**Chess+**

Robert Lucas

OCR Computer Science Project

Table of Contents

[Project Proposal 5](#_Toc124789501)

[Concept 5](#_Toc124789502)

[Stakeholders 5](#_Toc124789503)

[Key points from responses to questionnaire (10 people) 5](#_Toc124789504)

[Computational Methods Used 6](#_Toc124789505)

[Abstraction 6](#_Toc124789506)

[Thinking Ahead 6](#_Toc124789507)

[Decomposition 6](#_Toc124789508)

[Procedures 7](#_Toc124789509)

[Logic 7](#_Toc124789510)

[Concurrency 7](#_Toc124789511)

[Divide and Conquer 7](#_Toc124789512)

[Encapsulation 7](#_Toc124789513)

[Backtracking 8](#_Toc124789514)

[Heuristics 8](#_Toc124789515)

[Features based on existing solutions 8](#_Toc124789516)

[Features included in existing solutions 8](#_Toc124789517)

[Features missing from existing solutions 9](#_Toc124789518)

[Summary 10](#_Toc124789519)

[Approach based on existing solutions 10](#_Toc124789520)

[Solution Features 11](#_Toc124789521)

[Limitations 13](#_Toc124789522)

[Art 13](#_Toc124789523)

[Money 13](#_Toc124789524)

[Computational power 13](#_Toc124789525)

[Time 13](#_Toc124789526)

[Platform 13](#_Toc124789527)

[Requirements 13](#_Toc124789528)

[Success Criteria 13](#_Toc124789529)

[Design Flowcharts 14](#_Toc124789530)

[Overview 14](#_Toc124789531)

[Chess Manager 16](#_Toc124789532)

[Game Manager 17](#_Toc124789533)

[Pregame Networking Flow 17](#_Toc124789534)

[In-Game Networking Flow 19](#_Toc124789535)

[TODO: UI Flow diagram 19](#_Toc124789536)

[Development 20](#_Toc124789537)

[Automated and manual testing 20](#_Toc124789538)

[Validation Testing 21](#_Toc124789539)

[Encoding-Decoding Testing 22](#_Toc124789540)

[UID uniqueness testing 22](#_Toc124789541)

[Spreadsheet 22](#_Toc124789542)

[Testing with stakeholders 23](#_Toc124789543)

[Spreadsheet 23](#_Toc124789544)

[Version Control 23](#_Toc124789545)

[Developmental challenges encountered and solutions 23](#_Toc124789546)

[Accounting for all display sizes and ratios is difficult 23](#_Toc124789547)

[Entering play mode is frustrating 23](#_Toc124789548)

[Save games need to be flexible enough to support every gamemode 24](#_Toc124789549)

[The listener socket for servers doesn’t shut down properly 26](#_Toc124789550)

[Need a saved latest bug-free version to give to testers 27](#_Toc124789551)

[A ‘ghost’ user will sometimes appear in the player list when joining lobbies 27](#_Toc124789552)

[Trying to host twice in quick succession creates errors 28](#_Toc124789553)

[The AI doesn’t pick the best move 28](#_Toc124789554)

[Developing the sound system 29](#_Toc124789555)

[AI Freezing 30](#_Toc124789556)

[Connect 4 doesn’t fit the input system 31](#_Toc124789557)

[Connecting through a public IP stopped working 32](#_Toc124789558)

[Using multithreading for the MiniMax algorithm 33](#_Toc124789559)

[Missing chess AI optimization due to generalisation 33](#_Toc124789560)

[Versions during development 33](#_Toc124789561)

[First version with UI 33](#_Toc124789562)

[First version with help system and save system 35](#_Toc124789563)

[Large UI improvements, functional local play and AI added 37](#_Toc124789564)

[Large UI overhaul 40](#_Toc124789565)

[Solution 45](#_Toc124789566)

[Overview 45](#_Toc124789567)

[Game Mode System 45](#_Toc124789568)

[GameManagerData 45](#_Toc124789569)

[GameManager 45](#_Toc124789570)

[Board 46](#_Toc124789571)

[Piece 46](#_Toc124789572)

[Networking System 47](#_Toc124789573)

[Network Manager 47](#_Toc124789574)

[Client and Server 48](#_Toc124789575)

[Packet System 48](#_Toc124789576)

[AI System (MiniMax Algorithm) 49](#_Toc124789577)

[Help System 51](#_Toc124789578)

[UI System 52](#_Toc124789579)

[Validation System 52](#_Toc124789580)

[Save system 53](#_Toc124789581)

[Evaluation 54](#_Toc124789582)

[TODO: Success compared to initial goals 54](#_Toc124789583)

[TODO: Final Feedback 54](#_Toc124789584)

[TODO: Future Improvements 54](#_Toc124789585)

[Code and Screenshots 55](#_Toc124789586)

[Screenshots 55](#_Toc124789587)

[Code 55](#_Toc124789588)

Project Proposal

## Concept

Chess is an ancient game played by millions around the world and has seen many apps made for competitive and casual play however these tend to offer only the default versions of chess or slight spinoffs that still play on the same sized board with the same pieces but with changes to the rules.

I plan to make a chess game with support for adding many game modes as well as multiplayer support, save game support and minimax-based AI that will work with any new game mode.

While other chess options do exist, the key unique feature of this game is the many available game modes and the ability to save all of them. This will make the game more appealing to a casual audience who want to experience unique variations of chess, be able to play chess with more than one friend or play longer games that they can save and come back to.

The game modes will also include other established board games such as checkers and Viking Chess (Hnefatafl). While some of these can be played online (it’s worth noting that I couldn’t find an online version of Viking Chess), there isn’t a single place where you can play all of them which makes this game more convenient than the alternatives.

## Stakeholders

The stakeholders in this project are me and the players as all decisions should be made to either make development more feasible for me or to improve the player experience as this will bring in more players and improve player retention.

### Key points from responses to the questionnaire (10 people)

#### Do you have a Windows device?

* Almost everyone asked (8) had one
* One person had a Mac

Making the game exclusive to Windows shouldn’t reduce the player base by too much however making the game support Mac wouldn’t require much more development due to Unity’s platform independence.

While I have considered supporting Android / IOS, due to smaller screen sizes and different typical aspect ratios, the UI would have to be completely redone for mobile devices. Most mobile devices probably wouldn’t have enough power to run the AI at a reasonable speed.

#### Do you know how to play Chess?

* Around 1/3 of people knew how to play chess

This statistic means that the tutorials/help provided must not assume that the player knows how to play chess

The name of the game should not leave the impression that this game is only chess

#### What do you think of a game where you could play lots of unique board games against friends?

* Most people (7) liked this idea
* Some (2) mentioned that this would need to be different from the offerings of online sites

This game needs to have lots of game modes that either aren’t available in other places or aren’t available in one place.

#### Do you know how to forward ports?

* No one knew how to forward ports

The game needs to have detailed instructions on how to forward ports

In the future, the game could use an external proxy to no longer require port forwarding.

## Computational Methods Used

### Abstraction

Abstraction is necessary to hide complexity allowing new parts to be developed without needing an understanding of how the entire existing code base works.

It also allows unnecessary detail to be removed completely in some places making the game simpler to develop

* The game will be 2D and not realistic simplifying it reducing development time and making the board easier to understand for a user
* The game will use Unity to handle most of the rendering, IO and packaging the game into an executable reducing development time
* After the multiplayer system is developed, the game logic will only call exposed functions on the networking classes without having to worry about handling networking allowing different sections of the solution to remain more independent allowing for easier iteration on different parts of the code
* IP addresses will be encoded to alphabetic strings to make them easier to remember and pass on to friends

### Thinking Ahead

Thinking ahead is necessary to prevent bad decisions that can make the game difficult to develop/work with later. This is especially important as I need to make it very easy to add new game modes.

* As the game is symmetrical, code can be reused for both players decreasing development time
* Instead of writing a client and a client-server hybrid to allow one of the players to host, the client can be reused and a separate server can be made. The local client can then connect to the local server. This will reduce code duplication between a client and a client-server hybrid reducing development time and decreasing the chance of errors occurring due to mismatches between how a client and how a client-server works.
* The board can be serialised to save its state and this needs to be done in a smart way that can work for any game mode and any number of custom pieces with custom data as well as not breaking with future updates
* The networking code needs to be heavily pre-planned to ensure data is in the correct format and available to be sent

Decomposition

A project must be decomposed into smaller chunks to allow components to be developed one at a time making the process simpler.

* The networking library, game logic, AI and visuals can all be developed separately as they only interact with each other in limited ways allowing the project to be easier to manage
* These subsystems are further divided into smaller chunks such as the networking being divided into client and server and the chess manager being broken down into the game, save and input system

Procedures

Procedures are needed to complete a series of steps with each step being dependent on the ones before.

* The game will use procedures such as:
  + The network library going from receiving data to validating it to processing it to passing it to the game logic for it to be applied to the board
  + The client clicking on a square to move a piece, to the input manager receiving that, to the game manager validating the move, to the game manager applying the move, to the game manager checking for a check or checkmate and finally the game manager updating the visuals manager

Logic

* Logic will be used very frequently for example evaluating which moves a player can and can’t make or not letting more players join a lobby once it is full both of which are essential for the game to work

Concurrency

Concurrency is needed to run slow or blocking (such as waiting for a client connection) code that isn’t interdependent. This is especially important when using the Unity Engine as it has a single main thread and any code running on that thread directly affects framerate.

* The networking library will need to run concurrently to be able to receive and process data from other players independently of the game’s framerate
* The AI will need to run on a separate concurrently to the main thread as calculating the best move can take up to 30 seconds and the game shouldn’t freeze during this time as this can ruin the user experience

### Divide and Conquer

This technique is useful for splitting a task into smaller subtasks that can be completed in parallel with greater efficiency

* AI will run on multiple threads at the same time with each thread searching for different possible moves

Encapsulation

Encapsulation is used to control communication between classes to carefully select what class data can be modified externally

* Input manager will encapsulate Unity’s input system to make it easier to use and adding features such as support for multiple key presses
* To maintain a comprehensible class hierarchy, the ChessManager will encapsulate many methods from other managers to keep the relation tree more similar to a star than a mesh making it easier to understand and debug.

Backtracking

* The Mini-Max algorithm uses backtracking

Heuristics

Heuristics are useful for optimising algorithms based on existing knowledge

* The AI will use heuristics such as not going further down a path that has a very low score

## Features based on existing solutions

### Features included in existing solutions



^ Chess.com

Chess.com has a simplified 2D UI that will be easier to develop than a 3D one. It is also very easy to understand what is happening on the board due to contrasting colours and unique piece designs.

Pieces were probably created with an OOP approach as pieces share a lot of functionality and it would make creating new pieces easier

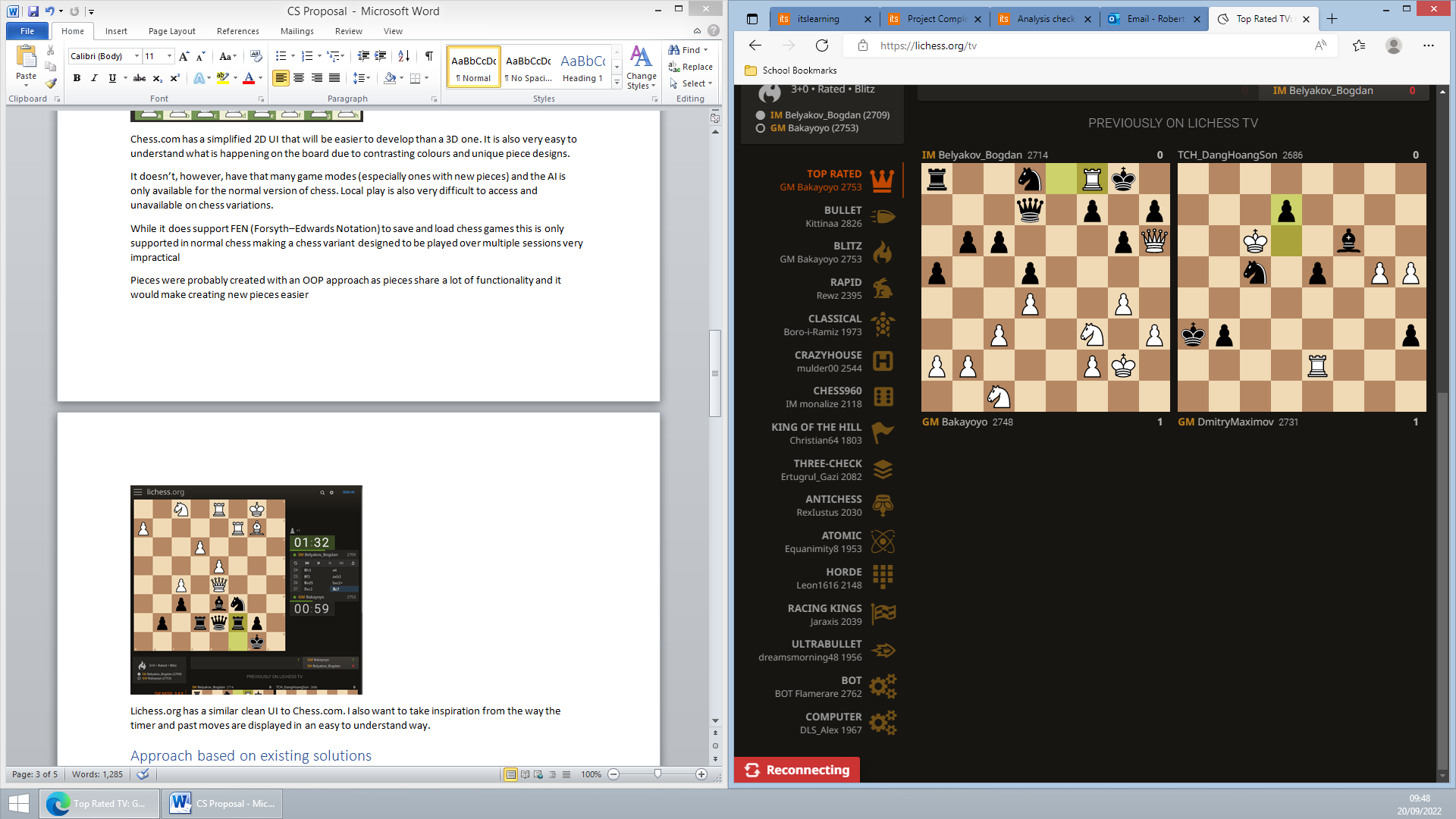


Lichess.org has a similar clean UI to Chess.com. I also want to take inspiration from the way the timer and past moves are displayed in an easy-to-understand way. It also displays the moves available to each piece and highlights a piece’s last moves.

### Features missing from existing solutions

Chess.com doesn’t, however, have that many game modes (especially ones with new pieces) and the AI is only available for the normal version of chess. Local play is also very difficult to access and unavailable on chess variations.

While it does support FEN (Forsyth–Edwards Notation) to save and load chess games this is only supported in normal chess making a chess variant designed to be played over multiple sessions very impractical



Lichess does also have quite a few modes however, like Chess.com, Lichess doesn’t stray too far from normal chess and doesn’t support saving games in progress that can’t be represented with FEN



This is a generic app on the Play Store with the board represented in 3D. I find this representation to be more cluttered and more difficult to understand.

A 3D view would also require more developer time as models for custom game modes would have to be created and there are extra complications around camera controls and ensuring that the contrast is good enough on all the pieces to see them clearly

I do, however, like that when you click on a piece it shows you a preview of legal moves and believe that it would greatly improve the user experience

### Summary

##### Key positives other solutions have

* 2D simplistic UI
* Being able to load games (at least for normal chess)
* Shows where pieces can move
* Shows where pieces moved on the last turn
* Simple piece icons

##### Unique components not found in most other solutions

* Support for a large number of game modes
* Support for saving every game mode
* Support for online and local play for every game mode
* AI support for every game mode

## Approach based on existing solutions

Most chess games will probably use an object-oriented approach to create the pieces as they need to share some functionality which can be provided by a parent class. I will do this and also use a similar approach for implementing multiple game modes as a large amount of functionality will be shared between them

A lot of chess AIs use the minimax algorithm with alpha-beta pruning and I will use this as I have some experience with it. The minimax algorithm (excluding some optimisations) will also work with any game mode as it requests all possible moves from a piece which custom pieces can also provide. I might also use Zobrist hashing to create transposition tables for the board but this would be a massive increase in complexity, especially for custom game modes.

Sockets are commonly used for low-level and efficient communications and, as I want very fine control over network communications, I will be using that.

Binary-serialised save files are frequently used in gaming (albeit not in the three examples above) so I will be using them as they do not need to be human-readable or editable and binary save files are very performant, space efficient and can be sent relatively easily with my custom networking framework. While this may provide some resilience to hacking due to their obfuscated nature, as this isn’t a competitive game, I won’t make any attempt to prevent save game modification.

I will be using C# for the programming language as it is very commonly used in game development especially as Unity and other game engines require its usage. It is also performant enough for my uses and scales well for large projects with the use of namespaces.

While most modern game development companies seem to be using extreme programming to create functional, frequent releases that prioritise community feedback improving the game over time, I will be using an iterative design process by developing this game first to a minimal prototype with only networking and then add local play and AI as doing things in small steps is easier and it is easier to reduce features for local play rather than add them back in for multiplayer. Finally, I will add improved UI and other quality-of-life features.

## Solution Features

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | Priority (1 – 5 essential) | Description | Explanation |
| Core local chess game | 1 | Core local player versus player chess game | - |
| Support for adding game modes | 2 | Creating the tools within my code to be able to quickly develop new game modes and allow other systems to interact with every game mode without having to write specialised code | Important as AI, Multiplayer, and game saves all require this to be working  Having a large number of available game modes is a key feature that differentiates this game from others |
| Functional UI | 3 | Functional UI that might not be easy to use but will allow testing of other features | Functional UI is needed during development to test features |
| Multiplayer | 4 | Allow players to play with each other across the internet  Must support working with any future game modes | Multiplayer is also a key differentiator that makes this game stand out.  It must be developed early so that some features such as save game design are built with it in mind |
| Help System | 4 | Help system that helps players understand how to use the game | A help system to guide players will allow them to learn how to play without trial and error which can cause frustration |
| AI | 4 | An AI that can play any game mode | An AI is important as players may not always have friends online to play against |
| Save games | 5 | Allows the player to save the current state of the game and load it later including loading it into a multiplayer game | Save games allow the existence of game modes that take a long time to play as they can be completed in multiple sessions |
| Good UI | 6 | An intuitive UI that is easy for a new player to use, possibly with tutorials | A good, intuitive UI will reduce the player's reliance on the help system and a bad UI can cause frustration |
| Animations | 7 | Animations for pieces moving from square to square | This will make where pieces have moved more obvious |
| Themes | 7 | Allow the user to change board themes | User customisation allows picking how they want their game to look increasing enjoyment |

## Limitations

### Art

This game will require art assets for each piece as well as a logo, fonts and UI elements. Because of my limited knowledge in this field and time limitations, I will use third-party assets for some components and simplistic elements for most other things

### Money

Third-party asset packs, animations packs, font packs and more cost money which has to be managed carefully as this project is on a small budget so in most cases free alternatives will be used

### Computational power

The AI can’t be too complex as the typical user system won’t have a powerful CPU and this game needs to be accessible to as many people as possible meaning that the AI may be quite limited in its skill

### Time

Development time is limited due to the lack of other developers and the limited time frame therefore each feature must be considered with the time it takes to make it

### Platform

While ideally this game would be made on a website, due to my more limited knowledge of JavaScript and the large amount of extra time required for creating a website, I’ve decided to use Unity and C# as an alternative. This would reduce player numbers as players would need to also have a game distribution platform such as Steam installed

This game also won’t be cross-platform as, despite Unity supporting IOS and Android, these require extra development time due to UI considerations and would require scaling back the AI even further

## Requirements

* A powerful computer capable of running Unity, Visual Studio 2022 and completing Unity builds in a reasonable amount of time
* Ability to code C#
* Ability to use sockets
* Ability to implement the minimax algorithm
* Ability to work with binary serialised data for networking and saves
* Ability to work with files and IO
* Ability to use Unity
* Ability to use Unity’s PlayerPrefs database system to save preferences
* Ability to use HTML for help pages
* Ability to use Python for generating help pages from HTML skeleton and for generating repetitive C# code
* A second computer capable of running the built game and connecting to a hotspot to test multiplayer support

## Success Criteria

|  |  |  |  |
| --- | --- | --- | --- |
| Criteria | Category | Measured by | Justification |
| All essential solution features included | Functionality  Usability | Tick list  Testing | The game must be complete |
| Error-free | Robustness | Stress testing (possibly automated) | Clients should never have to restart the game due to errors to ensure a good UX |
| Secure (optional) | Robustness | Ensure data is encrypted  Use Wireshark to inspect packets sent and ensure they are encrypted | Optional as no sensitive data is transferred |
| User-friendliness | Usability | Tests with the target audience | Users must be able to use the software without any external assistance |
| AI performance reasonable | Functionality | AI makes moves (on average) in under 1 minute on mid-range hardware | Users should be able to use the AI without requiring very powerful hardware |
| Gamemodes easy to develop and add | Functionality | Not having to modify other components of the project to add a game mode | This game should have regular game mode updates after launch to keep people interested and these should be easy to add. |
| First functional build released by end of November | Functionality | - | Ensures project is completed on time |

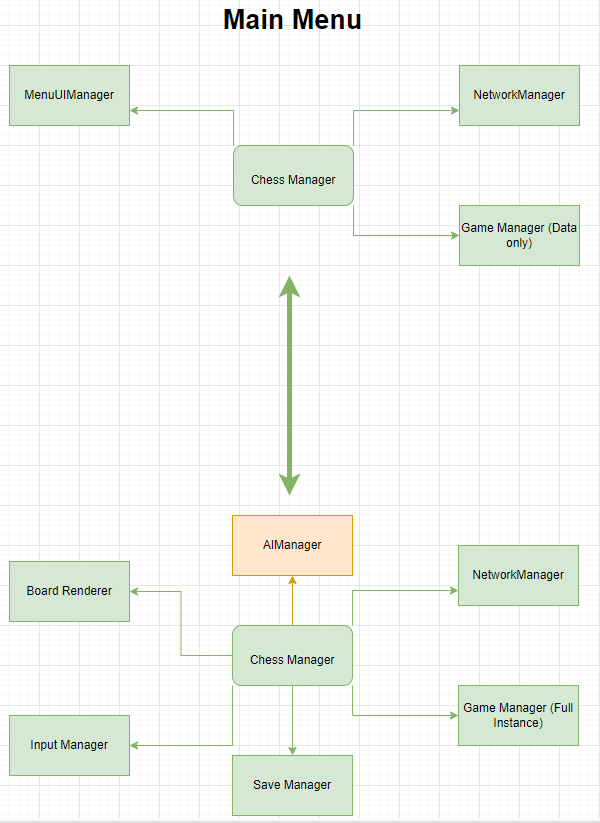
## Design Flowcharts

### Overview

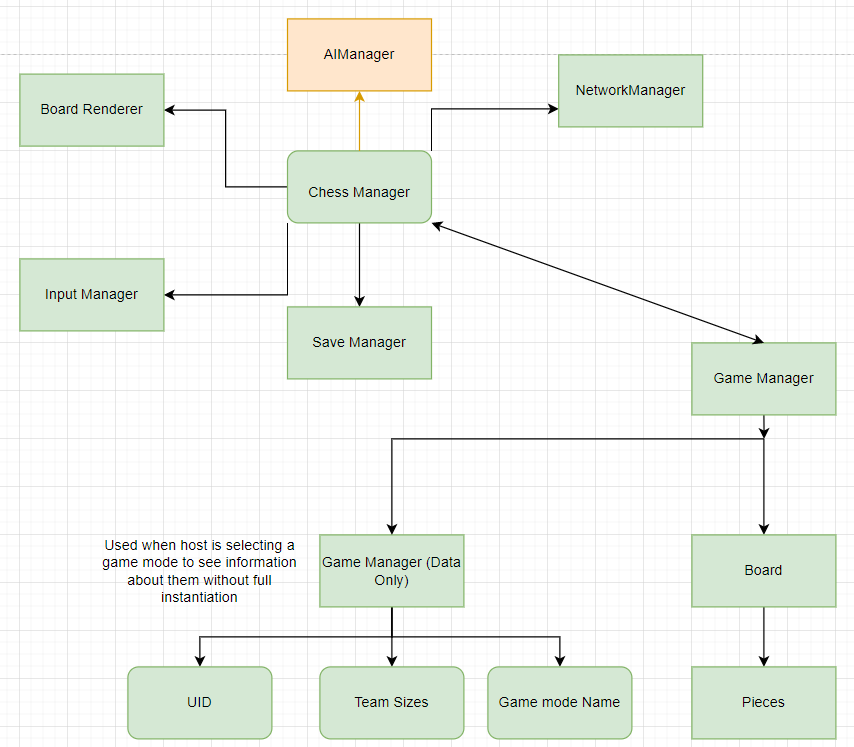
(Double lines represent references, arrows represent data flow)



### Chess Manager

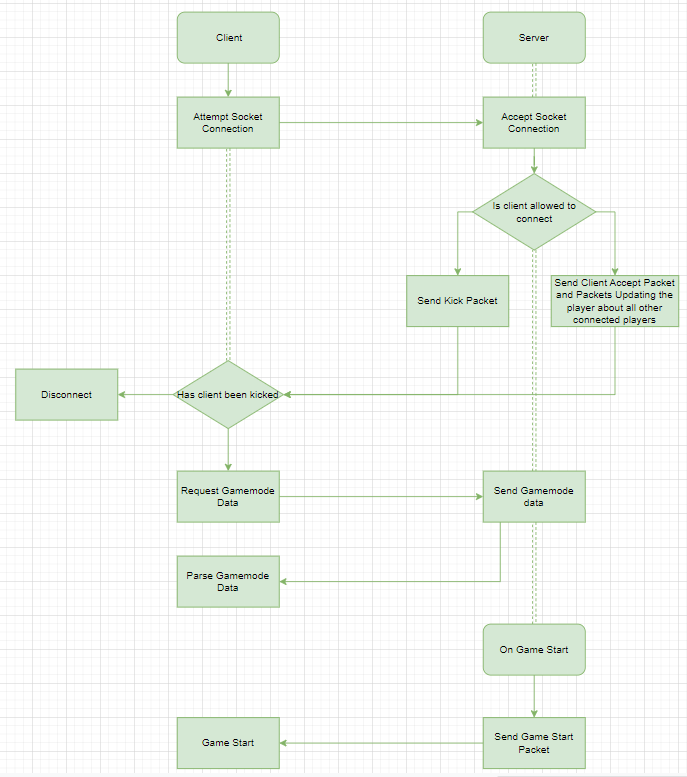
Chess Manager communications in menu (above) vs in-game (below)

### Game Manager

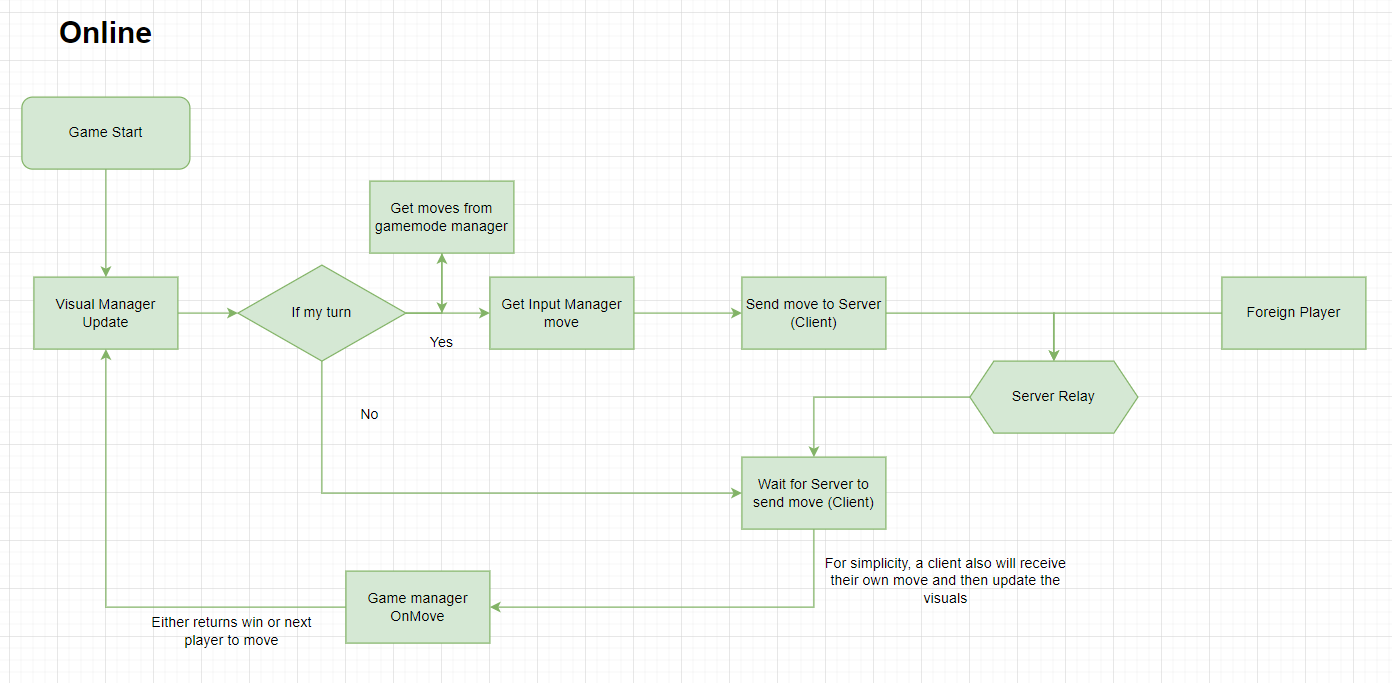


### Pregame Networking Flow

(Dotted lines represent waiting for event)



### In-Game Networking Flow





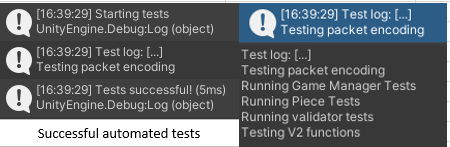
### TODO: UI Flow diagram

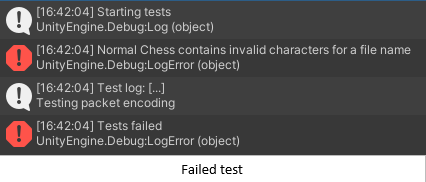
Development

## Automated and manual testing

Excel spreadsheets for testing will be created as the project progresses as the UI, AI and game modes are all subject to change and thus might require different tests.

Testing itself will be carried out through a combination of manual tests and automated tests. These automated tests will be in a class in Unity that is only compiled and ran in the editor where it runs after every code change. It will be excluded from the release build for optimisation purposes.

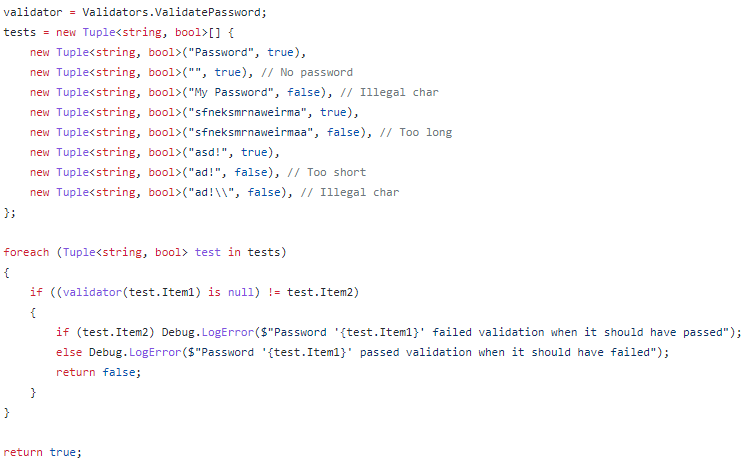






^ Statement ensures testing code isn’t included in a build at all

### Validation Testing

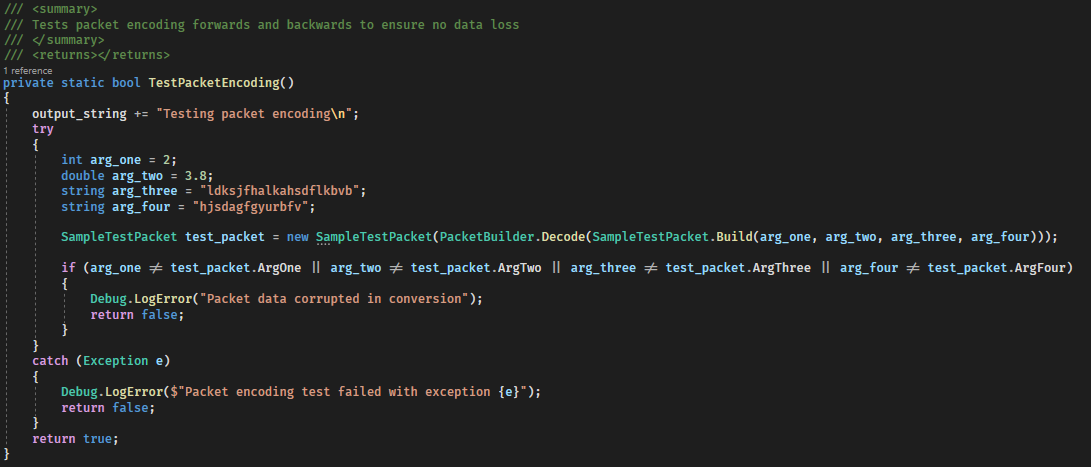


^ Snippet of the code that tests the below validator. Each line within the ‘tests’ array has a string input and a boolean expected outcome that is checked. Each boundary is also tested such as having a max length and a max length + 1 input.



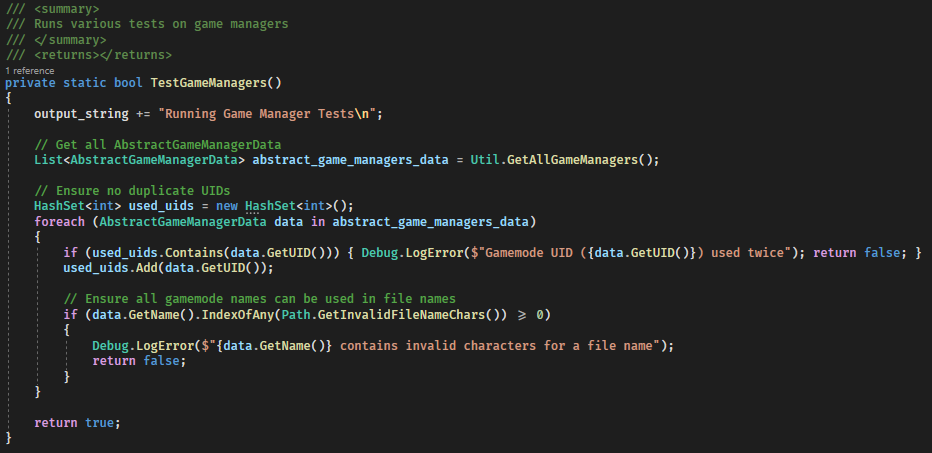
### Encoding-Decoding Testing

To test the networking system, I couldn’t test just encoding or just decoding as the implementation was likely to change later so instead, I checked for data loss or corruption when encoding and decoding data



### UID uniqueness testing

As my game serialisation system relies on unique identifiers for pieces and game manager, my automated tests include tests to ensure there are no identical UIDs



^ Ensuring game managers have unique UIDs. Also contains code to ensure gamemode names can be used in file names.

### Spreadsheet

[](Tests.xlsx)

If above link is broken search for the Tests.xlsx spreadsheet

## Testing with stakeholders

I decided to do frequent tests with shareholders switching shareholder every time to get focused feedback from various points of view. I recorded the feedback I received and the my response to the feedback in a spreadsheet.

### Spreadsheet

[](Feedback.xlsx)

If above link is broken search for the Feedback.xlsx spreadsheet

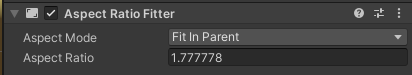
## Version Control

I decided to use GitHub for version control to allow me to rollback changes that broke components, compare code to previous versions and also to serve as a backup and way for testers to download the application. I later started using 2 branches for reasons stated here: Need a saved latest bug-free version to give to testers.

## Developmental challenges encountered and solutions

### Accounting for all display sizes and ratios is difficult

Accounting for all display sizes and ratios was very difficult and time consuming before I found a solution as the display could be portrait or landscape and this could cause, for example, centered and left aligned items to overlap. To solve this, I created a box that is fixed at 16:9 and will scale up to fill as much of the screen as possible. This means that for 16:9 screens there’ll be no change but for 4:3 screens the game will only fill a 16:9 box in it. This could create a letter-boxing effect that could be quite unpleasant however as most backgrounds in my game are black anyway it doesn’t make much of a difference



### Entering play mode is frustrating

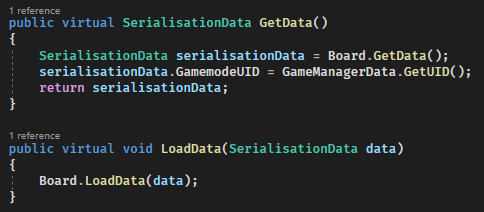
As the main menu initializes some classes, to run the game it must be run from the main menu. This is frustrating as you have to switch scenes, play, stop and switch back for every small change made. To solve this, I created a script that does that for you when you press Ctrl+Q



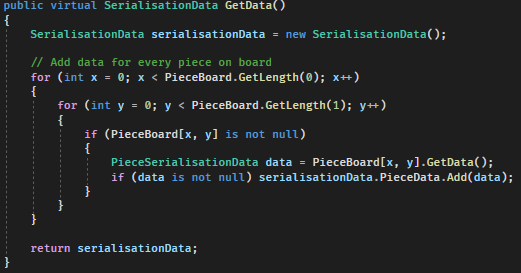
### Save games need to be flexible enough to support every gamemode

I need to be able to save games of various gamemodes and save board configurations which have pieces that may themselves have additional data. I used a combination of polymorphism and UIDs to achieve a robust system. The Gamemode is just stored as a UID with extra data being saved by overriding the ‘GetData’ and ‘Load Data’ methods.

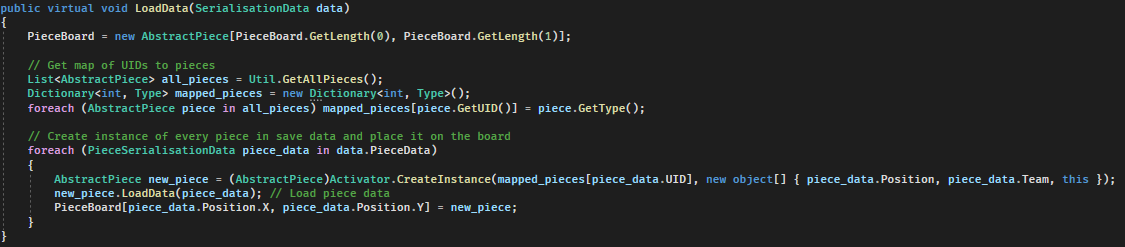
Default implementation for game manager:



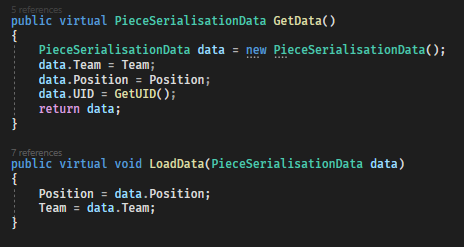
Board.GetData() default implementation:



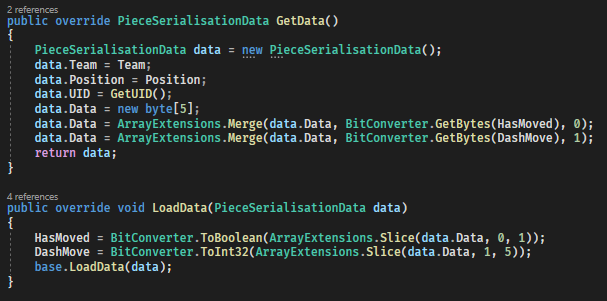
Board.LoadData() default implementation:



Piece default implementation:



An example of custom piece data might look like this (custom data stored in data.Data):



The file is formatted as follows

|  |  |
| --- | --- |
| **Data** | **Length** |
| Length (of full file) | Int – 4 bytes |
| GamemodeUID | Int – 4 bytes |
| TeamTurn | Int – 4 bytes |
| PlayerTurn | Int – 4 bytes |
| Ellapsed time | Long – 8 bytes |
| GameManagerDataLength | Int – 4 bytes |
| GameManagerData | Any type - Any length |
| BoardDataLength | Int – 4 bytes |
| BoardData | Any type - Any length |
| [Repeated for every piece:] |  |
| PieceTeam | Int – 4 bytes |
| PiecePositionX | Int – 4 bytes |
| PiecePositionY | Int – 4 bytes |
| PieceUID | Int – 4 bytes |
| PieceDataLength | Int – 4 bytes |
| PieceData | Any type - Any length |

### The listener socket for servers doesn’t shut down properly

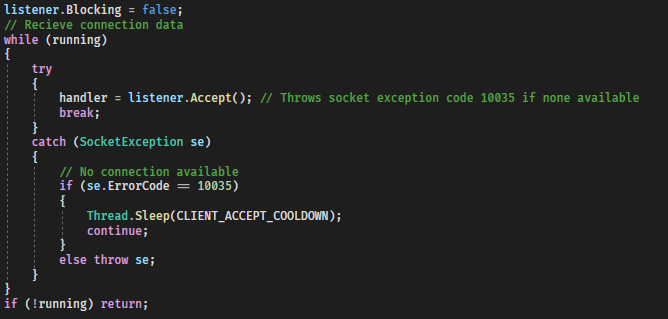
This code waits for a user to connect



However, if the host is shut down there is no way to disconnect this socket without restarting the program, even by throwing an exception. To solve this, I set my sockets to non-blocking



This caused another issue as listener.Accept() now throws an error if there is no client waiting so I wrapped it in a try catch loop



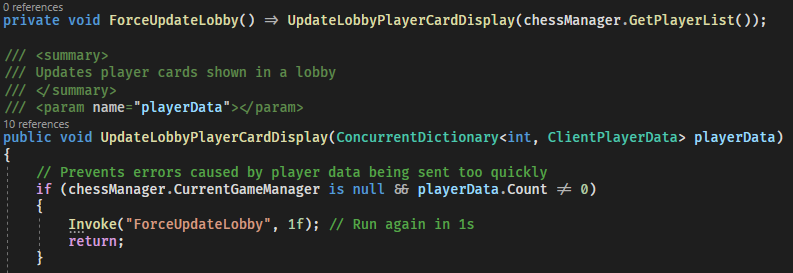
### Need a saved latest bug-free version to give to testers

There were a few occasions where a tester had time to review the game but there wasn’t a stable build available for them to use. To help solve this I used a ‘dev’ and a ‘master’ branch on GitHub with the ‘dev’ branch being used for the latest unstable version and the ‘master’ branch being ready for a tester. When the ‘dev’ branch reached a milestone like a feature being completed it would be merged into the ‘master’ branch.



### A ‘ghost’ user will sometimes appear in the player list when joining lobbies

This ‘ghost’ user didn’t affect the game at all however it was confusing for players. Through repeated testing with different conditions, I found that this happened when a player had a team assigned and then another player joined. The player information and the player team information would be sent in too rapid succession creating two players in the list. To remedy this, I made the UI wait until the information was ready before displaying it.



(The Invoke method runs a method after a set delay)

### Trying to host twice in quick succession creates errors

This was found to be due to the socket used for the server being in ‘TIME\_WAIT’ state. To remedy this (as everywhere I looked online, I couldn’t find a way to avoid this) I added a message asking the user to wait if the socket was still in that state





### The AI doesn’t pick the best move

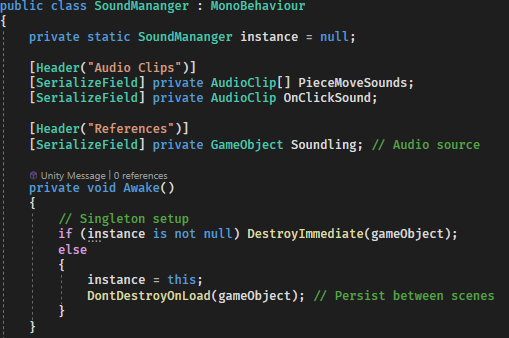
This was found to be due to the AI misidentifying some wins as bad and some losses as good

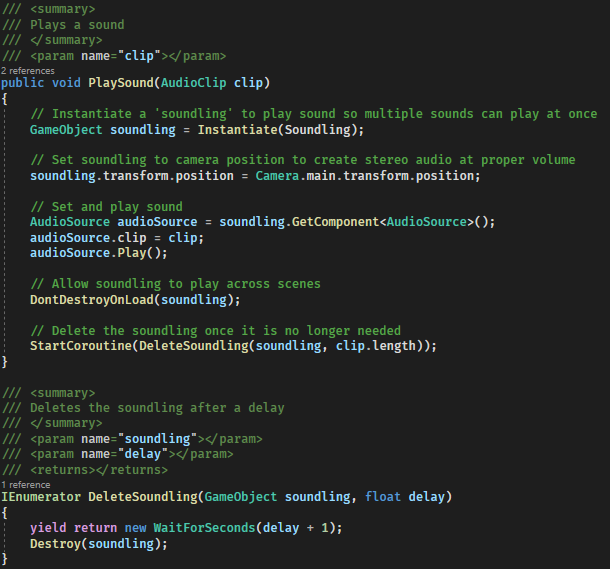


The else return lines had opposite values to what they have now sometimes causing a win to be seen as bad

### Developing the sound system

Originally, I was going to use a Unity ‘AudioSource’ to play sounds however this had two problems: it would be difficult to reference across scenes and it can only play one sound at a time. Instead, I used the Singleton design pattern to allow easy static referencing across scenes despite Unity not supporting static classes attached to GameObjects. This Singleton then creates an AudioSource that deletes itself after the sound has finished playing allowing multiple sounds to play at once. These also persist across scenes.





### AI Freezing

The AI sometimes froze for seemingly no reason after using Visual Studio’s built-in debugger and conditional breakpoints I found that this line of code wasn’t working



I use float.NaN (not a number) to stop the AI when it is out of time and this NaN propagates up the recursive MiniMax algorithm stopping it





The reason for this not working is that

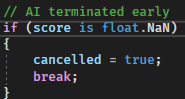


Which I found using Visual Studio’s interactive C# console

The reason behind this decision by the designers of the C# language is that 1 / 0 should not equal 2 / 0

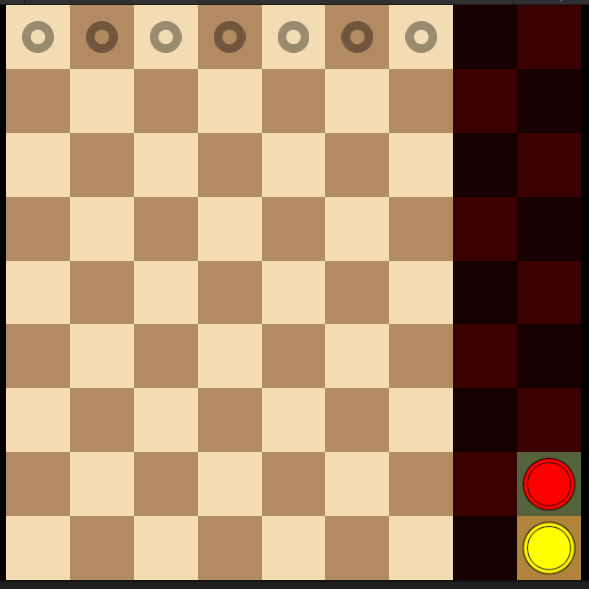
C# does have a different comparison operator ‘is’ which works here





### Connect 4 doesn’t fit the input system

As the input system is based around clicking on a piece and then where to move it too it seems impossible to implement connect four as that requires taking counter from off the board and putting them on however, I figured out a way to do this utilizing the blocked square I implemented for a different gamemode



By clicking on a piece and then a column you can emulate picking a piece up from off the board

### Connecting through a public IP stopped working

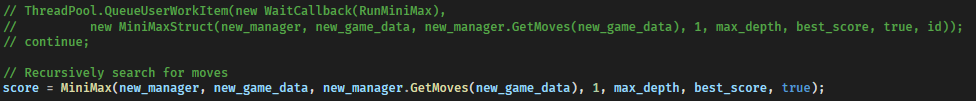
As I usually only have one computer available for development, I can’t frequently test connections through public IPs. They should behave in the same way as connecting through local IPs (except for port forwarding) so I continued development without frequently testing it. In a test with a stakeholder, I found public IPs to not be working anymore. To solve this, I used my phone and Unity’s cross platform support to test the public IP functionality as the devices have to be on different LANs and I could use cellular data to achieve that. I then went through previous versions available through GitHub and found that between November 22nd and December 5th it had stopped working. Analyzing code differences, I found this line to be the problem:



I believed this line allowed an address to reused which I wanted for the server. What this actually does it allow a socket to be bound to a port that is already in use which I don’t want to happen.

### Using multithreading for the MiniMax algorithm

I initially believed that multithreading wouldn’t improve the AIs performance as the Alpha-Beta pruning (see: AI System (MiniMax Algorithm)) relies on branches of the tree being explored sequentially and would get the largest optimisations from pruning branches from the first layer which I would prevent it from doing by multithreading. My initial assumption turned out to be correct as after implementing multithreading using C#s ‘ThreadPool’ I noted no increase in AI exploration depth. I did, however notice an increase in CPU usage which meant that while it didn’t hurt performance on my system which has a strong CPU, it might have been able to on other systems so I removed the feature.



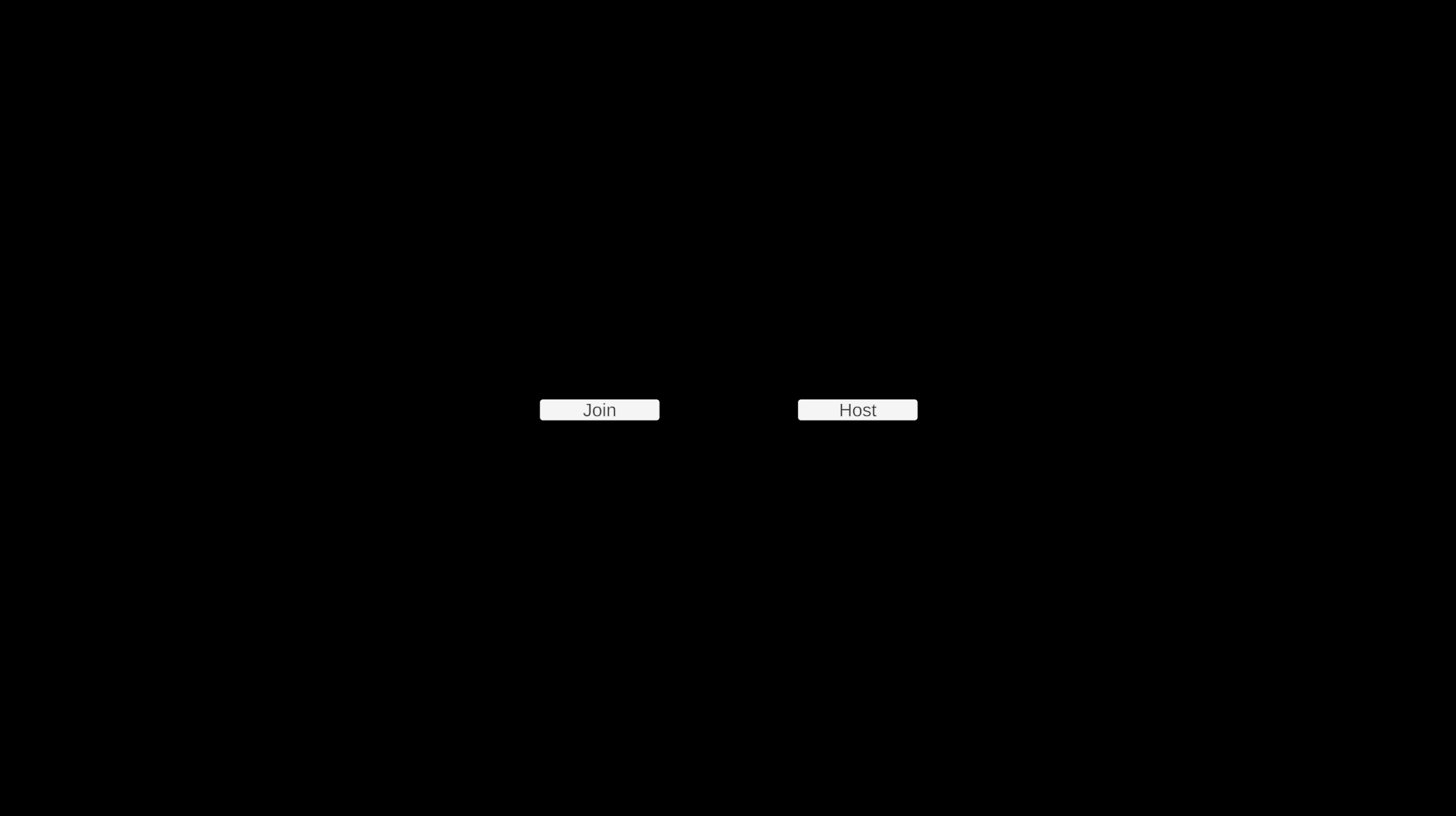
^ Multithreaded (commented) and single threaded code

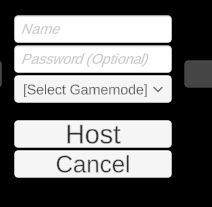
### Missing chess AI optimization due to generalisation

In my research on chess AI optimisations, I found many techniques that I cannot apply to this game due to generalisation. One example of this is the many forms of hashing. This is very difficult to do as hashing relies on a fixed amount of data that can uniquely identify a board position so that if it is seen again, it isn’t re-evaluated. There are various ways to generate these for chess however for this to work with multiple gamemodes I would need to create a custom hashing implementation per gamemode. I considered using the save game system however despite it being more than efficient enough for saving and loading, it is far too large for use in fast dictionary as a hash.

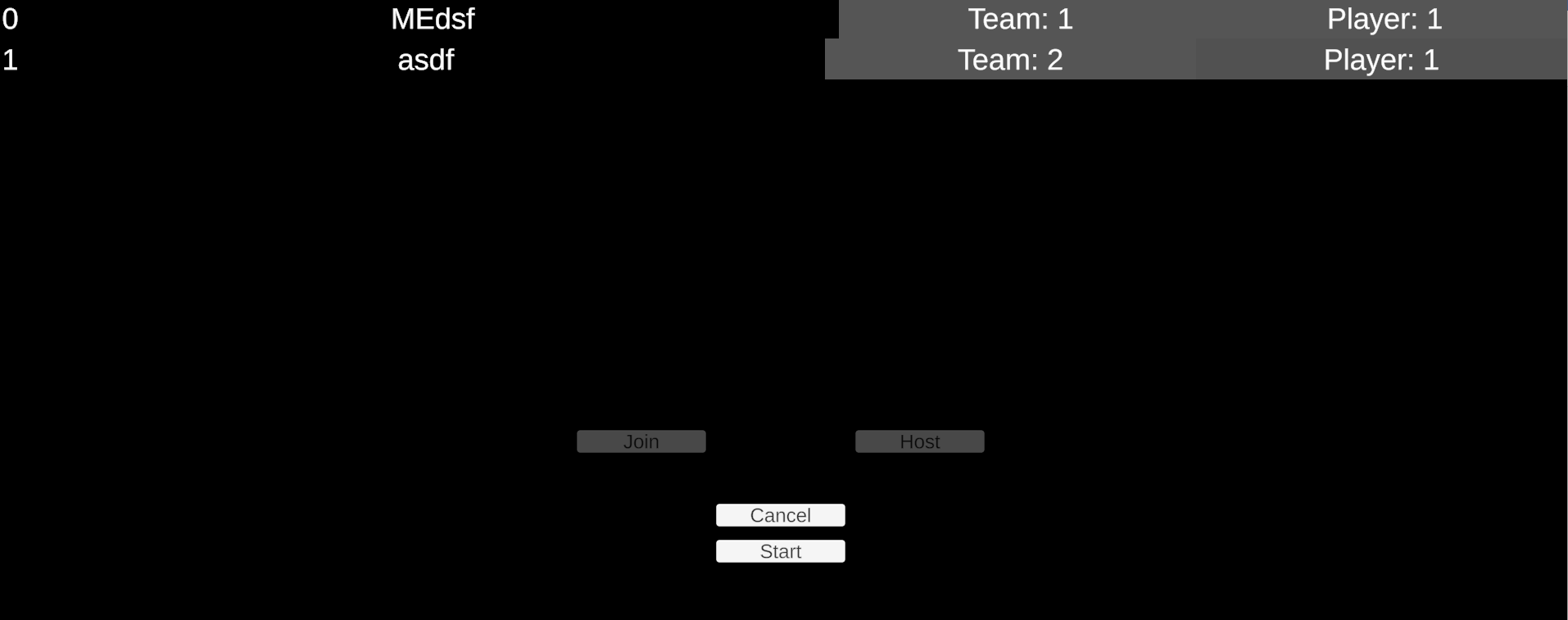
## Versions during development

### First version with UI

Main menu

 <- Host screen | Join screen -> 

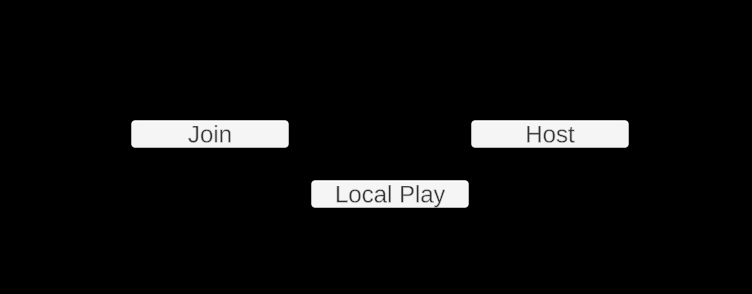
Lobby UI



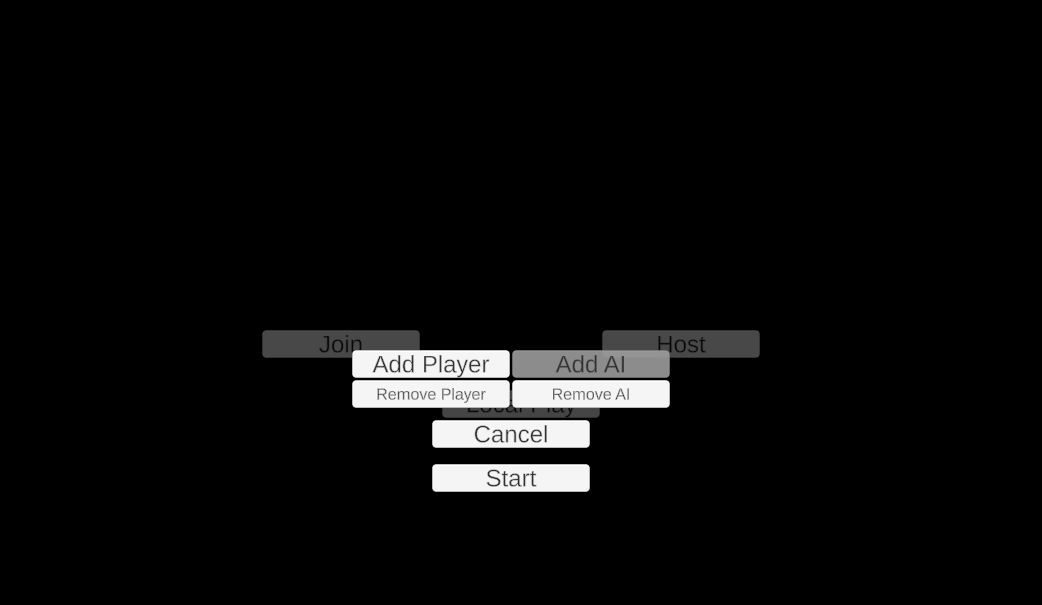
In game



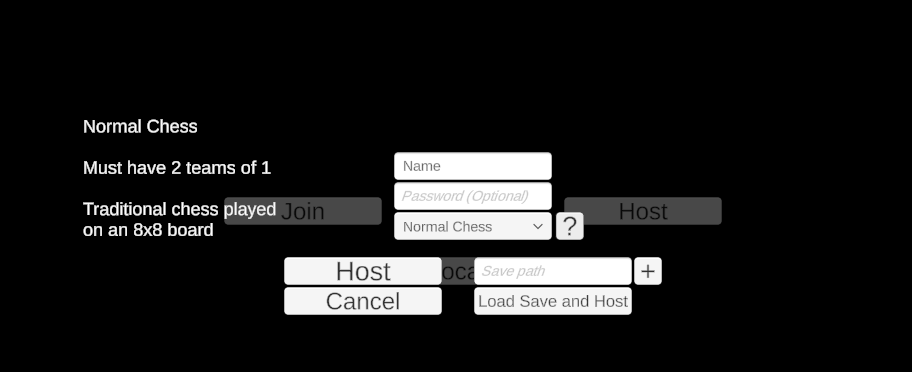
### First version with help system and save system

Main menu (local play not currently functional)

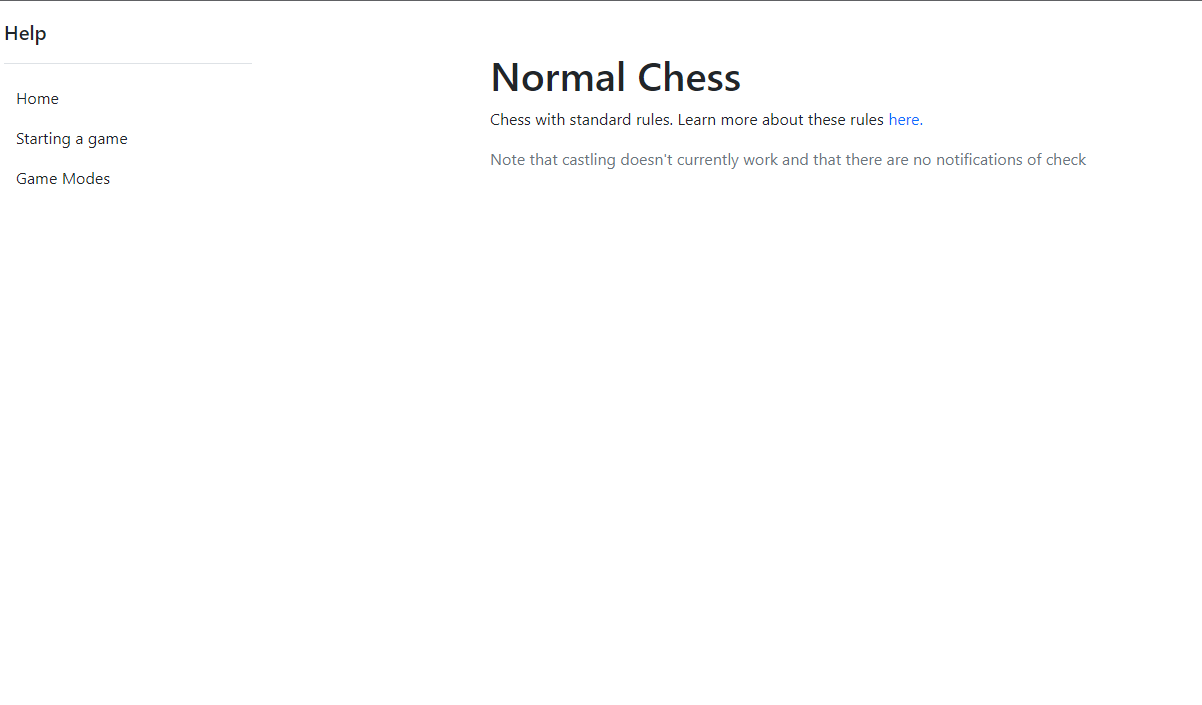
Local play screen (not functional)



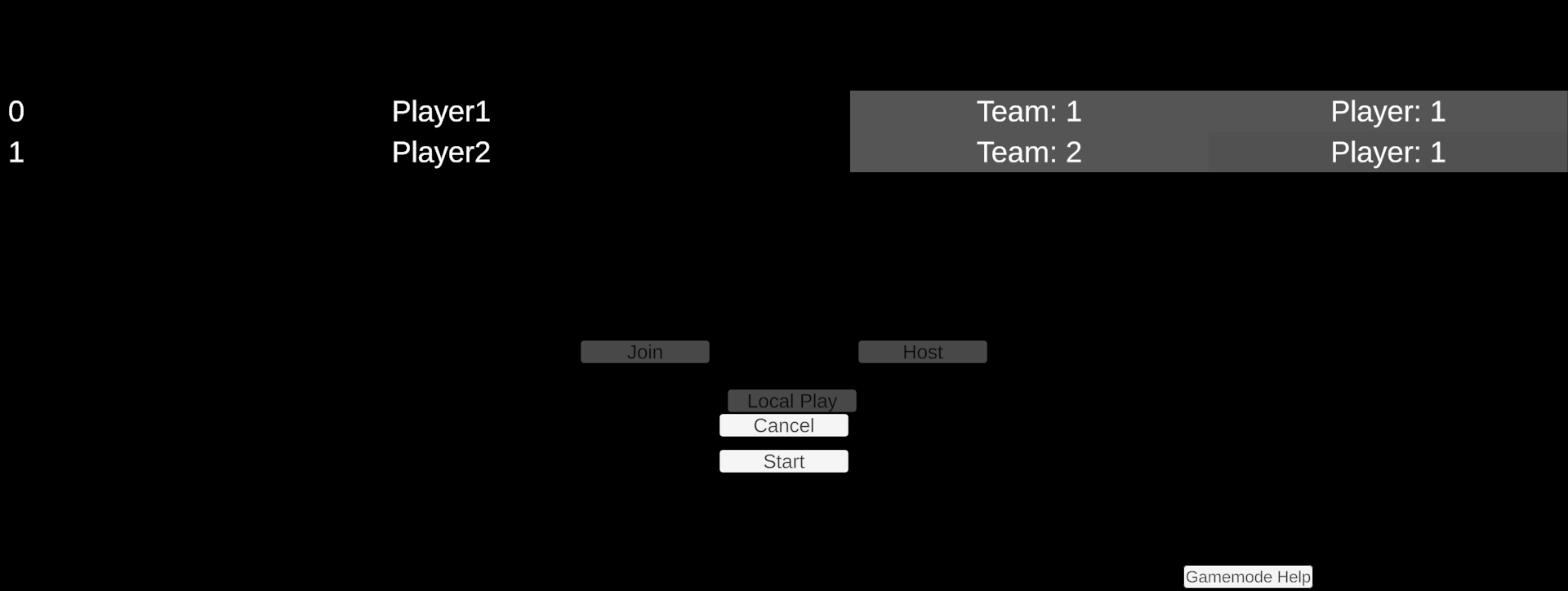
Host screen (new help button and load save option)



Help Screen



Lobby screen (with adjusted layout and gamemode help button)



In game (new timer and turn indicator) 

New escape menu (missing quit functionality)

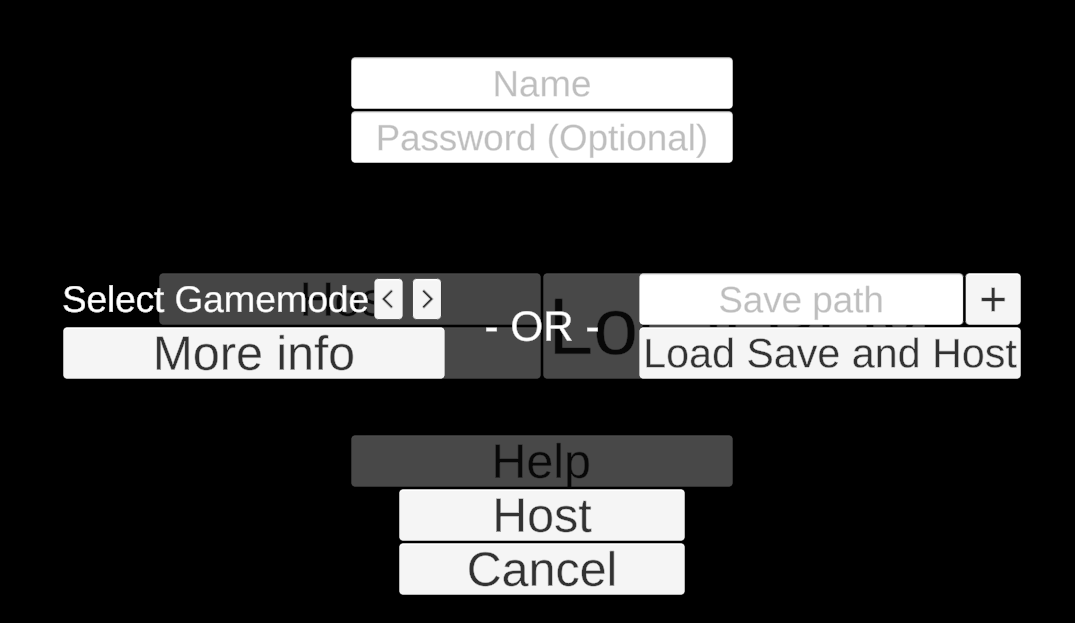


### Large UI improvements, functional local play and AI added

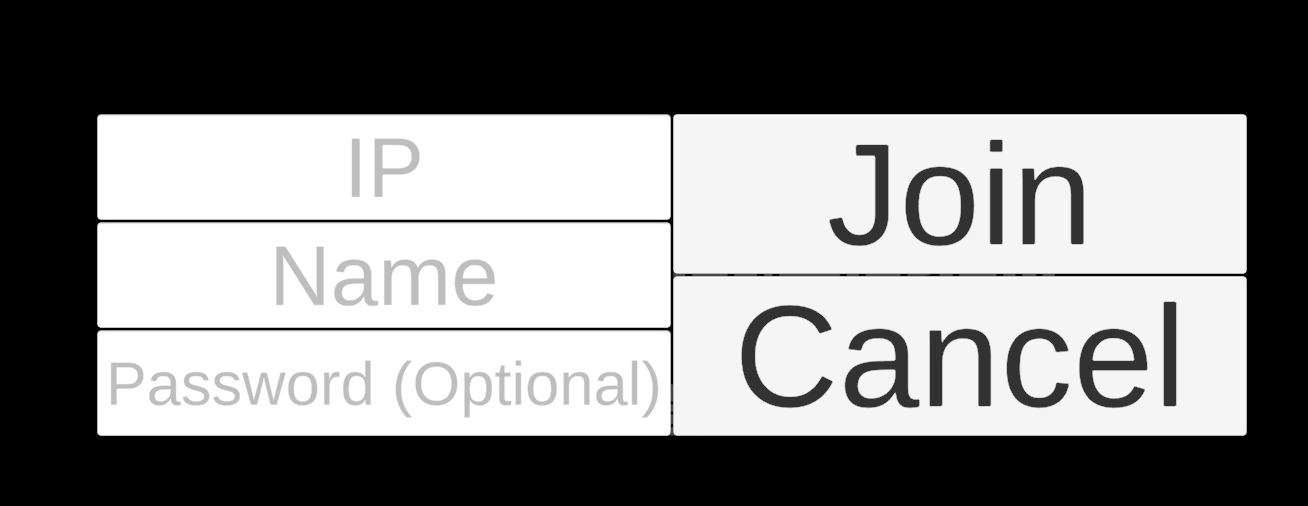
Main menu



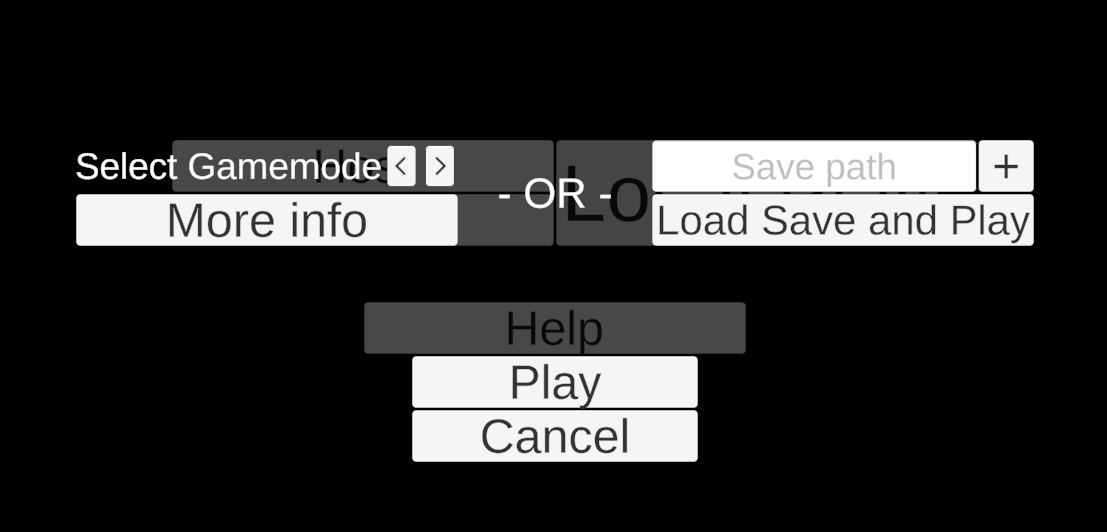
Host screen



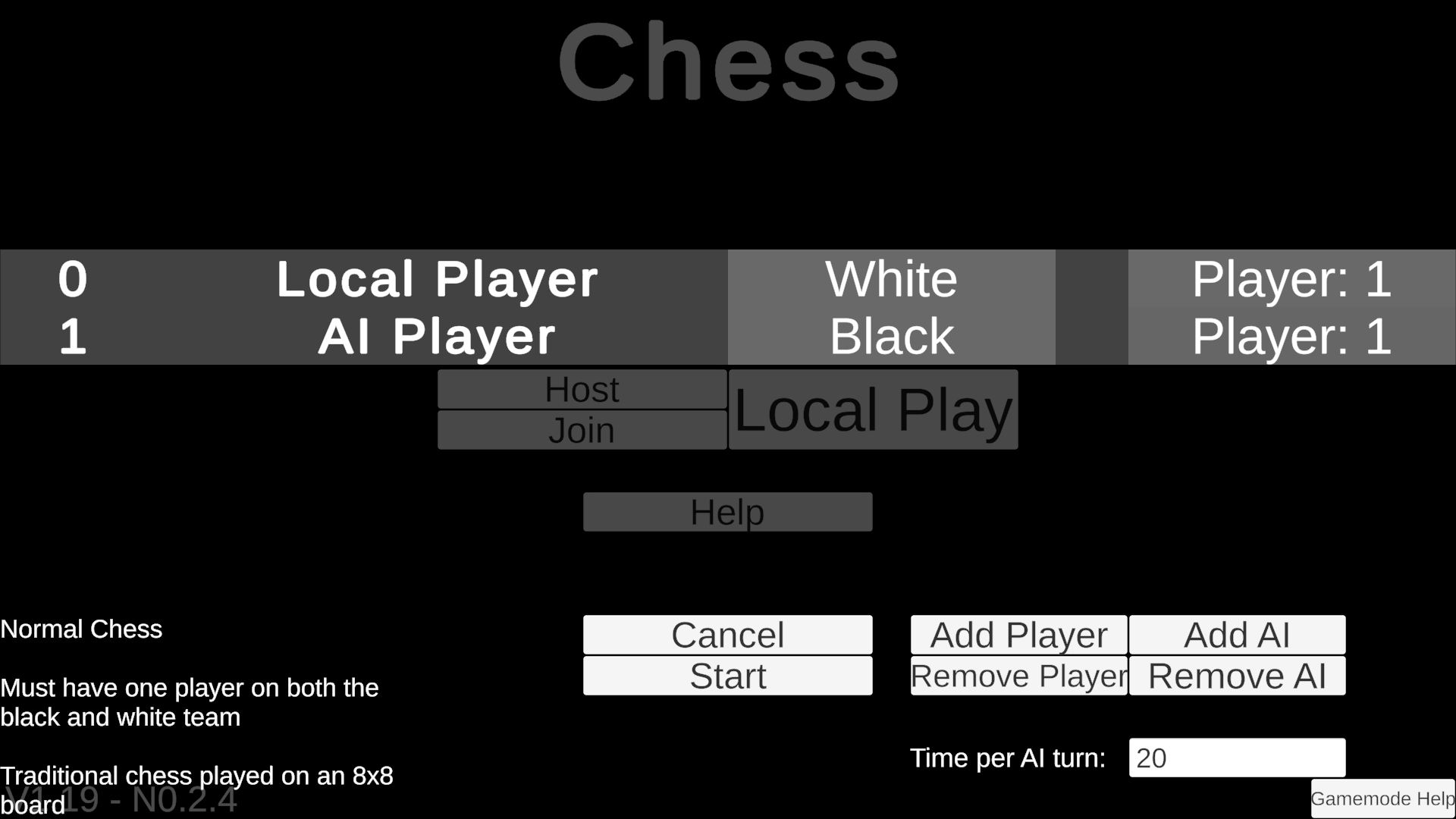
Join screen



Local play screen



Lobby screen



New escape menu scaling

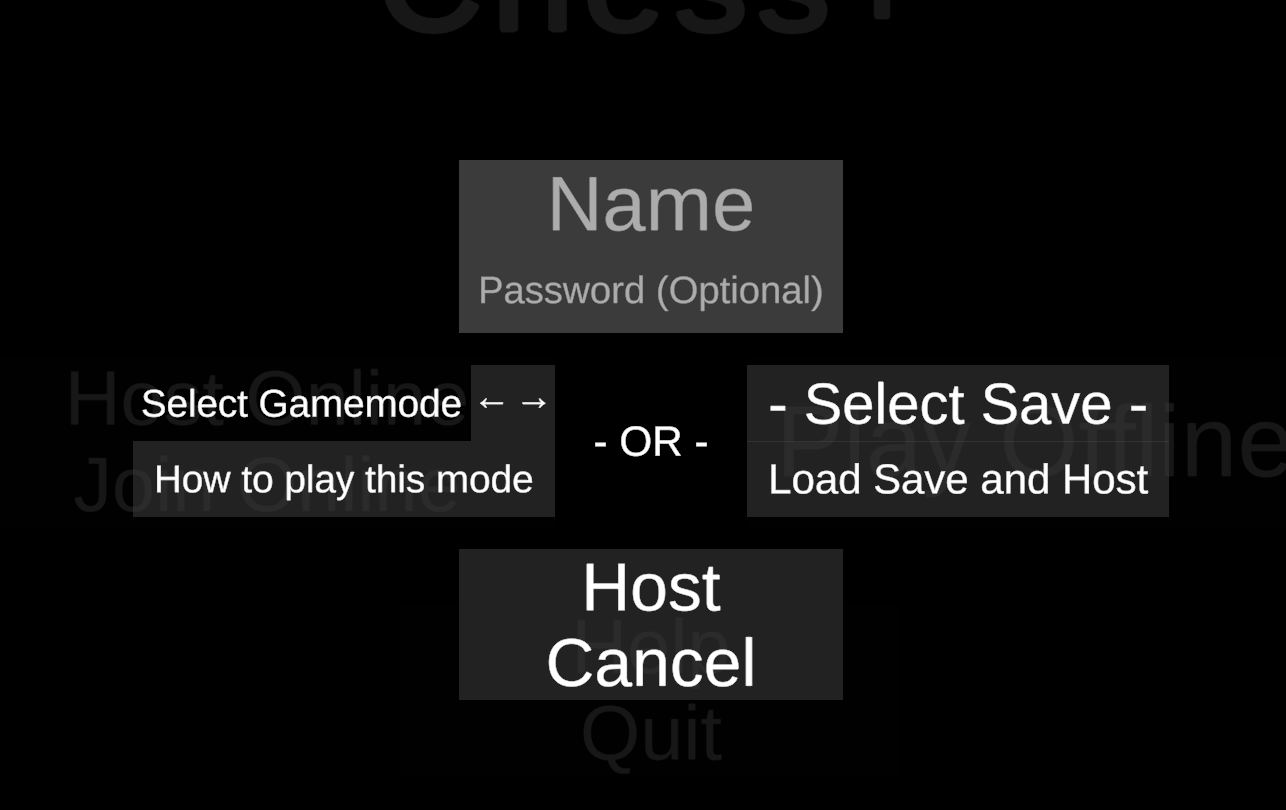


### Large UI overhaul

Main menu



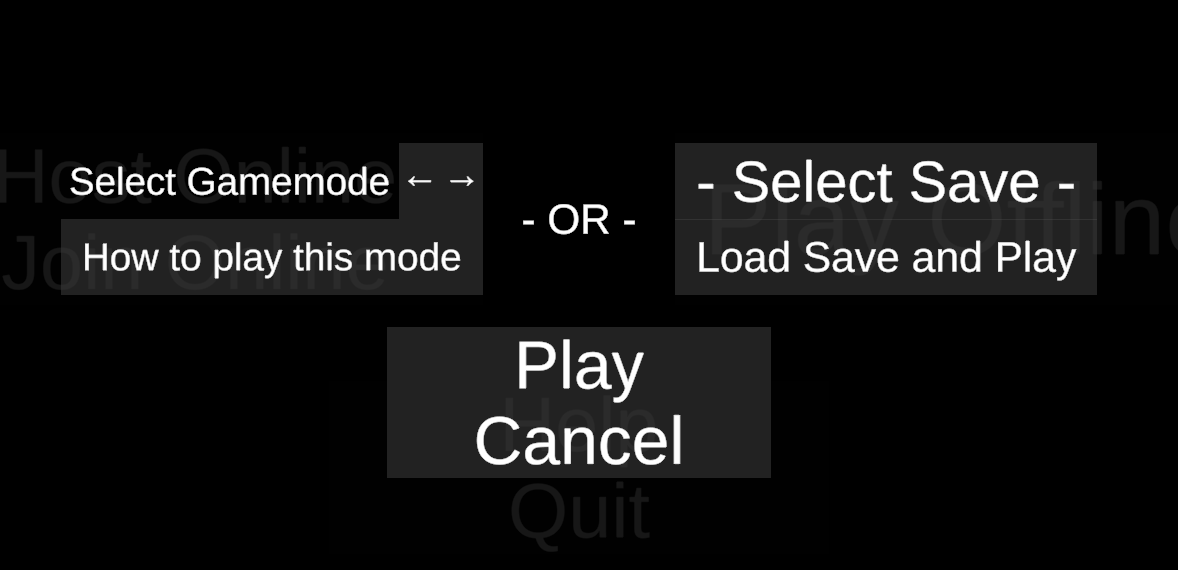
Host screen



Join screen



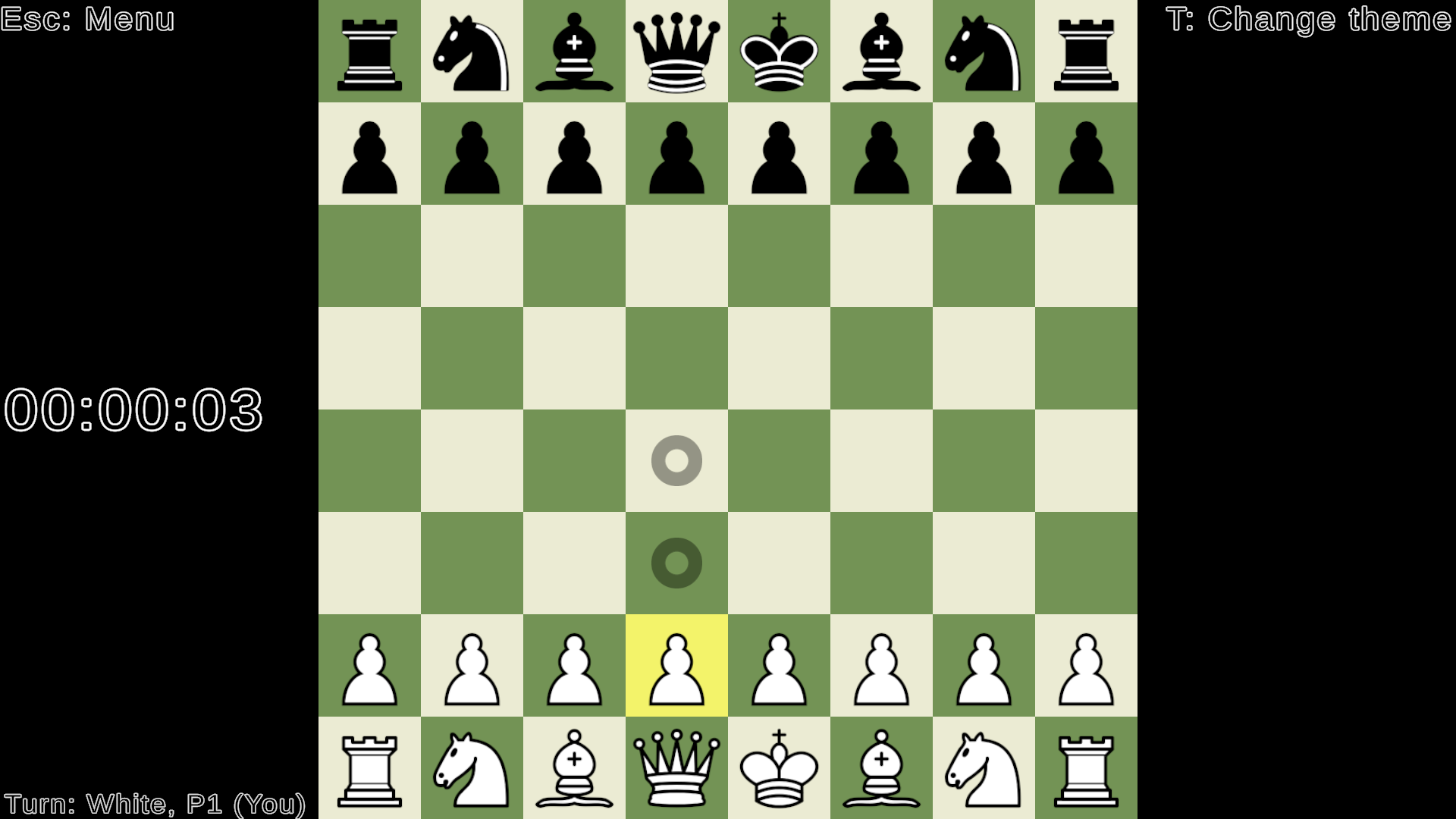
Local play screen



Lobby



In Game



Multiple themes



Escape menu



Solution

## Overview

The ChessManager will encapsulate functions of other classes and handle communication and coordination between different systems (listed below). The idea is to have mostly a tree (or star) topology (where classes can only communicate with those directly above or below them) to make the flow of data between classes easier to follow

## TODO: Game Mode System

This game will support many different game modes. This will be achieved using OOP techniques, namely polymorphism.

### GameManagerData

Uninitialized form of GameData used before full instance is needed e.g. in main menu when getting a list of all game modes

|  |  |  |
| --- | --- | --- |
| GameManagerData | Description | Type |
| **Data** | None |  |
| **Methods** |  |  |
| GetUID | Returns a unique ID for the game mode | Int |
| GetName | Returns the name of the game mode | String |
| GetTeamSizes | Returns the team sizes for this game | TeamSize[] |
| TeamAliases | Returns names for the teams e.g. { “White”, “Black” } | String[] |
| GetDescription | Returns a short description of the game mode (mainly used as a backup in case the help system doesn’t work) | String |
| Instantiate | Returns an instance of the full GameManager class | AbstractGameManager |

### GameManager

Controls a gamemode. Classes inheriting from GameManager might also implement additional data and methods

|  |  |  |
| --- | --- | --- |
| GameManager | Description | Type |
| **Data** |  |  |
| GameManagerData | A reference to the ‘Data’ version of this class | GameManagerData |
| Board | A reference to the Board class of this game mode – see below | AbstractBoard |
| **Methods** |  |  |
| GetData | Returns a serialised version of the game – ready to be saved | SerialisationData |
| LoadData | Loads serialised data | Void |
| GetMoves | Returns a list of available moves to make | List<Move> |
| OnNoMoves | Handles what happens if no moves are available. Usually just makes the opposing team win | Void |
| OnMove | Applies a selected move. Returns either the next player to play’s ID or the winning team ID in the form –(TeamID + 1). If the returned value is less than 0, a team has won. | Int |
| GetScore | Returns a score for the current position. Used by the AI to determine how good a position is | Float |
| Clone | Returns a clone of the game. Used by the AI to test different moves | AbstractGameManager |

### Board

|  |  |  |
| --- | --- | --- |
| Board | Description | Type |
| **Data** |  |  |
| PieceBoard | 2D piece array representing the positions of pieces currently on the board | PieceBoard[,] |
| **Methods** |  |  |
| GetData | Returns a serialised version of the board | SerialisationData |
| LoadData | Loads serialised data | Void |
| GetMoves | Returns a list of available moves | List<Move> |
| GetPiece | Returns a piece at a position or null if there is none there | AbstractPiece |
| GetBoardRenderInfo | Returns specifications for the board e.g. size, highlighted squares and removed squares | BoardRenderInfo |
| OnMove | Applies a selected move | Void |
| GetScore | Returns a score for the current board. Used by the AI to determine how good a position is | Float |
| Clone | Returns a clone of the board | AbstractBoard |

### Piece

|  |  |  |
| --- | --- | --- |
| Piece | Description | Type |
| **Data** |  |  |
| Position | The piece’s current position | V2 |
| AppearanceID | The piece’s current appearance | Int |
| Team | The team the piece belongs to | Int |
| Board | The board the piece is on | AbstractBoard |
| **Methods** |  |  |
| GetData | Returns a serialised version of the piece | PieceSerialisationData |
| LoadData | Loads serialised data | Void |
| GetMoves | Returns a list of available moves | List<Move> |
| OnMove | Applies a selected move | Void |
| GetValue | Returns the value of the piece. Used by the AI to determine how good a position is | Float |
| Clone | Returns a clone of the piece | AbstractPiece |

## Networking System

Handles all online communications

### Network Manager

|  |  |  |
| --- | --- | --- |
| NetworkManager | Description | Type |
| **Data** |  |  |
| chessManager | Reference to the ChessManager | ChessManager |
| server | Reference to the Server, if there is one | Server |
| client | Reference to the Client, if there is one | Client |
| **Methods** |  |  |
| Awake | Initialises NetworkManager | Void |
| OnApplicationQuit | Shuts down the NetworkManager when Unity exits | Void |
| OnLocalMove | Passes a move to the Client to be sent to other players | Void |
| GetPlayerList | Gets the list of all players from the client | ConcurrentDictionary<int, ClientPlayerData> |
| GetLocalPlayerID | Returns the ID of the local player | Int |
| Host | Hosts a game | Void |
| HostStartGame | Starts a hosted game | Void |
| Join | Joins a game | Void |
| ConnectionFailed | Handles what should happen on a connection failure | Void |
| GetPing | Calls the provided action with the ping as a parameter | Void |
| Stop | Shuts down the server and client if either are running | Void |
| **Client Callbacks** | Methods that directly map to methods of other classes used to maintain an understandable reference tree. | All Void |

### Client and Server

These bare many similarities and both function by having a receive thread that reads incoming messages one at a time and a send thread that sends messages one at a time (the server also has a thread for accepting new clients). The server primarily acts as a relay passing on messages from one client to all other clients such as a move update. See Pregame Networking Flow diagram for a representation of what happens in the lobby and In-Game Networking Flow for a representation of what happens during a game.

### Packet System

Every unit of data sent between the server and client is a ‘packet’ formatted in a specific way

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Description | Type | Length |
| PacketLen | Length of the entire packet | Int | 4 bytes |
| UID | Unique identifier for this type of packet | Int | 4 bytes |
| Content block 1 length | Length of the first block of content | Int | 4 bytes |
| Content block 1 | Any | Type according to packet type | Content block 1 length bytes |
| [Repeat last two for all content] |  |  |  |



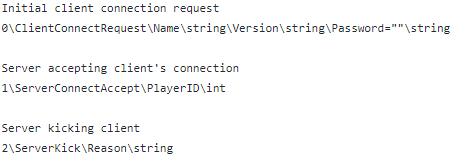
Packets, such as the one above, are automatically generated from a text file of requirements in conjunction with the ‘PacketBuilder’ that converts these to a byte array ready to send and back from received bytes allow the developer to never have to interact with raw bytes for sending and receiving data as usage of the system looks like this:



^ Encoding



^ Decoding

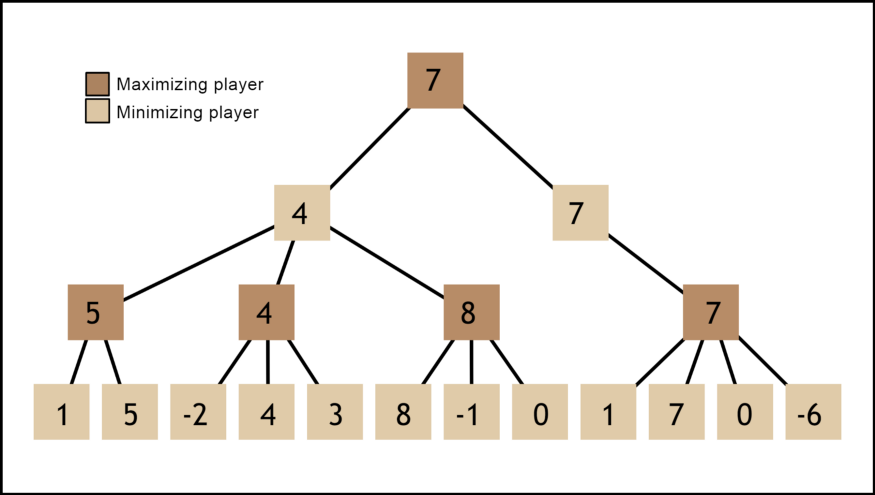


^ Packet generating file (Any line without a ‘\’ is a comment)

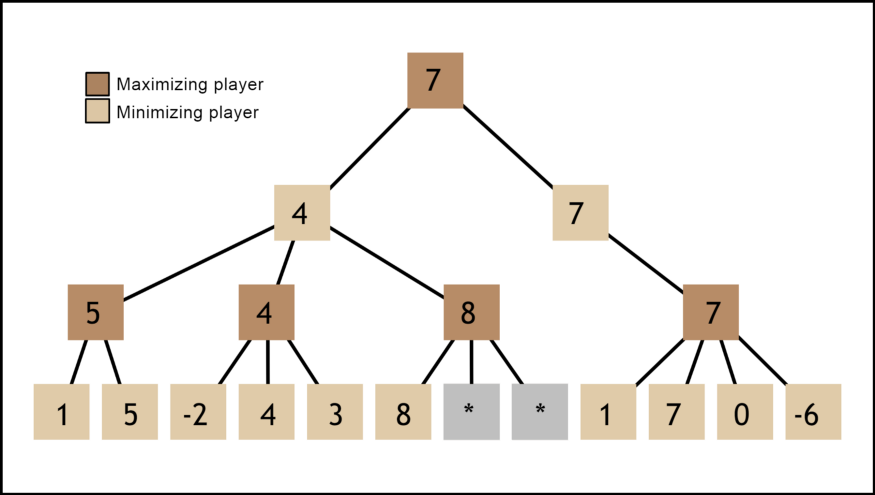
## AI System (MiniMax Algorithm)

The AI will use the minimax algorithm to calculate the best move and use Alpha-Beta pruning for optimisation. The AI will be given a time limit and try to look as many moves ahead as it can in that time frame. To allow this to work with all game modes, game modes must implement a cloning method and an evaluation method that returns a score for the current board.

The minimax algorithm works by performing an in-order traversal of all possible game states up to a set number of moves ahead with each node being a state and each branch a move. It then works backwards using a score for each leaf node assuming that the opponent will try to minimise the score and the AI will try to maximise it. This results in the AI maximising from the layer directly below the root node and the branch it picks will decide which move it makes.



^ Diagram explaining the minimax algorithm. Source (for this and next image): <https://medium.com/@SereneBiologist/the-anatomy-of-a-chess-ai-2087d0d565>



C

B

A

^ Diagram showing AB pruning.

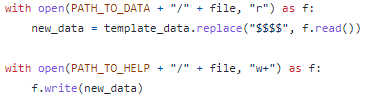
After node A receives the value 4 from node B, it moves on to node C. As C is maximising, once C gets its first value of 8, it doesn’t need to check any more children as we know it will return at least 8 and as A is minimising, A won’t take its value. In some scenarios AB Pruning can reduce end nodes searched by 99.8%

I considered using multithreading to improve performance but didn’t for reasons found here: Using multithreading for the MiniMax algorithm

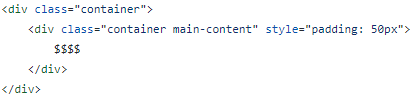
## Help System

The help system will use local static HTML and CSS files to show help information in the default browser. This allows rapid development and iteration on help menus with support for various layouts and images. Developing a help / tutorial system in Unity would be very slow due to the more manual approach to element scaling and positioning.

To create the static HTML pages a python program will take a template.html file and combine it with other HTML files to keep a consistent sidebar and style between pages and allow changes to this style to automatically propagate through all pages.



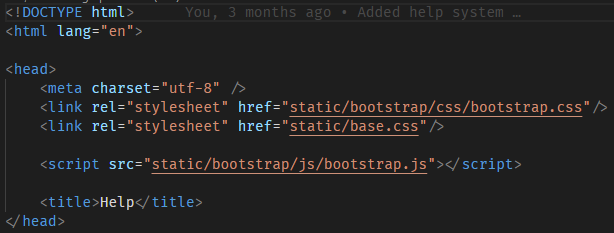
^ Code creating a new file based on the template.



^ Area where file is inserted into template

As the website is completely local, there is no web hosting requirement and no time waiting for a network connection

It uses the Bootstrap CSS library to create intuitive pages that scales well with screen sizes



^ Bootstrap inclusion



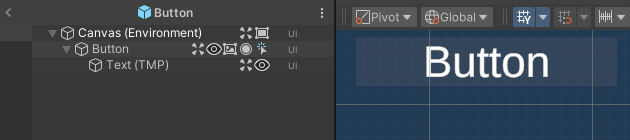
^ Using Bootstrap classes ‘container’ and ‘text-muted’



^ Help page for Atomic Chess

## UI System

The UI will be made of standardised components to ensure a consistent style and will use Unity’s prefab system allowing me to make changes to, for example, a button prefab that will propagate to all the buttons.



^ Any change made to the design of this button will change the designs of all buttons

The UI also uses a fixed aspect ratio as detailed here: Accounting for all display sizes and ratios is difficult

### TODO: VisualManager system

## Validation System

Username selection – Usernames must be displayed for all other users and thus must be validated to ensure they are readable, won’t interfere with other UI and aren’t too long which could cause UI bugs or even slow down networking. This will probably be done with a list of allowed characters and a minimum and maximum name length.

Password selection – This is mostly the same as username selection however this won’t need to be displayed and must support more character to allow strong passwords. There will be some character restriction to ensure characters used are available on most keyboards as passwords need to be entered by other users.

Team composition – Team compositions must be validated before a game is started as gamemodes have teams and expected player numbers. For example, chess wouldn’t work if there was only one team or two people on one team. The gamemode will provide the team sizes.

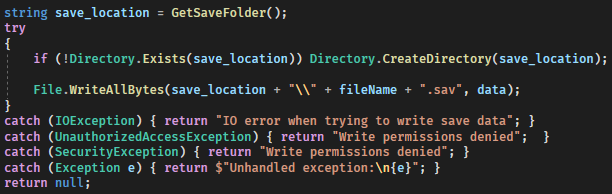
This validation system is tested by automated tests. See: Validation Testing

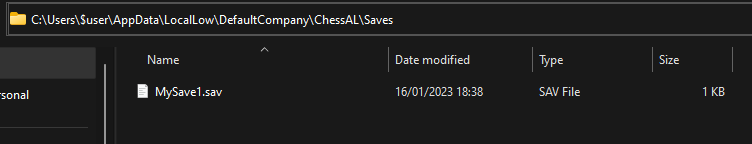
## Save system

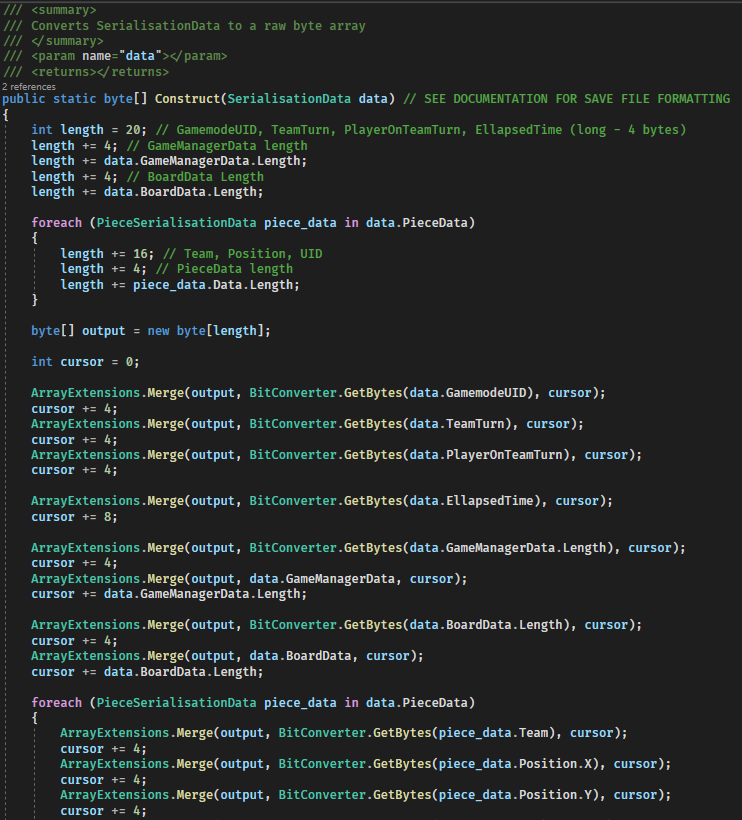
Saves are stored in a proprietary binary format. See: Save games need to be flexible enough to support every gamemode

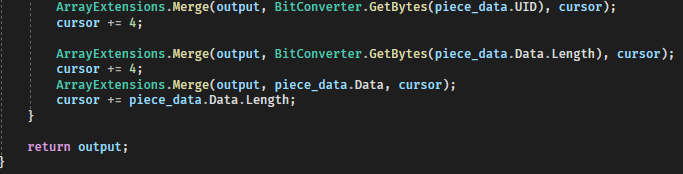
These are stored in the application’s ‘persistent data path’



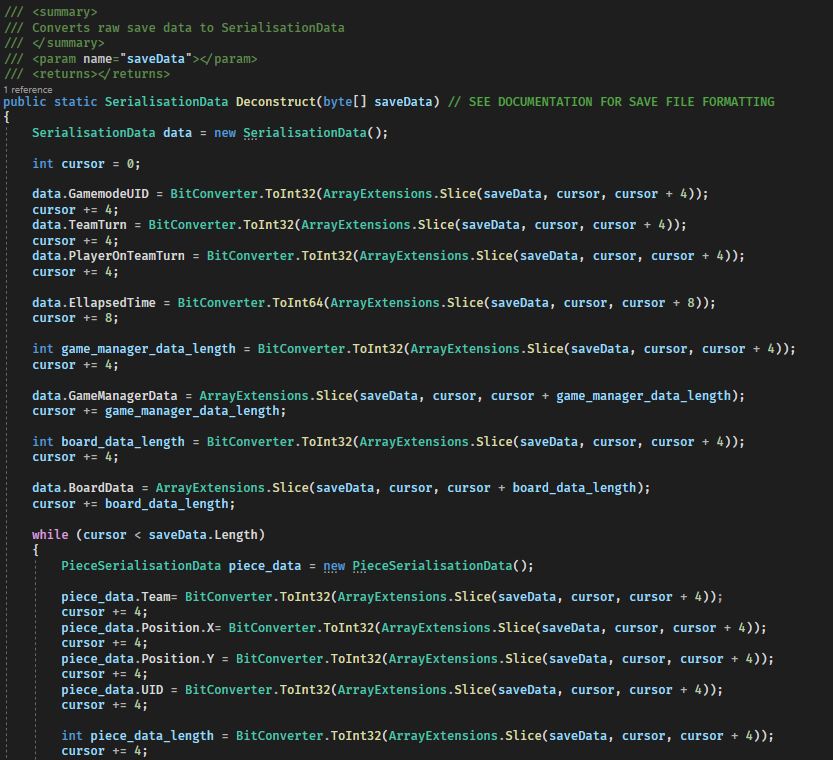


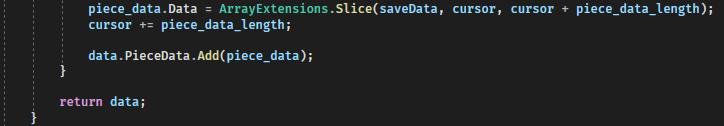


Serialisation:  




Deserialisation:





### TODO: Use of Unity’s PlayerPrefs system

Evaluation

## TODO: Success compared to initial goals

## TODO: Final Feedback

## TODO: Future Improvements

### TODO: Using a relay server or centralised server

### TODO: AI Improvements

### TODO: Art Improvements

### TODO: UI Improvements

### TODO: Additional gamemodes

Code and Screenshots

## Screenshots

## Code