Main>();
    if(changeInY == 2)
    {//moving a white pawn
        if (board.validBoardPos(xPos + changeInX, yPos + changeInY) == true && board.boardPositions[xPos + changeInX, yPos + changeInY] == null && board.boardPositions[xPos + changeInX, yPos + changeInY - 1] == null)
        {
            passiveMove(xPos + changeInX, yPos + changeInY);
        }
    }
    else
    {//moving a black pawn
        if (board.validBoardPos(xPos + changeInX, yPos + changeInY) == true && board.boardPositions[xPos + changeInX, yPos + changeInY] == null && board.boardPositions[xPos + changeInX, yPos + changeInY + 1] == null)
        {
            passiveMove(xPos + changeInX, yPos + changeInY);
        }
    }
}
```

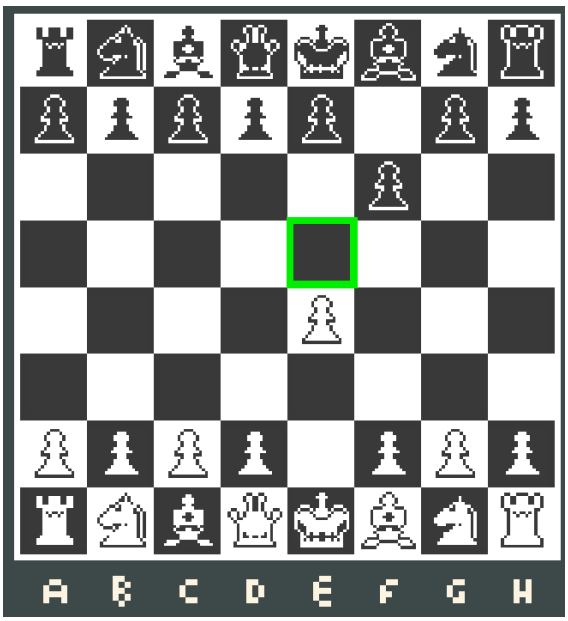
Lastly, we then must check if the piece has moved, else we cannot let the piece to do the move, However, we still must allow the piece to move two places forward.

```
case "WP":
    if (hasMoved == true)
    {
        //white
        pawnMove(xPos, yPos + 1);
    }
    else
    {
        pawnMove(xPos, yPos + 1);
        pawnNotMove(0,2);
    }
```

```
case "BP":
    if (hasMoved == true)
    {
        //black
        pawnMove(xPos, yPos - 1);
    }
    else
    {
        pawnMove(xPos, yPos - 1);
        pawnNotMove(0,-2);
    }
```

The pawn now can move two places forward:





Castling:

Castling is a move where the king and castle both move together, assuming that neither have moved yet and there are no pieces which remain next to each other.

```
public void castle()
{
    //used to check if the king can castle
    Main board = controller.GetComponent<Main>();
    if (board.boardPositions[xPos + 3, yPos].GetComponent<Pieces>().name[1] == 'R' && board.boardPositions[xPos + 2, yPos] == null && board.boardPositions[xPos + 1, yPos] == null
        && board.boardPositions[xPos + 3,yPos].GetComponent<Pieces>().playerColour == playerColour)
    {
        //used to see if the king can castle king side
        moveCastleTile(xPos + 2, yPos, board.boardPositions[xPos + 3, yPos], -1);
    } if(board.boardPositions[xPos - 4, yPos].GetComponent<Pieces>().name[1] == 'R' && board.boardPositions[xPos - 3, yPos] == null && board.boardPositions[xPos - 2, yPos] == null
        && board.boardPositions[xPos - 1, yPos] == null && board.boardPositions[xPos + 3, yPos].GetComponent<Pieces>().playerColour == playerColour)
    {
        //used to see if the king can castle queen side
        moveCastleTile(xPos - 2, yPos, board.boardPositions[xPos - 4, yPos], 2);
    }
}
```

We must then create a new type of move tile, this is because we will be moving two pieces at once, we can call this moveCastleTile. Also meaning we must add some variables to our MoveTile class:

```
//Used in castling
public bool isCastle = false;
public int castleXPos;
public int castleYPos;
public int changeInX;
public GameObject castlePiece; //references the castle if you are castling
```

We can update these variables in the function called below when instantiating the castle tile.

```
public void moveCastleTile(int x, int y, GameObject piece, int changeInX)
{
    //used to instaniate a move tile when the king castles
    GameObject mt = Instantiate(moveTiles, new Vector3((x * 0.66f) - 2.3f, (y * 0.66f) - 2.3f, -3.0f), Quaternion.identity);
    mt.gameObject.tag = "MovePlate";
    MoveTiles mtscript = mt.GetComponent<MoveTiles>();
    mtscript.setCurrentPiece(gameObject);
    mtscript.setBoardPos(x, y);
    mtscript.isCastle = true;
    mtscript.castlePiece = piece;
    mtscript.castleXPos = x;
    mtscript.castleYPos = y;
    mtscript.changeInX = changeInX;
}
```

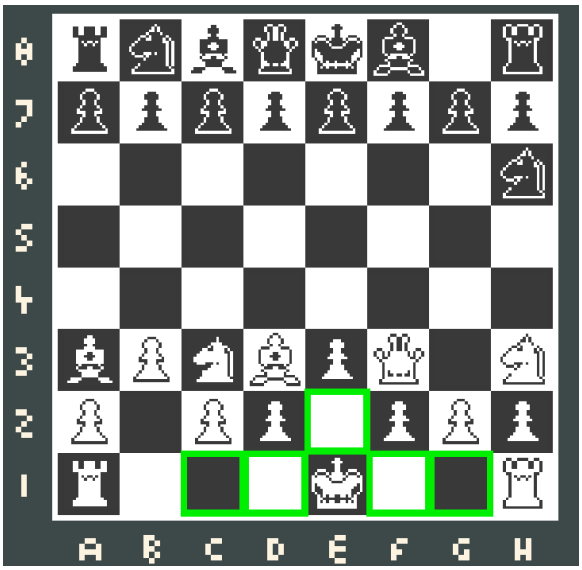
Then in the movetile class, we can now update the piece if the user clicks on the move tile as done below:

```
if (isCastle == true)
{
    controller.GetComponent<Main>().boardPositions[castleXPos - changeInX, castleYPos] = null;
    if(changeInX == 2)
    {
        changeInX = 1;
    }
    castlePiece.GetComponent<Pieces>().xPos = xBoard + changeInX;
    castlePiece.GetComponent<Pieces>().yPos = yBoard;
    castlePiece.GetComponent<Pieces>().setCoordPos();

    controller.GetComponent<Main>().assignBoardPos(castlePiece);
```

Lastly, we allow the piece to move by putting it into the available moves for which a king can do:

```
Main board = controller.GetComponent<Main>();
if (playerColour == "B" && hasMoved == false && (board.boardPositions[7, 7].GetComponent<Pieces>().hasMoved == false
    || board.boardPositions[0, 7].GetComponent<Pieces>().hasMoved == false))
{
    castle();
}
else if (playerColour == "W" && hasMoved == false && (board.boardPositions[0,0].GetComponent<Pieces>().hasMoved == false
    || board.boardPositions[7, 0].GetComponent<Pieces>().hasMoved == false))
{
    castle();
}
```

Now once under the correct circumstances, the piece can now move:



En Passant:

En passant is a move where if a pawn moves two places from its starting position, but an opposing pawn is either left or right to the pawn at the location it moves to, the piece will be able to be taken for only that move, as if it had only moved one space forward. As the move requires you to take a piece where the position is currently empty, and that you can only do it once per turn, this means we need to create many more functions and variables which will allow us to do so.

Firstly, we must know when en passant is possible, due to a pawn moving two places forward. This will be needed in the MoveTile class as we must be able to know if the move will allow a piece to en passant.

```
//Used in en Passant
public bool enPassant = false;
public bool attackEnPassant = false;
```

To determine whether or not a pawn can en passant, we must find the change in the x position value, which must be 2, and that the piece is a pawn, so when creating the MoveTile in the piece class, we must only allow it to be done with a pawn move of this type.

When a pawn moves two places forward, we can check whether or not a pawn is left or right to the piece, if it moved there, So by appending to our already made moveplate function for a pawn, this will allow us to know that a piece can check for an en passant:

```
int val = Math.Abs(yPos - y);
if (val == 2)
{
    mtscript.enPassant = true;
}
```

Once the user then selects this move to move a pawn two places forward, we then in the MoveTile class must check if any pawns either to the left or right of it at that position are of different colour, in which case it can then begin to set up en passant.

```
//used to allow enPassant
if (enPassant)
{
    enPassantMove();
}
```

```
public void enPassantMove()
{   //sets enpassant variables if a pawn does the certain requirements
    Main board = controller.GetComponent<Main>();
    if (board.validBoardPos(xBoard + 1, yBoard))
    {
        if (board.boardPositions[xBoard + 1, yBoard] != null)
        {
            if (board.boardPositions[xBoard + 1, yBoard].GetComponent<Pieces>().name[1] == 'P' &&
                board.boardPositions[xBoard + 1, yBoard].GetComponent<Pieces>().playerColour
                != currentPiece.GetComponent<Pieces>().playerColour)
                //if the piece is a pawn of different colour
            {
                board.boardPositions[xBoard + 1, yBoard].GetComponent<Pieces>().enPassantAttack = true;
                if (currentPiece.GetComponent<Pieces>().playerColour == "W")
                {//white pawn
                    //sets the position of where the piece can enpassant
                    board.boardPositions[xBoard + 1, yBoard].GetComponent<Pieces>().enPassantXPos = xBoard;
                    board.boardPositions[xBoard + 1, yBoard].GetComponent<Pieces>().enPassantYPos = yBoard - 1;
                }
                else
                { //black pawn
                    board.boardPositions[xBoard + 1, yBoard].GetComponent<Pieces>().enPassantXPos = xBoard;
                    board.boardPositions[xBoard + 1, yBoard].GetComponent<Pieces>().enPassantYPos = yBoard + 1;
                }
            }
        }
```

```
    }//repeats for the other side of the pawn
    if (board.validBoardPos(xBoard - 1, yBoard))
    {
        if (board.boardPositions[xBoard - 1, yBoard] != null)
        {
            if (board.boardPositions[xBoard - 1, yBoard].GetComponent<Pieces>().name[1] == 'P' &&
                board.boardPositions[xBoard - 1, yBoard].GetComponent<Pieces>().playerColour
                != currentPiece.GetComponent<Pieces>().playerColour)
            {
                board.boardPositions[xBoard - 1, yBoard].GetComponent<Pieces>().enPassantAttack = true;
                if (currentPiece.GetComponent<Pieces>().playerColour == "W")
                {
                    board.boardPositions[xBoard - 1, yBoard].GetComponent<Pieces>().enPassantXPos = xBoard;
                    board.boardPositions[xBoard - 1, yBoard].GetComponent<Pieces>().enPassantYPos = yBoard - 1;
                }
                else
                {
                    board.boardPositions[xBoard - 1, yBoard].GetComponent<Pieces>().enPassantXPos = xBoard;
                    board.boardPositions[xBoard - 1, yBoard].GetComponent<Pieces>().enPassantYPos = yBoard + 1;
                }
            }
        }
    }
```

We then need to create some more variables for our pieces class so we can store the positions where a piece can en passant and whether or not it can do that attack

```
//Used to allow pawns to en passant
public bool enPassantAttack = false;
public int enPassantXPos = -1;
public int enPassantYPos = -1;
```

Assuming that situation is built to the point where the piece can en passant, we can create a new type of movetile, which will allow the piece to move to the new position.

```
if (enPassantAttack)
{ //if the piece can enpassant
    if (board.boardPositions[enPassantXPos,enPassantYPos] == null)
    {//create move tile
        enPassantMovePawn(enPassantXPos, enPassantYPos);
    }
}
```

```
public void enPassantMovePawn(int x, int y)
{
    //Used to instantiate an enpassant attack tile
    GameObject at = Instantiate(moveTiles, new Vector3((x * 0.66f) - 2.3f, (y * 0.66f) - 2.3f, -3.0f), Quaternion.identity);
    at.gameObject.tag = "MovePlate";
    MoveTiles mtscript = at.GetComponent<MoveTiles>();
    mtscript.attackEnPassant = true;
    mtscript.setCurrentPiece(gameObject);
    mtscript.setBoardPos(x, y);
}
```

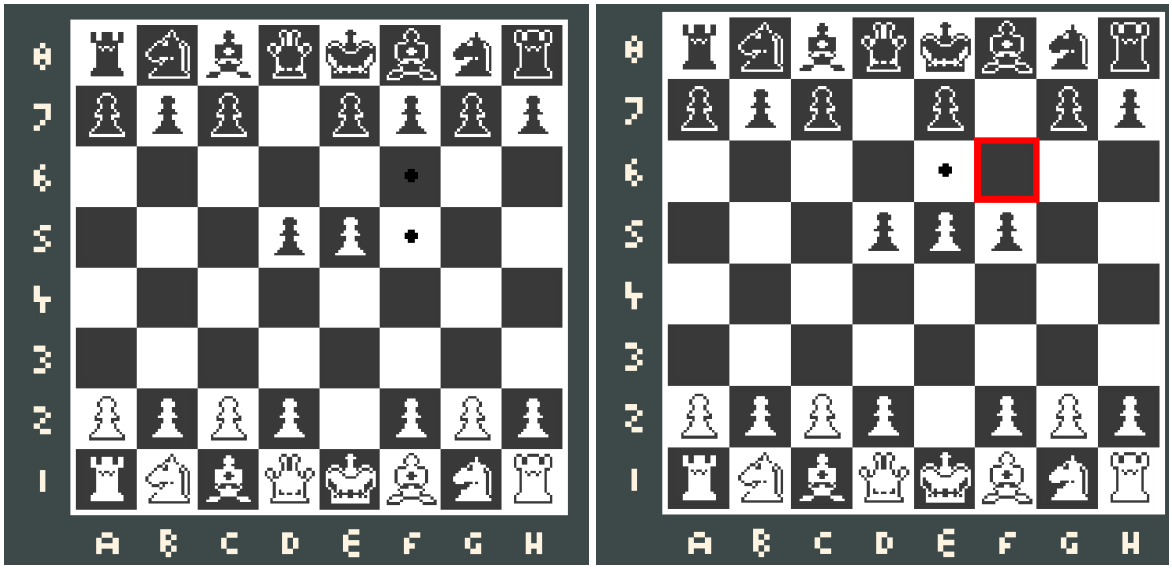
With this, we create a new type of attack, where it will remove the pawn that is below or above (depending on the colour of the piece) the pawn that has just moved.

```
else if (attackEnPassant)
{
    if (currentPiece.GetComponent<Pieces>().playerColour == "W")
    {
        //destroy piece below
        GameObject piece = board.boardPositions[xBoard, yBoard - 1];
        Destroy(piece);
    }
    else
    {
        //destroy piece above
        GameObject piece = board.boardPositions[xBoard, yBoard + 1];
        Destroy(piece);
    }
}
```

Lastly, to ensure that the pawn cannot en passant after a turn has passed, we must reset this variable for each pawn on the same side, so that it cannot do it another turn. So we must iterate through all the pawns, resetting the variable back to false, so that it cannot en passant until the requirements are met again:

```
//used to clear all enpassant pieces each turn if not used
if (board.currentPlayer == "white")
{
    for (int currentPiece = 8; currentPiece < board.whitePieces.Length; currentPiece++)
    {
        if (board.whitePieces[currentPiece] != null)
        {
            board.whitePieces[currentPiece].GetComponent<Pieces>().enPassantAttack = false;
        }
    }
}
else if (board.currentPlayer == "black")
{
    for (int currentPiece = 8; currentPiece < board.whitePieces.Length; currentPiece++)
    {
        if (board.blackPieces[currentPiece] != null)
        {
            board.blackPieces[currentPiece].GetComponent<Pieces>().enPassantAttack = false;
        }
    }
}
```

The final coded solution will allow the piece to move as so, and will reset assuming the selected move is not to en passant:



Promotion:

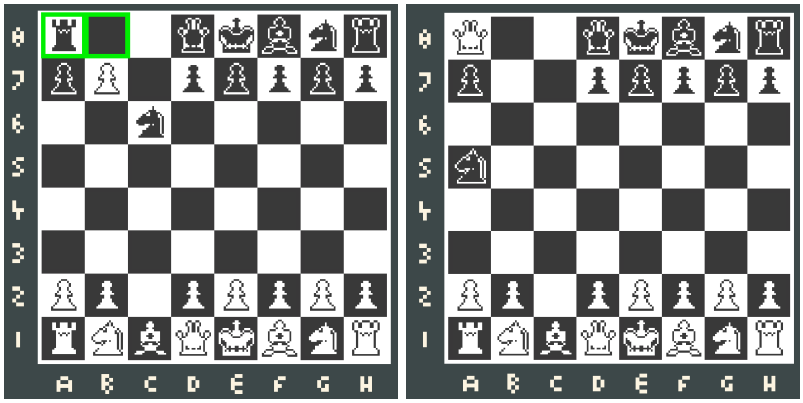
Promotion occurs when a pawn reaches the final y axis rank on the board (either at a position of 1 for black, or 8 for white), where the pawn can be changed out to any piece on the board, besides a pawn. For now, we will only allow the user to promote to a queen.

```
2 references
public void checkPromote()
{
    //used to check if a pawn can be promoted
    Main board = controller.GetComponent<Main>();
    if (yPos == 7)
    {
        Destroy(gameObject);
        board.setPiece(xPos, yPos, "W");
    }
    else if (yPos == 0)
    {
        Destroy(gameObject);
        board.setPiece(xPos, yPos, "B");
    }
}
```

Once we reach the parameters to promote a pawn, we destroy the piece, and assign a new queen to the game at the same position of the destroyed pawn.

```
public void setPiece(int x, int y, string colour)
{
    //used in promoting the pawn once it reaches the other side,
    //it creates a queen at the given position
    if (colour == "B")
    {
        blackPiecesLength ++;
        assignBoardPos(createPiece("BQ", x, y));
    }
    else
    {
        whitePiecesLength++;
        assignBoardPos(createPiece("WQ", x, y));
    }
}
```

This is demonstrated below:



Additional Improvements:

Move History:

To allow users to better analyse previous games, we can use the algebraic chess notation to notate how each user does a move. A detailed description can be found in the design phase, however, basically, it will be in the format of PieceName + Position + ('X' if taking a piece) + NewPosition. In some other circumstances, it may change, such as promoting, castling or at the end of the game.

Firstly, in the Main class, we need a way to store each move, we do so using a list as there can be an undetermined amount of moves in a game.

```
//used to store moves
public List<string> moveHistory = new List<string>();
```

Then the MoveTile class will store the move which the tile would create.

```
//used for moves to be stored onto a file
public string moveType = "";
```

Next, we need to translate the move to algebraic notation using the old position, name and new position. We then combine each variable to create the move.

```
public string translateNotation(int oldXAxis, int oldYAxis, int newXAxis, int newYAxis, bool isAttacking,string pieceName)
{
    //used to translate each move into algebraic notation to later be stored in moves history
    char[] vals = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' };
    string pieceNameDone = "";
    string attacking = "";
    if (isAttacking)
    {
        attacking = "x";
    }
    //Convert name to notation
    switch (pieceName)
    {
        case "BB":
        case "WB":
            pieceNameDone = "B";
            break;
        case "BKi":
        case "WKi":
            pieceNameDone = "K";
            break;
        case "BKn":
        case "WKn":
            pieceNameDone = "N";
            break;
        case "BP":
        case "WP":
            pieceNameDone = "P";
            break;
        case "BQ":
        case "WQ":
            pieceNameDone = "Q";
            break;
        case "BR":
        case "WR":
            pieceNameDone = "R";
            break;
    }
    //create full string
    return pieceNameDone + vals[oldXAxis] + (oldYAxis+1).ToString() + attacking + vals[newXAxis] + (newYAxis + 1).ToString();
}
```

Castling has a separate notation used when moving, being separately noted with either ‘0-0-0’ as a king side castle, and ‘0-0’ for a queenside castle.

```
if (changeInX == 2)
{
    //kingside castle
    moveType = "0-0-0";
    changeInX = 1;
}
else
{
    //queenside castle
    moveType = "0-0";
}
```

Lastly, pawns that are promoted have a separate notation of just the position it is moving to once you promote a piece.

```
//Used for pawn being promoted
if (currentPiece.GetComponent<Pieces>().name[1] == 'P')
{
    bool check = currentPiece.GetComponent<Pieces>().checkPromote(true);
    char[] vals = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' };
    if (check)
    {
        moveType = "P" + vals[xBoard] + (yBoard+1).ToString();
    }
}
```

We can then call the function to create the moveType variable. Since the notation is different from taking a piece to doing an ordinary move, we call the same function but with different variables.

```
if (isAttack)
{
    moveType = translateNotation(currentPiece.GetComponent<Pieces>().xPos, currentPiece.GetComponent<Pieces>().yPos,
        xBoard, yBoard, true, currentPiece.GetComponent<Pieces>().name);
```

```
else
{
    moveType = translateNotation(currentPiece.GetComponent<Pieces>().xPos, currentPiece.GetComponent<Pieces>().yPos,
        xBoard, yBoard, false, currentPiece.GetComponent<Pieces>().name);
}
```

Lastly, we update the move to our list.

```
board.moveHistory.Add(moveType);
```

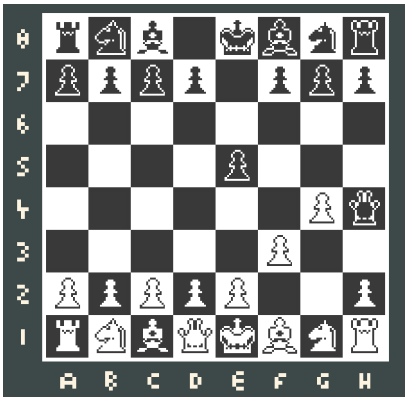
At the end of the game, we must update all the moves into a text file, to do this we need to use certain libraries:

```
using System.IO; //read write files
using System; //Date Time
```

Next, we create a function which gets the moves and creates a large string of every move done. Next, we create the path on which the file will write to, which is a combination of the current directory (being the location of where the current application is stored) and 'history.txt'. We then check if the path exists, assuming either it is the first time the application is opened, or that the user deleted it, we create a new file. Lastly, we update the full history, with a starting text which is in the format of 'Game At: currentTime, Move history:' where currentTime is the current time of when the function is called.

```
public void writeHistoryToFile()
{
    //This will write the list "moveHistory" to a file
    DateTime now = DateTime.Now; //gets current time
    string history = "\n\nGame at: "+ now.ToString("F") + "\nMove history:";
    for(int i = 0; i < moveHistory.Count(); i++)
    { //gets all moves and appends them to a string format
        history += " " + moveHistory[i];
    }
    string path = Environment.CurrentDirectory + "\\history.txt";
    if (!File.Exists(path)) //if the file does not exist
    {
        // Create a file to write to.
        string createText = Environment.NewLine;
        File.WriteAllText(path, createText);
    } //Writes the move history to file
    File.AppendAllText(path, history);
}
```

We then can call this function when the game ends in our game end function. The game below demonstrates the moves done through a game.



```
Game at: 31 December 2020 16:33:30
Move history: Pf2f3 Pe7e5 Pg2g4 Qd8h4 0-1
```

This completes stage two, where I implemented most of the chess rules and more complicated moves, however, currently there is more to add such as checkmate and the AI.

Feedback From Client:

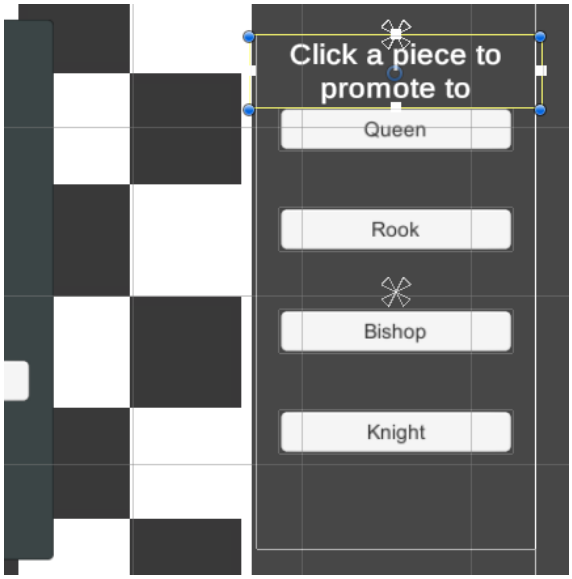
Client Feedback:

Client	Feedback
Oscar Butler	There still are some features missing such as checkmate stalemate and check, which are very important features to include. Furthermore, I would like to be able to choose which piece the pawn can promote to, instead of automatically promoting to a queen, since in certain circumstances, it may be better to do so.
Jake Elliot	Currently, there are some moves which I have yet to learn such as en passant and castling. It would be nice if there were either a description of all the moves in the game and rules.

Response:

Firstly, the “How To Play” menu which will hold the information on how to play a game will be included in later development stages during the end, as there are still things to implement such as more complicated rules such as checkmate, stalemate and the AI.

Secondly, to allow users to be able to select different promoting material, we need a menu which only appears during this. We can create a menu which is put to the side of the screen which has buttons which each allows the user to select different pieces:



We then want a way for the program to make the menu appear on the screen and allow the user to get the selected piece to appear on the board and instantiate it.

```
public void setPiece(int x, int y, string colour, int indexVal, string pieceName, bool isSelecting)
{
    //used in promoting the pawn once it reaches the other side, it creates a queen at the given position

    if (colour == "B" && againstAI)
    { //if it is the AI, it will automatically choose queen
        Destroy(blackPieces[indexVal]);
        blackPieces[indexVal] = createPiece("BQ", x, y);
        assignBoardPos(blackPieces[indexVal]);
    }
    else
    {
        if (!isSelecting)
        { //makes it so you cannot move until you choose a new piece
            canMove = false;
            //maximise menu
            GameObject.FindGameObjectWithTag("PromoteMenu").transform.localScale = new Vector3(1, 1, 1);
        }
        else
        {
            string piece = "";
            piece += colour;
            piece += pieceName;
            if (colour == "B")
            { //selecting black piece and assigns to board
                blackPieces[indexVal] = createPiece(piece, x, y);
                assignBoardPos(blackPieces[indexVal]);
            }
            else
            { //selecting white piece and assigns to board
                whitePieces[indexVal] = createPiece(piece, x, y);
                assignBoardPos(whitePieces[indexVal]);
            }
            //allows user to move
            canMove = true;
        }
    }
    setupAttackBoard();
}
```

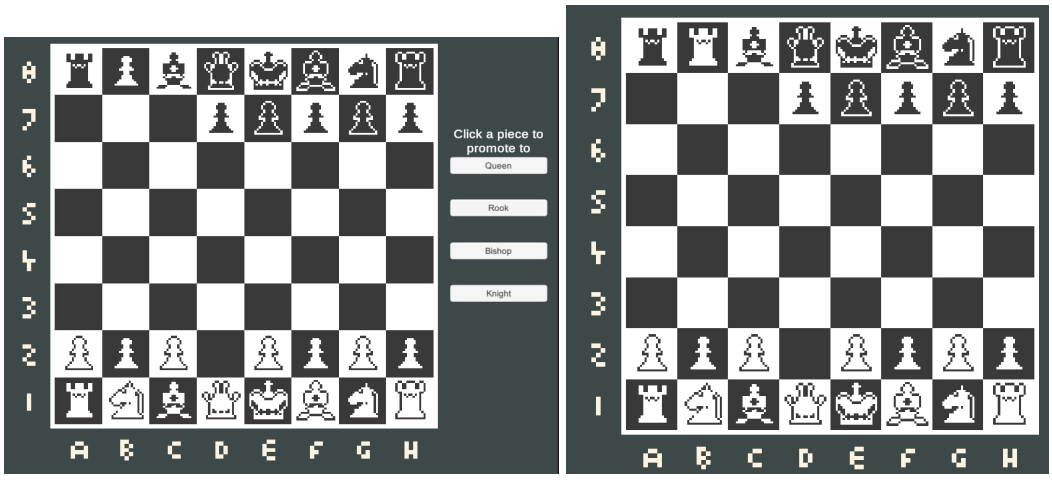
Next, we must assign each button such that it calls this function so that it will create the piece that user selects, whilst also minimising the menu so that it doesn't appear until another piece is promoted:

```
public void getQueen()
{
    //minimise the menu
    GameObject.FindGameObjectWithTag("PromoteMenu").transform.localScale = new Vector3(0,0,0);
    setPiece("Q");
}

0 references
public void getRook()
{
    //minimise the menu
    GameObject.FindGameObjectWithTag("PromoteMenu").transform.localScale = new Vector3(0, 0, 0);
    setPiece("R");
}

0 references
public void getBishop()
{
    //minimise the menu
    GameObject.FindGameObjectWithTag("PromoteMenu").transform.localScale = new Vector3(0, 0, 0);
    setPiece("B");
}

0 references
public void getKnight()
{
    //minimise the menu
    GameObject.FindGameObjectWithTag("PromoteMenu").transform.localScale = new Vector3(0, 0, 0);
    setPiece("Kn");
}
```

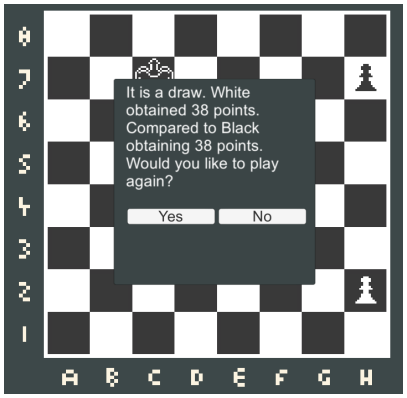
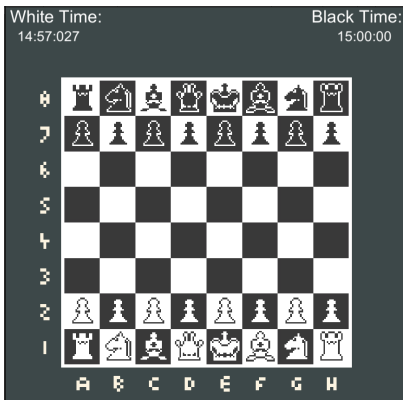
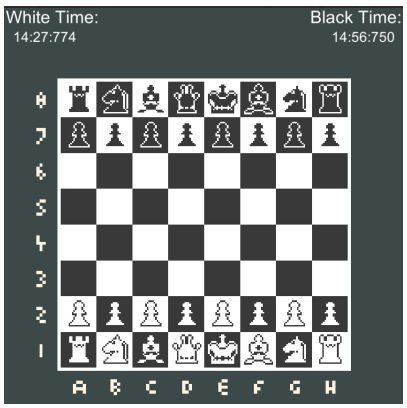
It will then find the pawn to replace by iterating through all the pawns and finding which one is at the end of the board. We then call the function to create the piece at the position at which the pawn used to be at

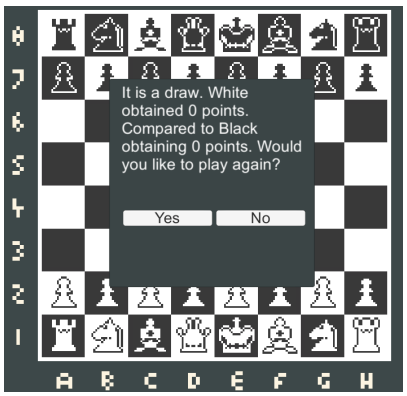
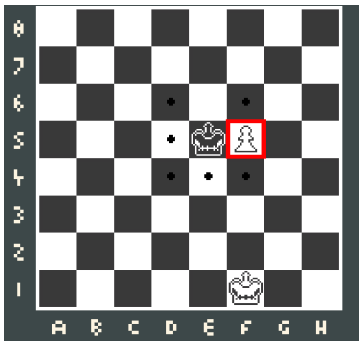
```
public void setPiece(string pieceName)
{
    int xpos = 0;
    int ypos = 0;
    //used to check if a pawn can be promoted
    Main board = controller.GetComponent<Main>();
    if (board.currentPlayer == "black")
    { //white selecting piece
        for (int i = 0; i < board.whitePieces.Length; i++)
        {
            if (board.whitePieces[i] != null)
            {
                if (board.whitePieces[i].GetComponent<Pieces>().name == "WP" && board.whitePieces[i].GetComponent<Pieces>().yPos == 7)
                { //If the piece is a pawn and is at the end of the board where it can promote
                    ypos = board.whitePieces[i].GetComponent<Pieces>().yPos;
                    xpos = board.whitePieces[i].GetComponent<Pieces>().xPos;
                    Destroy(board.whitePieces[i]); //destroy the pawn
                    board.setPiece(xpos, ypos, "W", i, pieceName, true); //create a new piece at the new xPos and yPos
                }
            }
        }
    }
    else
    { //black selecting piece
        for (int i = 0; i < board.blackPieces.Length; i++)
        {
            if (board.blackPieces[i] != null)
            {
                if (board.blackPieces[i].GetComponent<Pieces>().name == "BP" && board.blackPieces[i].GetComponent<Pieces>().yPos == 0)
                { //If the piece is a pawn and is at the end of the board where it can promote
                    ypos = board.blackPieces[i].GetComponent<Pieces>().yPos;
                    xpos = board.blackPieces[i].GetComponent<Pieces>().xPos;
                    Destroy(board.blackPieces[i]); //destroy the pawn
                    board.setPiece(xpos, ypos, "B", i, pieceName, true); //create a new piece at the new xPos and yPos
                }
            }
        }
    }
}
```

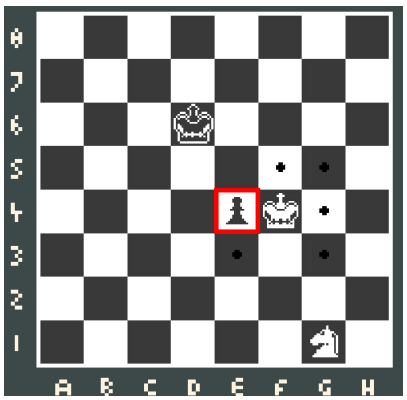
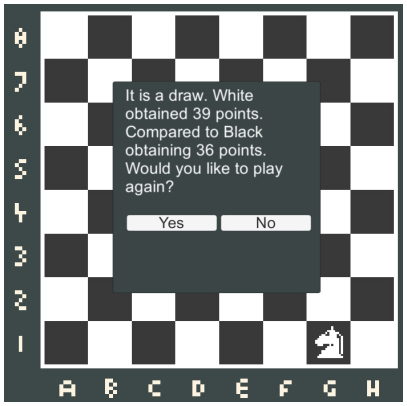
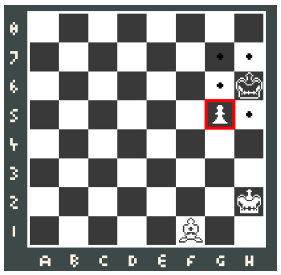
This will now allow users to select any piece type wanted once the pawn has reached the end of the board, this can be seen as so, where in this case the user selects the rook:

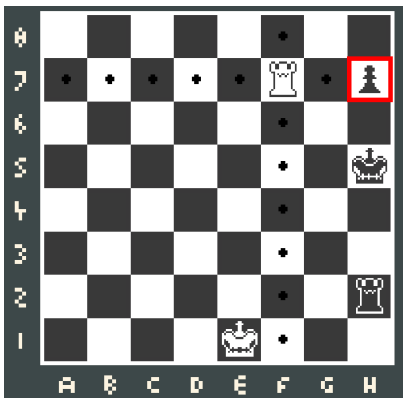
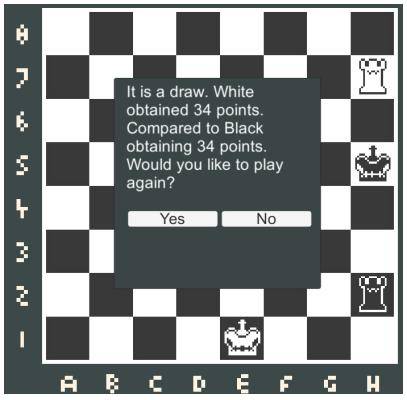
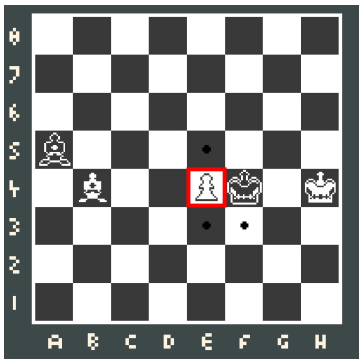
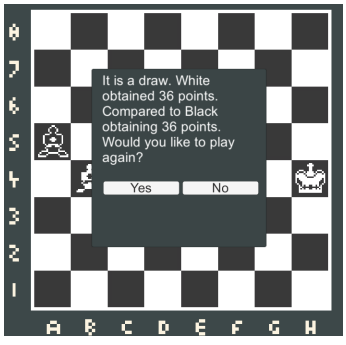


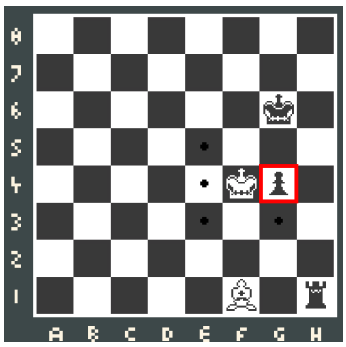
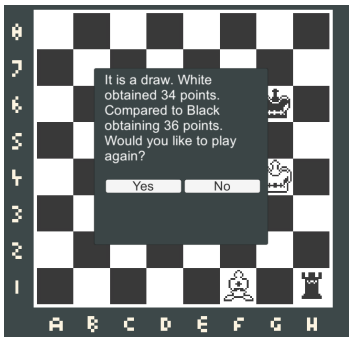
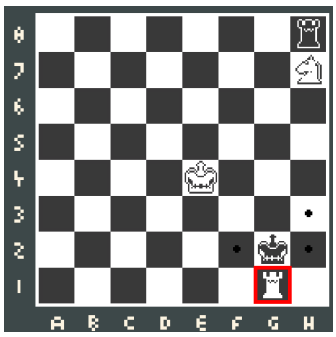
Tests:

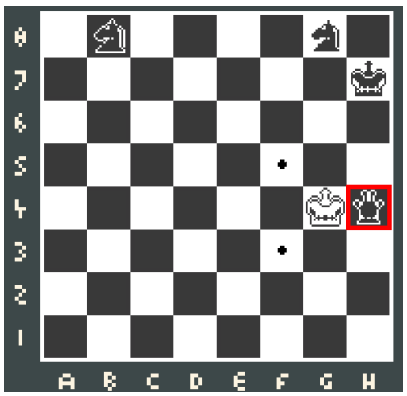
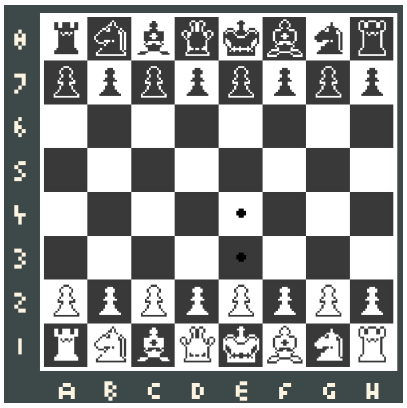
No	Test	Outcome	Evidence	Pass or Fail and actions taken
3f iv	No pawn is moved or piece is taken after 50 moves (meaning black and white moves)	Ends the game and both players draw. The “Play Again Menu” will also be opened.		**Pass**
v	The exact position has been repeated 3 times	Ends the game and both players draw. The “Play Again Menu” will also be opened.	**Position 1:**  **Position 2:**  **Position 3:**	**Pass**

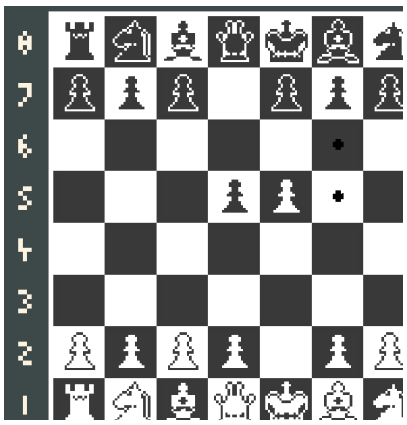
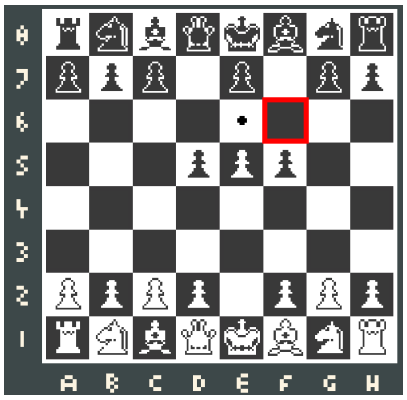
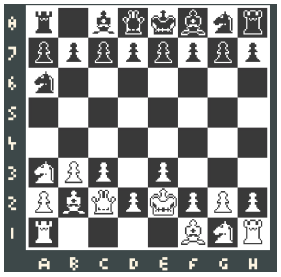
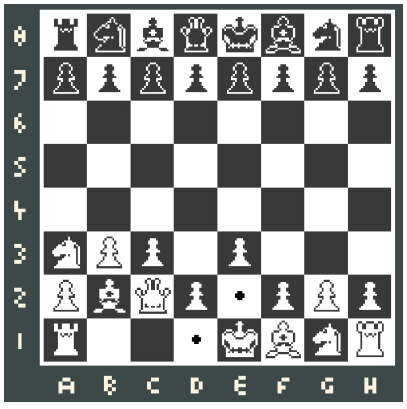
				
vi	Last remaining piece are king vs king	Ends the game and both players draw. The “Play Again Menu” will also be opened.	**Before:**  **After:** 	**Pass**
vii	Last remaining piece are knight, king vs king	Ends the game and both players draw. The “Play Again Menu” will also be opened.	**Before:**	**Pass**

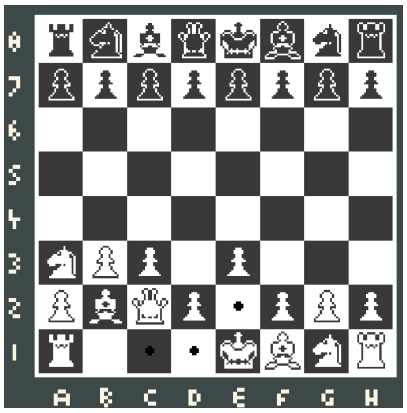
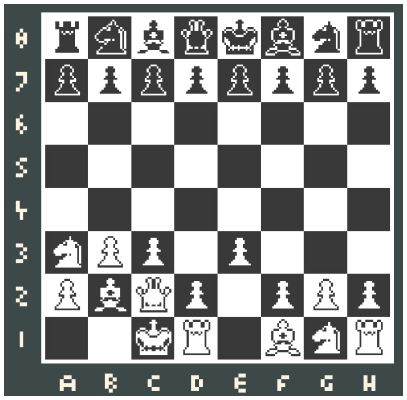
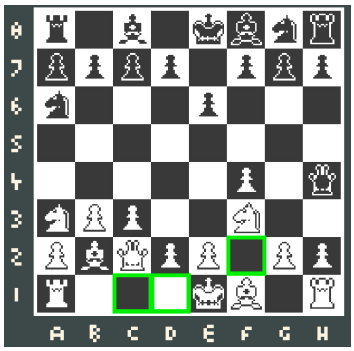
			 **After:** 	
viii	Last remaining piece are bishop king vs king	Ends the game and both players draw. The “Play Again Menu” will also be opened.	**Before:**  **After:** 	**Pass**
ix	Last remaining piece are rook, king vs rook, king	Ends the game and both players draw. The “Play Again	**Before:**	**Pass**

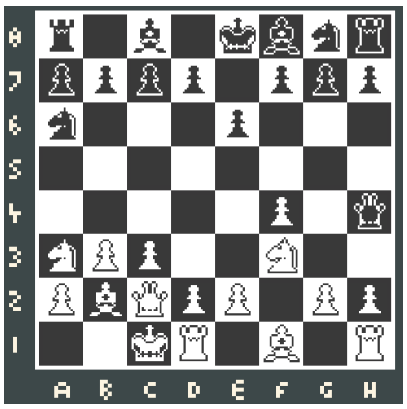
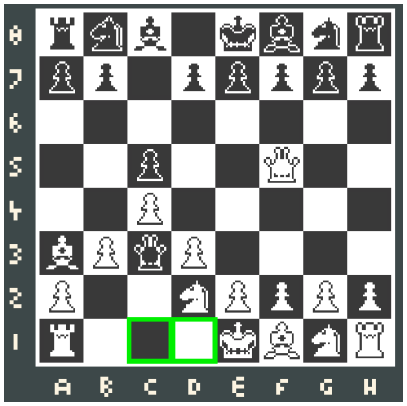
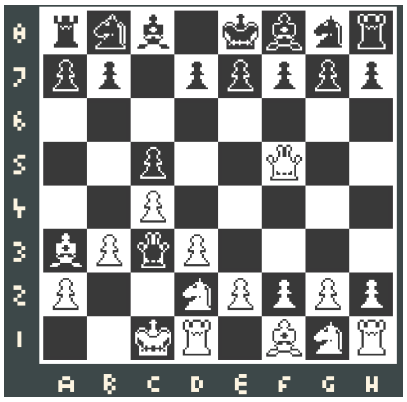
		Menu” will also be opened.	 **After:** 	
x	Last remaining piece are bishop, king vs bishop, king (where both bishops are on the same square)	Ends the game and both players draw. The “Play Again Menu” will also be opened.	**Before:**  **After:** 	**Pass**

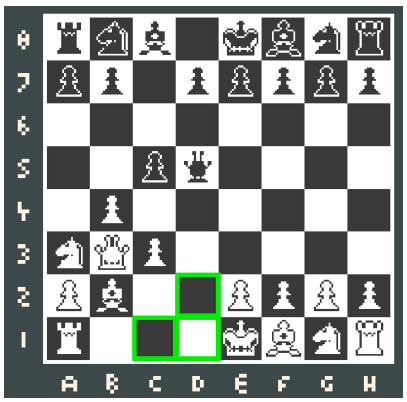
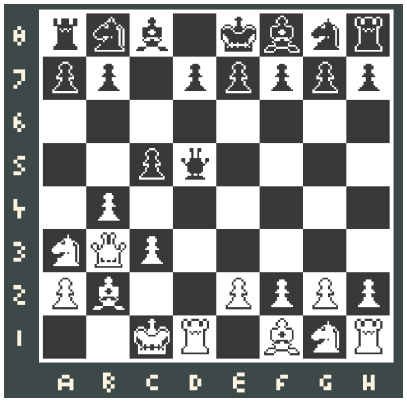
xi	Last remaining piece are rook, king vs bishop, king	Ends the game and both players draw. The “Play Again Menu” will also be opened.	**Before:**  **After:** 	**Pass**
xii	Last remaining piece are rook, king vs knight, king	Ends the game and both players draw. The “Play Again Menu” will also be opened.	**Before:**  **After:** 	**Pass**

xiii	Last remaining piece are knight, knight, king vs king	Ends the game and both players draw. The “Play Again Menu” will also be opened.	**Before:**  **After:** 	**Pass**
5ai	Clicking a pawn once it has not moved yet and is not being blocked	Two plates are created, if it is white both above the piece, else two below the piece		**Pass**
iii	Moving a pawn two spaces forward next to a pawn of opposite colour	The pawn should be able to en passant but for only one turn	**Before:**	**Pass**

			 **After:** 	
bi	The king or castle has moved and you try to castle	The piece will not be able to castle	**Before:**  **After:** 	**Pass**

ii	The king or castle hasn't moved and you try to castle	The piece should move	**Before:**  **After:** 	**Pass**
iii	Try to castle while under check	Castling occurs	**Before:**  **After:**	**Fail** The King should not be able to castle, we can fix this once we implement check in a later development stage.

				
iv	Moving the king into checkmate	The piece moves	**Before:**  **After:** 	**Fail** The King should not be able to castle, we can fix this once we implement checkmate in a later stage.
v	Castling through an attacking square	The castle occurs	**Before:**	**Fail** The King should not be able to castle, we can fix this once we implement attack squares in a later development

			 **After:** 	stage.

Stage Three Of Development:

In this stage, I will implement more king moves for checkmate and improve on castling. Some attempts to create an AI will also be made.

Advanced Moves:

Check:

One of the most important rules of chess is check, where if the king is currently under attack, or if the chosen move causes the king to be under attack, the move should be void and not be possible. To do this, we need to know which locations are currently being attacked by either side, and whether or not the king is currently in this location, or will be in an attacking location of the opposing side. To do this, we need to store the positions under attack by replicating the board and finding each positions under attack:

```
//stores attack positions of each colour
public bool[,] whiteAttackBoard = new bool[8, 8];
public bool[,] blackAttackBoard = new bool[8, 8];
```

Firstly, we want to set each position to false, and do this every time so there is no overlap between moves.

```
public void setupAttackBoard()
{
    //Used to create the attack board which stores the positions of each sides attacking positions
    for (int xBoard = 0; xBoard < 8; xBoard++)
    {
        for (int yBoard = 0; yBoard < 8; yBoard++)
        { //resets the board by setting each position to false
            whiteAttackBoard[xBoard, yBoard] = false;
            blackAttackBoard[xBoard, yBoard] = false;
        }
    }
```

Next, we iterate through all moves the pieces can do, however, we must change a large portion of the pieces script, so that instead of actually creating move tiles / attack tiles, it will update the new positions on the board:

```
for (int currentPos = 0; currentPos < 16; currentPos++)
{
    if (whitePieces[currentPos] != null) //null error stop
    { //gets all possible attacking positions of white side
        whitePieces[currentPos].GetComponent<Pieces>().possibleMoves(false);
    }
    if (blackPieces[currentPos] != null)//null error stop
    { //gets all possible attacking positions of black side
        blackPieces[currentPos].GetComponent<Pieces>().possibleMoves(false);
    }

}
```

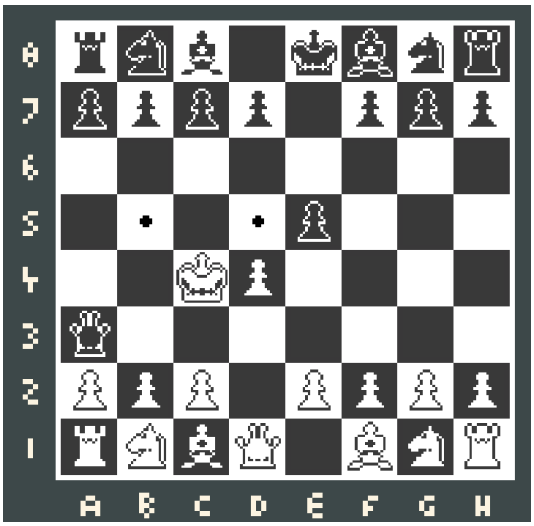
We then update all move types to call this function, which will update the position currently under attack to true.

```
public void setAttackBoardPos(int x, int y)
{
    Main board = controller.GetComponent<Main>();
    if(playerColour == "W")
    {
        board.whiteAttackBoard[x, y] = true;
    } else if (playerColour == "B")
    {
        board.blackAttackBoard[x, y] = true;
    }
}
```

Now, once we have a table which stores the positions currently being attack, we can now update the king's moves such that it will now not allow the king to move to a position currently being attacked, and since the king uses the inputMove function, we can just update it so it will check if the piece is a king, and the positions is currently not being attacked:

```
if ((name == "WKi" && board.blackAttackBoard[newXPos, newYPos] != true) || (name == "BKi" && board.whiteAttackBoard[newXPos, newYPos] != true)
    || (name != "BKi" && name != "WKi"))
{
    GameObject cp = board.boardPositions[newXPos, newYPos];
    if (cp == null)
    {
        passiveMove(newXPos, newYPos, isPlace);
    }
    else if (cp.GetComponent<Pieces>().playerColour != playerColour)
    {
        attackTile(newXPos, newYPos, isPlace);
    }
}
```

As seen below, the king now cannot move to any location which is currently being under attack:



However, we still have a problem, if a piece moves to another location, the king can become attacked after the fact, to solve this, we must emulate the move in advance. Firstly we must know whether or not the king is in check, if it is true, we must set the position to true, such that we now know the king is currently under check:

```
public bool inCheck()
{
    //Returns if in check or not, setting the king in check if it is
    if (currentPlayer == "white") //if the current player is white
    {
        if (blackAttackBoard[whitePieces[4].GetComponent<Pieces>().xPos, whitePieces[4].GetComponent<Pieces>().yPos])
        { //if king is being attacked, set check to true
            whitePieces[4].GetComponent<Pieces>().inCheck = true;
            return true;
        }
        else
        {
            whitePieces[4].GetComponent<Pieces>().inCheck = false;
        }
    }
    else
    {//if the current player is black
        if (whiteAttackBoard[blackPieces[4].GetComponent<Pieces>().xPos, blackPieces[4].GetComponent<Pieces>().yPos])
        {//if king is being attacked, set check to true
            blackPieces[4].GetComponent<Pieces>().inCheck = true;
            return true;
        }
        else
        {
            blackPieces[4].GetComponent<Pieces>().inCheck = false;
        }
    }
    return false;
}
```

Now that we know the king is currently under check in certain positions, we can now begin to emulate check, by emulating what the MoveTile class does, however without actually doing the move:

```
public bool moveOutCheck(int newXPos, int newYPos) //-> true = cannot move, false = possible move
{   //Check if the move is legal / won't cause check on own side
    bool isAttack = false;
    //replicating the piece
    int refXPos = xPos;
    int refYPos = yPos;
    int index = 0;
    GameObject refObj = gameObject;
```

Firstly, we must get each scenario setup, firstly if the position causes an attack. The following code shown will only be for the white side, as it is replicated on the black side, but with different values:

```
if (name!= "WKi" && name!= "BKi")
// The kings do not need to use this script
{
    if (playerColour== "W")
        //If the piece is white
    {
        if (board.boardPositions[newXPos,newYPos] != null)
            //check if attack move
        {
            isAttack = true;
            index = Array.IndexOf(board.blackPieces, board.boardPositions[newXPos, newYPos]);
            //get the piece that it is attacking
            refObj = board.blackPieces[index];
            board.blackPieces[index] = null;
            //removes the piece
            board.boardPositions[newXPos, newYPos] = null;
        }
```

Next, we can emulate en passant with similar values:

```
if (enPassantAttack && board.boardPositions[newXPos, newYPos] == null &&
    board.boardPositions[newXPos,newYPos-1].GetComponent<Pieces>().playerColour != playerColour)
{ //similar script as before but refers to when a piece can enpassent
    isAttack = true;
    index = Array.IndexOf(board.blackPieces, board.boardPositions[newXPos, newYPos - 1]);
    refObj = board.blackPieces[index];
    board.blackPieces[index] = null;
    board.boardPositions[newXPos, newYPos + 1] = null;
}
```

We then move the piece to the corresponding location and create a new attack board, as we have fully emulated the move and need to emulate the attack board:

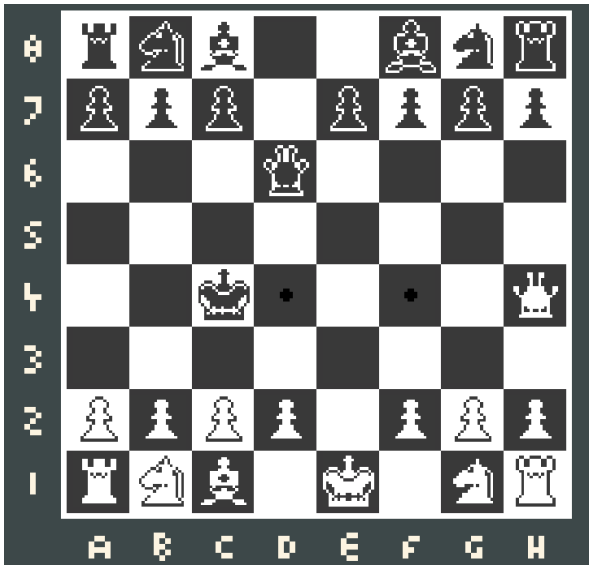
```
//Moves the piece to the corresponding location
board.boardPositions[xPos, yPos] = null;
xPos = newXPos;
yPos = newYPos;
board.assignBoardPos(gameObject);
//creates a new attackboard, to check if in check
board.setupAttackBoard();
```

We then check if the king is currently under check due to this, if so we reassign all the pieces and return true, meaning the move still causes the move to be under check, meaning that the move is illegal, else we return false:

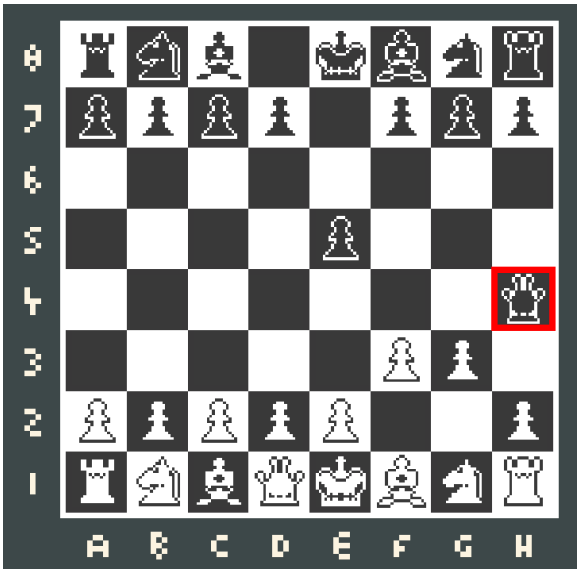
```
if (checkInCheck("W"))
{ //resets the board to the previous position
    board.boardPositions[newXPos, newYPos] = null;
    if (isAttack)
    {
        //reassigns old piece to board
        board.blackPieces[index] = refObj;
        board.assignBoardPos(board.blackPieces[index]);
    }
    xPos = refXPos;
    yPos = refYPos;
    board.assignBoardPos(gameObject);
    return true;
}
```

```
else
{//resets the board to the previous position
    board.boardPositions[newXPos, newYPos] = null;
    if (isAttack)
    {
        board.blackPieces[index] = refObj;
        board.assignBoardPos(board.blackPieces[index]);
        //reassigns the place
    }
    xPos = refXPos;
    yPos = refYPos;
    board.assignBoardPos(gameObject);
    return false;
}
```

This will mean that now a piece cannot move to a location where the piece will cause a check on their own side. As you can see below, the queen can only move to a position which stops the check, and in the other scenario, the piece can only capture the piece:



“Clicking the White Queen on H4”



“Clicking the White Pawn on G3”

However, there is currently a problem, if there are no legal moves on the board, the game won't be able to continue, as one side will be unable to do any moves. This is either known as checkmate or stalemate.

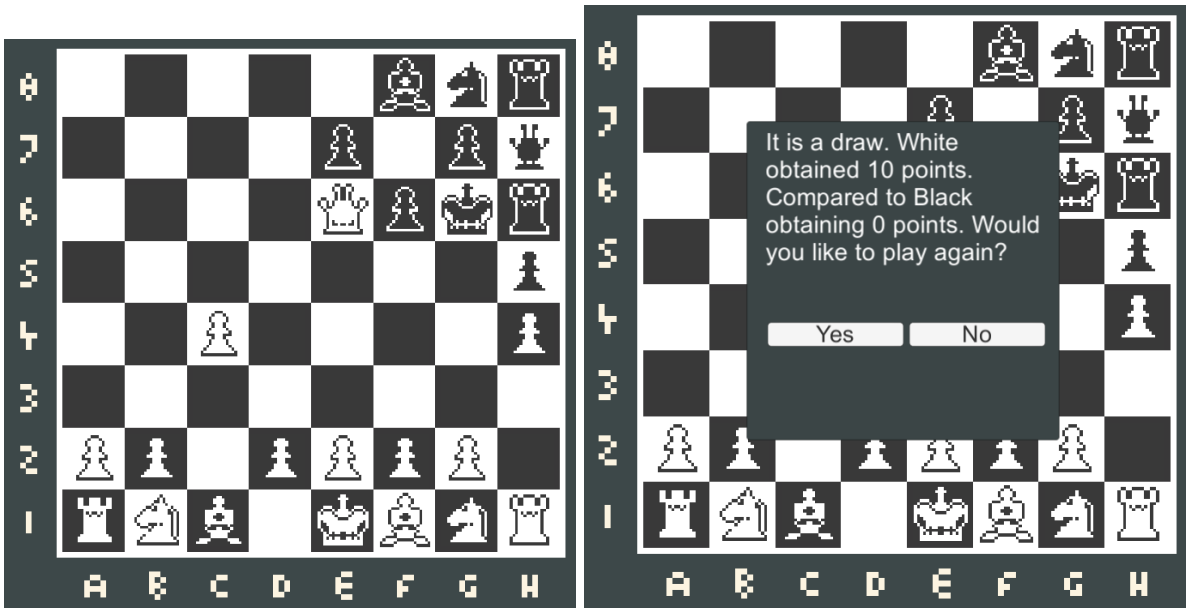
Stalemate and Checkmate:

Stalemate occurs if the king is currently not under check, and the side has no available legal moves. We first must check whether or not the king has any legal moves, which if this

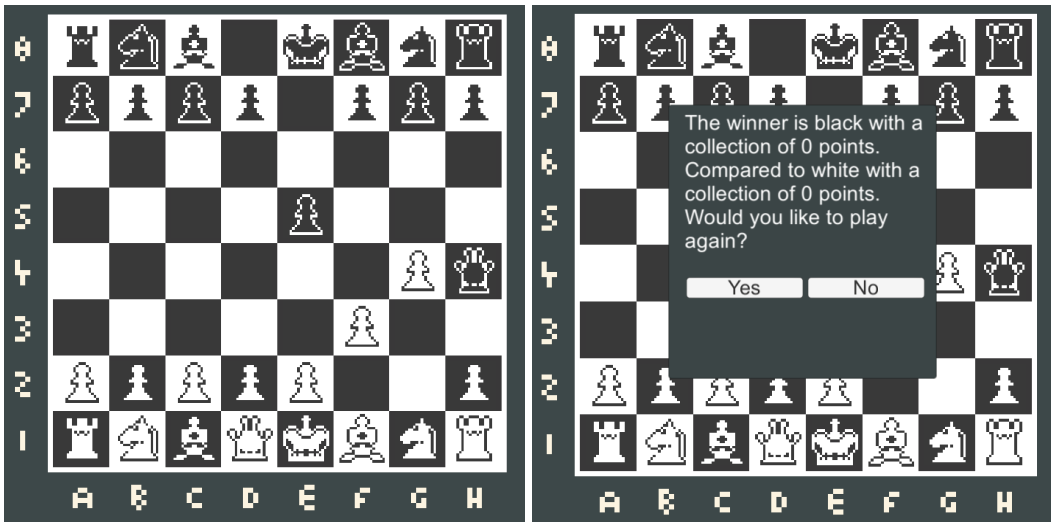
is not the case we then get all the moves the player can do, and if there are none, we then end the game.

```
//Seeing that it cant move anywhere (stalemate)
GameObject[] movePlates = GameObject.FindGameObjectsWithTag("MovePlate");
if (movePlates.Length == 0 && isPlace)
{//king have no moves
    if (playerColour == "W")
    {//iterate through all pieces, getting their moves
        for (int currentPiece = 0; currentPiece < 16; currentPiece++)
        {
            if (board.whitePieces[currentPiece] != null && board.whitePieces[currentPiece].GetComponent<Pieces>().name != "WKi")
            {
                board.whitePieces[currentPiece].GetComponent<Pieces>().possibleMoves(true);
            }
        }
        //get all the new moves
        GameObject[] newMovePlates = GameObject.FindGameObjectsWithTag("MovePlate");
        if (newMovePlates.Length == 0)
        {//if there are no legal moves, its stalemate, draw the game
            DestroyTiles();
            board.drawEnd();
        }
```

This is demonstrated below:



Lastly, implementing checkmate is very similar and only needs one change. Since we already implemented stalemate, we can already know if checkmate is possible, this is when the king is already in check and there are no available moves. We can use the exact script above to do this and can be seen below:



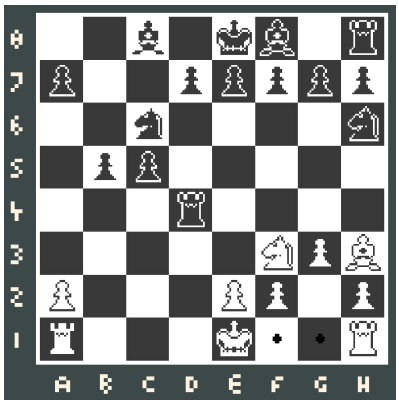
Improving Castling:

Currently, castling has a few issues where there are some illegal moves. This is where, if the king is already in check, the king cannot castle out of check, as well as if any position that the king passes through while the position is currently under attack, the king cannot move.

```
if (!inCheck && isPlace)
{

    Main board = controller.GetComponent<Main>();
    //check for castle
    if (playerColour == "B" && !hasMoved)
    {
        if (board.boardPositions[7, 7] != null)
        {//kingside castle
            if (!board.boardPositions[7, 7].GetComponent<Pieces>().hasMoved && board.boardPositions[6, 7] == null && board.boardPositions[5, 7] == null
                && board.whiteAttackBoard[5,7] != true)
            {
                castle(7,7);
            }
        }
        if (board.boardPositions[0, 7]!= null)
        {//queenside castle
            if (!board.boardPositions[0, 7].GetComponent<Pieces>().hasMoved && board.boardPositions[1, 7] == null && board.boardPositions[2, 7] == null
                && board.boardPositions[3, 7] == null && board.whiteAttackBoard[3,7] != true)
            {
                castle(0,7);
            }
        }
    }
```

```
else if (playerColour == "W" && !hasMoved)
{
    if (board.boardPositions[0, 0] != null)
    {//queenside castle
        if (!board.boardPositions[0, 0].GetComponent<Pieces>().hasMoved && board.boardPositions[1, 0] == null && board.boardPositions[2, 0] == null
            && board.boardPositions[3, 0] == null && board.blackAttackBoard[3, 0] != true)
        {
            castle(0,0);
        }
    }
    if (board.boardPositions[7, 0] != null)
    {//kingside castle
        if (!board.boardPositions[7, 0].GetComponent<Pieces>().hasMoved && board.boardPositions[6, 0] == null && board.boardPositions[5, 0] == null
            && board.blackAttackBoard[5,0] != true)
        {
            castle(7,0);
        }
    }
}
```

This will now mean that if the king is currently under attack, or any of the positions it passes through during castling, castling will not be a legal move. As you can see below, castle queen side due to an attack castle, but can still kingside castle:



AI Class:

Selecting a move:

Firstly, we need the AI to be able to select all the available moves that it can do. Since the AI will always be on the black side, we need to first ensure that the user can never move a black piece when playing against an AI. This can be done by updating our function in the Pieces class:

```
private void OnMouseUp()
{
    //When the user clicks it will destroy all tiles that had originally been there (if the player clicked on a piece and it didnt move), and
    //show all possible moves for the piece that was clicked on this is done by giving the prefab a 2d box collider
    Main board = controller.GetComponent<Main>();
    DestroyTiles();
    if (!board.gameOver && board.canMove)
    {//if the user can move
        if (!board.againstAI && board.currentPlayer[0].ToString() == playerColour.ToLower())
        {
            possibleMoves(true);
        }
        //if against AI (only white can select moves)
        else if (board.againstAI)
        {
            if (board.currentPlayer == "white")
            {
                possibleMoves(true);
            }
        }
    }
}
```

Next, in the AI class, we need to be able to select different moves. First, we iterate through all pieces and find each tile they make, and store the positions. These are all the moves the AI can make currently:

```
public void getMove()
{
    controller = GameObject.FindGameObjectWithTag("GameController");
    for (int currentVal = 0; currentVal < controller.GetComponent<Main>().blackPieces.Length; currentVal++)
    {
        if (controller.GetComponent<Main>().blackPieces[currentVal] != null)
        {
            controller.GetComponent<Main>().blackPieces[currentVal].GetComponent<Pieces>().possibleMoves(true);
        }
    }
```

We could randomly select these moves, however, this would not be great to do as it would have no clue on what is a good move. We could evaluate moves by how much a piece is worth, however there is a better method as written below.

Positional Analysis:

In chess, despite all pieces having a constant worth, technically there are places on the board where the piece may be worth more theoretically, such as a knight in the middle of the board may be worth more than if it were isolated in the corner of the board. We can follow a table as shown below to get the true value of the piece:

```
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],
[ -1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],
[  2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0 ],
[  2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0 ]
```

```
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -0.5,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[  0.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[ -1.0,  0.5,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]
```

```
[  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
[  0.5,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[  0.0,   0.0, 0.0,  0.5,  0.5,  0.0,  0.0,  0.0]
```

```
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0],
[ -1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0],
[ -1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0],
[ -1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0],
[ -1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0],
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]
```

```
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0],
[-4.0, -2.0,  0.0,  0.0,  0.0,  0.0, -2.0, -4.0],
[-3.0,  0.0,  1.0,  1.5,  1.5,  1.0,  0.0, -3.0],
[-3.0,  0.5,  1.5,  2.0,  2.0,  1.5,  0.5, -3.0],
[-3.0,  0.0,  1.5,  2.0,  2.0,  1.5,  0.0, -3.0],
[-3.0,  0.5,  1.0,  1.5,  1.5,  1.0,  0.5, -3.0],
[-4.0, -2.0,  0.0,  0.5,  0.5,  0.0, -2.0, -4.0],
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]
```

```
[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
[5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0],
[1.0,  1.0,  2.0,  3.0,  3.0,  2.0,  1.0,  1.0],
[0.5,  0.5,  1.0,  2.5,  2.5,  1.0,  0.5,  0.5],
[0.0,  0.0,  0.0,  2.0,  2.0,  0.0,  0.0,  0.0],
[0.5, -0.5, -1.0,  0.0,  0.0, -1.0, -0.5,  0.5],
[0.5,  1.0, 1.0,  -2.0, -2.0,  1.0,  1.0,  0.5],
[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0]
```

This however is only for white pieces, so we mirror the position such that it works for both black and white:

```
int newXPos = 7 - xPos;
int newYPos = 7 - yPos;
```

We can then create a large switch statement which then calculates the true value of each piece and how much the true value of the move would cost due to this:

```
public float calculateMove(GameObject moveTile)
{
    float moveVal = 0;
    if (moveTile.GetComponent<MoveTiles>().isAttack == true)
    {
        moveVal += controller.GetComponent<Main>().boardPositions[moveTile.GetComponent<MoveTiles>().xBoard, moveTile.GetComponent<MoveTiles>().yBoard].GetComponent<Pieces>().getPieceWorth();
    }
    switch (moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().name)
    {
        case ("WKi"):
        case ("BKi"):
            moveVal += getKingTable(moveTile.GetComponent<MoveTiles>().xBoard, moveTile.GetComponent<MoveTiles>().yBoard) -
                getKingTable(moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().xPos, moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().yPos);
            break;
        case ("WKn"):
        case ("BKn"):
            moveVal += getKnightTable(moveTile.GetComponent<MoveTiles>().xBoard, moveTile.GetComponent<MoveTiles>().yBoard) -
                getKnightTable(moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().xPos, moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().yPos);
            break;
        case ("WQ"):
        case ("BQ"):
            moveVal += getQueenTable(moveTile.GetComponent<MoveTiles>().xBoard, moveTile.GetComponent<MoveTiles>().yBoard) -
                getQueenTable(moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().xPos, moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().yPos);
            break;
        case ("WP"):
        case ("BP"):
            moveVal += getPawnTable(moveTile.GetComponent<MoveTiles>().xBoard, moveTile.GetComponent<MoveTiles>().yBoard) -
                getPawnTable(moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().xPos, moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().yPos);
            break;
        case ("WR"):
        case ("BR"):
            moveVal += getRookTable(moveTile.GetComponent<MoveTiles>().xBoard, moveTile.GetComponent<MoveTiles>().yBoard) -
                getRookTable(moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().xPos, moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().yPos);
            break;
    }
    return moveVal;
}
```

We can then call this function above for each move, which then returns the true value of the move. We can do this by expanding onto our getMove() function:

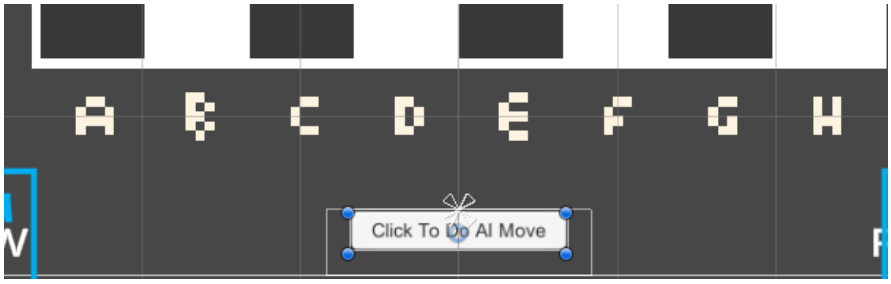
```
public void getMove()
{
    //gets moves
    controller = GameObject.FindGameObjectWithTag("GameController");
    for (int currentVal = 0; currentVal < controller.GetComponent<Main>().blackPieces.Length; currentVal++)
    {
        if (controller.GetComponent<Main>().blackPieces[currentVal] != null)
        {
            controller.GetComponent<Main>().blackPieces[currentVal].GetComponent<Pieces>().possibleMoves(true);
        }
    }
    GameObject[] moveTiles = GameObject.FindGameObjectsWithTag("MovePlate");
    //array of the worth of each move
    float[] calculatedMoves = new float[moveTiles.Length];
    for (int moveVal = 0; moveVal < moveTiles.Length; moveVal++)
    {
        //calculate moves
        calculatedMoves[moveVal] = calculateMove(moveTiles[moveVal]);
    }
    //get the highest and do the move
    moveTiles[getHighestVal(calculatedMoves)].GetComponent<MoveTiles>().doMove();

}
```

Lastly, we then need to get the highest value in the array by iterating through the calculated moves, and finding which value is the highest, then do this move:

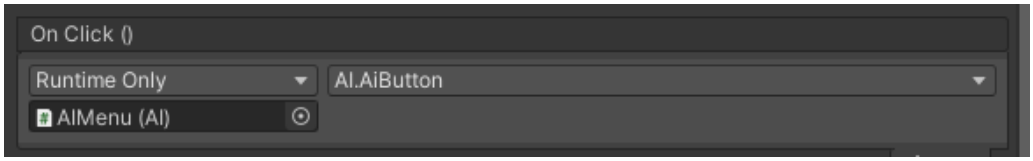
```
public int getHighestVal(float[] valArray)
{
    float highestVal = valArray[0];
    int returnIndex = 0;
    for (int currentVal = 1; currentVal < valArray.Length; currentVal++)
    {
        if (valArray[currentVal] >= highestVal)
        {
            highestVal = valArray[currentVal];
            returnIndex = currentVal;
        }
    }
    return returnIndex;
}
```

UI:

As the time for an AI to calculate each move is near millisecond, as this program is mainly meant for newer players, the likelihood of the user running out of time when thinking for each move may cause a large disadvantage to them. In this case, we can create a button which allows the AI to move, which the user must click to do that move to create an equal game with little to no time constraints:



We can then assign it the AI script:



```
public void AiButton()
{
    //When the user clicks on the AI button
    Main main = controller.GetComponent<Main>();
    //if the AI can move
    if (main.currentPlayer == "black" && main.againstAI == true
        && main.gameOver == false && main.canMove == true)
    {
        getMove();
    }
}
```

Lastly, we need to make sure that the button only appears when playing against the AI. This is done by finding the gameobject and enabling it during the start of the scene.

```
if(!againstAI)
{
    //If you are not playing against an AI, the AI button is not needed
    GameObject.FindGameObjectWithTag("AIButton").transform.localScale = new Vector3(0, 0, 0);
}
```

Implementing Minimax:

Minimax is an algorithm that allows an AI to have depth for the different moves. By getting all moves possible, and repeating for each move recursively, we can predict the moves taken by both the AI and the player. There is already a description of how Minimax works in the design, and can be seen implemented as so:

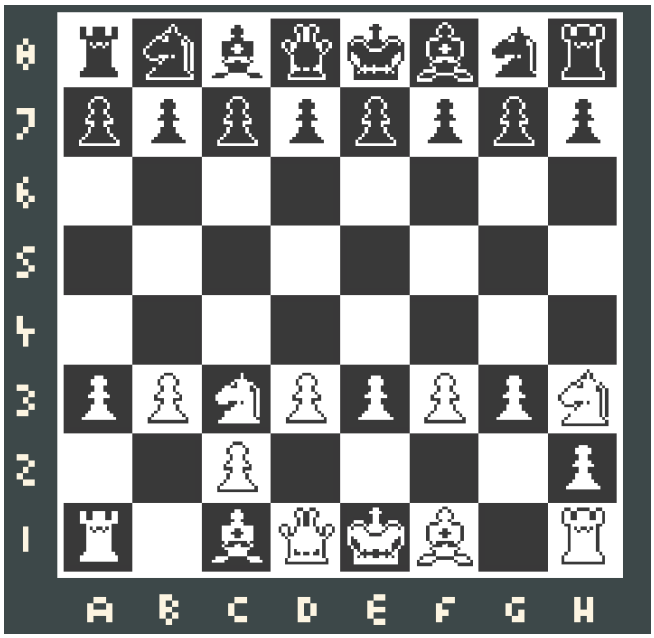
```
public float minimax(int depth, bool maximizingPlayer)
{
    if (depth == 0)
    {
        return calculatePosition();
    }

    if (maximizingPlayer)
    {//AI side
        float maxEval = -10000000000;
        //get all possible moves for the black side using the below arrays
        GameObject[] childMoves = getBlackMoves();
        for (int currentPos = 0; currentPos < childMoves.Length; currentPos++)
        {
            childMoves[currentPos].GetComponent<MoveTiles>().doMove();
            //calculate each move
            float evaluedMove = minimax(depth - 1, false);
            if (maxEval < evaluedMove)
            {
                maxEval = evaluedMove;
            }
        }//return calculated move / position
        return maxEval;
    }
```

```
else
{ //Player side
    float minEval = 10000000000;
    //get all possible moves for the white side using the below arrays
    GameObject[] childMoves = getWhiteMoves();
    for (int currentPos = 0; currentPos < childMoves.Length; currentPos++)
    {
        childMoves[currentPos].GetComponent<MoveTiles>().doMove();
        //calculate each move
        float evaluedMove = minimax(depth - 1, true);
        if (minEval > evaluedMove)
        {
            minEval = evaluedMove;
        }
    }//return calculated move / position
    return minEval;
```

We then can call this function for each moves in the doMove function:

```
public void getMove()
{
    for (int currentVal = 0; currentVal < controller.GetComponent<Main>().blackPieces.Length; currentVal++)
    {
        if (controller.GetComponent<Main>().blackPieces[currentVal] != null)
        {
            controller.GetComponent<Main>().blackPieces[currentVal].GetComponent<Pieces>().possibleMoves(true);
        }
    }
    GameObject[] moveTiles = GameObject.FindGameObjectsWithTag("MovePlate");


    float maxVal = -10000000;
    int index = 0;
    for (int currentTile = 0; currentTile < moveTiles.Length; currentTile++)
    {
        moveTiles[currentTile].GetComponent<MoveTiles>().doMove();
        float checkVal = minimax(0, false);
        if (maxVal > checkVal)
        {
            maxVal = checkVal;
            index = currentTile;
        }

    }
    moveTiles[index].GetComponent<MoveTiles>().doMove();

}
```

However, there is currently a problem. The AI currently can only do the move and not emulate it. This means the move will appear on the screen which is not what we want to happen, since we just want to calculate the moves.



Currently, due to this issue, I have had to remove the Minimax algorithm, and stick with what was used before, this drastically decreases the skill to beat the AI, however, due to the complexity of implementing it correctly, I had not enough time to do so.

To compensate for this, we can simulate a depth of one. We first can check if we move to a certain position, if we will be attacked. We can then assume that in most cases, the piece will be taken if this occurs. So we can then subtract the total score of the piece if this occurs. This can be implemented like so:

```
//determine if the piece can be taken if you move to this location using the attack board
if(controller.GetComponent<Main>().whiteAttackBoard[moveTile.GetComponent<MoveTiles>().xBoard, moveTile.GetComponent<MoveTiles>().yBoard] == true)
{ //(we subtract the value of the piece if so, assuming it would take the piece)
    moveVal -= controller.GetComponent<Main>().boardPositions[moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().xPos,
        moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().yPos].GetComponent<Pieces>().getPieceWorth() * 10;
}
```

Feedback From Client:

Client Feedback:

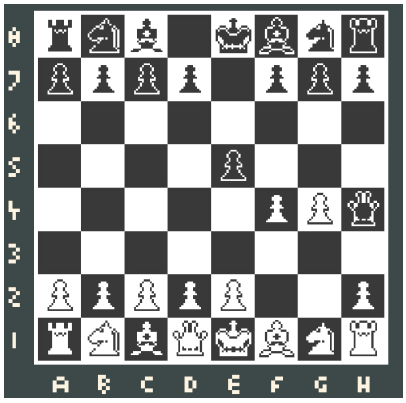
Client	Feedback
Oscar Butler	Currently the project is in a good state and since almost every promised feature has now been included, it is doing well. However, the AI is currently very weak and is quite easy to beat especially for an advanced player like me. It would be nice if also there were a dictionary of different openings so that the AI does not play the same opening each time.
Jake Elliot	The AI was challenging at first, however, due to the same opening being played each time, you can exploit this and play the best known opening

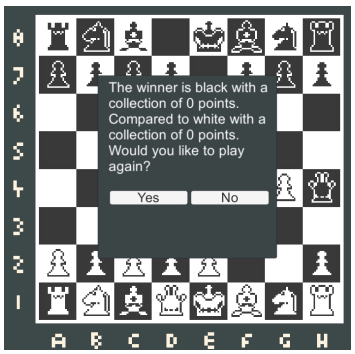
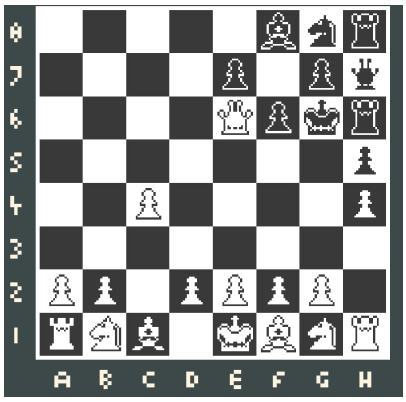
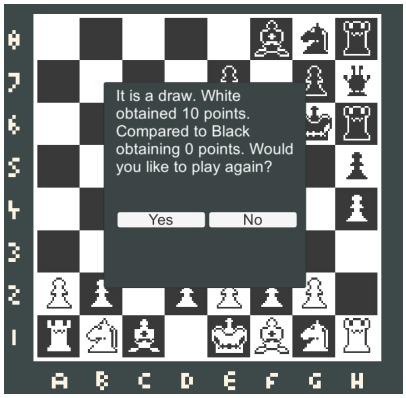
	against it, and the AI will be at a large disadvantage. It would be nice so that the AI plays a wider range of moves at the beginning since playing the same opening each time makes it easy to play against. Furthermore, the UI still needs to be changed to become more appealing as a blank screen with buttons that navigate through is not enough; images should be included to make the application more enticing.

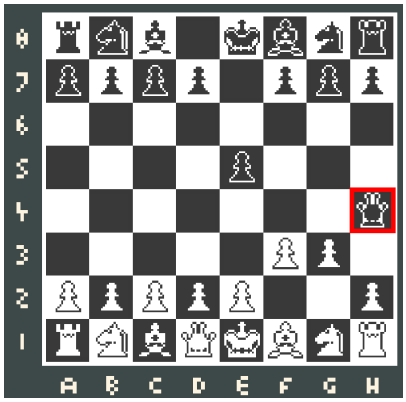
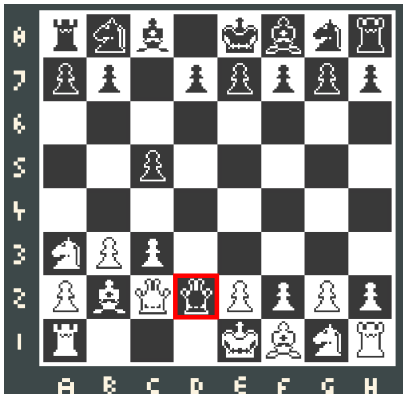
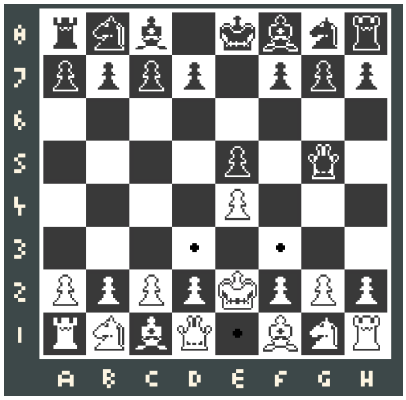
Response:

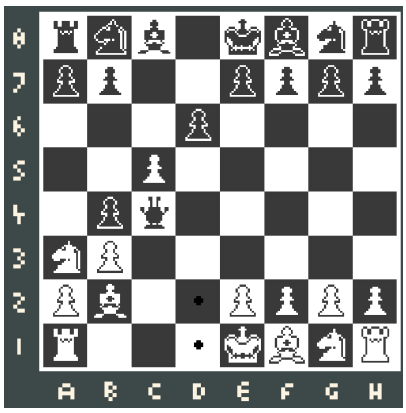
Due to the inability to being able to correctly implement the Minimax algorithm, the AI is currently very low skilled, however, the AI was meant to be able to challenge newer players and not more advanced players, since players who start out playing chess will have little knowledge of how to play, it would be unfair to pit them against a program with a much higher rating against it. At a later stage, I may try to implement a range of AI difficulties that the user can play against, so that every user has a range of AI to play against that may find it more challenging to beat than others. Lastly, in the last stage, the UI will be changed to accommodate a more enticing layout.

Tests:

No	Test	Outcome	Evidence	Pass or Fail and actions taken
3f ii	Once checkmate is reached	Ends the game and the current player wins. The “Play Again Menu” will also be opened.	**Before:**  **After:**	**Pass**

				
iii	Once stalemate is reached	Ends the game and both players draw. The “Play Again Menu” will also be opened.	**Before:**  **After:** 	**Pass**

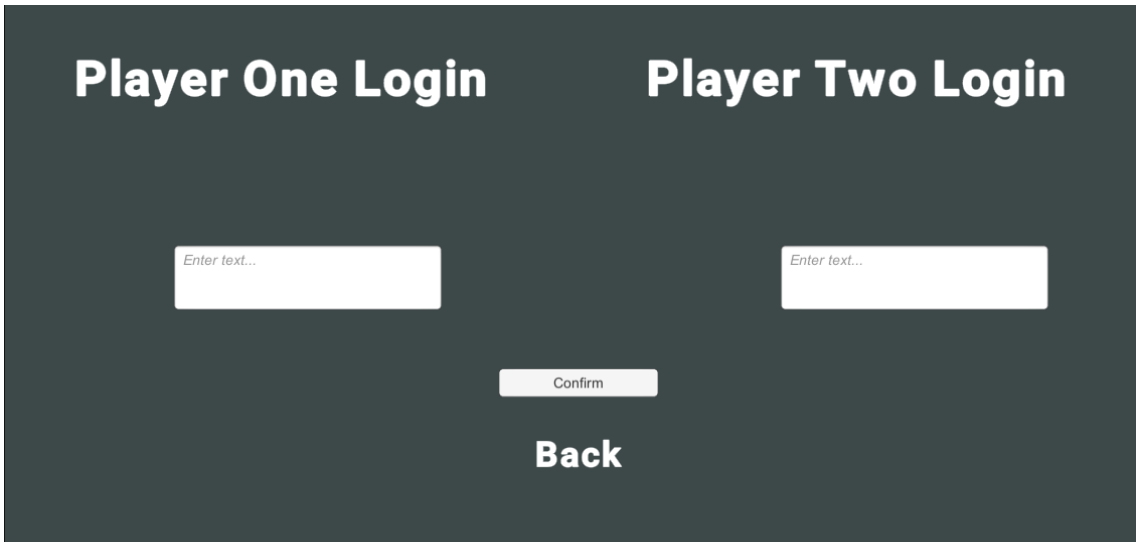
4d	Moving a piece into check of your own king / while you king is in check occurs	The piece should only be able to move if it will stop the check		**Pass**
5b iii	Try to castle while under check	The piece will not be able to move		**Pass**
iv	Moving the king into checkmate	The piece will not be able to move		**Pass**

v	Castling through an attacking square	The piece will not be able to move		**Pass**
7	Clicking the “Click to move AI piece” button	AI does a legal move and rotates to other players turn		**Pass**

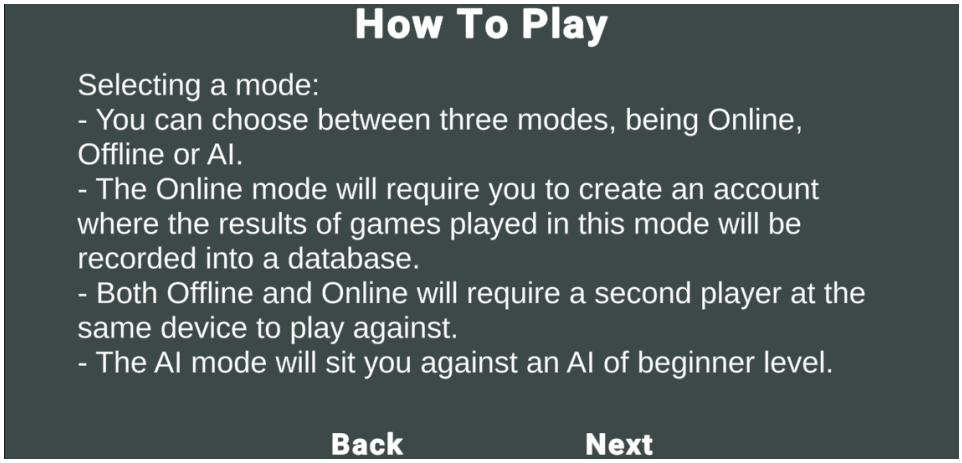
Final Stage Of Development:

In this final stage, I will make minor improvements to certain aspects of the project, however, due to time limitations, adaptations of code will be done to compensate for missing features.

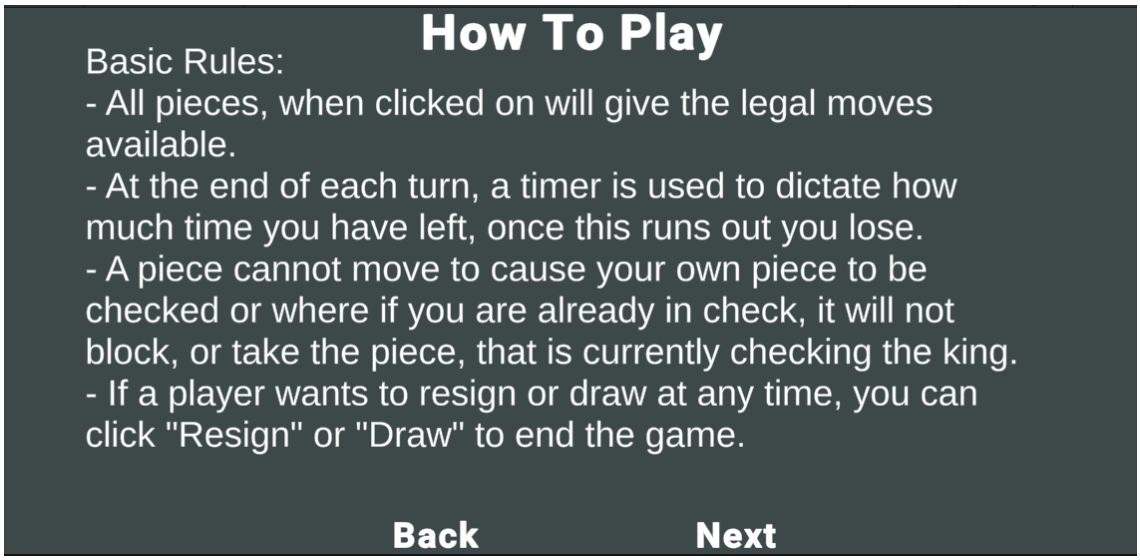
UI:
Firstly, I began to create a basic Online menu screen which will be assigned with functions later on in this stage. It consists of two input fields and buttons, which will get users to input their names, as well as buttons which start the game, or confirm their usernames.



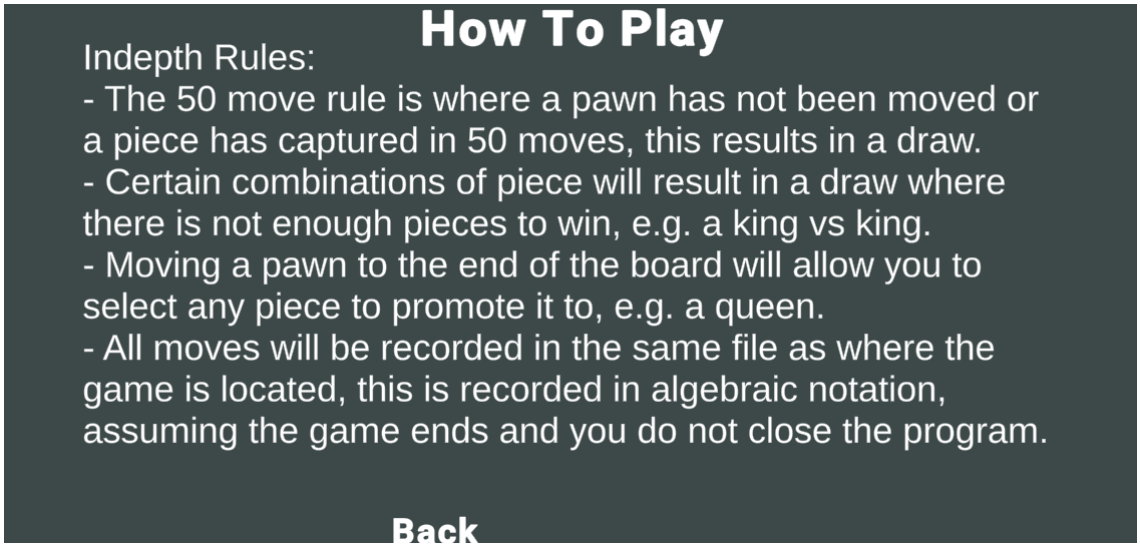
Next, I added onto the “How To Play” menu with basic rules on how to use the application, as well as more advanced rules which players might not be familiar with.
Menu 1:



Menu 2:



Menu 3:



Buttons have also been added to navigate through these menus.

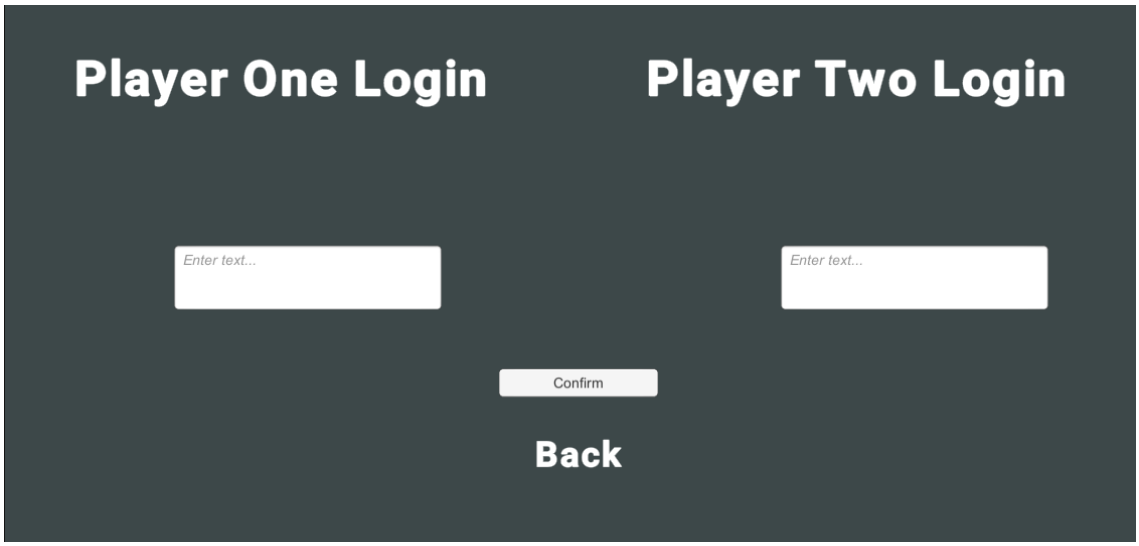
Remedial actions taken:

Currently, due to time limitations, I have yet to implement the database which stores user information about their games and skills. To compensate for this, I will allow users to create accounts, where all the data which previously was going to be available will just be stored onto their local machine. Whilst there are limitations with being able to access your account on a separate device, you can still have the program on a USB or external storage

device and store your account information on this, and as long as you have the device running the program off the external storage device, you can access your account via that.

Firstly, we must have a way for users to input data about their account. As security is not a huge focus of this project, to ensure that no important information is being stored which the user does not feel uncomfortable to be seen, we will not store any passwords or emails for validation since this information will not be securely stored. This means we will only store their username, which is used to identify each user, games won, games lost and games drew.

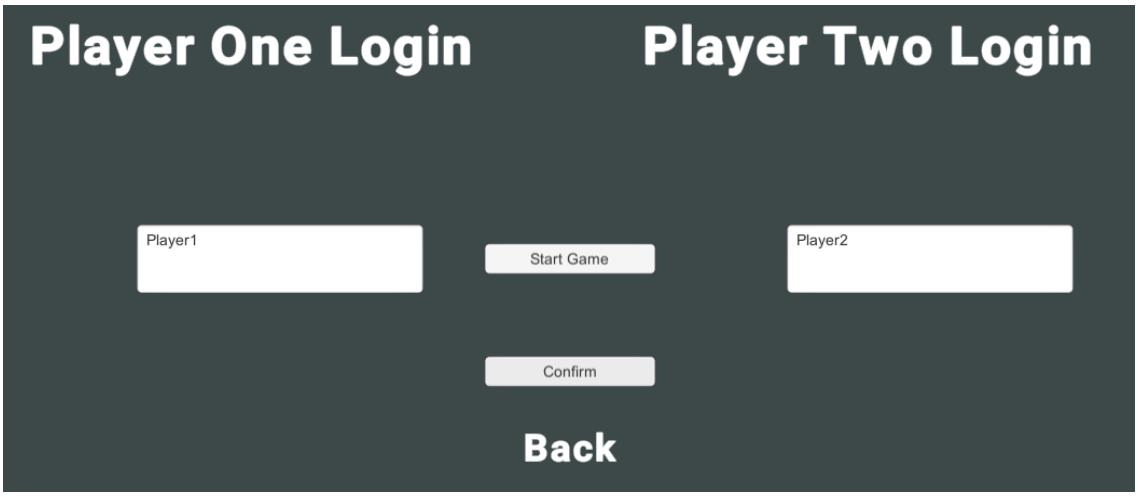
We then need a way for the user to input data. We can do this via the menu which we originally created for the “Online” section. As the only data needed to be input by the user is just their username, we only need one field for each user, as well as a button to submit this data.



We then need to ensure that the data being input is in the correct format, where it only consists of letters, numbers, underscores, is between length 4-16 and is not equal. We do only need to check this every time the “confirm” button is clicked. We assign it a function which calls below:

```
public void GetInputOnClickHandler()
{
    //Names contain only letters, numbers and underscore, has length 4-16 characters, and not equal
    if(Regex.IsMatch(inputPlayerOne.text, @"^[a-zA-Z0-9_]+$") && Regex.IsMatch(inputPlayerTwo.text, @"^[a-zA-Z0-9_]+$")
        && inputPlayerOne.text.Length <= 16 && inputPlayerTwo.text.Length <= 16 && inputPlayerOne.text.Length >=4 && inputPlayerTwo.text.Length >= 4
        && inputPlayerOne != inputPlayerTwo)
    {
        //Set names, and show a button to start the game
        playerOneUsername = inputPlayerOne.text;
        playerTwoUsername = inputPlayerTwo.text;
        GameObject.FindGameObjectWithTag("StartGame").transform.localScale = new Vector3(1, 1, 1);
    }
    else
    {
        //Minimise the start game button
        GameObject.FindGameObjectWithTag("StartGame").transform.localScale = new Vector3(0, 0, 0);
    }
}
```

Once the correct information is inputted, a new button appears to ensure that users cannot start an online game without the correct information. Once you click this button, an online game begins.




```
0 references
public void startOnlineGame()
{
    controller.GetComponent<Main>().isOnlineGame = true;
    controller.GetComponent<Main>().startGame();
}
```

Once the game ends, we then assign each user their respective points, and store this information back into the file. We must also do the same when reading from the file and getting the list of usernames and information. However, due to time constraints, I have yet to implement this. Currently, users can create accounts and play a game, but there has yet to be an implementation of a full login, sign-in and recording of information.

Conclusive Feedback From Client:

Client Feedback:

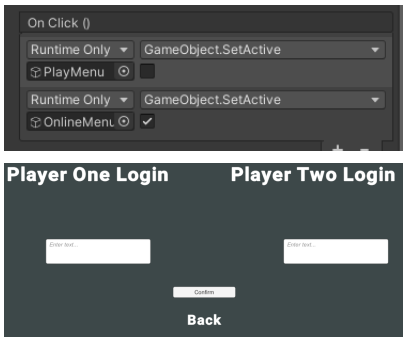
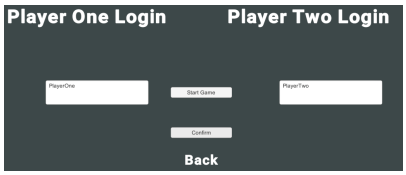
Client	Feedback
Oscar Butler	As many of the promised features are currently included in the program, I believe the project has gone very well. However, there are many small optimisations that could be made such as the AI, which in its current state, is very easy to beat and still has little knowledge. Secondly, whilst the Online feature wasn't too important, it would have been nice to record the games I have had with players I have played against on this application. However, as the basic chess game itself is fully working, with all the chess rules and moves implemented, this application is perfect for playing against others in a normal environment.

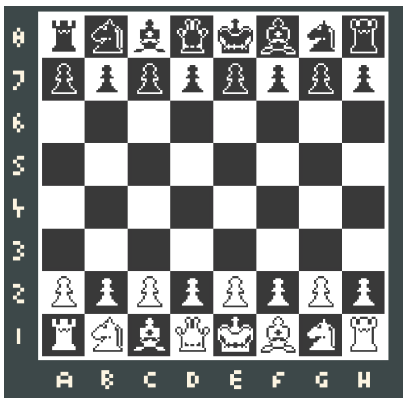
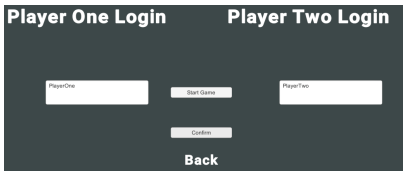
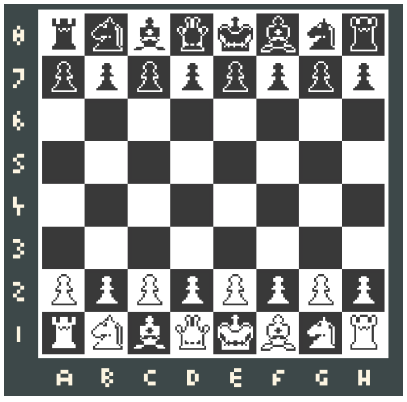
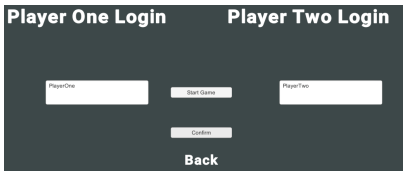
Jake Elliot	The Online option currently does need more developed onto it. I would like to be able to record the scores I play against for other players, however, as this problem can easily be fixed by doing it manually and recording it by hand, it isn't much of an issue. Secondly, the UI on this menu is very ugly, it would be nicer to see a more sophisticated menu screen.

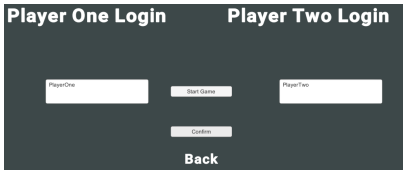
Response:

As the project is reaching the final stage, I do believe there are still many things to improve, as well as what has been already listed. Firstly, I would like to improve the UI to add more variety and colour. Whilst currently the Board UI is fine, the Main Menu still is only made of text and buttons. Furthermore, the AI is at a very weak stage, however, as this project is targeted towards mainly newer players, a weak AI is fine at first. If I were to have more time, I would develop the AI to have multiple modes to play against, each with their respective rankings, such that we reach a larger audience of players. Lastly, the Online mode does need more development, especially due to the fact that it doesn't record. However, as all these listed problems are currently not an essential part of the program, the program should still work nevertheless.

Tests:

No	Test	Outcome	Evidence	Pass or Fail
1c iii	Clicking on “Online” in the Play menu	Opens the “Login Menu”		**Pass**
ii	Inputting correct fields into the login screen	Login to the account and create a new game in an online environment	 Usernames: PlayerOne, PlayerTwo	**Pass**

				
di	Clicking on “Create Account” in the Login menu	Nothing, there is currently no menu for this.	**Yet to be implemented**	**Fail**
ii	Clicking on “Sign-in” in the Login menu	Creates a new game in an Online environment	 Usernames: PlayerOne, PlayerTwo 	**Pass**
2ai	Input correctly formatted information into each field when creating a new account	Signs into account	 Usernames: PlayerOne, PlayerTwo	**Partial Pass -** Account information is not stored after creating account
ii	Input already used email or account names when creating a new account	Nothing, there is currently no check for this	**Yet to be implemented**	**Fail**

bi	Inputting incorrect passwords with an email address when logging in	Nothing, there is currently no check for this	**Yet to be implemented**	**Fail**
ii	Inputting correct fields into the login screen	Login to the account and create a new game in an online environment	 Usernames: PlayerOne, PlayerTwo	**Partial Pass -** Account information is not store, so it cannot check for proper validation
8a	Player 1 wins, Player 2 loses	Nothing	**Yet to be implemented**	**Fail**
b	Player 1 loses, Player 2 wins	Nothing	**Yet to be implemented**	**Fail**
c	Both players draw	Nothing	**Yet to be implemented**	**Fail**

Multiple comprehensive videos testing functionality of the project can be located in the same folder as the project.

Future / Maintenance needed:

Currently, there are no aspects of code of data that relies on external inputs such as an API. This means that at its current state, there is no need to future proof code to work in cases where inputs may change. Many of the functions have already been tested in many different use cases and are unlikely to break. Furthermore, the project currently does not rely on any online database, however, in later stages of development, will need maintenance if implemented.

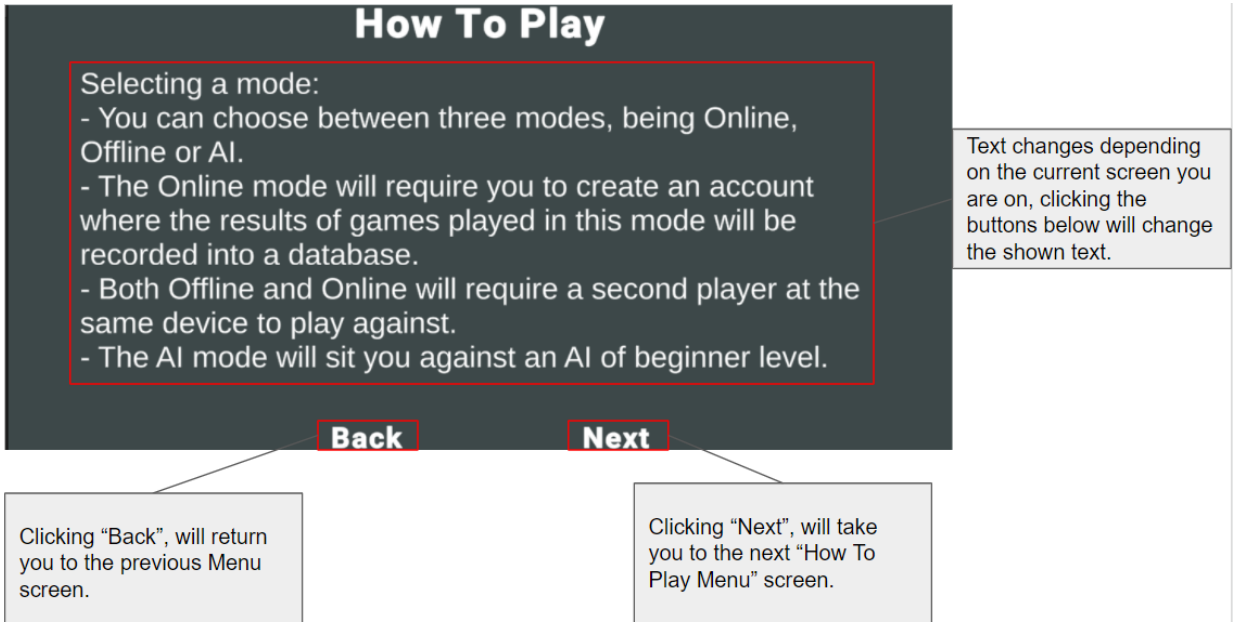
4. Evaluation of the Solution

Usability features

Main Menu:

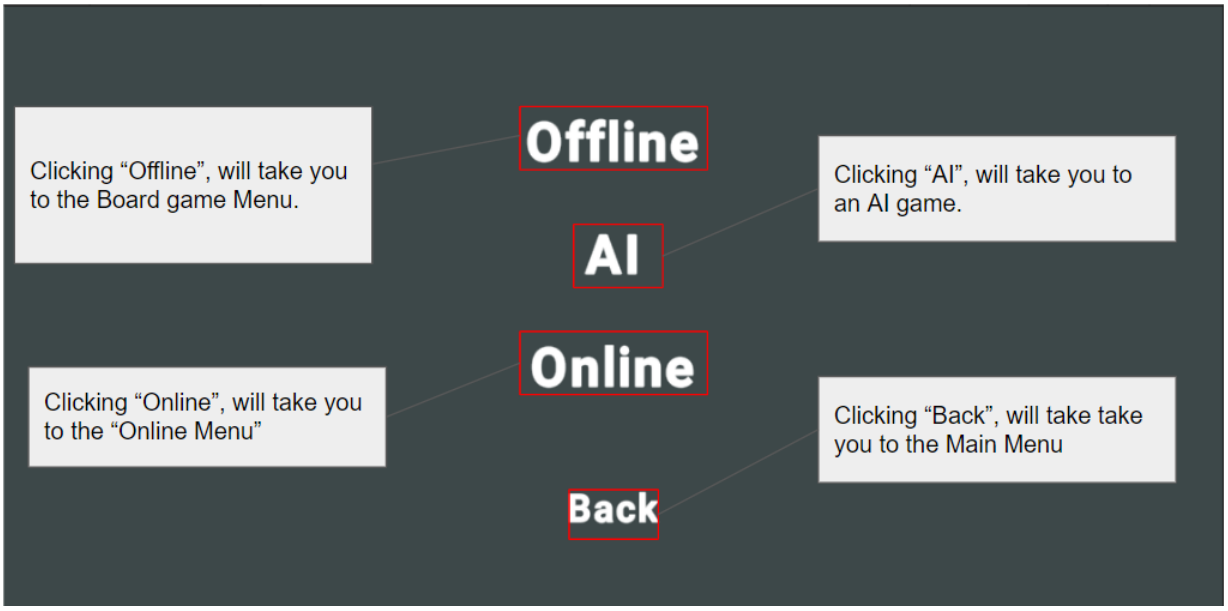
Currently, at this stage, the Main Menu UI is very bland and has no enticing imagery or interesting colour scheme. Whilst it has kept the theme of a "simplistic design" there are still many improvements which could be made to make the UI more appealing. This should be an essential task in following development stages to create a more enticing UI.

How To Play Menu:



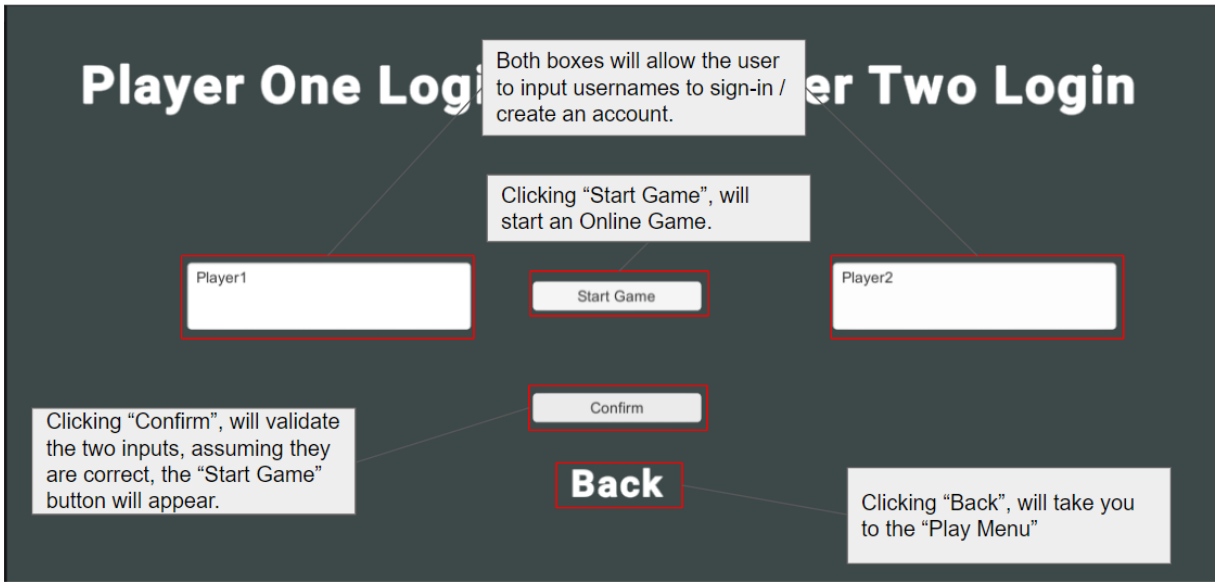
The How To Play Menu currently does offer a wide variety of information that the user will likely find very useful. Similarly to the Main Menu, some improvements could be made, however, seeing how this Menu is already filled with a large amount of information, where pictures are not necessary, images could be used to help describe certain rules easier.

Play Menu:



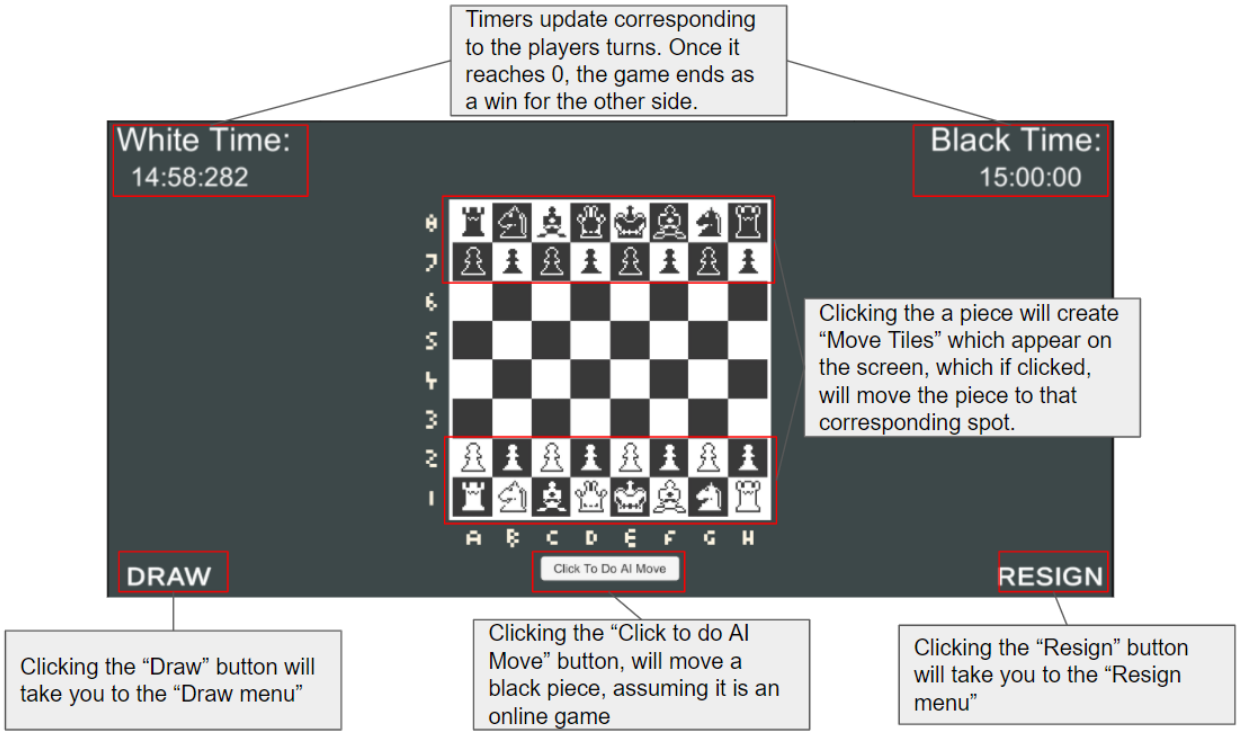
Similar to the Main Menu, the UI is very bland and could be improved, however, seeing how this is not the “Main” menu, it does not need as many improvements, since the use of a Main menu attracts users to your product.

Online Menu:



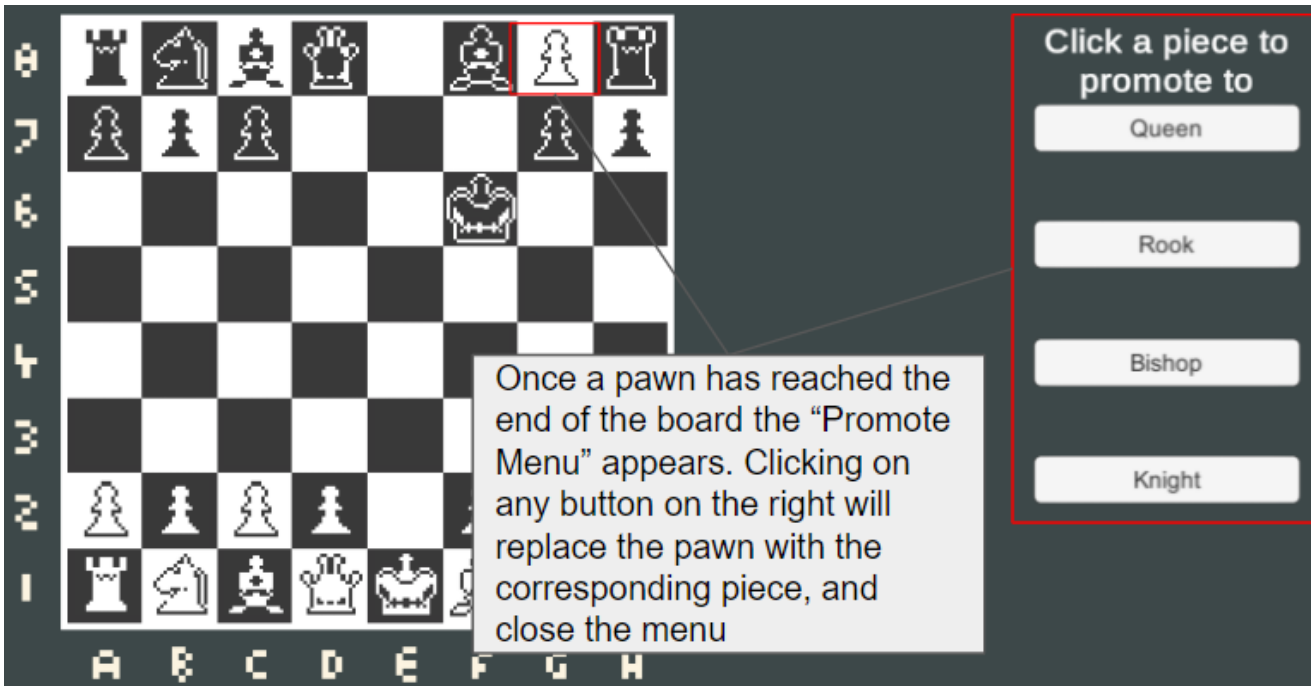
Currently, the Online Menu does lack the “Create account”, “Edit account details” or “View account details” Menus. This should be an essential feature to add on to the program in later stages, however, since we currently are not storing any of this information, and that this section has yet to be fully developed, there are currently no issues about the users account details, since we cannot change information on a users data as it currently doesn't exist.

Board Game / AI Game / Online Game:



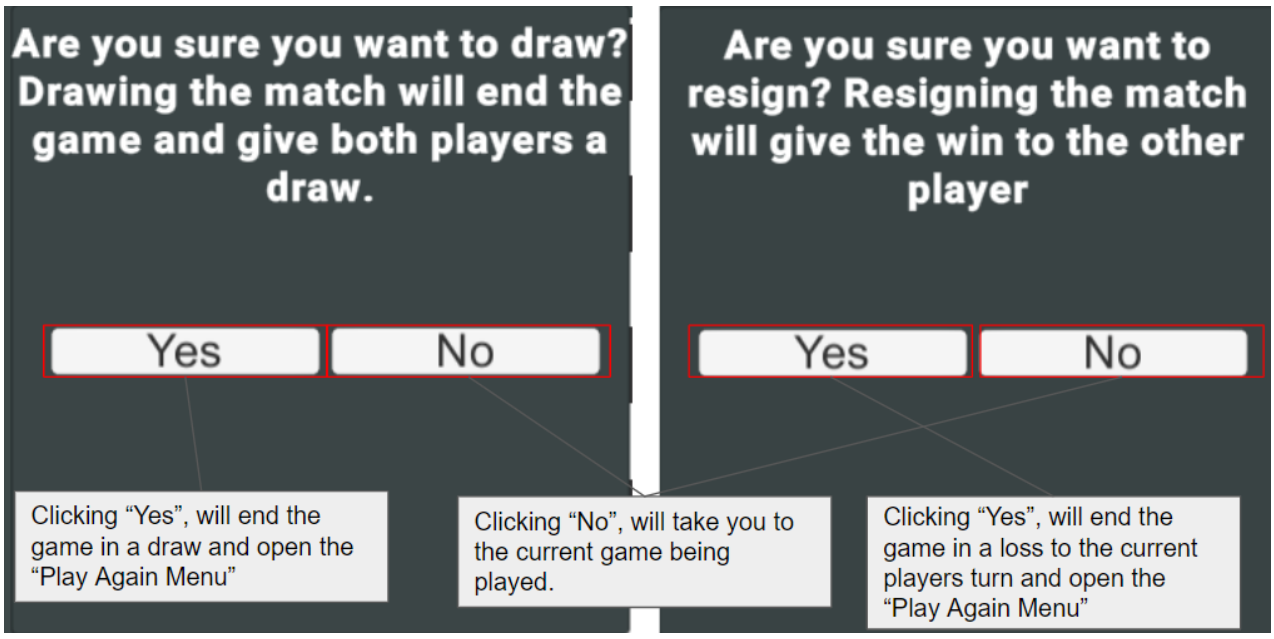
Currently, this section has the essential and proposed features which were analysis and design stage. Currently, there are no improvements which are needed to this section besides from minor UI improvements, however, at its current stage, it has reached a perfect place as nothing needs to change.

Promote Menu:



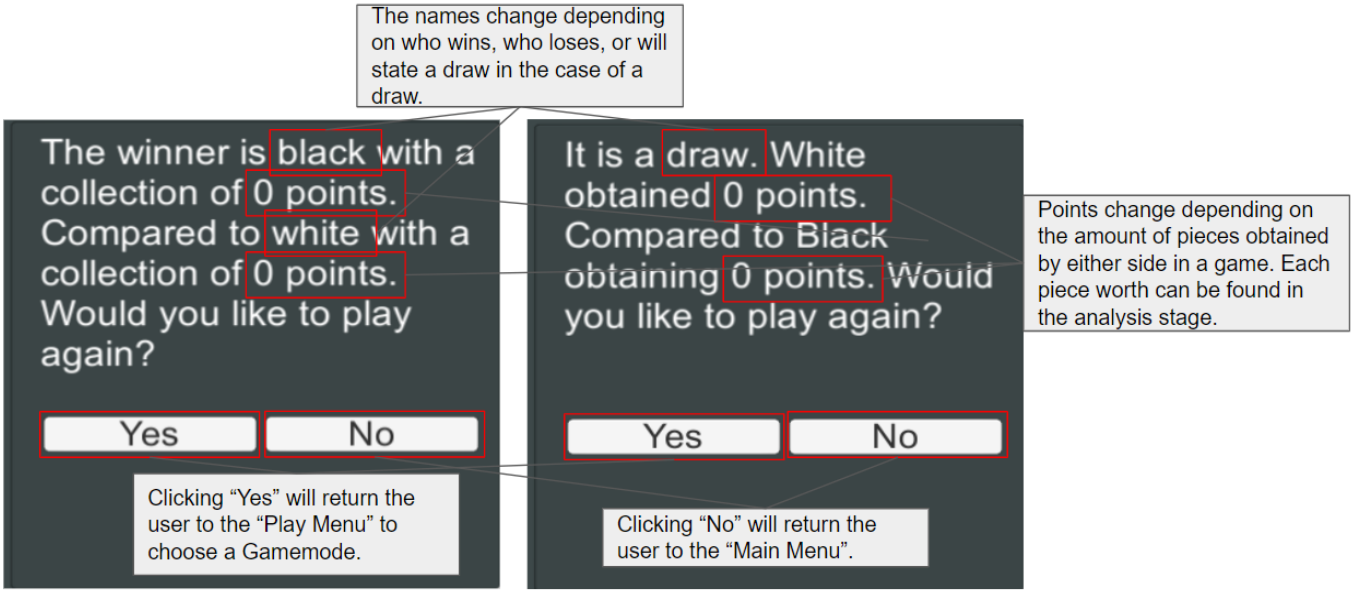
Likewise with the Board, little improvements are needed to this section as each button works perfectly as needed. However, the UI could be improved in later stages to include images of the promoting pieces, such that to newer players, they can more easily recognise the pieces and names that each button corresponds to.

Resign / Draw Offer Menu:



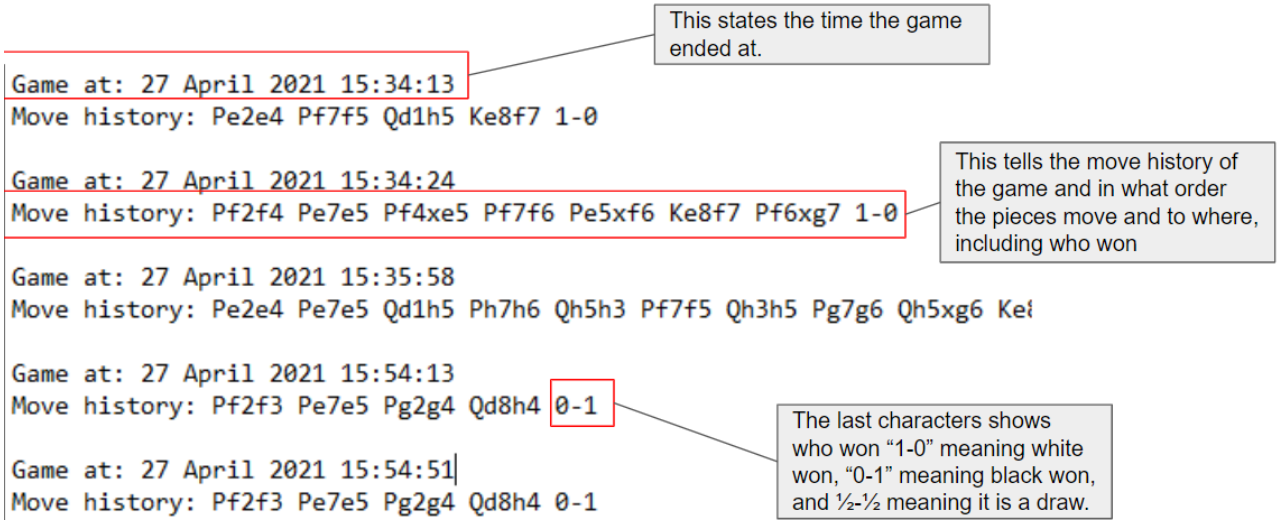
The “Resign” and “Draw” Menus currently work as needed. Little to no improvements are needed as despite it not having the most appealing UI, it is not essential to change in later stages compared to other UI.

Winner / Draw Menu:



Similarly as before, no improvements are needed / essential to the project. All options work as needed.

Move History file (located in the same folder as history.txt)



Lastly, the “History” text file works as intended, however, we could improve it. By including in the program a Menu which reads the text file, we can either recreate a game from a

position in that history, or have an AI play from that position. This would more easily help newer players to know how to play a better game and analyse their moves.

Conclusion:
At its current state, the essential parts of the program work as intended and as proposed. This includes the rules of the game, Main Menu, How to Play Menu and Play Menu UI. However, there are still many improvements needed to be made to the Online Menu, as a significant portion of that section has yet to be completed as proposed. Furthermore, UI in more essential parts of the program such as the Main Menu should be addressed in later stages due to the current appeal of it. Whilst overall the project has kept to a simplistic look, which was an important part of the design, there should still be some improvements made to the visuals of the UI to make the project more enticing.

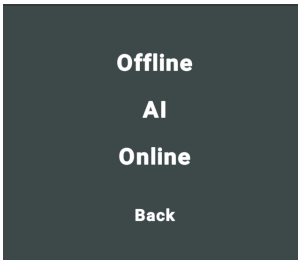
The Evaluation

Introduction:

The project can be broken down into sections, each which can be evaluated separately as they do not contribute to the success of their section. This includes: the UI, the chess rules, the AI, login system as well as feedback from clients. As each section can be allocated each corresponding success criteria, we can accurately scale the success of each section, as well as the whole project, by comparing the proposed solution to the final project.

UI:

The UI currently is in a state where each test has been passed and works without failure. As we can see in 'Objective 1' in the Analysis stage, we have achieved the success criteria of the project, where we include each option to the user to select in the main menu.

Success Criteria - Objective 1	Achieved? + Evidence
- Online, AI, Offline selections, each navigating to their corresponding gamemodes	Yes - 
- Main Menu screen	Yes - 

Additionally, more menu options were added such as the "How To Play" menu, which offers guidance to new players on more challenging rules. Furthermore, tests in section 1 have all been passed successfully, as well as the majority passed in section 2. The only failed tests involving the UI were to do with the "Online" section, which will be addressed in that section.

However, there could be many improvements made to the UI. Comparing this to other researched chess projects, we can see that each alternate project included an enticing UI with many images relating to Chess. However, one main objective was to stick to the theme of a basic UI which only includes the necessary information that the user needs. This is what I then primarily focused on such that the UI has a "clean" look to it as to not complicate the UI to either confuse the user, or make it less appealing.

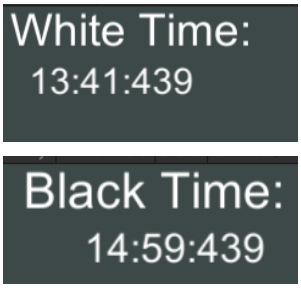
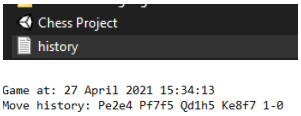
Firstly, the menu is a plain, mono-colour background with mono-colour buttons which directs users to each available section of the project. There are no enticing imagery or advanced UI techniques used to entice the user. Secondly, there is currently no audio, such as background music or simulating the sound of a piece moving once you move a piece. While none of these features were promised, they could have been a much better improvement over the current UI.

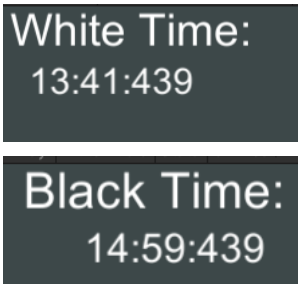
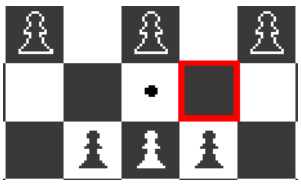
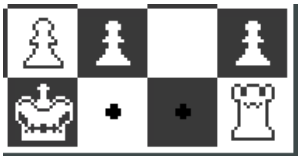
However, the UI for the main chess game itself is quite good, and offers all essential features in a way which is enticing to the user. Improvements were made to better the UI, which were requested by the stakeholders,such as to make it easier for users to understand aspects of the game easier, such as changing the colour of an attacking piece. Little changes do need to be made to the UI of the chess board / screen, as it has fully met the objectives set and no problems were found regarding it.

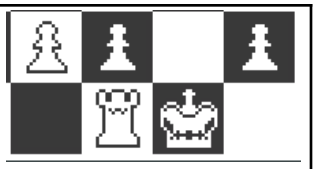
Overall, as all the requirements were met in the success criteria, as well as the UI tests found in section 1 and most of 2, the UI has reached a stage which is acceptable. There are still however, many improvements that could be made to the UI, but due to time limitations, were not met. In later development stages, the UI should be addressed, as many simple changes could be made to better the UI. However, at its current stage, the UI has met the objectives and requirements set, which therefore I believe, this stage has fully met.

Chess Rules:
As this section covers the main premise of the project, it was important to get the tests and success criteria which fall under this category passed. The current state of the chess game covers all the noted success criteria. All the chess rules have been included in the game such as: basic chess moves, more complicated chess moves such as en passant, check, checkmate, stalemate as well as more complicated chess rules such as threefold repetition.

Success Criteria - Objective 2	Achieved? + Evidence
- A GUI that allows users to move pieces around the board	Yes - 
- Rules which correspond to the correct pieces such as movement and taking	Yes - (Extensive evidence found in testing / development stage)
- Having a check for either check, checkmate or stalemate	Yes - (Extensive evidence found in

	testing / development stage)
- Implement more GUI which shows the players current score (collected pieces) as well as timers.	Yes - 
- The whole history of moves will be written into a file	Yes - 

Success Criteria - Objective 3	**Achieved? + Evidence**
- A timer which counts down and alternate per person	Yes - 
- Implement 50 move rule (when 50 moves are done without a piece being taken or a pawn moving)	Yes - (Extensive evidence found in testing / development stage)
- En passant (a pawn can take another pawn if it has just moved 2 places forward and lands next to it)	Yes - 
- Castling (neither the king or castle has moved and all pieces between have moved)	Yes - 

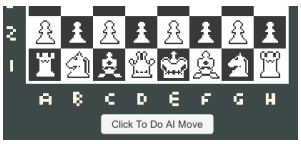
	
- Threefold repetition (the same position has been repeated 3 times)	Yes - (Extensive evidence found in testing / development stage)

There are still however some improvements which could be made, but are minimal. Currently, there is no way to rotate the board such that black is on the lower side of the board. This, however, is not essential, as both players will be playing on the same device, so rotating the board each move may confuse the players so keeping it simple still delivers a good and fair game.

As each criteria and test has been implemented and validated, as well as additional features added, the current stage of the main chess program is at a perfect state. There are little to no improvements needed to be made, only ones which do not affect the core gameplay itself. In later development stages, there should be minimal changes to this section, only ones which include minor changes that unless are important, should not be essential to include. Therefore, I believe, this stage has been fully completed and met.

AI:

Currently, the AI has been implemented and has successfully passed its tests, and the success criteria, such that the AI is aware how to play and does have some logic to it, the AI is at a state where it can definitely be improved on.

Success Criteria - Objective 4	**Achieved? + Evidence**
- Using either a API or prior knowledge to allow the AI to make moves which are legal as well as make logical sense.	Yes - (Evidence found in testing / development stage)
- Implementing this with a GUI to allow a user to play against an AI	Yes - 

Whilst, the AI has some form of logical understanding of how good a move is relative to either the positional gain it gets, or the pieces it takes, it has no understanding of depth.

Whilst an attempt to implement the Minimax algorithm, which gives depth to the AI, there were many errors met and whilst it came close, due to time limitations, it was changed for an easier method which did not include depth.

Slight improvements were made to compensate for this, such as simulating a depth of one using attack positions. This allowed the AI to have some form of understanding if a piece or position is being protected, so it could determine whether or not to do that specific move.

Many improvements could have also been made, one major one being a dictionary of openings for the AI. This would allow the AI to have a record of all openings to follow, to gain a greater position. Currently, the AI does the same opening each time, which could allow the user to play the best known moves opening against it, gaining a much better chance of winning. Secondly, the AI should have a better idea how to checkmate. In most endgame situations, the AI repeats the same moves which even in a winning position, will always end in a draw. This is due to the fact that currently the AI has no knowledge of how check or checkmate works. Luckily, this can easily be added due to the project including the availability of knowing whether a piece is currently being defended or attacked. Lastly, adding multiple difficulties that the user can select could allow the program to reach a wider audience of players with differing rankings.

Whilst the AI does work and does have logic, there are currently many things that could be done to improve it, as many new players will likely beat, or draw with the AI. However, due to the main target audience of this project being newer players, this is not a large issue. Furthermore, as all success criteria has been met, and all tests have been passed, In later stages of development, many of the listed improvements are not essential to the final product. Therefore, I believe the AI has met the standards and requirements set in the proposal stage.

Online:

A large portion of this section has yet to be completed, which due to time constraints, were changed to compensate and complete a much easier route. This included scrapping the Online database in place for a local one. This meant that a large portion of the success criteria, as well as tests, have yet to be met. Many of which were partially completed, however, still have a large amount of essential improvements needed to be made.

Success Criteria - Objective 5	Achieved? + Evidence
- GUI which allows user to enter username/email and password	Yes - 
- Database which stores encrypted version of the password	No
- User redirected to correct GUI when they pass authentication	Yes - 
- Pop up to say when the user has entered incorrect details	Yes - 
- Options to change account passwords / details when a username is forgotten (using questions)	No

Success Criteria - Objective 6	Achieved? + Evidence

- When a user has the correct credentials, they will be able to choose to see their account information. This includes the most recent game's history, wins, loss, draws etc.	No
- Manipulate data to add a ranking system	No

Currently, many significant changes should be made to the current "Online" section. Whilst this section is minimal, and has little effect on the overall project, as it is a proposed and promised feature, there are still many improvements which should be made.

Firstly, there is no recorded user information on the current project, meaning no data is stored either locally or online. Currently, we do have an input for signing in / creating an account, however, the data is not stored onto any permanent storage device. Furthermore, whilst the application is aware that the current game is playing in an "Online" environment, there is currently no update to user information in the case of a win or draw. This also means, there is currently no way to change account information and that since there is no data, there is no encrypted / hashed version of this data stored anywhere.

Next, in objective 6, we can see there is currently no menu which allows users to access their data. Currently, this is due to the fact that data is currently not even being stored at the moment.

Currently, the state of the Online section is partially completed, however, the more sophisticated sections of the application are still yet to be met. This is due to time limitations, which if later stages of development were available, should be addressed as a main issue.

Clients:

Client	Feedback
Oscar Butler	The project is currently at a good stage to release to users. It has a sleek, simplistic design which I enjoy. Furthermore, I believe that the rules in the How To Play menu are very useful to newer players and are described well, however, could be improved with images. However, currently, the Online menu has many improvements needed to be fixed or changed, as it has yet to be finished. The game rules currently are all realistic to a proper game of chess, and work well. I find it very useful that the game history is recorded to a file such that I can analyse my games much easier. However, I did find an issue where if you attack a piece, while also trying to en passant, the piece can no longer be attacked, this should be fixed as soon as possible, although, seeing how enpassant is a move that is unlikely to be played, and I've found only works in very specific circumstances, it shouldn't be a huge issue. The AI currently is quite easy, but should be semi-challenging to more beginner level

	players. However, it is quite frustrating that you have to click the button each time, and I would prefer it if the AI just moved after a fixed amount of time each time. In conclusion, the project is in a great state to release to users, however, the Online section should be addressed as soon as possible in later development stages.
Jake Elliot	The Main Menu UI currently is quite bland and lacks pictures or images. Furthermore, I found an issue where the resolution did not scale with my computer, making it not as appealing. However, I do think the board UI is very nice and all pieces are clearly shown on the board. There should be little changes made to the board as I believe it is perfect as it is. The game itself currently does implement many rules which I did find confusing to learn at first which made the How To Play menu very useful. I wish it could have described it in a more visual way however, it described them good enough for me to understand. I do wish however that you could drag pieces across the board instead of clicking, since most websites I use usually have this as default. Lastly, the history.txt file was useful, however, as I am not really familiar with how to play well, I feel as this will mainly benefit higher ranked players who already can analyse their games. The AI currently is quite challenging and took me a few attempts to beat it, though I found that it wasn't too familiar with checkmating, and was only able to do it once by pure luck I believe. There were many times it could, have yet didn't. Also, having to click the button each time is kind of annoying, but I did get used to it in the end. The project has gone well and there are not many things to add that I think that are essential to the program to drastically change it. However, the Online section still should be improved, as it doesn't seem completed yet. But the project has seemed to implement almost everything promised.

Response:
Regarding the issue found by Oscar, I fixed the issue which regarded calculating checks and when doing an en passant move. This was because it would calculate the move as a normal attack tile, despite being an en passant attack tile. Therefore, currently, there are no known game breaking glitches in the game.

There are some changes mentioned by the users which could be implemented in later stages, especially ones regarding the UI, which I also agree does need improvements. Furthermore, the most addressed problem was regarding the AI button to allow the piece to move. In later stages of development, it should be replaced where after a user does their turn, the AI will move after a certain amount of time each move. Since it is also unlikely the AI will lose due to lack of time, we can decrease the timer for the AI to have a more equal time.

In conclusion, I believe that due to the response by both the clients, the project is in a good state to release. This is because, as both users have a wide range of skill gaps and separate needs, they both the project is in a releasable state. This means that for most users, as both stakeholders represent many different users, we should have a large number of users who will also find this application useful.

Conclusion:
At the current state of the project, there are still many small improvements which could be made to improve small aspects of the project. However, these are not essential and do not drastically improve the current and proposed project idea. Given more time, many of these problems could easily be fixed in later stages of development, and should prioritise on the Online section.

Furthermore, there currently are no aspects of the code which need to be maintained over a long period of time, such as a database for the project to run on. This was because, due to time constraints, the database of user information was changed to be stored on the clients local machine. No parts of the code also rely on external APIs. This means even if there are no improvements made to the project over a significant period of time, the project will work perfectly as intended.

However, as previously explained, it was shown that the AI, UI and Gamerules section have fully met their success criteria and tests, with the Online section being partially completed. Furthermore, the stakeholders both agree the project is releasable in its current state. Therefore, due the extensive evidence, I believe the project has reached a satisfactory stage as users are able to access a working program which has the essential functions that have been proposed. Whilst there are still sections to improve on, the main focus of the program was to create a fully functional chess game, which I believe has been achieved to a satisfactory level.

5. Project Appendices

Main.cs

```
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using UnityEngine.UI;
using System.IO; //read write files
using System; //Date Time

//This class stores the information of the board (where pieces are) as well as is the main class which instantiates all
//the pieces. It also has functions for ending the game as well and validation of data.

public class Main : MonoBehaviour
{
    public GameObject ChessPieces; //used to reference ChessPiece prefab

    //Creating GameObject arrays
    //stores the locations of the game objects on an 8x8 board on a 2D array (xPos,yPos)
    public GameObject[,] boardPositions = new GameObject[8, 8];
    //used to store the current instantiated pieces
    public GameObject[] whitePieces = new GameObject[16];
    public GameObject[] blackPieces = new GameObject[16];
    //used to store the points gained from each
    public int blackPoints;
    public int whitePoints;
    //stores current amount of pieces on board
    public int blackPiecesLength = 16;
    public int whitePiecesLength = 16;
    //used to store the current player and if the game is over and if playing against AI
    public string currentPlayer = "white";
    public bool canMove = true;

    public bool againstAI = false;
    public bool isOnlineGame = false;

    //stores attack positions of each colour
    public bool[,] whiteAttackBoard = new bool[8, 8];
    public bool[,] blackAttackBoard = new bool[8, 8];
```

```
    //stores values for end game:
    public bool gameOver = false;

    //used to count for 50 move rule
    public float moves = 0.0f;

    //used to store moves
    public List<string> moveHistory = new List<string>();

    //used for threefold repetition
    public int[,] threeFoldHistory = new int[101,64];
    public int historyLength = 0;

    public void startGame()
    {
        //Makes all UI not needed invisible
        GameObject.FindGameObjectWithTag("PlayAgainUI").transform.localScale = new Vector3(0, 0, 0);
        GameObject.FindGameObjectWithTag("PromoteMenu").transform.localScale = new Vector3(0, 0, 0);
        //this assigns each piece and places them into the associated array
        whitePieces = new GameObject[] {
            createPiece("WR",0,0), createPiece("WKn",1,0), createPiece("WB",2,0), createPiece("WQ",3,0), createPiece("WKi",4,0),
            createPiece("WB",5,0), createPiece("WKn",6,0), createPiece("WR",7,0),
            createPiece("WP",0,1), createPiece("WP",1,1), createPiece("WP",2,1), createPiece("WP",3,1), createPiece("WP",4,1),
            createPiece("WP",5,1), createPiece("WP",6,1), createPiece("WP",7,1) };
        blackPieces = new GameObject[] {
            createPiece("BR", 0, 7), createPiece("BKn", 1, 7), createPiece("BB", 2, 7), createPiece("BQ", 3, 7), createPiece("BKi", 4,
7),
            createPiece("BB", 5, 7), createPiece("BKn", 6, 7), createPiece("BR", 7, 7),
            createPiece("BP", 0, 6), createPiece("BP", 1, 6), createPiece("BP", 2, 6), createPiece("BP", 3, 6), createPiece("BP", 4,
6),
            createPiece("BP", 5, 6), createPiece("BP", 6, 6), createPiece("BP", 7, 6) };

        for (int currentPiece = 0; currentPiece < 16; currentPiece++)
        {
            assignBoardPos(whitePieces[currentPiece]);
            assignBoardPos(blackPieces[currentPiece]);
        }

        //gets the attack positions for each side
        setupAttackBoard();

        //Starts white timer
        GameObject whiteTimer = GameObject.FindGameObjectWithTag("WhiteTimer");
        whiteTimer.GetComponent<Timer>().cyclePause();

        if (!againstAI)
```

```
    {
        //If you are not playing against an AI, the AI button is not needed
        GameObject.FindGameObjectWithTag("AIButton").transform.localScale = new Vector3(0, 0, 0);
    }
    convertBoardToInt();
}

public GameObject createPiece(string name, int xPos, int yPos)
{   //Creates a piece, assigning the name, xpos and ypos to it
    GameObject currentGO = Instantiate(ChessPieces, new Vector3(0, 0, -1), Quaternion.identity);
    Pieces cp = currentGO.GetComponent<Pieces>();
    cp.name = name;
    cp.xPos = xPos;
    cp.yPos = yPos;
    cp.assignPiece();
    return currentGO;
}
public void assignBoardPos(GameObject currentPiece)
{
    //gets xpos and ypos of the piece
    Pieces cp = currentPiece.GetComponent<Pieces>();
    boardPositions[cp.xPos, cp.yPos] = currentPiece;
}

public bool validBoardPos(int x, int y)
{
    //used to see if the positions which are used are valid in the board (must be between 0-7)
    if (x < 0 || y < 0 || x > 7 || y > 7)
    {
        return false;
    }
    return true;
}

public void setPiece(int x, int y, string colour, int indexVal, string pieceName, bool isSelecting)
{
    //used in promoting the pawn once it reaches the other side, it creates a queen at the given position

    if (colour == "B" && againstAI)
    { //if it is the AI, it will automatically choose queen
        Destroy(blackPieces[indexVal]);
        blackPieces[indexVal] = createPiece("BQ", x, y);
        assignBoardPos(blackPieces[indexVal]);
    }
    else
    {
        if (!isSelecting)
```

```
        { //makes it so you cannot move until you choose a new piece
            canMove = false;
            //maximise menu
            GameObject.FindGameObjectWithTag("PromoteMenu").transform.localScale = new Vector3(1, 1, 1);
        }
        else
        {
            string piece = "";
            piece += colour;
            piece += pieceName;
            if (colour == "B")
            { //selecting black piece and assigns to board
                blackPieces[indexVal] = createPiece(piece, x, y);
                assignBoardPos(blackPieces[indexVal]);
            }
            else
            { //selecting white piece and assigns to board
                whitePieces[indexVal] = createPiece(piece, x, y);
                assignBoardPos(whitePieces[indexVal]);
            }
            //allows user to move
            canMove = true;
        }
    }
    setupAttackBoard();
}

public void changeCurrentPlayer()
{
    //used to change the current player
    switch (currentPlayer)
    {
        case ("white"):
            currentPlayer = "black";
            break;
        case ("black"):
            currentPlayer = "white";
            break;
    }
    //rotates timers
    GameObject whiteTimer = GameObject.FindGameObjectWithTag("WhiteTimer");
    whiteTimer.GetComponent<Timer>().cyclePause();
    GameObject blackTimer = GameObject.FindGameObjectWithTag("BlackTimer");
    blackTimer.GetComponent<Timer>().cyclePause();
}

public void gameEnd(string takePiece)
```

```
{
    stopTimer();
    //called once the game ends and assigns values for the winner text
    gameOver = true;
    int winnerPoints = 0;
    string Winner = "white";
    int loserPoints = 0;
    string Loser = "black";
    switch (takePiece)
    {
        case ("WKi"): //black wins
        case ("white"):
            Winner = "black";
            Loser = "white";
            winnerPoints = blackPoints;
            loserPoints = whitePoints;
            moveHistory.Add("0-1");
            break;
        case ("BKi"): //white wins
        case ("black"):
            moveHistory.Add("1-0");
            winnerPoints = whitePoints;
            loserPoints = blackPoints;
            break;
    }
    writeHistoryToFile();
    //Finds certain gameobjects and changes the properties of them either disabling them, making them
    //visible or changing the text
    GameObject.FindGameObjectWithTag("ResignButton").SetActive(false);
    GameObject.FindGameObjectWithTag("DrawButton").SetActive(false);
    GameObject.FindGameObjectWithTag("PlayAgainUI").transform.localScale = new Vector3(1, 1, 1);
    GameObject.FindGameObjectWithTag("WinnerText").GetComponent<Text>().text =
        "The winner is " + Winner + " with a collection of " + winnerPoints.ToString() + " points. Compared to " + Loser +
        " with a collection of " + loserPoints.ToString() + " points. Would you like to play again?";
}

public void drawEnd()
{
    moveHistory.Add("½-½");
    stopTimer();
    writeHistoryToFile();
    //Used when there is a draw. Finds different game objects and assigns properties similar in
    //gameEnd() function
    GameObject.FindGameObjectWithTag("ResignButton").SetActive(false);
    GameObject.FindGameObjectWithTag("DrawButton").SetActive(false);
    GameObject.FindGameObjectWithTag("PlayAgainUI").transform.localScale = new Vector3(1, 1, 1);
    GameObject.FindGameObjectWithTag("WinnerText").GetComponent<Text>().text =
```

```
        "It is a draw. White obtained " + whitePoints.ToString() + " points. Compared to Black obtaining " +
blackPoints.ToString() + " points. Would you like to play again?";
    }

    public void stopTimer()
    {
        //stops both timers
        GameObject.FindGameObjectWithTag("WhiteTimer").GetComponent<Timer>().stopTimer();
        GameObject.FindGameObjectWithTag("BlackTimer").GetComponent<Timer>().stopTimer();
        GameObject.FindGameObjectWithTag("TimerUI").SetActive(false);
    }

    public void checkDraw()
    {
        //used to check if there is a draw due to insufficient material
        if (whitePiecesLength == 1 && blackPiecesLength == 1)//king vs king
        {
            drawEnd();
        } //everything below includes king
        else if ((whitePiecesLength == 2 && blackPiecesLength == 1) || (whitePiecesLength == 1 && blackPiecesLength == 2))
            //if one player has one piece and the other has two (this is to optimise the amount of checks)
        {
            if (((whitePieces[2] != null || whitePieces[5] != null) && blackPiecesLength == 1) || ((blackPieces[2] != null ||
blackPieces[5] != null) && whitePiecesLength == 1))
                //knight vs king
            {
                drawEnd();
            } else if (((whitePieces[1] != null || whitePieces[6] != null) && blackPiecesLength == 1) || ((blackPieces[1] != null ||
blackPieces[6] != null) && whitePiecesLength == 1))
                //bishop vs king
            {
                drawEnd();
            }
        } else if (whitePiecesLength == 2 && blackPiecesLength == 2)
        { //boths players have two pieces
            if ((whitePieces[0] != null || whitePieces[7] != null) && (blackPieces[0] != null || blackPieces[7] != null))
                // rook vs rook
            {
                drawEnd();
            }
            else if ((whitePieces[2] != null && blackPieces[5] != null) || (whitePieces[5] != null && blackPieces[2] != null))
                // bishop vs bishop (both being on the same square colour)
            {
                drawEnd();
            }
            else if ((whitePieces[0] != null || whitePieces[7] != null) && (blackPieces[2] != null || blackPieces[1] != null ||
blackPieces[5] != null || blackPieces[6] != null))
```

```
            //rook vs bishop
        {
            drawEnd();
        }
        else if ((blackPieces[0] != null || blackPieces[7] != null) && (whitePieces[2] != null || whitePieces[1] != null ||
whitePieces[5] != null || whitePieces[6] != null))
            // rook vs knight
        {
            drawEnd();
        }
    } else if ((whitePiecesLength == 3 && blackPiecesLength ==1) || (blackPiecesLength == 3 && whitePiecesLength ==
1))
    { //three pieces vs one
        if ((whitePieces[1] != null && whitePieces != null && blackPiecesLength == 1) || (blackPieces[1] != null &&
blackPieces[6] != null && whitePiecesLength == 1))
            //two knights vs king
        {
            drawEnd();
        }

    }
  }

  public void setupAttackBoard()
  {
    //Used to create the attack board which stores the positions of each sides attacking positions
    for (int xBoard = 0; xBoard < 8; xBoard++)
    {
        for (int yBoard = 0; yBoard < 8; yBoard++)
        { //resets the board by setting each position to false
            whiteAttackBoard[xBoard, yBoard] = false;
            blackAttackBoard[xBoard, yBoard] = false;
        }
    }

    for (int currentPos = 0; currentPos < 16; currentPos++)
    {
        if (whitePieces[currentPos] != null) //null error stop
        { //gets all possible attacking positions of white side
            whitePieces[currentPos].GetComponent<Pieces>().possibleMoves(false);
        }
        if (blackPieces[currentPos] != null)//null error stop
        { //gets all possible attacking positions of black side
            blackPieces[currentPos].GetComponent<Pieces>().possibleMoves(false);
        }

    }
```

```
        inCheck();
    }

    public bool inCheck()
    {
        //Returns if in check or not, setting the king in check if it is
        if (currentPlayer == "white") //if the current player is white
        {
            if (blackAttackBoard[whitePieces[4].GetComponent<Pieces>().xPos,
whitePieces[4].GetComponent<Pieces>().yPos])
            { //if king is being attacked, set check to true
                whitePieces[4].GetComponent<Pieces>().inCheck = true;
                return true;
            }
            else
            {
                whitePieces[4].GetComponent<Pieces>().inCheck = false;
            }
        }
        else
        {//if the current player is black
            if (whiteAttackBoard[blackPieces[4].GetComponent<Pieces>().xPos,
blackPieces[4].GetComponent<Pieces>().yPos])
            {//if king is being attacked, set check to true
                blackPieces[4].GetComponent<Pieces>().inCheck = true;
                return true;
            }
            else
            {
                blackPieces[4].GetComponent<Pieces>().inCheck = false;
            }
        }
        return false;
    }

    public void destroyTiles()
    { //finds all move plates that are tagged "MovePlate"
        GameObject[] movePlates = GameObject.FindGameObjectsWithTag("MovePlate");
        for (int i = 0; i < movePlates.Length; i++)
        { //destroys each one
            Destroy(movePlates[i]);
        }
    }


    public void writeHistoryToFile()
```

```
  {
    //This will write the list "moveHistory" to a file
    DateTime now = DateTime.Now; //gets current time
    string history = "\n\nGame at: "+ now.ToString("F") + "\nMove history:";
    for(int i = 0; i < moveHistory.Count(); i++)
    { //gets all moves and appends them to a string format
      history += " " + moveHistory[i];
    }
    string path = Environment.CurrentDirectory + "\\history.txt";
    if (!File.Exists(path)) //if the file does not exist
    {
      // Create a file to write to.
      string createText = Environment.NewLine;
      File.WriteAllText(path, createText);
    } //Writes the move history to file
    File.AppendAllText(path, history);
  }

  public bool convertBoardToInt()
  { //converts the current board positions to an integer format
    int[] intBoard = new int[64];
    for(int i =0; i < whitePieces.Length; i++)
    { //to distinguish from each piece, the array position is used, and if it is black, it is +17 to distinguish sides
      if(whitePieces[i] != null)
      {
        intBoard[whitePieces[i].GetComponent<Pieces>().xPos + whitePieces[i].GetComponent<Pieces>().yPos *8] = i +
1;
      } if (blackPieces[i] != null)
      {
        intBoard[blackPieces[i].GetComponent<Pieces>().xPos + blackPieces[i].GetComponent<Pieces>().yPos * 8] = i +
17;
      }
    }
    return checkRepition(intBoard);
  }

  public bool checkRepition(int[] boardPositions)
  {//used for threefold repetition
    int checkSum = 0;
    for (int currentCheck = 0; currentCheck < historyLength; currentCheck++)
    { //iterates through all stored moves
      int checkCurrent = 0;
      for(int valueCheck = 0; valueCheck < boardPositions.Length; valueCheck++)
      {
        if (threeFoldHistory[currentCheck,valueCheck] == boardPositions[valueCheck])
        { //if a square is equal to the same square on a seperate board
          checkCurrent++;
```

```
            }
        }
        if (checkCurrent == 64)
        { //if all 64 spots are the same
            checkSum++;
            if (checkSum == 2)
            { //if the position is repeated three times, end the game
                return true;
            }
        }
    }
    for (int i = 0; i < 64; i++)
    {//adds the move to history
        threeFoldHistory[historyLength, i] = boardPositions[i];
    }
    historyLength++;
    return false;
  }
}
```

Pieces.cs

```
using System;
using UnityEngine;

//This class stores the information of each piece which is instantiated as well as the possible moves of each piece

public class Pieces : MonoBehaviour
{

    // References
    public GameObject controller; //later assigned in "assignPiece()"
    public GameObject moveTiles; //creates a tile to allow pieces to move to (associated with MoveTiles class)
    //Creating the references for sprites, each assigned to a piece (used in assignPiece())
    public Sprite BB, BKi, BKn, BP, BQ, BR, WB, WKi, WKn, WP, WQ, WR;

    //Creating Board positions (x and y), both assigned -1 as they are not on the board yet. They range between (0-7) created an
    //8x8 board in total. They are assigned later on in the "MainGame" class.
    public int xPos = -1;
    public int yPos = -1;

    //The colour of the piece
    public string playerColour;
    //Private as I do not want code to be able to change the pieceWorth of pieces after they are declared
    private int pieceWorth = 0;
    //If the pawn has moved or not (while it is quite necessary for it to be a variable stored in non-pawn pieces, this makes it much easier
    //and does not affect any other pieces anyways
    public bool hasMoved = false;

    public bool inCheck = false;

    //Used to allow pawns to en passant
    public bool enPassantAttack = false;
    public int enPassantXPos = -1;
    public int enPassantYPos = -1;

    public void assignPiece()
    {
```

```
        //the GameObject "ChessPiecesController" is assigned a tag "GameController", this makes it
        //so you do find this object with functions
        controller = GameObject.FindGameObjectWithTag("GameController");
        //when an instance of a class is created, the name is parsed and with this can be used with a switch case to assign
the
        //corresponding sprite value (these are located in the file "Sprites''. This refers to this instance of the class and the
        //name is the name parsed in the MainGame class
        switch (name)
        {
            //Gets the component "SpriteRender" and has a variable called sprite which is used to assign the sprite value; we
            //then replace the sprite to the corresponding sprite value found in folder "Sprites". We also assign the
            //values for each piece (King is 0 as it technically has an infinite worth as once you get it you win anyway)
            //aswell as the colour of the piece
            case "BB":
                playerColour = "B";
                pieceWorth = 3;
                GetComponent<SpriteRenderer>().sprite = BB;
                break;
            case "BKi":
                playerColour = "B";
                pieceWorth = 90;
                GetComponent<SpriteRenderer>().sprite = BKi;
                break;
            case "BKn":
                playerColour = "B";
                pieceWorth = 3;
                GetComponent<SpriteRenderer>().sprite = BKn;
                break;
            case "BP":
                playerColour = "B";
                pieceWorth = 1;
                GetComponent<SpriteRenderer>().sprite = BP;
                break;
            case "BQ":
                playerColour = "B";
                pieceWorth = 9;
                GetComponent<SpriteRenderer>().sprite = BQ;
                break;
            case "BR":
                playerColour = "B";
                pieceWorth = 5;
                GetComponent<SpriteRenderer>().sprite = BR;
                break;
            case "WB":
                playerColour = "W";
                pieceWorth = 3;
                GetComponent<SpriteRenderer>().sprite = WB;
```

```
            break;
        case "WKi":
            playerColour = "W";
            pieceWorth = 90;
            GetComponent<SpriteRenderer>().sprite = WKi;
            break;
        case "WKn":
            playerColour = "W";
            pieceWorth = 3;
            GetComponent<SpriteRenderer>().sprite = WKn;
            break;
        case "WP":
            playerColour = "W";
            pieceWorth = 1;
            GetComponent<SpriteRenderer>().sprite = WP;
            break;
        case "WQ":
            playerColour = "W";
            pieceWorth = 9;
            GetComponent<SpriteRenderer>().sprite = WQ;
            break;
        case "WR":
            playerColour = "W";
            pieceWorth = 5;
            GetComponent<SpriteRenderer>().sprite = WR;
            break;
    }
    setCoordPos();
}

public void setCoordPos()
//This function transforms the position of the GameObjects, where we must translate the positions which
//are on the board (0-7) to coordinates on the screen so they are placed correctly.
{
    transform.position = new Vector3((xPos * 0.66f) - 2.3f, (yPos * 0.66f) - 2.3f);
}

public int getPieceWorth()
{
    return pieceWorth;
}
public void possibleMoves(bool isPlace)
{
    //isPlace true => moving piece
    //isPlace false => used to update attack positions
    //Gets the possible moves for a piece
    switch (name)
```

```
    {
        case "WB":
        case "BB":
            bishopMove(isPlace);
            break;
        case "WKi":
        case "BKi":
            kingMove(isPlace);
            break;
        case "WKn":
        case "BKn":
            knightMove(isPlace);
            break;
        case "WP":
            if (!hasMoved)
            {
                pawnNotMove(0, 2, isPlace);
            }
            pawnMove(xPos, yPos + 1, isPlace);
            break;
        case "BP":
            if (!hasMoved)
            {
                pawnNotMove(0, -2, isPlace);
            }
            pawnMove(xPos, yPos - 1, isPlace);
            break;
        case "WQ":
        case "BQ":
            queenMove(isPlace);
            break;
        case "WR":
        case "BR":
            rookMove(isPlace);
            break;
    }
}

public void bishopMove(bool isPlace)
{
    straightLineMove(1, 1, isPlace);
    straightLineMove(-1, -1, isPlace);
    straightLineMove(-1, 1, isPlace);
    straightLineMove(1, -1, isPlace);
}

public void knightMove(bool isPlace)
```

```
    {
        inputMove(xPos + 2, yPos + 1, isPlace);
        inputMove(xPos + 2, yPos - 1, isPlace);
        inputMove(xPos - 2, yPos + 1, isPlace);
        inputMove(xPos - 2, yPos - 1, isPlace);
        inputMove(xPos + 1, yPos + 2, isPlace);
        inputMove(xPos - 1, yPos + 2, isPlace);
        inputMove(xPos + 1, yPos - 2, isPlace);
        inputMove(xPos - 1, yPos - 2, isPlace);

    }

    public void kingMove(bool isPlace)
    {
        inputMove(xPos + 1, yPos + 1, isPlace);
        inputMove(xPos, yPos + 1, isPlace);
        inputMove(xPos + 1, yPos - 1, isPlace);
        inputMove(xPos + 1, yPos, isPlace);
        inputMove(xPos - 1, yPos, isPlace);
        inputMove(xPos - 1, yPos + 1, isPlace);
        inputMove(xPos, yPos - 1, isPlace);
        inputMove(xPos - 1, yPos - 1, isPlace);

        if (!inCheck && isPlace)
        {

            Main board = controller.GetComponent<Main>();
            //check for castle
            if (playerColour == "B" && !hasMoved)
            {
                if (board.boardPositions[7, 7] != null)
                {//kingside castle
                    if (!board.boardPositions[7, 7].GetComponent<Pieces>().hasMoved && board.boardPositions[6, 7] == null &&
board.boardPositions[5, 7] == null
                        && board.whiteAttackBoard[5,7] != true)
                    {
                        castle(7,7);
                    }
                }
                if (board.boardPositions[0, 7]!= null)
                {//queenside castle
                    if (!board.boardPositions[0, 7].GetComponent<Pieces>().hasMoved && board.boardPositions[1, 7] == null &&
board.boardPositions[2, 7] == null
                        && board.boardPositions[3, 7] == null && board.whiteAttackBoard[3,7] != true)
                    {
                        castle(0,7);
                    }
```

```
            }
        }
        else if (playerColour == "W" && !hasMoved)
        {
            if (board.boardPositions[0, 0] != null)
            {//queenside castle
                if (!board.boardPositions[0, 0].GetComponent<Pieces>().hasMoved && board.boardPositions[1, 0] == null &&
board.boardPositions[2, 0] == null
                    && board.boardPositions[3, 0] == null && board.blackAttackBoard[3, 0] != true)
                {
                    castle(0,0);
                }
            }
            if (board.boardPositions[7, 0] != null)
            {//kingside castle
                if (!board.boardPositions[7, 0].GetComponent<Pieces>().hasMoved && board.boardPositions[6, 0] == null &&
board.boardPositions[5, 0] == null
                    && board.blackAttackBoard[5,0] != true)
                {
                    castle(7,0);
                }
            }
        }
        //Seeing that it cant move anywhere (stalemate)
        GameObject[] movePlates = GameObject.FindGameObjectsWithTag("MovePlate");
        if (movePlates.Length == 0 && isPlace)
        {//king have no moves
            if (playerColour == "W")
            {//iterate through all pieces, getting their moves
                for (int currentPiece = 0; currentPiece < 16; currentPiece++)
                {
                    if (board.whitePieces[currentPiece] != null &&
board.whitePieces[currentPiece].GetComponent<Pieces>().name != "WKi")
                    {
                        board.whitePieces[currentPiece].GetComponent<Pieces>().possibleMoves(true);
                    }
                }
                //get all the new moves
                GameObject[] newMovePlates = GameObject.FindGameObjectsWithTag("MovePlate");
                if (newMovePlates.Length == 0)
                {//if there are no legal moves, its stalemate, draw the game
                    DestroyTiles();
                    board.drawEnd();
                }
            }
            else
            {
```

```
            for (int currentPiece = 0; currentPiece < 16; currentPiece++)
            {
                if (board.blackPieces[currentPiece] != null &&
board.blackPieces[currentPiece].GetComponent<Pieces>().name != "BKi")
                {
                    board.blackPieces[currentPiece].GetComponent<Pieces>().possibleMoves(true);
                }
            }
            GameObject[] newMovePlates = GameObject.FindGameObjectsWithTag("MovePlate");
            if (newMovePlates.Length == 0)
            {
                DestroyTiles();
                board.drawEnd();
            }
        }
        DestroyTiles();
    }
}

else if (inCheck && isPlace)
{
    //used to check for checkmate
    Main board = controller.GetComponent<Main>();

    GameObject[] movePlates = GameObject.FindGameObjectsWithTag("MovePlate");

    if (movePlates.Length == 0)
    {
        if (playerColour == "W")
        {
            for (int currentPiece = 0; currentPiece < 16; currentPiece++)
            {
                if (board.whitePieces[currentPiece] != null &&
board.whitePieces[currentPiece].GetComponent<Pieces>().name != "WKi")
                {
                    board.whitePieces[currentPiece].GetComponent<Pieces>().possibleMoves(true);
                }
            }
            GameObject[] newMovePlates = GameObject.FindGameObjectsWithTag("MovePlate");
            if (newMovePlates.Length == 0)
            {
                DestroyTiles();
                board.gameEnd(name);
            }
        }
        else
        {
```

```
                for (int currentPiece = 0; currentPiece < 16; currentPiece++)
                {
                    if (board.blackPieces[currentPiece] != null &&
board.blackPieces[currentPiece].GetComponent<Pieces>().name != "BKi")
                    {
                        board.blackPieces[currentPiece].GetComponent<Pieces>().possibleMoves(true);
                    }
                }
                GameObject[] newMovePlates = GameObject.FindGameObjectsWithTag("MovePlate");
                if (newMovePlates.Length == 0)
                {
                    DestroyTiles();
                    board.gameEnd(name);
                }
            }
            DestroyTiles();
        }
    }
  }

  public void queenMove(bool isPlace)
  {
    //Straight Lines
    straightLineMove(1, 0, isPlace);
    straightLineMove(0, 1, isPlace);
    straightLineMove(-1, 0, isPlace);
    straightLineMove(0, -1, isPlace);
    //Diagonal Lines
    straightLineMove(1, 1, isPlace);
    straightLineMove(-1, -1, isPlace);
    straightLineMove(-1, 1, isPlace);
    straightLineMove(1, -1, isPlace);
  }

  public void rookMove(bool isPlace)
  {
    //Straight Lines
    straightLineMove(1, 0, isPlace);
    straightLineMove(0, 1, isPlace);
    straightLineMove(-1, 0, isPlace);
    straightLineMove(0, -1, isPlace);
  }

  public void castle(int newXPos, int newYPos)
  {
    //used to check if the king can castle or not
    Main board = controller.GetComponent<Main>();
```

```
    if (board.boardPositions[newXPos, newYPos].GetComponent<Pieces>().name[1] == 'R' &&
board.boardPositions[newXPos, newYPos].GetComponent<Pieces>().playerColour == playerColour)
    {
        //used to see if the king can castle king side
        if (playerColour == "W" && board.blackAttackBoard[xPos + 2, yPos] != true && newXPos == 7)
        {
            moveCastleTile(xPos + 2, yPos, board.boardPositions[xPos + 3, yPos], -1);
        }
        else if (playerColour == "B" && board.whiteAttackBoard[xPos + 2, yPos] != true && newXPos == 7)
        {
            moveCastleTile(xPos + 2, yPos, board.boardPositions[xPos + 3, yPos], -1);
        }
        //used to see if the king can castle queen side
        if (playerColour == "W" && board.blackAttackBoard[xPos - 2, yPos] != true && newXPos == 0)
        {
            moveCastleTile(xPos - 2, yPos, board.boardPositions[xPos - 4, yPos], 2);
        }
        else if (playerColour == "B" && board.whiteAttackBoard[xPos - 2, yPos] != true && newXPos == 0)
        {
            moveCastleTile(xPos - 2, yPos, board.boardPositions[xPos - 4, yPos], 2);
        }
    }
}

public void pawnMove(int newXPos, int newYPos, bool isPlace)
{
    Main board = controller.GetComponent<Main>();
    if (isPlace)
    {
        if (board.validBoardPos(newXPos, newYPos) && board.boardPositions[newXPos, newYPos] == null)
        {
            passiveMovePawn(newXPos, newYPos, isPlace);
        }
        if (board.validBoardPos(newXPos + 1, newYPos) && board.boardPositions[newXPos + 1, newYPos] != null &&
board.boardPositions[newXPos + 1, newYPos].GetComponent<Pieces>().playerColour != playerColour)
        {
            attackTile(newXPos + 1, newYPos, isPlace);
        }
        if (board.validBoardPos(newXPos - 1, newYPos) && board.boardPositions[newXPos - 1, newYPos] != null &&
board.boardPositions[newXPos - 1, newYPos].GetComponent<Pieces>().playerColour != playerColour)
        {
            attackTile(newXPos - 1, newYPos, isPlace);
        } //used for en passant
        if (enPassantAttack)
        { //if the piece can en passant
            if (board.boardPositions[enPassantXPos,enPassantYPos] == null)
            {//create move tile
```

```
                enPassantMovePawn(enPassantXPos, enPassantYPos);
            }
        }
    }
    else
    {
        if (board.validBoardPos(newXPos + 1, newYPos))
        {
            setAttackBoardPos(newXPos + 1, newYPos);
        }
        if (board.validBoardPos(newXPos - 1, newYPos))
        {
            setAttackBoardPos(newXPos - 1, newYPos);
        }
    }

}

public void pawnNotMove(int changeInX, int changeInY, bool isPlace)
{
    //Used to allow a pawn to move two spaces if the piece has not moved yet
    Main board = controller.GetComponent<Main>();
    if(changeInY == 2)
    {
        if (board.validBoardPos(xPos + changeInX, yPos + changeInY) && board.boardPositions[xPos + changeInX, yPos + changeInY] == null && board.boardPositions[xPos + changeInX, yPos + changeInY - 1] == null)
        {
            passiveMovePawn(xPos + changeInX, yPos + changeInY, isPlace);
        }
    }
    else
    {
        if (board.validBoardPos(xPos + changeInX, yPos + changeInY) && board.boardPositions[xPos + changeInX, yPos + changeInY] == null && board.boardPositions[xPos + changeInX, yPos + changeInY + 1] == null)
        {
            passiveMovePawn(xPos + changeInX, yPos + changeInY, isPlace);
        }
    }

}

public void inputMove(int newXPos, int newYPos, bool isPlace)
{
    //Used to check if a certain location is empty and none other (used for king and knight)
    Main board = controller.GetComponent<Main>();
    if (isPlace)
    {
```

```
        if (board.validBoardPos(newXPos, newYPos))
        {
            if ((name == "WKi" && board.blackAttackBoard[newXPos, newYPos] != true) || (name == "BKi" &&
board.whiteAttackBoard[newXPos, newYPos] != true)
                || (name != "BKi" && name != "WKi"))
            {
                GameObject cp = board.boardPositions[newXPos, newYPos];
                if (cp == null)
                {
                    passiveMove(newXPos, newYPos, isPlace);
                }
                else if (cp.GetComponent<Pieces>().playerColour != playerColour)
                {
                    attackTile(newXPos, newYPos, isPlace);
                }
            }
        }
        else
        {
            if (board.validBoardPos(newXPos, newYPos))
            {
                setAttackBoardPos(newXPos, newYPos);
            }
        }

    }

    public void straightLineMove(int changeInX, int changeInY, bool isPlace)
    {
        int possibleXPos = xPos + changeInX;
        int possibleYPos = yPos + changeInY;
        Main board = controller.GetComponent<Main>();
        //use a while loops as we will create a moveTile every time a space is fre to move until it is not.
        while (board.validBoardPos(possibleXPos, possibleYPos) && board.boardPositions[possibleXPos, possibleYPos] ==
null)
        {
            //creates a move tile
            moveTile(possibleXPos, possibleYPos, isPlace);
            setAttackBoardPos(possibleXPos, possibleYPos);
            possibleXPos += changeInX; possibleYPos += changeInY;
        }

        if (board.validBoardPos(possibleXPos, possibleYPos) && board.boardPositions[possibleXPos,
possibleYPos].GetComponent<Pieces>().playerColour != playerColour)
        {
            //creates an attack tile
```

```
            attackTile(possibleXPos, possibleYPos, isPlace);
            setAttackBoardPos(possibleXPos, possibleYPos);
            if (board.validBoardPos(possibleXPos + changeInX, possibleYPos + changeInY) &&
(board.boardPositions[possibleXPos, possibleYPos].GetComponent<Pieces>().name == "WKi" ||
                board.boardPositions[possibleXPos, possibleYPos].GetComponent<Pieces>().name == "BKi"))
            {
                setAttackBoardPos(possibleXPos + changeInX, possibleYPos + changeInY);
            }
        }
        //This is used to also show it can attack the same piece as the same colour, incase that a piece is taken of same
colour and that you can still take it
        if (board.validBoardPos(possibleXPos, possibleYPos) && !isPlace)
        {
            setAttackBoardPos(possibleXPos, possibleYPos);
        }

    }

    public void passiveMove(int xPos, int yPos, bool isPlace)
    {
        //Used to create a move tile
        Main Board = controller.GetComponent<Main>();
        if(Board.validBoardPos(xPos,yPos) && Board.boardPositions[xPos, yPos] == null)
        {
            moveTile(xPos, yPos, isPlace);
        }
    }

    public void passiveMovePawn(int x, int y, bool isPlace)
    {
        if (isPlace)
        {
            if (!moveOutCheck(x, y))
            {
                //Used to instantiate a move tile
                GameObject mt = Instantiate(moveTiles, new Vector3((x * 0.66f) - 2.3f, (y * 0.66f) - 2.3f, -3.0f),
Quaternion.identity);
                mt.gameObject.tag = "MovePlate";
                MoveTiles mtscript = mt.GetComponent<MoveTiles>();
                mtscript.setCurrentPiece(gameObject);
                mtscript.setBoardPos(x, y);
                int val = Math.Abs(yPos - y);
                if (val == 2)
                {
                    mtscript.enPassant = true;
                }
            }
```

```
    }
  }

  public void enPassantMovePawn(int x, int y)
  {
    if (!moveOutCheck(x, y))
    {
      //Used to instantiate an attack tile
      GameObject at = Instantiate(moveTiles, new Vector3((x * 0.66f) - 2.3f, (y * 0.66f) - 2.3f, -3.0f),
Quaternion.identity);
      at.gameObject.tag = "MovePlate";
      MoveTiles mtscript = at.GetComponent<MoveTiles>();
      mtscript.attackEnPassant = true;
      mtscript.setCurrentPiece(gameObject);
      mtscript.setBoardPos(x, y);
    }
  }

  public void DestroyTiles()
  {
    //Finds all the gameobjects withb the tag "MovePlate" and destroys them
    GameObject[] movePlates = GameObject.FindGameObjectsWithTag("MovePlate");
    for (int i=0; i < movePlates.Length; i++)
    {
      Destroy(movePlates[i]);
    }
  }

  public void moveTile(int x,int y, bool isPlace)
  {
    if (isPlace)
    {
      if (isPlace && !moveOutCheck(x,y))
      {
        //Used to instantiate a move tile
        GameObject mt = Instantiate(moveTiles, new Vector3((x * 0.66f) - 2.3f, (y * 0.66f) - 2.3f, -3.0f),
Quaternion.identity);
        mt.gameObject.tag = "MovePlate";
        MoveTiles mtscript = mt.GetComponent<MoveTiles>();
        mtscript.setCurrentPiece(gameObject);
        mtscript.setBoardPos(x, y);
      }
    }
    else
    {
      setAttackBoardPos(x, y);
    }
```

```
    }

    public void attackTile(int x, int y, bool isPlace)
    {
        if (isPlace)
        {
            if (isPlace && !moveOutCheck(x,y))
            {
                //Used to instantiate an attack tile
                GameObject at = Instantiate(moveTiles, new Vector3((x * 0.66f) - 2.3f, (y * 0.66f) - 2.3f, -3.0f),
Quaternion.identity);
                at.gameObject.tag = "MovePlate";
                MoveTiles mtscript = at.GetComponent<MoveTiles>();
                mtscript.isAttack = true;
                mtscript.setCurrentPiece(gameObject);
                mtscript.setBoardPos(x, y);
            }
        }
        else
        {
            setAttackBoardPos(x, y);
        }
    }

    public void moveCastleTile(int x, int y, GameObject piece, int changeInX)
    {
        //used to instaniate a move tile when the king castles
        GameObject mt = Instantiate(moveTiles, new Vector3((x * 0.66f) - 2.3f, (y * 0.66f) - 2.3f, -3.0f),
Quaternion.identity);
        mt.gameObject.tag = "MovePlate";
        MoveTiles mtscript = mt.GetComponent<MoveTiles>();
        mtscript.setCurrentPiece(gameObject);
        mtscript.setBoardPos(x, y);
        mtscript.isCastle = true;
        mtscript.castlePiece = piece;
        mtscript.castleXPos = x;
        mtscript.castleYPos = y;
        mtscript.changeInX = changeInX;

    }

    public bool checkPromote(bool isPlace)
    {
        if(isPlace)
        {
            int indexVal = 16;
            //used to check if a pawn can be promoted
```

```
        Main board = controller.GetComponent<Main>();
        if (yPos == 7)
        {
            indexVal = Array.IndexOf(board.whitePieces, gameObject);
            board.setPiece(xPos, yPos, "W", indexVal, "default", false);
            return true;
        }
        else if (yPos == 0)
        {
            indexVal = Array.IndexOf(board.blackPieces, gameObject);
            board.setPiece(xPos, yPos, "B", indexVal, "default", false);
            return true;
        }
    }
    return false;

}

public void setAttackBoardPos(int x, int y)
{
    Main board = controller.GetComponent<Main>();
    if(playerColour == "W")
    {
        board.whiteAttackBoard[x, y] = true;
    } else if (playerColour == "B")
    {
        board.blackAttackBoard[x, y] = true;
    }
}

public bool moveOutCheck(int newXPos, int newYPos) //-> true = cannot move, false = possible move
{   //Check if the move is legal / won't cause check on own side
    bool isAttack = false;
    //replicating the piece
    int refXPos = xPos;
    int refYPos = yPos;
    int index = 0;
    GameObject refObj = gameObject;

    Main board = controller.GetComponent<Main>();
    if (name!= "WKi" && name!= "BKi")
    // The kings do not need to use this script
    {
        if (playerColour== "W")
            //If the piece is white
        {
            if (board.boardPositions[newXPos,newYPos] != null)
```

```
    //check if attack move
{
    isAttack = true;
    index = Array.IndexOf(board.blackPieces, board.boardPositions[newXPos, newYPos]);
    //get the piece that it is attacking
    refObj = board.blackPieces[index];
    board.blackPieces[index] = null;
    //removes the piece
    board.boardPositions[newXPos, newYPos] = null;
}
else if (enPassantAttack && board.boardPositions[newXPos, newYPos] == null &&
    board.boardPositions[newXPos,newYPos-1].GetComponent<Pieces>().playerColour != playerColour)
{ //similar script as before but refers to when a piece can enpassent
    isAttack = true;
    index = Array.IndexOf(board.blackPieces, board.boardPositions[newXPos, newYPos - 1]);
    refObj = board.blackPieces[index];
    board.blackPieces[index] = null;
    board.boardPositions[newXPos, newYPos + 1] = null;
}
//Moves the piece to the corresponding location
board.boardPositions[xPos, yPos] = null;
xPos = newXPos;
yPos = newYPos;
board.assignBoardPos(gameObject);
//creates a new attackboard, to check if in check
board.setupAttackBoard();
if (checkInCheck("W"))
{ //resets the board to the previous position
    board.boardPositions[newXPos, newYPos] = null;
    if (isAttack)
    {
        //reassigns old piece to board
        board.blackPieces[index] = refObj;
        board.assignBoardPos(board.blackPieces[index]);
    }
    xPos = refXPos;
    yPos = refYPos;
    board.assignBoardPos(gameObject);
    return true;
}
else
{//resets the board to the previous position
    board.boardPositions[newXPos, newYPos] = null;
    if (isAttack)
    {
        board.blackPieces[index] = refObj;
        board.assignBoardPos(board.blackPieces[index]);
```

```
                //reassigns the place
            }
            xPos = refXPos;
            yPos = refYPos;
            board.assignBoardPos(gameObject);
            return false;
        }
    }
    else if (playerColour == "B")
    {//else for black side
        if (board.boardPositions[newXPos, newYPos] != null)
        {
            isAttack = true;
            index = Array.IndexOf(board.whitePieces, board.boardPositions[newXPos, newYPos]);
            refObj = board.whitePieces[index];
            board.whitePieces[index] = null;
            board.boardPositions[newXPos, newYPos] = null;
        }
        else if (enPassantAttack && board.boardPositions[newXPos, newYPos] == null &&
board.boardPositions[newXPos, newYPos + 1].GetComponent<Pieces>().playerColour != playerColour)
        {
            isAttack = true;
            index = Array.IndexOf(board.whitePieces, board.boardPositions[newXPos, newYPos + 1]);
            refObj = board.whitePieces[index];
            board.whitePieces[index] = null;
            board.boardPositions[newXPos, newYPos + 1] = null;
        }
        board.boardPositions[xPos, yPos] = null;
        xPos = newXPos;
        yPos = newYPos;
        board.assignBoardPos(gameObject);
        board.setupAttackBoard();
        if (checkInCheck("B"))
        {
            board.boardPositions[newXPos, newYPos] = null;
            if (isAttack)
            {
                board.whitePieces[index] = refObj;
                board.assignBoardPos(board.whitePieces[index]);

            }
            xPos = refXPos;
            yPos = refYPos;
            board.assignBoardPos(gameObject);
            return true;
        }
        else
```

```
            {
                board.boardPositions[newXPos, newYPos] = null;
                if (isAttack)
                {
                    board.whitePieces[index] = refObj;
                    board.assignBoardPos(board.whitePieces[index]);

                }
                xPos = refXPos;
                yPos = refYPos;
                board.assignBoardPos(gameObject);
                return false;
            }
        }
        else
        {
            return false;
        }

    }
    else
    {
        return false;
    }
  }

  public bool checkInCheck(string colour)
  {
    Main board = controller.GetComponent<Main>();
    if (colour == "W")
    {//gets position of king in attack board, if true, the king is in check
        if (board.blackAttackBoard[board.whitePieces[4].GetComponent<Pieces>().xPos,
board.whitePieces[4].GetComponent<Pieces>().yPos])
        {
            return true;
        }
    }
    else if (colour == "B")
    {
        if (board.whiteAttackBoard[board.blackPieces[4].GetComponent<Pieces>().xPos,
board.blackPieces[4].GetComponent<Pieces>().yPos])
        {
            return true;
        }
    }
    return false;
  }
```

```
    private void OnMouseUp()
    {
        //When the user clicks it will destroy all tiles that had originally been there (if the player clicked on a piece and it
didn't move), and
        //show all possible moves for the piece that was clicked on this is done by giving the prefab a 2d box collider
        Main board = controller.GetComponent<Main>();
        DestroyTiles();
        board.setupAttackBoard();
        if (!board.gameOver && board.canMove)
        {//if the user can move
            if (!board.againstAI && board.currentPlayer[0].ToString() == playerColour.ToLower())
            {
                possibleMoves(true);
            }
            //if against AI (only white can select moves)
            else if (board.againstAI)
            {
                if (board.currentPlayer == "white" && playerColour == "W")
                {
                    possibleMoves(true);
                }
            }
        }
    }

}
```

MoveTile.cs

```
using UnityEngine;
public class MoveTiles : MonoBehaviour
{
    //Used to reference game objects
    public GameObject controller; // references the controller
    public GameObject currentPiece; //references the piece you are moving
    public GameObject castlePiece; //references the castle if you are castling

    public Sprite attackTile;
    //Used to store where the current piece wants to move to
    public int xBoard;
    public int yBoard;

    //Used in castling
    public bool isCastle = false;
    public int castleXPos;
    public int castleYPos;
    public int changeInX;

    //Used in en Passant
    public bool enPassant = false;
    public bool attackEnPassant = false;

    //Used to check to create an attack or move tile
    public bool isAttack = false;

    //used for moves to be stored onto a file
    public string moveType = "";

    public void Start()
    {
        if (isAttack || attackEnPassant)
        {
            gameObject.GetComponent<SpriteRenderer>().color = new Color(1, 0, 0, 1);
            gameObject.GetComponent<SpriteRenderer>().sprite = attackTile;
        }
    }
```

```
    public void OnMouseUp()
    {
        doMove();
    }

    public void setBoardPos(int x,int y)
    {
        xBoard = x;
        yBoard = y;
    }

    //used to reference the current piece
    public void setCurrentPiece(GameObject obj)
    {
        currentPiece= obj;
    }

    public void doMove()
    {
        //called when the tile is clicked on (used by assigning component box collider 2d)
        controller = GameObject.FindGameObjectWithTag("GameController");
        Main board = controller.GetComponent<Main>();
        //Used when an attack tile is made
        if (isAttack)
        {
            moveType = translateNotation(currentPiece.GetComponent<Pieces>().xPos,
currentPiece.GetComponent<Pieces>().yPos,
                xBoard, yBoard, true, currentPiece.GetComponent<Pieces>().name);
            attackMove();
        }
        else if (attackEnPassant)
        {
            moveType = translateNotation(currentPiece.GetComponent<Pieces>().xPos,
currentPiece.GetComponent<Pieces>().yPos, xBoard, yBoard, true, currentPiece.GetComponent<Pieces>().name);
            if (currentPiece.GetComponent<Pieces>().playerColour == "W")
            {
                //destroy piece below
                GameObject piece = board.boardPositions[xBoard, yBoard - 1];
                Destroy(piece);
            }
            else
            {
                //destroy piece above
                GameObject piece = board.boardPositions[xBoard, yBoard + 1];
                Destroy(piece);
            }
        }
```

```
        //Used when a castle tile is made
        else if (isCastle)
        {
            board.historyLength = 0;
            board.boardPositions[castleXPos - changeInX, castleYPos] = null;
            if (changeInX == 2)
            {
                //kingside castle
                moveType = "0-0-0";
                changeInX = 1;
            }
            else
            {
                //queenside castle
                moveType = "0-0";
            }
            //castling / moving the pieces
            castlePiece.GetComponent<Pieces>().xPos = xBoard + changeInX;
            castlePiece.GetComponent<Pieces>().yPos = yBoard;
            castlePiece.GetComponent<Pieces>().setCoordPos();
            board.assignBoardPos(castlePiece);
        }
        else
        {
            //normal move notation translation
            moveType = translateNotation(currentPiece.GetComponent<Pieces>().xPos,
currentPiece.GetComponent<Pieces>().yPos,
                xBoard, yBoard, false, currentPiece.GetComponent<Pieces>().name);
        }
        //changes the hasMoved variable for a piece to true if it hasnt moved yet
        if (!currentPiece.GetComponent<Pieces>().hasMoved)
        {
            currentPiece.GetComponent<Pieces>().hasMoved = true;
        }
        //changes the location of the current piece to the wanted location
        board.boardPositions[currentPiece.GetComponent<Pieces>().xPos, currentPiece.GetComponent<Pieces>().yPos] =
null;

        currentPiece.GetComponent<Pieces>().xPos = xBoard;
        currentPiece.GetComponent<Pieces>().yPos = yBoard;
        currentPiece.GetComponent<Pieces>().setCoordPos();

        board.assignBoardPos(currentPiece);

        //destroys all tiles
        currentPiece.GetComponent<Pieces>().DestroyTiles();
```

```
    //Used for pawn being promoted
    if (currentPiece.GetComponent<Pieces>().name[1] == 'P')
    {
        bool check = currentPiece.GetComponent<Pieces>().checkPromote(true);
        char[] vals = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' };
        if (check)
        {
            moveType = "P" + vals[xBoard] + (yBoard+1).ToString();
        }
    }

    //used to allow enPassant
    if (enPassant)
    {
        enPassantMove();
    }

    //used to clear all en passant pieces each turn if not used
    if (board.currentPlayer == "white")
    {
        for (int currentPiece = 8; currentPiece < board.whitePieces.Length; currentPiece++)
        {
            if (board.whitePieces[currentPiece] != null)
            {//sets each pawn to false
                board.whitePieces[currentPiece].GetComponent<Pieces>().enPassantAttack = false;
            }
        }
    }
    else if (board.currentPlayer == "black")
    {
        for (int currentPiece = 8; currentPiece < board.whitePieces.Length; currentPiece++)
        {
            if (board.blackPieces[currentPiece] != null)
            {//sets each pawn to false
                board.blackPieces[currentPiece].GetComponent<Pieces>().enPassantAttack = false;
            }
        }
    }

    board.moveHistory.Add(moveType);

    //50 moves rule
    if (currentPiece.GetComponent<Pieces>().name == "WP" || currentPiece.GetComponent<Pieces>().name == "BP" ||
attackEnPassant || isAttack)
    {//pawns and attacking moves reset the counter
        board.moves = 0.0f;
        board.historyLength = 0;
```

```
    }
    else
    {//increase counter (if 50, the game ends)
        board.moves += 0.5f;
        if (board.moves == 50f)
        {
            board.drawEnd();
        }
    }
    board.changeCurrentPlayer();

    board.setupAttackBoard();
    board.checkDraw(); //checks draw by lack of material

    //checks for threefold repition
    if (board.convertBoardToInt())
    {
        board.drawEnd();
    }
}




public void enPassantMove()
{   //sets enpassant variables if a pawn does the certain requirements
    Main board = controller.GetComponent<Main>();
    if (board.validBoardPos(xBoard + 1, yBoard))
    {
        if (board.boardPositions[xBoard + 1, yBoard] != null)
        {
            if (board.boardPositions[xBoard + 1, yBoard].GetComponent<Pieces>().name[1] == 'P' &&
                board.boardPositions[xBoard + 1, yBoard].GetComponent<Pieces>().playerColour
                != currentPiece.GetComponent<Pieces>().playerColour)
                //if the piece is a pawn of different colour
            {
                board.boardPositions[xBoard + 1, yBoard].GetComponent<Pieces>().enPassantAttack = true;
                if (currentPiece.GetComponent<Pieces>().playerColour == "W")
                {//white pawn
                    //sets the position of where the piece can en passant
                    board.boardPositions[xBoard + 1, yBoard].GetComponent<Pieces>().enPassantXPos = xBoard;
                    board.boardPositions[xBoard + 1, yBoard].GetComponent<Pieces>().enPassantYPos = yBoard - 1;
                }
                else
                { //black pawn
                    board.boardPositions[xBoard + 1, yBoard].GetComponent<Pieces>().enPassantXPos = xBoard;
                    board.boardPositions[xBoard + 1, yBoard].GetComponent<Pieces>().enPassantYPos = yBoard + 1;
```

```
            }
        }
    }
}//repeats for the other side of the pawn
if (board.validBoardPos(xBoard - 1, yBoard))
{
    if (board.boardPositions[xBoard - 1, yBoard] != null)
    {
        if (board.boardPositions[xBoard - 1, yBoard].GetComponent<Pieces>().name[1] == 'P' &&
            board.boardPositions[xBoard - 1, yBoard].GetComponent<Pieces>().playerColour
            != currentPiece.GetComponent<Pieces>().playerColour)
        {
            board.boardPositions[xBoard - 1, yBoard].GetComponent<Pieces>().enPassantAttack = true;
            if (currentPiece.GetComponent<Pieces>().playerColour == "W")
            {
                board.boardPositions[xBoard - 1, yBoard].GetComponent<Pieces>().enPassantXPos = xBoard;
                board.boardPositions[xBoard - 1, yBoard].GetComponent<Pieces>().enPassantYPos = yBoard - 1;
            }
            else
            {
                board.boardPositions[xBoard - 1, yBoard].GetComponent<Pieces>().enPassantXPos = xBoard;
                board.boardPositions[xBoard - 1, yBoard].GetComponent<Pieces>().enPassantYPos = yBoard + 1;
            }
        }
    }
}
}

public void attackMove()
{
    controller = GameObject.FindGameObjectWithTag("GameController");
    Main board = controller.GetComponent<Main>();
    if (board.boardPositions[xBoard, yBoard].name == "WKi" || board.boardPositions[xBoard, yBoard].name == "BKi")
    {
        //used to check if you take the king (only used if a glitch occurs as this shouldn't be possible)
        currentPiece.GetComponent<Pieces>().DestroyTiles();
        board.gameEnd(controller.GetComponent<Main>().boardPositions[xBoard, yBoard].name);
    }
    if (board.boardPositions[xBoard, yBoard].GetComponent<Pieces>().playerColour == "W")
    {
        //gets piece worth adding to corresponding colour
        board.blackPoints += board.boardPositions[xBoard, yBoard].GetComponent<Pieces>().getPieceWorth();
        board.whitePiecesLength -= 1;
    }
    else if (board.boardPositions[xBoard, yBoard].GetComponent<Pieces>().playerColour == "B")
    {
        board.whitePoints += board.boardPositions[xBoard, yBoard].GetComponent<Pieces>().getPieceWorth();
```

```
            board.blackPiecesLength -= 1;
        }
        GameObject piece = board.boardPositions[xBoard, yBoard];
        Destroy(piece);
    }


    public string translateNotation(int oldXAxis, int oldYAxis, int newXAxis, int newYAxis, bool isAttacking,string
pieceName)
    {
        //used to translate each move into algebraic notation to later be stored in moves history
        char[] vals = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' };
        string pieceNameDone = "";
        string attacking = "";
        if (isAttacking)
        {
            attacking = "x";
        }
        //Convert name to notation
        switch (pieceName)
        {
            case "BB":
            case "WB":
                pieceNameDone = "B";
                break;
            case "BKi":
            case "WKi":
                pieceNameDone = "K";
                break;
            case "BKn":
            case "WKn":
                pieceNameDone = "N";
                break;
            case "BP":
            case "WP":
                pieceNameDone = "P";
                break;
            case "BQ":
            case "WQ":
                pieceNameDone = "Q";
                break;
            case "BR":
            case "WR":
                pieceNameDone = "R";
                break;
        }
        //create full string
```

```
        return pieceNameDone + vals[oldXAxis] + (oldYAxis+1).ToString() + attacking + vals[newXAxis] + (newYAxis +
1).ToString();
    }
}
```

AI.cs

```
using UnityEngine;


public class AI : MonoBehaviour
{
    public GameObject controller;

    // Start is called before the first frame update
    private void Start()
    {
        controller = GameObject.FindGameObjectWithTag("GameController");
    }


    public void AiButton()
    {
        Main main = controller.GetComponent<Main>();

        if (main.currentPlayer == "black" && main.againstAI == true
            && main.gameOver == false && main.canMove == true)
        {
            getMove();
        }
    }
    public void getMove()
    {
        //gets moves
        controller = GameObject.FindGameObjectWithTag("GameController");
        for (int currentVal = 0; currentVal < controller.GetComponent<Main>().blackPieces.Length; currentVal++)
        {
            if (controller.GetComponent<Main>().blackPieces[currentVal] != null)
            {
                controller.GetComponent<Main>().blackPieces[currentVal].GetComponent<Pieces>().possibleMoves(true);
            }
        }
        GameObject[] moveTiles = GameObject.FindGameObjectsWithTag("MovePlate");
        //array of the worth of each move
        float[] calculatedMoves = new float[moveTiles.Length];
        for (int moveVal = 0; moveVal < moveTiles.Length; moveVal++)
        {
```

```
            //calculate moves
            calculatedMoves[moveVal] = calculateMove(moveTiles[moveVal]);
        }
        //get the highest and do the move
        moveTiles[getHighestVal(calculatedMoves)].GetComponent<MoveTiles>().doMove();

    }

    public int getHighestVal(float[] valArray)
    {
        float highestVal = valArray[0];
        int returnIndex = 0;
        for (int currentVal = 1; currentVal < valArray.Length; currentVal++)
        {
            if (valArray[currentVal] > highestVal)
            {
                highestVal = valArray[currentVal];
                returnIndex = currentVal;
            }
        }
        return returnIndex;
    }

    public float calculateMove(GameObject moveTile)
    {
        float moveVal = 0;
        //if attacking move
        if (moveTile.GetComponent<MoveTiles>().isAttack == true)
        {
            //calculate the worth of the piece being taken
            moveVal += controller.GetComponent<Main>().boardPositions[moveTile.GetComponent<MoveTiles>().xBoard,
moveTile.GetComponent<MoveTiles>().yBoard].GetComponent<Pieces>().getPieceWorth() * 10
                + getSwitchTable(moveTile.GetComponent<MoveTiles>().xBoard,
moveTile.GetComponent<MoveTiles>().yBoard,
                controller.GetComponent<Main>().boardPositions[moveTile.GetComponent<MoveTiles>().xBoard,
moveTile.GetComponent<MoveTiles>().yBoard].GetComponent<Pieces>().name);
        }
        //calculate the change in move worth
        string tempName = moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().name;
        moveVal += getSwitchTable(moveTile.GetComponent<MoveTiles>().xBoard,
moveTile.GetComponent<MoveTiles>().yBoard, tempName)
            - getSwitchTable(moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().xPos,
moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().yPos, tempName);

        //determine if the piece can be taken if you move to this location using the attack board
        if(controller.GetComponent<Main>().whiteAttackBoard[moveTile.GetComponent<MoveTiles>().xBoard,
moveTile.GetComponent<MoveTiles>().yBoard] == true)
```

```
    { //(we subtract the value of the piece if so, assuming it would take the piece)
        moveVal -=
controller.GetComponent<Main>().boardPositions[moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<
Pieces>().xPos,

moveTile.GetComponent<MoveTiles>().currentPiece.GetComponent<Pieces>().yPos].GetComponent<Pieces>().getPiece
Worth() * 10;
    }
    return moveVal;
  }

  public float getSwitchTable(int xPos, int yPos, string name)
  {
    switch (name)
    {
      case ("WKi"):
      case ("BKi"):
        return getKingTable(xPos,yPos);
      case ("WKn"):
      case ("BKn"):
        return getKnightTable(xPos, yPos);
      case ("WQ"):
      case ("BQ"):
        return getQueenTable(xPos, yPos);
      case ("WP"):
      case ("BP"):
        return getPawnTable(xPos, yPos);
      case ("WR"):
      case ("BR"):
        return getRookTable(xPos, yPos);
    }
    return 0;
  }

  public float getKnightTable(int xPos, int yPos)
  {
    int newXPos = 7 - xPos;
    int newYPos = 7 - yPos;
    float[,] knightTable =
      { {-5,-4,-3,-3,-3,-3,-4,-5 },
      {-4,-2,  0,  0,  0,  0,-2,-4 },
      {-3,  0, 1, 1.5f, 1.5f, 1,  0,-3 },
      {-3,  0.5f, 1.5f, 2, 2, 1.5f, 0.5f,-3 },
      {-3,  0, 1.5f, 2, 2, 1.5f,  0,-3 },
      {-3,  0.5f, 1, 1.5f, 1.5f, 1,  0.5f,-3 },
      {-4,-2,  0,  0.5f,  0.5f,  0,-2,-4 },
      {-5,-4,-3,-3,-3,-3,-4,-5 } };
```

```
    return knightTable[newXPos, newYPos];
}

public float getKingTable(int xPos, int yPos)
{
    int newXPos = 7 - xPos;
    int newYPos = 7 - yPos;
    float[,] kingTable =
        { {-3,-4,-4,-5,-5,-4,-4,-3 },
        {-3,-4,-4,-5,-5,-4,-4,-3 },
        {-3,-4,-4,-5,-5,-4,-4,-3 },
        {-3,-4,-4,-5,-5,-4,-4,-3 },
        {-2,-3,-3,-4,-4,-3,-3,-2 },
        {-1,-2,-2,-2,-2,-2,-2,-1 },
        {2, 2,  0,  0,  0,  0, 2, 2},
        {2, 3, 1,  0,  0, 1, 3, 2 } };
    return kingTable[newXPos, newYPos];
}

public float getQueenTable(int xPos, int yPos)
{
    int newXPos = 7 - xPos;
    int newYPos = 7 - yPos;
    float[,] queenTable =
        { {-2,-1,-1, -0.5f, -0.5f,-1,-1,-2 },
        {-1,  0,  0,  0,  0,  0,  0,-1 },
        {-1,  0,  0.5f,  0.5f,  0.5f,  0.5f,  0, -1 },
        {-0.5f,  0,  0.5f,  0.5f,  0.5f,  0.5f,  0, -0.5f },
        {0,  0,  0.5f,  0.5f,  0.5f,  0.5f,  0, -0.5f },
        {-1,  0.5f,  0.5f,  0.5f,  0.5f,  0.5f,  0,-1 },
        {-1,  0,  0.5f,  0,  0,  0,  0,-1 },
        {-2,-1,-1, -0.5f, -0.5f,-1,-1, -2 } };
    return queenTable[newXPos, newYPos];
}

public float getBishopTable(int xPos, int yPos)
{
    int newXPos = 7 - xPos;
    int newYPos = 7 - yPos;
    float[,] bishopTable =
        { {-2,-1,-1,-1,-1,-1,-1,-2 },
        {-1,  0,  0,  0,  0,  0,  0,-1 },
        {-1,  0,  0.5f, 1, 1,  0.5f,  0,-1 },
        {-1,  0.5f,  0.5f, 1, 1,  0.5f,  0.5f,-1 },
        {-1,  0, 1, 1, 1, 1,  0,-1 },
        {-1, 1, 1, 1, 1, 1, 1,-1 },
        {-1,  0.5f,  0,  0,  0,  0,  0.5f,-1 },
```

```
        {-2,-1,-1,-1,-1,-1,-1,-2 } };
    return bishopTable[newXPos, newYPos];
}

public float getRookTable(int xPos, int yPos)
{
    int newXPos = 7 - xPos;
    int newYPos = 7 - yPos;
    float[,] rookTable =
        {{0,  0,  0,  0,  0,  0,  0,  0 },
        {0.5f, 1, 1, 1, 1, 1, 1,  0.5f },
        {-0.5f,  0,  0,  0,  0,  0,  0, -0.5f },
        {-0.5f,  0,  0,  0,  0,  0,  0, -0.5f },
        {-0.5f,  0,  0,  0,  0,  0,  0, -0.5f },
        {-0.5f,  0,  0,  0,  0,  0,  0, -0.5f },
        {-0.5f,  0,  0,  0,  0,  0,  0, -0.5f },
        {0,  0,  0,  0.5f,  0.5f,  0,  0,  0 } };
    return rookTable[newXPos, newYPos];
}

public float getPawnTable(int xPos, int yPos)
{
    int newXPos = 7 - xPos;
    int newYPos = 7 - yPos;
    float[,] pawnTable =
        {{10, 10, 10, 10, 10, 10, 10, 10},
        {5, 5, 5, 5, 5, 5, 5, 5},
        {1, 1, 2, 3, 3, 2, 1, 1},
        {5, 5, 1, 2.5f, 2.5f, 1, 0.5f, 0.5f},
        {0, 0, 0, 2, 2, 0, 0, 0},
        {0.5f, -0.5f, -1, 0, 0, -1, -0.5f, 0.5f},
        {0.5f, 1, 1, -2, -2, 1, 1, 0.5f},
        {10, 10, 10, 10, 10, 10, 10, 10}};
    return pawnTable[newXPos, newYPos];
}
```

Timer.cs

```
using UnityEngine;
using UnityEngine.UI;

public class Timer : MonoBehaviour
{
    private GameObject controller;
    private bool paused = true;
    private float currentTime = 900; //15 minutes in seconds
    public Text timeText;

    void Start()
    {
        controller = GameObject.FindGameObjectWithTag("GameController");
    }

    // Update is called once per frame
    void Update()
    {
        if (!paused && controller.GetComponent<Main>().canMove)
        {
            if (currentTime > 0)
            {
                //calculate change in time per frame
                currentTime -= Time.deltaTime;
                convertToText();
            }
            else
            {
                //user runs out of time
                currentTime = 0f;
                convertToText();
                //ends the game
                Main endGame = controller.GetComponent<Main>();
                endGame.gameEnd(endGame.currentPlayer);
            }
        }
    }

    public void cyclePause()
    {
        paused = !paused;
    }
```

```
    public void convertToText()
    {
        //converting float to text in time format of minute : second : milliseconds
        timeText.text = string.Format("{0:00}:{1:00}:{2:000}", Mathf.FloorToInt(currentTime / 60),
Mathf.FloorToInt(currentTime % 60), Mathf.FloorToInt(currentTime % 1 * 1000));
    }

    public void stopTimer()
    {
        paused = true;
    }
}
```

BoardUI.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class BoardUI : MonoBehaviour
{
    public GameObject controller;


    // Start is called before the first frame update
    void Start()
    {
        controller = GameObject.FindGameObjectWithTag("GameController");
    }

    public void noButton()
    {
        //assigned to the game object "NoButton" in the "PlayAgain" menu to load the scene "Start Menu"
        SceneManager.LoadScene("Start Menu");
    }

    public void yesButton()
    {
        //assigned to the game object "YesButton" in the "PlayAgain"
        //menu to load the scene "Offline Game"
        //which loads the current scene you are on (plays again)
        SceneManager.LoadScene("Offline Game");
    }

    public void drawButton()
    {
        Main main = controller.GetComponent<Main>();
        main.drawEnd();
    }

    public void pauseStartGame()
    {
        Main main = controller.GetComponent<Main>();
        main.canMove = !main.canMove;
    }

    public void resignButton()
    {
```

```
    Main main = controller.GetComponent<Main>();
    //assigned to the game object "YesButton" in the "ResignMenu" to call the function
    //which ends the game
    if (main.currentPlayer == "white")
    {
        main.gameEnd("WKi");
    }
    else
    {
        main.gameEnd("BKi");
    }
}

public void backButton()
{
    //loads the next scene in the build manager section
    SceneManager.LoadScene(
        SceneManager.GetActiveScene().buildIndex - 1);
}

public void startGame()
{
    controller.GetComponent<Main>().startGame();
}

public void startAIGame()
{
    controller.GetComponent<Main>().againstAI = true;
    controller.GetComponent<Main>().startGame();
}

public void startOnlineGame()
{
    controller.GetComponent<Main>().isOnlineGame = true;
    controller.GetComponent<Main>().startGame();
}

public void getQueen()
{
    //minimise the menu
    GameObject.FindGameObjectWithTag("PromoteMenu").transform.localScale = new Vector3(0,0,0);
    setPiece("Q");
}

public void getRook()
{
    //minimise the menu
```

```
    GameObject.FindGameObjectWithTag("PromoteMenu").transform.localScale = new Vector3(0, 0, 0);
    setPiece("R");
  }

  public void getBishop()
  {
    //minimise the menu
    GameObject.FindGameObjectWithTag("PromoteMenu").transform.localScale = new Vector3(0, 0, 0);
    setPiece("B");
  }

  public void getKnight()
  {
    //minimise the menu
    GameObject.FindGameObjectWithTag("PromoteMenu").transform.localScale = new Vector3(0, 0, 0);
    setPiece("Kn");
  }

  public void setPiece(string pieceName)
  {
    int xpos = 0;
    int ypos = 0;
    //used to check if a pawn can be promoted
    Main board = controller.GetComponent<Main>();
    if (board.currentPlayer == "black")
    { //white selecting piece
      for (int i = 0; i < board.whitePieces.Length; i++)
      {
        if (board.whitePieces[i] != null)
        {
          if (board.whitePieces[i].GetComponent<Pieces>().name == "WP" &&
board.whitePieces[i].GetComponent<Pieces>().yPos == 7)
          { //If the piece is a pawn and is at the end of the board where it can promote
            ypos = board.whitePieces[i].GetComponent<Pieces>().yPos;
            xpos = board.whitePieces[i].GetComponent<Pieces>().xPos;
            Destroy(board.whitePieces[i]); //destroy the pawn
            board.setPiece(xpos, ypos, "W", i, pieceName, true); //create a new piece at the new xPos and yPos
          }
        }
      }
    }
    else
    { //black selecting piece
      for (int i = 0; i < board.blackPieces.Length; i++)
      {
        if (board.blackPieces[i] != null)
        {
```

```
                if (board.blackPieces[i].GetComponent<Pieces>().name == "BP" &&
board.blackPieces[i].GetComponent<Pieces>().yPos == 0)
                { //If the piece is a pawn and is at the end of the board where it can promote
                    ypos = board.blackPieces[i].GetComponent<Pieces>().yPos;
                    xpos = board.blackPieces[i].GetComponent<Pieces>().xPos;
                    Destroy(board.blackPieces[i]); //destroy the pawn
                    board.setPiece(xpos, ypos, "B", i, pieceName, true); //create a new piece at the new xPos and yPos
                }
            }
        }
    }
}
```

ButtonControl.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class ButtonControl : MonoBehaviour
{
    //these functions are used to allow functions to occur when a button is pressed
    public void offlineButton()
    {
        //loads the next scene in the build manager section
        SceneManager.LoadScene(
            SceneManager.GetActiveScene().buildIndex + 1);
    }

    public void quitButton()
    {
        //Quits the application
        Application.Quit();
```

```
    }
}
```

PlayInput.cs

```
using UnityEngine.UI;
using UnityEngine;
using System.Text.RegularExpressions;

public class PlayInput : MonoBehaviour
{
    public Button confirm;
    public InputField inputPlayerOne;
    public InputField inputPlayerTwo;

    public string playerOneUsername;
    public string playerTwoUsername;

    // Start is called before the first frame update
    void Start()
    {
        confirm.onClick.AddListener(GetInputOnClickHandler);
    }

    // Update is called once per frame
    public void GetInputOnClickHandler()
    {
        //Names contain only letters, numbers and underscore, has length 4-16 characters, and not equal
        if(Regex.IsMatch(inputPlayerOne.text, @"^[a-zA-Z0-9_]+$") && Regex.IsMatch(inputPlayerTwo.text,
@"^[a-zA-Z0-9_]+$")
            && inputPlayerOne.text.Length <= 16 && inputPlayerTwo.text.Length <= 16 && inputPlayerOne.text.Length >=4
&& inputPlayerTwo.text.Length >= 4
            && inputPlayerOne.text != inputPlayerTwo.text)
        {
            //Set names, and show a button to start the game
            playerOneUsername = inputPlayerOne.text;
            playerTwoUsername = inputPlayerTwo.text;
            GameObject.FindGameObjectWithTag("StartGame").transform.localScale = new Vector3(1, 1, 1);
        }
        else
        {
            //Minimise the start game button
            GameObject.FindGameObjectWithTag("StartGame").transform.localScale = new Vector3(0, 0, 0);
        }
    }
}
```