Chess+

Robert Lucas

OCR Computer Science Project

Table of Contents

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 and play all of them with AI. This will make the game more appealing to a casual audience who want to experience unique variations of chess, be able to play chess (or other board games) 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 against friends or AI 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.

I'm expecting the players to be of all ages however especially young players will be expected to get an older person to help them if they can't figure out how to play the game.

Stakeholder Questionnaire

A questionnaire is very important as interacting with the stakeholders is the only way you can ensure that you are creating a product that the stakeholders want and will enjoy.

Key Points from Responses to the Questionnaire (12 people)

Do you have a Windows device?

- Almost everyone asked (11) 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?

- 1/3 of the people aked 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 (9) 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 and Pattern Recognition

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
- Each game mode's code will interact with other systems in a predetermined way meaning that no new rendering, input, network, etc. code needs to be written for a new game mode to be added

Pattern recognition is important as identifying similarities between components can allow code reuse, reducing development time and can create better interfaces between components

- All game modes, boards and pieces will inherit from abstract versions of themselves that will have default implementations for many methods. This will allow code to be reused unless a game mode needs some custom behaviour
- Game modes, pieces and boards can inherit from each other allowing for even more code reuse

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

Object / Functional Orientation

C# is an object-orientated language and I will be using OOP features heavily. The game mode system will be entirely based on inheritance and polymorphism to provide a way for other components to interface with any game mode. It will also allow game modes to have default method implementations and allow game modes to inherit from each other to increase code reuse.

C# does support some functional programming paradigms which I will be using such as functions being treated as first-class citizens meaning that they can be passed as arguments to methods and stored under a name like other objects.

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 thread concurrent 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 Chess Manager 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 when it encounters a game-loss state

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

Advanced Programming Techniques Used

- Multi-dimensional arrays used for representing boards
- Complex algorithm – MiniMax with alpha-beta pruning
- Custom binary serialisation – saving game states
- Multithreading – used for networking and AI to not block the main thread and execute tasks concurrently
- Networking with sockets

- Use of OOP techniques such as polymorphism and inheritance

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

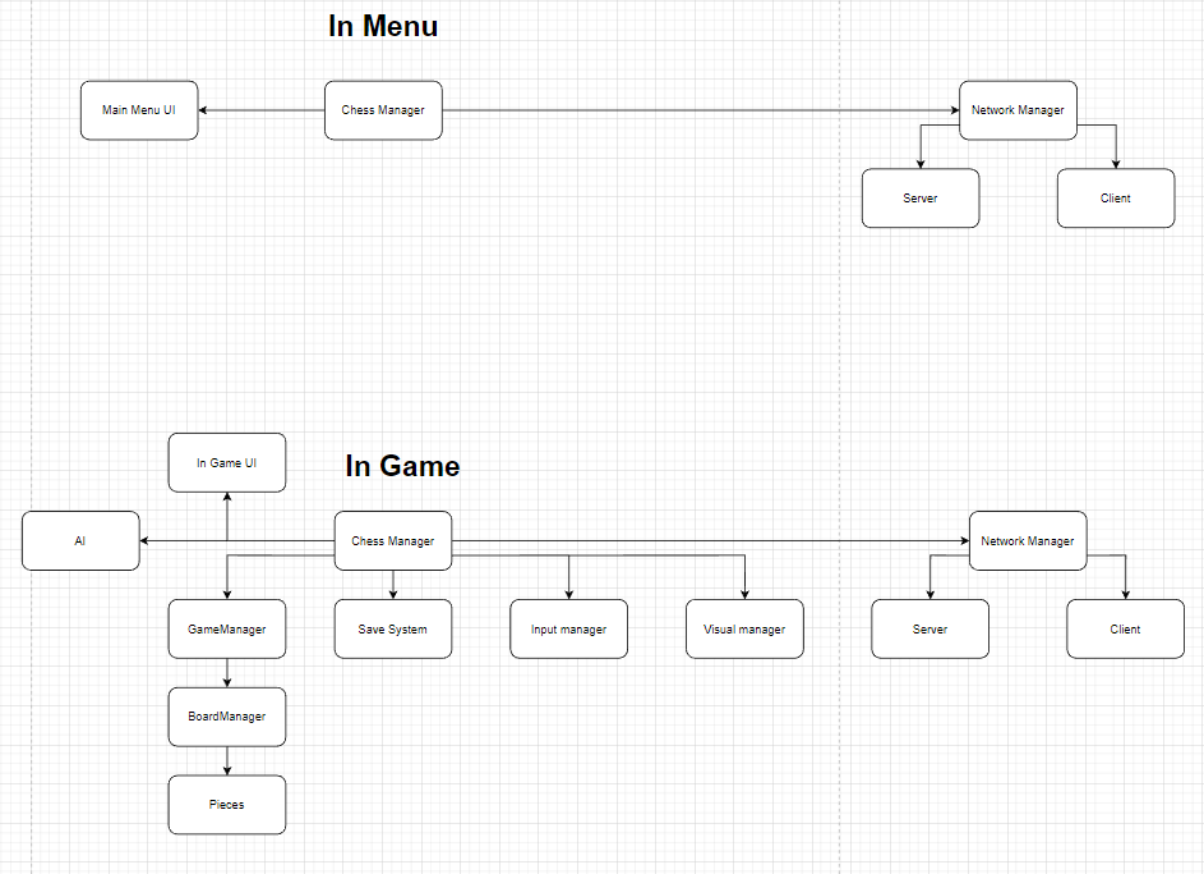
- A powerful computer capable of running Unity, Visual Studio 2022 and completing Unity builds in a reasonable amount of time
- Ability to code C#
- Ability to use sockets
- Ability to implement the minimax algorithm
- Ability to work with binary serialised data for networking and saves
- Ability to 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
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
Game modes 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 and Details

Overview Flowchart



Chess Manager

Controls everything. Handles logic such as In-Game flow (see flowchart below). Classes will follow a tree-like reference structure so a lot of calls will pass through the ChessManager e.g. Client.PieceMoved -> NetworkManager.PieceMoved -> ChessManager.PieceMoved -> GameManager.PieceMoved. This reference structure will be useful as if a bug is occurring with a component, I will know exactly which components can interact with it.

AI

The Chess Manager provides the game mode and current state and waits for the AI to respond with a move. The AI will be based on the MiniMax algorithm using Alpha-Beta pruning. This should work with any game mode, not just chess, however, requires the GameManager to implement a GetScore method to evaluate the strength of a position.

NetworkManager

When playing online, the NetworkManager starts the Client and/or Server which handle player connection, sending save-game data, and sending game mode data. When in-game, the ChessManager gives it moves to send to the server and the NetworkManager informs the ChessManager of incoming moves.

Server

When a client connects the Server will check the networking version of the joining client, check the password (if one is set) and transfer game mode information and the save-game (if one is being used). In-game, the Server acts as a relay forwarding incoming moves to all other clients. To increase code re-use. The server will act purely as a relay so the Client the host runs can be the same as the ones other players run. The only difference between the host and a client is that the host can start games and change team compositions. This is done directly through the NetworkManager, not by the local client sending a message to the local server.

Client

When connecting the Client receives game mode data and save-game data (if it is being used). When in-game, the client will send data to the server such as a move which will be broadcasted to all players and, when it receives data, call a specified method with the decoded data.

GameManager

The GameManager provides an interface for the current game mode and handles game logic such as whose turn it is. Retrieves a list of possible moves from the BoardManager. When given a move by the ChessManager (local, AI, or from NetworkManager), it applies it to the board. When applying a move it returns either the player whose turn it is next or a winning team.

BoardManager

Stores all the pieces on the board, the size of the board and provides a list of moves compiled from requesting possible moves from all the pieces

Pieces

Returns a list of the moves it can make

SaveSystem

ChessManager provides data fetched from GameManager that the SaveSystem then converts to a binary format and saves to a file. Can also de-serialise data. This de-serialised data can be given to GameManager to load the save.

InputManager

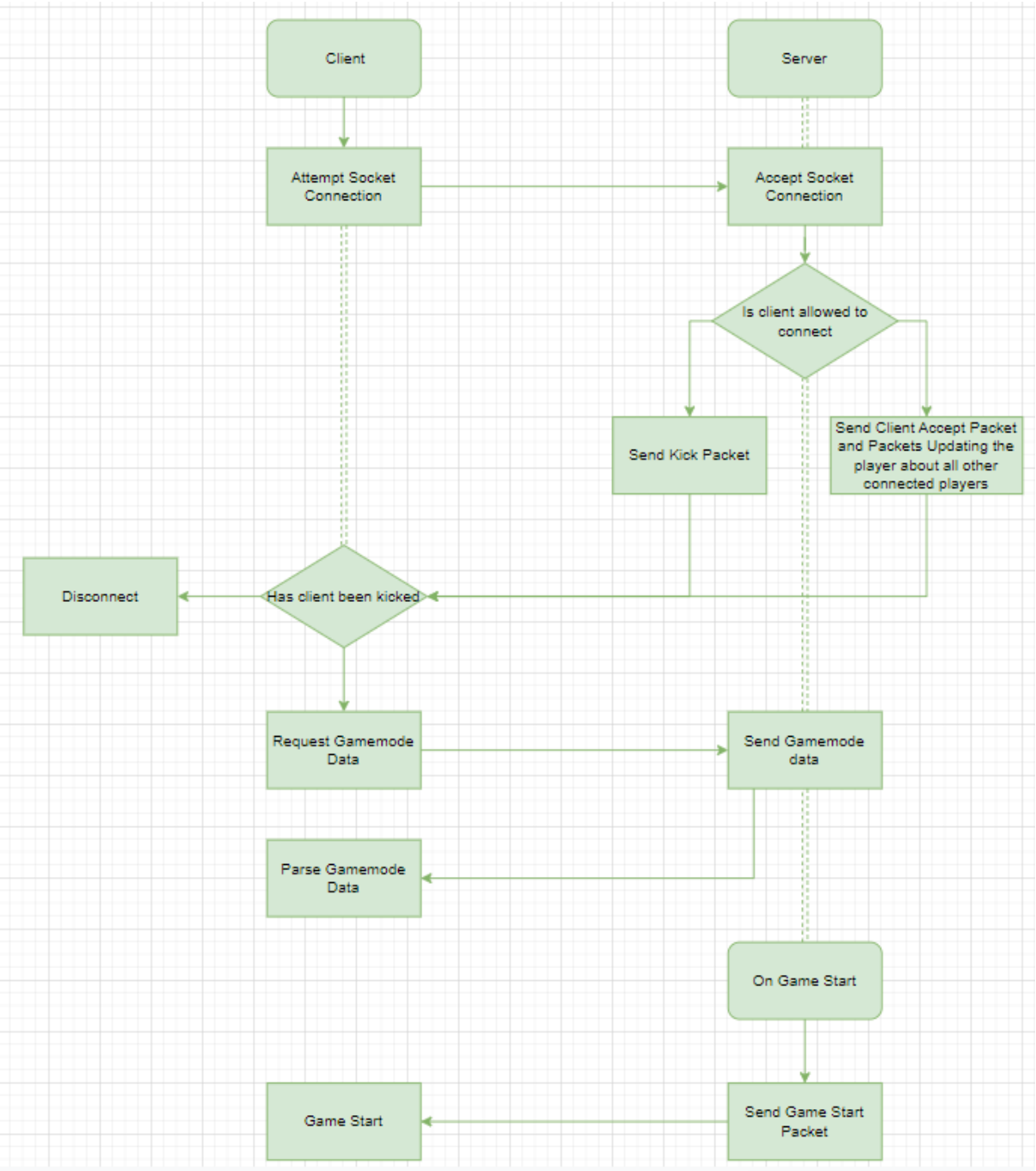
Gets inputs from the player such as where they are moving pieces and passes them to the ChessManager

VisualManager

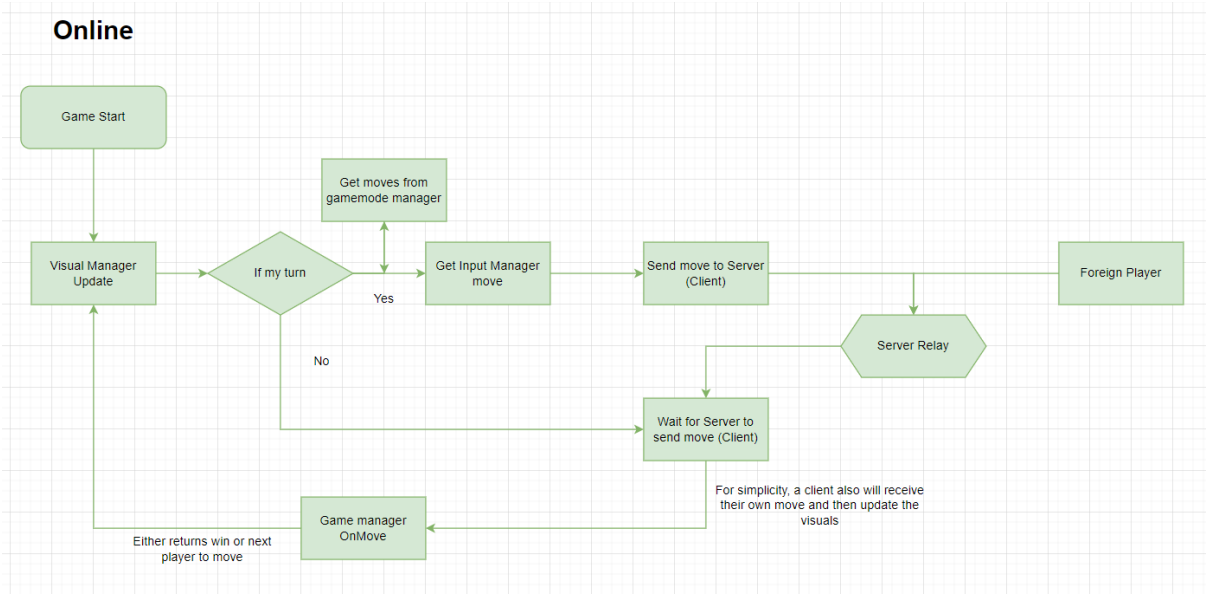
Renders the board and updates it based on data from the GameManager given to it by the

Pregame Networking Flowchart

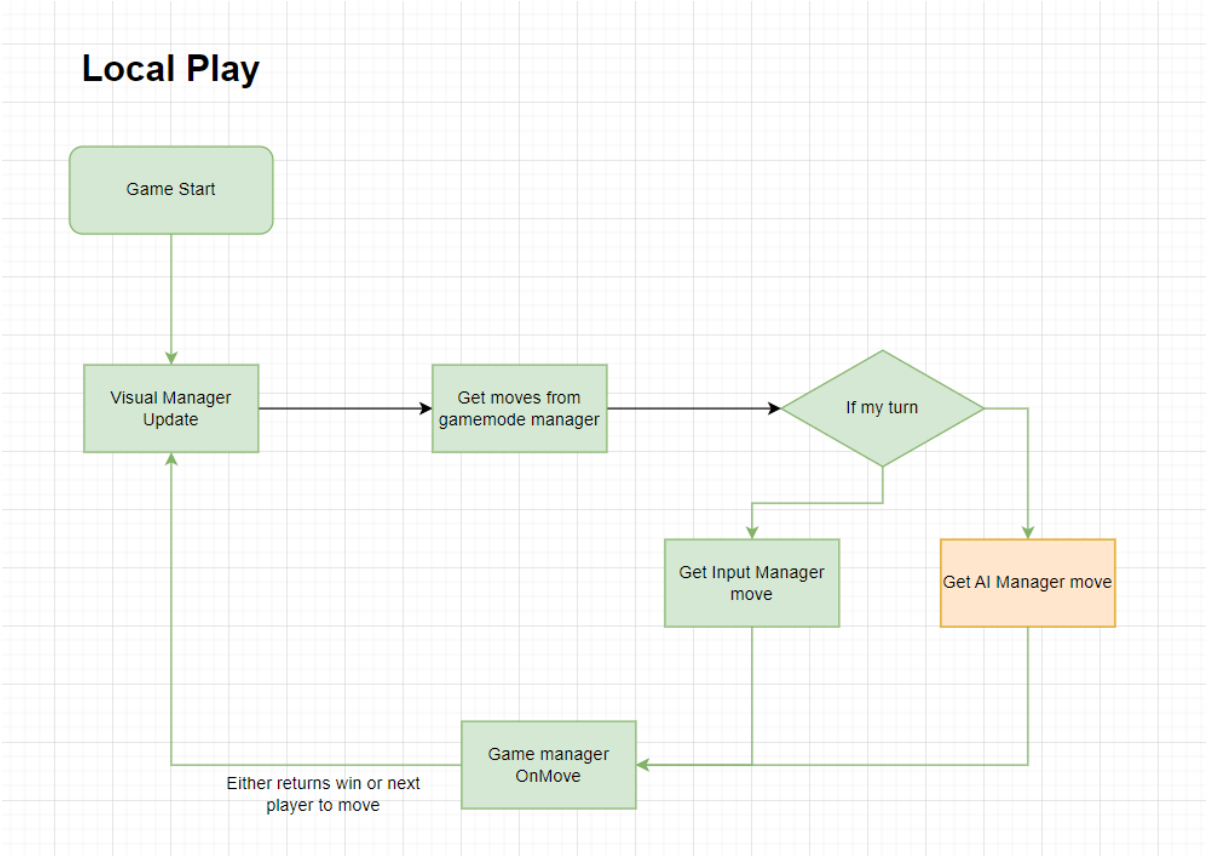
(Dotted lines represent waiting for an event)


Client
Server
Attempt Socket Connection
Accept Socket Connection
Is client allowed to connect
Send Kick Packet
Send Client Accept Packet and Packets Updating the player about all other connected players
Disconnect
Has client been kicked
Request Gamemode Data
Send Gamemode data
Parse Gamemode Data
On Game Start
Game Start
Send Game Start Packet

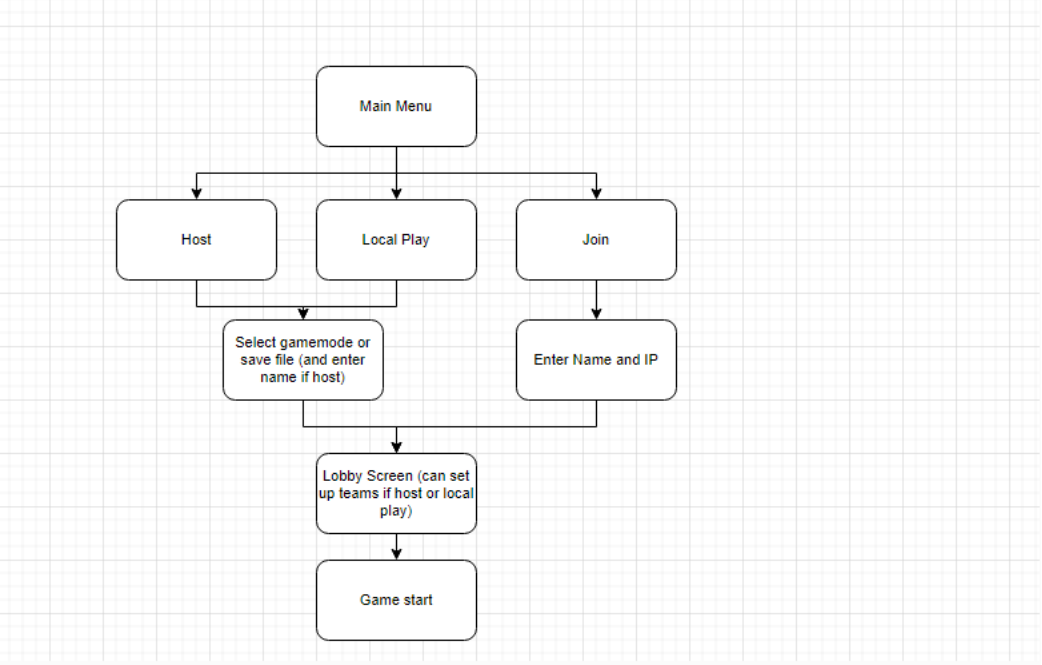
In-Game Networking Flowchart



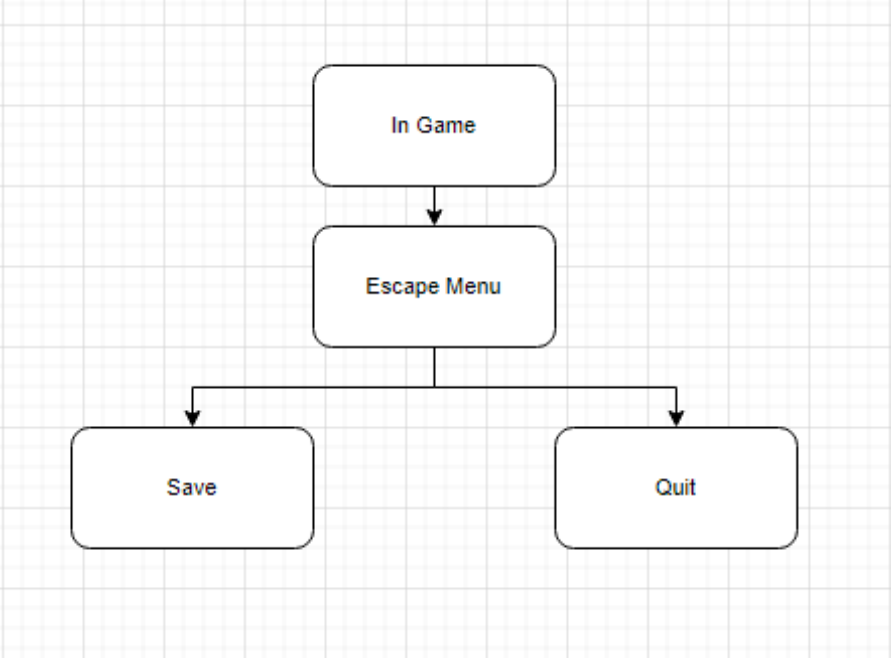
In-Game Local Play Flowchart



Main Menu UI Flowchart



In-Game UI Flowchart

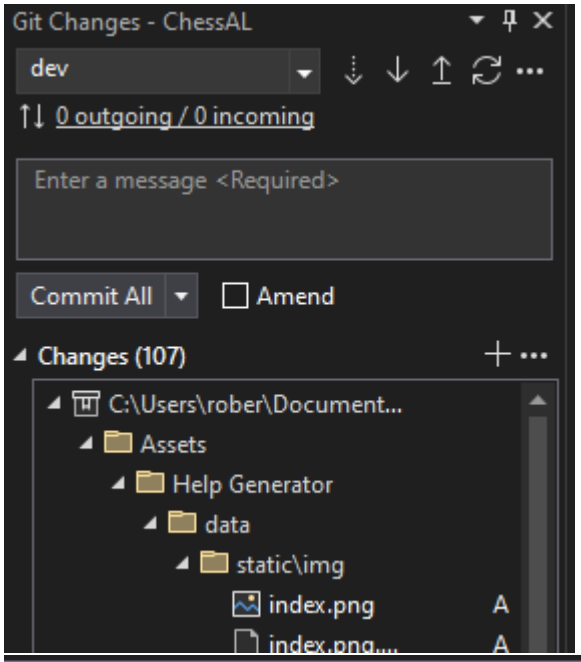


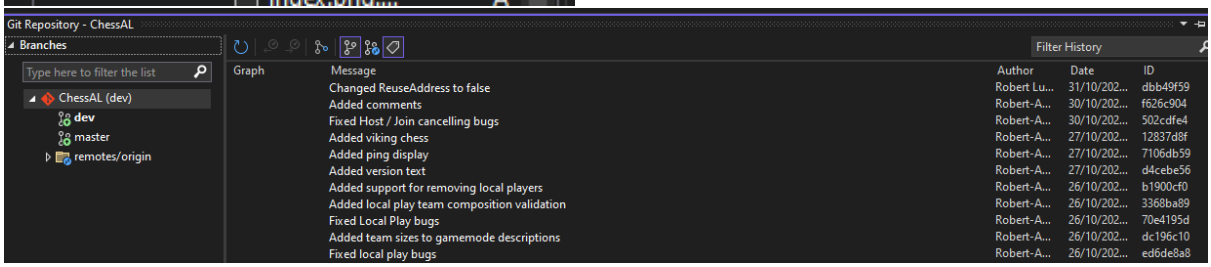
Development

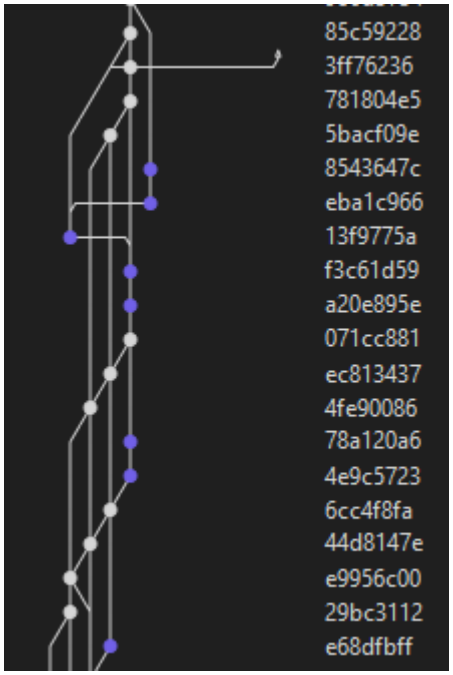
IDE Choice

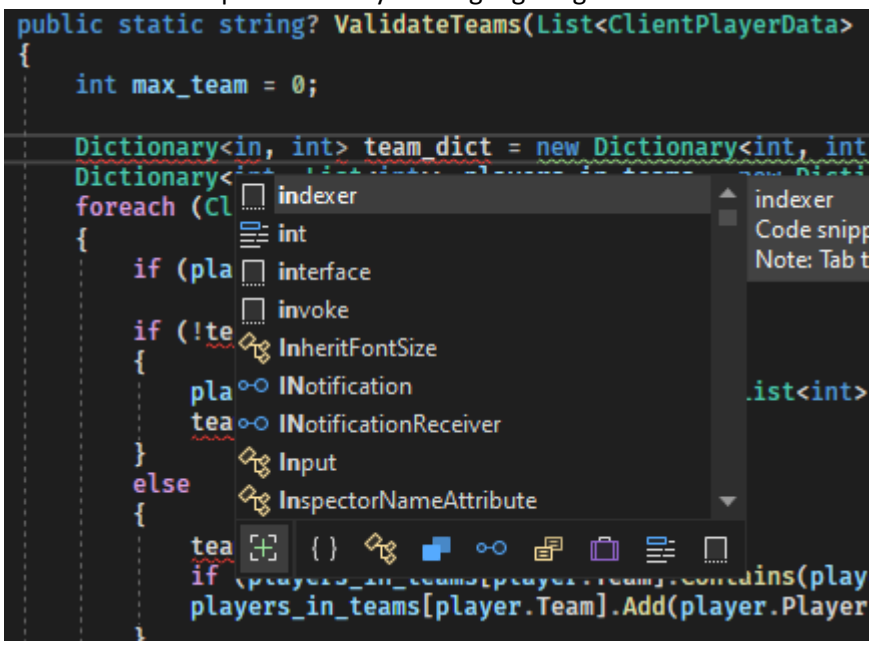
For this project, I used two IDEs for different tasks.

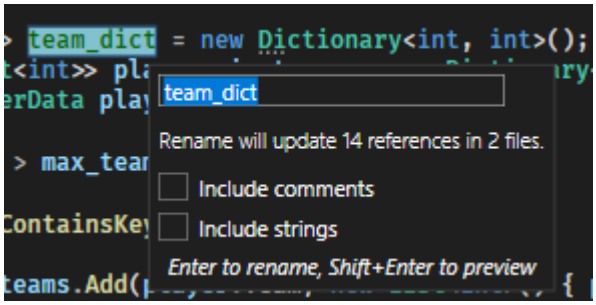
Visual Studio (C# and Git):

- Free
- Great Git integration
- 

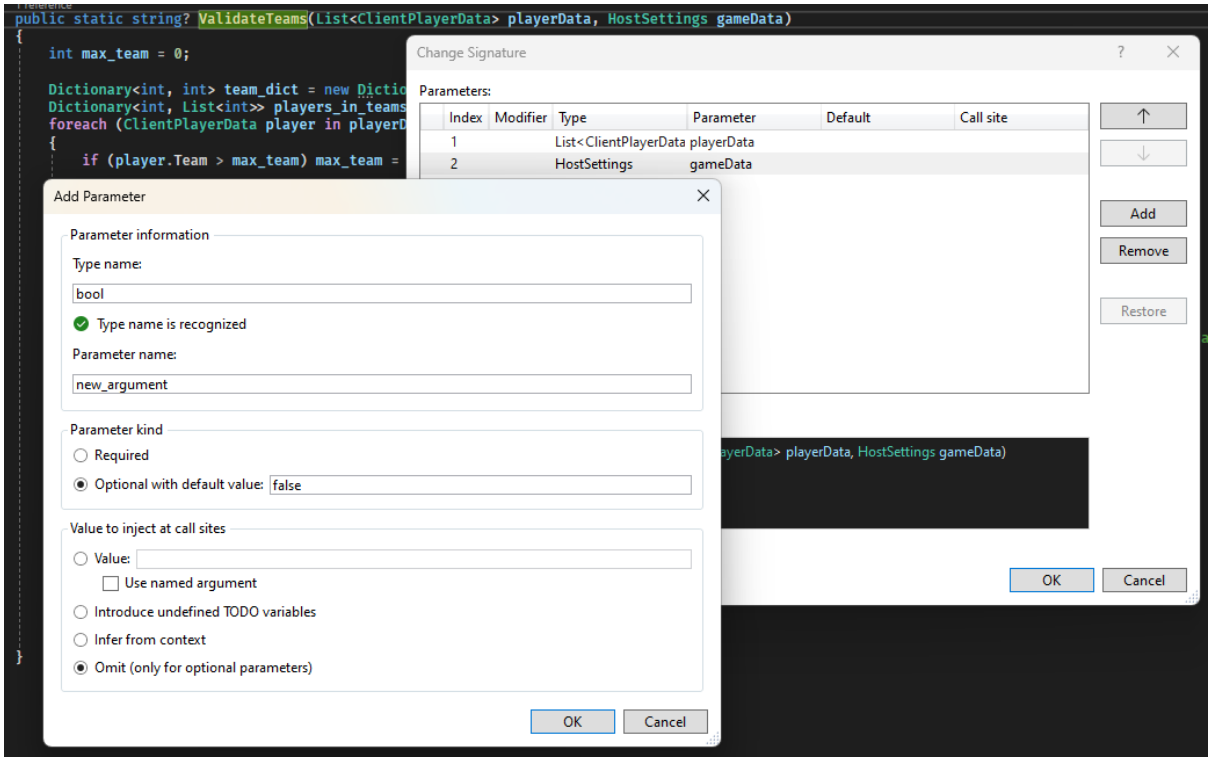
- 

- 

- Great C# autocompletion and syntax highlighting
- 

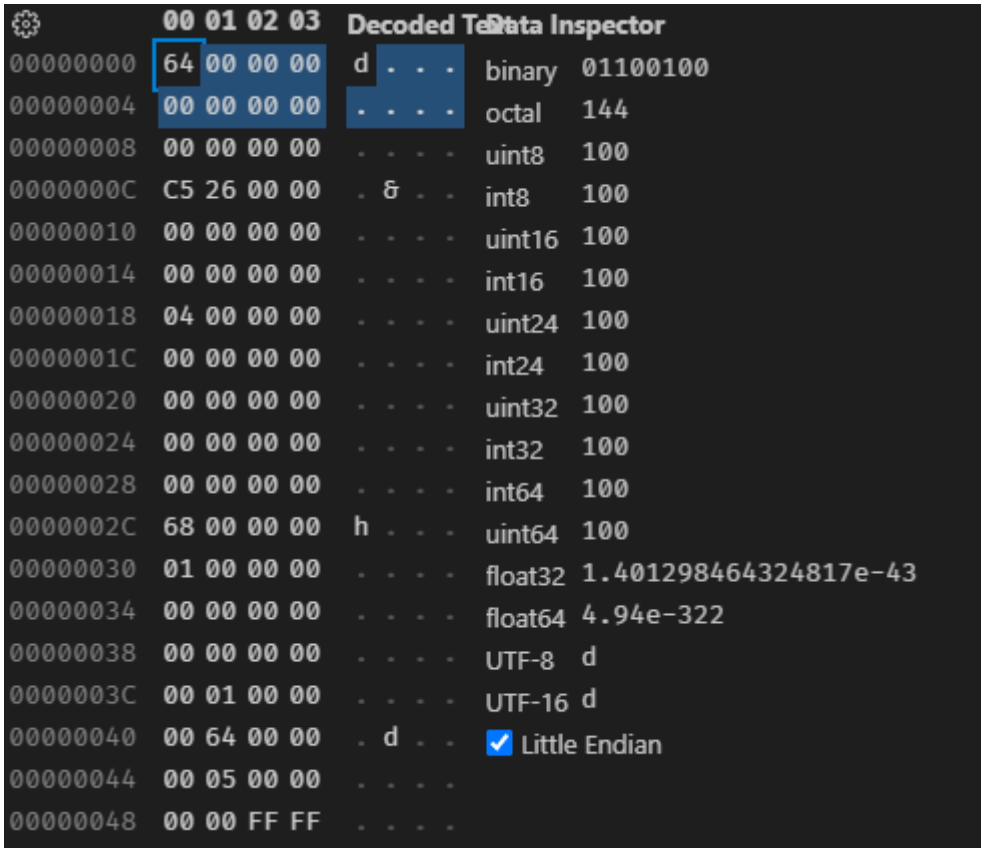
- Powerful C# refactoring for both variables and methods
- 

-

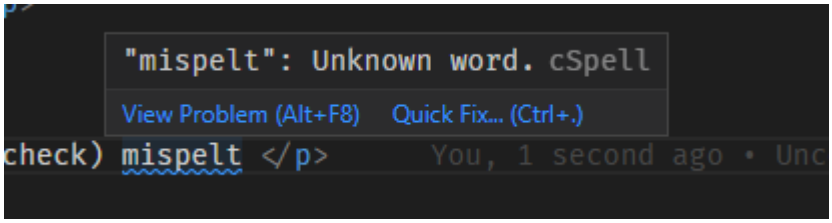


Visual Studio Code (Python, HTML, CSS, binary)

- Free
- More lightweight than Visual Studio
- Hex Editor Extension



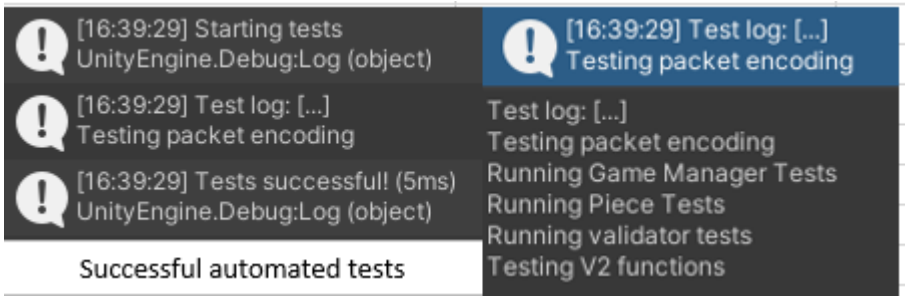
-
- Better Python and HTML integration

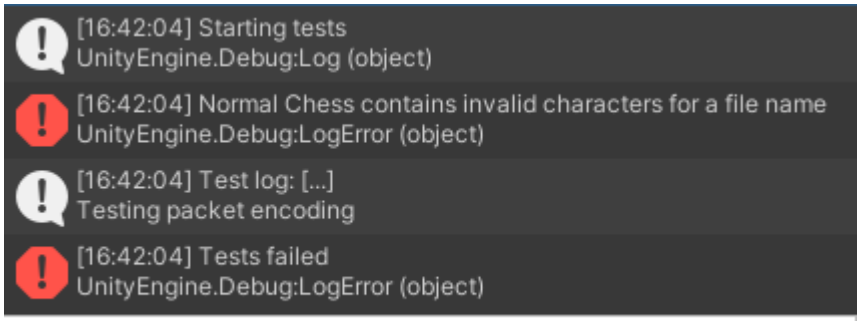
-
```python
from importlib.resources import path
import os
import shutil

PATH_TO_HELP = "../StreamingAssets/Help"
PATH_TO_STATIC = "data/static"
PATH_TO_DATA = "data"
TEMPLATE_FILE = "data/template.html"

# Read template file
with open(TEMPLATE_FILE, 'r') as f:
    template_data = f.read()

# Remove all existing files
if os.path.exists(PATH_TO_HELP):
    shutil.rmtree(PATH_TO_HELP)

# Create help directory
os.mkdir(PATH_TO_HELP)

# Copy static files (css, imgs, etc)
shutil.copytree(PATH_TO_STATIC, PATH_TO_HELP + "/" + PATH_TO_STATIC.split("/")[-1])
```
- Spell-checking extension for creating help pages without typos
- 

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 available in the editor where it runs after every code change. It will be excluded from the release build for optimisation purposes.





Failed test

```
// Don't run in build
#if UNITY_EDITOR
```

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

Validation Testing

```
validator = Validators.ValidatePassword;
tests = new Tuple<string, bool>[] {
    new Tuple<string, bool>("Password", true),
    new Tuple<string, bool>("", true), // No password
    new Tuple<string, bool>("My Password", false), // Illegal char
    new Tuple<string, bool>("sfneksmrnaweirma", true),
    new Tuple<string, bool>("sfneksmrnaweirmaa", false), // Too long
    new Tuple<string, bool>("asd!", true),
    new Tuple<string, bool>("ad!", false), // Too short
    new Tuple<string, bool>("ad!\\", false), // Illegal char
};

foreach (Tuple<string, bool> test in tests)
{
    if ((validator(test.Item1) is null) != test.Item2)
    {
        if (test.Item2) Debug.LogError($"Password '{test.Item1}' failed validation when it should have passed");
        else Debug.LogError($"Password '{test.Item1}' passed validation when it should have failed");
        return false;
    }
}

return true;
```

^ 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.

```
/// <summary>
/// Validates a game password
/// </summary>
/// <param name="password"></param>
/// <returns>Null if successful or a string error</returns>
public static string? ValidatePassword(string password)
{
    if (password == string.Empty) return null;

    const string allowed_chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567890_!@#$*&;:[]{}?.,<>~-=+|";

    // if (Util.RemoveInvisibleChars(name) != name) return "Contains invisible characters"; // Invisible characters

    if (password.Length > 16) return "Password must be shorter than 17 characters"; // Too long

    if (password.Length < 4) return "Password must be longer than 3 characters"; // Too short

    foreach (char c in password)
    {
        if (!allowed_chars.Contains(c))
        {
            if (c == ' ') return $"Character '{c}' (space) not allowed";
            return $"Character '{c}' not allowed";
        }
    }

    return null;
}
```

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

```
/// <summary>
/// Tests packet encoding forwards and backwards to ensure no data loss
/// </summary>
/// <returns></returns>
1 reference
private static bool TestPacketEncoding()
{
    output_string += "Testing packet encoding\n";
    try
    {
        int arg_one = 2;
        double arg_two = 3.8;
        string arg_three = "ldksjfhalkahsdflkbvb";
        string arg_four = "hjsdagfgyurbfv";

        SampleTestPacket test_packet = new SampleTestPacket(PacketBuilder.Decode(SampleTestPacket.Build(arg_one, arg_two, arg_three, arg_four)));

        if (arg_one ≠ test_packet.ArgOne || arg_two ≠ test_packet.ArgTwo || arg_three ≠ test_packet.ArgThree || arg_four ≠ test_packet.ArgFour)
        {
            Debug.LogError("Packet data corrupted in conversion");
            return false;
        }
    }
    catch (Exception e)
    {
        Debug.LogError($"Packet encoding test failed with exception {e}");
        return false;
    }
    return true;
}
```

UID Uniqueness Testing

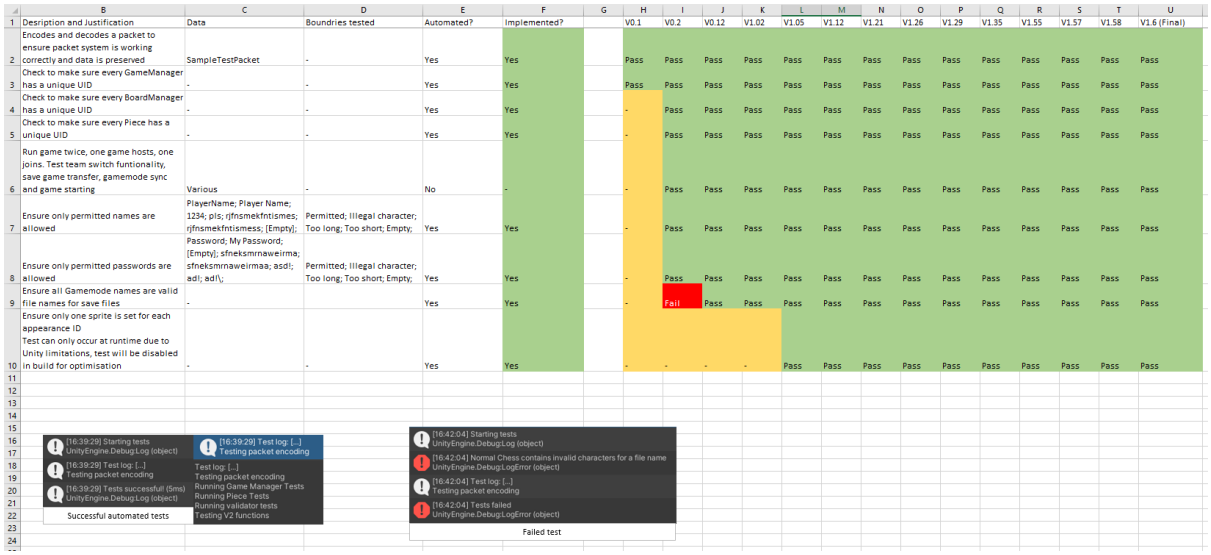
As my game serialisation system relies on unique identifiers for pieces and game manager, my automated tests include tests to ensure there are no identical UIDs

```
/// <summary>
/// Runs various tests on game managers
/// </summary>
/// <returns></returns>
1 reference
private static bool TestGameManagers()
{
    output_string += "Running Game Manager Tests\n";

    // Get all AbstractGameManagerData
    List<AbstractGameManagerData> abstract_game_managers_data = Util.GetAllGameManagers();

    // Ensure no duplicate UIDs
    HashSet<int> used_uids = new HashSet<int>();
    foreach (AbstractGameManagerData data in abstract_game_managers_data)
    {
        if (used_uids.Contains(data.GetUID())) { Debug.LogError($"Gamemode UID ({data.GetUID()}) used twice"); return false; }
        used_uids.Add(data.GetUID());

        // Ensure all gamemode names can be used in file names
        if (data.GetName().IndexOfAny(Path.GetInvalidFileNameChars()) >= 0)
        {
            Debug.LogError($"{data.GetName()} contains invalid characters for a file name");
            return false;
        }
    }

    return true;
}
```

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

Spreadsheet

	B	C	D	E
1	Desription and Justification	Data	Boundries tested	Automated?
2	Encodes and decodes a packet to ensure packet system is working correctly and data is preserved	SampleTestPacket	-	Yes
3	Check to make sure every GameManager has a unique UID	-	-	Yes
4	Check to make sure every BoardManager has a unique UID	-	-	Yes
5	Check to make sure every Piece has a unique UID	-	-	Yes
6	Run game twice, one game hosts, one joins. Test team switch funtionality, save game transfer, gamemode sync and game starting	Various	-	No
7	Ensure only permitted names are allowed	PlayerName; Player Name; 1234; pls; rjfnsmekfntismes; rjfnsmekfntismess; [Empty];	Permitted; Illegal character; Too long; Too short; Empty;	Yes
8	Ensure only permitted passwords are allowed	Password; My Password; [Empty]; sfneksmrnaweirma; sfneksmrnaweirmaa; asd!; ad!; ad!\;	Permitted; Illegal character; Too long; Too short; Empty;	Yes
9	Ensure all Gamemode names are valid file names for save files	-		Yes
10	Ensure only one sprite is set for each appearance ID Test can only occur at runtime due to Unity limitations, test will be disabled in build for optimisation	-	-	Yes



	Description and Justification	Data	Boundries tested	Automated?	Implemented?		V0.1	V0.2	V0.12	V1.02	V1.05	V1.12	V1.21	V1.26	V1.29	V1.35	V1.55	V1.57	V1.58	V1.6 (Final)
2	Encodes and decodes a packet to ensure packet system is working correctly and data is preserved	SampleTestPacket	-	Yes	Yes		Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
3	Check to make sure every GameManager has a unique UID	-	-	Yes	Yes		Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
4	Check to make sure every BoardManager has a unique UID	-	-	Yes	Yes		-	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
5	Check to make sure every Piece has a unique UID	-	-	Yes	Yes		-	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
6	Run game twice, one game hosts, one joins. Test team switch funtionality, save game transfer, gamemode sync and game starting	Various	-	No	-		-	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
7	Ensure only permitted names are allowed	PlayerName; Player Name; 1234; pls; rjfnsmekfntismes; rjfnsmekfntismess; [Empty];	Permitted; Illegal character; Too long; Too short; Empty;	Yes	Yes		-	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
8	Ensure only permitted passwords are allowed	Password; My Password; [Empty]; sfneksmrnaweirma; sfneksmrnaweirmaa; asd!; adl; adl\;	Permitted; Illegal character; Too long; Too short; Empty;	Yes	Yes		-	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
9	Ensure all Gamemode names are valid file names for save files	-		Yes	Yes		-	Fail	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
10	Ensure only one sprite is set for each appearance ID Test can only occur at runtime due to Unity limitations, test will be disabled in build for optimisation	-	-	Yes	Yes		-	-	-	-	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass

Testing with Stakeholders

I decided to do frequent tests with shareholders, switching shareholders every time to get focused feedback from various points of view. I recorded the feedback I received and my response to the feedback in a spreadsheet. These tests will be end-to-end meaning that in every test every key feature will be tested (game modes will be tested on a rota).

Spreadsheet

	A	B	C
1	Build Version	Feedback	Response to feedback
2	V1.02	- Game is functional - Unique gamemodes fun - Save games work and are useful - UI functional but unintuative and not beginner friendly - Gamemodes don't have proper descriptions that explain game rules	- UI is being worked on and will become more user friendly with tutorials - I will add UI elements that allow the client to learn about the rules of a gamemode
3	V1.17	- Aesthetic UI improvements - Models for each gamemode (don't reuse chess models for checkers) - Error when quitting to menu, have to restart game to play again	- These are aesthetic suggestions so they are a low priority and will be sorted later - This is a know bug and will be fixed in the next version
4	V1.24	- Game generally works well - Viking chess didn't implement all game rules - Adding sound for piece movements	- Viking chess has been correctly implemented as of V1.25 - Sounds will added soon
5	V1.25	-Unintuative UI - Viking chess AI doesn't work - Normal chess AI sometimes freezes - Sometimes players appear twice in lobbies - AI sometimes doesn't pick best move	- UI will be overhalled - Viking chess AI has already been fixed as of V1.27 - Normal chess AI will be looked at - Duplicate player bug will be looked at
6	V1.34	- No colour sheme / boring colours - No dividers between buttons - Make errors display in a nicer way - Online multiplayer (through public IP not working)	- Online multiplayer fixed as of version V1.35 - UI feedback is mixed therefore it will keep its current theme as it is functional, a low priority, and liked by a majority of people

7	V1.57	- Make miniturised help show up when switching gamemodes, not just when pressing help - If no saves available show message explaining that - Mouse inputs go through escape menu in 3D mode - Viking chess has 2 pieces missing - Atomic Chess AI doesn't recognise atomic moves - More example images in help pages	- Will implement miniturised help showing as switching through gamemodes - Will implement message showing no saves available - Will prevent 3D mouse inputs from working while escape menu is showing - Will add 2 pieces to Viking Chess - Will look into Atomic Chess
8	V1.57 (Second tester)	- Centre 3D camera rotation - Pieces don't make last winning move - 3D squares sometimes dissapear	- Camera has been centred as of V1.58 - Piece movement has been fixed as of V1.58 - Squares fixed as of V1.58

Version Control

I decided to use GitHub for version control to allow me to roll back changes that broke components, compare code to previous versions, serve as a backup and allow testers to download the application. I later started using 2 branches for reasons stated here: Keeping In-Dev and Stable Builds Separate (VCS). This ended up being very helpful as a backup for reasons stated here: Fixing Public IP Bug

Code Structuring and Formatting

To keep the coding style consistent and names informative I've tried to keep to a single naming and style convention where possible. Most of this was done by Visual Studio (my main IDE for this project other than Visual Studio Code which I used for the web-technology-based Help System) such as deciding where curly braces go and formatting the summaries.

Naming

Class Names	UpperCamelCase
Public Attributes	UpperCamelCase
Private Attributes	lowerCamelCase
Public Methods	UpperCamelCase
Private Methods	UpperCamelCase
Method Arguments	lowerCamelCase
Local Variables	snake_case

Variable Scoping and Namespaces

Where possible, I have tried to reduce namespace clutter as much as possible through various techniques. Firstly, I've tried to make as many attributes and methods private as possible which prevents typos, reduces the chance of methods being used incorrectly or attribute modification breaking a class's functionality, and improves autocompletion as private methods and attributes are hidden. Another way I've achieved this is by using namespaces. These force me to use import statements (or 'using' statements in C#) to reference classes in a different namespace. These improve autocompletion and prevent typos as before, but also make me explicitly import other modules when I want communication between them reducing the chance of me absentmindedly creating a dense web of references that are difficult to understand.

Use of Enums

Another thing I have used to improve code cleanliness is enums. When you need to represent a set number of states you can either do it using an integer or short with different values meaning different states however these are difficult to understand and not very readable. By using enums, you essentially give names to these states that can be used at other points in the program reducing typos and errors.

```
public enum KingAlive
{
    None = 0,
    White = 1,
    Black = 2,
    Both = 3,
}
```

Developmental Challenges Encountered and Solutions

Main Thread Queueing

As you can only interact with most Unity components from the main game thread, I needed to find a way to queue actions from different threads that require interaction with Unity such as those coming from the networking system. To achieve this, I added a queue to the ChessManager:

```
/// <summary>
/// A queue of actions to be excecuted on the main thread on the next frame
/// </summary>
private volatile Queue<Action> mainThreadActions = new Queue<Action>();
```

This queue is then copied, cleared and all its actions executed on the next frame:

```
// Called every frame
Unity Message | 0 references
public void Update()
{
    /*
     * Most unity functions can only be called from the main thread so this
     * goes through functions queued by other threads to run them
     * on the main thread during the next frame
     */

    Queue<Action> temp_main_thread_actions = mainThreadActions;
    mainThreadActions = new Queue<Action>();

    while (temp_main_thread_actions.Count > 0)
    {
        temp_main_thread_actions.Dequeue().Invoke();
    }
}
```

An action can be queued like this:

```
1 reference
private void HostStartGameFail(string reason) ⇒ mainThreadActions.Enqueue(() ⇒ { menuUIManager.HostStartGameFailed(reason); });
```

With this being the part that will be executed on the next frame:

```
menuUIManager.HostStartGameFailed(reason);
```

Networking System

The Networking System was complicated to implement as it needed to be robust, efficient and needed to interact correctly with the game. It also required a large amount of multithreading as it needed to be able to fulfil many different requirements. The Server needed to have a thread to accept (or reject) new clients at any time. This was done using the 'ClientAcceptThread':

The Client needed a thread to connect to the server as, otherwise, it would have frozen the game while connecting:

The Client and Server needed to use asynchronous sockets to receive data at any time:

The Client and Server needed a receive thread to handle received data one at a time from a queue as handling the data within the asynchronous receive of the sockets would lead to multiple things not designed to happen concurrently happening concurrently and would prevent the sockets from receiving more data while these were being handled:

The Client and Server needed a send thread to send data at any time without blocking other code while waiting for data to send:

Packet System and Generation

The packet system works by encoding the data to be sent in the format below:

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 the packet type	Content block 1 length bytes
[Repeat last two for all content]			

The packet length is used to separate packets if two are received and the UID is used to determine how to handle an incoming packet.

Each packet type is a class auto-generated from Python code to contain the content as attributes, a constructor to take a Packet class input and decode it, storing its values in the classes attributes, and a 'Build' method that takes all the content as arguments and returns a byte array that can be sent.

To send a packet, you call (PacketClassName).Build(arg1, arg2, arg3, …) and pass the byte array to the relevant method that can send it. To decode a packet you use PacketBuilder.Decode(incoming_bytes) on received bytes which returns a 'Packet' object. This object contains the UID which can be used to determine how the packet should be handled (see: Network On Packet Received Handling) and finally, the bytes are parsed into the packet class's constructor creating an object with the content as its attributes.

Packet encoding and decoding class:

```
/// <summary>
/// A generic packet - no specific types
/// </summary>
37 references
public struct Packet
{
    public int UID;
    public int From;
    public List<byte[]> Contents;

    1 reference
    public Packet(int uid, List<byte[]> contents, int from = -1)
    {
        UID = uid;
        Contents = contents;
        From = from;
    }
}
```

```
/// <summary>
/// Creates and decodes packets
/// </summary>
23 references
public static class PacketBuilder
{
    private static Encoding _encoder = new UTF8Encoding();

    public const int PacketLenLen = 4;
    public const int UIDLen = 4;
    public const int DataLenLen = 4;

    /// <summary>
    /// Decodes packet length from packet
    /// </summary>
    /// <param name="content"></param>
    /// <returns>Packet length</returns>
    3 references
    public static int GetPacketLength(byte[] content)
    {
        return BitConverter.ToInt32(ArrayExtensions.Slice(content, 0, PacketLenLen)) + PacketLenLen;
    }

    /// <summary>
    /// Builds the packet - Adds the UID and calculates lengths for each section of the packet
    /// </summary>
    /// <param name="UID"></param>
    /// <param name="contents"></param>
    /// <returns></returns>
    /// <exception cref="PacketDecodeError"></exception>
    14 references
    public static byte[] Build(int UID, List<byte[]> contents) // READ DOCUMENTATION TO SEE PACKET STRUCTURE
    {
        try
        {
            byte[] buffer = new byte[1024];
            int cursor = PacketLenLen;
            ArrayExtensions.Merge(buffer, BitConverter.GetBytes(UID), cursor);
            cursor += UIDLen;

            foreach (byte[] c in contents)
            {
                ArrayExtensions.Merge(buffer, BitConverter.GetBytes(c.Length), cursor);
                cursor += 4;
                ArrayExtensions.Merge(buffer, c, cursor);
                cursor += c.Length;
            }

            // Add packet length
            ArrayExtensions.Merge(buffer, BitConverter.GetBytes(cursor - 4), 0);

            return ArrayExtensions.Slice(buffer, 0, cursor);
        }
        catch (Exception e)
```

```
        {
            throw new PacketDecodeError("Error decoding packet: " + e);
        }
    }

    /// <summary>
    /// Converts a string to bytes with the current encoding system
    /// </summary>
    /// <param name="input"></param>
    /// <returns>Encoded bytes</returns>
    0 references
    public static byte[] ByteEncode(string input) => _encoder.GetBytes(input);

    /// <summary>
    /// Separates raw packet data into its separate components
    /// </summary>
    /// <param name="data"></param>
    /// <param name="from"></param>
    /// <returns></returns>
    4 references
    public static Packet Decode(byte[] data, int from = -1) // READ DOCUMENTATION TO SEE PACKET STRUCTURE
    {
        int cursor = 4;
        int UID = BitConverter.ToInt32(ArrayExtensions.Slice(data, cursor, cursor + UIDLen));
        cursor += UIDLen;

        List<byte[]> contents = new List<byte[]>();
        while (cursor < data.Length)
        {
            int data_len = BitConverter.ToInt32(
                ArrayExtensions.Slice(data, cursor, cursor + DataLenLen)
            );
            data_len = BitConverter.ToInt32(
                ArrayExtensions.Slice(data, cursor, cursor + DataLenLen)
            );
            cursor += DataLenLen;
            byte[] content = ArrayExtensions.Slice(data, cursor, cursor + data_len);
            cursor += data_len;
            contents.Add(content);
        }

        return new Packet(UID, contents, from);
    }
}
```

Example packet:

```
4 references
public class ClientConnectRequestPacket {
    public const int UID = 0;
    public string Name;
    public string Version;
    public string Password;
    1 reference
    public ClientConnectRequestPacket(Packet packet){
        Name = ASCIIEncoding.ASCII.GetString(packet.Contents[0]);
        Version = ASCIIEncoding.ASCII.GetString(packet.Contents[1]);
        Password = ASCIIEncoding.ASCII.GetString(packet.Contents[2]);
    }


    1 reference
    public static byte[] Build(string _Name, string _Version, string _Password="") {
        List<byte[]> contents = new List<byte[]>();
        contents.Add(ASCIIEncoding.ASCII.GetBytes(_Name));
        contents.Add(ASCIIEncoding.ASCII.GetBytes(_Version));
        contents.Add(ASCIIEncoding.ASCII.GetBytes(_Password));
        return PacketBuilder.Build(UID, contents);
    }
}
```

Example packet usage:

```
Send(ClientConnectRequestPacket.Build(PlayerName, NetworkSettings.VERSION, Password)); // Send connection request
```

```
// Decode connection data
ClientConnectRequestPacket initPacket = new ClientConnectRequestPacket(
    PacketBuilder.Decode(byte_data)
);
```

```
// Password incorrect
if (serverPassword ≠ "" && initPacket.Password ≠ serverPassword)
{
```

Packet Generator:

```
import os
import shutil
import sys
import time

# Path to packets
PATH = "..\\Packets"

start_time = time.time()

with open("Packets.txt", "r") as f:
    all_packets_split = f.read().split("\n")

index = 0
while index < len(all_packets_split):
    # Remove lines that don't contain '\' and thus are comments
    if "\\" not in all_packets_split[index]:
        all_packets_split.pop(index)
    else:
        index += 1

# Split all commands
all_packets = [i.split("\\") for i in all_packets_split]

# Delete all existing packets
for filename in os.listdir(PATH):
    file_path = os.path.join(PATH, filename)
    try:
        if os.path.isfile(file_path) or os.path.islink(file_path):
            os.unlink(file_path)
        elif os.path.isdir(file_path):
            shutil.rmtree(file_path)
    except Exception as e:
        print('Failed to delete %s. Reason: %s' % (file_path, e))

# Create packet
for j in all_packets:
    # Extract UID and name
    uid = int(j[0])
    packet_name = j[1]

    # Add default attribute
    attributes = [["UID", "int"]]

    # Add all other attributes
    x = 2
    while x < len(j):
```

```
        attributes.append([j[x], j[x+1]])
        x += 2

    # Set the filename
    filename = f"{uid}_{packet_name}Packet.cs"

    # Add common start of file
    data = f"""using System;…

    # Add UID attribute
    data += f"        public const int UID = {uid};\n"

    # Add all other attributes
    for i in attributes[1:]:
        splitted = i[0].split("=") # Remove default value e.g. a="2" → a
        data += "        public " + i[1] + " " + splitted[0] + ";" + "\n"

    # Add common lines
    data += f"""        …

    # Add attributes to constructor
    for j, i in enumerate(attributes[1:]):
        splitted = i[0].split("=")

        data += "            " + splitted[0] + " = "

        # Add decoder based on type
        if i[1] == "string":
            data += "Encoding.ASCII.GetString"      You, 13 minutes ago • Updated packet ge
        elif i[1] == "int":
            data += "BitConverter.ToInt32"
        elif i[1] == "double":
            data += "BitConverter.ToDouble"
        elif i[1] == "byte[]":
            pass
        else:
            print(f"Unsupported type: {i[1]}")
            sys.exit()

        data += f"(packet.Contents[{j}]);\n"
```

```
    # Add common line
    data += f"""        }}…

    # Add common line
    data += """        public static byte[] Build("""

    # Add attributes to build method parameters
    for i in attributes[1:]:
        splitted = i[0].split("=")
        data += i[1] + " _" + splitted[0]
        if len(splitted) > 1:
            data += "=" + splitted[1]
        data += ", "

    if len(attributes) > 1:
        data = data[:-2]

    # Add common line
    data += """) {…

    # Add encoded attributes to contents
    for i in attributes[1:]:
        splitted = i[0].split("=")
        if i[1] == "string":
            data += f"                Encoding.ASCII.GetBytes(_{splitted[0]}),\n"
        elif i[1] == "byte[]":
            data += f"                _{splitted[0]},\n"
        else:
            data += f"                BitConverter.GetBytes(_{splitted[0]}),\n"

    # Add common line
    data += """            };…

    # Write file
    with open(f"{PATH}\\{filename}", "w+") as f:
        f.write(data)

print(f"Time taken: {round((time.time()-start_time)*1000, 2)}ms")
```

Network On Packet Received Handling

To match the packet system's ease of use and extensibility, I needed a way to easily extend the functionality when a packet is received without resorting to a large if-else chain. To do this, I created a dictionary that maps packet IDs to methods as C# supports the functional paradigm of functions being treated as first-class citizens and being able to be stored and named as objects. The dictionary looks like this:

```
UIDtoAction = new Dictionary<int, Action<Packet>>
{
    { 1,  ServerAccept },
    { 2, ServerKick },
    { 3, PlayerInformationUpdate },
    { 6, PlayerDisconnect },
    { 5, PingResponse },
    { 201, GamemodeDataRecieve},
    { 202, OnGameStart },
    { 205, OnMoveUpdate },
};
```

The ‘Action<Packet>’ is a method that takes an object of type ‘Packet’ packet as its first (and only) argument.

This method then tries to map an incoming packet’s UID to a method and call it:

```
/// <summary>
/// Attempts to process a packet
/// </summary>
/// <param name="packet"></param>
/// <returns>Whether the packet was successfully processed</returns>
1 reference
public bool TryHandlePacket(Packet packet)
{
    if (!UIDtoAction.ContainsKey(packet.UID)) return false;

    UIDtoAction[packet.UID](packet);
    return true;
}
```

Example of a packet handling method:

```
// Server accepted connection
1 reference
public void ServerAccept(Packet packet)
{
    ServerConnectAcceptPacket acceptPacket = new ServerConnectAcceptPacket(packet);
    client.PlayerID = acceptPacket.PlayerID;
}
```

All of this occurs within the InternalClientPacketHandler or InternalServerPacketHandler which is called by the Client or Server respectively like this:

```
try
{
    Packet packet = PacketBuilder.Decode(content);
    bool handled = internalPacketHandler.TryHandlePacket(packet);

    if (!handled)
    {
        Debug.LogError($"Packet [UID:{packet.UID}] not handled");
    }
}
catch (PacketDecodeError)
{
    Debug.LogError("Packed decode error");
}
```

Shutting Down Networking

Shutting down networking was, surprisingly, one of the biggest challenges with networking. This was because I needed all the socket connections to be properly disposed of so they could be reused and I needed the Client to inform the Server that it was disconnecting. This was especially important as the client’s name needed to disappear from the player list so that another player could take their place on a team. This disconnection packet would be at the back of the send queue so I needed to ensure that all Client threads stopped running and then the send queue was flushed (all messages sent).

This is how I implemented this:

```
/// <summary>
/// Shutdown client
/// </summary>
2 references
public void Shutdown()
{
    Debug.Log("Sending disconnect");
    SendMessage(ClientDisconnectPacket.Build());
    Debug.Log("Client shutdown");

    if (connectionThread.ThreadState == ThreadState.Running) connectionThread.Abort();
    receiveThread.Abort();
    sendThread.Abort();

    Debug.Log("Flushing send queue");
    try { FlushSendQueue(); } catch (NullReferenceException) { }

    Debug.Log("Disconnecting sockets");

    try { handler?.Disconnect(false); } catch (SocketException) { }
    try { handler?.Shutdown(SocketShutdown.Both); } catch (SocketException) { }
    try { handler?.Close(0); } catch (SocketException) { }
    try { handler?.Dispose(); } catch (SocketException) { }
}
```

```
/// <summary>
/// Sends all remaining messages from the send queue.
/// </summary>
/// <exception cref="ThreadStateException">Throws ThreadStateException if send loop is running</exception>
1 reference
private void FlushSendQueue()
{
    if (sendThread.ThreadState == ThreadState.Running) throw new ThreadStateException("Send queue cannot be flushed while send loop is running!");

    while (!sendQueue.IsEmpty) SendMessageFromSendQueue();
}
```

In addition to this, I made the server intermittently check if clients were connected when the server wasn't busy:

```
while (running)
{
    // Nothing recieved
    if (recieveQueue.IsEmpty)
    {
        // Poll player to see if they are still connected
        while (!PlayerData.ContainsKey(nextToPoll))
        {
            nextToPoll++;
            if (nextToPoll > PlayerIDCounter) nextToPoll = 0;
        }

        bool alive = NetworkingUtils.SocketConnected(PlayerData[nextToPoll].Handler);

        if (!alive)
        {
            TryRemovePlayer(nextToPoll, "Socket poll failed");
            Debug.Log($"Player {nextToPoll} kicked due to socket poll fail");
        }

        Thread.Sleep(RECEIVE_COOLDOWN);

        continue;
    }
```

```
/// <summary>
/// Polls a socket to see if it is still connected. Used for kicking clients that didn't leave gracefully
/// </summary>
/// <param name="s"></param>
/// <returns></returns>
1 reference
public static bool SocketConnected(Socket s)
{
    bool part1 = s.Poll(1000, SelectMode.SelectRead);
    bool part2 = (s.Available == 0);
    if (part1 && part2)
        return false;
    else
        return true;
}
```

This was important because should the client's game process be suddenly killed, their Client code encountered a fatal error, or their computer turned off, the disconnect packet would never reach the Server.

Using C# Reflection to Avoid Creating Lists of Classes

In many parts of my program, I need a list of every class that inherits from a base class or I need a dictionary of UIDs mapped to these classes. Without reflection, this would require maintaining two or three lists of classes that I would need to update every time I added a new feature such as a game mode so that they showed up in game mode selection menus and could be used. Reflection allows me to get information about the structure of the program such as getting a list of all classes and filter them for ones that inherit from a specific class.

```
/// <summary>
/// Retrieves a list of all subclasses of AbstractGameManagerData
/// </summary>
/// <returns></returns>
3 references
public static List<AbstractGameManagerData> GetAllGameManagers()
{
    List<AbstractGameManagerData> game_managers_data = new List<AbstractGameManagerData>();

    // Get types from reflection
    foreach (Type type in System.Reflection.Assembly.GetExecutingAssembly().GetTypes()
             .Where(mytype => mytype.IsSubclassOf(typeof(AbstractGameManagerData))))
    {
        game_managers_data.Add((AbstractGameManagerData)Activator.CreateInstance(type)); // Create instance
    }

    // Order
    return game_managers_data.OrderBy(o => o.GetUID()).ToList();
}
```

In the example above I iterate through each type in the assembly:

```
foreach (Type type in System.Reflection.Assembly.GetExecutingAssembly().GetTypes()
```

Filtering out all those which aren't a subclass of the AbstractGameManager:

```
.Where(mytype => mytype.IsSubclassOf(typeof(AbstractGameManagerData))))
```

Then I create an instance of the type and add it to the list:

```
game_managers_data.Add((AbstractGameManagerData)Activator.CreateInstance(type)); // Create instance
```

Finally re-ordering it:

```
// Order
return game_managers_data.OrderBy(o => o.GetUID()).ToList();
```

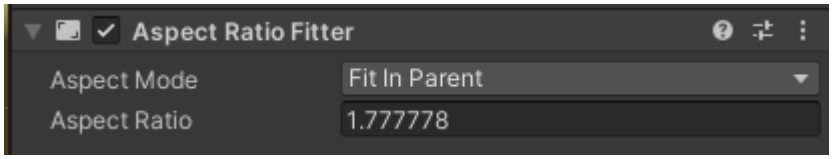
This allows me to add game modes, pieces and more without adding them to a constant list in another part of the program.

Another use for this is with testing. Instead of writing individual tests for each class, I used reflection and a generic test to test any new class I added without any extra code. In the example below, I use a method I wrote that retrieves all AbstractGameMangerData classes and test to make sure that their UIDs are unique and that their names can be used in save files.

```
/// <summary>
/// Runs various tests on game managers
/// </summary>
/// <returns></returns>
1 reference
private static bool TestGameManagers()
{
    output_string += "Running Game Manager Tests\n";

    // Get all AbstractGameManagerData
    List<AbstractGameManagerData> abstract_game_managers_data = Util.GetAllGameManagers();

    // Ensure no duplicate UIDs
    HashSet<int> used_uids = new HashSet<int>();
    foreach (AbstractGameManagerData data in abstract_game_managers_data)
    {
        if (used_uids.Contains(data.GetUID())) { Debug.LogError($"Gamemode UID ({data.GetUID()}) used twice"); return false; }
        used_uids.Add(data.GetUID());

        // Ensure all gamemode names can be used in file names
        if (data.GetName().IndexOfAny(Path.GetInvalidFileNameChars()) >= 0)
        {
            Debug.LogError($"{data.GetName()} contains invalid characters for a file name");
            return false;
        }
    }

    return true;
}
```

Accounting for all Display Sizes and Ratios

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, centred 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 from Another Scene

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

```
[InitializeOnLoad]
1 reference
public static class SimpleEditorUtils
{
    0 references
    static SimpleEditorUtils()
    {
        // Add method to event
        EditorApplication.playModeStateChanged += ModeChanged;
    }

    [MenuItem("Edit/Play-Unplay, But From Prelaunch Scene %q")]
    0 references
    public static void PlayFromPrelaunchScene()
    {
        // Stop if playing
        if (EditorApplication.isPlaying == true)
        {
            EditorApplication.isPlaying = false;
            return;
        }

        // Save scene
        EditorSceneManager.SaveCurrentModifiedScenesIfUserWantsTo();

        // Get current scene
        string current_scene = EditorSceneManager.GetActiveScene().path;

        // Save current scene path
        File.WriteAllText("Assets/Editor/active_scene.txt", current_scene);

        // Open main menu
        EditorSceneManager.OpenScene(
                    "Assets/Scenes/Main Menu.unity");

        // Play
        EditorApplication.isPlaying = true;
    }

    1 reference
    static void ModeChanged(PlayModeStateChange stateChange)
    {
        if (stateChange == PlayModeStateChange.EnteredEditMode) // Exited play mode
        {
            // Save scene
            EditorSceneManager.SaveCurrentModifiedScenesIfUserWantsTo();

            // Get saved scene path
            string current_scene = File.ReadAllText("Assets/Editor/active_scene.txt");

            // Open saved scene path
            if (current_scene != string.Empty && current_scene != EditorSceneManager.GetActiveScene().path) EditorSceneManager.OpenScene(current_scene);

            // Clear saved scene
            File.WriteAllText("Assets/Editor/active_scene.txt", string.Empty);
        }
    }
}
```

Flexible Save Games that Support Every Game Mode

I need to be able to save games of various game modes 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 game mode is just stored as a UID with extra data being saved by overriding the ‘`GetData`’ and ‘LoadData’ methods.

The default implementation for the GameManager:

```
1 reference
public virtual SerialisationData GetData()
{
    SerialisationData serialisationData = Board.GetData();
    serialisationData.GamemodeUID = GameManagerData.GetUID();
    return serialisationData;
}

1 reference
public virtual void LoadData(SerialisationData data)
{
    Board.LoadData(data);
}
```

`Board.GetData()` default implementation:

```
public virtual SerialisationData GetData()
{
    SerialisationData serialisationData = new SerialisationData();

    // Add data for every piece on board
    for (int x = 0; x < PieceBoard.GetLength(0); x++)
    {
        for (int y = 0; y < PieceBoard.GetLength(1); y++)
        {
            if (PieceBoard[x, y] is not null)
            {
                PieceSerialisationData data = PieceBoard[x, y].GetData();
                if (data is not null) serialisationData.PieceData.Add(data);
            }
        }
    }

    return serialisationData;
}
```

`Board.LoadData()` default implementation:

```
public virtual void LoadData(SerialisationData data)
{
    PieceBoard = new AbstractPiece[PieceBoard.GetLength(0), PieceBoard.GetLength(1)];

    // Get map of UIDs to pieces
    List<AbstractPiece> all_pieces = Util.GetAllPieces();
    Dictionary<int, Type> mapped_pieces = new Dictionary<int, Type>();
    foreach (AbstractPiece piece in all_pieces) mapped_pieces[piece.GetUID()] = piece.GetType();

    // Create instance of every piece in save data and place it on the board
    foreach (PieceSerialisationData piece_data in data.PieceData)
    {
        AbstractPiece new_piece = (AbstractPiece)Activator.CreateInstance(mapped_pieces[piece_data.UID], new object[] { piece_data.Position, piece_data.Team, this });
        new_piece.LoadData(piece_data); // Load piece data
        PieceBoard[piece_data.Position.X, piece_data.Position.Y] = new_piece;
    }
}
```

Piece default implementation:

```
5 references
public virtual PieceSerialisationData GetData()
{
    PieceSerialisationData data = new PieceSerialisationData();
    data.Team = Team;
    data.Position = Position;
    data.UID = GetUID();
    return data;
}

7 references
public virtual void LoadData(PieceSerialisationData data)
{
    Position = data.Position;
    Team = data.Team;
}
```

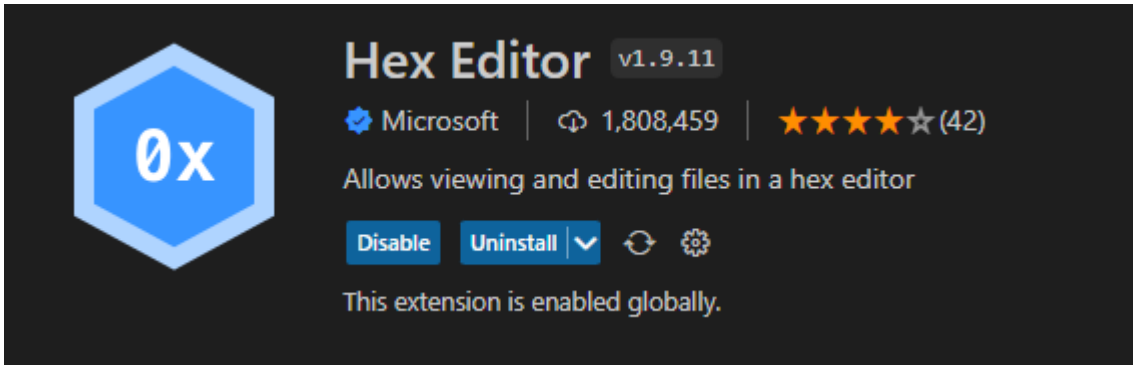
An example of custom piece data might look like this (custom data stored in `data.Data`):

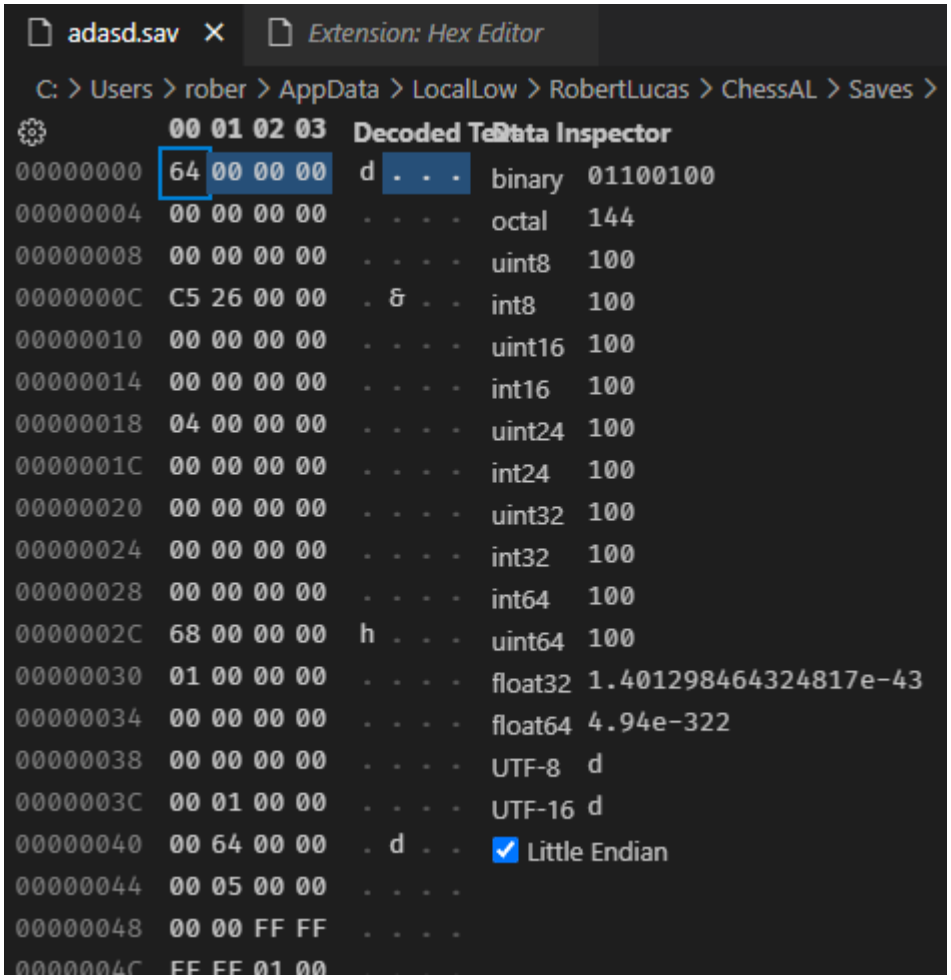
```
2 references
public override PieceSerialisationData GetData()
{
    PieceSerialisationData data = new PieceSerialisationData();
    data.Team = Team;
    data.Position = Position;
    data.UID = GetUID();
    data.Data = new byte[5];
    data.Data = ArrayExtensions.Merge(data.Data, BitConverter.GetBytes(HasMoved), 0);
    data.Data = ArrayExtensions.Merge(data.Data, BitConverter.GetBytes(DashMove), 1);
    return data;
}

4 references
public override void LoadData(PieceSerialisationData data)
{
    HasMoved = BitConverter.ToBoolean(ArrayExtensions.Slice(data.Data, 0, 1));
    DashMove = BitConverter.ToInt32(ArrayExtensions.Slice(data.Data, 1, 5));
    base.LoadData(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
Elapsed 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

Working with raw bytes is, of course, error-prone and I encountered many bugs trying to implement the save game system, without the luxury of C#'s detailed error messages as the issue was either incorrect data interpretation or there not being the right amount of bytes. Visual Studio Code's Hex Editor extension was incredibly useful for debugging these files.





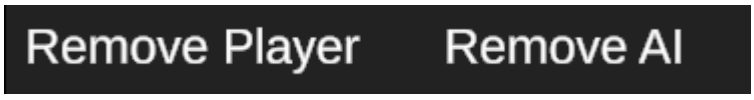
Making text sizes consistent

When UI components scale with screen size it is difficult to make the text scale correctly too. While Unity does have an ‘auto size’ option allowing the text to expand to fit its container, texts of different lengths will scale to different font sizes as, if there is less text, it can expand more.

To solve this, I created a ‘Font Size Inheritance’ script that gets the font size of a ‘parent’ text object and copies its font. This allows text to expand as much as possible while being constrained to staying the same size as a ‘parent’ text object.

```
/// <summary>
/// Class that makes text's font size tied to another font asset
/// </summary>
[RequireComponent(typeof(TMP_Text))]
Unity Script (7 asset references) | 0 references
public class InheritFontSize : MonoBehaviour
{
    [SerializeField] private TMP_Text InheritFrom;

    private TMP_Text _text = null;

    // Copy and update font size
    Unity Message | 0 references
    private void OnEnable()
    {
        if (_text is null) _text = GetComponent<TMP_Text>();
        _text.enableAutoSizing = false;
    }

    Unity Message | 0 references
    private void Start()
    {
        if (_text is null) _text = GetComponent<TMP_Text>();
        _text.enableAutoSizing = false;
    }

    Unity Message | 0 references
    private void OnValidate()
    {
        _text = GetComponent<TMP_Text>();
        _text.fontSize = InheritFrom.fontSize;
    }

    Unity Message | 0 references
    void Update()
    {
        if (_text.fontSize ≠ InheritFrom.fontSize)
        {
            _text.fontSize = InheritFrom.fontSize;
        }
    }
}
```

Example:



Here the ‘Remove AI’ text could expand more but doesn’t because it is constrained to being at most the size of the larger ‘Remove Player’ text.

Making the Listener Socket Shut Down Properly

This code waits for a user to connect

```
handler = listener.Accept();
```

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

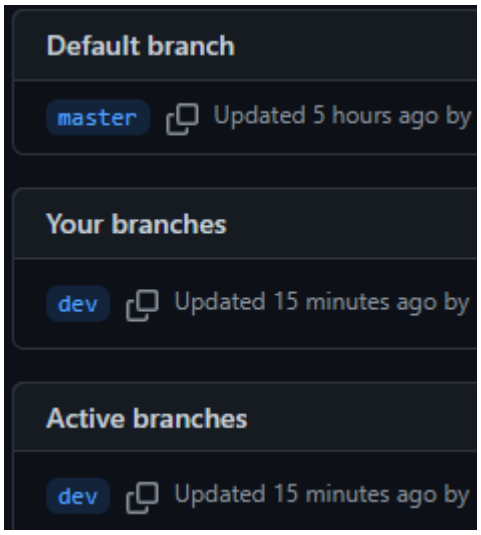
```
listener.Blocking = false;
// Recieve connection data
```

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

```
listener.Blocking = false;
// Recieve connection data
while (running)
{
    try
    {
        handler = listener.Accept(); // Throws socket exception code 10035 if none available
        break;
    }
    catch (SocketException se)
    {
        // No connection available
        if (se.ErrorCode == 10035)
        {
            Thread.Sleep(CLIENT_ACCEPT_COOLDOWN);
            continue;
        }
        else throw se;
    }
}
if (!running) return;
```

Keeping In-Dev and Stable Builds Separate (VCS)

There were a few occasions when 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 at all times. When the 'dev' branch reached a milestone like a feature being completed it would be merged into the 'master' branch.



Fixing the 'Ghost' User when Playing Online

This 'ghost' user which appeared on the scoreboard when joining an online game 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.

```
0 references
private void ForceUpdateLobby() ⇒ UpdateLobbyPlayerCardDisplay(chessManager.GetPlayerList());

/// <summary>
/// Updates player cards shown in a lobby
/// </summary>
/// <param name="playerData"></param>
10 references
public void UpdateLobbyPlayerCardDisplay(ConcurrentDictionary<int, ClientPlayerData> playerData)
{
    // Prevents errors caused by player data being sent too quickly
    if (chessManager.CurrentGameManager is null && playerData.Count ≠ 0)
    {
        Invoke("ForceUpdateLobby", 1f); // Run again in 1s
        return;
    }
```

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

Preventing Errors caused by Hosting Twice in Quick Succession

This was found to be due to the socket used for the server being in the '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 by iterating through all active TCP connections and checking if any of them are using the game's port.

```
1 reference
public bool Host(HostSettings settings, Action onPlayersChange, Action onGameStart)
{
    if (NetworkingUtils.PortInUse(NetworkSettings.PORT)) return false; // Check if port is in use
```

```
1 reference
public static bool PortInUse(int port)
{
    IPGlobalProperties ipProperties = IPGlobalProperties.GetIPGlobalProperties();
    IPEndPoint[] ipEndPoints = ipProperties.GetActiveTcpListeners();


    foreach (IPEndPoint endPoint in ipEndPoints)
    {
        if (endPoint.Port == port) // Port found
        {
            return true;
        }
    }
    return false;
}
```

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

```
if (next_player < 0) // A team has won
{
    if (GUtil.TurnDecodeTeam(next_player) ≠ gameData.LocalPlayerTeam)
    {
        if (maximising) score = float.MinValue;
        else return float.MinValue; // Best possible value for minimiser so instantly return
    }
    else
    {
        if (maximising) return float.MaxValue; // Best possible value for maximiser so instantly return
        else score = float.MinValue;
    }
}
```

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

Developing the Sound System

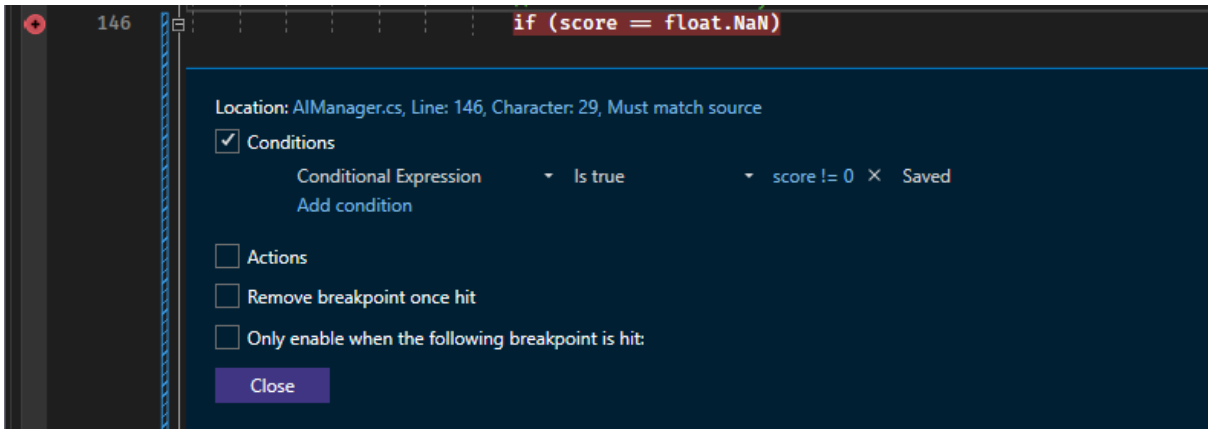
Originally, I was going to use a Unity 'AudioSource' to play sounds however this had two problems: it would be difficult to reference across scenes and it can only play one sound at a time. Instead, I used the Singleton design pattern to allow easy static referencing across scenes despite Unity not supporting 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.

```
public class SoundMananger : MonoBehaviour
{
    private static SoundMananger instance = null;

    [Header("Audio Clips")]
    [SerializeField] private AudioClip[] PieceMoveSounds;
    [SerializeField] private AudioClip OnClickSound;

    [Header("References")]
    [SerializeField] private GameObject Soundling; // Audio source

    Unity Message | 0 references
    private void Awake()
    {
        // Singleton setup
        if (instance is not null) DestroyImmediate(gameObject);
        else
        {
            instance = this;
            DontDestroyOnLoad(gameObject); // Persist between scenes
        }
    }
```

```
/// <summary>
/// Plays a sound
/// </summary>
/// <param name="clip"></param>
2 references
public void PlaySound(AudioClip clip)
{
    // Instantiate a 'soundling' to play sound so multiple sounds can play at once
    GameObject soundling = Instantiate(Soundling);

    // Set soundling to camera position to create stereo audio at proper volume
    soundling.transform.position = Camera.main.transform.position;

    // Set and play sound
    AudioSource audioSource = soundling.GetComponent<AudioSource>();
    audioSource.clip = clip;
    audioSource.Play();

    // Allow soundling to play across scenes
    DontDestroyOnLoad(soundling);

    // Delete the soundling once it is no longer needed
    StartCoroutine(DeleteSoundling(soundling, clip.length));
}

/// <summary>
/// Deletes the soundling after a delay
/// </summary>
/// <param name="soundling"></param>
/// <param name="delay"></param>
/// <returns></returns>
1 reference
IEnumerator DeleteSoundling(GameObject soundling, float delay)
{
    yield return new WaitForSeconds(delay + 1);
    Destroy(soundling);
}
```

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

```
// Exit if over time
if ((current_depth == max_depth - 1 || current_depth == max_depth) && IsOverTime()) return float.NaN;
```

```
// Return if algorithm is terminating
if (score is float.NaN) return float.NaN;
```

The reason for this not working is that

```
> float.NaN == float.NaN
false
```

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`

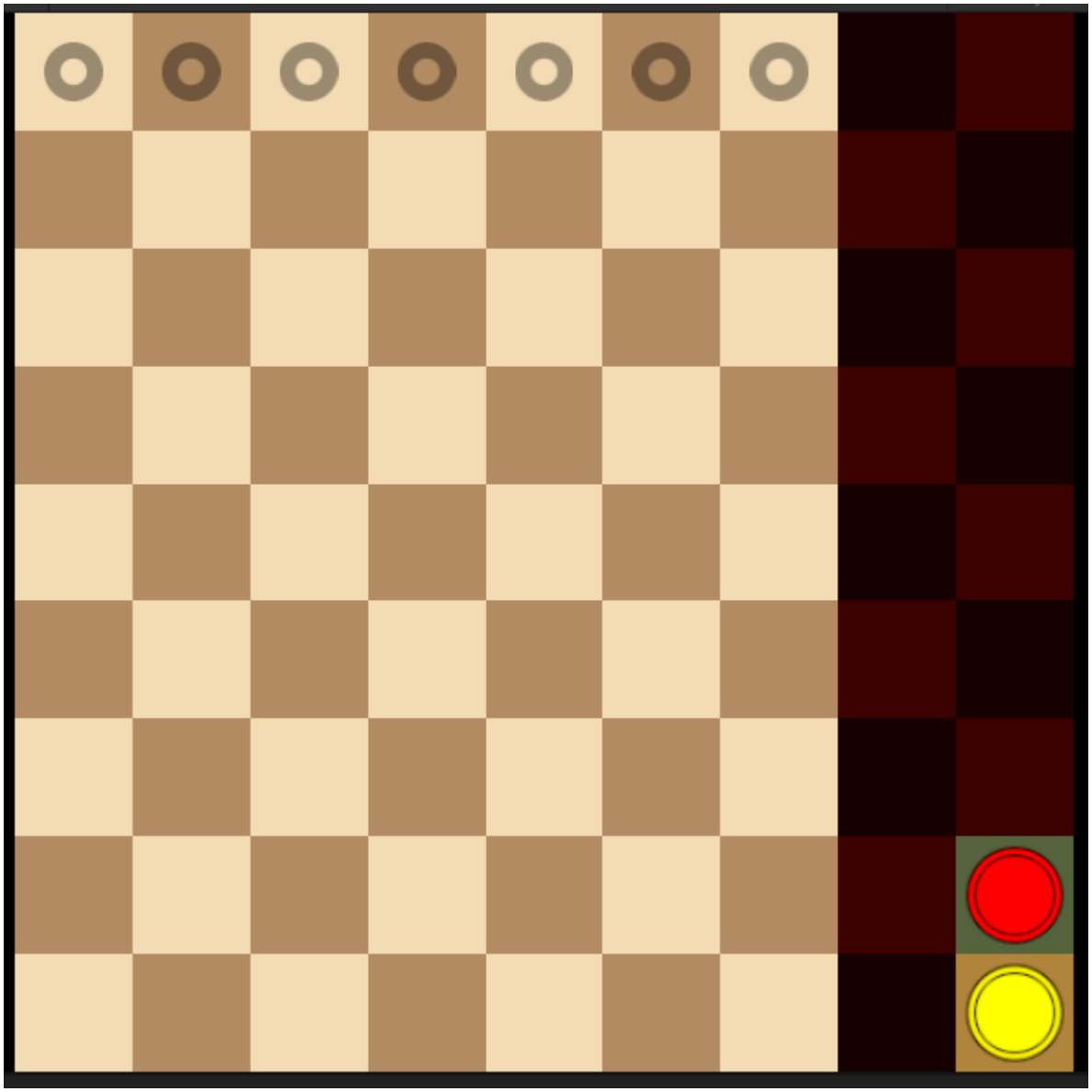
C# does have a different comparison operator 'is' which works here

```
> float.NaN is float.NaN
true
```

```
// AI terminated early
if (score is float.NaN)
{
    cancelled = true;
    break;
}
```

Fitting Connect Four into 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 counters 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 game mode



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

Fixing Public IP Bug

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. Analysing code differences, I found this line to be the problem:

```
// *** CAUSES ERRORS ***
// listener.SetSocketOption(SocketOptionLevel.IP, SocketOptionName.ReuseAddress, true);
```

I believed this line allowed an address to be reused which I wanted for the server. What this does in reality is 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.

```
// ThreadPool.QueueUserWorkItem(new WaitCallback(RunMiniMax),
//          new MiniMaxStruct(new_manager, new_game_data, new_manager.GetMoves(new_game_data), 1, max_depth, best_score, true, id));
// continue;

// Recursively search for moves
score = MiniMax(new_manager, new_game_data, new_manager.GetMoves(new_game_data), 1, max_depth, best_score, true);
```

^ Multithreaded (commented) and single-threaded code

Missing Chess AI Optimisations 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 game modes I would need to create a custom hashing implementation per game mode. 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 a fast dictionary as a hash.

The generalisation has also prevented me from using heuristics as these would also require a per-game-mode implementation.

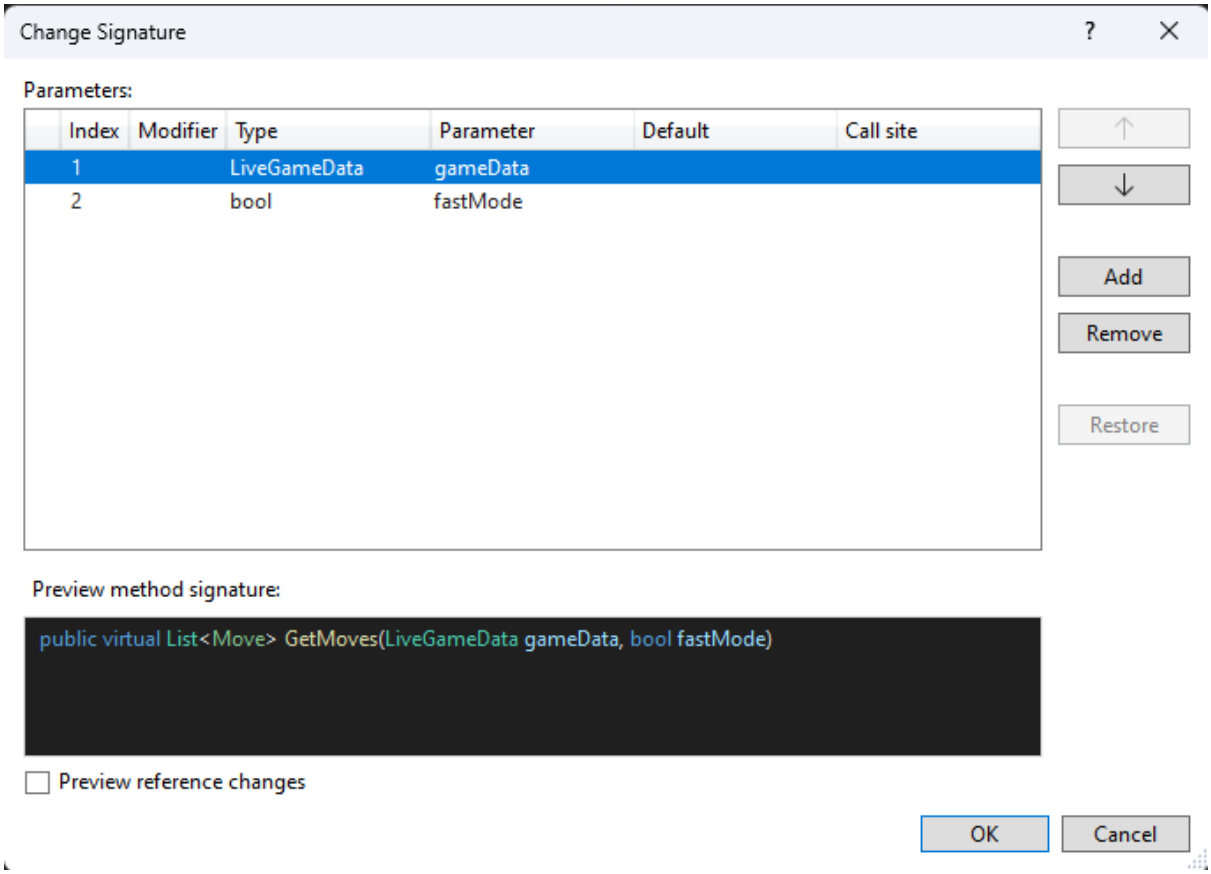
Allowing the AI to look Further Ahead when Playing Chess

What makes chess difficult for the AI to play is that for every possible move, it needs to check that this move doesn't put you in check. This results in finding moves going from an O(n) to O(n^2) time complexity. When traversing a move tree with millions of nodes this time increase compounds preventing the AI from looking that far ahead. To improve this, I added a `fastMode` parameter to the `GetMoves` which allowed me to avoid the test for check. To make the AI still care about check (as previously it only saw it as bad because no moves available meant a game loss) I gave the king a very high value and to stop it from making illegal moves, I made the first `GetMoves` not use the faster mode and only from the second layer on would it retrieve moves using fast mode.

```
public override List<Move> GetMoves(LiveGameData gameData, bool fastMode)
{
```

```
if (!fastMode)
{
    // Test if possible moves leave king in check
```

As `GetMoves` originates from an abstract class, it needed to be changed in many places. Visual Studio's 'Change Signature' feature allows you to add parameters to not just a class, but all classes that override the method too. It also allows you to set a default value for the new parameter in places that call the method. In my case, I set the default value to false as this wouldn't change anything and only set it to true where needed.

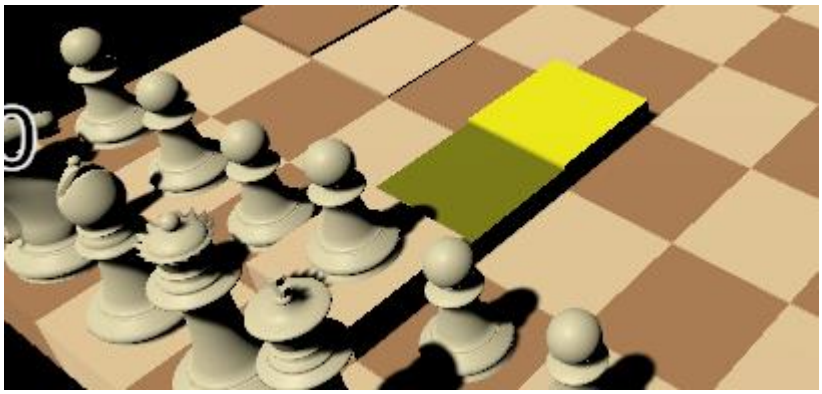


AI using fast mode:

```
List<Move> new_moves = new_manager.GetMoves(new_game_data, fastMode: true);
```

Making 3D Squares Move Smoothly

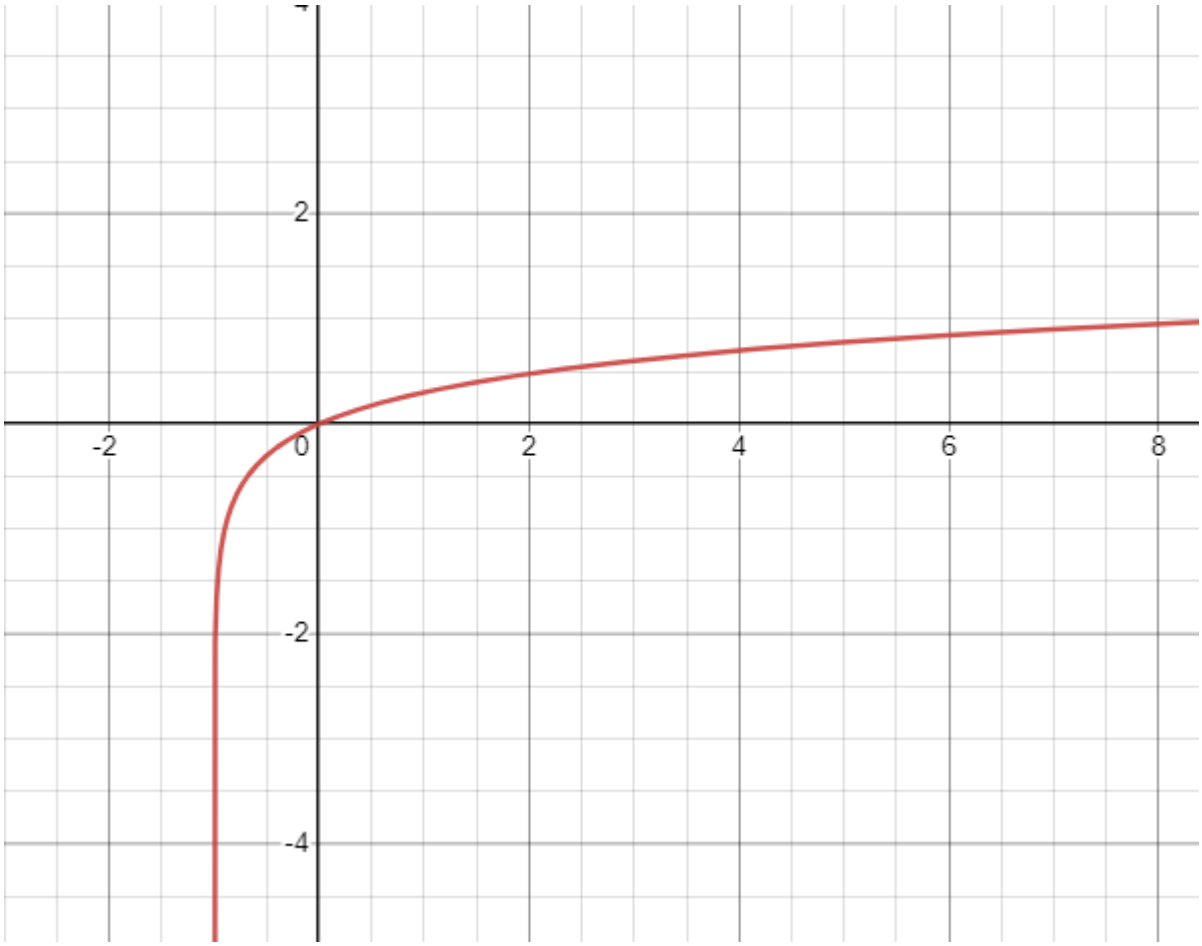
When a square with a piece on it is selected, I want the square to move up to show that the piece is selected like this:



I didn't however want it to jump up suddenly and it would be difficult to code an animation (such as one using a trigonometric wave) as these squares can be raised and lowered as needed based on what the cursor is hovering over, what moves are available and based on the ripple system. I, therefore, needed a way to smoothly move a square from its current height to any other.

To achieve this, I used a Log function as it gets lower the smaller the input maxing the square slow down as it approached its target position

Log(x-1):



Code:

```
float target_delta = targetDisplacement[x, y] + 1 - squares[x, y].transform.position.y;
if (target_delta == 0) { return; }
squares[x, y].transform.position += Vector3.up * Mathf.Log(target_delta, 2) * Time.deltaTime * 10;
squares[x, y].transform.position = new Vector3(squares[x, y].transform.position.x, Mathf.Clamp(squares[x, y].transform.position.y, -10, 10), squares[x, y].transform.position.z);
```

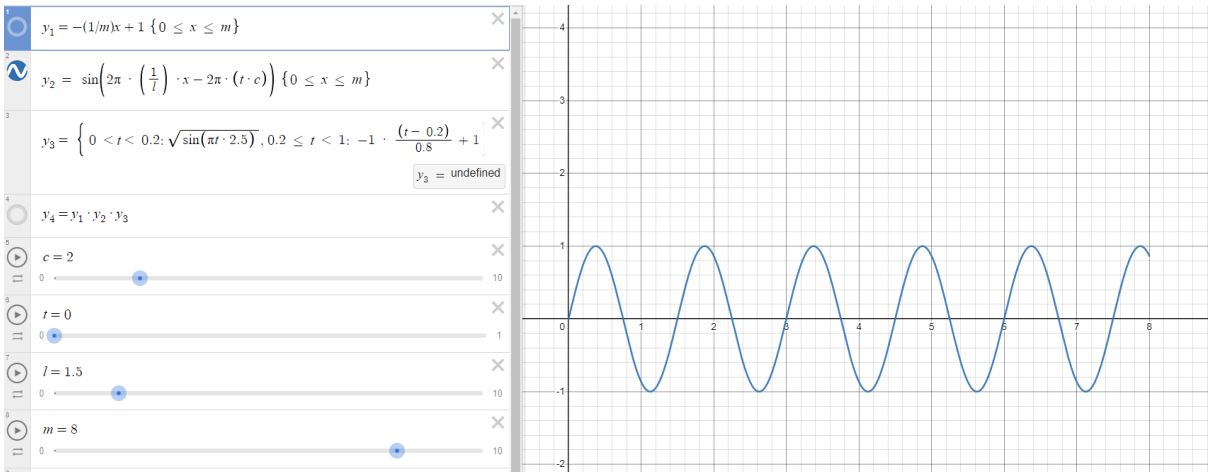
Implementing the 3D Ripple Effect

Some positive feedback on a previous game I made led me to decide to implement a 3D interface to the game. One of the most well-received features of the previous game was a 'ripple' effect after a piece was taken. This was, however, made up of hardcoded animation frames and wouldn't work with this game's need to support multiple board sizes.

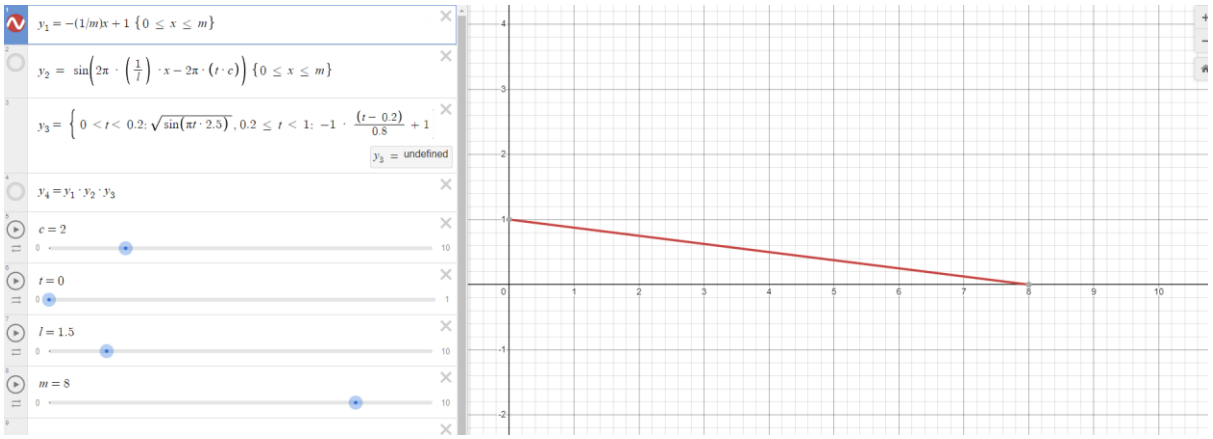
Variable	Description
X	Distance from ripple centre
Y	Height
C	Cycles (i.e waves per ripple)
T	Time (0 – 1, can be scaled to make it longer)
L	Individual wave length
M	Maximum ripple distance

To achieve this effect, I first used a sin wave (y_2). The 2pi * (1/L) controls the width of the sin wave

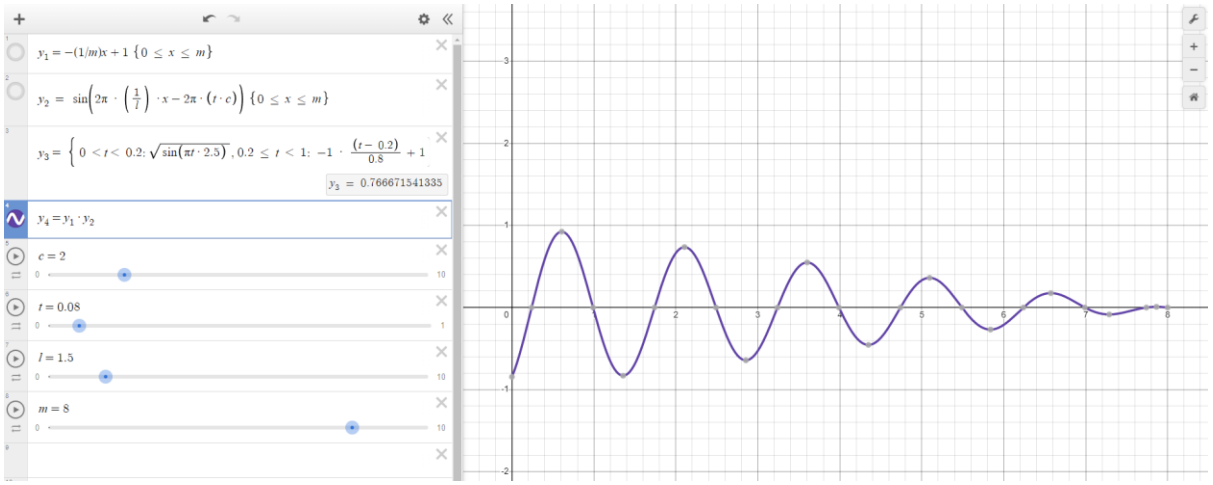
And the 2pi*(t * c) makes the wave cycle C times as T goes from 0 -> 1



Next, I multiplied y_2 by y_1: a straight line that starts at (0, 1) and ends at (m, 0) to make the effect of the ripple weaker as it got further from the origin

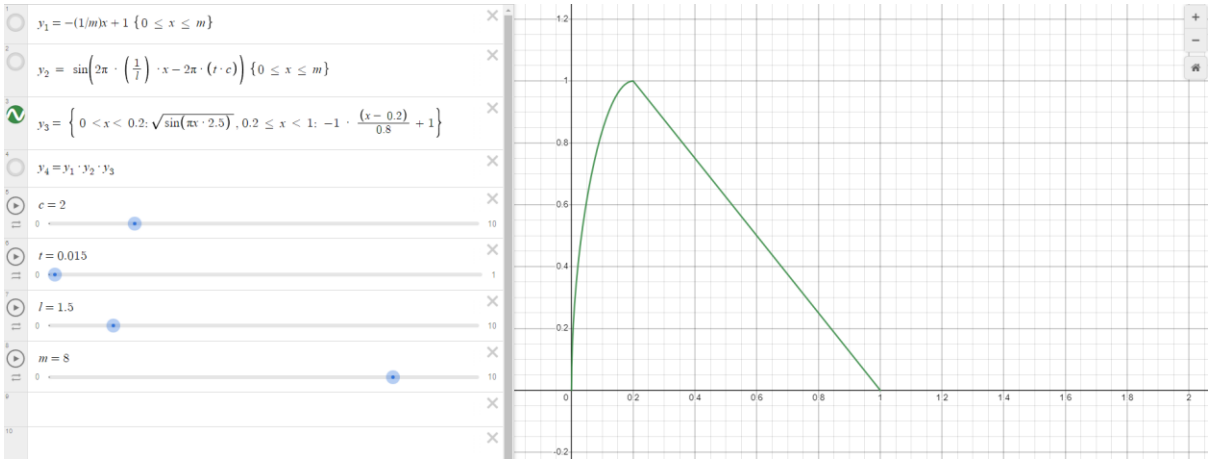


$y_1 * y_2$:

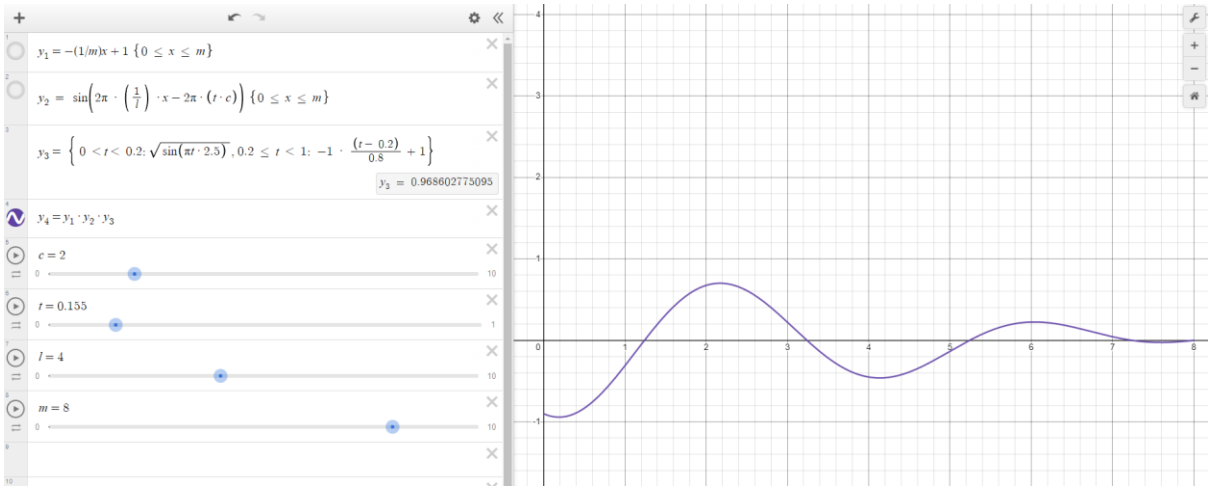


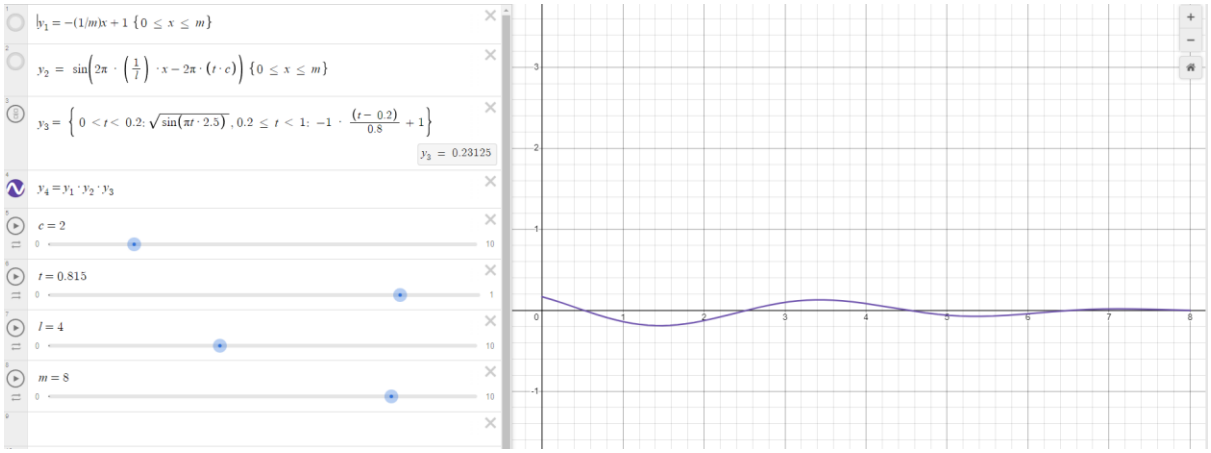
Finally, I multiplied it by y_3, a line that between t = 0 and t = 0.2 follows the first quarter of the root of a sin wave and after that falls to 0 linearly by t = 1, to make the squares not jump into their initial wave positions and to make the wave disappear over time.

y_3 (x is time, not distance):



These three graphs multiplied together give this:



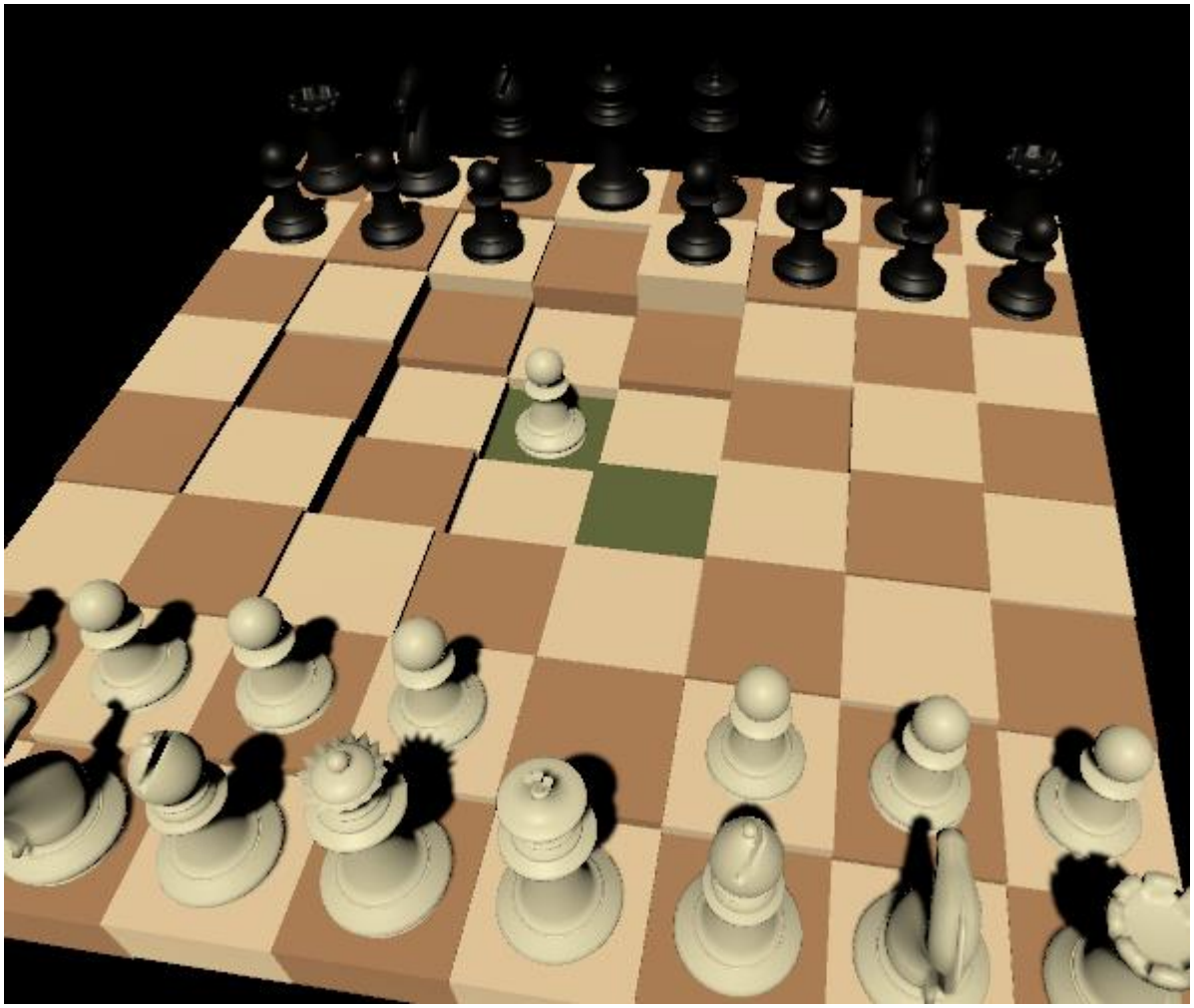


Note that the graph has a small amplitude at the start and end of t to ease in and out of it.

The program code looks like this:

```
/// <summary>
/// Creates ripples between an origin point and max distance
/// </summary>
/// <param name="time">How much time (0 - 1) has passed relative to the max time</param>
/// <param name="cycles">How many ripples should be created</param>
/// <param name="distance">How far the point is from the origin</param>
/// <param name="max_distance">The furthest point the ripples can reach</param>
/// <param name="ripple_length">How long each ripple should be</param>
/// <returns>Height at given distance</returns>
1 reference
public static float Ripple(float time, float cycles, float distance, float max_distance, float ripple_length)
{
    // y = -(1 / max_distance)x + 1
    float amplitude_falloff = -(1 / max_distance) * distance + 1;

    // y = sin(2pi * (1/ripple_length) * x - 2pi * (t * c))
    float ripple = Mathf.Sin(2 * Mathf.PI * (1 / ripple_length) * distance - 2 * Mathf.PI * (time * cycles));

    float overall_amplitude;
    if (time < 0.2f)
    {
        overall_amplitude = Mathf.Sqrt(Mathf.Sin(Mathf.PI * time * 2.5f));
    }
    else
    {
        overall_amplitude = (-1 * ((time-0.2f) / 0.8f)) + 1;
    }

    return ripple * amplitude_falloff * overall_amplitude;
}
```

Final result (immediately after taking the piece)



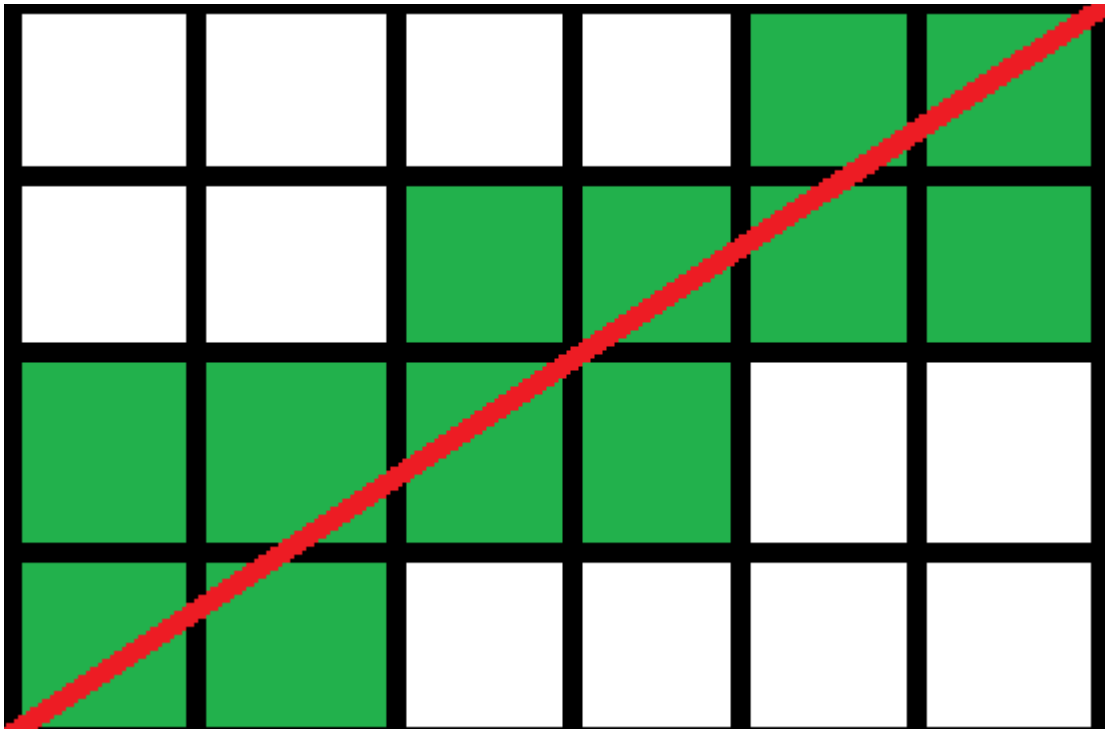
Making Pieces Jump

I found that pieces sliding through each other in the 3D mode didn't look nice so I decided to make pieces jump over this. To keep the modularity of the program intact, I couldn't write a per-game-mode ruleset for whether a piece should jump. Instead, I programmatically determined whether there was a piece between the start and end positions and, if there was, make the piece jump.

I detected whether there was a piece between the start and end using this code:

```
// Iterate through squares between start and end. If piece found between, jump
for (int i = 0; i < count; i++)
{
    Vector2 current_pos = from.Vector2() + (delta * (i / (float)count));
    V2 min_pos = new V2((int)Mathf.Floor(current_pos.x), (int)Mathf.Floor(current_pos.y));
    V2 max_pos = new V2((int)Mathf.RoundToInt(current_pos.x), (int)Mathf.RoundToInt(current_pos.y));
    if (min_pos ≠ from && min_pos ≠ to && pieces.ContainsKey(min_pos))
    {
        jump = true; break;
    }
    if (max_pos ≠ from && max_pos ≠ to && pieces.ContainsKey(max_pos))
    {
        jump = true; break;
    }
}
```

This iterates along a straight line between the start and end and at each point floors and ceilings (rounds down and rounds up) the position ensuring that all squares along the path are checked



Working Out which Pieces have Moved

As my game needs to feature pieces sliding, I need to figure out which piece has moved where. While I could have done this by making the GameManager provide extra data about a move however this would add more requirements for the GameManager that I would have to implement across all current and future game modes. It also complicates situations where multiple pieces are moving such as castling in chess. To solve this, I constructed a hash table of current piece positions and then tracked which pieces had moved.

This snippet also contains code to check for changes in appearance and apply them:

```
// Rebuild hash table
for (int x = 0; x < boardRenderInfo.BoardSize; x++)
{
    for (int y = 0; y < boardRenderInfo.BoardSize; y++)
    {
        if (ChessManager.GameManager.Board.PieceBoard[x, y] is null)
        {
            currentBoardHash[x, y] = 0;
            currentBoardAppearance[x, y] = 0;
        }
        else
        {
            currentBoardHash[x, y] = ChessManager.GameManager.Board.PieceBoard[x, y].GetHashCode();

            if (currentBoardAppearance[x, y] != ChessManager.GameManager.Board.PieceBoard[x, y].AppearanceID)
            {
                VisualManager3D.UpdateAppearance(new V2(x, y));
                UpdatePiece(ChessManager.GameManager.Board.PieceBoard[x, y]);
            }
            currentBoardAppearance[x, y] = ChessManager.GameManager.Board.PieceBoard[x, y].AppearanceID;
        }
    }
}
```

I then use this code to check whether a piece has been moved and, if so, apply this move:

```
void CheckSlide(V2 from, V2 to)
{
    if (ChessManager.GameManager.Board.PieceBoard[to.X, to.Y] is not null &&
        currentBoardHash[from.X, from.Y] == ChessManager.GameManager.Board.PieceBoard[to.X, to.Y].GetHashCode())
    {
        // 2D Move
        GameObject piece = pieces2D[from];
        piece.GetComponent<PieceController2D>().MoveTo(to, from, pieceTravelTime);

        if (boardRenderInfo.Allows3D)
        {
            // 3D destroy if destination not empty
            if (currentBoardHash[to.X, to.Y] != 0) VisualManager3D.DestroyPiece(to);
            // 3D Move
            VisualManager3D.Move(from, to);
        }

        // 2D Move
        pieces2D.Remove(from);

        // 2D destroy if destination not empty
        if (currentBoardHash[to.X, to.Y] != 0)
        {
            currentBoardHash[to.X, to.Y] = 0;
            Destroy(pieces2D[new V2(to.X, to.Y)]);
            pieces2D.Remove(new V2(to.X, to.Y));
        }

        // 2D Move
        pieces2D.Add(to, piece);

        // Hash table updates
        currentBoardHash[to.X, to.Y] = currentBoardHash[from.X, from.Y];
        currentBoardHash[from.X, from.Y] = 0;
        currentBoardAppearance[to.X, to.Y] = currentBoardAppearance[from.X, from.Y];
        currentBoardAppearance[from.X, from.Y] = 0;
    }
}
```

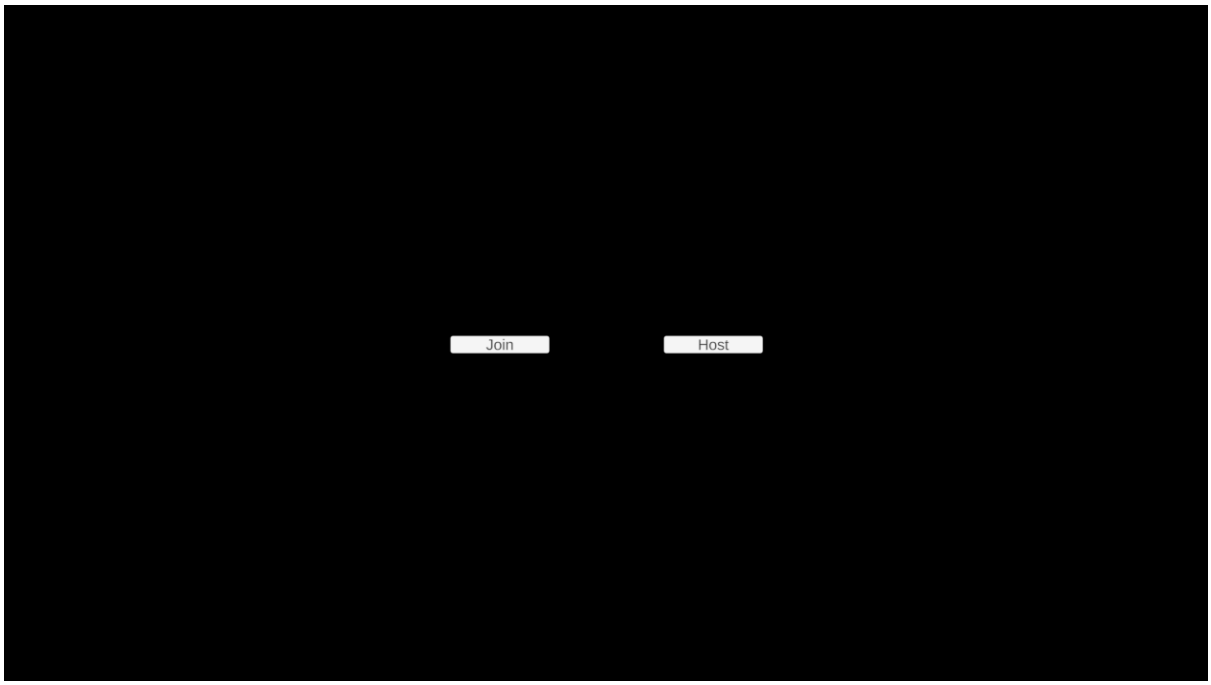
To ensure the maximum chance of the move being interpreted, I first check if a piece has been moved from Move.From to Move.To, then check all other pieces for sliding and resolve the rest of the differences by making pieces appear and disappear with no transition:

```
// Check for sliding moves
if (hasMove) CheckSlide(move.From, move.To);

// Check for more sliding moves
for (int x1 = 0; x1 < boardRenderInfo.BoardSize; x1++)
{
    for (int y1 = 0; y1 < boardRenderInfo.BoardSize; y1++)
    {
        // If empty continue
        if (currentBoardHash[x1, y1] == 0 ||
            currentBoardHash[x1, y1] == (ChessManager.GameManager.Board.PieceBoard[x1, y1]?.GetHashCode() ?? 0))
                continue;

        for (int x2 = 0; x2 < boardRenderInfo.BoardSize; x2++)
        {
            for (int y2 = 0; y2 < boardRenderInfo.BoardSize; y2++)
            {
                CheckSlide(new V2(x1, y1), new V2(x2, y2));
            }
        }
    }
}

// Resolve difference between previous and current board state
for (int x = 0; x < boardRenderInfo.BoardSize; x++)
{
    for (int y = 0; y < boardRenderInfo.BoardSize; y++) {
        if (currentBoardHash[x, y] != (ChessManager.GameManager.Board.PieceBoard[x, y]?.GetHashCode() ?? 0))
        {
            if (currentBoardHash[x, y] != 0)
            {
                // Destroy if already occupied
                Destroy(pieces2D[new V2(x, y)]);
                pieces2D.Remove(new V2(x, y));
                if (boardRenderInfo.Allows3D) VisualManager3D.DestroyPiece(new V2(x, y));
            }
            if (ChessManager.GameManager.Board.PieceBoard[x, y] is not null)
            {
                // Place piece
                AddPiece(ChessManager.GameManager.Board.PieceBoard[x, y]);
                currentBoardAppearance[x, y] = ChessManager.GameManager.Board.PieceBoard[x, y].AppearanceID;
                if (boardRenderInfo.Allows3D) VisualManager3D.Create(ChessManager.GameManager.Board.PieceBoard[x, y]);
            }
        }
    }
}
```

Screenshots from Development

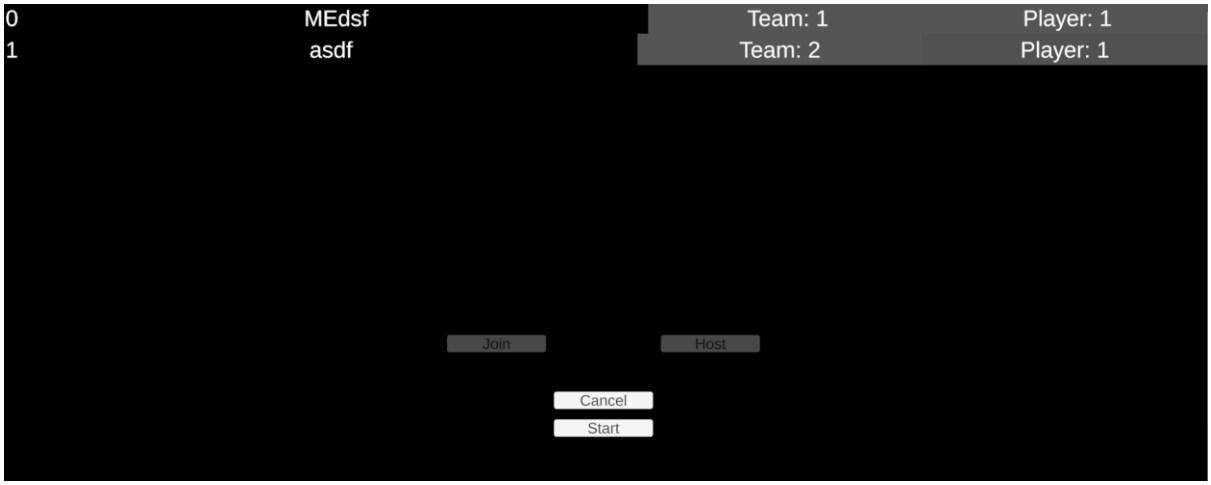
First Version with UI

Main menu:



<- Host screen | Join screen ->

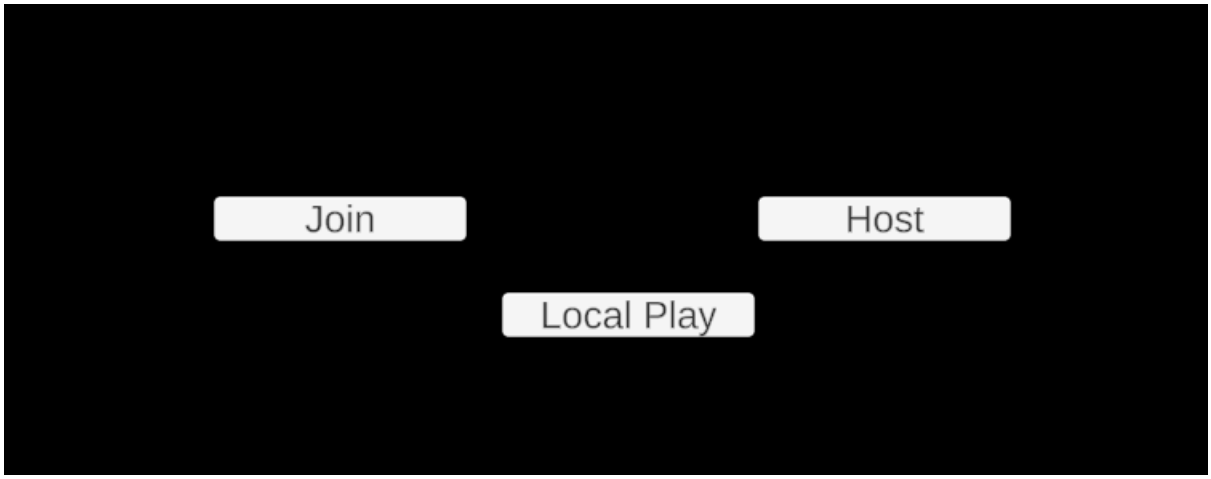
Lobby UI:



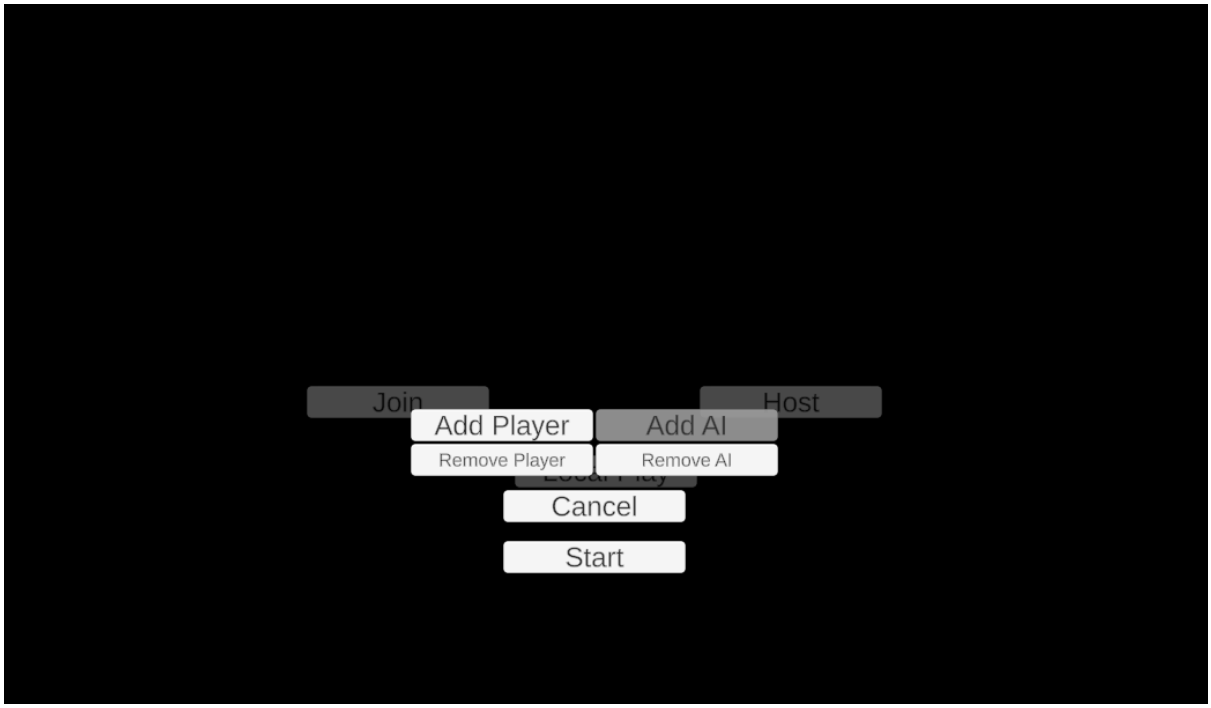
In-game:

First Version with Help System and Save System

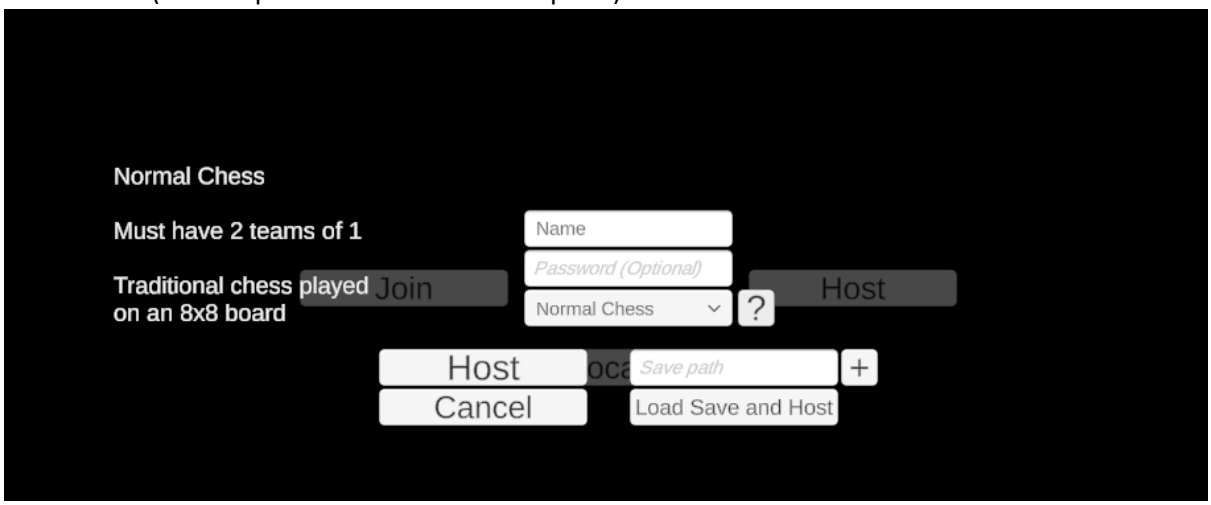
Main menu (local play not currently functional):



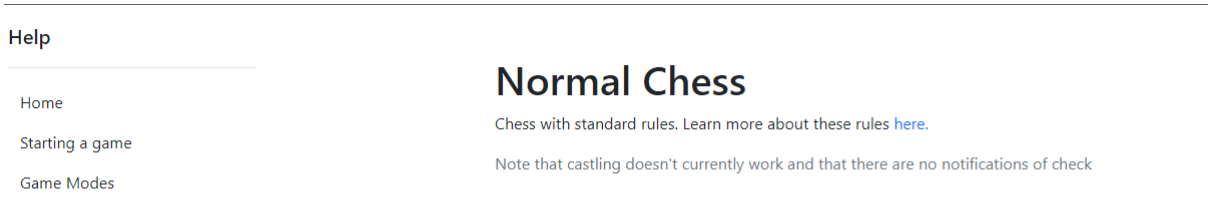
Local play screen (not functional):



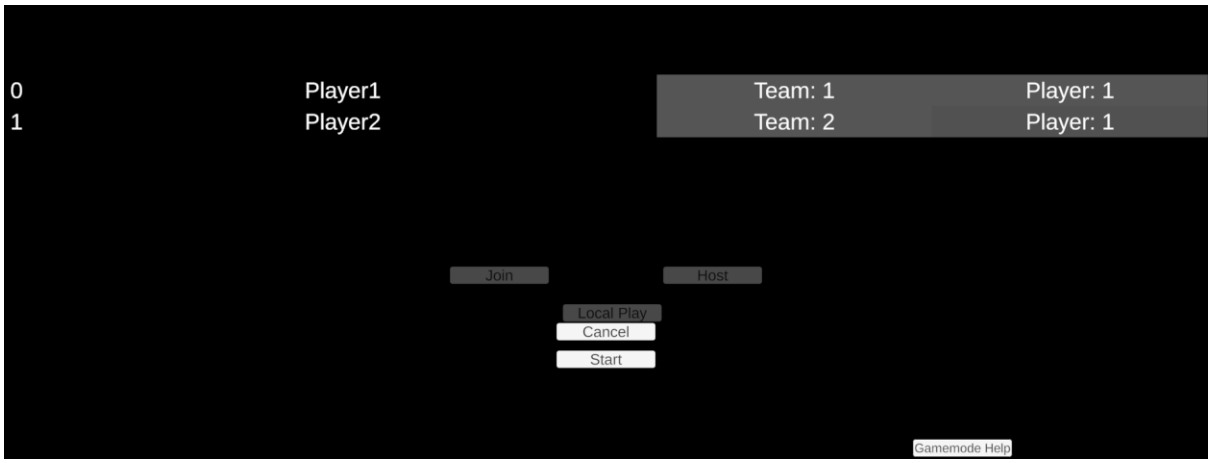
Host screen (new help button and load save option):



Help Screen:

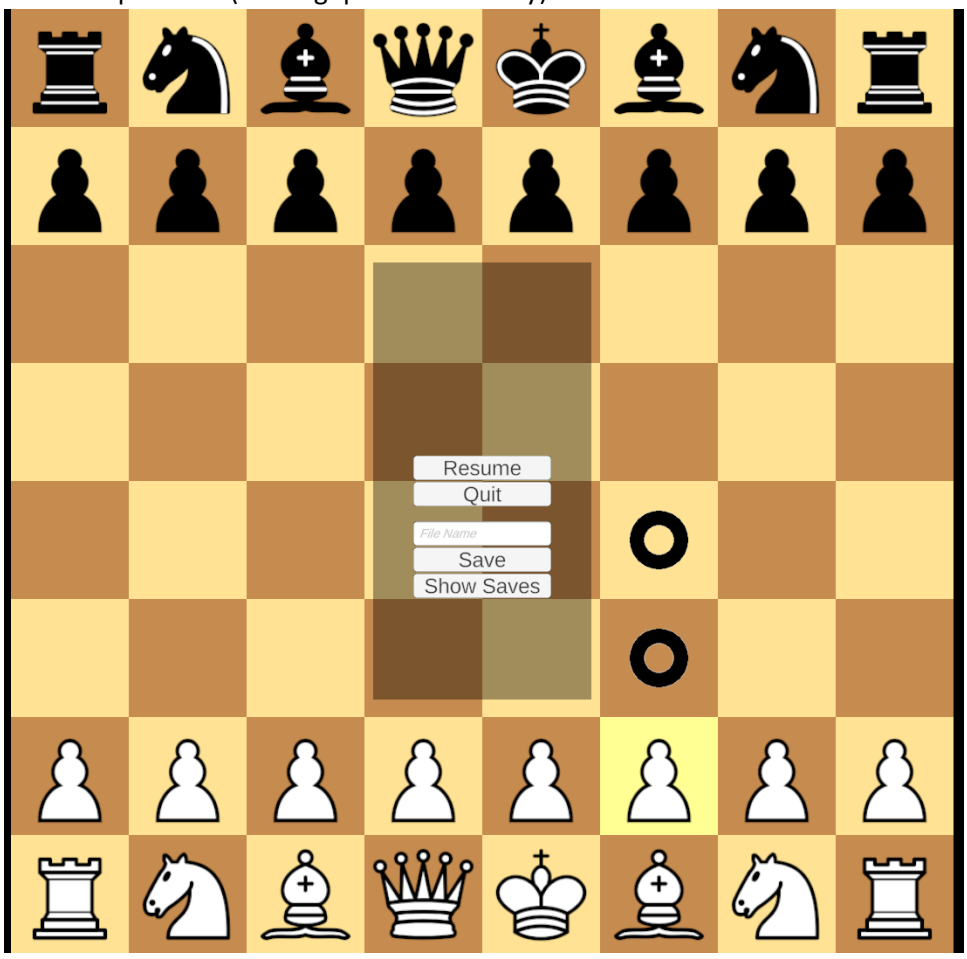


Lobby screen (with adjusted layout and game mode help button):



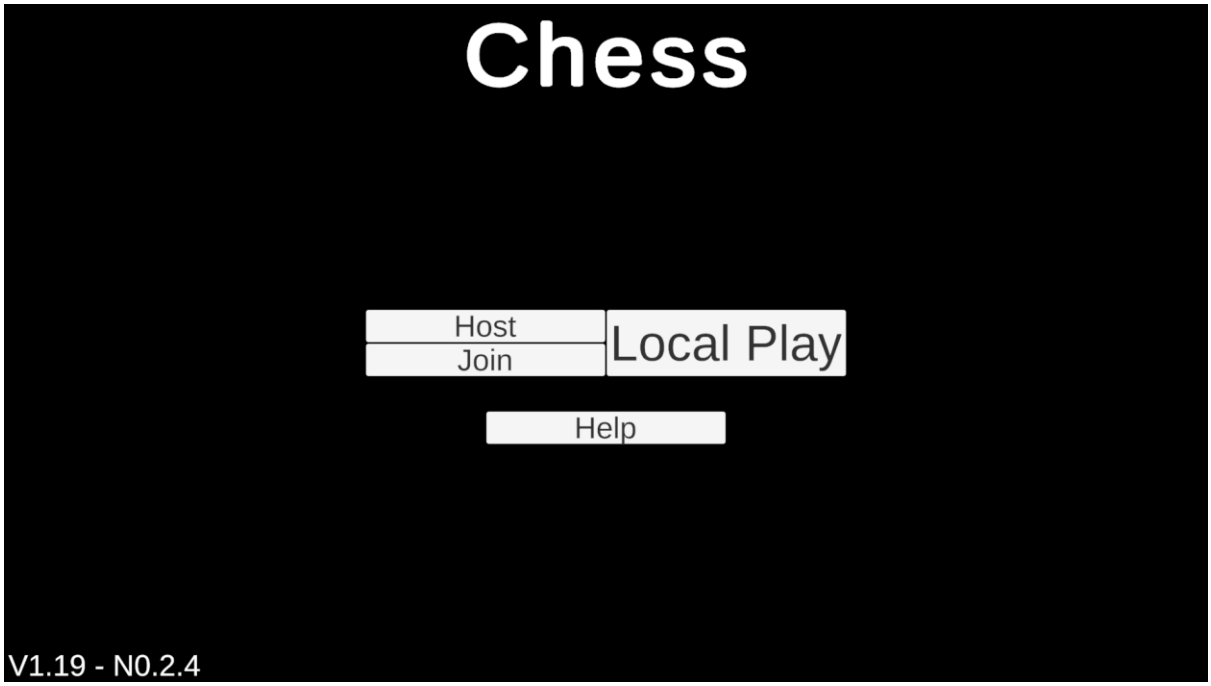
In-game (new timer and turn indicator):

New escape menu (missing quit functionality):

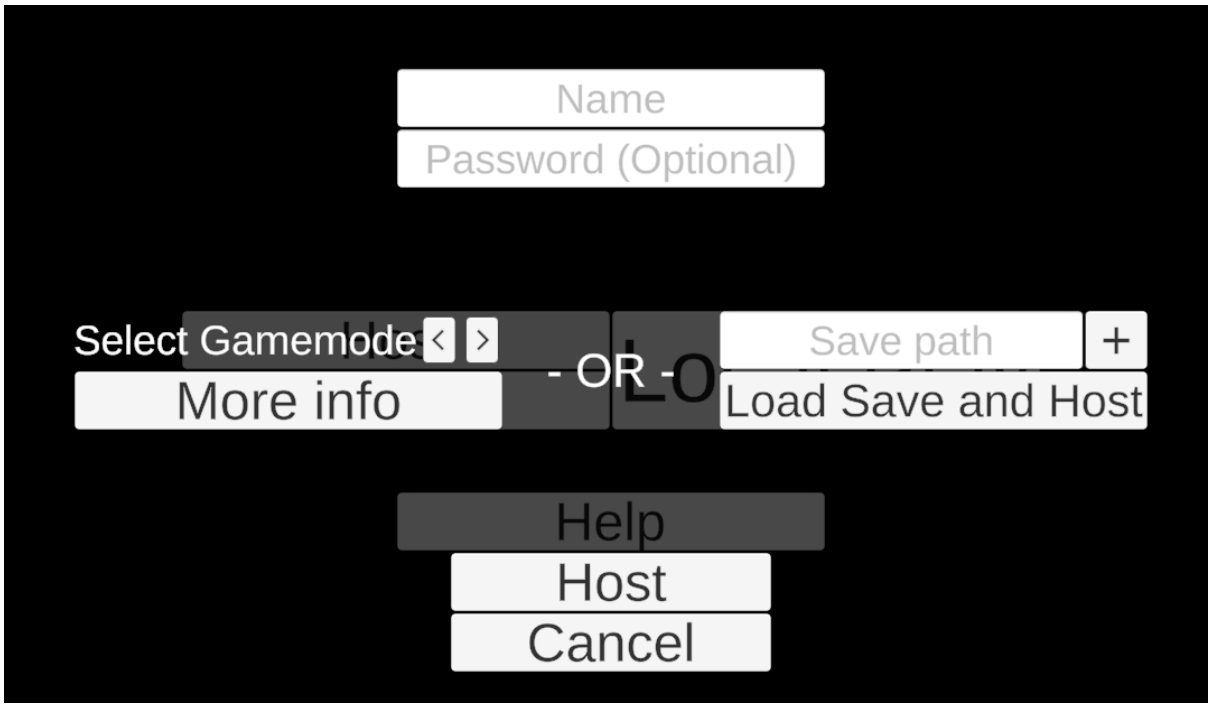


Large UI Improvements, Functional Local Play and AI Added

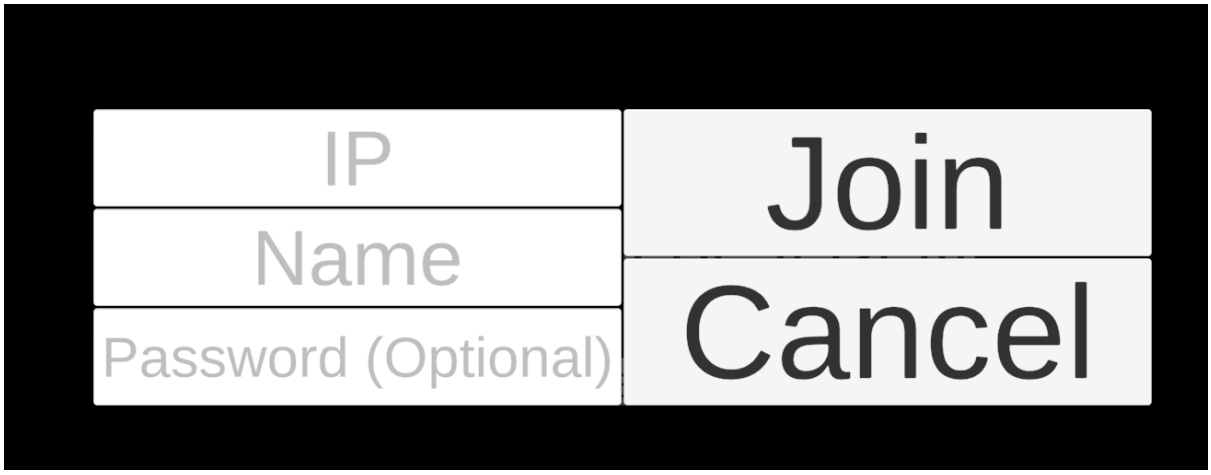
Main menu:



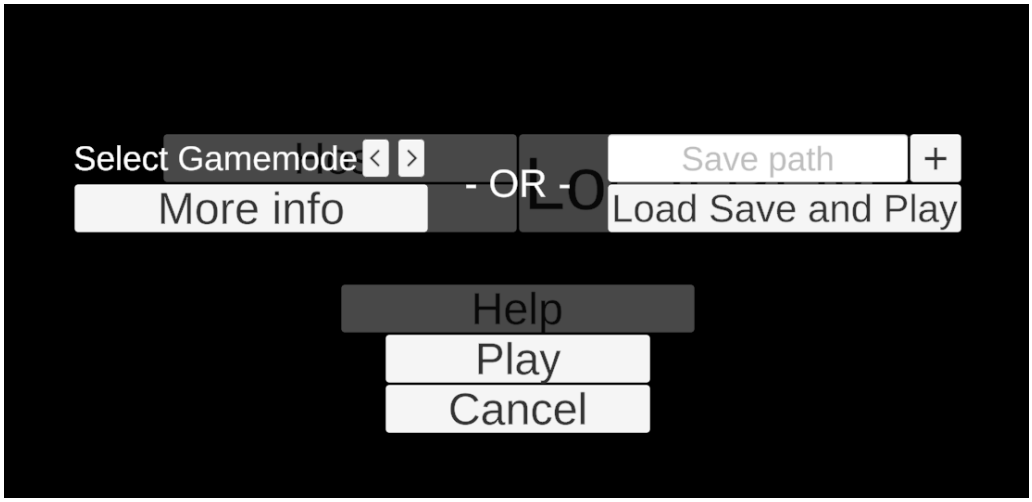
Host screen:



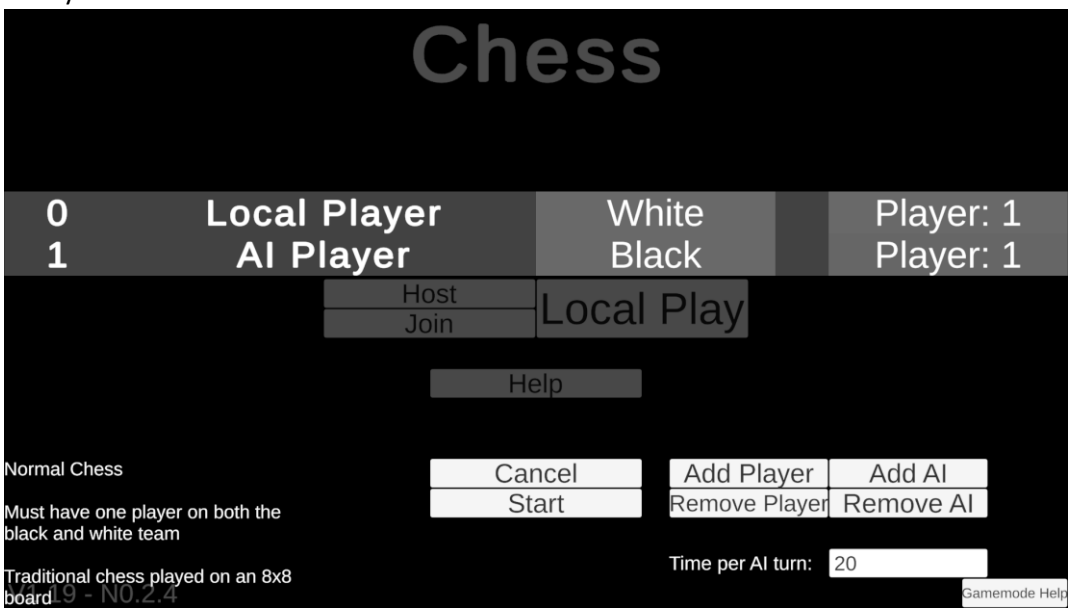
Join screen:



Local play screen:



Lobby screen:

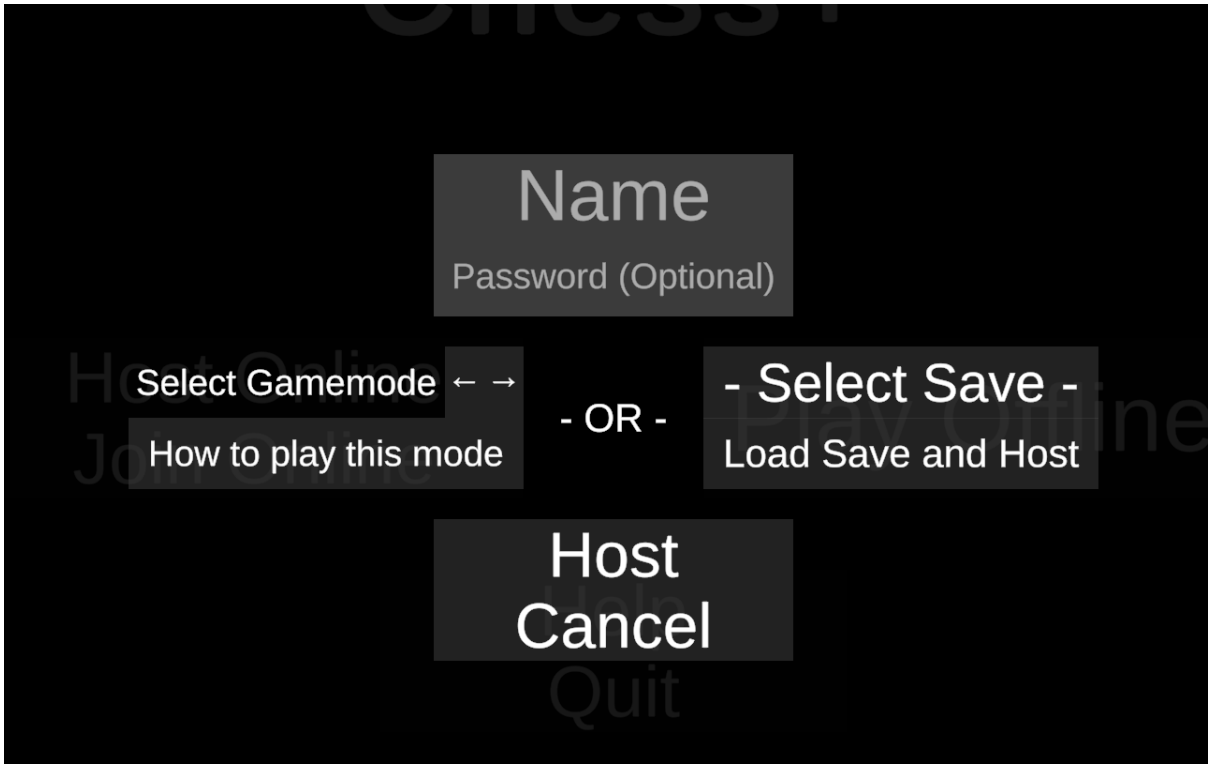


New escape menu scaling:

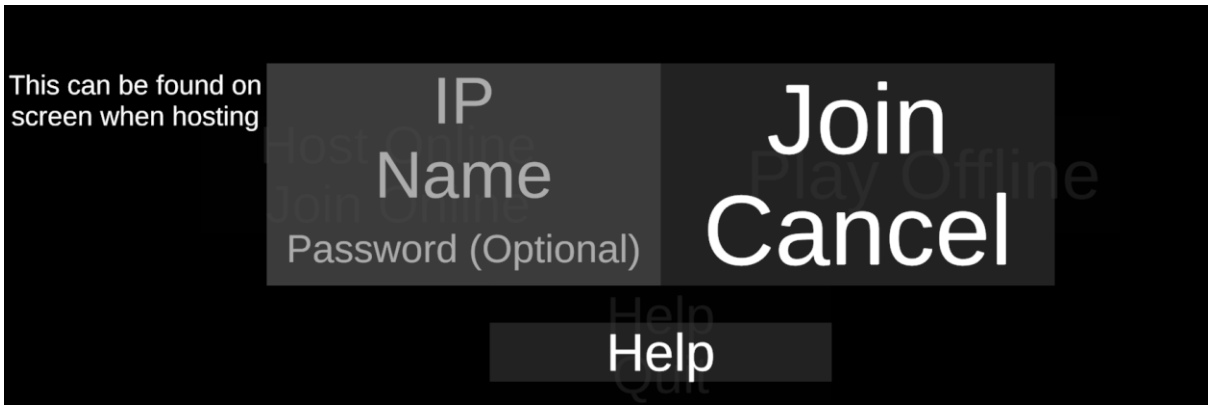
Large UI Overhaul

Main menu:

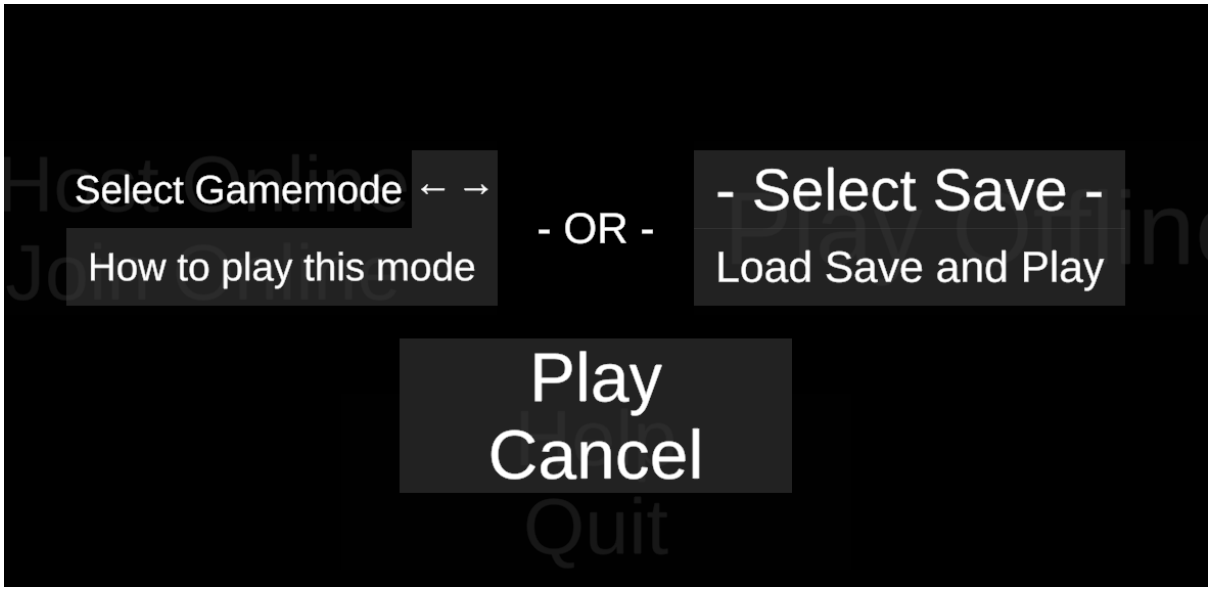
Host screen:



Join screen:

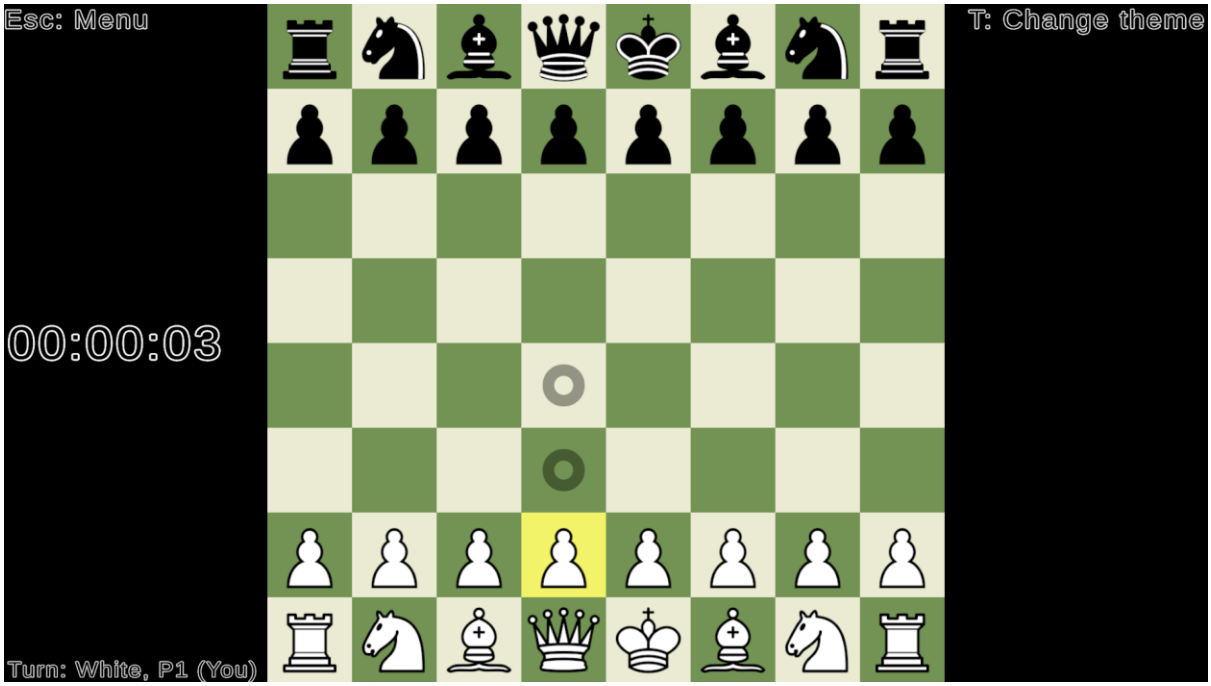


Local play screen:



Lobby:

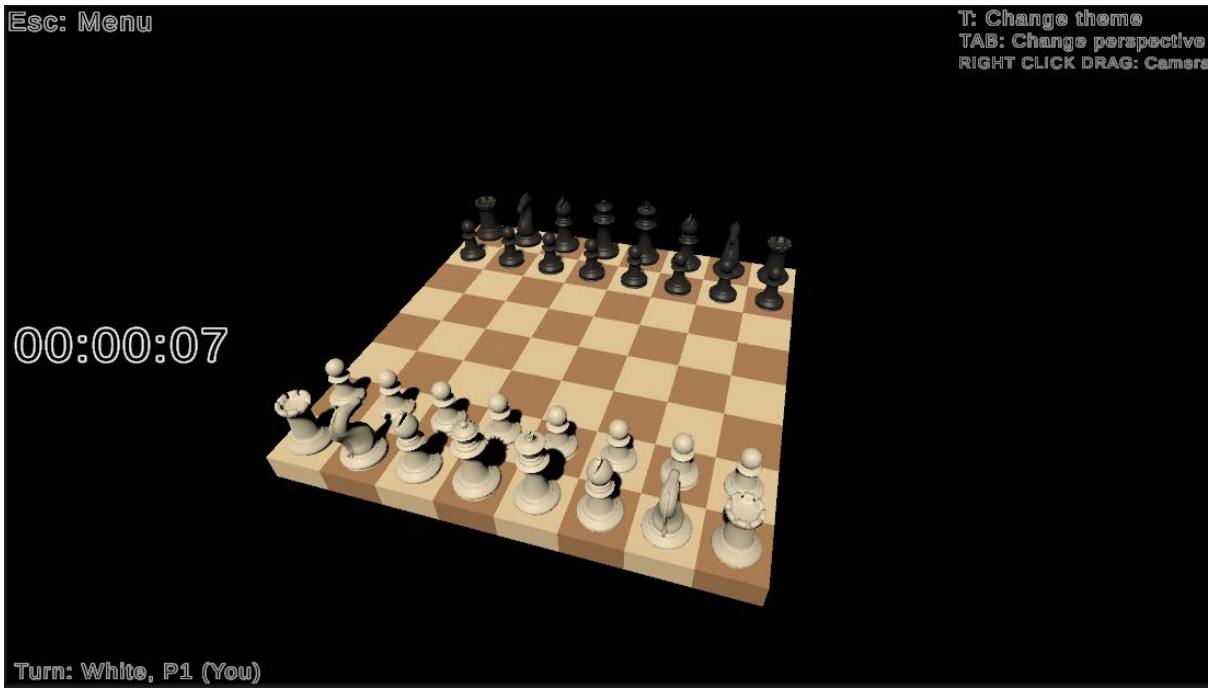
In Game:



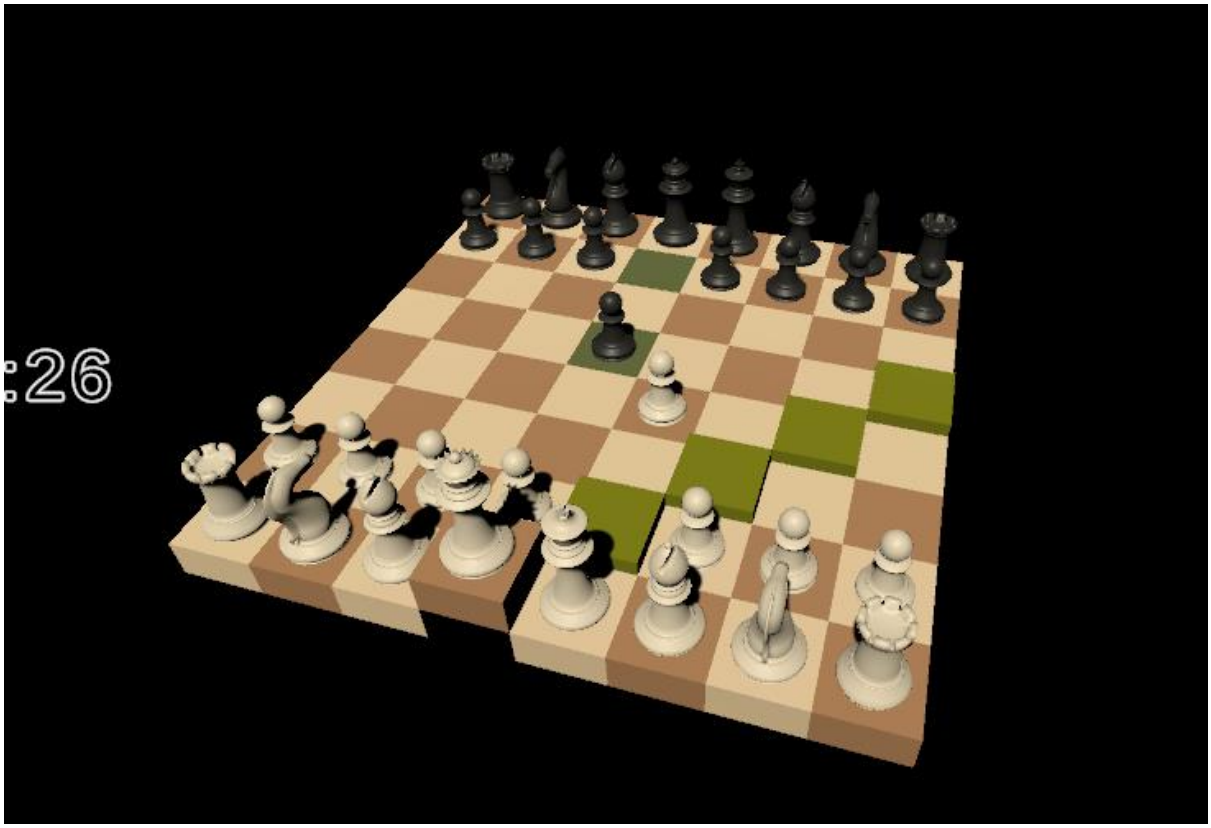
Multiple themes:

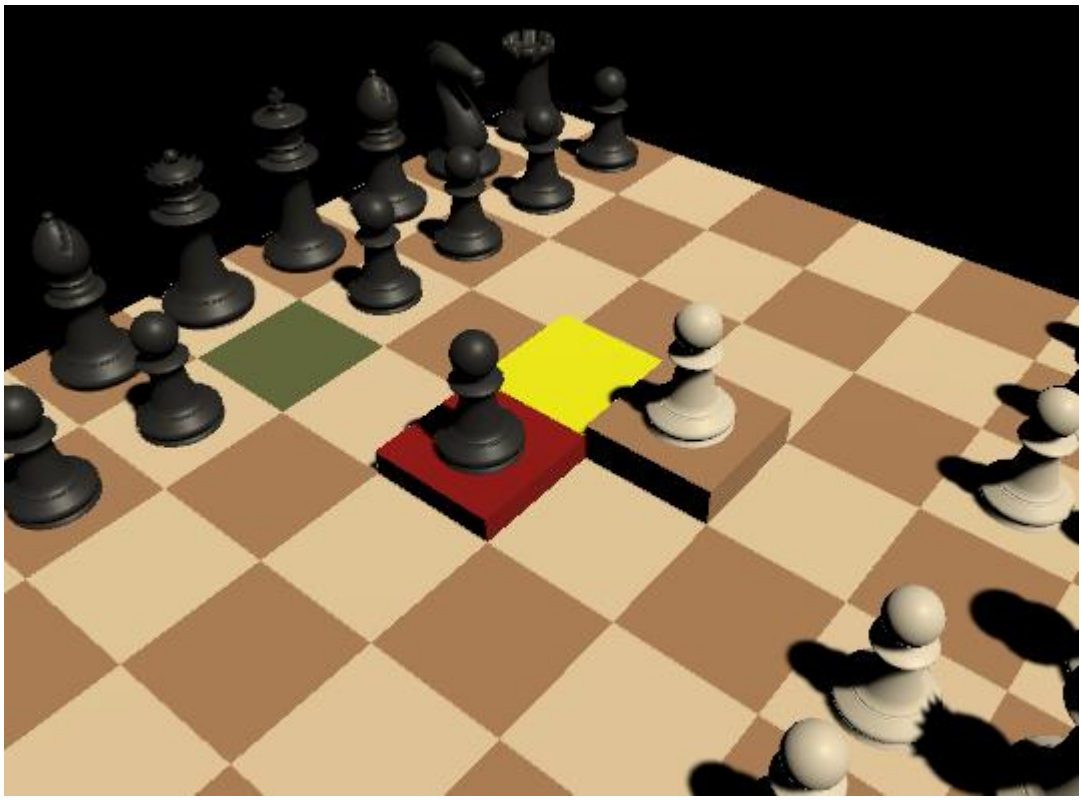
Escape menu:

3D Mode



Shows last move and possible moves:


:26



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.

For a more detailed view of how the systems interact see: Design Flowcharts and Details

ChessManager

The ChessManager handles interactions between different systems to maintain a tree reference structure however this leads to a large number of methods that are just encapsulations which will be omitted from the data table. To see what these methods do view the methods they encapsulate.

The ChessManager also handles interactions between other threads and Unity. As most interactions with Unity must happen on the main thread, the ChessManager queues method calls and runs them on the next Unity frame.

So this:

```
private void HostStartGameFail(string reason) => mainThreadActions.Enqueue(() => { menuUIManager.HostStartGameFailed(reason); });
```

Is an encapsulation of `menuUIManager.HostStartGameFailed(reason)` and will be run with this code on the next frame:

```
// Called every frame
public void Update()
{
    /*
     * Most unity functions can only be called from the main thread so this
     * goes through functions queued by other threads to run them
     * on the main thread during the next frame
     */

    Queue<Action> temp_main_thread_actions = mainThreadActions;
    mainThreadActions = new Queue<Action>();

    while (temp_main_thread_actions.Count > 0)
    {
        temp_main_thread_actions.Dequeue().Invoke();
    }
}
```

For more information about this solution see: Main Thread Queueing

Game Mode System

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

The list of GameManagers (actually GameManagerDatas) is retrieved using the 'GetAllGameManagers' method in the static Util class. It uses reflection to retrieve a list of all classes that inherit from AbstractGameManagerData and uses the Activator class to instantiate them.

```
/// <summary>
/// Retrieves a list of all subclasses of AbstractGameManagerData
/// </summary>
/// <returns></returns>
3 references
public static List<AbstractGameManagerData> GetAllGameManagers()
{
    List<AbstractGameManagerData> game_managers_data = new List<AbstractGameManagerData>();

    // Get types from reflection
    foreach (Type type in System.Reflection.Assembly.GetExecutingAssembly().GetTypes()
             .Where(mytype => mytype.IsSubclassOf(typeof(AbstractGameManagerData))))
    {
        game_managers_data.Add((AbstractGameManagerData)Activator.CreateInstance(type)); // Create instance
    }

    // Order
    return game_managers_data.OrderBy(o => o.GetUID()).ToList();
}
```

GameManagerData

An uninitialized form of GameData used before the full instance is needed e.g. in the main menu when getting a list of all game modes. The 'Instantiate' method is used when the full class is needed.

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

Abstract version:

```
/// <summary>
/// A reduced version of AbstractGameManager for use before a game has been selected
/// </summary>
public abstract class AbstractGameManagerData
{
    /// <summary>
    ///
    /// </summary>
    /// <returns>Unique UID of this GameMode</returns>
    public abstract int GetUID();

    /// <summary>
    ///
    /// </summary>
    /// <returns>Non-unique name of this GameMode</returns>
    public abstract string GetName();

    /// <summary></summary>
    /// <returns>{ Team1Size, Team2Size, Team3Size, ... }</returns>
    public abstract TeamSize[] GetTeamSizes();

    /// <summary>
    /// Returns a list of aliases for teams e.g. { "White", "Black" }
    /// </summary>
    /// <returns></returns>
    public virtual string[] TeamAliases() { return new string[0]; }

    public virtual string GetDescription() => "No description";

    public abstract AbstractGameManager Instantiate();
}
```

Example implementation:

```
public class GameManagerData : AbstractGameManagerData
{
    public override AbstractGameManager Instantiate()
    {
        return new GameManager(this);
    }

    public override int GetUID() => 100;

    public override string GetName() => "Normal Chess";

    public override string GetDescription()
    {
        return @"Normal Chess

Must have one player on both the black and white team

Traditional chess played on an 8x8 board";
    }

    public override TeamSize[] GetTeamSizes() => new TeamSize[] { new TeamSize(1, 1), new TeamSize(1, 1) };

    public override string[] TeamAliases() => new string[] { "White", "Black" };
}
```

GameManager

Controls a game mode. 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

Abstract version:

```
/// <summary>
/// Top level gamemode control
/// </summary>
public abstract class AbstractGameManager
{
    public AbstractGameManagerData GameManagerData;
    public AbstractBoard Board;

    public AbstractGameManager(AbstractGameManagerData gameManagerData)
    {
        this.GameManagerData = gameManagerData;
    }

    /// <summary>
    /// Returns the serialised game
    /// </summary>
    /// <returns></returns>
    public virtual SerialisationData GetData()
    {
        SerialisationData serialisationData = Board.GetData();
        serialisationData.GamemodeUID = GameManagerData.GetUID();
        return serialisationData;
    }

    /// <summary>
    /// Loads serialisation data into the game
    /// </summary>
    /// <param name="data"></param>
    public virtual void LoadData(SerialisationData data)
    {
        Board.LoadData(data);
    }

    /// <summary>
    /// Returns a list of possible moves
    /// </summary>
    /// <returns></returns>
    public virtual List<Move> GetMoves(LiveGameData gameData, bool fastMode)
    {
        return Board.GetMoves(gameData);
    }
```

```
/// <summary>
/// Pass turn on if no moves are available
/// </summary>
/// <returns>Next player's turn / Winning team (negative TeamID - 1)</returns>
public virtual int OnNoMoves(LiveGameData gameData)
{
    return GUtil.TurnEncodeTeam(GUtil.SwitchTeam(gameData)); ;
}

/// <summary>
/// Handles an incoming move (local or foreign)
/// </summary>
/// <param name="from"></param>
/// <param name="to"></param>
/// <returns>Next player's turn / Winning team (negative TeamID - 1)</returns>
public virtual int OnMove(Move move, LiveGameData gameData)
{
    Board.OnMove(move);
    return GUtil.SwitchPlayerTeam(gameData);
}

/// <summary>
/// Gets a score for the current board. Mostly used for AI
/// </summary>
/// <param name="gameData"></param>
/// <returns></returns>
public virtual float GetScore(LiveGameData gameData)
{
    return Board.GetScore(gameData);
}

/// <summary>
/// Returns a clone of the game
/// </summary>
/// <returns></returns>
public abstract AbstractGameManager Clone();
```

Example implementation:

```
public class GameManager : AbstractGameManager
{
    public GameManager(AbstractGameManagerData d) : base(d)
    {
        Board = new Board(this);
    }

    public override List<Move> GetMoves(LiveGameData gameData, bool fastMode)
    {
        return Board.GetMoves(gameData);
    }

    private int CheckForWin(LiveGameData gameData)
    {
        bool full = true;
        for (int x = 0; x < 7; x++)
        {
            if (Board.GetPiece(new V2(x, 8)) is null) { full = false; break; }
        }
        if (full) return GUtil.TurnEncodeTeam(GUtil.SwitchTeam(gameData));

        // Horizontal
        for (int y = 0; y < 9; y++)
        {…
        }

        // Vertical
        for (int x = 0; x < 7; x++)
        {…
        }

        // Diagonal / pt.1
        for (int y = 0; y < 9; y++)
        {…
        }

        // Diagonal / pt.2
        for (int x = 1; x < 7; x++)
        {…
        }
```

```
        public override int OnMove(Move move, LiveGameData gameData)
        {
            Board.OnMove(move);

            int winner = CheckForWin(gameData);
            if (winner >= 0) return GUtil.SwitchPlayerTeam(gameData);
            else return winner;
        }

        public override AbstractGameManager Clone()
        {
            GameManager new_game_manager = new GameManager(GameManagerData);
            new_game_manager.Board = (Board as Board).Clone(new_game_manager);

            return new_game_manager;
        }

        public override float GetScore(LiveGameData gameData) => 0f;
    }
}
```

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

Abstract version:

```
/// <summary>
/// Handles the board's configuration and stores pieces
/// </summary>
public abstract class AbstractBoard
{
    public AbstractPiece[,] PieceBoard;

    public AbstractGameManager GameManager;

    public AbstractBoard(AbstractGameManager gameManager)
    {
        this.GameManager = gameManager;
    }

    /// <summary>
    /// Returns serialised data
    /// </summary>
    /// <returns></returns>
    public virtual SerialisationData GetData()
    {
        SerialisationData serialisationData = new SerialisationData();

        // Add data for every piece on board
        for (int x = 0; x < PieceBoard.GetLength(0); x++)
        {
            for (int y = 0; y < PieceBoard.GetLength(1); y++)
            {
                if (PieceBoard[x, y] is not null)
                {
                    PieceSerialisationData data = PieceBoard[x, y].GetData();
                    if (data is not null) serialisationData.PieceData.Add(data);
                }
            }
        }

        return serialisationData;
    }
```

```
/// <summary>
/// Loads serialised data
/// </summary>
/// <param name="data"></param>
public virtual void LoadData(SerialisationData data)
{
    PieceBoard = new AbstractPiece[PieceBoard.GetLength(0), PieceBoard.GetLength(1)];

    // Get map of UIDs to pieces
    List<AbstractPiece> all_pieces = Util.GetAllPieces();
    Dictionary<int, Type> mapped_pieces = new Dictionary<int, Type>();
    foreach (AbstractPiece piece in all_pieces) mapped_pieces[piece.GetUID()] = piece.GetType();

    // Create instance of every piece in save data and place it on the board
    foreach (PieceSerialisationData piece_data in data.PieceData)
    {
        AbstractPiece new_piece = (AbstractPiece)Activator.CreateInstance(mapped_pieces[piece_data.UID], new object[] { piece_data.Position, piece_data.Team, this });
        new_piece.LoadData(piece_data); // Load piece data
        PieceBoard[piece_data.Position.X, piece_data.Position.Y] = new_piece;
    }
}

/// <summary>
/// Returns a list of possible moves
/// </summary>
/// <param name="gameData"></param>
/// <returns></returns>
public virtual List<Move> GetMoves(LiveGameData gameData)
{
    IEnumerable<Move> moves = new List<Move>();
    for (int x = 0; x < PieceBoard.GetLength(0); x++)
    {
        for (int y = 0; y < PieceBoard.GetLength(1); y++)
        {
            if (PieceBoard[x, y] is not null && PieceBoard[x, y].Team == gameData.CurrentTeam) moves = moves.Concat(PieceBoard[x, y].GetMoves());
        }
    }

    return GUtil.RemoveBlocked(moves.ToList(), this);
}
```

```
/// <summary>
/// Returns a piece at the given position
/// </summary>
/// <param name="position"></param>
/// <returns></returns>
public AbstractPiece GetPiece(V2 position)
{
    return PieceBoard[position.X, position.Y];
}

/// <summary>
/// Returns the render information for the board
/// </summary>
/// <returns></returns>
public abstract BoardRenderInfo GetBoardRenderInfo();

/// <summary>
/// Applies a move
/// </summary>
/// <param name="move"></param>
public virtual void OnMove(Move move)
{
    for (int x = 0; x < PieceBoard.GetLength(0); x++)
    {
        for (int y = 0; y < PieceBoard.GetLength(1); y++)
        {
            if (PieceBoard[x, y] is not null) PieceBoard[x, y].OnMove(move, move.From == new V2(x, y));
        }
    }
}

/// <summary>
/// Returns a score for the board
/// </summary>
/// <param name="gameData"></param>
/// <returns></returns>
public virtual float GetScore(LiveGameData gameData)
{
    float total = 0;

    for (int x = 0; x < PieceBoard.GetLength(0); x++)
    {
        for (int y = 0; y < PieceBoard.GetLength(1); y++)
```

```
                if (PieceBoard[x, y] is not null)
                {
                    if (PieceBoard[x, y].Team == gameData.LocalPlayerTeam)
                    {
                        total += PieceBoard[x, y].GetValue();
                    }
                    else
                    {
                        total -= PieceBoard[x, y].GetValue();
                    }
                }
            }
        }

        return total;
    }

    /// <summary>
    /// Returns a clone of the board
    /// </summary>
    /// <param name="newGameManager"></param>
    /// <returns></returns>
    public abstract AbstractBoard Clone(AbstractGameManager newGameManager);
}
}
```

Example implementation:

```
/// <summary>
/// Checkers Board
/// </summary>
public class Board : AbstractBoard
{
    public Board(AbstractGameManager gameManager, bool initialise = true) : base(gameManager)
    {
        if(initialise) InitialiseBoard();
    }

    protected void InitialiseBoard()
    {
        PieceBoard = new AbstractPiece[8, 8];

        for (int i = 0; i < 8; i += 2) PieceBoard[i, 0] = new CheckersPiece(new V2(i, 0), 0, this);
        for (int i = 1; i < 8; i += 2) PieceBoard[i, 1] = new CheckersPiece(new V2(i, 1), 0, this);
        for (int i = 0; i < 8; i += 2) PieceBoard[i, 2] = new CheckersPiece(new V2(i, 2), 0, this);

        for (int i = 1; i < 8; i += 2) PieceBoard[i, 5] = new CheckersPiece(new V2(i, 5), 1, this);
        for (int i = 0; i < 8; i += 2) PieceBoard[i, 6] = new CheckersPiece(new V2(i, 6), 1, this);
        for (int i = 1; i < 8; i += 2) PieceBoard[i, 7] = new CheckersPiece(new V2(i, 7), 1, this);
    }

    public override BoardRenderInfo GetBoardRenderInfo()
    {
        return new BoardRenderInfo(8, new List<V2>(), null, true);
    }

    public override AbstractBoard Clone(AbstractGameManager newGameManager)
    {
        Board board = new Board(newGameManager, false);
        AbstractPiece[,] pieceBoard = new AbstractPiece[8, 8];
        for (int x = 0; x < PieceBoard.GetLength(0); x++)
        {
            for (int y = 0; y < PieceBoard.GetLength(0); y++)
            {
                if (PieceBoard[x, y] is not null) pieceBoard[x, y] = PieceBoard[x, y].Clone(board);
            }
        }

        board.PieceBoard = pieceBoard;

        return board;
    }
}
```

Piece

Data	**Description**	**Type**
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

Abstract version:

```
/// <summary>
/// A game piece
/// </summary>
public abstract class AbstractPiece
{
    public V2 Position;
    /// <summary>
    /// ID corresponding to sprites in the VisualManager
    /// </summary>
    public int AppearanceID;

    public int Team;

    public AbstractBoard Board;

    public AbstractPiece(V2 position, int team, AbstractBoard board)
    {
        Position = position;
        Team = team;
        Board = board;
    }

    /// <summary>
    /// Returns a list of possible moves for this piece
    /// </summary>
    /// <returns></returns>
    public virtual List<Move> GetMoves() { return new List<Move>(); }

    /// <summary>
    /// Returns the piece's UID
    /// </summary>
    /// <returns></returns>
    public abstract int GetUID();

    /// <summary>
    /// Returns the piece's serialised data
    /// </summary>
    /// <returns></returns>
    public virtual PieceSerialisationData GetData()
    {
        PieceSerialisationData data = new PieceSerialisationData();
        data.Team = Team;
        data.Position = Position;
        data.UID = GetUID();
```

```
        return data;
    }

    /// <summary>
    /// Loads serialised data into the piece
    /// </summary>
    /// <param name="data"></param>
    public virtual void LoadData(PieceSerialisationData data)
    {
        Position = data.Position;
        Team = data.Team;
    }

    /// <summary>
    /// Applies a move to the piece
    /// </summary>
    /// <param name="move"></param>
    /// <param name="thisPiece"></param>
    public virtual void OnMove(Move move, bool thisPiece) { }

    /// <summary>
    /// Returns the piece's value to its team
    /// </summary>
    /// <returns></returns>
    public virtual float GetValue() => 1f;

    /// <summary>
    /// Returns a clone of the piece
    /// </summary>
    /// <param name="newBoard"></param>
    /// <returns></returns>
    public abstract AbstractPiece Clone(AbstractBoard newBoard);
}
```

Example implementation:

```
public class KingPiece : NormalChessPiece
{
    public bool HasMoved;

    public KingPiece(V2 position, int team, AbstractBoard board) : base(position, team, boar
    {
        AppearanceID = 101;
        if (team != 0) AppearanceID += 6;
    }

    public override List<Move> GetMoves()
    {
        List<Move> moves = new List<Move>()
        {
            new Move(Position, Position + new V2(1, 1)),
            new Move(Position, Position + new V2(1, -1)),
            new Move(Position, Position + new V2(-1, 1)),
            new Move(Position, Position + new V2(-1, -1)),
            new Move(Position, Position + new V2(1, 0)),
            new Move(Position, Position + new V2(0, 1)),
            new Move(Position, Position + new V2(-1, 0)),
            new Move(Position, Position + new V2(0, -1)),
        };

        // Castle logic
        if (!HasMoved)
        {
            if (Board.GetPiece(Position + new V2(-4, 0)) is not null
                && typeof(RookPiece) == Board.GetPiece(Position + new V2(-4, 0)).GetType()
                && !((RookPiece)Board.GetPiece(Position + new V2(-4, 0))).HasMoved)
            {
                bool clear = true;
                for (int x = 1; x <= 3; x++)
                {
                    if (Board.GetPiece(new V2(x, Position.Y)) is not null)
                    {
                        clear = false; break;
                    }
                }
                if (clear) moves.Add(new Move(Position, Position + new V2(-2, 0)));
            }
            if (Board.GetPiece(Position + new V2(3, 0)) is not null
                && typeof(RookPiece) == Board.GetPiece(Position + new V2(3, 0)).GetType()
                && !((RookPiece)Board.GetPiece(Position + new V2(3, 0))).HasMoved)
```

```
            {
                bool clear = true;
                for (int x = 5; x <= 6; x++)
                {
                    if (Board.GetPiece(new V2(x, Position.Y)) is not null)
                    {
                        clear = false; break;
                    }
                }
                if (clear) moves.Add(new Move(Position, Position + new V2(2, 0)));
            }
        }

        return GUtil.RemoveFriendlies(GUtil.RemoveBlocked(moves, Board), Board);
    }

    public override void OnMove(Move move, bool thisPiece)
    {
        if (!thisPiece) return;

        HasMoved = true;

        // Castle
        if (move.To.X - move.From.X == 2)
        {
            Board.PieceBoard[move.To.X - 1, move.To.Y] = Board.PieceBoard[move.To.X + 1, move.To.Y];
            Board.PieceBoard[move.To.X - 1, move.To.Y].Position = new V2(move.To.X - 1, move.To.Y);
            Board.PieceBoard[move.To.X - 1, move.To.Y].OnMove(move, false);
            (Board.PieceBoard[move.To.X - 1, move.To.Y] as RookPiece).HasMoved = true;
            Board.PieceBoard[move.To.X + 1, move.To.Y] = null;
        }
        else if (move.To.X - move.From.X == -2)
        {
            Board.PieceBoard[move.To.X + 1, move.To.Y] = Board.PieceBoard[move.To.X - 2, move.To.Y];
            Board.PieceBoard[move.To.X + 1, move.To.Y].Position = new V2(move.To.X + 1, move.To.Y);
            Board.PieceBoard[move.To.X + 1, move.To.Y].OnMove(move, false);
            (Board.PieceBoard[move.To.X + 1, move.To.Y] as RookPiece).HasMoved = true;
            Board.PieceBoard[move.To.X - 2, move.To.Y] = null;
        }
    }

    public override PieceSerialisationData GetData()
    {
        PieceSerialisationData data = new PieceSerialisationData();
        data.Team = Team;
```

```
        data.Position = Position;
        data.UID = GetUID();
        data.Data = BitConverter.GetBytes(HasMoved);
        return data;
    }

    public override void LoadData(PieceSerialisationData data)
    {
        Position = data.Position;
        Team = data.Team;
        HasMoved = BitConverter.ToBoolean(data.Data);
    }

    public override AbstractPiece Clone(AbstractBoard newBoard)
    {
        KingPiece k = new KingPiece(Position, Team, newBoard);
        k.HasMoved = HasMoved;
        return k;
    }

    public override int GetUID() => 101;

    public override float GetValue() => 1000f;
}
```

Networking System

Handles all online communications. The NetworkManager is responsible for starting and stopping the Client and Server and is also responsible for passing communications between the Client and/or Server and the ChessManager.

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 is running	Void
Client Callbacks	Methods that directly map to methods of other classes are used to maintain an understandable reference tree.	All Void

```
/// <summary>
/// Provides an interface for MonoBehaviour classes with the Client and Server
/// </summary>
public class NetworkManager : MonoBehaviour
{
    private ChessManager chessManager = default!;

    private Server? server = null;
    private Client? client = null;

    public bool IsHost { get { return server is not null; } }

    // Runs once
    private void Awake()
    {
        DontDestroyOnLoad(this);
        chessManager = FindObjectOfType<ChessManager>();
    }

    // Shuts down client and server correctly
    private void OnApplicationQuit() => Stop();

    #region Client encapsulations
    public void OnLocalMove(int nextPlayer, V2 from, V2 to) => client?.OnLocalMove(nextPlayer, from, to);

    /// <summary>
    /// Gets the player list from the client
    /// </summary>
    /// <returns></returns>
    /// <exception cref="NullReferenceException"></exception>
    public ConcurrentDictionary<int, ClientPlayerData> GetPlayerList() => client!.PlayerData;

    public int GetLocalPlayerID() => client!.PlayerID;
    #endregion

    /// <summary>
    /// Hosts a game
    /// </summary>
    public bool Host(HostSettings settings, Action onPlayersChange, Action onGameStart)
    {
        if (NetworkingUtils.PortInUse(NetworkSettings.PORT)) return false; // Check if port is in use

        ServerGameData gameData = new ServerGameData(settings.GameMode, settings.SaveData);
```

```
        // Start server
        server = new Server(gameData, settings.Password);
        server.Start();

        // Start local client
        client = new Client("127.0.0.1", settings.Password, settings.PlayerName, this, OnHostSuccessOrFail);
        client.Connect();
        return true;
    }

    /// <summary>
    /// Starts a game
    /// </summary>
    /// <returns>Null if successful or a string error</returns>
    public string? HostStartGame() => server?.StartGame();

    /// <summary>
    /// Joins a game
    /// </summary>
    /// <param name="settings"></param>
    public void Join(JoinSettings settings)
    {
        client = new Client(settings.IP, settings.Password, settings.PlayerName, this, OnJoinSuccessOrFail);
        client.Connect();
    }

    /// <summary>
    /// Called on connection failure to correctly shut down server and client
    /// </summary>
    private void ConnectionFailed() => Stop();

    /// <summary>
    /// Calls the callback with the ping as the parameter
    /// </summary>
    /// <param name="callback"></param>
    public void GetPing(Action<int> callback) => client?.GetPing(callback);

    /// <summary>
    /// Shuts down networking
    /// </summary>
    public void Stop()
    {
        Debug.LogWarning("Network Stop");
        server?.Shutdown();
```

```
        server = null;
        client?.Shutdown();
        client = null;
        Debug.LogWarning("Network Stop finished");
    }

    // Encapsulated methods the client calls
    #region Client Callbacks
    public void OnHostSuccessOrFail(string? status)
    {
        if (status is not null)
        {
            chessManager.HostFailed(status);
            ConnectionFailed();
            return;
        }

        chessManager.HostSucceed();
        OnPlayersChange();
    }
    public void OnJoinSuccessOrFail(string? status)
    {
        if (status is not null)
        {
            chessManager.JoinFailed(status);
            ConnectionFailed();
            return;
        }

        chessManager.JoinSucceed();
        OnPlayersChange();
    }
    public void OnClientKick(string reason) => chessManager.JoinFailed(reason);
    public void OnPlayersChange() => chessManager.PlayerListUpdate();
    public void OnGamemodeRecieve(int gameMode, byte[] saveData) => chessManager.GameDataRecived(gameMode, saveData);
    public void OnGameStart() => chessManager.OnGameStart();
    public void OnForeignMove(int nextPlayer, V2 from, V2 to) => chessManager.OnForeignMoveUpdate(nextPlayer, from, to);
    public void HostSetTeam(int playerID, int team, int playerInTeam) => server?.SetTeam(playerID, team, playerInTeam);
    public int FindNextNonFullTeam(int currentTeam, TeamSize[] teamSizes) => server!.FindNextNonFullTeam(currentTeam, teamSizes);
    #endregion
```

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 the (Pregame Networking Flow) diagram for a representation of what happens in the lobby and the (In-Game Networking Flow) diagram for a representation of what happens during a game.

The best way to understand the Client and Server's functionality is by understanding the function of the threads they use:

- Client Accept Thread
    - This is Server-exclusive and waits for a client to try connecting, then handles whether they can or can't connect for reasons such as password checking, networking version, and whether a game is currently in-progress.
    - If a client connects successfully, it starts to listen asynchronously for new messages from it
- Client Join Thread
    - This is Client-exclusive and runs once when the Client starts to try to connect to the server
    - If connection is accepted, it starts to listen asynchronously for messages
- Send Thread
    - Both the Client and Server have this thread and it sends messages from the send queue, if there are any available
    - The only difference between Client and Server version is that the server has to specify a recipient.
- Receive Thread
    - Both the Client and Server have this thread
    - When a new message is received asynchronously it is put in the 'Content Queue'. The Receive thread dequeues one message at a time and handles it. This is important as the packets need to be handled one at a time to not interfere with game logic that expects only one change to happen at a time.

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 the packet type	Content block 1 length bytes
[Repeat last two for all content]			

```
namespace Networking.Packets.Generated
{
    public class SampleTestPacket {
        public const int UID = 100;
        public int ArgOne;
        public double ArgTwo;
        public string ArgThree;
        public string ArgFour;
        public SampleTestPacket(Packet packet){
            ArgOne = BitConverter.ToInt32(packet.Contents[0]);
            ArgTwo = BitConverter.ToDouble(packet.Contents[1]);
            ArgThree = ASCIIEncoding.ASCII.GetString(packet.Contents[2]);
            ArgFour = ASCIIEncoding.ASCII.GetString(packet.Contents[3]);
        }

       public static byte[] Build(int _ArgOne, double _ArgTwo, string _ArgThree, string _ArgFour
           List<byte[]> contents = new List<byte[]>();
           contents.Add(BitConverter.GetBytes(_ArgOne));
           contents.Add(BitConverter.GetBytes(_ArgTwo));
           contents.Add(ASCIIEncoding.ASCII.GetBytes(_ArgThree));
           contents.Add(ASCIIEncoding.ASCII.GetBytes(_ArgFour));
           return PacketBuilder.Build(UID, contents);
    }
    }
}
```

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 allowing the developer to never have to interact with raw bytes for sending and receiving data as usage of the system looks like this:

```
handler.Send(ClientConnectRequestPacket.Build(PlayerName, NetworkSettings.VERSION, Password));
```

^ Encoding

```
ServerConnectAcceptPacket acceptPacket = new ServerConnectAcceptPacket(packet);
client.PlayerID = acceptPacket.PlayerID;
```

^ Decoding

```
Initial client connection request
0\ClientConnectRequest\Name\string\Version\string\Password=""\string

Server accepting client's connection
1\ServerConnectAccept\PlayerID\int

Server kicking client
2\ServerKick\Reason\string
```

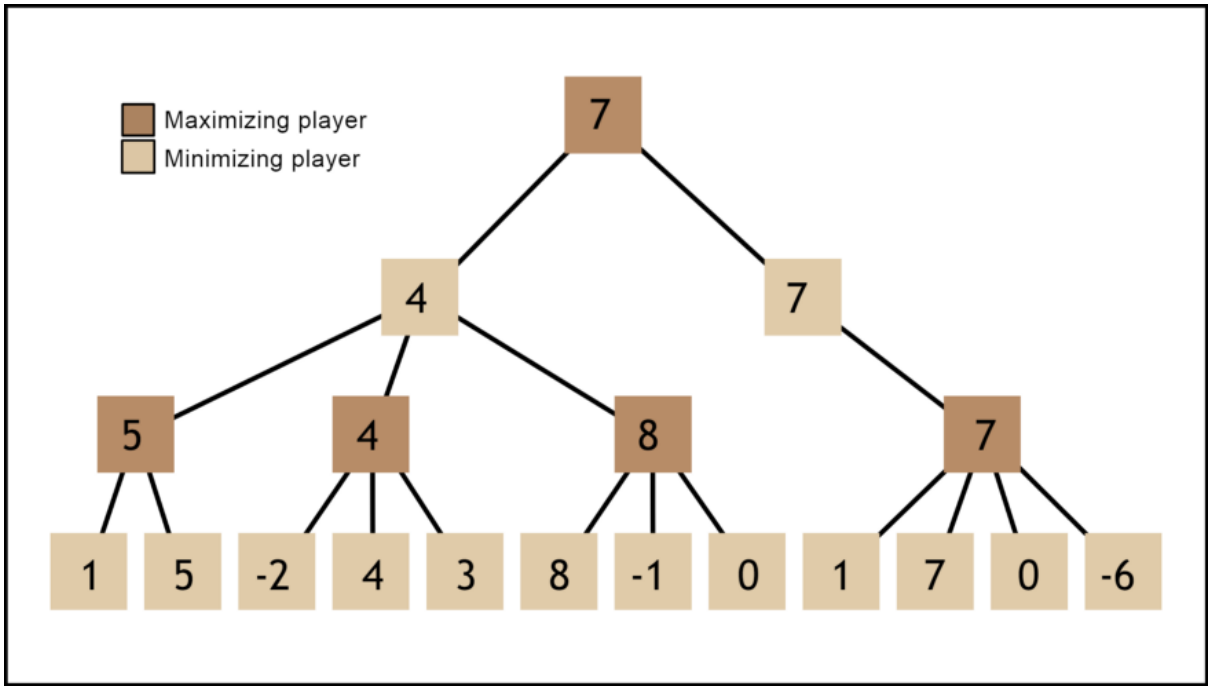
^ Packet generating file (Any line without a '\' is a comment)

For more details see: Packet System and Generation

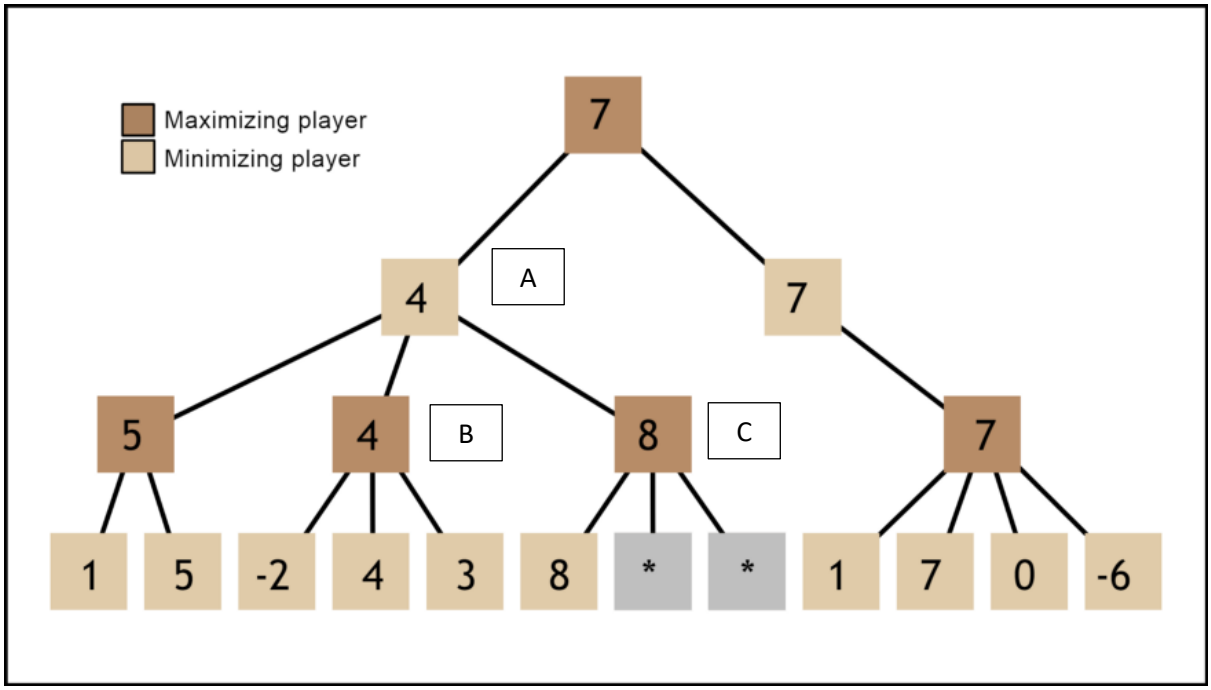
AI System (MiniMax Algorithm)

The AI will use the minimax algorithm to calculate the best move and use Alpha-Beta pruning for optimisation. The AI is given a time limit and tries 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 the next image): https://medium.com/@SereneBiologist/the-anatomy-of-a-chess-ai-2087d0d565



^ 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

Implementation:

```
/// <summary>
/// Implementation of the MiniMax algorithm with AB pruning
/// </summary>
/// <param name="gameManager"></param>
/// <param name="gameData"></param>
/// <param name="moves"></param>
/// <param name="current_depth"></param>
/// <param name="max_depth"></param>
/// <param name="prev_best"></param>
/// <param name="prev_maximising"></param>
/// <returns></returns>
2 references
private static float MiniMax(AbstractGameManager gameManager, LiveGameData gameData, List<Move> moves, int current_depth, int max_depth, float prev_best, bool prev_maximising)
{
    // Return if at max depth
    if (current_depth == max_depth) return gameManager.GetScore(gameData);

    float best_score;
    bool maximising;

    // Set maximising or minimising
    if (gameData.CurrentTeam == gameData.LocalPlayerTeam)
    {
        best_score = float.MinValue;
        maximising = true;
    }
    else
    {
        best_score = float.MaxValue;
        maximising = false;
    }

    foreach (Move move in moves)
    {
        // Exit if over time
        if ((current_depth == max_depth - 1 || current_depth == max_depth) && IsOverTime()) return float.NaN;

        // Apply move
        AbstractGameManager new_manager = gameManager.Clone();
        int next_player = new_manager.OnMove(move, gameData);

        float move_score;

        if (next_player < 0) // A team has won
        {
            if (GUtil.TurnDecodeTeam(next_player) != gameData.LocalPlayerTeam)
            {
                if (maximising) move_score = float.MinValue + current_depth;
                else return float.MinValue + current_depth; // Best possible value for minimiser so instantly return
            }
            else
            {
                if (maximising) return float.MaxValue - current_depth; // Best possible value for maximiser so instantly return
                else move_score = float.MaxValue - current_depth;
            }
        }
        else
        {
            // Set move to next player
            LiveGameData new_game_data = gameData.Clone();
            new_game_data.CurrentPlayer = next_player;

            List<Move> new_moves = new_manager.GetMoves(new_game_data, fastMode: true);

            // Recursion
            move_score = MiniMax(new_manager, new_game_data, new_moves, current_depth + 1, max_depth, best_score, maximising);
        }


        // Return if algorithm is terminating
        if (move_score is float.NaN) return float.NaN;

        if (maximising && move_score > best_score) best_score = move_score;
        else if (!maximising && move_score < best_score) best_score = move_score;


        // AB Pruning
        if ((best_score > prev_best && !prev_maximising && maximising) || (best_score <  prev_best && prev_maximising && !maximising))
        {
            return best_score;
        }

    }

    return best_score;
}
```

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.

```
with open(PATH_TO_DATA + "/" + file, "r") as f:
    new_data = template_data.replace("$$$$", f.read())

with open(PATH_TO_HELP + "/" + file, "w+") as f:
    f.write(new_data)
```

^ Code creating a new file based on the template.

```
<div class="container">
    <div class="container main-content" style="padding: 50px">
        $$$$
    </div>
</div>
```

^ Area where the file is inserted into the template

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

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

```
<!DOCTYPE html>      You, 3 months ago • Added help system …
<html lang="en">

<head>
    <meta charset="utf-8" />
    <link rel="stylesheet" href="static/bootstrap/css/bootstrap.css" />
    <link rel="stylesheet" href="static/base.css" />

    <script src="static/bootstrap/js/bootstrap.js"></script>

    <title>Help</title>
</head>
```

^ Bootstrap inclusion

```
<div class="container">
    <img src="static/img/400_take_example.png"></img>
    <p class="text-muted" style="margin-bottom: 0">Before take above. After take below.</p>
    <img src="static/img/400_take_example_2.png"></img>
</div>
```

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

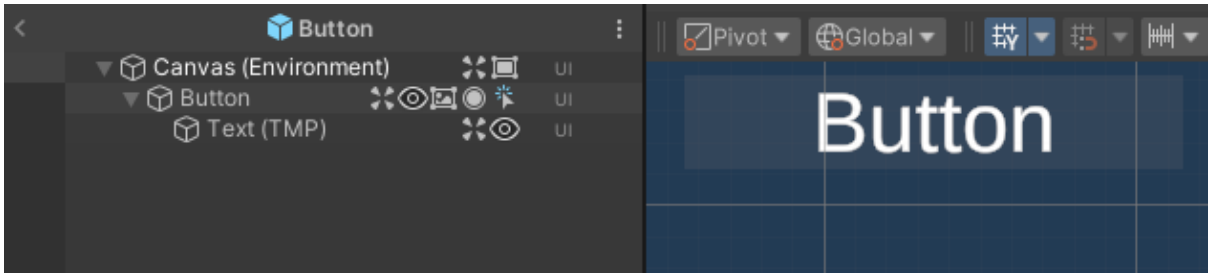
^ Help page for Atomic Chess

Implementation for opening help:

```
8 references
public static class HelpSystem
{
    public const string IN_GAME_PAGE_NAME = "in_game";

    /// <summary>
    /// Opens help homepage
    /// </summary>
    1 reference
    public static void OpenHelp()
    {
#if PLATFORM_STANDALONE_WIN || UNITY_STANDALONE_WIN || UNITY_EDITOR_WIN
        Process.Start(Application.dataPath + "/StreamingAssets/Help/index.html");
#endif
    }

    /// <summary>
    /// Opens help page for a specific gamemode
    /// </summary>
    /// <param name="gamemodeID"></param>
    3 references
    public static void OpenHelp(int gamemodeID)
    {
#if PLATFORM_STANDALONE_WIN || UNITY_STANDALONE_WIN || UNITY_EDITOR_WIN
        Process.Start(Application.dataPath + "/StreamingAssets/Help/" + gamemodeID + ".html");
#endif
    }

    /// <summary>
    /// Opens specified help page
    /// </summary>
    /// <param name="pageName"></param>
    3 references
    public static void OpenHelp(string pageName)
    {
#if PLATFORM_STANDALONE_WIN || UNITY_STANDALONE_WIN || UNITY_EDITOR_WIN
        Process.Start(Application.dataPath + "/StreamingAssets/Help/" + pageName + ".html");
#endif
    }
}
```

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: Using C# Reflection to Avoid Creating Lists of Classes

In many parts of my program, I need a list of every class that inherits from a base class or I need a dictionary of UIDs mapped to these classes. Without reflection, this would require maintaining two or three lists of classes that I would need to update every time I added a new feature such as a game mode so that they showed up in game mode selection menus and could be used. Reflection allows me to get information about the structure of the program such as getting a list of all classes and filter them for ones that inherit from a specific class.

```
/// <summary>
/// Retrieves a list of all subclasses of AbstractGameManagerData
/// </summary>
/// <returns></returns>
3 references
public static List<AbstractGameManagerData> GetAllGameManagers()
{
    List<AbstractGameManagerData> game_managers_data = new List<AbstractGameManagerData>();

    // Get types from reflection
    foreach (Type type in System.Reflection.Assembly.GetExecutingAssembly().GetTypes()
             .Where(mytype => mytype.IsSubclassOf(typeof(AbstractGameManagerData))))
    {
        game_managers_data.Add((AbstractGameManagerData)Activator.CreateInstance(type)); // Create instance
    }

    // Order
    return game_managers_data.OrderBy(o => o.GetUID()).ToList();
}
```

In the example above I iterate through each type in the assembly:

```
foreach (Type type in System.Reflection.Assembly.GetExecutingAssembly().GetTypes()
```

Filtering out all those which aren't a subclass of the AbstractGameManager:

```
.Where(mytype => mytype.IsSubclassOf(typeof(AbstractGameManagerData))))
```

Then I create an instance of the type and add it to the list:

```
game_managers_data.Add((AbstractGameManagerData)Activator.CreateInstance(type)); // Create instance
```

Finally re-ordering it:

```
// Order
return game_managers_data.OrderBy(o ⇒ o.GetUID()).ToList();
```

This allows me to add game modes, pieces and more without adding them to a constant list in another part of the program.

Another use for this is with testing. Instead of writing individual tests for each class, I used reflection and a generic test to test any new class I added without any extra code. In the example below, I use a method I wrote that retrieves all AbstractGameMangerData classes and test to make sure that their UIDs are unique and that their names can be used in save files.

```
/// <summary>
/// Runs various tests on game managers
/// </summary>
/// <returns></returns>
1 reference
private static bool TestGameManagers()
{
    output_string += "Running Game Manager Tests\n";

    // Get all AbstractGameManagerData
    List<AbstractGameManagerData> abstract_game_managers_data = Util.GetAllGameManagers();

    // Ensure no duplicate UIDs
    HashSet<int> used_uids = new HashSet<int>();
    foreach (AbstractGameManagerData data in abstract_game_managers_data)
    {
        if (used_uids.Contains(data.GetUID())) { Debug.LogError($"Gamemode UID ({data.GetUID()}) used twice"); return false; }
        used_uids.Add(data.GetUID());

        // Ensure all gamemode names can be used in file names
        if (data.GetName().IndexOfAny(Path.GetInvalidFileNameChars()) ≥ 0)
        {
            Debug.LogError($"{data.GetName()} contains invalid characters for a file name");
            return false;
        }
    }

    return true;
}
```

Accounting for all Display Sizes and Ratios, and has some text elements to copy the font size of other text elements to ensure consistency: Making text sizes consistent.

VisualManager System

When in a game, the VisualManager (and VisualManager3D if the game mode supports 3D) render the board and pieces and handle pieces moving, being taken, squares being highlighted and more. As all game modes must support 2D and only some support 3D, the VisualManager handles all the calculations and algorithms for deciding how to move pieces (see: Working Out which Pieces have Moved) and mostly calls 3D methods to apply these moves.

The VisualManager receives board information (size, highlighted squares, blocked squares etc) from the active game modes board (which derives from AbstractBoard, see: Board). It then receives incoming moves from the ChessManager and relays outgoing moves to the ChessManager as well.

For more information on how the 3D board appearance and animations work see: (Making 3D Squares Move Smoothly), (Implementing the 3D Ripple Effect) and (Making Pieces Jump)

The 3D camera is controlled by the CameraManager script which uses three pivots: a vertical one to make the camera orbit, a horizontal one to change camera elevation and one that slides forward and back. As each pivot is a child of the one before and inherits its rotation and the camera is a child of the last pivot, the camera can orbit, change elevation and zoom.

Camera implementation:

```
/// <summary>
/// Rotates the camera given a Vector 2 mouse movement input
/// </summary>
/// <param name="look"></param>
1 reference
public void RotateCamera(Vector2 look)
{
    camera_rotation.x -= look.y;
    camera_rotation.y += look.x;

    camera_rotation.x = Mathf.Clamp(camera_rotation.x, 0, 90);

    rotationalPivot.transform.localRotation = Quaternion.Euler(0, camera_rotation.y, 0);
    upDownPivot.transform.localRotation = Quaternion.Euler(camera_rotation.x, 0, 0);
}
```

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 characters to allow strong passwords. There will be some character restrictions 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 game modes 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 game mode will provide the team sizes.

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

Validation implementation examples:

```
/// <summary>
/// Validates a player's name
/// </summary>
/// <param name="name"></param>
/// <returns>Null if successful or a string error</returns>
4 references
public static string? ValidatePlayerName(string name)
{
    const string allowed_chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567890_";

    // if (Util.RemoveInvisibleChars(name) != name) return "Contains invisible characters"; // Invisible characters

    if (name.Length > 16) return "Name must be shorter than 17 characters"; // Too long

    if (name.Length < 4) return "Name must be longer than 3 characters"; // Too short

    foreach (char c in name)
    {
        if (!allowed_chars.Contains(c))
        {
            if (c == ' ') return $"Character '{c}' (space) not allowed";
            return $"Character '{c}' not allowed";
        }
    }

    return null;
}
```

```
/// <summary>
/// Validates a file name
/// </summary>
/// <param name="fileName"></param>
/// <returns></returns>
1 reference
public static string? ValidateFileName(string fileName)
{
    if (fileName.IndexOfAny(Path.GetInvalidFileNameChars()) >= 0)
    {
        return $"File name contains illegal character '{fileName[fileName.IndexOfAny(Path.GetInvalidFileNameChars())]}'";
    }

    if (fileName.Length > 16) return "File name must be shorter than 17 character";
    if (fileName.Length < 4) return "File name must be longer than 3 character";

    return null;
}
```

```
/// <summary>
/// Ensures that the team compositions are correct for the game to start
/// </summary>
/// <returns>Null if successful or a string error</returns>
1 reference
public static string? ValidateTeams(List<ServerPlayerData> playerData, ServerGameData gameData)
{

    int max_team = 0;
    Dictionary<int, int> team_dict = new Dictionary<int, int>();
    Dictionary<int, List<int>> players_in_teams = new Dictionary<int, List<int>>();
    foreach (ServerPlayerData player in playerData)
    {
        if (player.Team > max_team) max_team = player.Team;

        if (!team_dict.ContainsKey(player.Team))
        {
            players_in_teams.Add(player.Team, new List<int>() { player.PlayerInTeam });
            team_dict.Add(player.Team, 1);
        }
        else
        {
            team_dict[player.Team]++;
            if (players_in_teams[player.Team].Contains(player.PlayerInTeam) && player.Team != -1) return "Duplicate player in team"; // Duplicate player in team (not applicable to spectators)
            players_in_teams[player.Team].Add(player.PlayerInTeam);
        }
    }

    if (max_team != gameData.TeamSizes.Length - 1) return "Wrong number of teams"; // Wrong number of teams

    for (int i = 0; i < max_team; i++)
    {
        if (!team_dict.ContainsKey(i) && gameData.TeamSizes[i].Min > 0) return "Team missing"; // Team missing

        if (team_dict[i] < gameData.TeamSizes[i].Min) return "Team too small"; // Team too small
        if (team_dict[i] > gameData.TeamSizes[i].Max) return "Team too big"; // Team too big
    }

    return null;
}
```

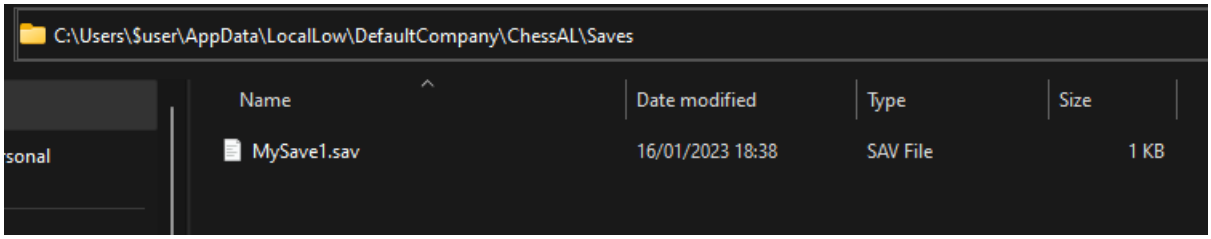
Save system

Saves are stored in a proprietary binary format for better networking performance as JSON and other text-based serialisation file formats are very inefficient for both size and speed. I also chose a proprietary format as the save file does not need to be read by a human and as it makes it easier for game mode components to save custom data. For details on the file structure see: Flexible Save Games that Support Every Game Mode

These are stored in the application's 'persistent data path'

```
4 references
public static string GetSaveFolder() => Application.persistentDataPath + "\\Saves";
```

```
string save_location = GetSaveFolder();
try
{
    if (!Directory.Exists(save_location)) Directory.CreateDirectory(save_location);

    File.WriteAllBytes(save_location + "\\" + fileName + ".sav", data);
}
catch (IOException) { return "IO error when trying to write save data"; }
catch (UnauthorizedAccessException) { return "Write permissions denied";  }
catch (SecurityException) { return "Write permissions denied"; }
catch (Exception e) { return $"Unhandled exception:\n{e}"; }
return null;
```



Serialisation:

```
/// <summary>
/// Converts SerialisationData to a raw byte array
/// </summary>
/// <param name="data"></param>
/// <returns></returns>
2 references
public static byte[] Construct(SerialisationData data) // SEE DOCUMENTATION FOR SAVE FILE FORMATTING
{
    int length = 20; // GamemodeUID, TeamTurn, PlayerOnTeamTurn, EllapsedTime (long - 4 bytes)
    length += 4; // GameManagerData length
    length += data.GameManagerData.Length;
    length += 4; // BoardData Length
    length += data.BoardData.Length;

    foreach (PieceSerialisationData piece_data in data.PieceData)
    {
        length += 16; // Team, Position, UID
        length += 4; // PieceData length
        length += piece_data.Data.Length;
    }

    byte[] output = new byte[length];

    int cursor = 0;

    ArrayExtensions.Merge(output, BitConverter.GetBytes(data.GamemodeUID), cursor);
    cursor += 4;
    ArrayExtensions.Merge(output, BitConverter.GetBytes(data.TeamTurn), cursor);
    cursor += 4;
    ArrayExtensions.Merge(output, BitConverter.GetBytes(data.PlayerOnTeamTurn), cursor);
    cursor += 4;

    ArrayExtensions.Merge(output, BitConverter.GetBytes(data.EllapsedTime), cursor);
    cursor += 8;

    ArrayExtensions.Merge(output, BitConverter.GetBytes(data.GameManagerData.Length), cursor);
    cursor += 4;
    ArrayExtensions.Merge(output, data.GameManagerData, cursor);
    cursor += data.GameManagerData.Length;

    ArrayExtensions.Merge(output, BitConverter.GetBytes(data.BoardData.Length), cursor);
    cursor += 4;
    ArrayExtensions.Merge(output, data.BoardData, cursor);
    cursor += data.BoardData.Length;

    foreach (PieceSerialisationData piece_data in data.PieceData)
    {
        ArrayExtensions.Merge(output, BitConverter.GetBytes(piece_data.Team), cursor);
        cursor += 4;
        ArrayExtensions.Merge(output, BitConverter.GetBytes(piece_data.Position.X), cursor);
        cursor += 4;
        ArrayExtensions.Merge(output, BitConverter.GetBytes(piece_data.Position.Y), cursor);
        cursor += 4;
```

```
        ArrayExtensions.Merge(output, BitConverter.GetBytes(piece_data.UID), cursor);
        cursor += 4;

        ArrayExtensions.Merge(output, BitConverter.GetBytes(piece_data.Data.Length), cursor);
        cursor += 4;
        ArrayExtensions.Merge(output, piece_data.Data, cursor);
        cursor += piece_data.Data.Length;
    }

    return output;
}
```

Deserialization:

```
/// <summary>
/// Converts raw save data to SerialisationData
/// </summary>
/// <param name="saveData"></param>
/// <returns></returns>
1 reference
public static SerialisationData Deconstruct(byte[] saveData) // SEE DOCUMENTATION FOR SAVE FILE FORMATTING
{
    SerialisationData data = new SerialisationData();

    int cursor = 0;

    data.GamemodeUID = BitConverter.ToInt32(ArrayExtensions.Slice(saveData, cursor, cursor + 4));
    cursor += 4;
    data.TeamTurn = BitConverter.ToInt32(ArrayExtensions.Slice(saveData, cursor, cursor + 4));
    cursor += 4;
    data.PlayerOnTeamTurn = BitConverter.ToInt32(ArrayExtensions.Slice(saveData, cursor, cursor + 4));
    cursor += 4;

    data.EllapsedTime = BitConverter.ToInt64(ArrayExtensions.Slice(saveData, cursor, cursor + 8));
    cursor += 8;

    int game_manager_data_length = BitConverter.ToInt32(ArrayExtensions.Slice(saveData, cursor, cursor + 4));
    cursor += 4;

    data.GameManagerData = ArrayExtensions.Slice(saveData, cursor, cursor + game_manager_data_length);
    cursor += game_manager_data_length;

    int board_data_length = BitConverter.ToInt32(ArrayExtensions.Slice(saveData, cursor, cursor + 4));
    cursor += 4;

    data.BoardData = ArrayExtensions.Slice(saveData, cursor, cursor + board_data_length);
    cursor += board_data_length;

    while (cursor < saveData.Length)
    {
        PieceSerialisationData piece_data = new PieceSerialisationData();

        piece_data.Team= BitConverter.ToInt32(ArrayExtensions.Slice(saveData, cursor, cursor + 4));
        cursor += 4;
        piece_data.Position.X= BitConverter.ToInt32(ArrayExtensions.Slice(saveData, cursor, cursor + 4));
        cursor += 4;
        piece_data.Position.Y = BitConverter.ToInt32(ArrayExtensions.Slice(saveData, cursor, cursor + 4));
        cursor += 4;
        piece_data.UID = BitConverter.ToInt32(ArrayExtensions.Slice(saveData, cursor, cursor + 4));
        cursor += 4;

        int piece_data_length = BitConverter.ToInt32(ArrayExtensions.Slice(saveData, cursor, cursor + 4));
        cursor += 4;
        piece_data.Data = ArrayExtensions.Slice(saveData, cursor, cursor + piece_data_length);
        cursor += piece_data_length;

        data.PieceData.Add(piece_data);
    }

    return data;
}
```

Use of Unity's PlayerPrefs system

To save something simple, such as the current theme, I used Unity's PlayerPrefs system which is essentially an internal database. For example, here I save the current theme:

```
1 reference
public void CycleTheme()
{
    currentTheme++;
    if (currentTheme >= themes.Length) currentTheme = 0;
    UpdateTheme();
    PlayerPrefs.SetInt(PLAYER_PREFS_THEME_KEY, currentTheme); // Store theme
}
```

And here I load it on game start:

```
// Load saved theme
if (PlayerPrefs.HasKey(PLAYER_PREFS_THEME_KEY))
{
    currentTheme = PlayerPrefs.GetInt(PLAYER_PREFS_THEME_KEY);
    if (currentTheme >= themes.Length) currentTheme = 0;
}
```

V2

Originally, V2 was meant to replace Unity's built-in Vector2Int datatype which can store two integers (x and y) with a datatype that stored x and y as chars (1-byte unsigned integers) to reduce packet size for networking as it would be a quarter the size of integers however, I found the conversions frustrating to implement when working with V2s' chars, the packet length decrease to not be that important and using other numbers that don't indicate a real square such as negative numbers to be useful. In the end, I kept V2s (with ints instead of chars) as they had a shorter name and I had more control over implementations such as the ToString() method:

```
0 references
public override string ToString()
{
    return $"({X}, {Y})";
}
```

The hash method allows it to be used in dictionaries:

```
0 references
public override int GetHashCode()
{
    int hCode = X ^ Y; // Make hash code combination of X and Y
    return hCode.GetHashCode();
}
```

And the constructor allows me to initialise V2s with a single value which isn't possible with Vector2Ints:

```
– references
public V2(int xy)
{
    X = xy;
    Y = xy;
}
```

Evaluation

Success compared to initial goals

Criteria	Category	Measured by	Success
All essential solution features included	Functionality Usability	Tick list Testing	Completed
Error-free	Robustness	Stress testing (possibly automated)	In the playtests of the final version no users encountered a bug
User-friendliness	Usability	Tests with the target audience	Completed – in the final playtest users used on-screen prompts and the help system to learn how to play the game
AI performance reasonable	Functionality	AI makes moves (on average) in under 1 minute on mid-range hardware	On medium hardware the chess AI (the most complex one) can look up to 5 moves ahead, in other game modes, it can look further (up to 9 in connect 4)
Game modes easy to develop and add	Functionality	Not having to modify other components of the project to add a game mode	The game mode system allows for the implementation of various game modes without the need for modifying other parts of the program. For example, to add a new game mode, Othello, you would need to create a GameManager, BoardManager and scripts to control individual pieces. The game mode system would render this game automatically with no need for code additions and the AI would be able to play it with no extra code either.
First functional build released by end of November	Functionality	-	The first functional build (although not containing all features and with various bugs) was released at the end of October

Changes to Initial Plan

3D

While 3D was initially not meant to be included due to the belief that it's difficult to create a version of chess in 3D that's easy to understand and control and also due to the belief that it would be difficult to make it generalised/not need game mode specific code. It ended up being a very well-received feature and, if a user doesn't like it, they can simply not use it.

Alphabetically Encoded IP Addresses

I initially planned to encode IP addresses to alphabetic string to make them easier to share with other people however I ended up not implementing this, not due to implementation complexity, but because I believe that it would prevent more technical users from diagnosing multiplayer issues and that if the server I used to obtain the public IP was blocked by a firewall or offline, the public IP they

would get by looking online would be in a different format to those usually given in the game causing confusion.

Implementation plan from encoding:

```
(0 - 255).(0 - 255).(0 - 255).(0 - 255)
8bytes.8bytes.8bytes.8bytes

2^(4*8) = 4294967296 combinations
log26 = 6.8 -> 7

26^7 = 8031810176 combinations
7 letter code


-----------------------------------
192.168.0.1
11000000.10101000.00000000.00000001
= 3232235521
MOD 26 = 21 = v

DIV 26 = 124316750
MOD 26 = 12 = m

DIV 26 = 4781413
MOD 26 = 13 = n

DIV 26 = 183900
MOD 26 = 2 = c

DIV 26 = 7073
MOD 26 = 1 = b

DIV 26 = 272
MOD 26 = 12 = m

DIV 26 = 10
MOD 26 = 10 = k

Code: VMNCBMK
```

```
-----------------------------------
VMNCBMK

K + (M * 26) + (N * 26 * 26) ...
= 3232235521
bin = 11000000101010000000000000000001
= 11000000.10101000.00000000.00000001
= 192.168.0.1
```

AI Multithreading

The AI was initially meant to be multithreaded however, after testing, I found the AI to be about the same speed as alpha-beta pruning relies on moves being searched sequentially so the increased speed due to multithreading was cancelled out due to reduced optimisations. In addition to this, the

increased CPU usage could reduce in-game framerates and requires more processing power. I address this more here: Using Multithreading for the MiniMax Algorithm

Final Feedback

Build Version	Feedback	Response to feedback
V1.6 Final - Older Teenager	- No team point tracker - No way to return to lobby after match - Good way of combining gamemodes - 3D mode looks good - Stable - AI presents fun challenge	- Addressed in main document
V1.6 Final (Second Tester) - Older Teenager	- Looks good - Stable - Wide variety of gamemodes - Hosting system difficult (due to port forwarding) - UI is functional but could be more visually interesting	- Addressed in main document
V1.6 Final (Third Tester) - Older Teenager	- Help system good with useful images - Looks good other than control hints which look a bit out of place - UI is functional - Good variety of gamemodes - AI is a good addition for playing alone	- Addressed in main document
V1.6 Final (Fourth Tester) - Adult	- Good sound design - Pieces recognisable - Board isn't rotated based on which side you're playing - Checkers queen difficult to recognise in 3D - Good animations for piece movements in 2D - Great 3D animations for selecting, moving and taking pieces	- Addressed in main document
V1.6 Final (Fifth Tester) - Under 10	- UI understandable - Understood how to start a game with a small amount of trial and error - Understood how to use in-game controls from previous experience playing chess games and from in-game control hints - Thought 3D mode looked great - Didn't know that 'AI' meant playing against a computer but said that they would have understood if it was 'CPU' as previous games played have used this to represent a computer	- Addressed in main document

Key Criticism

- No point tracker
    - While this could be difficult to implement as the points are tuned for the AI and not the players, a secondary point system could be implemented. While possible, showing icons for pieces taken would not be easy to implement as currently the GameManager has no way to output taken pieces and a piece moving to where an enemy piece was doesn't guarantee that in the current game mode that means a take.
- No way to return to the lobby after a match
    - While difficult to implement with the current networking system as it is not meant to be resettable, this would be a large quality of life improvement as retyping IPs is cumbersome. This could be implemented in the next version
- Hosting system difficult to use
    - This was mainly due to the difficulty of port forwarding and sharing IPs. In the future the game could have cloud servers or, at the very least, a relay server that both sides would connect to removing the need for port forwarding.
- Board not being rotated towards you
    - While this is very frustrating if you are, for example, playing black in chess, a very good example for why there isn't a solution to this is the connect four game mode

which should not be rotated based on team. The best solution for this would be for the GameManagerData or BoardManager class to contain extra information about which direction each team is facing from. This could be included in a future update

- Control hints look a bit out of place
    - This could be addressed in a future version with either a settings option or having a togglable overlay that tells you the controls
- Checkers queen difficult to recognise in 3D
    - The 3D model could be updated to stand out more in a future version

Future Improvements

MacOS and Linux Versions

The Unity Engine allows me to build the game with no code changes on MacOS and Linux making this a very easy port. There would be some minimal changes to make such as the way the saves folder is shown as currently, it is Windows exclusive:

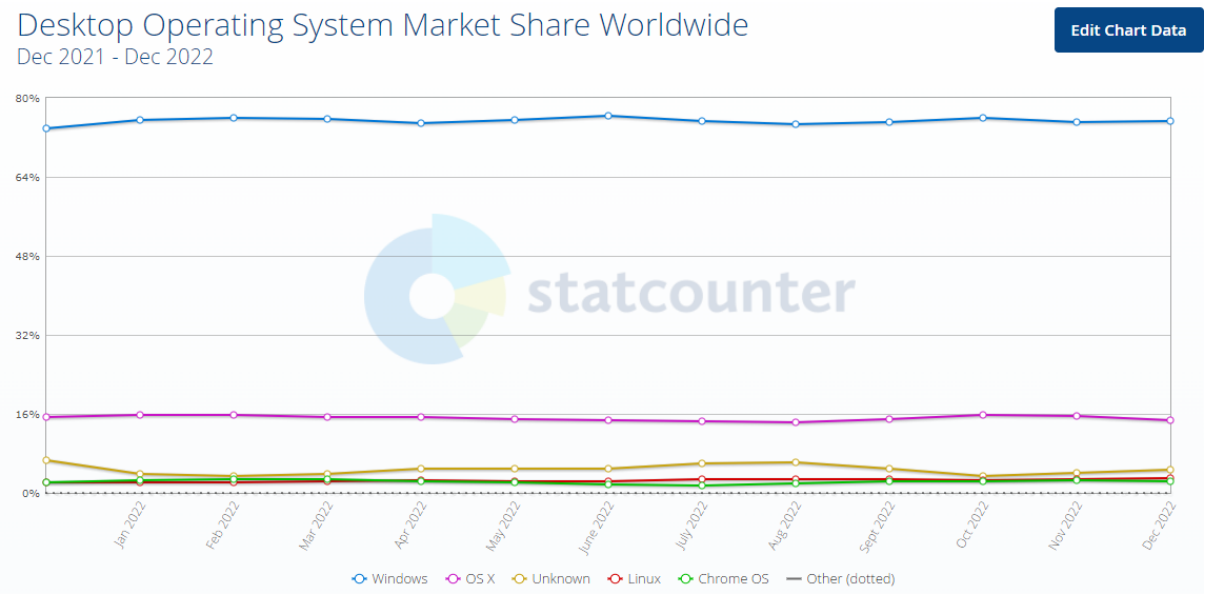
```
        // Windows only open file explorer
#if PLATFORM_STANDALONE_WIN || UNITY_STANDALONE_WIN || UNITY_EDITOR_WIN
        Process.Start(new ProcessStartInfo()
        {
            FileName = save_location + "\\",
            UseShellExecute = true,
            Verb = "open"
        });
#endif
```

Another example of this Windows specificity is the Help System as the way it opens the HTML help files is Windows exclusive:

```
    public static void OpenHelp()
    {
#if PLATFORM_STANDALONE_WIN || UNITY_STANDALONE_WIN || UNITY_EDITOR_WIN
        Process.Start(Application.dataPath + "/StreamingAssets/Help/index.html");
#endif
    }
```

However, there are easy ways to implement these in the other operating systems.

The biggest hurdle currently is my not having access to a Mac device. As the proportion of people using Linux is very low, I don't believe it would be worth supporting.



Source: https://gs.statcounter.com/os-market-share/desktop/worldwide

Mobile Version

As I am using the Unity Engine, I can currently build the game for Android or IOS however the UI is designed for a keyboard, mouse and large screen so buttons would have to be made bigger and controls only available through key presses would have to get on-screen buttons. The 3D mode uses a right click + mouse move to pan the camera and a left click to select squares. To implement this on mobile I'd need to add a toggle between camera movement and square selection.

All these differences between PC and mobile would be difficult to keep track of and separating the two versions would require implementing all features twice.

Another issue is the reduced computing power. As mobile phones tend to have weaker processors, the AI performance maybe he heavily degraded resulting in an AI that is too easy to beat and the lower thermal capacity of mobile devices may result in the device getting uncomfortably hot while searching for moves.

The Help System which uses local, static HTML files would also be difficult to port as I would require Android System Webview integration on Android and would need to research how to support it on IOS as I am not familiar with the platform.

Using a Relay Server or Centralised Server

Currently, users connect to each other directly using an IP to play together. While this works reasonably well for play on a LAN, when trying to play with others over the internet port forwarding needs to be set up which is quite complicated and technical. With a relay of some kind, both players would connect through the relay removing the need for port forwarding. With a centralised server, this could be expanded to having a game browser so you could play with random people and join friends just by searching their names. The server could track player statistics and potentially an in-game currency that could be wagered on matches. While this would take a long time to implement due to the complexities of cloud computing, the relay only needs to establish connections between clients and forward whatever it receives without needing to process it at all and, as for the centralised server, the separation of client and server in the current code wouldn't make it too

difficult to port the server to a dedicated one however implementing the game browser and statistics tracking would be complicated as it would also require authentication.

AI Improvements

A future improvement I could make would be to add a transposition table – a dictionary matching hashes unique to a board state to their calculated value. This would allow previously seen board positions to be looked up instead of recalculated improving speed. It could also improve AI intelligence as if the AI is looking 5 moves ahead and a position is found at depth 1, its score will be calculated by looking 4 moves ahead. If the same board position is later found at depth 4, the score found at depth one will be used and, as that one was calculated by looking ahead 4 moves, it will have a combined depth of 8 moves.

The main difficulty with implementing a transposition table is adding support for any game mode while keeping the hash size low and without requiring per-game-mode implementations. This rules out using the savegame as a hash or using a combination of C#s built-in `GetHashCode()` or game-mode-specific solutions such as Zobrist Hashing for chess however I do believe that with enough time I could find a solution.

Art/Modelling Improvements

Art wasn't a priority in this project resulting in many areas in which it could be improved. An example of this is in Viking Chess where the 3D models used are the same as those in Chess when in most real versions they have pieces more closely resembling soldiers:

Source: https://imgur.com/gallery/zX50Z

UI Improvements

While the UI is functional, I believe that it could be improved with a different style and more animations and transitions such as fading elements in and out. A different style would, however, require a large amount of custom art/graphic design which is why I went with the current style.

The UI could also be improved by creating different layouts for different aspect ratios as currently, it is a one-size-fits-all system which could leave some space unused on less conventional aspect ratios. If the game was ported to mobile devices this would be absolutely necessary as mobile devices come in many different aspect ratios and people might want to play in portrait mode.

Additional Game Modes

While I believe that the game modes currently supported are varied and interesting, as a key component of this project has been making new game modes easy to add, it wouldn't be difficult to add extra game modes such as Go or Othello.

Returning to Lobby

One improvement I heard suggested a couple of times was having the ability to return to a lobby after a game to play another. This would be a great improvement for online play over local play due to the details you have to re-enter to play online. I didn't implement this feature due to the time it would take as, for example, the game mode and save game (if one exists) are transferred during the initial handshake when other one-time processes such as password checking occur. It would take a very large code rewrite to reset all the variables that change during the game and make save game and game mode changing available at any time.

Miscellaneous

Improving Packet Documentation

The packets generated by Python code have no documentation or comments as my Python code did not support this. These could not be added manually as every time the generated packets are updated, all files are completely reset to the generated version.

Glossary

Board

Handles the board of the current game mode. Notably implements GetRenderInfo which returns details about how the board should look such as its size.

ChessManager

At the very top level of the program - orchestrates all other components.

Client

Handles connection and communication with the Server. Handles sending messages at the request of the ChessManager (going through the NetworkManager) and informs the ChessManager of incoming updates (also through the NetworkManager).

GameManager

Manages a game mode and its inputs and outputs including retrieving possible moves, applying moves, getting a score for the board position, cloning itself and saving to and loading from binary data.

GameManagerData

Uninstantiated form of GameManager containing key data for selecting a game mode before one is chosen. Notably contains a UID, name and description for the game mode in addition to the number of teams, their sizes and, optionally, aliases for the teams such as ‘White’ and ‘Black’

NetworkManager

Starts the Client and Server. Acts as a middle-man between the ChessManager and Client/Server when the ChessManager requests a message be sent or the Client has a new update for the ChessManager.

Packet

Encoded item of data sent by the Client or Server. Contains a UID that the recipient uses to know how to decode it.

Piece

An individual piece on the board. Has methods for getting the moves it can make, its value and its appearance.

Server

Handles Client connections. Functions primarily as a relay between clients and barely interacts with the rest of the program. Notable exceptions include validating connections and host-exclusive options such as changing teams and starting the game.

TeamSize

A struct containing a minimum and maximum size for a team.

VisualManager

Handles the appearance of the 2D board. Calls methods on VisualManagerThreeD to copy its movements

VisualManagerThreeD

Handles the appearance of the 3D board. Most logic is handled by the VisualManager with the VisualManagerThreeD only applying updates.

V2

‘Vector2’. 2D vector containing an integer x and y.