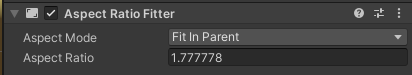
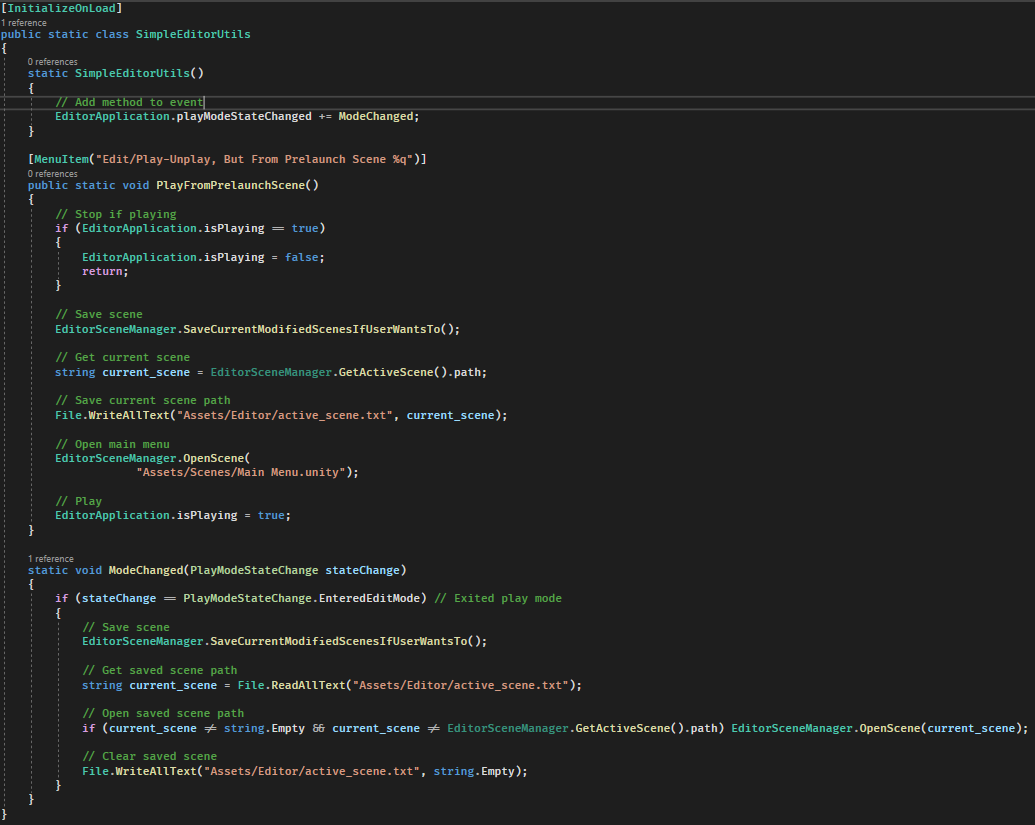
# Accounting for all display sizes and ratios is difficult

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



# Entering play mode is frustrating

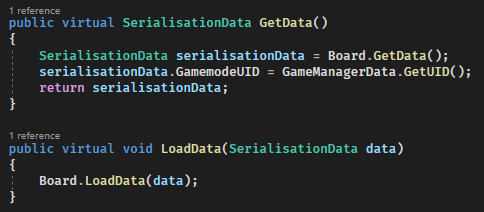
As the main menu initializes some classes, to run the game it must be run from the main menu. This is frustrating as you have to switch scenes, play, stop and switch back for every small change made. To solve this I created a script that does that for you when you press Ctrl+Q



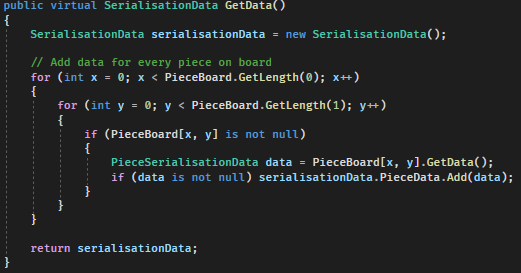
# Need to be able to save games which can have custom data

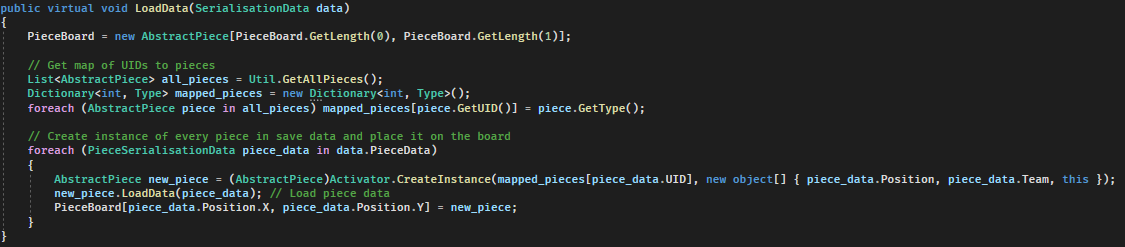
I need to be able to save games of various gamemodes and save board configurations which have pieces that may themselves have additional data. I used a combination of polymorphism and UIDs to achieve a robust system. The Gamemode is just stored as a UID with extra data being saved by overriding the ‘GetData’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

Implementation for game manager

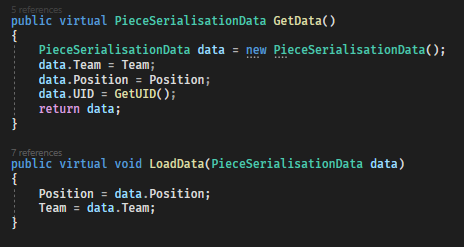


Board implementation

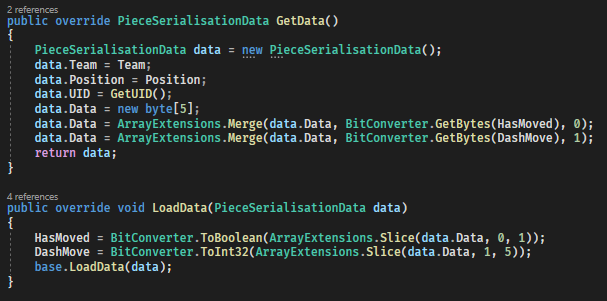




Piece implementation



An example of custom piece data might look like this



The file is formatted as follows

|  |  |
| --- | --- |
| **Data** | **Length** |
| Length (of full file) | Int – 4 bytes |
| GamemodeUID | Int – 4 bytes |
| TeamTurn | Int – 4 bytes |
| PlayerTurn | Int – 4 bytes |
| Ellapsed time | Long – 8 bytes |
| GameManagerDataLength | Int – 4 bytes |
| GameManagerData | Any length |
| BoardDataLength | Int – 4 bytes |
| BoardData | 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 length |

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

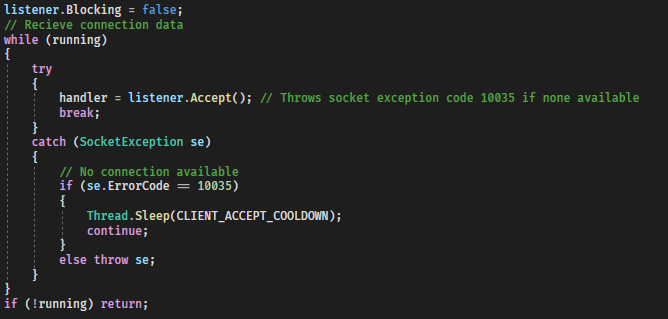
This code waits for a user to connect



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

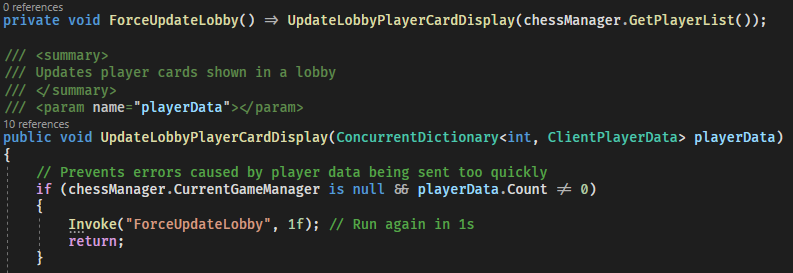


This caused another issue as listener.Accept() now throws an error if there is no client waiting so I wrapped it in a try catch loop



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

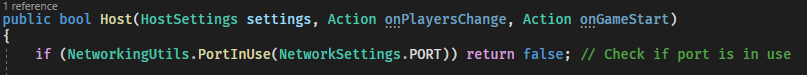
This ‘ghost’ user didn’t affect the game at all however it was confusing for players. Through repeated testing with different conditions, I found that this happened when a player had a team assigned and then another player joined. The player information and the player team information would be sent in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



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

# Trying to host twice in quick succession creates errors

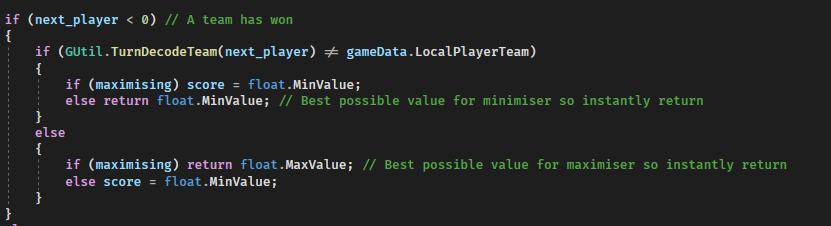
This was found to be due to the socket used for the server being in ‘TIME\_WAIT’ state. To remedy this (as everywhere I looked online, I couldn’t find a way to avoid this) I added a message asking the user to wait if the socket was still in that state





# The AI doesn’t pick the best move

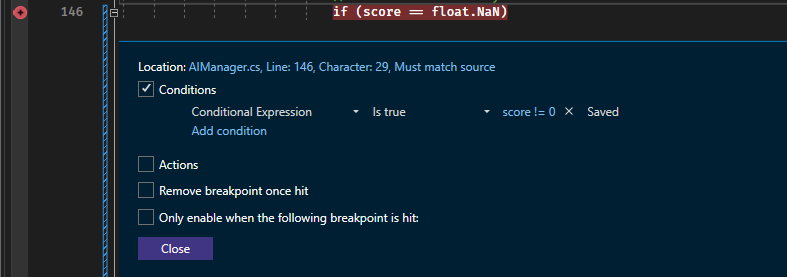
This was found to be due to the AI misidentifying some wins as bad and some losses as good



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

# The AI sometimes freezes

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



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





The reason for this not working is that

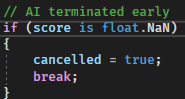


Which I found using Visual Studio’s interactive C# console

The reason behind this decision is that 1 / 0 should not equal 2 / 0

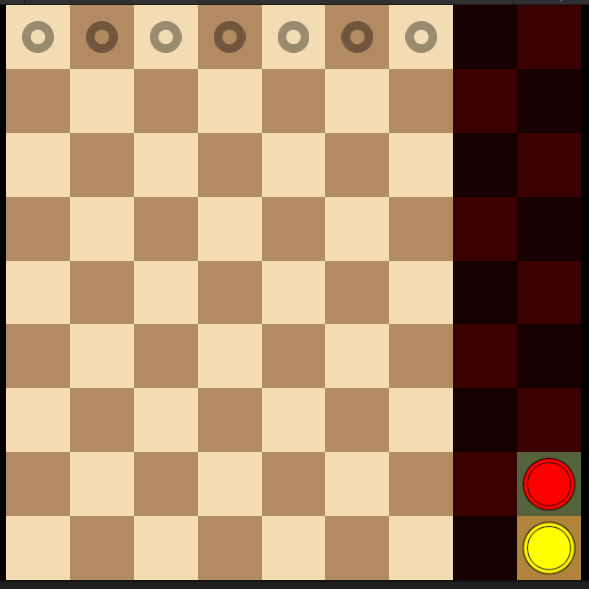
C# does have a different comparison operator ‘is’ which works here





# Connect 4 doesn’t fit the input system

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



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