

Three-Dimensional Chess<https://github.com/tomortimer/ThreeDimensionalChess>Contents

Contents.....	1
Research.....	2
Introduction .....	2
Analysis of Current Systems .....	3
Sampling .....	11
Interview with an End User – Competitive Chess Player.....	11
Surveying Questions: .....	12
Survey Results: .....	13
UI Mock-up .....	17
End User Feedback on UI Mock-up .....	22
Requirements.....	23
Design .....	28
Flowchart .....	28
Class Diagram.....	29
Sequence Diagrams .....	30
Ruleset .....	33
Algorithms.....	34
Database Design .....	39
Technical Solution Highlights.....	41
Linked List Maintenance .....	41
List Operations .....	43
Stack Operations.....	45
Object Oriented Programming: Polymorphism .....	47
Recursive Algorithms .....	49
Complex Algorithms: Merge Sort .....	52
Object Oriented Programming: Dynamic Generation of Objects Based on User Input.....	53
Cross Table Parameterised SQL .....	54
Defensive Programming .....	56
Good Exception Handling .....	57
Implementation .....	58
UserInterface.cs .....	58
Chess.cs.....	85

Board.cs .....	94
Square.cs.....	103
Piece.cs .....	104
Bishop.cs .....	109
King.cs .....	112
Knight.cs.....	115
Pawn.cs .....	116
Queen.cs .....	120
Rook.cs .....	121
ThreatSuperPiece.cs .....	123
SimulatedBoard.cs .....	133
DatabaseHandler.cs .....	135
GameInfo.cs .....	142
Player.cs .....	143
Sorter.cs .....	145
Data Structures Implementation .....	157
List.cs.....	157
ListNode.cs.....	161
Stack.cs .....	162
File Structure.....	163
Evaluation and Testing.....	164
Test Plan.....	165
Test Results .....	171
Testing Video .....	171
Evaluation Plan .....	173
Evaluation of Requirements .....	173
End User Assessment.....	179
Potential Improvements .....	180
Project Summary .....	181

## Research

### Introduction

I intend to create a client to play three-dimensional chess. A variant of chess played on an 8x8x8 cube of cells that may be occupied by pieces. As well as standard chess moves, pieces can move in the third dimension. An easy way to visualise the game is as if there are 8 chess boards stacked on

top of each other and a piece can occupy any square on those boards as well as being able to move between them. To drive the design of this game and client, I have researched how chess is conventionally played through a computer as well as the physical attempts at playing this game or similar variants of chess.

### Analysis of Current Systems

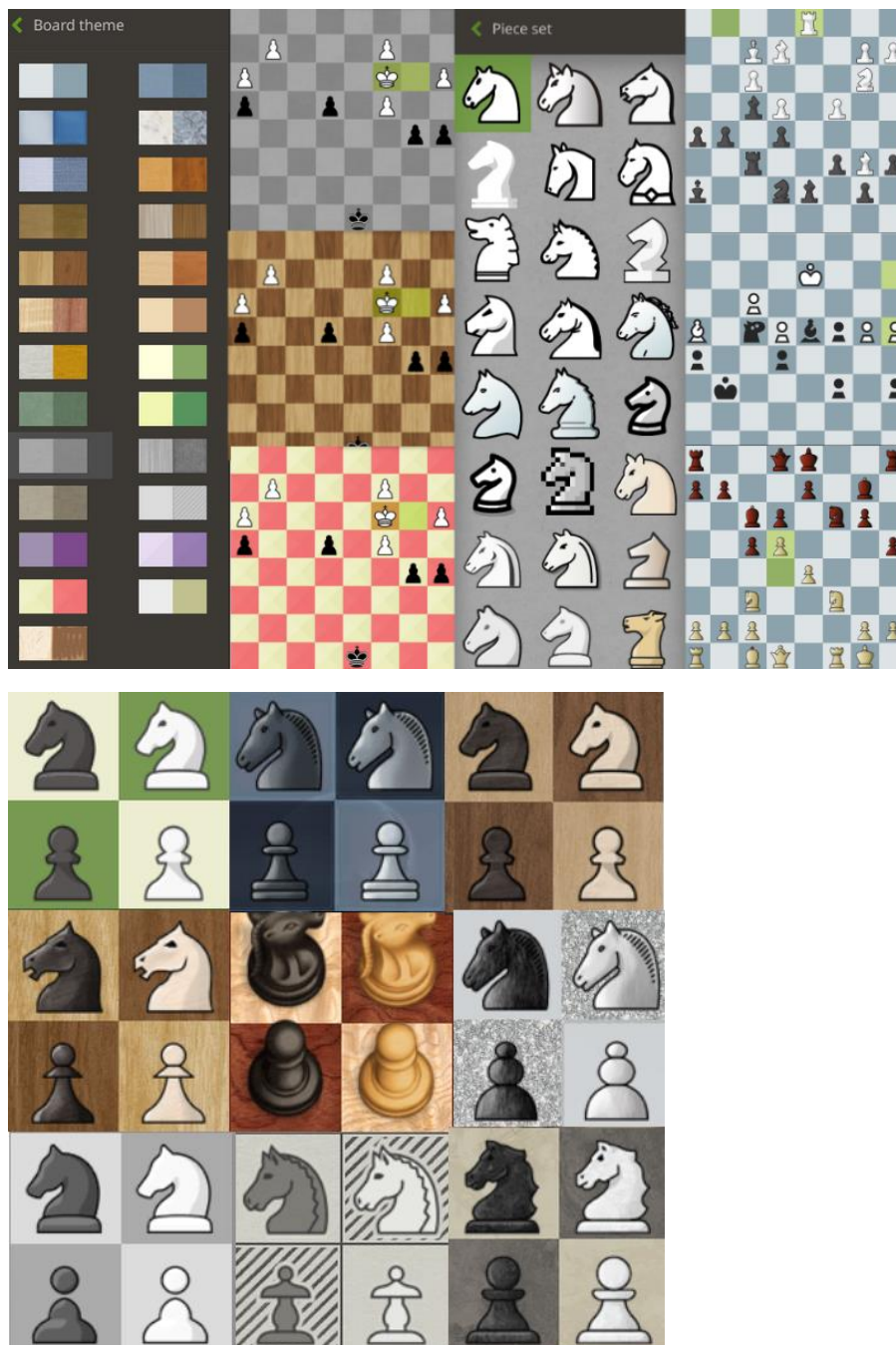
There are two parts to this section, one being understanding the function and features of online/digital chess and the other being the analysis of alternate chess rulesets which may be similar to the one I'm designing here.

#### Online Chess:

Since the late 20<sup>th</sup> century, people have used computers to play chess, either for pitting two programs against each other or allowing people to play chess together over long distances using the internet. This means that over decades of design and iteration, chess on computers has been continually improved to widely approved of state. The most popular means of playing chess online today are chess.com and, a wide margin later, lichess.org.

Both of these sites feature a menu which branches into various chess puzzles, learning tools and community features, as one would expect from an online platform – as well as, of course, the simple option to play a game of chess. These features can be disregarded mostly, as my game will not have online functionality, we are simply here to learn more about the chess interface with a computer, but we may still consider their significance to the end user later.

Both platforms offer the option to customise the look of the board and pieces, with a variety of colour schemes and piece designs. Though the ones that most appeal to my aesthetic sensibilities are those piece designs that are more **minimalist, yet bold**, and colour schemes that are muted, giving more attention to the pieces. Looking at these options, it's important to note **that the black squares are often a far softer colour in comparison to the black pieces** (if they are the same colour) to prevent confusion. The same can be said of the white squares, with their off-white tone. In the case of the board being completely different colours from the piece set, like some of the examples here, the colour distance between the "white" and "black" shades of each square is important. They should be distinct, but it should remain visible that they are the same base colour or material.



Though piece design varies slightly by set, the outline is always the same or very similar, meaning that pieces are easily recognisable at a quick glance. This is something I have taken note of, *I will use minimalist icons for the pieces, using their very recognisable silhouettes to reduce visual clutter* in an already complicated game.

To build on the importance of square colours, there are a host of *other colours used in tandem with standard chess colours to convey meaning* that would otherwise be obvious in a game of physical chess or information that is simple to convey digitally but would be difficult to show on a physical game. Digital chess very much builds on the foundations of physical chess to make the game more accessible.

The examples of this we can see, even in the graphic above to show different board options, where *a yellow shade is applied to a square to show that a piece moved from or to it on the last move*. This

shows an important part of physical chess that is taken for granted, one is unlikely to miss an opponent picking up and placing a piece, but in the fast-paced digital environment, moves can easily be missed, especially if they don't change the overall layout of the board much. Another example would be selecting a piece, **which highlights possible moves in another shade, possibly green, yellow or red** – this being a feature that is impractical with physical chess. These features are very important as they **streamline the experience and also make it much easier**, without these the digital chess experience would be clunky and frustrating. As a result of the importance of both these features, I intend to fully implement them.

#### Resulting Requirements:

1.4 - When the user clicks on a friendly piece, squares that the piece could possibly move to should be tinted in blue

1.5 - When the user clicks on an enemy piece, squares that the piece could possibly move to should be tinted in red

1.7 - After each move, the square a piece started on and the square that a piece finished in, should be shaded in yellow

1.71 - Reset this for each move so only one pair of squares is shaded yellow at a time

An important part of the online chess experience, that we will also notice later when surveying potential users, is the significance of tournament play and as a result **rankings**. Chess since its inception has been a heavily competitive game and this is reflected in attitudes to the game. Another result of this **competitive nature**, is the desire of the players to improve, looking back on their games for inspiration or to see their flaws. These factors have inspired another important function of the final product, **to store past games and be able to rank players by their Wins and Losses**. This will be **implemented on a local level**, due to the lack of online functionality in the intended final product as it is above the scope of the project. -> Requirements: 3.4, 4.1, 7.2

#### Resulting Requirements:

3.4 - When the game ends it should automatically be saved to the database

4.1 - Players can be stored in a database

4.11 - The number of games won, lost and drawn by each player should be recorded

4.11a - The total number of games played should be recorded

4.11b - Record the number of games won as white and the number of games won as black

4.11c - Record the number of games played as each colour

7.2 - The "Players" menu should show a list of all players displayed as a table

7.21 - Using a navigation panel at the top of the table, the list can be sorted according to "Name", "Date Registered", "Overall Wins", "Winrate as Black", "Winrate as White"

Chess.com features options to change the **sound** of pieces as they are moved, lichess does not offer this functionality – this is a result of trying to make the digital chess experience **more intuitive and welcoming** to people who have only played chess physically. Having options to choose from is a nice feature, however most of the sounds are overly jarring and **I would opt to simply have only one set of sounds**, those being similar to the default, **wooden sounds** of lichess. This is because I have identified the **extra sounds as being unnecessary** and taking more resources than it's worth, for my small-scale project.

#### Resulting Requirement:

1.41c - A sound should play when a piece is moved onto a square:

A hollow wooden sound for moving onto an empty square

A solid wooden sound for taking an opponent's piece

1.	e4	e5
2.	Nf3	Nf6
3.	Nxe5	Nxe4
4.	f3	d6
5.	Nc3	Nxc3
6.	dx3	<b>h6</b>

Moves in both clients use standard, algebraic chess notation to record moves, having them in a list such as this allows some useful features. One use is that a game can be [replayed by stepping through the list from the beginning](#), whether digitally or physically, [this notation is used for any medium](#) and is therefore widely understood. The storage of moves in this manner also allows [them to be rewinded](#) easily, moving the game back to an earlier state – this being one of the [optional rules](#) found in digital chess. The ability to rewind moves in such a manner would be useful for a complex version of chess, such as 3D chess, where it is easy to make mistakes, thus this is one of the optional rules I would like to implement.

#### Resulting Requirements:

##### 6.1 - Optional Rule 1: Players can rewind the moves of the game

6.11 - Next to the list of moves in the UI (See 1.7) there should be a rewind button

6.12 - This button should undo the last move, returning the board to the state it was in before the last move was made

6.13 - Any moves made after having rewinded should overwrite the move list and delete any entries later than it, to avoid creating problems when exporting/importing games

Leading on from optional rules, and the competitive nature of chess, chess.com and lichess and offer the option to have a move timer running alongside the game. If your timer runs out on your turn, then you lose. Whilst this is an interesting option that could [drive familiarity](#) with 3D Chess and add to the [competitive nature](#). However, when researching chess variants, I found little mention of chess clocks being used, due to the base game being more obtuse now. I will follow in this vein, since it would make the game [significantly less fun and harder due to new time pressures](#).

An [optional setting](#) that is offered by other chess clients, not so much the two mentioned here since they focus on online chess, is that ability to [swap the orientation of the board](#), based on which player's turn it is. With digital chess this is very much an [ease of access feature](#), making the game seem more [akin to playing with a physical board](#). This is something to be considered to induce a familiarity with the game however, for now, [I am unsure how effective or useful this would be with such a different board shape](#).

The standard notation would also be useful for storing the current match, with past moves intact, however we run into two issues. The first being that [this notation is 2D](#), with the first 8 letters of the alphabet used for the X axis and numbers up to 8 used for the Y axis. A solution to this problem would be to add another digit, signifying Z axis – for example [Raumschach](#)(see chess variants analysis) using capital letters A-E for its third dimension. This would clash with using capital letters for pieces in notation. There is also the option of using the first 8 Greek letters, which forms the least

conflict as seen in *Kubikschach*, however this wouldn't necessarily work smoothly with my programming language and it's also cumbersome to type out Greek letters. So, I have identified another solution, which takes elements from both: [using the last 8 letters of the alphabet\(lowercase\) and making the "x" for capture be a capital "X" instead](#). The other issue is that pawns are no longer constrained to ranks due to their 3D movement, this is simply solved by [using a P for pawn in notation and not cutting them for conciseness](#).

Pure Coordinate Notation:

```
<move descriptor> ::= <from square><to square>[<promoted to>]
<square>          ::= <file letter><rank number>
<file letter>     ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h'
<rank number>    ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8'
<promoted to>    ::= 'q' | 'r' | 'b' | 'n'
```

Long Algebraic Notation:

```
<LAN move descriptor piece moves> ::= <Piece symbol><from square>['-'
'|'x']<to square>
<LAN move descriptor pawn moves>  ::= <from square>['-'|'x']<to
square>['='<promoted to>]
<Piece symbol> ::= 'N' | 'B' | 'R' | 'Q' | 'K'
```

The program will internally use a version of [Long Algebraic Notation](#) as it is [easiest for the computer to parse](#), being fully unambiguous, except for piece colours which can be much more easily inferred. The reason for using LAN instead of PCN is to [allow for move rewinding more simply](#). A Simplified Algebraic Notation may be converted to be shown to the players for familiarity's sake however it is advisable that LAN is the format used for storage too. As discussed above, some symbols will need to be changed for working with three dimensions, resulting in the below notation:

Long Algebraic Three-Dimensional Notation (Square is redefined):

```
<LA3DN move descriptor piece moves> ::= <Piece symbol><from square>['-'
'|'X' <Piece symbol>]<to square>['='<promoted to>]
<square> ::= <file letter><rank number><depth letter>
<depth letter> ::= 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
<Piece symbol> ::= 'N' | 'B' | 'R' | 'Q' | 'K' | 'P'
```

Resulting Requirements:

1.6 - There should be a list of past moves represented in a list from the most recent move to the first move

1.61 - The move representation should use chess notation, with the last 8 letters of the alphabet to represent the coordinates in the third dimension

4.02 - Saved games can be loaded from the Load Game menu (See 7.12)

4.02a - The pieces positions are loaded and applied to the board

4.02b - The move history is loaded into the Move History List in the UI

7.12d - Upon selecting a game to open, the user should be taken to intermediary screen where it shows the game settings and current move. The user can then confirm this is the game they intend to load and upon clicking on a large, simple "Start" button the game is loaded.



### Chess Variants:

There are records of three-dimensional chess having been played physically by Lionel Kieseritzky who developed *Kubikschach*, which is German for “Cube chess.” This required **eight 8x8 boards to be laid out next to each other and labelled**, commentators described it as the most “mentally indigestible” form of the game. Thus, leading to the creation of *Raumschach*, “Space Chess,” which used a **5x5x5 board** though there are also other, less prolific, versions which reduced the number of vertical boards or size of the board. With the aid of a computer program, this obtuse version of chess should hopefully be easier to understand; by providing both the **ease of moving a piece of adjacent boards** and by also **rendering a 3D overview of the board to help understanding of the game in a way not physically possible before.**

### **Resulting Requirements:**

**1.0 - Store an 8x8x8 three dimensional board (512 cells total)**

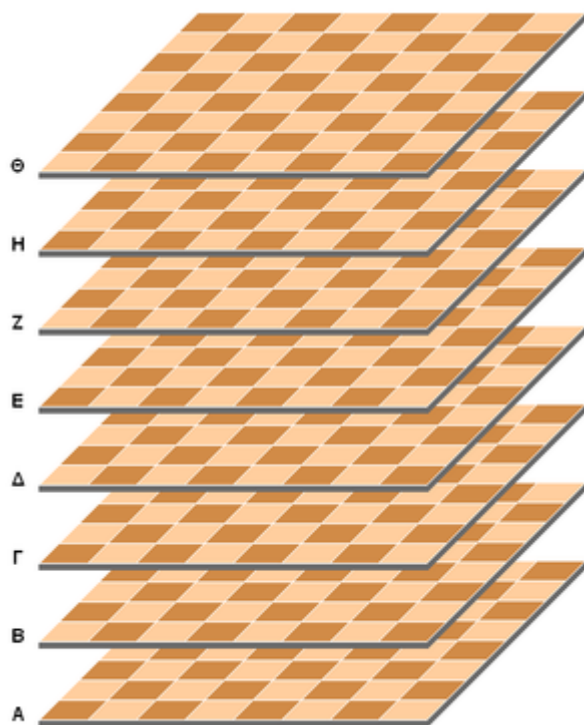
**1.8 - There should be a button which changes the UI to a screen showing a 3D representation of the board**

**1.81 - Any piece selected on the 2D view will have its possible moves shown on the 3D representation**

**1.82 - Include a toggle to show the outlines of every square otherwise only draw outlines of possible move squares**

**1.83 - Show the square moved to and from in yellow**

**1.84 - Have a button to return the view to the 2D representation of the board**

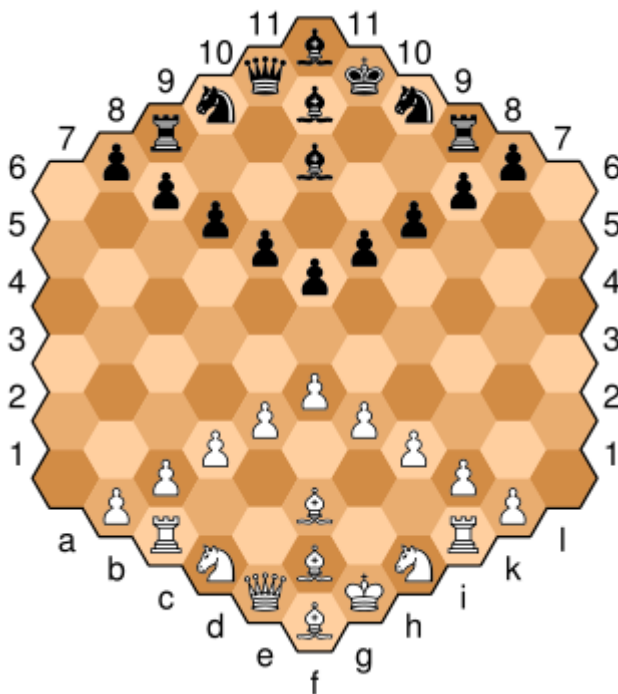


Another interesting point to note about *Raumschach*, is the way that the moves are described. “Rooks move through the 6 faces of the cube and Bishops move through the 12 edges of the cube,” this has heavily **inspired how I’ve approached the ruleset of my 3D chess.** Though to **allow new players to be more familiar with the game** I’ve cut some extra rules, such as the inclusion of a “Unicorn” piece which “moves through the 8 corners of the cube,” which is an interesting concept in itself but **introduces too much unfamiliarity**, a problem I will talk about in a few paragraphs.



Simplistic versions of extra-dimensional chess appeal to casual and competitive players alike, as the **unfamiliarity of a chess variant evens out the playing field** between the two players. For example, *parallel worlds chess*, which features only three board depth but an extra army, with the third board featuring as a middle ground for the pieces to traverse to the other board with. At the other end of the spectrum is *5D Chess – With Multiverse Time Travel*, a version of chess that is so complicated people “win/lose without realising,” however it is still **widely popular due to its novelty**. As a result, I can infer that I shouldn’t be overly worried about how complex the game is, because **even this idea taken to its extremes is playable and enjoyable with only a basic understanding of chess**.

Some variants of chess are called *Fairy Chess*, this means that there has been in a **change in the way some rules work**, such as win condition being changed from checkmate, or the **behaviour of a piece**. As mentioned above, about *Raumschach*, this new or altered pieces created a degree of unfamiliarity that **pushes away potential players** more than simply changing the board to a point where it’s harder to understand. As a result, I decided to **avoid any fairy chess pieces** and subsequently avoid the implementation of *Raumschach*’s “unicorn” piece.



Even different two-dimensional **board shapes can pose issues**. In this picture of *hexagonal chess* we can see that there are now three different shades of square to account for the increased numbers of cells in contact. This is thankfully **not an issue for 3D chess**, since the design remains cubic and thus there are only two colours of cell (**Requirement: 1.01b, shown below**). Another interesting point that this shape of board raises is that **changing the shape of the board can have unforeseen consequences on the way a piece moves**. Here, Rooks can now move in six directions and bishops are very different, becoming almost like Knight’s as the ruleset identifies the quality of bishops that they stay on the same colour rather than their natural, diagonal movement. **This affected the way I approached the pawn’s rules**, the pawn is to move towards the opponent’s piece-line; therefore, in white’s case, a pawn moves positively on the Z and Y axis, and a black pawn moves negatively on the same axis. **With each colour starting at opposite edges (not faces) of the cube** (as is detailed in the *Raumschach* starting positions below).

**Resulting Requirements:**

**1.01b - Square 0 is black(A1), square one is white(B1): Each odd square is white (Alternate the**

starting square colour of each row)

2.1 - There should be a Pawn piece

2.11a - A white Pawn can move up the board or forwards on the z axis by one cell

2.11b - A black Pawn can move down the board or backwards on the z axis by one cell (from white's perspective)

2.12a - It may not move over another piece, it must stop on the cell before it

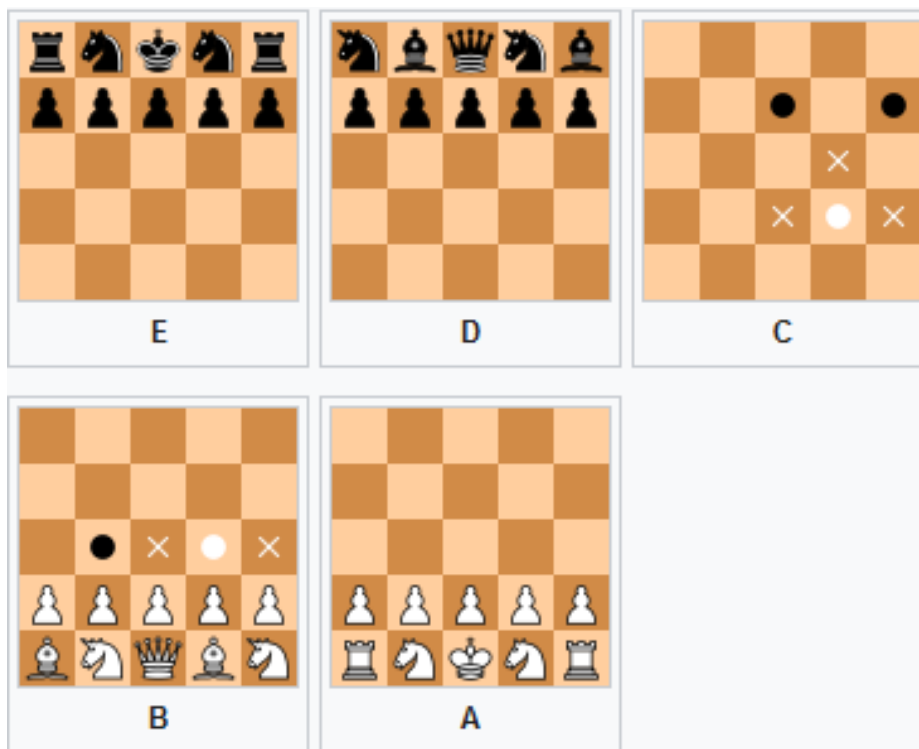
2.12b - It may not finish its move on the same square as a piece of the same colour

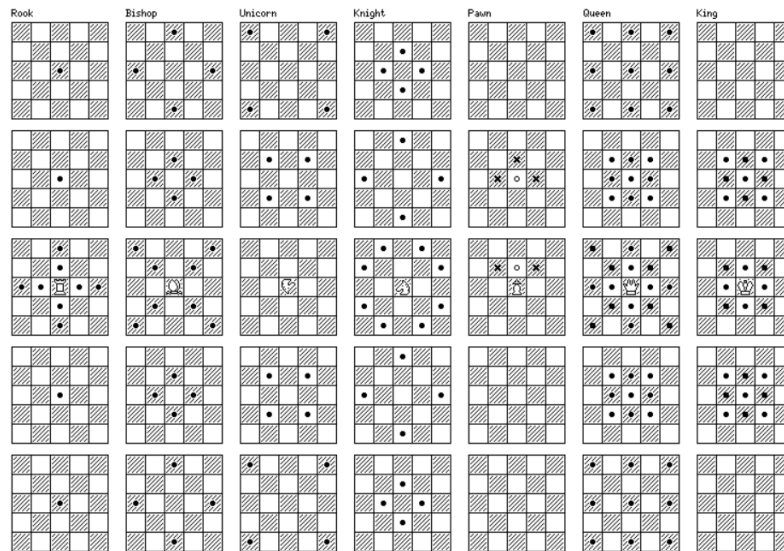
2.13 - A Pawn can take an enemy piece by moving one cell diagonally (on one plane) and finishing its turn on the cell of the piece of the opposite colour

2.13a - A Pawn can only move diagonally to take a piece

2.13b - This move must still be towards the promotion rank, so white pawns cannot capture diagonally one deep and one down; black pawns cannot capture 6diagonally one deep and one up.

2.14 - If a Pawn reaches the end of the board, row 8z for any white pawn and row 1s for any black pawn, it is promoted. The player chooses any piece other than a King or Pawn for it to become.





Raumschach move diagram

Something important to consider about *Raumschach* is that the rules stipulate a lack of pawn double moving on their first turn, subsequently there is no En Passant, and also a lack of castling. To me the lack of pawn double moving makes sense, as it can be difficult to decide how it should be carried out – with further complications arising when you allow for En Passant, as the three-dimensional board means that there are multiple paths to one space with a double move therefore opening up two cells at risk of En Passant, making diagonal pawn double moving very disadvantageous anyway. The lack of castling looks to be mostly due to the 5x5x5 board size and the different starting positions of *Raumschach*. This ruleset is similar to another chess variant, *Los Alamos Chess*, a version of chess that was created for the MANIAC I computer at Los Alamos to play in 1956. It also featured a lack of bishop as well as the reduced board size and lesser rule complexity, to reduce the amount of time it would take the computer to decide on moves. This raises a concern of difficulty implementing these rules and moves for a computer and also on a three-dimensional board. As a result of it being a stipulation of the rules of *Raumschach* which is derived from the *Kubikschach* that my program is based on, I have chosen to not implement pawn double moving or En Passant.

### Sampling

I have interviewed several target users, primarily chess players of varying skill and involvement to collect opinions, recommendations, and data to help shape the final product. I asked about existing chess clients to incorporate their good features and negate their pitfalls into my program, despite it being a different game – a lot of the logistics are shared. After explaining the game, I asked about how a potential player might best like it represented visually. I also sent out a survey to a greater variety of people who may not have as much experience with chess, to see how a more general population currently views chess and this proposed alternative.

### Interview with an End User – Competitive Chess Player

In order to inform the design of my project I spoke with an experienced and competitively successful chess player, Ciaran Brightley-Davies. Whilst potential end users are not all of the same group of him, I believe he will be able to represent some of their concerns too where they overlap. Furthermore, his intimate knowledge of chess and playing chess on a computer will be very useful in deciding on the technicalities of the system.

Q: "What are your main concerns when playing a variant of chess – such as 3D Chess?"

A: "It depends a lot on the complexity of the variant. Whilst I already know how each chess piece works, the shape of the board or extra rules can greatly change that. Therefore, the way you convey possible moves is important. As well as how you interact with the board, I wouldn't want to be having to type in the coordinate of each move or piece!"

**Resulting Requirements:**

1.4 - When the user clicks on a friendly piece, squares that the piece could possibly move to should be tinted in blue (In accordance with piece moving rules and 2.03)

1.41a - If the user then click on one of the highlighted blue squares, the piece should be moved to that square

1.81 - Any piece selected on the 2D view will have its possible moves shown on the 3D representation

1.82 - Include a toggle to show the outlines of every square otherwise only draw outlines of possible move squares

Q: "What aspects of chess on a computer are the most important for your experience?"

A: "The same points about conveying information apply to normal chess too, ease of use is very important to me. As a competitive player, the rankings appeal to me greatly, I like to see my progress and how I match up against other players. In order to see progress, it's also important to be able to review games and review my mistakes."

**Resulting Requirements:**

4.01 - When a game is saved it should include the move history

4.02 - Saved games can be loaded from the Load Game menu (See 7.12)

4.02a - The pieces positions are loaded and applied to the board

4.02b - The move history is loaded into the Move History List in the UI

4.03 - Any extra optional rules should be stored and subsequently loaded when the game is loaded

I will interview him again later with regards to my user interface, once I have created a model of it.

**Surveying Questions:**

- How often do you play chess?
- Do you consider yourself a casual or competitive chess player?

These questions gauge how well potential players know the current game which may influence how they engage with the end product.

- How do you currently play chess?

This question will help me understand how familiar with a chess interface a potential user may be and whether this will influence the design of the UI.

- What do you like about current chess clients?
- What do you dislike about current chess clients?
- What do you think the most important feature or qualities of a chess client are?
- When playing chess, how important are the kinetic feel of the pieces (sound/visual feedback)?

These four questions are used to gather information about other existing forms of digital chess, their best and worst factors, so that I may implement the best and most popular parts of digital

chess whilst steering away from any pitfalls. Although I am creating a different variant of chess, the same considerations in terms of logistics and user taste should apply.

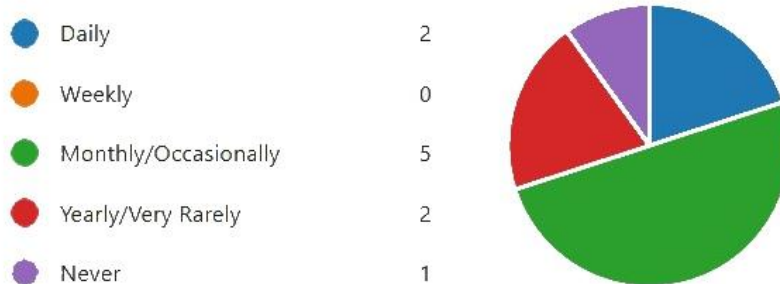
- What chess variants are you familiar with or have heard of?
- What did you like about these variants?
- What did you think was unnecessary about these variants?

Similar to the questions about current chess clients, these three are created to figure out the best and worst parts to implement or avoid. Although the variance in answers will be higher here due to a much more open-ended question due to asking about variants as a whole not only asking about a current service. Some of this may be irrelevant, feedback that's too specific to one variant of chess, but general considerations about game complexity, time and usability can be gleaned from these responses too.

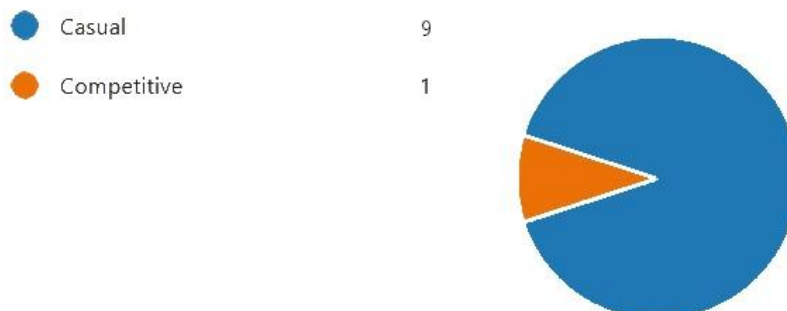
### Survey Results:

Kinds of Chess Players:

#### 1. How often do you play chess?



#### 2. Do you consider yourself a casual or competitive chess player?

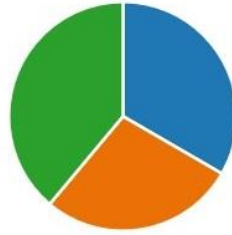


A large majority of people who answered the survey were **casual players**, though their **frequency, and presumably experience, varied**. This may influence later responses in the survey as I am sampling a largely casual audience here, this is suitable for a chess alternative thankfully – as **casual players are more likely to try out different forms of the game on a whim**. It also means that most of the ergonomics and design targeted at this group will be applicable when thinking about how a **person of the general population may approach the game** – since most of the population are either casual chess players or don't play chess.

Current Chess Clients:

## 3. How would you currently play chess?

Online (Mobile)	6
Online (At a computer)	5
In Person (with a chessboard)	7



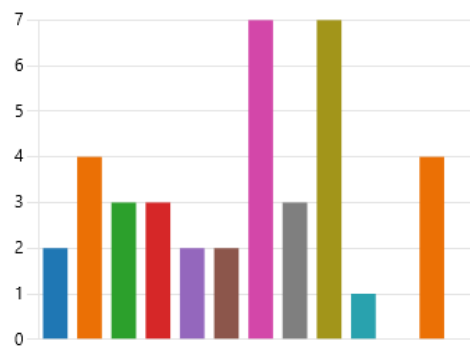
This shows that some people may not be familiar with playing chess via computer, therefore I should heavily weigh in the accessibility and familiarity considerations of current chess services, analysed above as current systems.

These two aspects of the feedback solidified the importance of requirement:

1.4 - When the user clicks on a friendly piece, squares that the piece could possibly move to should be tinted in blue (In accordance with piece moving rules and 2.03)

## 4. What do you like about current chess clients?

Save/Load Games	2
Play over long distances	4
Customisability	3
Rankings	3
Pre-moving	2
Automatic Clocks	2
AI Opponents	7
Possible Move Telegraphs	3
Easy/Quick Access To Chess	7
Rewind Through Moves	1
Sharing Boards	0
Review Games	4
Other	0



Here we can see what features of digital chess other users have identified as important. Similar to my conclusions in the first section, accessibility, game review are very important features, followed by other widely agreed upon features such as rankings and move telegraphing. Unfortunately, some popular features are outside of the scope of my project, that being long distance play and AI opponents, the latter being too intensive to tailor to a 3D dimensional board in the given time.

## Resulting Requirements:

## 4.0 - A game can be saved into a database

4.01 - When a game is saved it should include the move history

4.02 - Saved games can be loaded from the Load Game menu (See 7.12)

4.02a - The pieces positions are loaded and applied to the board

4.02b - The move history is loaded into the Move History List in the UI

4.03 - Any extra optional rules should be stored and subsequently loaded when the game is loaded (See 6: Optional Rules for optional rules)

4.04 - Which players took part in the game and which colour each player was

4.05 - The outcome of the game, this will be "White Win", "Black Win", "Stalemate" or "Ongoing"

4.06 - The date the game begins should also be recorded

#### 4.1 - Players can be stored in a database

4.11 - The number of games won, lost and drawn by each player should be recorded

4.11a - The total number of games played should be recorded

4.11b - Record the number of games won as white and the number of games won as black

4.11c - Record the number of games played as each colour

4.12 - The name of the player should be recorded

4.13 - The date the player is registered should be recorded

4.2 - At the end of each turn the current state of the game should be saved to the database

5.2 - When the game is started each player may select a player profile from the list, fetched from 4.1

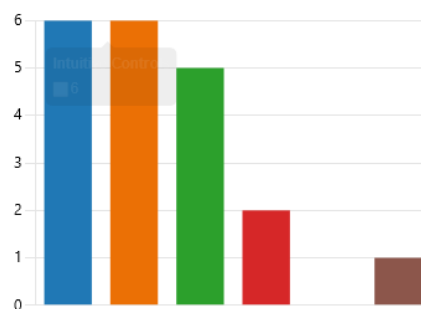
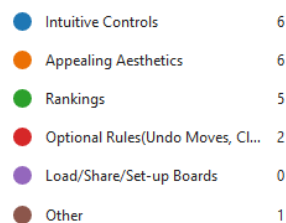
5.21 - The player should have the option to make a new player profile

2	anonymous	I haven't actually used one, but I would fear having to enter in beginning and ending coordinates for each move. I would find that less tangible.
3	anonymous	ease of cheating,

There were few meaningful responses to the question about disliked features. These raise one concern that I am very much already aware of, that being the [ease of use](#). The second concern made me pause for thought however I don't think it should be an issue due to [there not being widely used and tested AIs for three-dimensional chess](#).

6. What do you think are the most important features of a chess client?

[More Details](#)

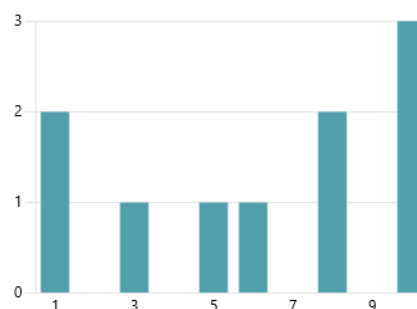


7. How important is the sound and kinetic feel of the pieces when playing online?

[More Details](#)

[Insights](#)

6.20  
Average Rating



With the first question, participants were allowed to choose two answers, to further cut down to the core important features of chess clients. We can see here, despite the largely casual test group,



rankings were seen as important, speaking to the competitive nature of chess and the ease of use came up again – being a primary concern for the casual players and also the few competitive players who are already familiar with such software.

#### Resulting Requirements:

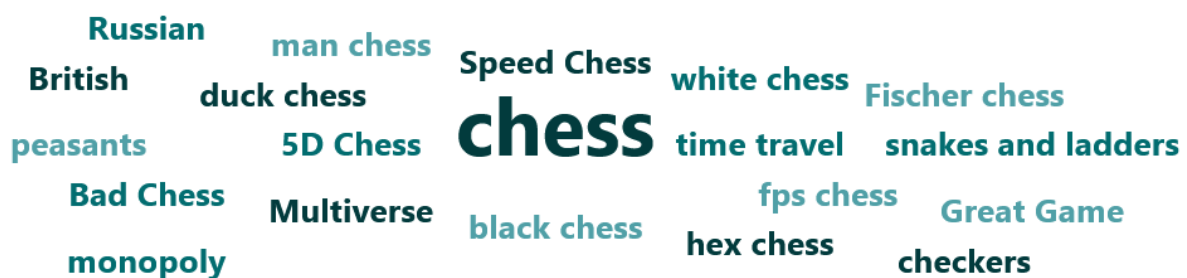
- 1.1 - Ability to look at board from front(z coordinate is constant), side(x coordinate is constant), top(y coordinate is constant)
  - 1.11a - User can view each layer of the board from any of the angles(front, side, top)
  - 1.11b - Each layer is an 8x8 representation of the board
  - 1.11c - User can move up or down a layer by a pair of opposite arrow buttons
- 1.2 - Pieces are represented on the board by icons, they are limited to one square
- 1.3 - When the user clicks on a piece, the UI should display its coordinates
- 1.41a - If the user then click on one of the highlighted blue squares, the piece should be moved to that square
- 5.0 - Chess has two players, one controls the white pieces and the other controls the black pieces
  - 5.1 - The game must keep track of who's turn it is
    - 5.11 - Players can only move one piece on their turn and then it is the other player's turn
    - 5.12 - The UI should display who's turn it is

The second question surprised me with the variance in results, with some being vehement that sound design was important whereas it meant less than nothing for others. Overall, the population of the survey seem to regard the sound of the game as important. As I stated in my analysis of current systems, I don't want to overwhelm the user with too many choices so a simple, effective sound design should be suitable.

#### Resulting Requirement:

- 1.41c - A sound should play when a piece is moved onto a square:
  - A hollow wooden sound for moving onto an empty square
  - A solid wooden sound for taking an opponent's piece

Chess Variants that surveyees were familiar with:



Most of these chess variants fall into one of two categories, either a variant that changes the composition of each player's set of pieces or a variant that changes the shape of the board. Some variants that change the shape of the board also require new or more pieces, such as in *Russian Chess* or *Hex Chess*. In terms of modern chess variants, *Bad Chess* and *5D Chess (With Multiverse-Time-Travel)* are both very popular. The former being of the piece switching variant, *Bad Chess* generates a random set of pieces for each player to use, similar to *Fischer chess (a.k.a. 960Chess)*. The latter being of the board variant, adding dimensions to the board and creating an increasingly complex game with multiple vectors for checkmate, similar to what I aim to create. When we look at criticisms later, we will see where these two categories diverge. A third category of chess variants could be those that change basic rules, such as *Speed Chess* – whilst I plan to integrate optional rules

such as clocks and rewinding moves, I do not plan to alter them to such extremes like this, only to suit the 3D board. These suggestions prompted me into more research into chess variants, particularly those multidimensional few which I analyse above, as current systems, and also prompted me to consider the compensations that must be made for having a bigger, more complicated board. Later we will see how the arrangement of pieces, particularly pawns went through a few iterations.

#### Resulting Requirements:

2.6 - At the start of the game, set the board out as follows(Taking origin as (0,0,0):

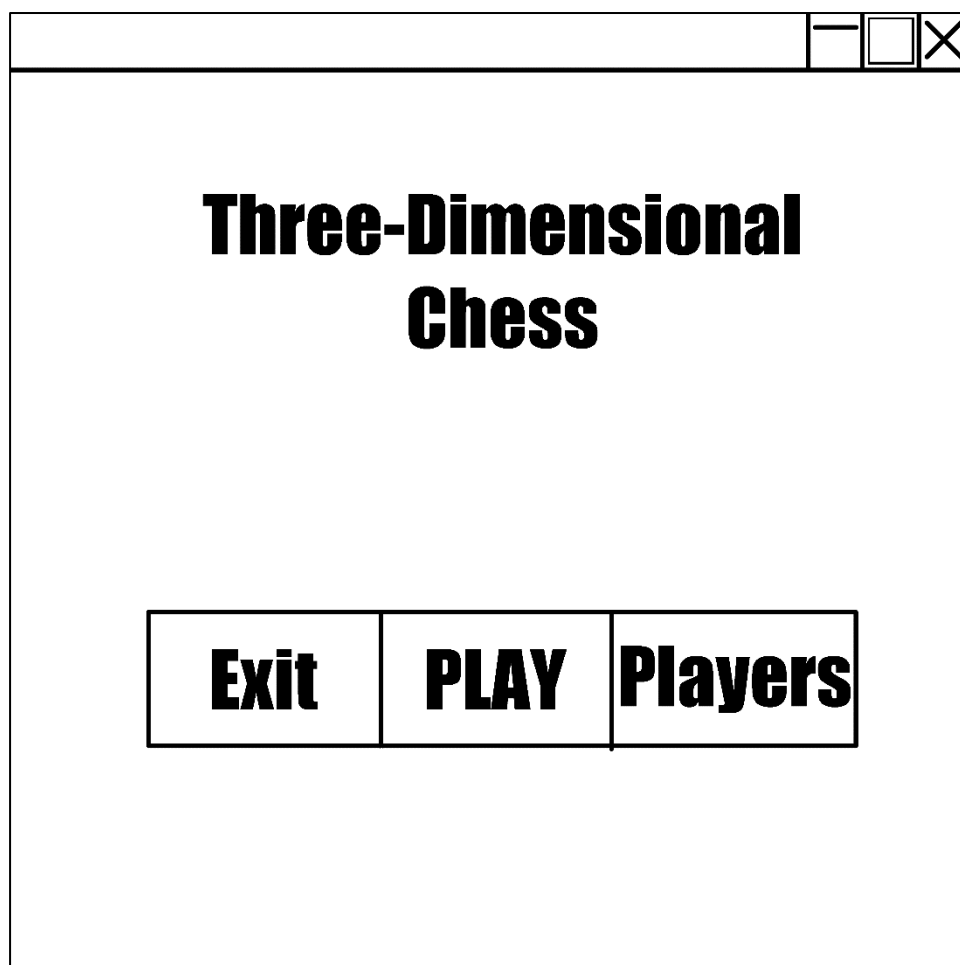
2.6a - White: Starting in the bottom left, closest corner (0,0,0). Place a white rook. To its right place a Knight (1,0,0), Bishop(2,0,0), Queen(3,0,0), King(4,0,0), Bishop(5,0,0), Knight(6,0,0), Rook(7,0,0)  
For the rows (x, 1, 0), (x, 1, 1): every square should be filled by a pawn

2.6b - Black: Starting in the top left, farthest corner (0,7,7). Place a black rook. To its right place a Knight(1,7,7), Bishop(2,7,7), Queen(3,7,7), King(4,7,7), Bishop(5,7,7), Knight(6,7,7), Rook(7,7,7)  
For the rows (x,7,6) and (x,6,7): every square should be filled by a black pawn

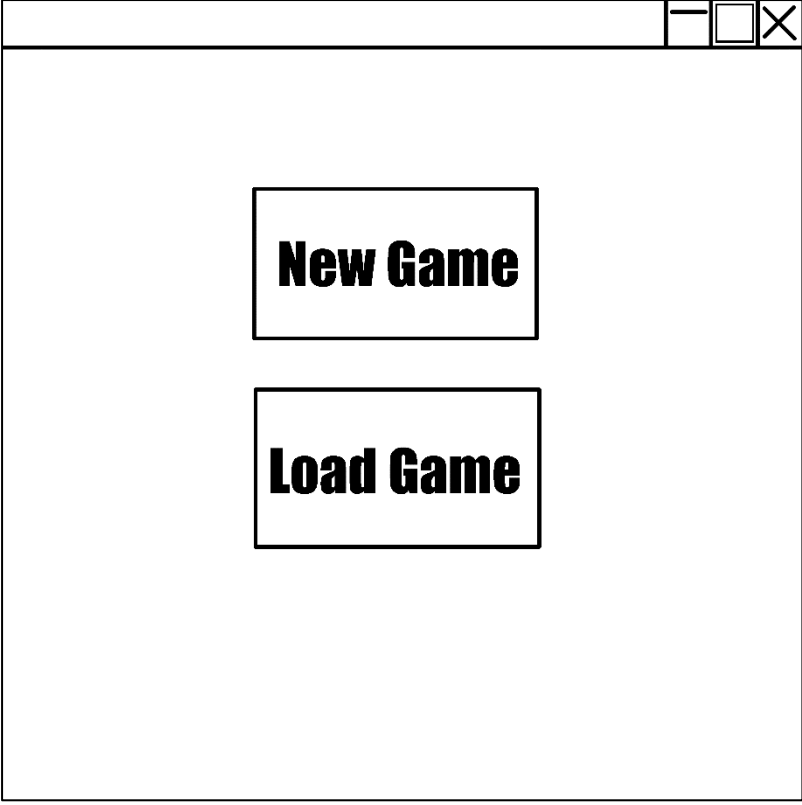
The results from this survey highlighted some important concerns over parts of the final product, such as how to interact with such a complex board as well as raising some requested features such as rankings and for the interface to be attractive.

#### UI Mock-up

Main Menu:

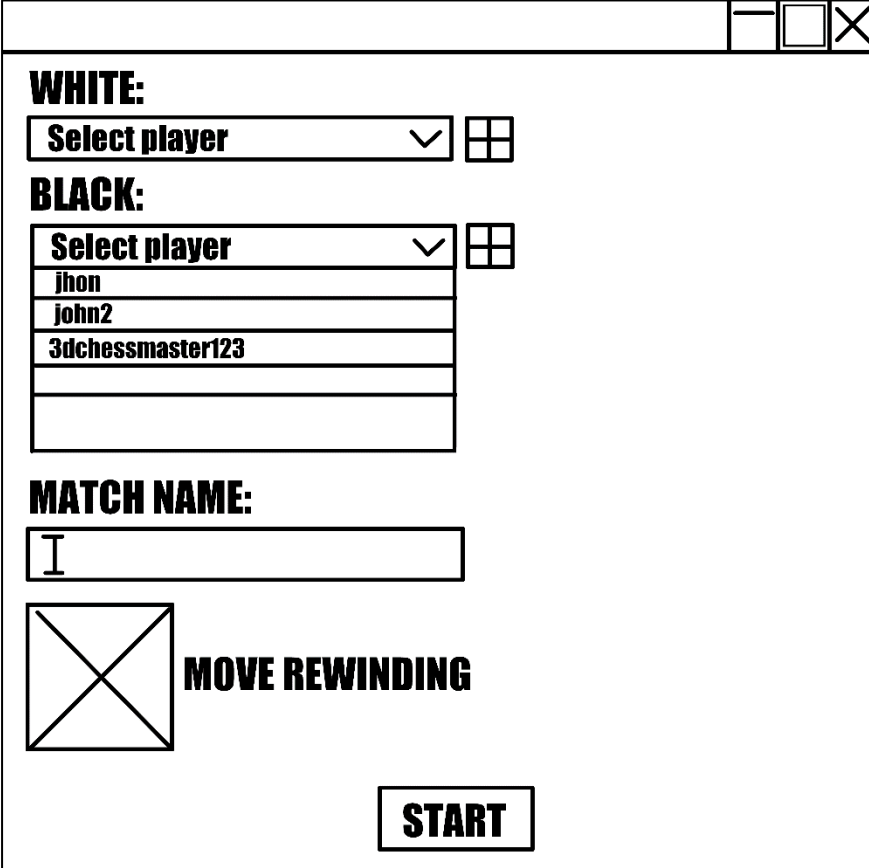


Play Menu:



A window titled "Play Menu" with a standard title bar (minimize, maximize, close buttons). The window contains two large, centered buttons: "New Game" and "Load Game".

New Game Menu:



A window titled "New Game Menu" with a standard title bar (minimize, maximize, close buttons). The window contains the following elements:

- WHITE:** A dropdown menu labeled "Select player" with a checkmark icon, followed by a 2x2 grid icon.
- BLACK:** A dropdown menu labeled "Select player" with a checkmark icon, followed by a 2x2 grid icon. Below the dropdown is a list of player names: "jhon", "john2", and "3dchessmaster123".
- MATCH NAME:** A text input field with a cursor.
- MOVE REWINDING:** A square button with a large 'X' icon.
- START:** A rectangular button.

New Player Pop-up Window:

—□×

**Enter new player name:**  
  
**CONFIRM**

Load Game Menu:

—□×

Name	Date	Outcome	White	Black
testgame	07/09	ongoing	jane	john
£5 bet	23/05	white win	3dchssmaster	jhon

**LOAD GAME****IMPORT FROM FILE**

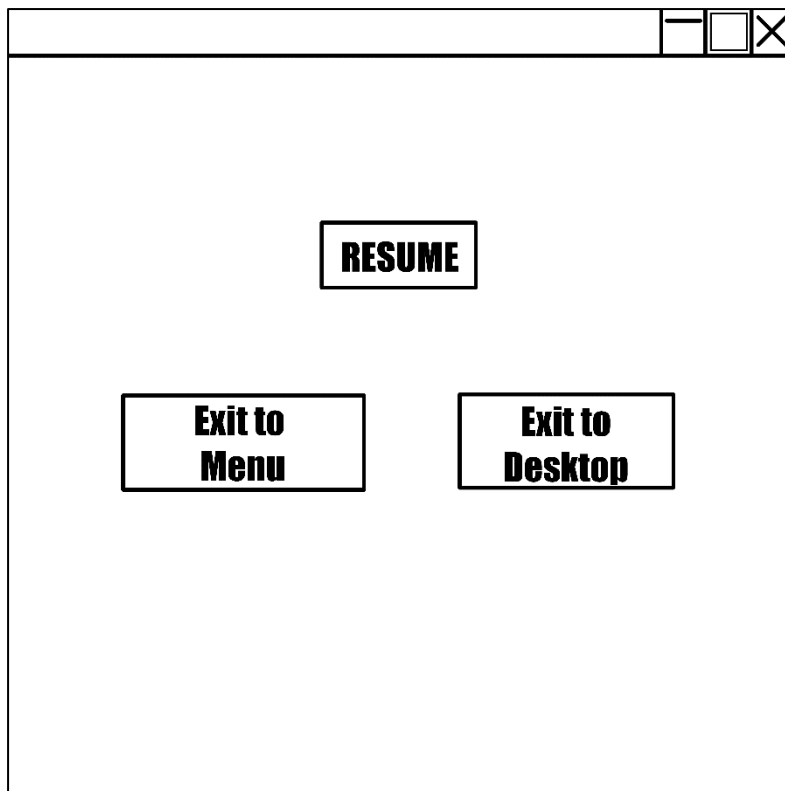
Confirm Game Menu:

[illegible]

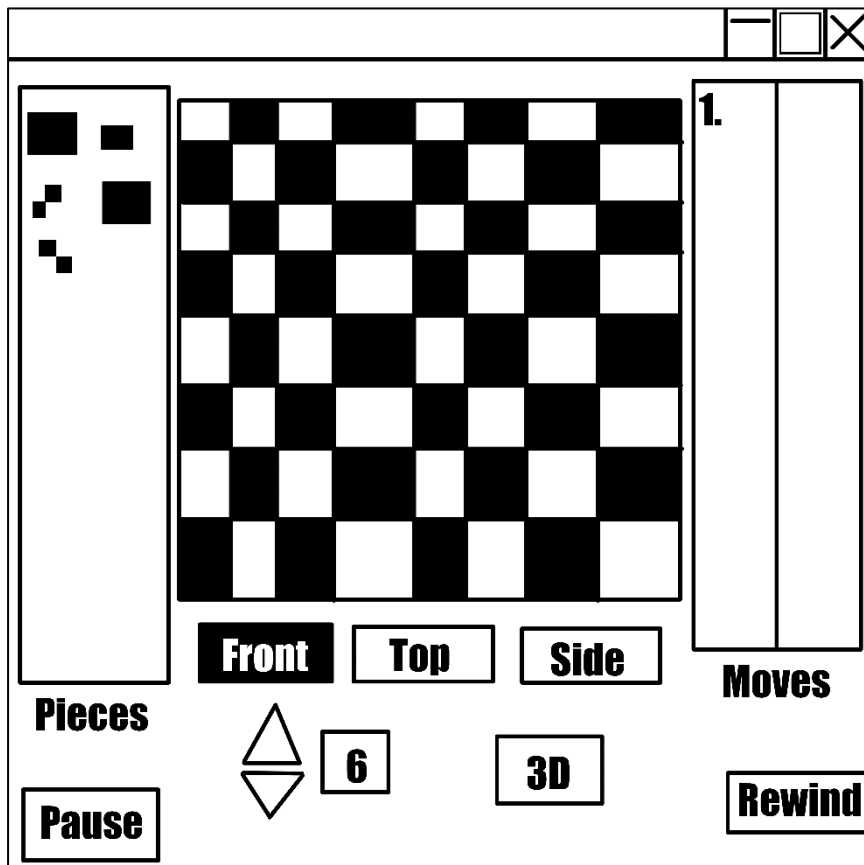
Players Menu (from Main):

[illegible]

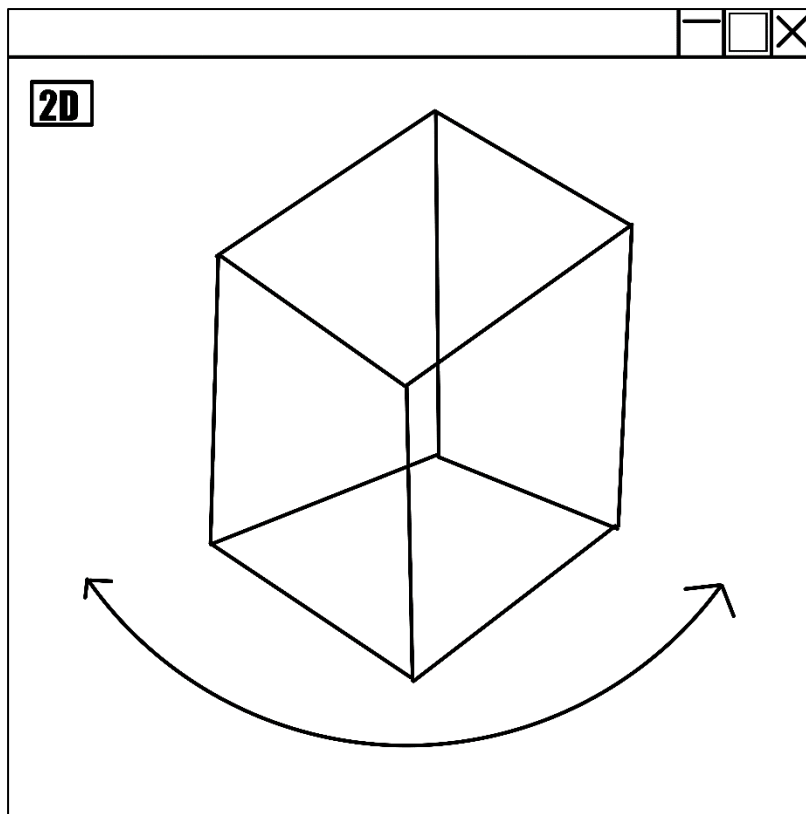
Pause Menu:



Game UI (2D/Moving View):



Game UI (3D/Overview):



### End User Feedback on UI Mock-up

This interview is also with Ciaran Brightley-Davies, following me producing the UI mock-up based on my research.

Q: "Do you think the games and players tables display sufficient information?"

"The table of players is a good substitute for online rankings, as long as you're able to sort it by the columns you've added. And I think the last accessed date for the Games table is very useful, just that extra contextual information to differentiate between games is a good idea: often when I play chess online they're simply ordered by most recent to least recent, so again, if you can sort by that attribute, it would be very useful."

**Resulting Requirements:**

7.12b - This list (of games) should be able to be sorted using navigation panels at the top of the menu element, this criteria should be sort by "Name", "Date", "Outcome", "White Player Name" and "Black Player Name"

7.21 - Using a navigation panel at the top of the table, the list (of players) can be sorted according to "Name", "Date Registered", "Overall Wins", "Winrate as Black", "Winrate as White"

Q: "Are there any more menu screens that you think would improve ease of access for the user?"

"Not a menu screen per se, but I think you should add the options to forfeit or agree to a draw from the pause menu. These are quite important in competitive chess and although this is a variant, you should probably still have the same functionality. Otherwise, I like the confirmation screen that shows you the last move made before you load a game, that's very useful and will save a lot of time."

**Resulting Requirements:**

7.34 - There should be two buttons, one labelled "White Forfeit" and one labelled "Black



Forfeit”

7.34a - Clicking on this button should take the user to a screen where they confirm their decision

7.34b - Once their decision has been confirmed, end the game and save the outcome depending on which player forfeited

7.35 - There should be another button, labelled “Agree to draw”

7.35a - Clicking on this button will take the user to a screen where they confirm their decisions

7.35b - After confirming their decision, end the game as a draw and save the outcome

Q: “If anything, what would you add?”

“I think, definitely make sure that the pieces’ possible moves are displayed on the 3D cube. It might also be an idea to show the boundaries of each cell or cube on the board to give an idea of scale, the distances between pieces and so forth.”

**Resulting Requirements:**

1.81 - Any piece selected on the 2D view will have its possible moves shown on the 3D representation

1.82 - Include a toggle to show the outlines of every square otherwise only draw outlines of possible move squares

## Requirements

### 1: Board & UI:

1.0 - Store an 8x8x8 three dimensional board (512 cells total)

1.01a - Adds squares from left to right, bottom to top, then near to far.

1.01b - Square 0 is black(A1), square one is white(B1): Each odd square is white (Alternate the starting square colour of each row)

1.1 - Ability to look at board from front(z coordinate is constant), side(x coordinate is constant), top(y coordinate is constant)

1.11a - User can view each layer of the board from any of the angles(front, side, top)

1.11b - Each layer is an 8x8 representation of the board

1.11c - User can move up or down a layer by a pair of opposite arrow buttons

1.2 - Pieces are represented on the board by icons, they are limited to one square

1.3 - When the user clicks on a friendly piece, squares that the piece could possibly move to should be tinted in blue (In accordance with piece moving rules and 2.03)

1.31a - If the user then click on one of the highlighted blue squares, the piece should be moved to that square

1.31b - The icon representing the piece should overwrite what is currently on the cell that is being moved to

1.31c - A sound should play when a piece is moved onto a square:

A hollow wooden sound for moving onto an empty square

A solid wooden sound for taking an opponent’s piece

1.4 - When the user clicks on an enemy piece, squares that the piece could possibly move to should be tinted in red(In accordance with piece moving rules and 2.03)

1.5 - There should be a list of past moves represented in a list from the most recent move to the first move

1.51 - The move representation should use chess notation, with the last 8 letters of the alphabet to represent the coordinates in the third dimension

1.6 - After each move, the square a piece started on and the square that a piece finished in, should be shaded in yellow

1.61 - Reset this for each move so only one pair of squares is shaded yellow at a time

1.7 - There should be a button which changes the UI to a screen showing a 3D representation of the board

1.71 - Any piece selected on the 2D view will have its possible moves shown on the 3D representation

1.72 - Include a toggle to show the outlines of every square otherwise only draw outlines of possible move squares

1.73 - Show the square moved to and from in yellow

1.74 - Have a button to return the view to the 2D representation of the board

## 2: Pieces:

2.0 - There should be a King piece

2.01 - The King may move one square in any direction, this includes diagonally by one square on one plane

2.02 - If a piece of the opposite colour may move onto the King's square (at the end of the opponent's turn) then the King is in check

2.02a - If the King is in check there are multiple resolutions:

The King may move to a square that is not threatened by an opponent's piece

The piece threatening the King can be taken, provided this doesn't put the King in check

A piece may move to block the line of cells towards the king, intercepting the line of check

2.03 - A player may not put their own King in check, either by moving the King or another of their pieces

2.04 - If the King cannot escape check by any of the listed resolutions, it is checkmate and the player of the trapped King has lost, the other player winning

2.05 - The UI notifies a player if their King is in check with a line of texts

2.1 - There should be a Pawn piece

2.11a - A white Pawn can move up the board or forwards on the z axis by one cell

2.11b - A black Pawn can move down the board or backwards on the z axis by one cell (from white's perspective)

2.12a - It may not move over another piece, it must stop on the cell before it

2.12b - It may not finish its move on the same square as a piece of the same colour

2.13 - A Pawn can take an enemy piece by moving one cell diagonally (on one plane) and finishing its turn on the cell of the piece of the opposite colour

2.13a - A Pawn can only move diagonally to take a piece

2.13b - This move must still be towards the promotion rank, so white pawns cannot capture diagonally one deep and one down; black pawns cannot capture diagonally one deep and one up.

2.14 - If a Pawn reaches the end of the board, row **8z** for any white pawn and row **1s** for any black pawn, it is promoted. The player chooses any piece other than a King or Pawn for it to become.

2.2 - There should be a Rook piece

2.21a- The Rook can move any number of cells in a straight line (parallel with the x, y or z axis)

2.21b - It may take a piece of the opposite colour by finishing its move on that piece's

square

2.21c - It may not move over pieces, it can only move up to that piece

2.21d - It may not finish its turn on the same square as a piece of the same colour

2.21e - It may not move over the edges of the board, it can only move up to the edge of the board

### 2.3- There should be a Bishop piece

2.31a - The Bishop can move any number of cells diagonally on one plane

2.31b - It may take a piece of the opposite colour by finishing its move on that piece's square

2.31c - It may not move over pieces, it can only move up to that piece

2.31d - It may not finish its turn on the same square as a piece of the same colour

2.31e - It may not move over the edges of the board, it can only move up to the edge of the board

### 2.4 - There should be a Knight piece

2.41a - The Knight can move two cells on one axis and then one cell on a different axis

2.41b - It may move any direction on this axis, for example, up or down on Y-Axis

2.41c - It may take a piece of the opposite colour by finishing its move on that piece's square

2.41d - It may move over pieces of any colour

2.41e - It may not end its move on a piece of the same colour

2.41f - It may not move over the edges of the board, moves that end off the board are not valid

### 2.5 - There should be a Queen piece

2.51a- The Queen may move as a combination of the Rook and the Bishop

2.21a- The Queen can move any number of cells in a straight line (parallel with the x, y or z axis)

2.31- The Queen can move any number of cells diagonally on one plane

Note: See 2.21 and 2.31 for full moving rules of relevant pieces

2.51b - It may take a piece of the opposite colour by finishing its move on that piece's square

2.51c - It may not move over pieces of any colour

2.51d - It may not end its move on a piece of the same colour

2.51e - It may not move over the edges of the board, moves that end off the board are not valid

### 2.6 - At the start of the game, set the board out as follows(Taking origin as (0,0,0):

2.6a - White: Starting in the bottom left, closest corner (0,0,0). Place a white rook. To its right place a Knight (1,0,0), Bishop(2,0,0), Queen(3,0,0), King(4,0,0), Bishop(5,0,0), Knight(6,0,0), Rook(7,0,0)  
For the rows (x, 1, 0), (x, 1, 1): every square should be filled by a pawn

2.6b - Black: Starting in the top left, farthest corner (0,7,7). Place a black rook. To its right place a Knight(1,7,7), Bishop(2,7,7), Queen(3,7,7), King(4,7,7), Bishop(5,7,7), Knight(6,7,7), Rook(7,7,7)  
For the rows (x,7,6) and (x,6,7): every square should be filled by a black pawn so that the non-pawn pieces are completely encapsulated by pawns.

## 3: Game Rules:

3.0 - White always moves first

3.1 - As per 2.02, check must always be resolved

3.2 - If check cannot be resolved, the player who cannot move their King and it is under threat has lost and the other player has won

- 3.21 - This is also the end of the game and no more moves can be made after it
- 3.3 - If a player cannot move on their turn and their King is not in check, it is a stalemate, a.k.a. a draw. This is the end of the game and no more moves can be made after it.
- 3.4 - When the game ends it should automatically be saved to the database

#### 4: Game and Player Storage:

- 4.0 - A game can be saved into a database
  - 4.01 - When a game is saved it should include the move history
  - 4.02 - Saved games can be loaded from the Load Game menu (See 7.12)
    - 4.02a - The pieces positions are loaded and applied to the board
    - 4.02b - The move history is loaded into the Move History List in the UI
  - 4.03 - Any extra optional rules should be stored and subsequently loaded when the game is loaded (See 6: Optional Rules for optional rules)
  - 4.04 - Which players took part in the game and which colour each player was
  - 4.05 - The outcome of the game, this will be "White Win", "Black Win", "Stalemate" or "Ongoing"
  - 4.06 - The date the game begins should also be recorded
- 4.1 - Players can be stored in a database
  - 4.11 - The number of games won, lost and drawn by each player should be recorded
    - 4.11a - The total number of games played should be recorded
    - 4.11b - Record the number of games won as white and the number of games won as black
    - 4.11c - Record the number of games played as each colour
  - 4.12 - The name of the player should be recorded
  - 4.13 - The date the player is registered should be recorded
- 4.2 - At the end of each turn the current state of the game should be saved to the database

#### 5: Players:

- 5.0 - Chess has two players, one controls the white pieces and the other controls the black pieces
- 5.1 - The game must keep track of who's turn it is
  - 5.11 - Players can only move one piece on their turn and then it is the other player's turn
  - 5.12 - The UI should display who's turn it is
- 5.2 - When the game is started each player may select a player profile from the list, fetched from 4.1
  - 5.21 - The player should have the option to make a new player profile

#### 6: Optional Rules:

- 6.0 - Before starting a new game, the players will be asked whether they want to use any optional rules
- 6.1 - Optional Rule 1: Players can rewind the moves of the game
  - 6.11 - Next to the list of moves in the UI (See 1.7) there should be a rewind button
  - 6.12 - This button should undo the last move, returning the board to the state it was in before the last move was made
  - 6.13 - Any moves made after having rewinded should overwrite the move list and delete any entries later than it, to avoid creating problems when exporting/importing games

#### 7: Menus:

- 7.0 - Upon starting the program the user should be faced with a main menu, composed of large, simple buttons
  - 7.01 - The first option on this menu should be "Play" - This will take the user to another menu, detailed in 7.1

- 7.02 - The second option on this menu should be "Players" - This will take the user to another menu, detailed in 7.2
- 7.03 - The third and final option on this menu should be "Exit" - This will close the program
- 7.1 - The "Play" menu should have two large, simple buttons, those being "New Game" and "Load Game" which will take the user to their respective menus (detailed in 7.11 and 7.12)
- 7.11 - The "New Game" menu will have multiple buttons (Please see the UI Mockup for arrangement and size)
  - 7.11a - On the left, there should be a dropdown list of established player profile for the White player to select from
  - 7.11b - Next to the dropdown menu, have a button to create a new player profile which opens a popup to go through it
  - 7.11c - Have the same options mirrored for black on the other side of the UI
  - 7.11e - Have a tickbox for whether to use Rewinds (Optional Rule 1; See 6.1)
  - 7.11f - Have an entry box at the top to name the match, otherwise generate a random string
  - 7.11g - Have a large, simple "Start" button at the centre base of the UI, which starts the game
- 7.12 - The "Load Game" menu will have multiple buttons (Please see the UI Mockup for arrangement and size)
  - 7.12a - This menu should be mainly composed of a large list of games stored in the database (see 4.0) displayed as a table
  - 7.12b - This list should be able to be sorted using navigation panels at the top of the menu element, this criteria should be sort by "Name", "Date", "Outcome", "White Player Name" and "Black Player Name"
  - 7.12d - Upon selecting a game to open, the user should be taken to intermediary screen where it shows the game settings and current move. The user can then confirm this is the game they intend to load and upon clicking on a large, simple "Start" button the game is loaded
  - 7.12e - If a completed game is loaded, automatically enable rewind moves and change the name so that if it's saved again, it isn't saved back onto the completed game but onto a new position in the database
- 7.2 - The "Players" menu should show a list of all players displayed as a table
  - 7.21 - Using a navigation panel at the top of the table, the list can be sorted according to "Name", "Date Registered", "Overall Wins", "Winrate as Black", "Winrate as White"
  - 7.22 - There should be a button that takes the user to a new player creation screen, labelled accordingly
- 7.3 - Whilst in a game, there should be a menu icon on the bottom left corner of the UI, clicking this element will pause the game. Making other elements uninteractable and displaying a menu with large simple buttons.
  - 7.31 - The first button should be a "Resume" button which returns the game to its unpaused state
  - 7.32 - There should be another button, labelled "Quit to menu", which saves the game and returns the user to the Main Menu
  - 7.33 - There should be another button, labelled "Quit to desktop", which saves the game and closes the program
  - 7.34 - There should be two buttons, one labelled "White Forfeit" and one labelled "Black Forfeit"
    - 7.34a - Clicking on this button should take the user to a screen where they confirm

their decision

7.34b - Once their decision has been confirmed, end the game and save the outcome depending on which player forfeited

7.35 - There should be another button, labelled "Agree to draw"

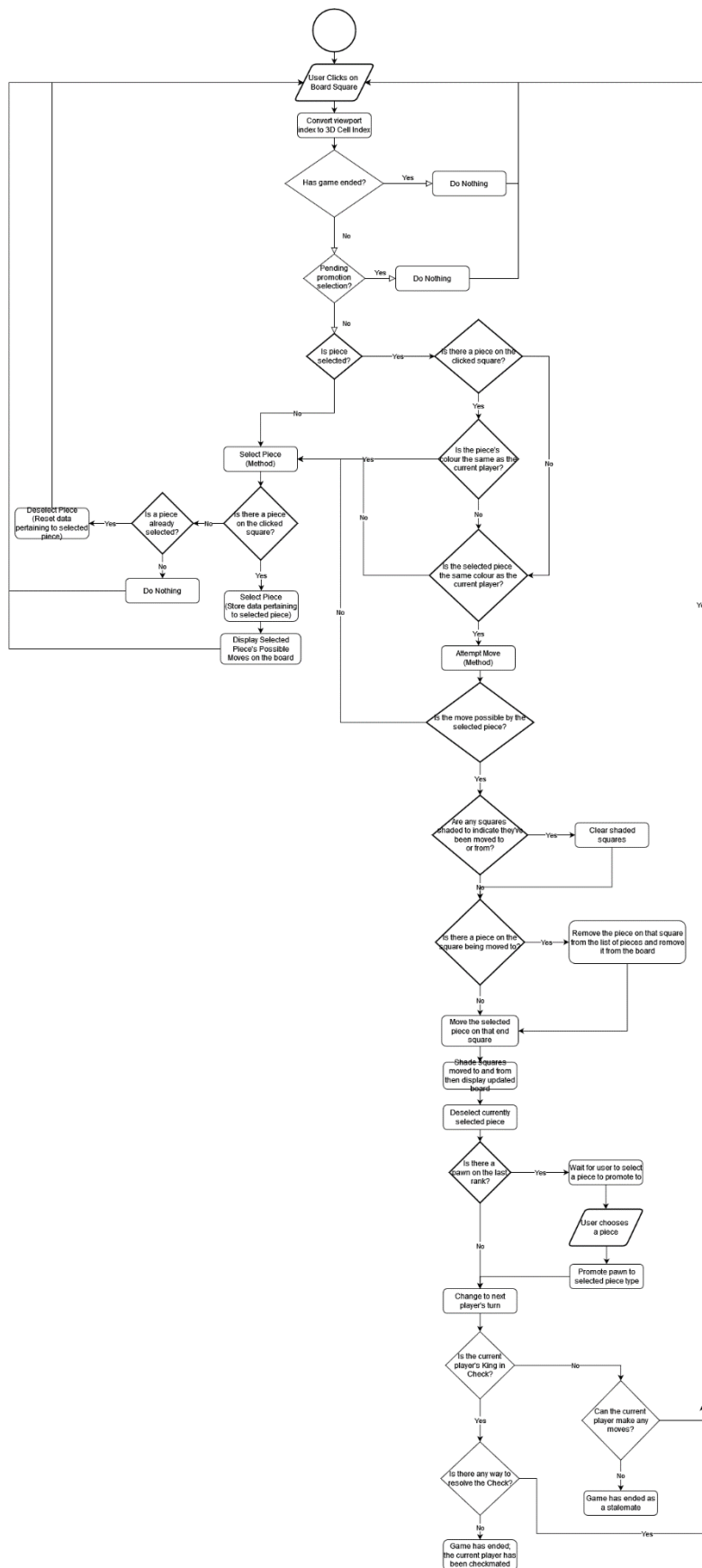
7.35a - Clicking on this button will take the user to a screen where they confirm their decisions

7.35b - After confirming their decision, end the game as a draw and save the outcome

## Design

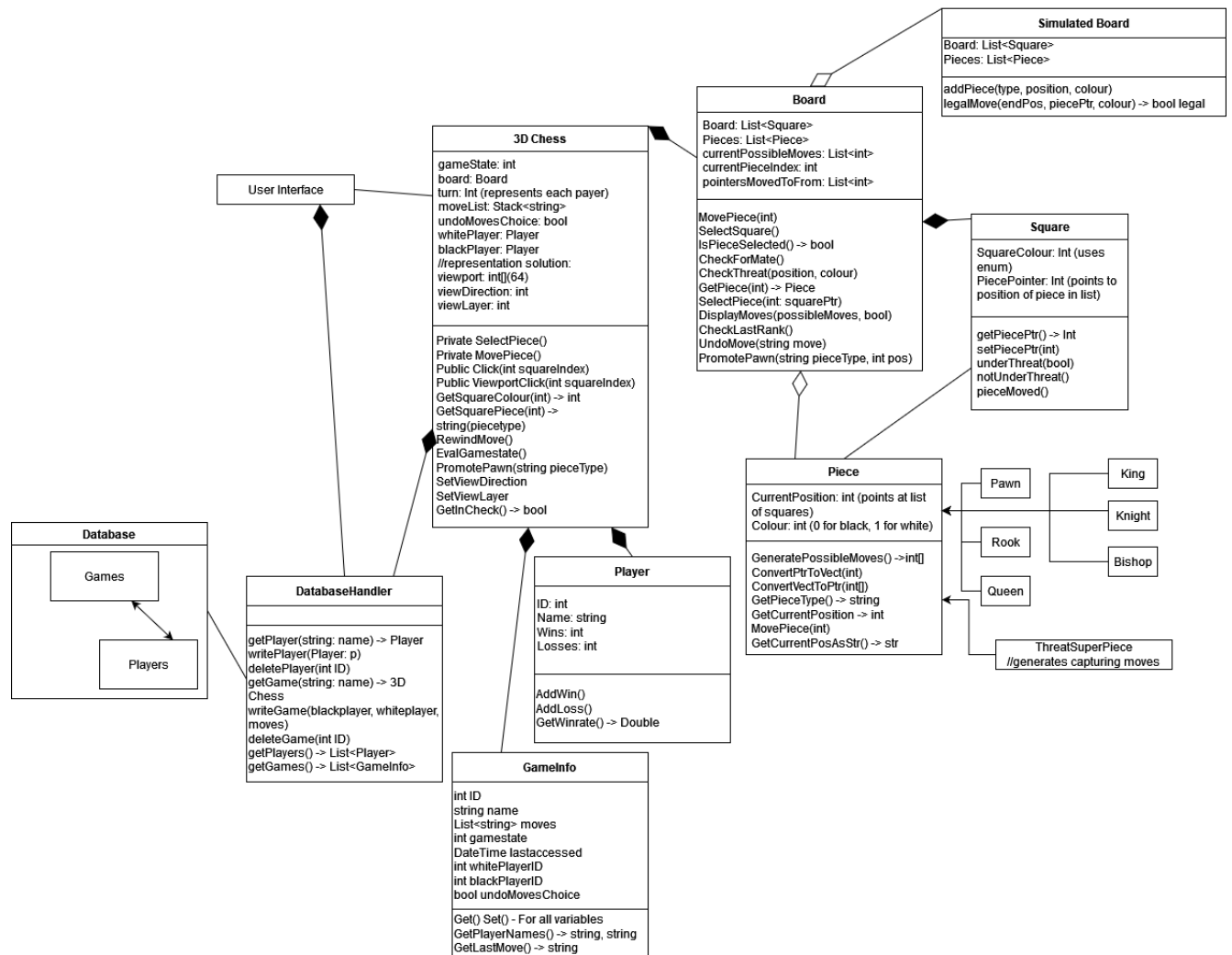
### Flowchart

This flowchart will help us understand how the main loop of the game and any conditions that influence the state of the program, starting from the beginning of the game.



## Class Diagram



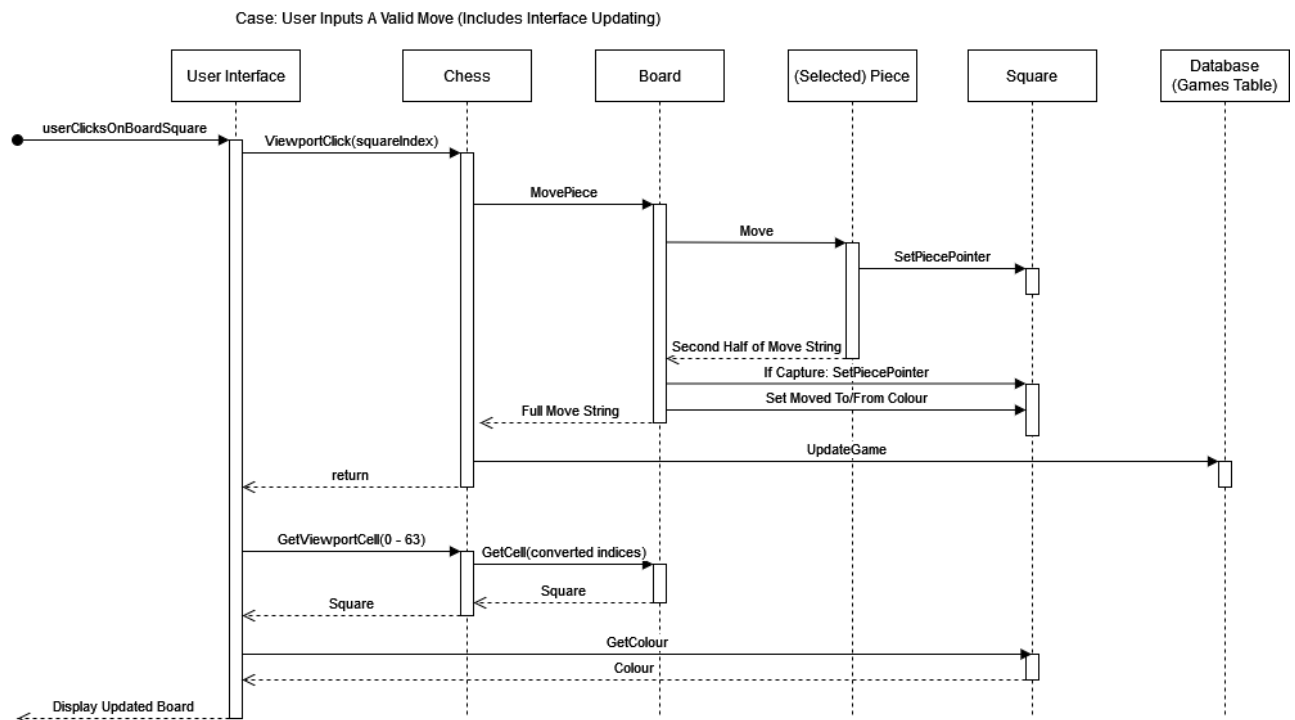


Simulated Board: (explain here)

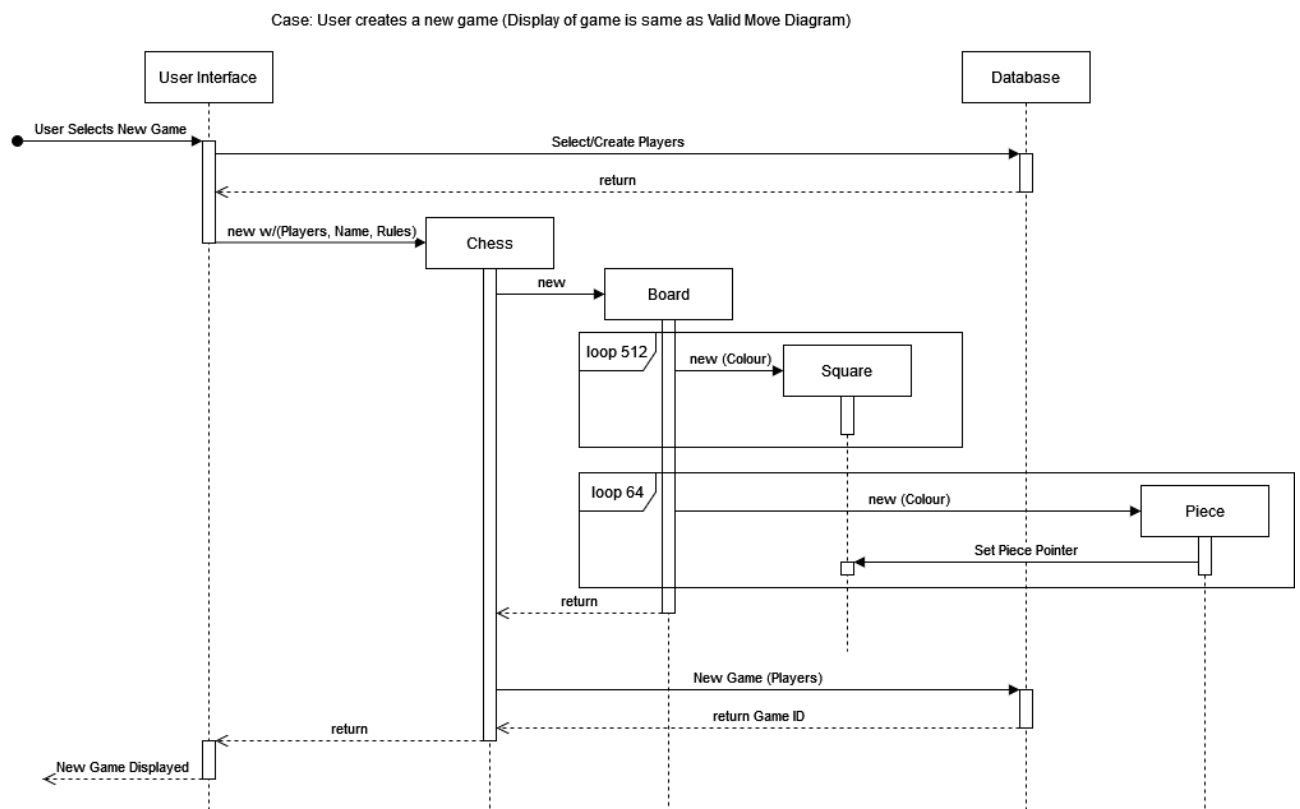
Threat Super Piece: (also explain)

## Sequence Diagrams

This diagram shows the path through the program if a valid move is input, this is the same for capturing and not-capturing moves, as the adjusting of the pointer in SetPiecePointer overwrites. The Board then uses the First Half of Move String to see whether other pointers need to be adjusted.

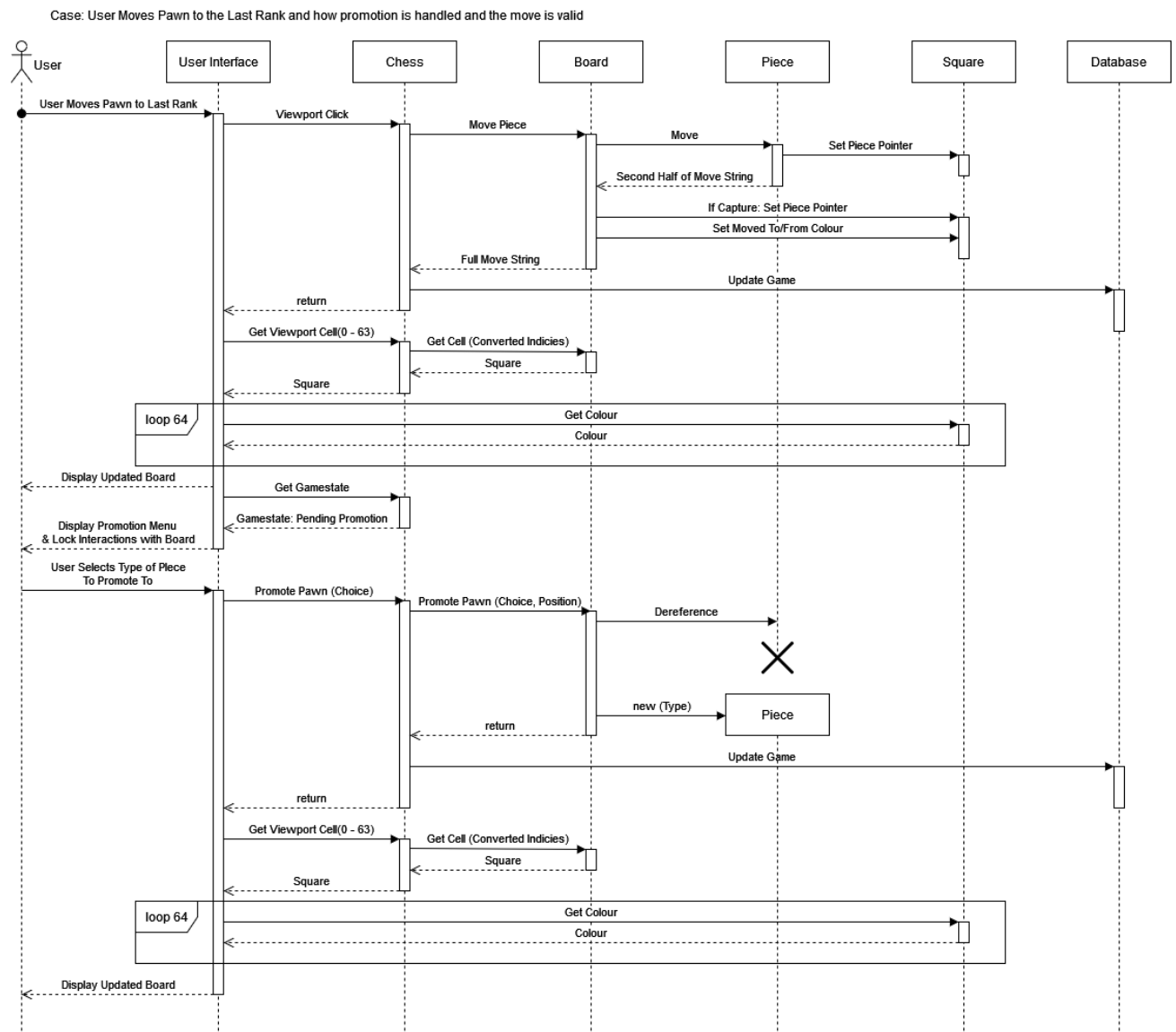


The process of creating a new game via the menus, how the objects are subsequently initialised and with what information.

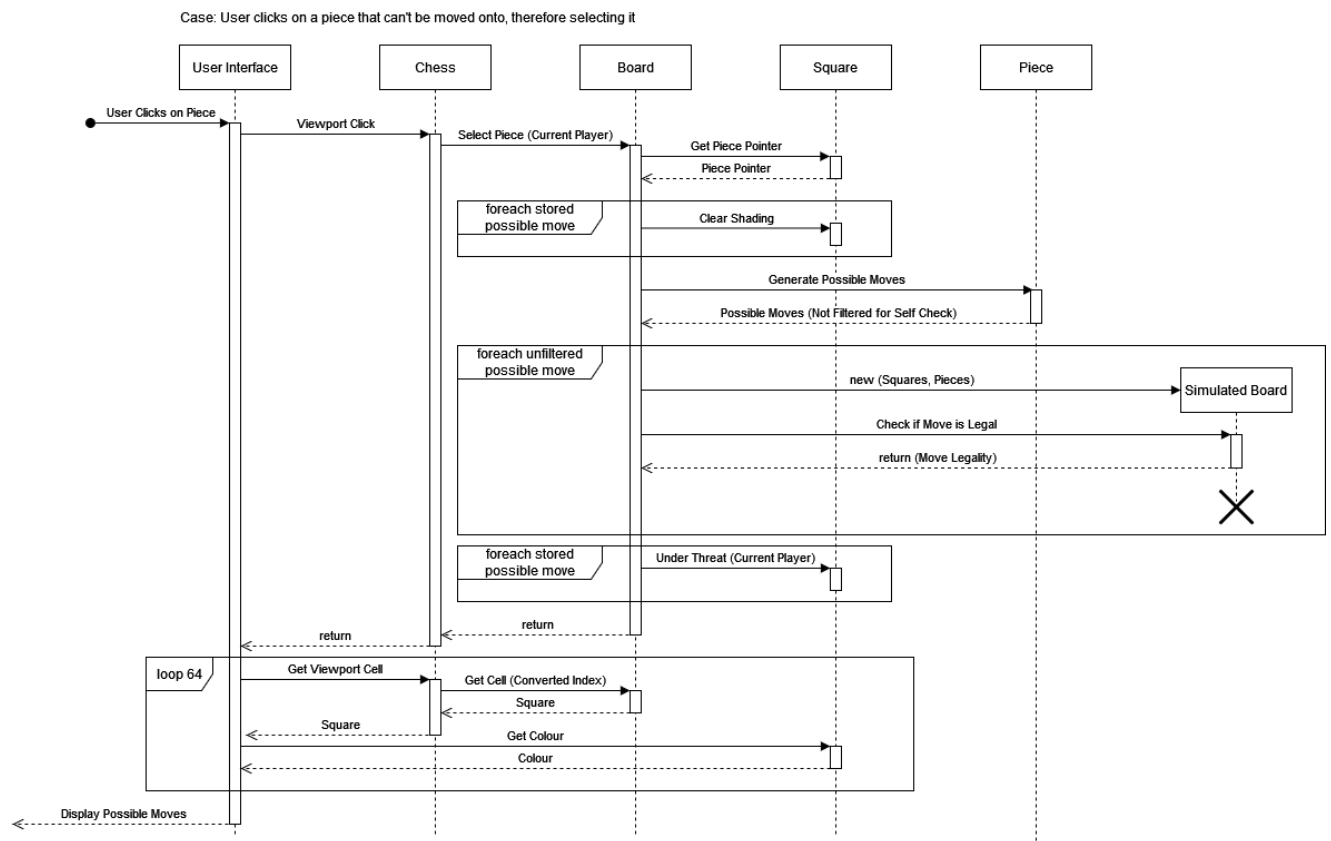


This next diagram shows the path through the code when a pawn is moved to the last rank and thus must be promoted before the next move. The best way to handle this is by not taking input on the board whilst a promotion is pending. The Chess object can store the position of the piece to be

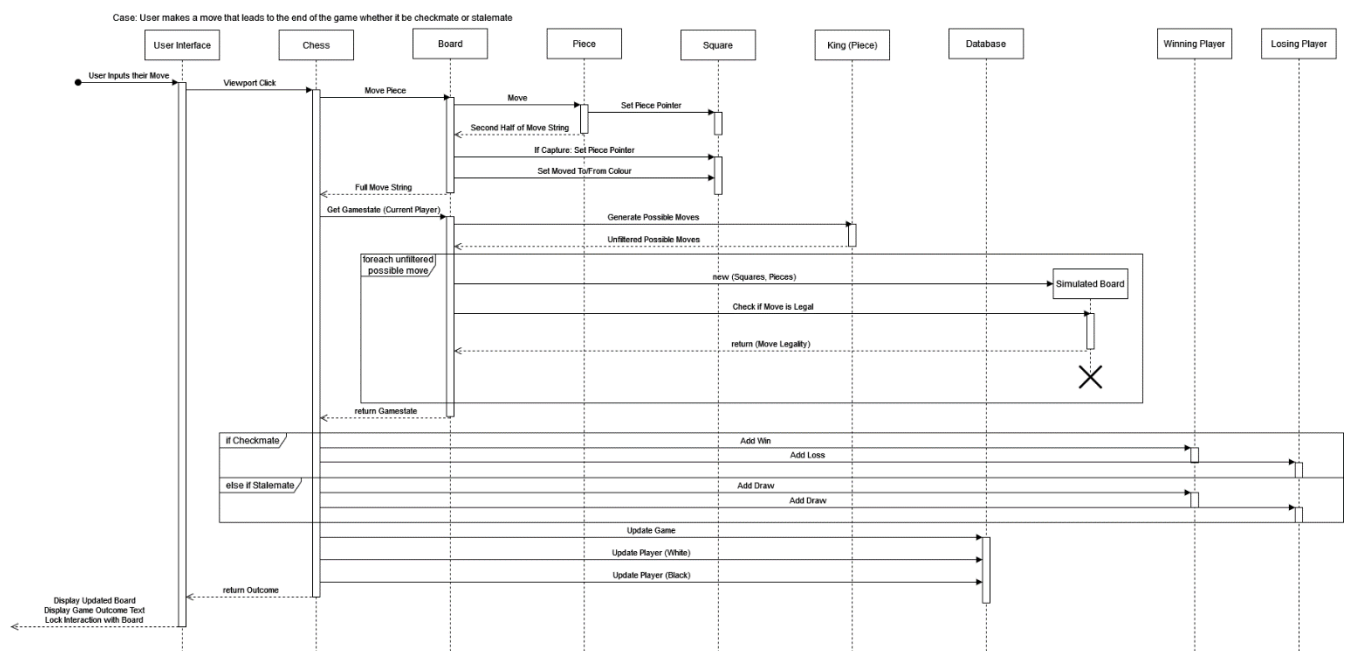
promoted, there will only ever be one piece pending promotion due to the game rules and the fact that we freeze other interactions whilst waiting for the promotion input.



This next diagram showcases the messages between objects when a piece is selected. This is useful to understand how moves are filtered when generated. Physically possible moves are returned by the piece and then the board filters out self-check moves with the simulated board due to having access to more information.



This diagram shows the extra steps that are taken when a move made by the user ends the game. It also gives us a small insight into the interactions with the database. While these diagrams show how the objects interact with each other, to understand the conditions that lead to this outcome it is best to look at the above flowchart.

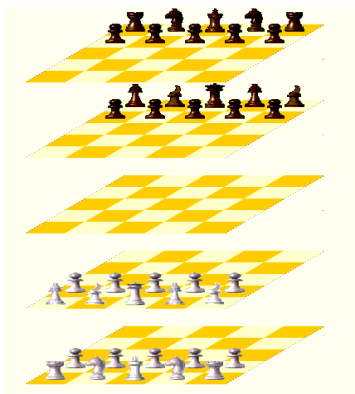


## Ruleset

In this section I will be talking about designing the rules used for the game and sometimes how this influences the program design.

### Starting position

It was hard to find starting positions for a game of *Kubikschach* however starting positions for *Raumschach* were readily available. By noting the important features of these positions, we can come to starting positions for Three-Dimensional Chess. The important thing to note is the double row of pawns, one row of which blocks the diagonal. Here, the other pieces are stacked in a double row as well, because our board is 8x8x8 that is unnecessary for us. By combining these two facts, we should have what looks like a normal chess starting position from our front view and then another row of pawns transformed by 1 on the z-axis (from the standard pawn starting position) to block the diagonal. The same goes for black, which has the extra effect of meaning that the unguarded row is in an opposite direction for each colour.



### Algorithms

#### Storing a 3D Board

Something very important to consider for this project was how the board shall be stored. I could've used a three-dimensional array but instead I opted to use a one-dimensional array and then use calculations with powers of 8 to access the correct cells. To illustrate this concept, here's the pseudocode for converting a 3D coordinate to an index for the 512 Length array of cells:

```
coordinate[] <- { x, y, z }
squarePtr <- 0
squarePtr += coordinate[0]
squarePtr += coordinate[1] * 8
squarePtr += coordinate[2] * 64
return squarePtr
```

#### Representing a 3D board in 2D

To build the UI I opted to use a lightweight graphic engine called Raylib, which is written in C but has bindings for C# so it was easy to use. I chose this library due to it being easy to switch between a 2D and 3D representation of the board. Furthermore, it was made it easy to take input on the board, as it includes methods to get the position of the cursor and where it has clicked, thereby making it easy to calculate where on the board the user has clicked.

One downside of using this library was that there aren't methods or objects for buttons, such as in Windows Forms or WPF, therefore I had to check whether every click was in the bounds of each button.

When representing the board in 3D it is quite simple to display, we can iterate through every Square object in the board and then draw a cube in its position, calculated from its index, in the window – with any shading or a piece graphic if it has a piece on it.

Representing the board in 2D is more complicated, as we will not be able to see the whole board at once. The way I chose to approach this is by displaying an 8x8 slice of the board to the user, using a “viewport” construction. The game class holds an array of indices for these 64 cells and when drawing the board, we iterate through these and draw 2D squares in the window – complete with shading and pieces. The user can then look at the board from the top or side and step through the layers of the board. Here’s some pseudocode to demonstrate how those pointers are calculated:

```
originPointer <- 0
if viewDirection == Front:
    originPointer = boardLayer * 64
    for y = 0 to 8:
        for x = 0 to 8:
            viewport[(8 * y) + x] = originPointer + (x * 1) + (y * 8)
elif viewDirection == Side:
    originPointer = boardLayer * 1
    for y = 0 to 8:
        for x = 0 to 8:
            viewport[(8 * y) + x] = originPointer + (x * 64) + (y * 8)
elif viewDirection == Top:
    originPointer = boardLayer * 8
    for y = 0 to 8:
        for x = 0 to 8:
            viewport[(8 * y) + x] = originPointer + (x * 1) + (y * 64)
```

### Calculating Checkmate and Check

Calculating checkmate can be done in a couple of steps. The first prerequisite is understanding the method for finding whether a particular square is under threat – this can be used to see if the King is in Check. The way that I will do this is by first registering the colour of the King on the target square. Then we will generate moves for each type of piece (except pawns and kings) from that square. For example, from the King’s occupied square we generate the moves of a rook, then if any of those moves have a rook of the opponent’s colour on it, we know that rook is threatening the space which the King is on. This can also be applied for bishops, queens and knights. Kings cannot move into the threat zone of another King, thankfully this kind of move should be eliminated earlier generation and therefore made impossible – so we don’t have to check for this. For pawn’s we need to calculate the inverse of their capturing moves as they are not identical in both directions, like with other pieces.

If the King is not in check, we don’t have to look any further into this since checkmate is impossible. However, if the King is in check there are a few more things to consider. The simplest thing to first consider is if the piece threatening the King can be captured – we can use the same method that we used to calculate Check for this. For each move that could possibly capture threatening piece, we’ll need to check that this doesn’t also place the King in Check, which would make it an illegal move (see Move Generation: General Cases).

If the piece cannot be captured, we should now see if it can be blocked, this is more complex than seeing if the piece can be captured but works very similarly. This option to block Check is only available for Rooks, Bishops and Queens so we can ignore this if the threatening piece is a knight. If it is one of these pieces, we calculate the line of moves inbetween the piece and the King and then check if each of those squares are “under threat” - as if it was a piece. If any of these move-lines can be blocked, then it is not checkmate. In the same way as capturing the threatening piece, we’ll need to check that blocking it isn’t a self-Check. This can be done simply by generating and filtering the moves for all a player’s pieces.

If none of these moves are available, then it is checkmate and it’s likely easier to check which player is the loser as their King will be on the checkmated square.

### Calculating Stalemate

Stalemate is extremely unlikely with such a large board of highly mobile pieces, however it should be considered as a potential end of the game. I considered a few methods to reduce the load of this algorithm such as only checking for stalemate once a certain *low* number of pieces has been reached or only a certain number of moves in; however, these ideas just added more conditions into the algorithm. For simplicity and assuming that the computational cost of calculating the stalemate isn’t very high, these ideas will not be implemented.

To calculate stalemate definitively there is only one clear option, to see if any move is possible. To streamline this, when initialising the board object and placing pieces the order in which they are added to the collection of pieces should be in order of move generation complexity (this being Pawn, Knight, Rook, Bishop, King, Queen). Then, when calculating stalemate, we can iterate through this list in the order of complexity and generate the moves of each piece – when a piece returns any moves we can quit this subroutine, having deemed that the game is not in a state of stalemate.

### Combining checking for Checkmate and Stalemate

Since the checks we’re performing for checkmate and stalemate are so similar, it’s efficient to combine them. This can be done by reversing some of the logic for the checkmate algorithm. First, we iterate through the pieces of one player and attempt to generate moves for each of them, including eliminating self-check moves. We can exit as soon as we return any moves, which preserves the average efficiency of this calculation.

Then, if we weren’t able to generate any moves, meaning the current player can’t move, we’ve reached one of two outcomes. If the current player’s King is under threat (in check) then, they have been checkmated and have lost. Otherwise, if the King is not under threat but has no moves regardless, it is a stalemate.

Here’s an example of this algorithm in pseudocode:

EvaluateGamestate(currentPlayer):

foreach piece in Pieces:

    unfilteredMoves <- piece.GeneratePossibleMoves()

    filteredMoves <- Empty List

    foreach move in unfilteredMoves:

        simulatedBoard <- new SimulatedBoard(currentBoardData)

        legal <- simulatedBoard.IsMoveLegal(move)

        if legal: filteredMoves.Append(move)



```

king <- GetPiece(currentPlayer)
If filterMoves.Length > 0:

    tsp <- new ThreatSuperPiece(position=king.position, colour=king.colour)
    threateningMoves <- tsp.GeneratePossibleMoves()
    If threateningMoves.Length > 0:
        # this means king is under threat
        return Checkmate (current player has lost)

    Else:

        return Stalemate
    EndIf
Else:

    return Game is ongoing

EndIf
EndFunc

```

### Move Generation: General Cases

When generating moves, the same constraints will always be applied to these possible moves so we can talk about the checks generally.

1. Pieces can't move off the edges of the board, so when generating moves we want to handle them in coordinate form so that we can check they are between 1 and 8, meaning that the piece has not gone off the board.
2. Pieces can't finish their turn on the same square as a piece of the same colour as themselves therefore when generating moves we need to check if there is a piece already on that square. If the piece is the same colour, the proposed move is invalid. If the piece is a different colour, it is a valid move and would also be a capture. The way we can check this is by passing a reference to the list of pieces and using the piece references stored in each Square to check the colour of the piece on a square if there is one.
3. Pieces can't move in a way that would place their own King in check. To check for this, we need to simulate every move that is generated and then check if the King is under threat. This is most simply done in the board class which has access to all the piece and square information. The board creates a skeleton copy of itself which enacts the move and then returns whether it's valid by checking for Check or Checkmate as detailed above.
  - a. In C# objects are passed by reference, therefore when creating the skeleton board, we must pass the pieces and squares and then copy the information from them, otherwise we would simply copy the references to the pieces and influence the actual game by checking for Check.

### Pieces and Polymorphism

All pieces inherit from a Piece class, this is useful as there are some functions that all piece perform the same. Converting between different coordinate formats and the actual operation of moving a piece are all performed the same way. Furthermore, this makes it simple to store all the pieces in a collection together as the collection can be defined as containing Piece objects. The polymorphic methods of the piece types are the generate move function, as each type of piece can move in a

different way, by defining this as an abstract method, all instances of inheritance from Piece must implement this method.

#### Generation of Moves: Bishops, Rooks, Queens and Kings

Bishops, Rooks and Queens are unique as a group of pieces because they can move any number of squares in one direction on their turn. Due to this property, I chose to generate the moves of these pieces by recursion. This recursion has multiple exit cases, it stops upon reaching the edge of the board or a friendly piece which blocks the way.

The Queen is a composite of a Bishop and a Rook, thus in its Generate Possible Moves method it calls upon the Generate Moves methods of those two pieces. This means that the Bishop and Rook have an interface to directly change their position, rather than only being able to move through the normal move method. The Queen still implements the normal Move Piece method but overrides it (it being virtual) to also update the position of these internal pieces.

The King can make similar moves but only one cell in each direction. So rather than implementing internal pieces, like the Queen, I chose to adapt the methods from the Bishop and Rook so that they only run once, it's as simple as removing the recursive calls.

#### Generation of Moves: Knights

One of the useful qualities of the Knight is its finite number of moves. With this I considered a few avenues for generating these possible moves. The method I settled on in the end was very simple and likely more efficient than those I researched. Of the methods I researched one stood out to me as interesting, by simulating the size of the board, one could plot a sphere with radius of the magnitude of a Knight's move (which is always the same) and any points where the surface of this sphere intersects with the centre of a square is a valid move. Despite this intriguing geometric solution, I opted for the simpler listing of possible moves, in parallel arrays of x, y and z transformations.

Then the piece iterates and reconstructs each move by using the value in each array to transform the current position of the piece before the standard checks are applied.

#### Generation of Moves: Pawns

Pawns in three dimensions are quite different, because the row of cells for promotion is at the opposite corner of the board: the pawn can move deeper or vertically towards their promotion rank. This means a pawn can only generate two non-capturing moves, because this is very simple we can check each case individually. For a black pawn for example, we check whether it is against the front face ( $z = 0$ ) or the bottom face ( $y = 0$ ) and eliminate the move that would move off the board in that direction. Because these are non-capturing, there can't be any piece on the target square either.

Pawn captures are slightly harder to generate, we inspect these separately from non-capturing moves because the vector of the move that a pawn makes when capturing is different to the capturing move. In 2D chess, pawns can only take diagonally, this is true in 3D chess and we apply our idea of a piece only being able to move on one plane at a time. Like with non-capturing moves, we check whether the pawn is against a face and quickly eliminate that possible capture. If the pawn could move diagonally to take, we generated the pointer for that square and see if there is a piece of the opposite colour on it, if so it is valid.

This is where polymorphism is very useful because the pawn, rather than checking moves after generating them, checks them before generating them due to the finite and unique nature of having

capturing and non-capturing moves. Furthermore, the pawn moves differently depending on what colour it is, rendering the polymorphism even more crucial.

### Data Structure Design

Stack -> Storing moves with modified LAN which can be parsed in both directions to allow for rewinding of moves by popping stack. Furthermore, stack is customised so that string representation is a CSV list of all values in the stack from first added value to last added value – to allow easy reconstruction of the stack from a file.

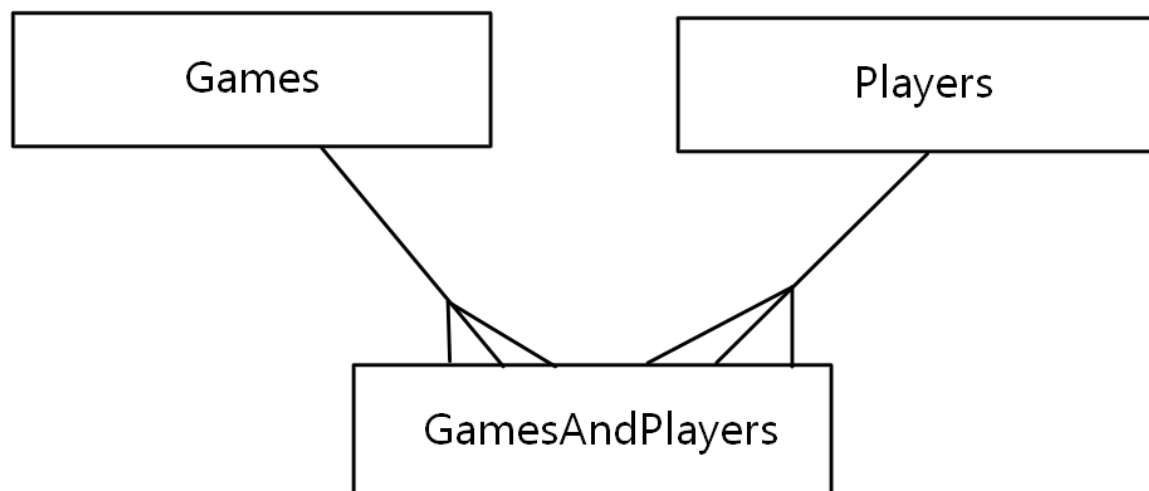
List -> Used to store pieces, number of pieces is variable

Used to store board squares for quick access

Possible moves of selected piece, variable number of possible moves and also can be checked quickly to see if a proposed move is contained within the list

### Database Design

The database will be composed of three related tables, games, players and a joint table to relate their keys.



Each player record will record their number of losses, wins and draws when playing as each colour, this can later be reconstructed into more digestible win-rates when we display a list of players to the user – these attributes may still be useful for sorting the records. The two other most important attributes of a player are their name and of course their Primary key in the table, as we will be using this as a foreign key in the game table.

Each game record will store basic information about the game, such as its name and when it was last accessed. More interestingly and pertaining to the information of the game, the entire list of moves made in the game is stored in order, as well as whether the “Undo Move” rule is allowed and the outcome of the game (this is set to ongoing if the game hasn’t ended yet). Two attributes of the game are the foreign keys of each player, using their primary keys from the player table as a unique identifier as opposed to a Name, which may be non-unique. When displaying a list of games to the user we can use these Foreign Keys to get the name of the players in the game.

The third table will simply be used to relate the players (and their IDs) and the games (using their IDs also).

### Player Table Creation:

```
CREATE TABLE player (  
  playerId INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
  name TEXT NOT NULL,  
  whiteLosses INTEGER NOT NULL,  
  blackLosses INTEGER NOT NULL,  
  whiteDraws INTEGER NOT NULL,  
  blackDraws INTEGER NOT NULL,  
  whiteWins INTEGER NOT NULL,  
  blackWins INTEGER NOT NULL,  
  date DATE NOT NULL  
);
```

### Game Table Creation:

```
CREATE TABLE game (  
  gameId INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
  name TEXT NOT NULL,  
  moveList TEXT NOT NULL,  
  gameState INTEGER NOT NULL,  
  lastAccessed DATE NOT NULL,  
  undoMoves BOOLEAN NOT NULL  
);
```

### Join Table Creation:

```
CREATE TABLE gamesPlayers (  
  gameId INTEGER NOT NULL,  
  whitePlayerID INTEGER NOT NULL,  
  blackPlayerID INTEGER NOT NULL  
);
```

Most of the time we will want to get all the information about a record in the database, to be presented to the user, in the players menu, we want to display all information about all the players in a sortable table for the user:

```
SELECT * FROM player;
```

In the case of games, we also want to get at which players are in the game which requires us to access the join table and perform a cross-table query:

```
SELECT game.gameID, game.name, game.moveList, game.gamestate, game.lastAccessed,  
game.undoMoves, gamesPlayers.whitePlayerID, gamesPlayers.blackPlayerID FROM game, gamesPlayers  
WHERE game.gameID=gamesPlayers.gameID;
```

To access a singular game, most likely to load it we simply parameterise this query:

```
SELECT game.gameID, game.name, game.moveList, game.gamestate, game.lastAccessed,  
game.undoMoves, gamesPlayers.whitePlayerID, gamesPlayers.blackPlayerID FROM game, gamesPlayers  
WHERE game.gameID=$input AND game.gameID=gamesPlayers.gameID;
```

However, in a lot of the other cases for the player table we only need their name and ID (to ensure entries remain unique). For example, in the dropdown menu for a selecting a player on the new game menu, we can query:

```
SELECT playerId, name FROM player;
```

Deleting records requires multiple queries since you can't delete from multiple tables at once in SQLite. For example, deleting a game requires us to delete the ID from two tables.

```
DELETE FROM game WHERE gameId=$input;
DELETE FROM gamesPlayers WHERE gameId=$input;
```

Thus, when deleting a player, we need to also delete all games that the player appears in. Due to singular DELETE statements we need to also select the IDs of games that the player is in and then pipe this into our game deletion function.

## Technical Solution Highlights

In this section I will highlight some of the skills that I have employed in implementing my solution as well as any notable algorithms or neat implementations, including:

- Linked List Maintenance
- List Operations
- Stack Operations
- Complex Mathematical Model
- Object Oriented Programming: Polymorphism
- Recursive Algorithms
- Merge Sort
- Object Oriented Programming: Dynamic Generation of Objects based on User Input
- Cross-Table Parameterised SQL
- Defensive Programming
- Good Exception Handling

### Linked List Maintenance

In order to store collections of objects of variable length for quick access, I implemented lists in my project. The implementation consists of two classes: ListNodes which contains the data to be stored (or a reference in case of an object), using C# Generics, and List which manages the collection of Nodes, able to navigate through them to access data at certain indexes.

ListNode Class:

```
class ListNode<T>
{
    private T data;
    public ListNode<T> next = null;

    //constructor
    public ListNode(T inp)
    {
        data = inp;
    }

    //get set methods
    public T GetData() { return data; }
    public void SetData(T inp) { data = inp; }
}
```

List Class:

A notable feature of my list implementation is the implementation of an Indexer interface, so that elements in the list can be referenced alike to an array or natively implemented list, such as “list[2]”,

rather than having than exposing a public method such as “list.GetAt(2)”. This renders the code more readable.

```
class List<T> d
{
    //head node pointing to nothing by default
    private ListNode<T> head = null;

    public List() { }
    public List(T inp)
    {
        Add(inp);
    }
    //constructor that takes an array as input
    public List(T[] arr)
    {
        foreach (T o in arr) { Add(o); }
    }

    //adds to end of list
    public void Add(T inp)
    {
        ListNode<T> tmp = head;
        //checks if head is null, if it is it starts the list there with a new node
        if (head != null)
        {
            //moves through to end of list
            while (tmp.next != null)
            {
                tmp = tmp.next;
            }
            //sets next reference to a new node
            tmp.next = new ListNode<T>(inp);
        }
        else { head = new ListNode<T>(inp); }
    }

    //allows the use of indexers to access the list
    public T this[int i]
    {
        //uses private methods
        get { return RetrieveAt(i, head); }
        set { SetAt(i, head, value); }
    }

    //private set method
    private void SetAt(int i, ListNode<T> node, T inp)
    {
        if (i < 0) { throw new IndexOutOfRangeException(); }
        ListNode<T> tmp = node;
        //runs while i >= 0, fine to do this here since using ints
        while (i > 0)
        {
            //if tmp is empty then that means index has run out of bounds
            if (tmp == null) { throw new IndexOutOfRangeException(); }
        }
    }
}
```

```

        tmp = tmp.next;
        i--;
    }
    tmp.SetData(inp);
}

public T RemoveAt(int i)
{
    ListNode<T> tmp = head;
    T ret = default;
    if (i == 0)
    {
        if (tmp == null) { throw new ArgumentOutOfRangeException(); }
        ret = head.GetData();
        head = tmp.next;
    }
    else
    {
        i--;
        //RETRIEVES THE LIST ITEM BEFORE THE ONE TO REMOVE
        while (i > 0)
        {
            //if tmp is empty that means index is out of bounds
            if (tmp == null) { throw new ArgumentOutOfRangeException(); }
            tmp = tmp.next;
            i--;
            if (tmp.next == null) { throw new ArgumentOutOfRangeException(); }
        }
        ret = tmp.next.GetData();
        tmp.next = tmp.next.next;
    }

    return ret;
}

//Other methods...

//this method can be used to store the list or compare it, retaining order. Then later reconstructed using a
Split(',')
public string ConvertToString()
{
    string tmpstr = "";
    for (int x = 0; x < Count(); x++)
    {
        tmpstr = tmpstr + Convert.ToString(RetrieveAt(x, head));
        //if not the last record add a comma, can be used to split list later !!!
        if (x != Count() - 1) { tmpstr = tmpstr + ","; }
    }
    return tmpstr;
}
}

```

## List Operations

Lists are used extensively throughout the program, one example of how they're operated upon is the AddPiece method in Board.cs:

```
//...
public void AddPiece(string type, int pos, int colour)
{
    Piece p = null;
    switch (type)
    {
        case "R":
            p = new Rook(pos, colour);
            break;
        case "P":
            p = new Pawn(pos, colour);
            break;
        case "K":
            p = new King(pos, colour);
            break;
        case "B":
            p = new Bishop(pos, colour);
            break;
        case "N":
            p = new Knight(pos, colour);
            break;
        case "Q":
            p = new Queen(pos, colour);
            break;
        case "TSP":
            p = new ThreatSuperPiece(pos, colour);
            break;
    }

    pieces.Add(p);

    board[pos].SetPiecePointer(pieces.Count() - 1);
}
//...
```

Another example of list operations, would be the complex operations required to promote a pawn – also in Board.cs:

```
//...
public bool PromotePawn(string pieceType, int squarePtr)
{
    //find the index of the piece in the list
    int piecePtr = board[squarePtr].GetPiecePointer();
    bool success = false;
    //rereference piece in list to a new piece of promoted variant
    switch (pieceType)
    {
        case "Q":
            pieces[piecePtr] = new Queen(squarePtr, pieces[piecePtr].GetColour());
            success = true;
            break;
        case "B":
```



```

        pieces[piecePtr] = new Bishop(squarePtr, pieces[piecePtr].GetColour());
        success = true;
        break;
    case "R":
        pieces[piecePtr] = new Rook(squarePtr, pieces[piecePtr].GetColour());
        success = true;
        break;
    case "N":
        pieces[piecePtr] = new Knight(squarePtr, pieces[piecePtr].GetColour());
        success = true;
        break;
    }
    return success;
}
//...

```

### Stack Operations

A stack is used to handle the list of past moves, we see these stack operations employed in many of the functions that perform or change the move history (in Chess.cs).

```

public void UndoMove()
{
    if (!moveList.IsEmpty() && undoMovesAllowed)
    {
        string m = moveList.Pop();
        board.UndoMove(m);
        playerTurn = (playerTurn + 1) % 2;
        switch (state)
        {
            case (int)Gamestates.WhiteW:
                playerTurn = (int)Colours.White;
                break;
            case (int)Gamestates.BlackW:
                playerTurn = (int)Colours.Black;
                break;
            case (int)Gamestates.Stalemate:
                playerTurn = (moveList.Count() + 1) % 2;
                break;
        }
        EvalGamestate();
        db.UpdateGame(moveList.ConvertToString(), state, ID);
    }
}

private void AttemptMove(int squareIndex)
{
    //player turn is checked in board method
    string move = board.MovePiece(squareIndex, playerTurn);

    //if the returning notation is not null, then the move has been effected, next player's turn
    if (move != null)
    {
        playerTurn = (playerTurn + 1) % 2;
    }
}

```

//gamestate is evaluated at the start of a player's turn, will be unnoticeable to players but makes the maths easier

```

inCheck = board.CheckCheck(playerTurn);
EvalGamestate();
//add check/stalemate/checkmate symbols here
switch (state)
{
    //lock undo once game is complete, if game is reopened from menu it opens a undoable copy
    case (int)Gamestates.Stalemate:
        //add ÷ symbol for stalemate
        move = move + "÷";
        undoMovesAllowed = false;
        whitePlayer.AddWhiteDraw();
        blackPlayer.AddBlackDraw();
        break;
    case (int)Gamestates.WhiteW:
        // add # symbol for checkmate
        move = move + "#";
        undoMovesAllowed = false;
        whitePlayer.AddWhiteWin();
        blackPlayer.AddBlackLoss();
        break;
    case (int)Gamestates.BlackW:
        // add # symbol for checkmate
        move = move + "#";
        undoMovesAllowed = false;
        blackPlayer.AddBlackWin();
        whitePlayer.AddWhiteLoss();
        break;
    default:
        //this is if game is ongoing but still need to append + if in check for movelist readability
        if (inCheck)
        {
            move = move + "+";
        }
        break;
}
if (move.Contains("="))
{
    state = (int)Gamestates.PendingPromo;
    //reverse playerTurn change
    playerTurn = (playerTurn - 1) % 2;
    //holding square int here to ref piece down the line
    pendingMove = squareIndex;
    //select piece so that player is more aware of it
    SelectPiece(squareIndex);
    //push move if promo, just don't forget to pop it later
}
moveList.Push(move);
db.UpdateGame(moveList.ConvertToString(), state, ID);
db.UpdatePlayer(whitePlayer);
db.UpdatePlayer(blackPlayer);
}
}

```

Stacks are also used to reverse the order of Lists, particularly when displaying sorted data.

```
public List<Player> Reverse(List<Player> list)
{
    //reverse a list simply, using a stack
    Stack<Player> stck = new Stack<Player>();
    for(int x = 0; x < list.Count(); x++)
    {
        stck.Push(list[x]);
    }
    List<Player> ret = new List<Player>();
    for (int x = 0; x < list.Count(); x++)
    {
        ret.Add(stck.Pop());
    }
    return ret;
}
```

### Object Oriented Programming: Polymorphism

All the different types of pieces inherit from a Piece class (in Piece.cs). It makes use of abstract and virtual methods to control the polymorphism of inheriting pieces.

```
abstract class Piece
{
    //stores current position on board
    public int currentPosition;
    public int colour;

    //constructor for piece
    public Piece(int startPos, int col)
    {
        currentPosition = startPos;
        colour = col;
    }

    public abstract string GetPieceType();

    //method to calculate all possible moves by a piece - returns a list, must be implemented on a per piece basis
    public abstract List<int> GeneratePossibleMoves(List<Square> board, List<Piece> pieces);

    public virtual string MovePiece(int endPosition, List<Square> board, List<Piece> pieces)
    {
        //setup string to return LA3DN data
        string data = "";
        //dereference captured piece here
        int targetPiecePtr = board[endPosition].GetPiecePointer();
        if (targetPiecePtr != -1)
        {
            data += "X" + pieces[targetPiecePtr].GetPieceType();
            pieces.RemoveAt(targetPiecePtr);
        }
        else { data += "-"; }
        data += ConvertPosToStr(endPosition);
    }
}
```

```

        currentPosition = endPosition;
        return data;
    }
    //...

```

Here's an example of how some classes which inherit differ heavily (in Queen.cs):

```

public override string GetPieceType() { return "Q"; }

//override virtual move piece method to move internal pieces with piece
public override string MovePiece(int endPosition, List<Square> board, List<Piece> pieces)
{
    //setup string to return LA3DN data
    string data = "";
    //dereference captured piece here
    int targetPiecePtr = board[endPosition].GetPiecePointer();
    if (targetPiecePtr != -1)
    {
        data += "X" + pieces[targetPiecePtr].GetPieceType();
        pieces.RemoveAt(targetPiecePtr);
    }
    else { data += "-"; }
    data += ConvertPosToStr(endPosition);

    currentPosition = endPosition;
    internalBishop.ForceMove(endPosition);
    internalRook.ForceMove(endPosition);

    return data;
}

public override List<int> GeneratePossibleMoves(List<Square> board, List<Piece> pieces)
{
    List<int> moves = new List<int>();

    //generate moves using internal pieces and then append to main list of moves
    List<int> tmp = internalRook.GeneratePossibleMoves(board, pieces);
    for (int x = 0; x < tmp.Count(); x++) { moves.Add(tmp[x]); }
    tmp = internalBishop.GeneratePossibleMoves(board, pieces);
    for (int x = 0; x < tmp.Count(); x++) { moves.Add(tmp[x]); }

    return moves;
}

```

As you can see above, each piece implements GeneratePossibleMoves in a different way, using the override keyword, this is useful because each piece moves in a different way. Because of the abstract keyword in Piece.cs, GeneratePossibleMove must be implemented in every piece's class which ensures all pieces have the same interfaces. Here is another example of the GeneratePossibleMoves override (in Knight.cs):

```

public override List<int> GeneratePossibleMoves(List<Square> board, List<Piece> pieces)
{
    List<int> moves = new List<int>();

```

```

//possible moves of knight
int[] xMoves = { 1, 1, -1, -1, 2, 2, -2, -2, 1, 1, -1, -1, 2, 2, -2, -2, 0, 0, 0, 0, 0, 0, 0, 0 };
int[] yMoves = { 2, -2, 2, -2, 1, -1, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, -1, -1, 2, 2, -2, -2 };
int[] zMoves = { 0, 0, 0, 0, 0, 0, 0, 0, 2, -2, 2, -2, 1, -1, 1, -1, 2, -2, 2, -2, 1, -1, 1, -1 };

//iterate through and check moves
for (int i = 0; i < xMoves.Length; i++)
{
    int[] pos = ConvertPtrToVect(currentPosition);
    pos[0] += xMoves[i];
    pos[1] += yMoves[i];
    pos[2] += zMoves[i];

    //check move hasn't gone off board
    if (pos[0] > -1 && pos[0] < Constants.boardDimensions && pos[1] > -1 && pos[1] <
        Constants.boardDimensions && pos[2] > -1 && pos[2] < Constants.boardDimensions)
    {
        int targetPiecePtr = board[ConvertVectToPtr(pos)].GetPiecePointer();
        //add pointer version of move to list if its valid
        //filter empty and capture moves here
        if (targetPiecePtr == -1)
        {
            moves.Add(ConvertVectToPtr(pos));
        }
        else if (pieces[targetPiecePtr].GetColour() != this.colour)
        {
            moves.Add(ConvertVectToPtr(pos));
        }
    }
}

return moves;
}

```

## Recursive Algorithms

Throughout the project, I have made good use of recursion to tackle problems and simplify code. One such example of a recursive algorithm, is that for generating the possible moves of a rook. The rook checks squares in one direction until it reaches a friendly piece, an enemy piece or the edge of the board; this is the base case and recursion then unwinds from there. The GeneratePossibleMoves function runs this recursive algorithm in each 6 directions the rook can move then collects the results. From Rook.cs:

```

//implementation of generate possible moves, doing this recursively because rooks can move to the edge
of the board
public override List<int> GeneratePossibleMoves(List<Square> board, List<Piece> pieces)
{
    List<int> moves = new List<int>();

    for (int direction = (int)Directions.Right; direction <= (int)Directions.Backwards; direction++)
    {
        List<int> tmp = new List<int>();
        //calls recursive directional move generator
    }
}

```

```

    tmp = GenerateNextMove(direction, board, currentPosition, pieces);
    //appends all moves generated into moves list
    for (int x = 0; x < tmp.Count(); x++)
    {
        //filter out when invalid moves have been reached
        if (tmp[x] != -1) { moves.Add(tmp[x]); }
    }
}

return moves;
}

private List<int> GenerateNextMove(int dir, List<Square> board, int pos, List<Piece> pieces)
{
    int[] vect = ConvertPtrToVect(pos);
    //converts direction into an index that addresses the relevant part of a 3-Length Vector Array: (0&1)-> 0,
    (2&3)-> 1, (4&5) -> 2
    int arrayIndex = (dir / 2) % 3;

    //switch to transform position in direction specified - will be using base 8 things
    switch (dir)
    {
        case (int)Directions.Right:
            //moves right one square
            pos++;
            vect[arrayIndex]++;
            break;
        case (int)Directions.Left:
            //moves left one square
            pos--;
            vect[arrayIndex]--;
            break;
        case (int)Directions.Up:
            //moves up one square
            pos += 8;
            vect[arrayIndex]++;
            break;
        case (int)Directions.Down:
            //moves down one square
            pos -= 8;
            vect[arrayIndex]--;
            break;
        case (int)Directions.Forwards:
            //moves deeper on z axis by one square
            pos += 64;
            vect[arrayIndex]++;
            break;
        case (int)Directions.Backwards:
            //moves back on z axis by one square
            pos -= 64;
            vect[arrayIndex]--;
            break;
    }
}

```

```

List<int> moves = new List<int>();

//checks that piece hasn't gone off the edge of the board
if (vect[arrayIndex] < Constants.boardDimensions && vect[arrayIndex] > -1)
{
    //checks if there is a piece on the square
    int targetPtr = board[pos].GetPiecePointer();
    if (targetPtr != -1)
    {
        //now check if piece is friendly or enemy
        Piece target = pieces[targetPtr];
        if (target.GetColour() != colour)
        {
            //unwind recursion from here
            moves.Add(pos);
            return moves;
        } // return a -1 to be filtered out later
        else
        {
            moves.Add(-1);
            return moves;
        }
    }
    else
    {
        moves.Add(pos);
        List<int> newMoves = new List<int>();
        newMoves = GenerateNextMove(dir, board, pos, pieces);
        //go deeper in recursion here
        for (int x = 0; x < newMoves.Count(); x++)
        {
            moves.Add(newMoves[x]);
        }
    }
}

} // return a -1 to be filtered out later, prevent null errors
else { moves.Add(-1); }

return moves;
}

```

I also make use of recursion in my linked list maintenance, as we can see with functions such as RemoveAt (in List.cs):

```

public T RemoveAt(int i)
{
    ListNode<T> tmp = head;
    T ret = default;
    if (i == 0)
    {
        if (tmp == null) { throw new ArgumentOutOfRangeException(); }
        ret = head.GetData();
        head = tmp.next;
    }
    else

```

```

{
    i--;
    //RETRIEVES THE LIST ITEM BEFORE THE ONE TO REMOVE
    while (i > 0)
    {
        //if tmp is empty that means index is out of bounds
        if (tmp == null) { throw new ArgumentOutOfRangeException(); }
        tmp = tmp.next;
        i--;
        if (tmp.next == null) { throw new ArgumentOutOfRangeException(); }
    }
    //dereferences next node and sets reference to next node over
    ret = tmp.next.GetData();
    tmp.next = tmp.next.next;
}

return ret;
}

```

### Complex Algorithms: Merge Sort

Another recursive algorithm that I implemented, though not original, was the Merge Sort. I used this sort to quickly sort lists of players and games when presenting them to the user to be selected. From Sorter.cs:

```

public List<Player> MergeSortString(List<Player> list)
{
    //exit recursion
    if (list.Count() <= 1) { return list; }

    //split list into two
    int midpoint = list.Count() / 2;
    List<Player> left = new List<Player>();
    List<Player> right = new List<Player>();
    for(int x = 0; x < midpoint; x++)
    {
        left.Add(list[x]);
    }
    for(int x = midpoint; x < list.Count(); x++)
    {
        right.Add(list[x]);
    }
    //recurse with each half
    left = MergeSortString(left);
    right = MergeSortString(right);
    List<Player> sorted = MergeString(left, right);
    return sorted;
}

private List<Player> MergeString(List<Player> left, List<Player> right)
{
    List<Player> ret = new List<Player>();
    // combine and sort elements
    while(left.Count() > 0 || right.Count() > 0)
    {

```



```

//check both lists have elemnts to compare
if(left.Count() > 0 && right.Count() > 0)
{
    //compare and reorder elemnts
    if (left[0].GetName()[0] <= right[0].GetName()[0])
    {
        ret.Add(left[0]);
        left.RemoveAt(0);
    }
    else
    {
        ret.Add(right[0]);
        right.RemoveAt(0);
    }
}
//otherwise add remaining contents of list
else if(left.Count() > 0)
{
    ret.Add(left[0]);
    left.RemoveAt(0);
}
else
{
    ret.Add(right[0]);
    right.RemoveAt(0);
}
}
return ret;
}

```

### Object Oriented Programming: Dynamic Generation of Objects Based on User Input

The model is robust enough to dynamically generate objects based on user input, a good example of this and a core part of the rules of chess would be the algorithms used to implement pawn promotion. Firstly, the pawn has reached that position due to User Input and then we receive further User Input to replace that piece with a piece of the user's choice.

From UserInterface.cs, taking user input:

```

if (game.GetGameState() == (int)Gamestates.PendingPromo)
{
    //check for pawn promotion selection ehre
    bool queenSelected = Raylib.CheckCollisionPointRec(mousePos, queenPromoRec);
    bool rookSelected = Raylib.CheckCollisionPointRec(mousePos, rookPromoRec);
    bool bishopSelected = Raylib.CheckCollisionPointRec(mousePos, bishopPromoRec);
    bool knightSelected = Raylib.CheckCollisionPointRec(mousePos, knightPromoRec);
    if (queenSelected) { game.PromotePawn("Q"); }
    else if (rookSelected) { game.PromotePawn("R"); }
    else if (bishopSelected) { game.PromotePawn("B"); }
    else if (knightSelected) { game.PromotePawn("N"); }
}

```

From Board.cs, acting on user input:

```

public bool PromotePawn(string pieceType, int squarePtr)
{

```

```

int piecePtr = board[squarePtr].GetPiecePointer();
bool success = false;
//reference piece in list to a new piece of promoted variant
switch (pieceType)
{
    case "Q":
        pieces[piecePtr] = new Queen(squarePtr, pieces[piecePtr].GetColour());
        success = true;
        break;
    case "B":
        pieces[piecePtr] = new Bishop(squarePtr, pieces[piecePtr].GetColour());
        success = true;
        break;
    case "R":
        pieces[piecePtr] = new Rook(squarePtr, pieces[piecePtr].GetColour());
        success = true;
        break;
    case "N":
        pieces[piecePtr] = new Knight(squarePtr, pieces[piecePtr].GetColour());
        success = true;
        break;
}
return success;
}

```

### Cross Table Parameterised SQL

In my project I store a list of games (ongoing and finished) and players, I implemented a database to handle this. SQL was useful for accessing records based on certain identifiers and getting access to the large amounts of data, simply and quickly. Here are a few examples of the queries and the code that interacts with it.

This method is used to create a new game in the database, it also returns the ID of the newly created game further up the program so it can be accessed later.

```

public int CreateGame(string name, bool undo, int whiteID, int blackID)
{
    SQLiteConnection dbConnection = new SQLiteConnection("Data Source=database.db");
    dbConnection.Open();
    SQLiteCommand comm = dbConnection.CreateCommand();

    comm.CommandText = @"
INSERT INTO game (name, moveList, gamestate, lastAccessed, undoMoves)
VALUES ($name, $empty, $state, $date, $undo);";

    //insert params
    comm.Parameters.AddWithValue("$name", name);
    comm.Parameters.AddWithValue("$empty", "");
    comm.Parameters.AddWithValue("$state", (int)Gamestates.Ongoing);
    comm.Parameters.AddWithValue("$date", DateTime.Today);
    comm.Parameters.AddWithValue("$undo", undo);

    comm.ExecuteNonQuery();
}

```

```

//once game is created need to return its ID to higher level
comm.CommandText = "SELECT * FROM game ORDER BY gameId DESC LIMIT 1;";
SQLiteDataReader reader = comm.ExecuteReader();
reader.Read();
int ret = reader.GetInt32(0);
reader.Close();

//update join table with game info
comm.CommandText = @"
INSERT INTO gamesPlayers (gameID, whitePlayerID, blackPlayerID)
VALUES ($game, $white, $black);";
comm.Parameters.AddWithValue("$game", ret);
comm.Parameters.AddWithValue("$white", whiteID);
comm.Parameters.AddWithValue("$black", blackID);
comm.ExecuteNonQuery();

dbConnection.Close();
return ret;
}

```

This example is cross-table and parameterised, it's used for loading all the relevant information about a game. Only player IDs are fetched since then we can load player information when it's needed.

```

public GameInfo GetGame(int inp)
{
    SQLiteConnection dbConnection = new SQLiteConnection("Data Source=database.db");
    dbConnection.Open();
    SQLiteCommand comm = dbConnection.CreateCommand();

    comm.CommandText = "SELECT game.gameID, game.name, game.moveList, game.gamestate,
game.lastAccessed, game.undoMoves, gamesPlayers.whitePlayerID, gamesPlayers.blackPlayerID FROM
game, gamesPlayers WHERE game.gameID=$input AND game.gameID=gamesPlayers.gameID;";
    comm.Parameters.AddWithValue("$input", inp);
    SQLiteDataReader reader = comm.ExecuteReader();

    //pull values
    reader.Read();
    int ID = reader.GetInt32(0);
    string name = reader.GetString(1);
    string moveListRepr = reader.GetString(2);
    int gamestate = reader.GetInt32(3);
    DateTime lastAccessed = reader.GetDateTime(4);
    bool undoMoves = reader.GetBoolean(5);
    int whiteID = reader.GetInt32(6);
    int blackID = reader.GetInt32(7);

    GameInfo tmp = new GameInfo(ID, name, moveListRepr, gamestate, lastAccessed, whiteID, blackID,
undoMoves);
    reader.Close();
    dbConnection.Close();

    return tmp;
}

```

## Defensive Programming

In order to control the flow of the program, I made good use of defensive programming in my implementation. This prevents dangerous, error-causing inputs to reach algorithms, such as negative indices reaching lists, and ensures the game plays correctly, pieces are not able to move off the board.

In Chess.cs, the main input method “Click” doesn’t run any instructions if there is a promotion pending or the game has finished:

```
//...
public void Click(int squareIndex)
{
    //can't do anything once game is over - separate method for rewind and view buttons
    if (playerTurn != -1 && state != (int)Gamestates.PendingPromo)
    {
    }
    //...
```

In Board.cs, it is checked that a move is possible and that it is the correct player’s turn before enacting it:

```
//...
    int startSquarePtr = pieces[currentPieceIndex].GetCurrentPosition();

    //target move should be in possible move list and also check that piece belongs to current player
    if (currentPossibleMoves.Contains(targetSquarePtr) && pieces[currentPieceIndex].GetColour() ==
currentPlayer)
    {
    }
    //...
```

Also on Board.cs, we check if there is a piece on the square by inspecting the pointer and checking it before continuing:

```
//...
public void SelectPiece(int squarePtr, int currentPlayer)
{
    //do nothing if square is empty
    //list is setup here in case needed
    int piecePtr = board[squarePtr].GetPiecePointer();
    List<int> moveList = new List<int>();
    if (piecePtr != -1)
    {
    }
    //...
```

Returning to Chess.cs, we use defensive programming to ensure our viewport is safe and limit which cells can be accessed.

```
//...
public Square GetViewportCell(int ptr)
{
    //takes a one dimensional pointer for the viewport
    if(ptr > -1 && ptr < 64)
    {
        return board.GetSquare(viewport[ptr]);
    }
}
```

```

    else { throw new ArgumentOutOfRangeException(); }
}
//...

```

In fact, a core part of the stack maintenance and operations (in Stack.cs) requires defensive programming so that we don't reach any null errors by trying to Pop and empty stack.

```

//...
public T Pop()
{
    T ret = default;
    //checks stack isn't empty
    if (stack.Count() > 0)
    {
        ret = stack.RemoveAt(stack.Count() - 1);
    }
    return ret;
}
//...

```

### Good Exception Handling

In order to ensure safe running of the code and to handle if bad inputs do manage to reach some functions, I have used exception handling. Here is an example from List.cs:

```

//...
private T RetrieveAt(int i, ListNode<T> node)
{
    if (i < 0) { throw new IndexOutOfRangeException(); }
    ListNode<T> tmp = node;
    while (i > 0)
    {
        //if tmp points to null then that means index has run out of bounds
        if (tmp == null) { throw new IndexOutOfRangeException(); }
        tmp = tmp.next;
        i--;
    }
    return tmp.GetData();
}
//...

```

Here is an example from DatabaseHandler.cs where the output of the function is dependent on whether the database operation was successful, note the try{} catch() block to achieve this.

```

//...
public bool DeleteGame(int inp)
{
    SQLiteConnection dbConnection = new SQLiteConnection("Data Source=database.db");
    dbConnection.Open();
    SQLiteCommand comm = dbConnection.CreateCommand();

    comm.CommandText = @"
DELETE FROM game WHERE gameId=$input;
DELETE FROM gamesPlayers WHERE gameId=$input;

```

```

";
    comm.Parameters.AddWithValue("$input", inp);

    bool ret = true;
    try
    {
        comm.ExecuteNonQuery();
    }
    catch (Exception e)
    {
        //probably expecting an SQLLogic or SQLArgument exception here
        Console.WriteLine("Experienced Error: " + e.Message);
        ret = false;
    }
    dbConnection.Close();

    return ret;
}
//...

```

## Implementation

### UserInterface.cs

```

using Raylib_cs;
using System;
using System.Numerics;
using System.Reflection.PortableExecutable;

namespace ThreeDimensionalChess
{
    class UserInterface
    {
        static class UIConstants
        {
            /* ----- graphical notes -----
            origin points to start drawing for board - 550x550
            top left: (225, 24)(due to integer div) bottom left: (225, 500)
            top right: (769, 24)(769 due to 550/8 = 68.75) bottom right: (769, 500)
            is close enough to desired values
            -----
            viewport controls area:
            (225, 24) to (769, 500)
            */

            public const int boardXOrigin = 225;
            public const int boardYOrigin = 500;
            public const int squareSide = 550 / 8;

            public const int windowWidth = 1000;
            public const int windowHeight = 680;
        }

        enum PieceTextureIndices

```

```
{
    //another neat little enum to deal with loading textures from the texture list
    WhitePawn, // 0
    WhiteRook, // 1
    WhiteKnight, // 2
    WhiteBishop, // 3
    WhiteQueen, // 4
    WhiteKing, // 5
    BlackPawn, // 6
    BlackRook, // 7
    BlackKnight, // 8
    BlackBishop, // 9
    BlackQueen, // 10
    BlackKing // 11
}

enum UIModes
{
    MainMenu, // 0
    PlayersList, // 1
    NewLoadChoice, // 2
    NewGameMenu, // 3
    GamesList, // 4
    ConfirmGame, // 5
    CreatePlayer, // 6
    GameUI2D, // 7
    PauseMenu, // 8
    ConfirmForfeitStalemate, // 9
    GameUI3D // 10
}

static void Main()
{
    Raylib.InitWindow(UIConstants.windowWidth, UIConstants.windowHeight, "Three-Dimensional Chess");
    //make it so esc doesn't close window
    Raylib.SetExitKey(0);
    Chess game = null;
    int mode = (int)UIModes.MainMenu;
    //set this as necessary
    int lastMode = 0;
    DatabaseHandler database = new DatabaseHandler();
    Raylib.InitAudioDevice();

    //load sounds
    Sound emptySquare = Raylib.LoadSound("resources/emptySquare.mp3");
    Sound captureSquare = Raylib.LoadSound("resources/captureSquare.mp3");

    //load textures
    List<Texture2D> textures = new List<Texture2D>();
    Texture2D WhitePawn = Raylib.LoadTexture("resources/WhP.png");
    textures.Add(WhitePawn); // 0
    Texture2D WhiteRook = Raylib.LoadTexture("resources/WhR.png");
    textures.Add(WhiteRook); // 1
    Texture2D WhiteKnight = Raylib.LoadTexture("resources/WhN.png");
    textures.Add(WhiteKnight); // 2
```

```

Texture2D WhiteBishop = Raylib.LoadTexture("resources/WhB.png");
textures.Add(WhiteBishop); // 3
Texture2D WhiteQueen = Raylib.LoadTexture("resources/WhQ.png");
textures.Add(WhiteQueen); // 4
Texture2D WhiteKing = Raylib.LoadTexture("resources/WhK.png");
textures.Add(WhiteKing); // 5
Texture2D BlackPawn = Raylib.LoadTexture("resources/BIP.png");
textures.Add(BlackPawn); // 6
Texture2D BlackRook = Raylib.LoadTexture("resources/BIR.png");
textures.Add(BlackRook); // 7
Texture2D BlackKnight = Raylib.LoadTexture("resources/BIN.png");
textures.Add(BlackKnight); // 8
Texture2D BlackBishop = Raylib.LoadTexture("resources/BIB.png");
textures.Add(BlackBishop); // 9
Texture2D BlackQueen = Raylib.LoadTexture("resources/BIQ.png");
textures.Add(BlackQueen); //10
Texture2D BlackKing = Raylib.LoadTexture("resources/BIK.png");
textures.Add(BlackKing); //11

//some button placements, store Rec structures here to make life easier
// rectangle struct - {xOrigin, yOrigin, width, height}
// can't place in constants because they are objects
// -- Main Menu Rectangles --
Vector2 titlePos = new Vector2(200, 200);
Rectangle exitButton = new Rectangle(250, 400, 500 / 3, 100);
bool exitButtonClicked = false;
Rectangle playButton = new Rectangle(250 + (500/3), 400, 500 / 3, 100);
Rectangle playerButton = new Rectangle(250 + 2*(500/3), 400, 500 / 3, 100);
// -- Players List Values --
int playerListIndex = 0;
int selectedPlayerID = -1;
string sortMode = "ID"; // 0 is ascending by default, 1 is reverse
int sortOrder = 0;
List<Player> playersList = new List<Player>();
Rectangle firstListButton = new Rectangle(805, 10, UIConstants.windowWidth - 805 - 5, 75);
Rectangle secondListButton = new Rectangle(805, 95, UIConstants.windowWidth - 805 - 5, 75);
Rectangle backButton = new Rectangle(UIConstants.windowWidth - 85, UIConstants.windowHeight - 85, 75, 75);
// -- Create Player Values --
string entryStr = "";
Rectangle finaliseNewPlayerButton = new Rectangle(350, 300, 300, 100);
Rectangle entryBox = new Rectangle(250, 190, 500, 100);
// -- New / Load Game Choice Menu --
Rectangle newGame = new Rectangle(400, 225, 200, 75);
Rectangle loadGame = new Rectangle(400, 310, 200, 75);
// -- Confirm Game Choice
// -- New Game Menu --
Rectangle whitePlayerDropDown = new Rectangle(10, 150, 300, 50);
Rectangle addWhitePlayer = new Rectangle(320, 150, 50, 50);
int whitePlayerID = 0;
Rectangle blackPlayerDropDown = new Rectangle(690, 150, 300, 50);
Rectangle addBlackPlayer = new Rectangle(630, 150, 50, 50);
int blackPlayerID = 0;
Rectangle undoMovesTickbox = new Rectangle(10, 410, 75, 75);
Rectangle gameNameEntryBox = new Rectangle(250, 410, 500, 100);

```



```

Rectangle startGameButton = new Rectangle(350, 520, 300, 100);
bool gameCanStart = false;
bool undoMovesChoice = true;
int whitePlayerListIndex = -1; // when -1, drop down is closed
int blackPlayerListIndex = -1;
bool newBlackFlag = false;
bool newWhiteFlag = false;
// -- Load Game Values --
List<GameInfo> games = new List<GameInfo>();
int gameListIndex = 0;
GameInfo selectedGame = null;
// -- Game UI 2D Rectangles --
Rectangle frontButton = new Rectangle(10, 10, 200, 75);
Rectangle topButton = new Rectangle(10, 95, 200, 75);
Rectangle sideButton = new Rectangle(10, 180, 200, 75);
Rectangle queenPromoRec = new Rectangle(10, 265, 100, 100);
Rectangle rookPromoRec = new Rectangle(110, 265, 100, 100);
Rectangle bishopPromoRec = new Rectangle(10, 365, 100, 100);
Rectangle knightPromoRec = new Rectangle(110, 365, 100, 100);
Rectangle moveListRec = new Rectangle(780, 10, 210, 505);
Rectangle undoMoveButton = new Rectangle(780, 525, 210, 50);
// -- Pause Menu Elements --
Rectangle resumeButton = new Rectangle(350, 150, 300, 75);
Rectangle exitMenuButton = new Rectangle(350, 225, 300, 75);
Rectangle exitDesktopButton = new Rectangle(350, 300, 300, 75);
Rectangle whiteForfeitButton = new Rectangle(40, 500, 300, 75);
Rectangle mutualAgreementStalemateButton = new Rectangle(350, 500, 300, 75);
Rectangle blackForfeitButton = new Rectangle(660, 500, 300, 75);
// -- Forfeit/Draw Menu --
int proposedOutcome = -1;
// -- Game Ui 3D --
Rectangle viewModeToggleButton = new Rectangle(10, 595, 75, 75);
Camera3D camera = new Camera3D();
camera.Position = new Vector3(10f, 10f, 10f);
camera.Target = new Vector3(0f, 0f, 0f);
camera.Up = new Vector3(0f, 1f, 0f);
camera.FovY = 45f;
camera.Projection = CameraProjection.CAMERA_PERSPECTIVE;
Vector3 CubePos = new Vector3(0f, 0f, 0f);
bool viewmodelWiresChoice = false;
Rectangle wiremodeToggleButton = new Rectangle(10, 510, 75, 75);

while (!Raylib.WindowShouldClose() && !exitButtonClicked)
{
    // ----- update and input here -----
    // switch for current part of UI
    switch (mode)
    {
        case (int)UIModes.MainMenu:
            if (Raylib.IsMouseButtonPressed(MouseButton.MOUSE_BUTTON_LEFT))
            {
                Vector2 mousePos = Raylib.GetMousePosition();
                exitButtonClicked = Raylib.CheckCollisionPointRec(mousePos, exitButton);
            }
    }
}

```

```

    bool playButtonClicked = Raylib.CheckCollisionPointRec(mousePos, playButton);
    bool playersButtonClicked = Raylib.CheckCollisionPointRec(mousePos, playerButton);
    if (playButtonClicked) { mode = (int)UIModes.NewLoadChoice; }
    if (playersButtonClicked) { mode = (int)UIModes.PlayersList; }
}
break;
case (int)UIModes.PlayersList:
    if (Raylib.IsMouseButtonPressed(MouseButton.MOUSE_BUTTON_LEFT))
    {
        Vector2 mousePos = Raylib.GetMousePosition();

        //take clicks on the table
        if(10 <= mousePos.X && mousePos.X <= 800 && 10 <= mousePos.Y && mousePos.Y <= (10 +
(13*50)))
        {
            int row = ((int)mousePos.Y - 10) / 50;
            int column = -1;
            if (10 <= mousePos.X && mousePos.X < 200) { column = 0; }
            else if (200 <= mousePos.X && mousePos.X < 300) { column = 1; }
            else if (300 <= mousePos.X && mousePos.X < 500) { column = 2; }
            else if(500 <= mousePos.X && mousePos.X < 650) { column = 3; }
            else { column = 4; }

            //affect table now

            if(row == 0)
            {
                string prevMode = sortMode;
                switch (column)
                {
                    case 0:
                        sortMode = "name";
                        break;
                    case 1:
                        sortMode = "date";
                        break;
                    case 2:
                        sortMode = "winRatio";
                        break;
                    case 3:
                        sortMode = "whiteWR";
                        break;
                    case 4:
                        sortMode = "blackWR";
                        break;
                }
                if (prevMode == sortMode) { sortOrder = (sortOrder + 1) % 2; }
                else { sortOrder = 0; }
            }
            else
            {
                row--;
                if(row < playersList.Count()) { selectedPlayerID = playersList[row].GetID(); }
            }
        }
    }
}

```

```

    bool createPlayerPressed = Raylib.CheckCollisionPointRec(mousePos, secondListButton);
    bool deletePlayerPressed = Raylib.CheckCollisionPointRec(mousePos, firstListButton);
    bool backButtonPressed = Raylib.CheckCollisionPointRec(mousePos, backButton);

    if (createPlayerPressed) { mode = (int)UIModes.CreatePlayer; lastMode = (int)UIModes.PlayersList;
}
    if(deletePlayerPressed && selectedPlayerID != -1) { database.DeletePlayer(selectedPlayerID);
selectedPlayerID = -1; }
    if (backButtonPressed) { mode = (int)UIModes.MainMenu; playerListIndex = 0; }
    }
    //use arrow keys to move up or down table
    if (Raylib.IsKeyPressed(KeyboardKey.KEY_DOWN))
    {
        if(playerListIndex < playersList.Count() - 1) { playerListIndex++; }
    }
    if (Raylib.IsKeyPressed(KeyboardKey.KEY_UP))
    {
        if(playerListIndex > 0) { playerListIndex--; }
    }
    break;
case (int)UIModes.CreatePlayer:
    if (Raylib.IsMouseButtonPressed(MouseButton.MOUSE_BUTTON_LEFT))
    {
        Vector2 mousePos = Raylib.GetMousePosition();

        bool finaliseButtonPressed = Raylib.CheckCollisionPointRec(mousePos, finaliseNewPlayerButton);
        bool backButtonPressed = Raylib.CheckCollisionPointRec(mousePos, backButton);
        if (finaliseButtonPressed)
        {
            //shouldn't reach this without having moved from another mode so move back to that
            if (entryStr != "")
            {
                int ID = database.AddPlayer(entryStr);
                mode = lastMode;
                lastMode = 0;
                entryStr = "";
                if (newWhiteFlag) { whitePlayerID = ID; newWhiteFlag = false; }
                else if (newBlackFlag) { blackPlayerID = ID; newBlackFlag = false; }
            }
        }
        if (backButtonPressed) { mode = lastMode; lastMode = 0; newBlackFlag = false; newWhiteFlag = false;
    }
}

//raylib uses an input queue for keys(if they are occuring on the same frame), parse this here
int keyPressed = Raylib.GetCharPressed();
while(keyPressed > 0)
{
    //only accept character keys, limit names to 20 char (only 10 is displayed in table anyway)
    if(keyPressed > 31 && keyPressed < 126 && entryStr.Length < 20)
    {
        entryStr += (char)keyPressed;
    }
    keyPressed = Raylib.GetCharPressed();
}

```

```

    }
    if (Raylib.IsKeyPressed(KeyboardKey.KEY_BACKSPACE))
    {
        if(entryStr.Length <= 1) { entryStr = ""; }
        else
        {
            entryStr = entryStr.Substring(0, entryStr.Length - 1);
        }
    }
    //had to put this here as it wouldn't work as a composite eval with the button bool above for some
reason
    if (Raylib.IsKeyPressed(KeyboardKey.KEY_ENTER))
    {
        if (entryStr != "")
        {
            int ID = database.AddPlayer(entryStr);
            mode = lastMode;
            lastMode = 0;
            entryStr = "";
            if (newWhiteFlag) { whitePlayerID = ID; newWhiteFlag = false; }
            else if (newBlackFlag) { blackPlayerID = ID; newBlackFlag = false; }
        }
    }
    break;
case (int)UIModes.NewLoadChoice:
    if (Raylib.IsMouseButtonPressed(MouseButton.MOUSE_BUTTON_LEFT))
    {
        Vector2 mousePos = Raylib.GetMousePosition();

        bool newGameSelected = Raylib.CheckCollisionPointRec(mousePos, newGame);
        bool loadGameSelected = Raylib.CheckCollisionPointRec(mousePos, loadGame);
        bool backButtonPressed = Raylib.CheckCollisionPointRec(mousePos, backButton);

        if (newGameSelected) { mode = (int)UIModes.NewGameMenu; }
        if (loadGameSelected) { mode = (int)UIModes.GamesList; }
        if (backButtonPressed) { mode = (int)UIModes.MainMenu; }
    }
    break;
case (int)UIModes.NewGameMenu:
    if (Raylib.IsMouseButtonPressed(MouseButton.MOUSE_BUTTON_LEFT))
    {
        Vector2 mousePos = Raylib.GetMousePosition();

        bool whiteDropDownSelected = Raylib.CheckCollisionPointRec(mousePos, whitePlayerDropDown);
        //check which name has been selected
        if (mousePos.X >= 10 && mousePos.X <= 310 && mousePos.Y >= 200 && mousePos.Y <= 400 &&
whitePlayerListIndex != -1)
        {
            int row = ((int)mousePos.Y - 200) / 50;
            row += whitePlayerListIndex;
            if (row < playersList.Count() && row != -1)
            {
                whitePlayerID = playersList[row].GetID();
            }
        }
    }

```

```

//toggle drop down menu
if (whitePlayerListIndex >= 0) { whitePlayerListIndex = -1; }
else if (whiteDropDownSelected) { whitePlayerListIndex = 0; }

//repeat for black
bool blackDropDownSelected = Raylib.CheckCollisionPointRec(mousePos, blackPlayerDropDown);
if(mousePos.X >= 690 && mousePos.X <= 990 && mousePos.Y >= 200 && mousePos.Y <= 400 &&
blackPlayerListIndex != -1)
{
    int row = ((int)mousePos.Y - 200) / 50;
    row += blackPlayerListIndex;
    if (row < playersList.Count() && row != -1)
    {
        blackPlayerID = playersList[row].GetID();
    }
}
//toggle drop down
if(blackPlayerListIndex >= 0) { blackPlayerListIndex = -1; }
else if (blackDropDownSelected) { blackPlayerListIndex = 0; }

//process other buttons
bool backButtonPressed = Raylib.CheckCollisionPointRec(mousePos, backButton);
bool addBlackPlayerPressed = Raylib.CheckCollisionPointRec(mousePos, addBlackPlayer);
bool addWhitePlayerPressed = Raylib.CheckCollisionPointRec(mousePos, addWhitePlayer);
bool undoMovesTickboxPressed = Raylib.CheckCollisionPointRec(mousePos, undoMovesTickbox);
bool startButtonPressed = Raylib.CheckCollisionPointRec(mousePos, startGameButton);
if (backButtonPressed)
{
    whitePlayerID = 0;
    blackPlayerID = 0;
    whitePlayerListIndex = -1;
    blackPlayerListIndex = -1;
    mode = (int)UIModes.NewLoadChoice;
}
if (addBlackPlayerPressed)
{
    //if adding a new player, raise flag to automatically assign id
    lastMode = (int)UIModes.NewGameMenu;
    newBlackFlag = true;
    mode = (int)UIModes.CreatePlayer;
}
if (addWhitePlayerPressed)
{
    lastMode = (int)UIModes.NewGameMenu;
    newWhiteFlag = true;
    mode = (int)UIModes.CreatePlayer;
}
if (undoMovesTickboxPressed) { undoMovesChoice = !undoMovesChoice; }
if (startButtonPressed)
{
    if(gameCanStart)
    {
        string gameName = entryStr;
        if (entryStr == "") { entryStr = database.GetPlayer(whitePlayerID).GetName() +
database.GetPlayer(blackPlayerID).GetName(); }
    }
}

```

```

        game = new Chess(whitePlayerID, blackPlayerID, gameName, undoMovesChoice);
        mode = (int)UIModes.GameUI2D;
    }
}
//use arrow keys to move up or down selection - limit at ends of lists
if (Raylib.IsKeyPressed(KeyboardKey.KEY_DOWN))
{
    if (whitePlayerListIndex < playersList.Count() - 1 && whitePlayerListIndex > -1) {
whitePlayerListIndex++; }
    else if(blackPlayerListIndex < playersList.Count() - 1 && blackPlayerListIndex > -1) {
blackPlayerListIndex++; }
    }
    if (Raylib.IsKeyPressed(KeyboardKey.KEY_UP))
    {
        if (whitePlayerListIndex > 0) { whitePlayerListIndex--; }
        else if(blackPlayerListIndex > 0) { blackPlayerListIndex--; }
    }

    //raylib uses an input queue for keys(if they are occuring on the same frame), parse this here
    keyPressed = Raylib.GetCharPressed();
    while (keyPressed > 0)
    {
        //only accept character keys, limit names to 20 char (only 10 is displayed in table anyway)
        if (keyPressed > 31 && keyPressed < 126 && entryStr.Length < 20)
        {
            entryStr += (char)keyPressed;
        }
        keyPressed = Raylib.GetCharPressed();
    }
    if (Raylib.IsKeyPressed(KeyboardKey.KEY_BACKSPACE))
    {
        if (entryStr.Length <= 1) { entryStr = ""; }
        else
        {
            entryStr = entryStr.Substring(0, entryStr.Length - 1);
        }
    }
    break;
case (int)UIModes.GamesList:
    if (Raylib.IsMouseButtonPressed(MouseButton.MOUSE_BUTTON_LEFT))
    {
        Vector2 mousePos = Raylib.GetMousePosition();

        //take clicks on the table
        if (10 <= mousePos.X && mousePos.X <= 800 && 10 <= mousePos.Y && mousePos.Y <= (10 + (13 *
50)))
        {
            int row = ((int)mousePos.Y - 10) / 50;
            int column = -1;
            if (10 <= mousePos.X && mousePos.X < 200) { column = 0; }
            else if (200 <= mousePos.X && mousePos.X < 300) { column = 1; }
            else if (300 <= mousePos.X && mousePos.X < 450) { column = 2; }
            else if (450 <= mousePos.X && mousePos.X < 620) { column = 3; }
            else { column = 4; }

```

```

//affect table now

if (row == 0)
{
    string prevMode = sortMode;
    switch (column)
    {
        case 0:
            sortMode = "name";
            break;
        case 1:
            sortMode = "date";
            break;
        case 2:
            sortMode = "gamestate";
            break;
        case 3:
            sortMode = "white";
            break;
        case 4:
            sortMode = "black";
            break;
    }
    if (prevMode == sortMode) { sortOrder = (sortOrder + 1) % 2; }
    else { sortOrder = 0; }
}
else
{
    row--;
    if (row < games.Count()) { selectedGame = games[row]; }
}
}

//using same locations as from
bool loadGameButtonPressed = Raylib.CheckCollisionPointRec(mousePos, firstListButton);
bool deleteGamePressed = Raylib.CheckCollisionPointRec(mousePos, secondListButton);
bool backButtonPressed = Raylib.CheckCollisionPointRec(mousePos, backButton);
if (loadGameButtonPressed) { mode = (int)UIModes.ConfirmGame; }
else if (deleteGamePressed && selectedGame != null) {
    database.DeleteGame(selectedGame.GetGameID()); selectedGame = null; }
else if (backButtonPressed) { mode = (int)UIModes.NewLoadChoice; selectedGame = null; }
}

//use arrow keys to move up or down table
if (Raylib.IsKeyPressed(KeyboardKey.KEY_DOWN))
{
    if (gameListIndex < games.Count() - 1) { gameListIndex++; }
}
if (Raylib.IsKeyPressed(KeyboardKey.KEY_UP))
{
    if (gameListIndex > 0) { gameListIndex--; }
}
break;
case (int)UIModes.ConfirmGame:
    if (Raylib.IsMouseButtonPressed(MouseButton.MOUSE_BUTTON_LEFT))

```

```

{
    Vector2 mousePos = Raylib.GetMousePosition();

    bool confirmGamePressed = Raylib.CheckCollisionPointRec(mousePos, startGameButton);
    bool backButtonPressed = Raylib.CheckCollisionPointRec(mousePos, backButton);

    if (confirmGamePressed)
    {
        game = new Chess(selectedGame);
        selectedGame = null;
        mode = (int)UIModes.GameUI2D;
    } else if (backButtonPressed)
    {
        selectedGame = null;
        mode = (int)UIModes.GamesList;
    }
}
break;
case (int)UIModes.GameUI2D:
    //track mouse clicks
    if (Raylib.IsMouseButtonPressed(MouseButton.MOUSE_BUTTON_LEFT))
    {
        //take mouse x y and find which element was clicked using coordinate geometry
        Vector2 mousePos = Raylib.GetMousePosition();

        //this is if mouse is in board borders
        if (225 <= mousePos.X && mousePos.X <= 769 && 24 <= mousePos.Y && mousePos.Y <= (500 +
        UIConstants.squareSide))
        {
            //calculate square index
            int x = (Convert.ToInt32(mousePos.X) - 225) / UIConstants.squareSide;
            int y = 7 - ((Convert.ToInt32(mousePos.Y) - 25) / UIConstants.squareSide);
            string lastMove = game.GetLastMove();
            game.ViewportClick(x + (y * 8));
            string newLastMove = game.GetLastMove();
            //play sounds
            if (lastMove != newLastMove && newLastMove.Contains('-'))
            {
                Raylib.PlaySound(emptySquare);
            } else if (lastMove != newLastMove && newLastMove.Contains('X'))
            {
                Raylib.PlaySound(captureSquare);
            }
        }
    }

    //step through board using triangle buttons
    bool upButtonPressed = Raylib.CheckCollisionPointTriangle(mousePos, new Vector2(360, 570), new
    Vector2(330, 600), new Vector2(390, 600));
    bool downButtonPressed = Raylib.CheckCollisionPointTriangle(mousePos, new Vector2(610, 570), new
    Vector2(640, 600), new Vector2(670, 570));
    //could list all buttons and give them numerical values then switch statement here?
    if (upButtonPressed) { game.IncrementViewLayer(); }
    else if (downButtonPressed) { game.DecrementViewLayer(); }

    //check collision with viewDirection buttons

```



```

bool frontButtonPressed = Raylib.CheckCollisionPointRec(mousePos, frontButton);
bool topButtonPressed = Raylib.CheckCollisionPointRec(mousePos, topButton);
bool sideButtonPressed = Raylib.CheckCollisionPointRec(mousePos, sideButton);
bool pauseButtonPressed = Raylib.CheckCollisionPointRec(mousePos, backButton);
bool undoMovesPressed = Raylib.CheckCollisionPointRec(mousePos, undoMoveButton);
bool viewmodeTogglePressed = Raylib.CheckCollisionPointRec(mousePos, viewmodeToggleButton);
if (frontButtonPressed)
{
    game.SetViewDirection((int)ViewDirections.Front);
}
else if (topButtonPressed)
{
    game.SetViewDirection((int)ViewDirections.Top);
}
else if (sideButtonPressed)
{
    game.SetViewDirection((int)ViewDirections.Side);
}
else if (pauseButtonPressed)
{
    mode = (int)UIModes.PauseMenu;
}
else if (undoMovesPressed)
{
    game.UndoMove();
}
else if (viewmodeTogglePressed)
{
    mode = (int)UIModes.GameUI3D;
}

if (game.GetGamestate() == (int)Gamestates.PendingPromo)
{
    //check for pawn promotion selection ehre
    bool queenSelected = Raylib.CheckCollisionPointRec(mousePos, queenPromoRec);
    bool rookSelected = Raylib.CheckCollisionPointRec(mousePos, rookPromoRec);
    bool bishopSelected = Raylib.CheckCollisionPointRec(mousePos, bishopPromoRec);
    bool knightSelected = Raylib.CheckCollisionPointRec(mousePos, knightPromoRec);
    if (queenSelected) { game.PromotePawn("Q"); }
    else if (rookSelected) { game.PromotePawn("R"); }
    else if (bishopSelected) { game.PromotePawn("B"); }
    else if (knightSelected) { game.PromotePawn("N"); }
}
}

//step through board using keys
if (Raylib.IsKeyPressed(KeyboardKey.KEY_UP)) { game.IncrementViewLayer(); }
if (Raylib.IsKeyPressed(KeyboardKey.KEY_DOWN)) { game.DecrementViewLayer(); }
break;
case (int)UIModes.PauseMenu:
if (Raylib.IsMouseButtonPressed(MouseButton.MOUSE_BUTTON_LEFT))
{
    Vector2 mousePos = Raylib.GetMousePosition();

    bool resumeButtonPressed = Raylib.CheckCollisionPointRec(mousePos, resumeButton);
    bool exitToMenuPressed = Raylib.CheckCollisionPointRec(mousePos, exitMenuButton);
    bool exitToDesktopPressed = Raylib.CheckCollisionPointRec(mousePos, exitDesktopButton);
    bool whiteForfeitPressed = Raylib.CheckCollisionPointRec(mousePos, whiteForfeitButton);

```

```

    bool blackForfeitPressed = Raylib.CheckCollisionPointRec(mousePos, blackForfeitButton);
    bool agreeToDrawPressed = Raylib.CheckCollisionPointRec(mousePos,
mutualAgreementStalemateButton);
    if (resumeButtonPressed) { mode = (int)UIModes.GameUI2D; }
    else if (exitToDesktopPressed) { exitButtonClicked = true; }
    else if (exitToMenuPressed)
    {
        //clean up variables then exit to menu
        game = null;
        selectedGame = null;
        whitePlayerID = 0;
        blackPlayerID = 0;
        mode = (int)UIModes.MainMenu;
    }
    else if (whiteForfeitPressed && game.GetGamestate() == (int)Gamestates.Ongoing)
    {
        mode = (int)UIModes.ConfirmForfeitStalemate;
        proposedOutcome = (int)Gamestates.BlackW;
    }
    else if (blackForfeitPressed && game.GetGamestate() == (int)Gamestates.Ongoing)
    {
        mode = (int)UIModes.ConfirmForfeitStalemate;
        proposedOutcome = (int)Gamestates.WhiteW;
    } else if (agreeToDrawPressed && game.GetGamestate() == (int)Gamestates.Ongoing)
    {
        mode = (int)UIModes.ConfirmForfeitStalemate;
        proposedOutcome = (int)Gamestates.Stalemate;
    }
    }
    break;
case (int)UIModes.ConfirmForfeitStalemate:
    if (Raylib.IsMouseButtonPressed(MouseButton.MOUSE_BUTTON_LEFT))
    {
        Vector2 mousePos = Raylib.GetMousePosition();

        bool confirmPressed = Raylib.CheckCollisionPointRec(mousePos, startGameButton);
        bool backPressed = Raylib.CheckCollisionPointRec(mousePos, backButton);
        if (confirmPressed)
        {
            game.ManualEndGame(proposedOutcome);
            mode = (int)UIModes.GameUI2D;
            proposedOutcome = -1;
        } else if (backPressed) { mode = (int)UIModes.PauseMenu; proposedOutcome = -1; }
        }
        break;
case (int)UIModes.GameUI3D:
    if (Raylib.IsMouseButtonPressed(MouseButton.MOUSE_BUTTON_LEFT))
    {
        Vector2 mousePos = Raylib.GetMousePosition();
        bool viewmodeTogglePressed = Raylib.CheckCollisionPointRec(mousePos,
viewmodeToggleButton);
        bool wiremodeTogglepressed = Raylib.CheckCollisionPointRec(mousePos,
wiremodeToggleButton);
        if (viewmodeTogglePressed)
        {

```

```

        mode = (int)UIModes.GameUI2D;
    }else if (wiremodeTogglepressed)
    {
        viewmodelWiresChoice = !viewmodelWiresChoice;
    }
}
break;
}

// ----- draw here -----
Raylib.BeginDrawing();
Raylib.ClearBackground(Color.WHITE);
//switch to draw elements based on Ui mode
switch (mode)
{
    case (int)UIModes.MainMenu:
        UpdateMainMenu(titlePos, exitButton, playButton, playerButton);
        break;
    case (int)UIModes.PlayersList:
        playersList = UpdatePlayersTable(playerListIndex, database, sortMode, sortOrder,
selectedPlayerID);
        UpdatePlayersListButtons(firstListButton, secondListButton, backButton, "Delete");
        break;
    case (int)UIModes.CreatePlayer:
        UpdateCreatePlayer(entryStr, entryBox, finaliseNewPlayerButton, backButton);
        break;
    case (int)UIModes.NewLoadChoice:
        UpdateNewLoadButtons(newGame, loadGame, backButton);
        break;
    case (int)UIModes.NewGameMenu:
        //if both players have chosen non identical profiles the game start button becomes available
        if (whitePlayerID != blackPlayerID && whitePlayerID != 0 && blackPlayerID != 0 &&
entryStr.Length > 0)
        {
            gameCanStart = true;
        }
        else { gameCanStart = false; }
        UpdateNewGameButtons(undoMovesTickbox, gameNameEntryBox, startGameButton,
backButton, entryStr, gameCanStart, undoMovesChoice);
        playersList = UpdatePlayerNamesNewGame(whitePlayerDropDown, blackPlayerDropDown,
whitePlayerID, blackPlayerID, addWhitePlayer, addBlackPlayer, whitePlayerListIndex, blackPlayerListIndex,
database);
        break;
    case (int)UIModes.GamesList:
        games = UpdateGamesTable(gameListIndex, sortMode, sortOrder, database, selectedGame);
        //uses same positions as in player list so can use same rectangles
        UpdateLoadGameButtons(firstListButton, secondListButton, backButton);
        break;
    case (int)UIModes.ConfirmGame:
        UpdateConfirmGameMenu(startGameButton, backButton, selectedGame);
        break;
    case (int)UIModes.GameUI2D:
        int state = game.GetGamestate();

        UpdateBoard(game, textures);

```

```

        UpdateViewportControls(game, frontButton, topButton, sideButton);
        if (state == (int)Gamestates.PendingPromo) { UpdatePromoWindow(game, queenPromoRec,
rookPromoRec, bishopPromoRec, knightPromoRec, textures); }
        UpdateGameUI(backButton, moveListRec, game, undoMoveButton, game.GetIsUndoAllowed(),
viewmodeToggleButton);
        break;
        case (int)UIModes.PauseMenu:
            UpdatePauseMenu(resumeButton, exitMenuButton, exitDesktopButton, whiteForfeitButton,
blackForfeitButton, mutualAgreementStalemateButton, game.GetGamestate());
            break;
        case (int)UIModes.ConfirmForfeitStalemate:
            UpdateConfirmForfeitStalemate(startGameButton, backButton, proposedOutcome);
            break;
        case (int)UIModes.GameUI3D:
            Raylib.UpdateCamera(ref camera, CameraMode.CAMERA_ORBITAL);
            Raylib.BeginMode3D(camera);
            Update3DRepresentation(CubePos, viewmodelWiresChoice, game, camera, textures);
            Raylib.EndMode3D();
            Update3DRepresentationControls(viewmodeToggleButton, wiremodeToggleButton,
viewmodelWiresChoice);
            break;
    }

    Raylib.EndDrawing();
}

//unload textures
for(int i = 0; i < textures.Count(); i++)
{
    Raylib.UnloadTexture(textures[i]);
}
//unload sfx
Raylib.UnloadSound(emptySquare);
Raylib.UnloadSound(captureSquare);

Raylib.CloseWindow();
}
static void UpdateMainMenu(Vector2 textPos, Rectangle exit, Rectangle play, Rectangle player)
{
    Raylib.DrawText("Three Dimensional Chess", (int)textPos.X, (int)textPos.Y, 50, Color.BLACK);
    Raylib.DrawRectangleLinesEx(exit, 1, Color.BLACK);
    Raylib.DrawText("Exit", (int)exit.X + 45, (int)exit.Y + 30, 40, Color.BLACK);
    Raylib.DrawRectangleLinesEx(play, 1, Color.BLACK);
    Raylib.DrawText("Play", (int)play.X + 45, (int)play.Y + 30, 40, Color.BLACK);
    Raylib.DrawRectangleLinesEx(player, 1, Color.BLACK);
    Raylib.DrawText("Players", (int)player.X + 5, (int)player.Y + 30, 40, Color.BLACK);
}

static List<Player> UpdatePlayersTable(int startIndex, DatabaseHandler db, string sortMode, int sortOrder,
int selectedPlayerID)
{
    //Get a list of all players
    List<Player> players = db.GetPlayers();
    Sorter sorter = new Sorter();

```

```

switch (sortMode)
{
    case "name":
        players = sorter.MergeSortString(players);
        break;
    case "date":
        players = sorter.MergeSortDate(players);
        break;
    case "winRatio":
        players = sorter.MergeSortWins(players);
        break;
    case "whiteWR":
        players = sorter.MergeSortWhiteWR(players);
        break;
    case "blackWR":
        players = sorter.MergeSortBlackWR(players);
        break;
}
if(sortOrder == 1)
{
    players = sorter.Reverse(players);
}
//setup rectangles to be transformed
Rectangle baseRec = new Rectangle(10, 10, 790, 50);

for (int y = 0; y < (660 / 50); y++)
{
    Raylib.DrawRectangleLinesEx(new Rectangle(baseRec.X, baseRec.Y + (y * 50), baseRec.Width,
baseRec.Height), 1, Color.BLACK);
    //fill in with values from table
    if (y > 0 && startIndex + y - 1 < players.Count()) {

        Player tmp = players[startIndex + y - 1];
        if(tmp.GetID() == selectedPlayerID)
        {
            Raylib.DrawRectangle((int)baseRec.X, (int)baseRec.Y + (y*50), (int)baseRec.Width,
(int)baseRec.Height, Color.YELLOW);
        }
        //using position values from columns drawn above
        string playerName = tmp.GetName();
        //shorten name if it would write over column
        if(playerName.Length > 10){ playerName = playerName.Substring(0, 10);}
        Raylib.DrawText(playerName, 15, 23 + (y * 50), 30, Color.BLACK);
        DateOnly joinDate = tmp.GetJoinDate();
        Raylib.DrawText(joinDate.ToString().Substring(0, joinDate.ToString().Length - 5), 205, 23 + (y * 50),
30, Color.BLACK);
        string winRatio = tmp.GetTotalWins() + "/" + tmp.GetTotalLosses() + "/" + tmp.GetDraws();
        Raylib.DrawText(winRatio, 305, 23 + (y * 50), 30, Color.BLACK);
        string whiteWR = String.Format("{0:0.00}", tmp.GetWhiteWinrate()) + "%";
        string blackWR = String.Format("{0:0.00}", tmp.GetBlackWinrate()) + "%";
        Raylib.DrawText(whiteWR, 505, 23 + (y * 50), 30, Color.BLACK);
        Raylib.DrawText(blackWR, 655, 23 + (y*50), 30, Color.BLACK);
    }
}

```

```

//draw columns in here
Raylib.DrawLine(200, 10, 200, 660, Color.BLACK);
Raylib.DrawText("Name", 15, 23, 30, Color.BLACK);
Raylib.DrawLine(300, 10, 300, 660, Color.BLACK);
Raylib.DrawText("Date", 205, 23, 30, Color.BLACK);
Raylib.DrawLine(500, 10, 500, 660, Color.BLACK);
Raylib.DrawText("W/L/D", 305, 23, 30, Color.BLACK);
Raylib.DrawLine(650, 10, 650, 660, Color.BLACK);
Raylib.DrawText("White WR", 505, 23, 30, Color.BLACK);
Raylib.DrawText("Black WR", 655, 23, 30, Color.BLACK);
// draw outline
Raylib.DrawRectangleLines(10, 10, 790, 50 + (12 * 50), Color.BLACK);

return players;
}

static void UpdatePlayersListButtons(Rectangle func, Rectangle create, Rectangle back, string funcText)
{
    Raylib.DrawRectangleLinesEx(func, 1, Color.BLACK);
    Raylib.DrawText(funcText, (int)func.X + 25, (int)func.Y + 25, 30, Color.BLACK);
    Raylib.DrawRectangleLinesEx(create, 1, Color.BLACK);
    Raylib.DrawText("Add Player", (int)create.X + 10, (int)create.Y + 25, 30, Color.BLACK);
    Raylib.DrawRectangleLinesEx(back, 1, Color.BLACK);
    //draw a little back icon triangle
    Raylib.DrawTriangle(new Vector2(back.X + 70, back.Y + 5), new Vector2(back.X + 5, back.Y + (75 / 2)),
new Vector2(back.X + 70, back.Y + 70), Color.BLACK);
}

static List<GameInfo> UpdateGamesTable(int startIndex, string sortMode, int sortOrder, DatabaseHandler
db, GameInfo selectedGame)
{
    List<GameInfo> games = db.GetGames();
    Sorter sorter = new Sorter();
    switch (sortMode)
    {
        case "name":
            games = sorter.MergeSortName(games);
            break;
        case "date":
            games = sorter.MergeSortDate(games);
            break;
        case "gamestate":
            games = sorter.MergeSortState(games);
            break;
        case "white":
            games = sorter.MergeSortWhitePlayerName(games);
            break;
        case "black":
            games = sorter.MergeSortBlackPlayerName(games);
            break;
    }
    if(sortOrder == 1) { games = sorter.Reverse(games); }
    //setup rectangles to be transformed
    Rectangle baseRec = new Rectangle(10, 10, 790, 50);

```

```

    for (int y = 0; y < (660 / 50); y++)
    {
        Raylib.DrawRectangleLinesEx(new Rectangle(baseRec.X, baseRec.Y + (y * 50), baseRec.Width,
baseRec.Height), 1, Color.BLACK);
        //fill in with values from table
        if (y > 0 && startIndex + y - 1 < games.Count())
        {
            GameInfo tmp = games[startIndex + y - 1];
            //if game is equal to one selected tint it
            if (selectedGame != null && tmp.GetGameID() == selectedGame.GetGameID())
            {
                Raylib.DrawRectangle((int)baseRec.X, (int)baseRec.Y + (y * 50), (int)baseRec.Width,
(int)baseRec.Height, Color.YELLOW);
            }
            //using position values from columns drawn above
            string gameName = tmp.GetName();
            //shorten name if it would write over column
            if (gameName.Length > 10) { gameName = gameName.Substring(0, 10); }
            Raylib.DrawText(gameName, 15, 23 + (y * 50), 30, Color.BLACK);
            DateOnly lastAccessed = DateOnly.FromDateTime(tmp.GetLastAccessed());
            Raylib.DrawText(lastAccessed.ToString().Substring(0, lastAccessed.ToString().Length - 5), 205, 23 + (y
* 50), 30, Color.BLACK);
            string state = tmp.GetGamestate();
            Raylib.DrawText(state, 305, 23 + (y * 50), 30, Color.BLACK);
            string white = db.GetPlayer(tmp.GetWhitePlayerID()).GetName();
            if (white.Length > 10) { white = white.Substring(0, 10); }
            string black = db.GetPlayer(tmp.GetBlackPlayerID()).GetName();
            if (black.Length > 10) { black = black.Substring(0, 10); }
            Raylib.DrawText(white, 455, 23 + (y * 50), 30, Color.BLACK);
            Raylib.DrawText(black, 625, 23 + (y * 50), 30, Color.BLACK);
        }
    }
    //put here so yellow rectangle is behind
    //draw columns in here
    Raylib.DrawLine(200, 10, 200, 660, Color.BLACK);
    Raylib.DrawText("Name", 15, 23, 30, Color.BLACK);
    Raylib.DrawLine(300, 10, 300, 660, Color.BLACK);
    Raylib.DrawText("Date", 205, 23, 30, Color.BLACK);
    Raylib.DrawLine(450, 10, 450, 660, Color.BLACK);
    Raylib.DrawText("State", 305, 23, 30, Color.BLACK);
    Raylib.DrawLine(620, 10, 620, 660, Color.BLACK);
    Raylib.DrawText("White", 455, 23, 30, Color.BLACK);
    Raylib.DrawText("Black", 625, 23, 30, Color.BLACK);
    // draw outline
    Raylib.DrawRectangleLines(10, 10, 790, 50 + (12 * 50), Color.BLACK);

    return games;
}
static void UpdateLoadGameButtons(Rectangle load, Rectangle delete, Rectangle back)
{
    Raylib.DrawRectangleLinesEx(load, 1, Color.BLACK);
    Raylib.DrawText("Load Game", (int)load.X + 5, (int)load.Y + 25, 30, Color.BLACK);
    Raylib.DrawRectangleLinesEx(delete, 1, Color.BLACK);
    Raylib.DrawText("Delete Game", (int)delete.X + 5, (int)delete.Y + 25, 30, Color.BLACK);
}

```

```

Raylib.DrawRectangleLinesEx(back, 1, Color.BLACK);
//draw a little back icon triangle
Raylib.DrawTriangle(new Vector2(back.X + 70, back.Y + 5), new Vector2(back.X + 5, back.Y + (75 / 2)),
new Vector2(back.X + 70, back.Y + 70), Color.BLACK);
}

```

```

static void UpdateConfirmGameMenu(Rectangle confirm, Rectangle back, GameInfo info)
{
    Raylib.DrawRectangleLinesEx(confirm, 1, Color.BLACK);
    Raylib.DrawText("Confirm", (int)confirm.X + 70, (int)confirm.Y + 27, 40, Color.BLACK);
    string lastMove = info.GetMoves()[info.GetMoves().Count() - 1];
    Raylib.DrawText("Last Move: " + lastMove, (int)confirm.X - 40, (int)confirm.Y - 100, 40, Color.BLACK);
    Raylib.DrawRectangleLinesEx(back, 1, Color.BLACK);
    //draw a little back icon triangle
    Raylib.DrawTriangle(new Vector2(back.X + 70, back.Y + 5), new Vector2(back.X + 5, back.Y + (75 / 2)),
new Vector2(back.X + 70, back.Y + 70), Color.BLACK);
}

```

```

static void UpdateCreatePlayer(string inp, Rectangle entry, Rectangle doneButton, Rectangle back)
{
    Raylib.DrawText("Enter player name:", (int)entry.X, (int)entry.Y - 31, 30, Color.BLACK);
    Raylib.DrawRectangleLinesEx(entry, 1, Color.BLACK);
    //show input text as user types it
    Raylib.DrawText(inp, (int)entry.X + 5, (int)entry.Y + 40, 40, Color.BLACK);
    Raylib.DrawRectangleLinesEx(doneButton, 1, Color.BLACK);
    Raylib.DrawText("Confirm", (int)doneButton.X + 70, (int)doneButton.Y + 27, 40, Color.BLACK);
    Raylib.DrawRectangleLinesEx(back, 1, Color.BLACK);
    Raylib.DrawTriangle(new Vector2(back.X + 70, back.Y + 5), new Vector2(back.X + 5, back.Y + (75 / 2)),
new Vector2(back.X + 70, back.Y + 70), Color.BLACK);
}

```

```

static void UpdateNewLoadButtons(Rectangle create, Rectangle load, Rectangle back)
{
    Raylib.DrawRectangleLinesEx(create, 1, Color.BLACK);
    Raylib.DrawText("New Game", (int)create.X + 25, (int)create.Y + 23, 30, Color.BLACK);
    Raylib.DrawRectangleLinesEx(load, 1, Color.BLACK);
    Raylib.DrawText("Load Game", (int)load.X + 25, (int)load.Y + 23, 30, Color.BLACK);
    Raylib.DrawRectangleLinesEx(back, 1, Color.BLACK);
    Raylib.DrawTriangle(new Vector2(back.X + 70, back.Y + 5), new Vector2(back.X + 5, back.Y + (75 / 2)),
new Vector2(back.X + 70, back.Y + 70), Color.BLACK);
}

```

```

static void UpdateNewGameButtons(Rectangle undoMoves, Rectangle entry, Rectangle start, Rectangle
back, string inp, bool canStart, bool undoMovesChoice)
{
    Raylib.DrawRectangleLinesEx(undoMoves, 1, Color.BLACK);
    Raylib.DrawText("Undo Moves?", (int)undoMoves.X, (int)undoMoves.Y + (int)undoMoves.Height + 2, 30,
Color.BLACK);
    if(undoMovesChoice)
    {
        //show that the box has been selected
        Raylib.DrawLine((int)undoMoves.X, (int)undoMoves.Y, (int)undoMoves.X + 75, (int)undoMoves.Y + 75,
Color.BLACK);
        Raylib.DrawLine((int)undoMoves.X, (int)undoMoves.Y + 75, (int)undoMoves.X + 75, (int)undoMoves.Y,
Color.BLACK);
    }
}

```



```

    }
    Raylib.DrawRectangleLinesEx(entry, 1, Color.BLACK);
    Raylib.DrawText("Enter game name:", (int)entry.X + 120, (int)entry.Y - 31, 30, Color.BLACK);
    //show input text as user types it
    Raylib.DrawText(inp, (int)entry.X + 5, (int)entry.Y + 40, 40, Color.BLACK);
    Color startGameCol = Color.BLACK;
    if (!canStart)
    {
        startGameCol = Color.GRAY;
    }
    Raylib.DrawRectangleLinesEx(start, 1, startGameCol);
    Raylib.DrawText("Start Game", (int)start.X + 40, (int)start.Y + 27, 40, startGameCol);
    Raylib.DrawRectangleLinesEx(back, 1, Color.BLACK);
    Raylib.DrawTriangle(new Vector2(back.X + 70, back.Y + 5), new Vector2(back.X + 5, back.Y + (75 / 2)),
new Vector2(back.X + 70, back.Y + 70), Color.BLACK);
}

static List<Player> UpdatePlayerNamesNewGame(Rectangle whiteBase, Rectangle blackBase, int
whitePlayer, int blackPlayer, Rectangle addWhite, Rectangle addBlack, int whiteListIndex, int blackListIndex,
DatabaseHandler db)
{
    List<Player> players = db.GetPlayers();
    Raylib.DrawRectangleLinesEx(whiteBase, 1, Color.BLACK);
    Raylib.DrawText("Select white player:", (int)whiteBase.X + 2, (int)whiteBase.Y - 31, 30, Color.BLACK);
    //if a player has been selected write their name in
    if(whitePlayer > 0)
    {
        Player p = db.GetPlayer(whitePlayer);
        Raylib.DrawText(p.GetName(), 15, 10 + (int)whiteBase.Y, 30, Color.BLACK);
    }
    int limit = 4;
    if(players.Count() < limit) { limit = players.Count(); }
    //drop down menu here
    if(whiteListIndex != -1)
    {
        for(int y = 0; y < limit; y++)
        {
            if(y + whiteListIndex < players.Count())
            {
                string playerName = players[y + whiteListIndex].GetName();
                if(playerName.Length > 15) { playerName = playerName.Substring(0, 15); }
                Raylib.DrawRectangleLinesEx(new Rectangle(whiteBase.X, whiteBase.Y + 50 + (y*50),
whiteBase.Width, whiteBase.Height), 1, Color.BLACK);
                Raylib.DrawText(playerName, 15, (int)whiteBase.Y + 55 + (y * 50), 30, Color.BLACK);
            }
        }
    }

    //now repeat for black
    Raylib.DrawRectangleLinesEx(blackBase, 1, Color.BLACK);
    Raylib.DrawText("Select black player:", (int)blackBase.X - 6, (int)blackBase.Y - 31, 30, Color.BLACK);
    if(blackPlayer > 0)
    {
        Player b = db.GetPlayer(blackPlayer);
        Raylib.DrawText(b.GetName(), (int)blackBase.X + 5, (int)blackBase.Y + 10, 30, Color.BLACK);
    }
}

```

```

    }
    if(players.Count() < limit) { limit = players.Count(); }
    if(blackListIndex != -1)
    {
        for(int y = 0; y < limit; y++)
        {
            if(y + blackListIndex < players.Count())
            {
                string playerName = players[y + blackListIndex].GetName();
                if(playerName.Length > 15) { playerName = playerName.Substring(0, 15); }
                Raylib.DrawRectangleLinesEx(new Rectangle(blackBase.X, blackBase.Y + 50 + (y * 50),
blackBase.Width, blackBase.Height), 1, Color.BLACK);
                Raylib.DrawText(playerName, (int)blackBase.X + 5, (int)blackBase.Y + 55 + (y * 50), 30,
Color.BLACK);
            }
        }
    }

    //draw add player buttons
    Raylib.DrawRectangleLinesEx(addWhite, 1, Color.BLACK);
    Raylib.DrawLine((int)addWhite.X + ((int)addWhite.Width / 2), (int)addWhite.Y + 10, (int)addWhite.X +
((int)addWhite.Width / 2), (int)addWhite.Y + (int)addWhite.Height - 10, Color.BLACK);
    Raylib.DrawLine((int)addWhite.X + 10, (int)addWhite.Y + ((int)addWhite.Height / 2), (int)addWhite.X +
(int)addWhite.Width - 10, (int)addWhite.Y + ((int)addWhite.Height / 2), Color.BLACK);
    Raylib.DrawRectangleLinesEx(addBlack, 1, Color.BLACK);
    Raylib.DrawLine((int)addBlack.X + ((int)addBlack.Width / 2), (int)addBlack.Y + 10, (int)addBlack.X +
((int)addBlack.Width / 2), (int)addBlack.Y + (int)addBlack.Height - 10, Color.BLACK);
    Raylib.DrawLine((int)addBlack.X + 10, (int)addBlack.Y + ((int)addBlack.Height / 2), (int)addBlack.X +
(int)addBlack.Width - 10, (int)addBlack.Y + ((int)addBlack.Height / 2), Color.BLACK);
    return players;
}
static void UpdateBoard(Chess game, List<Texture2D> textures)
{
    int offset = UIConstants.squareSide;
    for (int y = 0; y < 8; y++)
    {
        for (int x = 0; x < 8; x++)
        {
            //grab the relevant cell from the viewport array in the game
            Square cell = game.GetViewportCell((y * 8) + x);
            //calculate the draw areas
            int xPos = UIConstants.boardXOrigin + (x * offset);
            int yPos = UIConstants.boardYOrigin - (y * offset);
            //draw filled if black square, draw outline if white square
            switch (cell.GetColour())
            {
                case (int)Colours.Black:
                    Raylib.DrawRectangle(xPos, yPos, offset, offset, Color.BLACK);
                    break;
                case (int)Colours.White:
                    Raylib.DrawRectangleLines(xPos, yPos, offset, offset, Color.BLACK);
                    break;
                case (int)Colours.BlackBlue:
                    Raylib.DrawRectangle(xPos, yPos, offset, offset, Color.DARKBLUE);
                    break;
            }
        }
    }
}

```

```

    case (int)Colours.WhiteBlue:
        Raylib.DrawRectangle(xPos, yPos, offset, offset, Color.BLUE);
        break;
    case (int)Colours.BlackRed:
        Raylib.DrawRectangle(xPos, yPos, offset, offset, Color.MAROON);
        break;
    case (int)Colours.WhiteRed:
        Raylib.DrawRectangle(xPos, yPos, offset, offset, Color.RED);
        break;
    case (int)Colours.BlackYellow:
        Raylib.DrawRectangle(xPos, yPos, offset, offset, Color.BLACK);
        Raylib.DrawRectangle(xPos, yPos, offset, offset, Raylib.Fade(Color.YELLOW, 0.8f));
        break;
    case (int)Colours.WhiteYellow:
        Raylib.DrawRectangle(xPos, yPos, offset, offset, Color.YELLOW);
        break;
}
if (cell.GetPiecePointer() != -1)
{
    Piece p = game.GetPieceDirect(cell.GetPiecePointer());

    string type = p.GetPieceType();
    int col = p.GetColour();
    if (col == (int)Colours.White)
    {
        //draw each texture, referencing list using above indices, + 4 to each pos to centre 60x60
texture
        switch (type)
        {
            case "P":
                Raylib.DrawTexture(textures[0], xPos + 4, yPos + 4, Color.WHITE);
                break;
            case "R":
                Raylib.DrawTexture(textures[1], xPos + 4, yPos + 4, Color.WHITE);
                break;
            case "N":
                Raylib.DrawTexture(textures[2], xPos + 4, yPos + 4, Color.WHITE);
                break;
            case "B":
                Raylib.DrawTexture(textures[3], xPos + 4, yPos + 4, Color.WHITE);
                break;
            case "Q":
                Raylib.DrawTexture(textures[4], xPos + 4, yPos + 4, Color.WHITE);
                break;
            case "K":
                Raylib.DrawTexture(textures[5], xPos + 4, yPos + 4, Color.WHITE);
                break;
        }
    }
    else
    {
        switch (type) {
            case "P":
                Raylib.DrawTexture(textures[6], xPos + 4, yPos + 4, Color.WHITE);
                break;

```

```

        case "R":
            Raylib.DrawTexture(textures[7], xPos + 4, yPos + 4, Color.WHITE);
            break;
        case "N":
            Raylib.DrawTexture(textures[8], xPos + 4, yPos + 4, Color.WHITE);
            break;
        case "B":
            Raylib.DrawTexture(textures[9], xPos + 4, yPos + 4, Color.WHITE);
            break;
        case "Q":
            Raylib.DrawTexture(textures[10], xPos + 4, yPos + 4, Color.WHITE);
            break;
        case "K":
            Raylib.DrawTexture(textures[11], xPos + 4, yPos + 4, Color.WHITE);
            break;
    }
}
}
}
}
}

```

`static void` UpdateViewportControls(Chess game, Rectangle frontButton, Rectangle topButton, Rectangle sideButton)

```

{
    //draw control triangles
    //up triangle
    Color upCol = Color.BLACK;
    if(game.GetViewLayer() == 8) { upCol = Color.GRAY; }
    Raylib.DrawTriangle(new Vector2(360, 570), new Vector2(330, 600), new Vector2(390, 600), upCol);
    //down triangle
    Color downCol = Color.BLACK;
    if(game.GetViewLayer() == 1) { downCol = Color.GRAY; }
    Raylib.DrawTriangle(new Vector2(610, 570), new Vector2(640, 600), new Vector2(670, 570), downCol);

    //draw view buttons
    //front view button
    Raylib.DrawRectangleLinesEx(frontButton, 1, Color.BLACK);
    Raylib.DrawText("Front", 68, 35, 30, Color.BLACK);
    //top view button
    Raylib.DrawRectangleLinesEx(topButton, 1, Color.BLACK);
    Raylib.DrawText("Top", 68, 120, 30, Color.BLACK);
    //side view button
    Raylib.DrawRectangleLinesEx(sideButton, 1, Color.BLACK);
    Raylib.DrawText("Side", 68, 205, 30, Color.BLACK);

    //draw text to show 3d coords
    string coordText = "X: x, Y: x, Z: x";
    switch (game.GetViewDirection())
    {
        case (int)ViewDirections.Front:
            coordText = coordText.Substring(0, 15) + game.GetViewLayer();
            break;
        case (int)ViewDirections.Side:

```

```

        coordText = coordText.Substring(0, 3) + game.GetViewLayer() + coordText.Substring(4);
        break;
    case (int)ViewDirections.Top:
        coordText = coordText.Substring(0, 8) + game.GetViewLayer() + coordText.Substring(9);
        break;
    }
    Raylib.DrawText(coordText, 395, 575, 30, Color.BLACK);
}

static void UpdatePromoWindow(Chess game, Rectangle queenRec, Rectangle rookRec, Rectangle
bishopRec, Rectangle knightRec, List<Texture2D> textures)
{
    //draw cells for promotion - borders make collisions clear
    Rectangle sourceRec = new Rectangle(0, 0, 60, 60); // for loading texture from source

    Raylib.DrawRectangleLinesEx(queenRec, 1, Color.BLACK);
    Raylib.DrawRectangleLinesEx(rookRec, 1, Color.BLACK);
    Raylib.DrawRectangleLinesEx(bishopRec, 1, Color.BLACK);
    Raylib.DrawRectangleLinesEx(knightRec, 1, Color.BLACK);

    //draw relevant player's pieces
    if (game.GetCurrentPlayer().GetColour() == 0)
    {
        Raylib.DrawTexturePro(textures[(int)PieceTextureIndices.BlackQueen], sourceRec, queenRec, new
Vector2(0, 0), 0, Color.WHITE);
        Raylib.DrawTexturePro(textures[(int)PieceTextureIndices.BlackRook], sourceRec, rookRec, new
Vector2(0, 0), 0, Color.WHITE);
        Raylib.DrawTexturePro(textures[(int)PieceTextureIndices.BlackBishop], sourceRec, bishopRec, new
Vector2(0, 0), 0, Color.WHITE);
        Raylib.DrawTexturePro(textures[(int)PieceTextureIndices.BlackKnight], sourceRec, knightRec, new
Vector2(0, 0), 0, Color.WHITE);
        // this else case should only arise if player is white but doesn't hurt to check
        else if (game.GetCurrentPlayer().GetColour() == 1)
        {
            Raylib.DrawTexturePro(textures[(int)PieceTextureIndices.WhiteQueen], sourceRec, queenRec, new
Vector2(0, 0), 0, Color.WHITE);
            Raylib.DrawTexturePro(textures[(int)PieceTextureIndices.WhiteRook], sourceRec, rookRec, new
Vector2(0, 0), 0, Color.WHITE);
            Raylib.DrawTexturePro(textures[(int)PieceTextureIndices.WhiteBishop], sourceRec, bishopRec, new
Vector2(0, 0), 0, Color.WHITE);
            Raylib.DrawTexturePro(textures[(int)PieceTextureIndices.WhiteKnight], sourceRec, knightRec, new
Vector2(0, 0), 0, Color.WHITE);
        }

        Raylib.DrawText("Promotion", 40, 467, 30, Color.BLACK);
    }
}

static void UpdateGameUI(Rectangle pause, Rectangle moveList, Chess game, Rectangle undo, bool
undoEnabled, Rectangle viewmode)
{
    //draw pause button
    Raylib.DrawRectangleLinesEx(pause, 1, Color.BLACK);
    Raylib.DrawRectangle((int)pause.X + 18, (int)pause.Y + 10, 15, (int)pause.Height - 20, Color.BLACK);
    Raylib.DrawRectangle((int)pause.X + (int)pause.Width - 32, (int)pause.Y + 10, 15, (int)pause.Height - 20,
Color.BLACK);
}

```

```

//draw toggle button
Raylib.DrawRectangleLinesEx(viewmode, 1, Color.BLACK);
Raylib.DrawText("3D", (int)viewmode.X + 10, (int)viewmode.Y + 15, 50, Color.BLACK);

//draw moveList
Raylib.DrawRectangleLinesEx(moveList, 1, Color.BLACK);
Stack<string> moves = game.GetMoveList();
int numMoves = moves.Count();
int ctr = 0;
while (!moves.IsEmpty() && ctr < 17)
{
    string moveStr = (numMoves - ctr) + ". " + moves.Pop();
    switch((numMoves - ctr)%2)
    {
        case 1:
            Raylib.DrawRectangleLines(960, (int)moveList.Y + (ctr * 30)+2, 20, 20, Color.BLACK);
            break;
        case 0:
            Raylib.DrawRectangle(960, (int)moveList.Y + (ctr * 30)+3, 20, 20, Color.BLACK);
            break;
    }
    Raylib.DrawText(moveStr, (int)moveList.X + 4, (int)moveList.Y + (ctr * 30)+3, 20, Color.BLACK);
    ctr++;
}

//draw undo moves button
if (undoEnabled)
{
    Raylib.DrawRectangleLinesEx(undo, 1, Color.BLACK);
    Raylib.DrawText("Undo Move", (int)undo.X + 20, (int)undo.Y + 12, 30, Color.BLACK);
}

//draw text when there's a winner
switch (game.GetGamestate())
{
    case (int)Gamestates.WhiteW:
        string playerName = game.GetWhitePlayerName();
        Raylib.DrawText(playerName + " Wins!", 350, 630, 30, Color.BLACK);
        break;
    case (int)Gamestates.BlackW:
        playerName = game.GetBlackPlayerName();
        Raylib.DrawText(playerName + " Wins!", 350, 630, 30, Color.BLACK);
        break;
    case (int)Gamestates.Stalemate:
        Raylib.DrawText("Stalemate", 425, 630, 30, Color.BLACK);
        break;
    default:
        playerName = game.GetCurrentPlayer().GetName();
        Raylib.DrawText(playerName + "'s Turn", 350, 615, 30, Color.BLACK);
        if (game.GetInCheck())
        {
            Raylib.DrawText(playerName + " in check", 350, 645, 30, Color.BLACK);
        }
        break;
}

```

```

}

static void UpdatePauseMenu(Rectangle resume, Rectangle exitMenu, Rectangle exitDesktop, Rectangle
whiteForf, Rectangle blackForf, Rectangle stalemate, int state)
{
    Raylib.DrawRectangleLinesEx(resume, 1, Color.BLACK);
    Raylib.DrawText("Resume", (int)resume.X + 95, (int)resume.Y + 25, 30, Color.BLACK);
    Raylib.DrawRectangleLinesEx(exitMenu, 1, Color.BLACK);
    Raylib.DrawText("Exit To Menu", (int)exitMenu.X + 50, (int)exitMenu.Y + 25, 30, Color.BLACK);
    Raylib.DrawRectangleLinesEx(exitDesktop, 1, Color.BLACK);
    Raylib.DrawText("Exit To Desktop", (int)exitDesktop.X + 30, (int)exitDesktop.Y + 25, 30, Color.BLACK);
    if(state == (int)Gamestates.Ongoing)
    {
        Raylib.DrawRectangleLinesEx(whiteForf, 1, Color.BLACK);
        Raylib.DrawText("White Forfeit", (int)whiteForf.X + 50, (int)whiteForf.Y + 25, 30, Color.BLACK);
        Raylib.DrawRectangleLinesEx(blackForf, 1, Color.BLACK);
        Raylib.DrawText("Black Forfeit", (int)blackForf.X + 50, (int)blackForf.Y + 25, 30, Color.BLACK);
        Raylib.DrawRectangleLinesEx(stalemate, 1, Color.BLACK);
        Raylib.DrawText("Agree To Draw", (int)stalemate.X + 30, (int)stalemate.Y + 25, 30, Color.BLACK);
    }
}

static void UpdateConfirmForfeitStalemate(Rectangle confirm, Rectangle back, int proposedOutcome)
{
    Raylib.DrawRectangleLinesEx(confirm, 1, Color.BLACK);
    Raylib.DrawText("Confirm", (int)confirm.X + 70, (int)confirm.Y + 27, 40, Color.BLACK);
    Raylib.DrawRectangleLinesEx(back, 1, Color.BLACK);
    Raylib.DrawTriangle(new Vector2(back.X + 70, back.Y + 5), new Vector2(back.X + 5, back.Y + (75 / 2)),
new Vector2(back.X + 70, back.Y + 70), Color.BLACK);
    string context = "";
    switch (proposedOutcome)
    {
        case (int)Gamestates.Stalemate:
            context = "Agree To Draw?";
            break;
        case (int)Gamestates.WhiteW:
            context = "Black Forfeit?";
            break;
        case (int)Gamestates.BlackW:
            context = "White Forfeit?";
            break;
    }
    Raylib.DrawText(context, (int)confirm.X + 5, (int)confirm.Y - 32, 30, Color.BLACK);
}

static void Update3DRepresentation(Vector3 cubePos, bool wiresSelected, Chess game, Camera3D camera,
List<Texture2D> textures)
{
    Vector3 baseCubePos = new Vector3(-3.5f, -3.5f, -3.5f);
    Rectangle sourceRec = new Rectangle(0, 0, 60, 60); // for loading texture from source
    Raylib.DrawCubeWires(cubePos, 8, 8, 8, Color.BLACK);
    for(int z = 0; z < 8; z++)
    {
        for (int x = 0; x < 8; x++)
        {

```

```

for (int y = 0; y < 8; y++)
{
    Vector3 tmpVect = new Vector3(baseCubePos.X + x, baseCubePos.Y + y, baseCubePos.Z + z);
    Square cell = game.GetCell(x + (y * 8) + (z * 64));
    if (wiresSelected) { Raylib.DrawCubeWires(tmpVect, 1, 1, 1, Color.BLACK); }
    if (cell.GetPiecePointer() != -1)
    {
        Piece p = game.GetPieceDirect(cell.GetPiecePointer());

        string type = p.GetPieceType();
        int col = p.GetColour();
        Texture2D pieceText = new Texture2D();
        if (col == (int)Colours.White)
        {
            //select each piece
            switch (type)
            {
                case "P":
                    pieceText = textures[0];
                    break;
                case "R":
                    pieceText = textures[1];
                    break;
                case "N":
                    pieceText = textures[2];
                    break;
                case "B":
                    pieceText = textures[3];
                    break;
                case "Q":
                    pieceText = textures[4];
                    break;
                case "K":
                    pieceText = textures[5];
                    break;
            }
        }
        else
        {
            switch (type)
            {
                case "P":
                    pieceText = textures[6];
                    break;
                case "R":
                    pieceText = textures[7];
                    break;
                case "N":
                    pieceText = textures[8];
                    break;
                case "B":
                    pieceText = textures[9];
                    break;
                case "Q":
                    pieceText = textures[10];
                    break;
            }
        }
    }
}

```



```

        break;
    case "K":
        pieceText = textures[11];
        break;
    }
}
//draw piece onto 3d board here
Raylib.DrawBillboardRec(camera, pieceText, sourceRec, tmpVect, new Vector2(1f, 1f),
Color.WHITE);
}
//now draw colour of cell
Color cellCol = Color.BLANK;
switch (cell.GetColour())
{
    case (int)Colours.WhiteBlue: cellCol = Color.BLUE; break;
    case (int)Colours.BlackBlue: cellCol = Color.DARKBLUE; break;
    case (int)Colours.WhiteRed: cellCol = Color.RED; break;
    case (int)Colours.BlackRed: cellCol = Color.MAROON; break;
    case (int)Colours.WhiteYellow: cellCol = Color.YELLOW; break; // NOTE: need to fade this
    case (int)Colours.BlackYellow: cellCol = Color.YELLOW; break;
}
if (!cellCol.Equals(Color.BLANK))
{
    Raylib.DrawCube(tmpVect, 1f, 1f, 1f, Raylib.Fade(cellCol, 0.9f));
    if (!wiresSelected) { Raylib.DrawCubeWires(tmpVect, 1, 1, 1, Color.BLACK); }
}
}
}
}
}
static void Update3DRepresentationControls(Rectangle viewmodeToggle, Rectangle wireToggle, bool
wireChoice)
{
    //draw reverse toggle button
    Raylib.DrawRectangleLinesEx(viewmodeToggle, 1, Color.BLACK);
    Raylib.DrawText("2D", (int)viewmodeToggle.X + 10, (int)viewmodeToggle.Y + 15, 50, Color.BLACK);
    //draw wire toggle button
    Raylib.DrawRectangleLinesEx(wireToggle, 1, Color.BLACK);
    Raylib.DrawText("Enable Wires", (int)wireToggle.X, (int)wireToggle.Y - 31, 30, Color.BLACK);
    if (wireChoice)
    {
        Raylib.DrawLine((int)wireToggle.X, (int)wireToggle.Y, (int)wireToggle.X + (int)wireToggle.Width,
(int)wireToggle.Y + (int)wireToggle.Height, Color.BLACK);
        Raylib.DrawLine((int)wireToggle.X, (int)wireToggle.Y + (int)wireToggle.Height, (int)wireToggle.X +
(int)wireToggle.Width, (int)wireToggle.Y, Color.BLACK);
    }
}
}

```

### Chess.cs

```

using System;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

```

```

namespace ThreeDimensionalChess
{
    static class Constants
    {
        //initialise constants
        //board is always cubic, initialise side length as such
        public const int boardDimensions = 8;
        //resulting size of cubic board
        public const int boardSize = boardDimensions * boardDimensions * boardDimensions;
    }

    enum Gamestates
    {
        Stalemate, // 0
        WhiteW, // 1 - White wins by black being checkmated
        BlackW, // 2 - Black wins by white being checkmated
        Ongoing, // 3
        PendingPromo // 4 - Used to wait program for pawn promotion
    }

    enum ViewDirections
    {
        Front, //0
        Side, //1
        Top // 2
    }

    class Chess
    {
        public int playerTurn;
        private bool inCheck;
        private bool changeBoardDir;
        private bool undoMovesAllowed;
        private Board board;
        private Stack<string> moveList;
        private int pendingMove;
        private Player whitePlayer;
        private Player blackPlayer;
        //array of pointers that compose the 2D viewport
        private int[] viewport;
        private int viewDir;
        private int viewLayer;
        //extra stuff
        DatabaseHandler db;
        private int ID;
        private int state;

        public Chess(int whiteID, int blackID, string name, bool undoMoves)
        {
            //init variables to defaults - starting turn, starting check and gamestate & create board
            playerTurn = (int)Colours.White;
            board = new Board();
            inCheck = false;
            moveList = new Stack<string>();
            pendingMove = 0;
        }
    }
}

```

```

    db = new DatabaseHandler();
    //init player objects, grab name based on ID from db
    whitePlayer = db.GetPlayer(whiteID);
    whitePlayer.SetColour((int)Colours.White);
    blackPlayer = db.GetPlayer(blackID);
    blackPlayer.SetColour((int)Colours.Black);
    /*whitePlayer = new Player(0, "name", 0, 0, 0, 0, 0, DateTime.Today);
    whitePlayer.SetColour((int)Colours.White);
    blackPlayer = new Player(0, "name", 0, 0, 0, 0, 0, DateTime.Today);
    blackPlayer.SetColour((int)Colours.Black);*/
    state = (int)Gamestates.Ongoing;
    //create game in database
    ID = db.CreateGame(name, undoMoves, whiteID, blackID);
    //init games rules
    undoMovesAllowed = undoMoves;
    changeBoardDir = true;
    //init viewport (as front view white)
    viewLayer = 0;
    viewDir = 0;
    viewport = new int[64];
    UpdateViewport();
}

//constructor for loading games from GameInfo
public Chess(GameInfo info)
{
    playerTurn = (int)Colours.White;
    board = new Board();
    inCheck = false;
    moveList = new Stack<string>();
    pendingMove = 0;
    db = new DatabaseHandler();
    //load values from info obj
    ID = info.GetGameID();
    undoMovesAllowed = info.GetUndoMoves();
    state = info.GetGamestateAsInt();
    //init players
    whitePlayer = db.GetPlayer(info.GetWhitePlayerID());
    whitePlayer.SetColour((int)Colours.White);
    blackPlayer = db.GetPlayer(info.GetBlackPlayerID());
    blackPlayer.SetColour((int)Colours.Black);
    //init viewport (as front view white)
    changeBoardDir = true;
    viewLayer = 0;
    viewDir = 0;
    viewport = new int[64];
    UpdateViewport();
    //enact saved moves
    List<string> moves = info.GetMoves();
    while (moves.Count() > 0 && moves[0] != "")
    {
        //using list like a queue
        string tmp = moves.RemoveAt(0);
        if (!(tmp == "Mutual Agreement÷" || tmp == "White#" || tmp == "Black#")) { ParseMove(tmp); }
        else
    }

```

```

        {
            undoMovesAllowed = true;
            ID = db.CreateGame(info.GetName() + "-Copy", true, info.GetWhitePlayerID(),
info.GetBlackPlayerID());
        }
    }
    //if game is finished enabled undo moves
    if (!moveList.IsEmpty() && (moveList.Peek().Contains('#') || moveList.Peek().Contains('÷')))
    {
        ID = db.CreateGame(info.GetName() + "-Copy", true, info.GetWhitePlayerID(),
info.GetBlackPlayerID());
        undoMovesAllowed = true;
        UndoMove();
    }
    if (moveList.Peek() != null && moveList.Peek().Contains('+')) { inCheck = true; }
}

public void Click(int squareIndex)
{
    //can't do anything once game is over - separate method for rewind and view buttons
    if (playerTurn != -1 && state != (int)Gamestates.PendingPromo)
    {
        bool pieceSelected = board.IsPieceSelected();
        if (pieceSelected == true)
        {
            Piece selectedPiece = board.GetSelectedPiece();
            Piece pieceOnSquare = board.GetPiece(squareIndex);
            if (pieceOnSquare == null)
            {
                //moves onto square with selected piece if its empty
                AttemptMove(squareIndex);
            }
            else if (pieceOnSquare.GetColour() == playerTurn)
            {
                SelectPiece(squareIndex);
            }
            else
            {
                //Getting to this point already eliminates player selecting their own piece due to above
condition, failsafes in attemptmove anyway
                //attempts move if player has pieces of opposite colour selected
                if (selectedPiece.GetColour() != pieceOnSquare.GetColour())
                {
                    AttemptMove(squareIndex);
                }
                else
                {
                    //this case should be only triggered when going from selecting one enemy piece to selecting
another enemy piece
                    SelectPiece(squareIndex);
                }
            }
        }
    }
}

```

```

        else
        {
            SelectPiece(squareIndex); //simple outcome if no piece selected
        }
    }
    UpdateViewport();
}

public void ViewportClick(int boardIndex)
{
    Click(viewport[boardIndex]);
    UpdateViewport();
}

private void AttemptMove(int squareIndex)
{
    //player turn is checked in board method
    string move = board.MovePiece(squareIndex, playerTurn);

    //if the returning notation is not null, then the move has been effected, next player's turn
    if (move != null)
    {
        playerTurn = (playerTurn + 1) % 2;
        //gamestate is evaluated at the start of a player's turn, will be unnoticeable to players but makes the
        maths easier
        inCheck = board.CheckCheck(playerTurn);
        EvalGamestate();
        //add check/stalemate/checkmate symbols here
        switch (state)
        {
            //lock undo once game is complete, if game is reopened from menu it opens a undoable copy
            case (int)Gamestates.Stalemate:
                //add ÷ symbol for stalemate
                move = move + "÷";
                undoMovesAllowed = false;
                whitePlayer.AddWhiteDraw();
                blackPlayer.AddBlackDraw();
                break;
            case (int)Gamestates.WhiteW:
                // add # symbol for checkmate
                move = move + "#";
                undoMovesAllowed = false;
                whitePlayer.AddWhiteWin();
                blackPlayer.AddBlackLoss();
                break;
            case (int)Gamestates.BlackW:
                // add # symbol for checkmate
                move = move + "#";
                undoMovesAllowed = false;
                blackPlayer.AddBlackWin();
                whitePlayer.AddWhiteLoss();
                break;
            default:
                //this is if game is ongoing but still need to append + if in check for movelist readability
                if (inCheck)

```

```

        {
            move = move + "+";
        }
        break;
    }
    if (move.Contains("="))
    {
        state = (int)Gamestates.PendingPromo;
        //reverse playerTurn change
        playerTurn = (playerTurn - 1) % 2;
        //holding square int here to ref piece down the line
        pendingMove = squareIndex;
        //select piece so that player is more aware of it
        SelectPiece(squareIndex);
        //push move if promo, just don't forget to pop it later
    }
    moveList.Push(move);
    db.UpdateGame(moveList.ConvertToString(), state, ID);
    db.UpdatePlayer(whitePlayer);
    db.UpdatePlayer(blackPlayer);
}

}

private void EvalGamestate()
{
    state = board.GetGamestate(playerTurn);
    //end game if gamestate isn't ongoing
    if (state != (int)Gamestates.Ongoing) { playerTurn = -1; }
}

public void PromotePawn(string pieceType)
{
    bool success = board.PromotePawn(pieceType, pendingMove);
    if (success)
    {
        pendingMove = -1;
        string move = moveList.Pop();
        move = move + pieceType;
        moveList.Push(move);
        playerTurn = (playerTurn + 1) % 2;
        state = (int)Gamestates.Ongoing;
        db.UpdateGame(moveList.ConvertToString(), state, ID);
    }
    UpdateViewport();
}

public void SetViewDirection(int mode)
{
    //if piece is selected, use that piece's relevant co-ord as layer value
    viewDir = mode;
    if (board.IsPieceSelected())
    {
        int[] pieceVect = board.GetSelectedPiece().GetCurrentPosAsVect();
        switch (mode)

```

```

    {
        case (int)ViewDirections.Front:
            viewLayer = pieceVect[2];
            break;
        case (int)ViewDirections.Side:
            viewLayer = pieceVect[0];
            break;
        case (int)ViewDirections.Top:
            viewLayer = pieceVect[1];
            break;
    }
}
else
{
    //default for side viewlayer is 7 because I just can't understand the game otherwise for some reason
    if(viewDir == (int)ViewDirections.Side) { viewLayer = 7; }
    else if(changeBoardDir)
    {
        //otherwise sets viewlayer to 0/7 depending on player turn - can be toggled as a gamerule
        if(playerTurn == 0) { viewLayer = 7; }
        else { viewLayer = 0; }
    }
}
UpdateViewport();
}

```

```

public void IncrementViewLayer() { if (viewLayer < 7) { viewLayer++; UpdateViewport(); } }
public void DecrementViewLayer() { if(viewLayer > 0) { viewLayer--; UpdateViewport(); } }

```

```

private void UpdateViewport()
{
    int originPointer;
    //grab pointers for relevant slice of board
    switch (viewDir)
    {
        case (int)ViewDirections.Front:
            // first transform origin by viewlayer
            originPointer = (viewLayer * 64);
            //then fill out viewport coords from origin pointer
            for(int y = 0; y < 8; y++)
            {
                for(int x = 0; x < 8; x++)
                {
                    viewport[(8 * y) + x] = originPointer + x + (y * 8);
                }
            }
            break;
        case (int)ViewDirections.Side:
            //transform origin by viewlayer
            originPointer = viewLayer; // is actually viewLayer * 1
            for (int y = 0; y < 8; y++)
            {
                for(int x = 0; x < 8; x++)
                {
                    //maps depth coordinates to x coordinates of viewport

```

```

        viewport[(8 * y) + x] = originPointer + (x * 64) + (y * 8);
    }
}
break;
case (int)ViewDirections.Top:
    //transform origin by viewlayer
    originPointer = (viewLayer * 8);
    for(int y = 0; y < 8; y++)
    {
        for(int x = 0; x < 8; x++)
        {
            //maps depth coordinates to y coordinates of viewport
            viewport[(8 * y) + x] = originPointer + x + (y * 64);
        }
    }
    break;
}
}

public Square GetViewportCell(int ptr)
{
    //takes a one dimensional pointer for the viewport
    //defensive programming
    if(ptr > -1 && ptr < 64)
    {
        return board.GetSquare(viewport[ptr]);
    }
    else { throw new ArgumentOutOfRangeException(); }
}

public int GetViewDirection()
{
    return viewDir;
}

public int GetViewLayer()
{
    //add one for user readability
    return viewLayer + 1;
}

private void SelectPiece(int squareIndex)
{
    board.SelectPiece(squareIndex, playerTurn);
}

//ONLY USE FOR LOADING GAME DATA - formatted by this program or otherwise assuming notation is
perfect
private void ParseMove(string m)
{
    //assumes move is valid
    moveList.Push(m);
    board.ParseMove(m, playerTurn);
    playerTurn = (playerTurn + 1) % 2;
    EvalGamestate();
}

```



```

}

public void UndoMove()
{
    if (!moveList.IsEmpty() && undoMovesAllowed)
    {
        string m = moveList.Pop();
        board.UndoMove(m);
        playerTurn = (playerTurn + 1) % 2;
        switch (state)
        {
            case (int)Gamestates.WhiteW:
                playerTurn = (int)Colours.White;
                break;
            case (int)Gamestates.BlackW:
                playerTurn = (int)Colours.Black;
                break;
            case (int)Gamestates.Stalemate:
                playerTurn = (moveList.Count() + 1) % 2;
                break;
        }
        EvalGamestate();
        db.UpdateGame(moveList.ConvertToString(), state, ID);
    }
}

//use this for handling accepted draws/forfeits
public void ManualEndGame(int outcome)
{
    switch (outcome)
    {
        {
            //lock undo once game is complete, if game is reopened from menu it opens a undoable copy
            case (int)Gamestates.Stalemate:
                //mutual agreement is the correct term for a agreed upon draw
                moveList.Push("Mutual Agreement÷");
                state = (int)Gamestates.Stalemate;
                undoMovesAllowed = false;
                whitePlayer.AddWhiteDraw();
                blackPlayer.AddBlackDraw();
                playerTurn = -1;
                break;
            case (int)Gamestates.WhiteW:
                // black forfeit
                moveList.Push("Black#");
                state = (int)Gamestates.WhiteW;
                undoMovesAllowed = false;
                whitePlayer.AddWhiteWin();
                blackPlayer.AddBlackLoss();
                playerTurn = -1;
                break;
            case (int)Gamestates.BlackW:
                // white forfeit
                moveList.Push("White#");
                state = (int)Gamestates.BlackW;
                undoMovesAllowed = false;

```

```

        blackPlayer.AddBlackWin();
        whitePlayer.AddWhiteLoss();
        playerTurn = -1;
        break;
    }
    db.UpdateGame(moveList.ConvertToString(), state, ID);
    db.UpdatePlayer(whitePlayer);
    db.UpdatePlayer(blackPlayer);
}

public Player GetCurrentPlayer()
{
    Player ret;
    if(playerTurn == 0) { ret = blackPlayer; }
    else { ret = whitePlayer; }
    return ret;
}

public Piece GetPieceDirect(int ptr)
{
    return board.GetPieceDirect(ptr);
}

public bool GetInCheck() { return inCheck; }
public int GetGamestate() { return state; }
public bool GetIsUndoAllowed() { return undoMovesAllowed; }

public Stack<string> GetMoveList() { return moveList.Clone(); }

public string GetWhitePlayerName() { return whitePlayer.GetName(); }
public string GetBlackPlayerName() { return blackPlayer.GetName(); }
public string GetLastMove() { return moveList.Peek(); }
public Square GetCell(int index) { return board.GetSquare(index); }
}
}

```

### Board.cs

```

using System;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ThreeDimensionalChess
{
    class Board
    {
        //one dimensional list used to store board - used modular arithmetic on vectors to navigate it
        private List<Square> board = new List<Square>();
        private List<Piece> pieces = new List<Piece>();
        //storage of selected piece and its moves
        private List<int> currentPossibleMoves = new List<int>();
        private int currentPieceIndex;
    }
}

```

```

private List<int> pointersMovedToFrom = new List<int>();

public Board(bool custom = false)
{
    int colourCtr = 0;
    //intialise all 512 squares of the 3D board
    for (int x = 0; x < Constants.boardSize; x++)
    {
        //ADD ONE FOR EACH NEW LINE
        if (x % 8 == 0 && x != 0) { colourCtr++; }
        //ADD ONE FOR EACH NEW BOARD
        if (x % 64 == 0 && x != 0) { colourCtr++; }
        //adds a new square to the board, performs mod on x to get colour, Colours enum is stored in
        Square.cs even squares are black(with 0 also being black), odd are white
        board.Add(new Square(colourCtr % 2));
        colourCtr++;
    }
    //always place kings so that they are first in list
    //only place pieces in default positions if not custom setup, otherwise force user to place kings first
    if (!custom)
    {
        AddPiece("K", 508, 0);
        AddPiece("K", 4, 1);
        //place the rest of the pieces in order of complexity (facilitates efficient stalemate checking later)
        //white pawns
        for (int i = 0; i < 2; i++)
        {
            for (int x = 0; x < Constants.boardDimensions; x++)
            {
                int pos = 8 + x + (64 * i);
                AddPiece("P", pos, 1);
            }
        }
        //black pawns
        for (int i = 0; i < 2; i++)
        {
            for (int x = 0; x < Constants.boardDimensions; x++)
            {
                int pos = 496 + x - (64 * i);
                AddPiece("P", pos, 0);
            }
        }
        //rooks
        AddPiece("R", 0, 1);
        AddPiece("R", 7, 1);
        AddPiece("R", 511, 0);
        AddPiece("R", 504, 0);
        //knights
        AddPiece("N", 1, 1);
        AddPiece("N", 6, 1);
        AddPiece("N", 510, 0);
        AddPiece("N", 505, 0);
        //bishops
        AddPiece("B", 2, 1);
        AddPiece("B", 5, 1);
    }
}

```

```

        AddPiece("B", 509, 0);
        AddPiece("B", 506, 0);
        //queens
        AddPiece("Q", 507, 0);
        AddPiece("Q", 3, 1);
    }
    currentPieceIndex = -1;
}

public bool IsPieceSelected()
{
    bool ret = false;
    if (currentPieceIndex != -1) { ret = true; }
    return ret;
}

public Piece GetSelectedPiece()
{
    return pieces[currentPieceIndex];
}

//takes a square index as input
public Piece GetPiece(int inp)
{
    Piece tmp = null;
    int ptr = board[inp].GetPiecePointer();
    //if there is a piece on square returns piece otherwise returns null
    if (ptr != -1) { tmp = pieces[ptr]; }
    return tmp;
}

public Piece GetPieceDirect(int ptr)
{
    Piece tmp = null;
    if(ptr != -1)
    {
        tmp = pieces[ptr];
    }
    else { throw new ArgumentOutOfRangeException(); }
    return tmp;
}

public void AddPiece(string type, int pos, int colour)
{
    Piece p = null;
    switch (type)
    {
        case "R":
            p = new Rook(pos, colour);
            break;
        case "P":
            p = new Pawn(pos, colour);
            break;
        case "K":
            p = new King(pos, colour);

```

```

        break;
    case "B":
        p = new Bishop(pos, colour);
        break;
    case "N":
        p = new Knight(pos, colour);
        break;
    case "Q":
        p = new Queen(pos, colour);
        break;
    case "TSP":
        p = new ThreatSuperPiece(pos, colour);
        break;
    }

    pieces.Add(p);

    board[pos].SetPiecePointer(pieces.Count() - 1);
}

//this needs to be triggered when a square is clicked on, parameter is index of square
public void SelectPiece(int squarePtr, int currentPlayer)
{
    //do nothing if square is empty
    //list is setup here in case needed
    int piecePtr = board[squarePtr].GetPiecePointer();
    List<int> moveList = new List<int>();
    if (piecePtr != -1)
    {
        //deselects piece if its already selected
        if (piecePtr != currentPieceIndex)
        {
            //clear colours on squares first
            for (int x = 0; x < currentPossibleMoves.Count(); x++)
            {
                board[currentPossibleMoves[x]].NotUnderThreat();
                if (pointersMovedToFrom.Contains(currentPossibleMoves[x])) {
board[currentPossibleMoves[x]].PieceMoved(); }
            }

            currentPieceIndex = piecePtr;
            moveList = pieces[piecePtr].GeneratePossibleMoves(board, pieces); //this only returns physically
possible moves

            //filter out null moves
            List<int> filteredMoves = new List<int>();
            for (int x = 0; x < moveList.Count(); x++)
            {
                //FILTER OUT SELF CHECK MOVES HERE
                //this may slow down everything a lot - create simulated board and try the move here
                SimulatedBoard tmp = new SimulatedBoard(board, pieces);
                bool legal = tmp.LegalMove(moveList[x], currentPieceIndex,
pieces[currentPieceIndex].GetColour());
                if (legal && moveList[x] != -1) { filteredMoves.Add(moveList[x]); }
            }

```

```

        currentPossibleMoves = filteredMoves;
        bool friendly = true;
        if (pieces[currentPieceIndex].GetColour() != currentPlayer) { friendly = false; }
        DisplayMoves(currentPossibleMoves, friendly);
    }
}
else
{
    //resets selection data
    currentPieceIndex = -1;
    for (int x = 0; x < currentPossibleMoves.Count(); x++)
    {
        board[currentPossibleMoves[x]].NotUnderThreat();
        if (pointersMovedToFrom.Contains(currentPossibleMoves[x])) {
board[currentPossibleMoves[x]].PieceMoved(); }
    }
    currentPossibleMoves = new List<int>();
}
}

//subroutine for moving piece, no need to take piece in parameter - since currently selected piece is already
//stored
public string MovePiece(int targetSquarePtr, int currentPlayer)
{
    //LA3DN move will be returned in this string
    string move = null;

    int startSquarePtr = pieces[currentPieceIndex].GetCurrentPosition();
    //target move should be in possible move list and also check that piece belongs to current player
    if (currentPossibleMoves.Contains(targetSquarePtr) && pieces[currentPieceIndex].GetColour() ==
currentPlayer)
    {
        //reset colours of last moves - make sure that list isn't empty first
        if (pointersMovedToFrom.Count() > 0)
        {
            board[pointersMovedToFrom[0]].NotUnderThreat();
            board[pointersMovedToFrom[1]].NotUnderThreat();
            pointersMovedToFrom.RemoveAt(0);
            pointersMovedToFrom.RemoveAt(0);
        }

        int targetPiecePtr = board[targetSquarePtr].GetPiecePointer();

        //store first half of move data here (origin point and piece type)
        string moveDataFirstHalf = pieces[currentPieceIndex].GetPieceType() +
pieces[currentPieceIndex].GetCurrentPosAsStr();

        //effects the move, shades squares yellow, store squares to reset them later
        string moveDataSecondHalf = pieces[currentPieceIndex].MovePiece(targetSquarePtr, board, pieces);
        board[startSquarePtr].SetPiecePointer(-1);
        board[startSquarePtr].PieceMoved();
        pointersMovedToFrom.Add(startSquarePtr);
    }
}

```

```

    //added logic here to prevent ptr misalignment - if a piece is taken and its pointer is less than the
    current piece, we need to offset current piece ptr by one
    if (moveDataSecondHalf.Contains("X"))
    {
        if (targetPiecePtr < currentPieceIndex) { currentPieceIndex--; }
        for (int x = targetPiecePtr; x < pieces.Count(); x++)
        {
            board[pieces[x].GetCurrentPosition()].DecrementPiecePointer();
        }
    }
    board[targetSquarePtr].SetPiecePointer(currentPieceIndex);
    board[targetSquarePtr].PieceMoved();
    pointersMovedToFrom.Add(targetSquarePtr);
    //resets selection data
    currentPieceIndex = -1;
    for (int x = 0; x < currentPossibleMoves.Count(); x++)
    {
        board[currentPossibleMoves[x]].NotUnderThreat();
        if (pointersMovedToFrom.Contains(currentPossibleMoves[x])) {
board[currentPossibleMoves[x]].PieceMoved(); }
    }
    currentPossibleMoves = new List<int>();
    //check if a pawn has reached end rank
    string promotion = CheckLastRank();

    //aggregate string parts
    move = moveDataFirstHalf + moveDataSecondHalf + promotion;

}
else
{
    SelectPiece(targetSquarePtr, currentPlayer);
}

return move;
}

public int GetGamestate(int currentPlayer)
{
    int ret = (int)Gamestates.Stalemate;

    for (int x = 0; x < pieces.Count(); x++)
    {
        if (pieces[x].GetColour() == currentPlayer)
        {
            List<int> moveList = pieces[x].GeneratePossibleMoves(board, pieces);
            //filter out null moves
            List<int> filteredMoves = new List<int>();
            for (int y = 0; y < moveList.Count(); y++)
            {
                if (moveList[y] != -1)
                {
                    //FILTER OUT SELF CHECK MOVES HERE
                    SimulatedBoard tmp = new SimulatedBoard(board, pieces);
                    bool legal = tmp.LegalMove(moveList[y], x, pieces[x].GetColour());

```

```

        if (legal) { filteredMoves.Add(moveList[y]); }
    }
}
//if this is greater than zero, games is neither stale nor checkmate
if (filteredMoves.Count() > 0)
{
    ret = (int)Gamestates.Ongoing;
    x = pieces.Count();
}
}
}
// if game has not been set to ongoing here, then it is either stale or checkmate
if (ret != (int)Gamestates.Ongoing)
{
    bool checkMate = CheckThreat(pieces[currentPlayer].GetCurrentPosition(), currentPlayer);
    if (checkMate && currentPlayer == (int)Colours.Black) { ret = (int)Gamestates.WhiteW; }
    else if (checkMate && currentPlayer == (int)Colours.White) { ret = (int)Gamestates.BlackW; }
}

return ret;
}

private void DisplayMoves(List<int> possibleMoves, bool friendly)
{
    for (int x = 0; x < possibleMoves.Count(); x++)
    {
        board[possibleMoves[x]].UnderThreat(friendly);
    }
}

private string CheckLastRank()
{
    string ret = "";
    //check promotion linearly
    for(int x = 0; x < 8; x++)
    {
        //white pieces
        int piecePtr = board[x + (7 * 64) + (7 * 8)].GetPiecePointer();
        if(piecePtr != -1 && pieces[piecePtr].GetPieceType() == "P" && pieces[piecePtr].GetColour() ==
(int)Colours.White)
        {
            ret = "=";
        }

        //black pieces
        piecePtr = board[x].GetPiecePointer();
        if(piecePtr != -1 && pieces[piecePtr].GetPieceType() == "P" && pieces[piecePtr].GetColour() ==
(int)Colours.Black)
        {
            ret = "=";
        }
    }
    return ret;
}

```



```
public bool PromotePawn(string pieceType, int squarePtr)
{
    int piecePtr = board[squarePtr].GetPiecePointer();
    bool success = false;
    ///reference piece in list to a new piece of promoted variant
    switch (pieceType)
    {
        case "Q":
            pieces[piecePtr] = new Queen(squarePtr, pieces[piecePtr].GetColour());
            success = true;
            break;
        case "B":
            pieces[piecePtr] = new Bishop(squarePtr, pieces[piecePtr].GetColour());
            success = true;
            break;
        case "R":
            pieces[piecePtr] = new Rook(squarePtr, pieces[piecePtr].GetColour());
            success = true;
            break;
        case "N":
            pieces[piecePtr] = new Knight(squarePtr, pieces[piecePtr].GetColour());
            success = true;
            break;
    }
    return success;
}

private bool CheckThreat(int squarePtr, int currentPlayer)
{
    bool threat = false;
    ThreatSuperPiece tmp = new ThreatSuperPiece(squarePtr, currentPlayer);
    ///see if there are ANY pieces threatening current square
    List<int> threatMoves = tmp.GeneratePossibleMoves(board, pieces);
    if (threatMoves.Count() > 0) { threat = true; }

    return threat;
}

public bool CheckCheck(int player)
{
    bool check = CheckThreat(pieces[player].GetCurrentPosition(), player);
    return check;
}

public void ParseMove(string move, int currentPlayer)
{
    ///separate pieces from notation and then enact
    string startCoord = move.Substring(1, 3);
    int startSquare = pieces[0].ConvertStrPosToPtr(startCoord);
    SelectPiece(startSquare, currentPlayer);
    string endCoord = "";
    if (move.Contains('X'))
    {
        endCoord = move.Substring(6, 3);
    }
}
```

```

else { endCoord = move.Substring(5, 3); }
//check if promotion, if so enact
int targetSquare = pieces[0].ConvertStrPosToPtr(endCoord);
MovePiece(targetSquare, currentPlayer);
if (move.Contains('='))
{
    string promotedPiece = move.Substring(move.Length - 1, 1);
    PromotePawn(promotedPiece, targetSquare);
}
}
public void UndoMove(string move)
{
    //grab parts of move, like in ParseMove
    string startCoord = move.Substring(1, 3);
    int startSquare = pieces[0].ConvertStrPosToPtr(startCoord);
    string endCoord = "";
    if (move.Contains('X'))
    {
        endCoord = move.Substring(6, 3);
    }
    else { endCoord = move.Substring(5, 3); }
    int endSquare = pieces[0].ConvertStrPosToPtr(endCoord);
    int piecePtr = board[endSquare].GetPiecePointer();
    //undo promotion if there is one
    if (move.Contains('='))
    {
        pieces[piecePtr] = new Pawn(endSquare, pieces[piecePtr].GetColour());
    }
    //reverse move now
    pieces[piecePtr].MovePiece(startSquare, board, pieces);
    board[endSquare].SetPiecePointer(-1);
    board[startSquare].SetPiecePointer(piecePtr);
    //undo capture if there is one
    if (move.Contains('X'))
    {
        string pieceType = Convert.ToString(move[5]);
        int pieceCol = (pieces[piecePtr].GetColour() + 1) % 2;
        AddPiece(pieceType, endSquare, pieceCol);
    }
    //clear shading
    if (pointersMovedToFrom.Count() > 0)
    {
        board[pointersMovedToFrom[0]].NotUnderThreat();
        board[pointersMovedToFrom[1]].NotUnderThreat();
        pointersMovedToFrom.RemoveAt(0);
        pointersMovedToFrom.RemoveAt(0);
    }
    currentPieceIndex = -1;
    for (int x = 0; x < currentPossibleMoves.Count(); x++)
    {
        board[currentPossibleMoves[x]].NotUnderThreat();
    }
}

public Square GetSquare(int ptr)

```

```

    {
        return board[ptr];
    }

    public void SetSquareBlue(int ptr) { board[ptr].SetSquareBlue(); }

    public int GetSquareColour(int ptr)
    {
        return board[ptr].GetColour();
    }
}

```

### Square.cs

```

using System;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ThreeDimensionalChess
{
    enum Colours
    {
        Black, // 0
        White, // 1
        BlackBlue, // 2 - Used for when a friendly move may go onto a black square
        WhiteBlue, // 3 - Used for when a friendly move may go onto a white square
        BlackRed, // 4 - Used for when an enemy move may go onto a black square
        WhiteRed, // 5 - Used for when an enemy move may go onto a white square
        BlackYellow, // 6 - Used for when a move has just gone to or from a black square
        WhiteYellow // 7 - Used for when a move has just gone to or from a white square
    }

    class Square
    {
        private int squareColour;
        //points to current piece in list
        private int piecePointer;

        //constructor of board should initialise squares in for loop and pass colour values
        public Square(int colour)
        {
            squareColour = colour;
            //-1 represents a lack of piece - converter for data binding will handle the displaying of this
            piecePointer = -1;
        }

        public void SetSquareBlue() { squareColour = (int)Colours.BlackBlue; }

        public int GetPiecePointer() { return piecePointer; }
        public void SetPiecePointer(int ptr) { piecePointer = ptr; }
    }
}

```

```
public int GetColour() { return squareColour; }

//change colour values of square
public void UnderThreat(bool friendly)
{
    //reset colour first
    NotUnderThreat();
    if (friendly)
    {
        squareColour += 2;
    }
    else
    {
        squareColour += 4;
    }
}
public void NotUnderThreat()
{
    //returns square colour to normal
    //uses mod, since black squares are even, white squares are odd
    if (squareColour % 2 == 0)
    {
        squareColour = (int)Colours.Black;
    }
    else
    {
        squareColour = (int)Colours.White;
    }
}

public void PieceMoved()
{
    //black squares are even, white squares are odd, shade yellow accordingly
    if (squareColour % 2 == 0)
    {
        squareColour = (int)Colours.BlackYellow;
    }
    else
    {
        squareColour = (int)Colours.WhiteYellow;
    }
}

public void DecrementPiecePointer()
{
    piecePointer--;
}
}
```

### Piece.cs

```
using System;
using System.Linq;
```

```

namespace ThreeDimensionalChess
{
    //neat little enum to help with generating moves
    enum Directions
    {
        Right, // 0- Positive movement on x
        Left, // 1 - Negative Movement on X
        Up, // 2 - Positive movement on Y
        Down, // 3 - Negative movement on y
        Forwards, // 4 - Positive movement on z
        Backwards, // 5 - Negative movement on z
    }
    abstract class Piece
    {
        public int[] movementVect = new int[3];
        //stores current position on board
        public int currentPosition;
        public int colour;

        //constructor for piece
        public Piece(int startPos, int col)
        {
            currentPosition = startPos;
            colour = col;
        }

        public abstract string GetPieceType();

        //method to calculate all possible moves by a piece - returns a list, must be implemented on a per piece
        //basis
        public abstract List<int> GeneratePossibleMoves(List<Square> board, List<Piece> pieces);

        public virtual string MovePiece(int endPosition, List<Square> board, List<Piece> pieces)
        {
            //setup string to return LA3DN data
            string data = "";
            //dereference captured piece here
            int targetPiecePtr = board[endPosition].GetPiecePointer();
            if (targetPiecePtr != -1)
            {
                data += "X" + pieces[targetPiecePtr].GetPieceType();
                pieces.RemoveAt(targetPiecePtr);
            }
            else { data += "-"; }
            data += ConvertPosToStr(endPosition);
            currentPosition = endPosition;
            return data;
        }

        public int GetCurrentPosition()
        {
            return currentPosition;
        }
    }
}

```

//self explanatory names, useful for eachother

```
public int[] ConvertPtrToVect(int inp)
```

```
{
```

//uses modular arithmetic around base 8 to convert a position in the list of squares to a 3D coordinate/vector of the piece's position

```
int[] vect = { 0, 0, 0 };
```

```
int remainder = inp % 64;
```

```
vect[2] = inp / 64;
```

```
vect[1] = remainder / 8;
```

```
vect[0] = remainder % 8;
```

```
return vect;
```

```
}
```

```
public int ConvertVectToPtr(int[] arr)
```

```
{
```

// if this method doesn't receive a three dimensional vector in array form, it will throw an argument exception - thankfully this should never happen

```
if (arr.Length != 3) { throw new ArgumentException(); }
```

```
int[] currentPosVect = ConvertPtrToVect(currentPosition);
```

```
int movePtr = 0;
```

// uses base 8 system to convert three dimensional vectors into pointers for the list of board squares

```
movePtr += arr[0];
```

```
movePtr += arr[1] * 8;
```

```
movePtr += arr[2] * 64;
```

```
return movePtr;
```

```
}
```

```
public int GetColour()
```

```
{
```

```
return colour;
```

```
}
```

//returns a LA3DN compatible coordinate representation

```
public string ConvertPosToStr(int pos)
```

```
{
```

```
int[] vect = ConvertPtrToVect(pos);
```

```
string ret = "";
```

//parse x coord

```
switch (vect[0])
```

```
{
```

```
case 0:
```

```
ret += "a";
```

```
break;
```

```
case 1:
```

```
ret += "b";
```

```
break;
```

```
case 2:
```

```
ret += "c";
```

```
break;
```

```
case 3:
```

```
ret += "d";
```

```
break;
```

```
        case 4:
            ret += "e";
            break;
        case 5:
            ret += "f";
            break;
        case 6:
            ret += "g";
            break;
        case 7:
            ret += "h";
            break;
    }
    //parse y coord
    vect[1]++;
    ret += vect[1];
    //parze z coord
    switch (vect[2])
    {
        case 0:
            ret += "s";
            break;
        case 1:
            ret += "t";
            break;
        case 2:
            ret += "u";
            break;
        case 3:
            ret += "v";
            break;
        case 4:
            ret += "w";
            break;
        case 5:
            ret += "x";
            break;
        case 6:
            ret += "y";
            break;
        case 7:
            ret += "z";
            break;
    }
    return ret;
}

public int ConvertStrPosToPtr(string pos)
{
    int[] vect = new int[3];
    switch (pos[0])
    {
        case 'a':
            vect[0] = 0;
            break;
```

```
    case 'b':
        vect[0] = 1;
        break;
    case 'c':
        vect[0] = 2;
        break;
    case 'd':
        vect[0] = 3;
        break;
    case 'e':
        vect[0] = 4;
        break;
    case 'f':
        vect[0] = 5;
        break;
    case 'g':
        vect[0] = 6;
        break;
    case 'h':
        vect[0] = 7;
        break;
}
//because pos[1] is a char, need to offset by -49 to get the actual number it represents and then -1 to get
a 0-7 digit
vect[1] = Convert.ToInt32(pos[1]) - 49;
switch (pos[2])
{
    case 's':
        vect[2] = 0;
        break;
    case 't':
        vect[2] = 1;
        break;
    case 'u':
        vect[2] = 2;
        break;
    case 'v':
        vect[2] = 3;
        break;
    case 'w':
        vect[2] = 4;
        break;
    case 'x':
        vect[2] = 5;
        break;
    case 'y':
        vect[2] = 6;
        break;
    case 'z':
        vect[2] = 7;
        break;
}
return ConvertVectToPtr(vect);
}
```



```

public string GetCurrentPosAsStr()
{
    return ConvertPosToStr(currentPosition);
}

public int[] GetCurrentPosAsVect()
{
    return ConvertPtrToVect(currentPosition);
}
}
}

```

### Bishop.cs

```

using System;

namespace ThreeDimensionalChess
{
    class Bishop : Piece
    {
        public Bishop(int startPosition, int colour) : base(startPosition, colour) { }

        public override string GetPieceType() { return "B"; }

        public override List<int> GeneratePossibleMoves(List<Square> board, List<Piece> pieces)
        {
            List<int> moves = new List<int>();

            //loop around recursive move generator, bishops can move in 12 directions
            for (int direction = 0; direction < 12; direction++)
            {
                List<int> tmp = new List<int>();
                tmp = GenerateNextMove(direction, board, currentPosition, pieces);

                //append generate moves to main list
                for (int x = 0; x < tmp.Count(); x++)
                {
                    //filter out invalid moves
                    if (tmp[x] != -1) { moves.Add(tmp[x]); }
                }
            }

            return moves;
        }

        private List<int> GenerateNextMove(int dir, List<Square> board, int pos, List<Piece> pieces)
        {
            int[] vect = ConvertPtrToVect(pos);

            //large switch to transform piece, going clockwise around each board, front -> top -> side
            switch (dir)
            {
                case 0:
                    pos += 9;

```

```
    vect[0]++;
    vect[1]++;
    break;
case 1:
    pos -= 7;
    vect[0]++;
    vect[1]--;
    break;
case 2:
    pos -= 9;
    vect[0]--;
    vect[1]--;
    break;
case 3:
    pos += 7;
    vect[0]--;
    vect[1]++;
    break;
//moves from top view
case 4:
    pos += 65;
    vect[0]++;
    vect[2]++;
    break;
case 5:
    pos -= 63;
    vect[0]++;
    vect[2]--;
    break;
case 6:
    pos -= 65;
    vect[0]--;
    vect[2]--;
    break;
case 7:
    pos += 63;
    vect[0]--;
    vect[2]++;
    break;
//moves from side view
case 8:
    pos += 72;
    vect[1]++;
    vect[2]++;
    break;
case 9:
    pos += 56;
    vect[1]--;
    vect[2]++;
    break;
case 10:
    pos -= 72;
    vect[1]--;
    vect[2]--;
    break;
```

```

        case 11:
            pos -= 56;
            vect[1]++;
            vect[2]--;
            break;
    }

    List<int> moves = new List<int>();

    //check that the piece hasn't gone off the board
    if (vect[0] < Constants.boardDimensions && vect[0] > -1 && vect[1] < Constants.boardDimensions &&
        vect[1] > -1 && vect[2] < Constants.boardDimensions && vect[2] > -1)
    {
        //checks if there is a piece on the square
        int targetPtr = board[pos].GetPiecePointer();
        if (targetPtr != -1)
        {
            Piece target = pieces[targetPtr];
            if (target.GetColour() != colour)
            {
                //unwind recursion from here
                moves.Add(pos);
                return moves;
            } // return a -1 to be filtered out later
            else
            {
                moves.Add(-1);
                return moves;
            }
        }
        else
        {
            moves.Add(pos);
            List<int> newMoves = new List<int>();
            newMoves = GenerateNextMove(dir, board, pos, pieces);
            //go deeper in recursion here
            for (int x = 0; x < newMoves.Count(); x++)
            {
                moves.Add(newMoves[x]);
            }
        }
    }
    else { moves.Add(-1); }

    return moves;
}

//used for queen when handling internal move
public void ForceMove(int endPosition)
{
    currentPosition = endPosition;
}
}
}

```

King.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ThreeDimensionalChess
{
    class King : Piece
    {
    public King(int startPosition, int colour) : base(startPosition, colour) { }

    public override string GetPieceType() { return "K"; }

    public override List<int> GeneratePossibleMoves(List<Square> board, List<Piece> pieces)
    {
        List<int> moves = new List<int>();

        //process up down and across moves here
        for (int dir = (int)Directions.Right; dir <= (int)Directions.Backwards; dir++)
        {
            int pos = currentPosition;
            int[] vect = ConvertPtrToVect(pos);
            //switch transforms position
            switch (dir)
            {
                case (int)Directions.Right:
                    //moves right one square
                    pos++;
                    vect[0]++;
                    break;
                case (int)Directions.Left:
                    //moves left one square
                    pos--;
                    vect[0]--;
                    break;
                case (int)Directions.Up:
                    //moves up one square
                    pos += 8;
                    vect[1]++;
                    break;
                case (int)Directions.Down:
                    //moves down one square
                    pos -= 8;
                    vect[1]--;
                    break;
                case (int)Directions.Forwards:
                    //moves deeper on z axis by one square
                    pos += 64;
                    vect[2]++;
                    break;
                case (int)Directions.Backwards:
                    //moves back on z axis by one square
                    pos -= 64;
                    vect[2]--;
                    break;
            }
            if (pos < 0 || pos > 255 || vect[0] < 0 || vect[0] > 7 || vect[1] < 0 || vect[1] > 7 || vect[2] < 0 || vect[2] > 7)
                continue;
            if (board[pos].colour == colour)
                continue;
            moves.Add(pos);
        }
    }
}
```

```
        pos -= 64;
        vect[2]--;
        break;
    }

    //check if piece has gone off edge here
    if (EdgeCheck(vect)) { moves.Add(pos); }
}

//process diagonal moves here
for (int dir = 0; dir < 12; dir++)
{
    int pos = currentPosition;
    int[] vect = ConvertPtrToVect(pos);
    //switch transforms position, collapse for readability, switch is in terms of 8 sided board, change later?
    switch (dir)
    {
        case 0:
            pos += 9;
            vect[0]++;
            vect[1]++;
            break;
        case 1:
            pos -= 7;
            vect[0]++;
            vect[1]--;
            break;
        case 2:
            pos -= 9;
            vect[0]--;
            vect[1]--;
            break;
        case 3:
            pos += 7;
            vect[0]--;
            vect[1]++;
            break;
        //moves from top view
        case 4:
            pos += 65;
            vect[0]++;
            vect[2]++;
            break;
        case 5:
            pos -= 63;
            vect[0]++;
            vect[2]--;
            break;
        case 6:
            pos -= 65;
            vect[0]--;
            vect[2]--;
            break;
        case 7:
            pos += 63;
```

```

        vect[0]--;
        vect[2]++;
        break;
//moves from side view
case 8:
    pos += 72;
    vect[1]++;
    vect[2]++;
    break;
case 9:
    pos += 56;
    vect[1]--;
    vect[2]++;
    break;
case 10:
    pos -= 72;
    vect[1]--;
    vect[2]--;
    break;
case 11:
    pos -= 56;
    vect[1]++;
    vect[2]--;
    break;
    }

//check if piece has gone off edge before adding to move of lists
if (EdgeCheck(vect)) { moves.Add(pos); }
}

//loop to check if pieces on squares
for (int x = 0; x < moves.Count(); x++)
{
    int targetPos = board[moves[x]].GetPiecePointer();
    if (targetPos != -1)
    {
        Piece targetPiece = pieces[targetPos];
        //if piece of same colour on square remove move
        if (targetPiece.GetColour() == colour)
        {
            moves.RemoveAt(x);
            x--;
        }
    }
}

return moves;
}

private bool EdgeCheck(int[] vect)
{
    //returns true if move is safe
    bool safe = false;

```

```

    if (vect[0] > -1 && vect[0] < Constants.boardDimensions && vect[1] > -1 && vect[1] <
        Constants.boardDimensions && vect[2] > -1 && vect[2] < Constants.boardDimensions) { safe = true; }
    return safe;
}
}
}

```

### Knight.cs

```
using System;
```

```
namespace ThreeDimensionalChess
```

```
{
    class Knight : Piece
    {
```

```
public Knight(int startPosition, int colour) : base(startPosition, colour) { }
```

```
public override string GetPieceType()
{
    return "N";
}
```

```
public override List<int> GeneratePossibleMoves(List<Square> board, List<Piece> pieces)
{
    List<int> moves = new List<int>();
```

```
    //possible moves of knight
```

```
    int[] xMoves = { 1, 1, -1, -1, 2, 2, -2, -2, 1, 1, -1, -1, 2, 2, -2, -2, 0, 0, 0, 0, 0, 0, 0, 0 };
    int[] yMoves = { 2, -2, 2, -2, 1, -1, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, -1, -1, 2, 2, -2, -2 };
    int[] zMoves = { 0, 0, 0, 0, 0, 0, 0, 0, 2, -2, 2, -2, 1, -1, 1, -1, 2, -2, 2, -2, 1, -1, 1, -1 };

```

```
    //iterate through and check moves
```

```
    for (int i = 0; i < xMoves.Length; i++)
```

```
    {
        int[] pos = ConvertPtrToVect(currentPosition);
        pos[0] += xMoves[i];
        pos[1] += yMoves[i];
        pos[2] += zMoves[i];

```

```
        //check move hasn't gone off board
```

```
        if (pos[0] > -1 && pos[0] < Constants.boardDimensions && pos[1] > -1 && pos[1] <
            Constants.boardDimensions && pos[2] > -1 && pos[2] < Constants.boardDimensions)
```

```
        {
            int targetPiecePtr = board[ConvertVectToPtr(pos)].GetPiecePointer();

```

```
            //add pointer version of move to list if its valid
```

```
            //filter empty and capture moves here
```

```
            if (targetPiecePtr == -1)
```

```
            {
                moves.Add(ConvertVectToPtr(pos));
            }

```

```
            else if (pieces[targetPiecePtr].GetColour() != this.colour)
```

```
            {
                moves.Add(ConvertVectToPtr(pos));
            }
        }
    }
}
```

```

    }
}

return moves;

}
}
}

```

### Pawn.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ThreeDimensionalChess
{
    class Pawn : Piece
    {

public Pawn(int position, int colour) : base(position, colour) { }

public override string GetPieceType()
{
    return "P";
}

public override List<int> GeneratePossibleMoves(List<Square> board, List<Piece> pieces)
{
    List<int> moves = new List<int>();
    //move in opposite directions based on colour
    //uses private method, makes move simpler to program
    moves = GenerateNextMove(currentPosition, board, pieces);

    //add capture moves here
    //RULE: pawns can make both advancing captures from top and front view, only one option for
    //advancing capture is available on side view
    int[] posVect = ConvertPtrToVect(currentPosition);
    int pos = currentPosition;
    if (colour == (int)Colours.White)
    {
        //edge check for top down advancing captures
        if (posVect[0] != 0 && posVect[2] != 7)
        {
            int target = pos + 64 - 1;
            int targetPiecePtr = board[target].GetPiecePointer();
            if (targetPiecePtr != -1)
            {
                if (pieces[targetPiecePtr].GetColour() != colour) { moves.Add(target); }
            }
        }
        if (posVect[0] != 7 && posVect[2] != 7)
        {

```



```

    int target = pos + 64 + 1;
    int targetPiecePtr = board[target].GetPiecePointer();
    if (targetPiecePtr != -1)
    {
        if (pieces[targetPiecePtr].GetColour() != colour) { moves.Add(target); }
    }
}

//edge check for front view advancing captures
if (posVect[0] != 0 && posVect[1] != 7)
{
    int target = pos + 8 - 1;
    int targetPiecePtr = board[target].GetPiecePointer();
    if (targetPiecePtr != -1)
    {
        if (pieces[targetPiecePtr].GetColour() != colour) { moves.Add(target); }
    }
}
if (posVect[0] != 7 && posVect[1] != 7)
{
    int target = pos + 8 + 1;
    int targetPiecePtr = board[target].GetPiecePointer();
    if (targetPiecePtr != -1)
    {
        if (pieces[targetPiecePtr].GetColour() != colour) { moves.Add(target); }
    }
}

//edge check for side view advancing capture (singular)
if (posVect[1] != 7 && posVect[2] != 7)
{
    int target = pos + 8 + 64;
    int targetPiecePtr = board[target].GetPiecePointer();
    if (targetPiecePtr != -1)
    {
        if (pieces[targetPiecePtr].GetColour() != colour) { moves.Add(target); }
    }
}
}
else
{
    //if this doesn't work may need to switch signs on x transformations (y trans for last statement)
    //becuase its hard to conceptualise
    //edge check for top down captures (black)
    if (posVect[0] != 7 && posVect[2] != 0)
    {
        int target = pos - 64 + 1;
        int targetPiecePtr = board[target].GetPiecePointer();
        if (targetPiecePtr != -1)
        {
            if (pieces[targetPiecePtr].GetColour() != colour) { moves.Add(target); }
        }
    }
}
if (posVect[0] != 0 && posVect[2] != 0)
{

```

```

        int target = pos - 64 - 1;
        int targetPiecePtr = board[target].GetPiecePointer();
        if (targetPiecePtr != -1)
        {
            if (pieces[targetPiecePtr].GetColour() != colour) { moves.Add(target); }
        }
    }

    //edge check for front view captures (black)
    if (posVect[0] != 7 && posVect[1] != 0)
    {
        int target = pos - 8 + 1;
        int targetPiecePtr = board[target].GetPiecePointer();
        if (targetPiecePtr != -1)
        {
            if (pieces[targetPiecePtr].GetColour() != colour) { moves.Add(target); }
        }
    }
    if (posVect[0] != 0 && posVect[1] != 0)
    {
        int target = pos - 8 - 1;
        int targetPiecePtr = board[target].GetPiecePointer();
        if (targetPiecePtr != -1)
        {
            if (pieces[targetPiecePtr].GetColour() != colour) { moves.Add(target); }
        }
    }

    //edge check for side view captures
    if (posVect[1] != 0 && posVect[2] != 0)
    {
        int target = pos - 8 - 64;
        int targetPiecePtr = board[target].GetPiecePointer();
        if (targetPiecePtr != -1)
        {
            if (pieces[targetPiecePtr].GetColour() != colour) { moves.Add(target); }
        }
    }
}

return moves;
}

private List<int> GenerateNextMove(int pos, List<Square> board, List<Piece> pieces)
{
    List<int> ret = new List<int>();
    List<int[]> vects = new List<int[]>();

    //transform position, by one across and one up individually
    //direction depends on piece colour
    if (colour == (int)Colours.White)
    {
        //across move
        ret.Add(pos + 64);
        int[] tmp1 = ConvertPtrToVect(pos);
    }
}

```

```

    tmp1[2]++;
    vects.Add(tmp1);

    //up move
    ret.Add(pos + 8);
    int[] tmp2 = ConvertPtrToVect(pos);
    tmp2[1]++;
    vects.Add(tmp2);
}
else
{
    //across move
    ret.Add(pos - 64);
    int[] tmp1 = ConvertPtrToVect(pos);
    tmp1[2]--;
    vects.Add(tmp1);

    //up move
    ret.Add(pos - 8);
    int[] tmp2 = ConvertPtrToVect(pos);
    tmp2[1]--;
    vects.Add(tmp2);
}

List<int> retMoves = new List<int>();
//edge check
for (int i = 0; i < ret.Count(); i++)
{
    //use statement add valid moves into another list, was having an error where lists were no longer
    //parallel after removing nodes
    if (vects[i][2] > -1 && vects[i][2] < Constants.boardDimensions && vects[i][1] > -1 && vects[i][1] <
    Constants.boardDimensions)
    {
        retMoves.Add(ret[i]);
    }
}

//occupied space check
//written out twice because of same issue as above for loop
if (retMoves.Count() == 2)
{
    int piecePtr = board[retMoves[1]].GetPiecePointer();
    if (piecePtr != -1)
    {
        //remove move because can't move forward onto piece
        retMoves.RemoveAt(1);
    }
}
if (retMoves.Count() >= 1)
{
    int piecePtr = board[retMoves[0]].GetPiecePointer();
    if (piecePtr != -1)
    {
        //remove move because can't move forward onto piece
        retMoves.RemoveAt(0);
    }
}

```

```
    }
}
```

```
    return retMoves;
}
}
}
```

### Queen.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace ThreeDimensionalChess
```

```
{
    class Queen : Piece
    {
```

```
        //create internal pieces that can be used to generate moves
```

```
Bishop internalBishop;
```

```
Rook internalRook;
```

```
public Queen(int startPosition, int col) : base(startPosition, col)
{
    internalBishop = new Bishop(startPosition, col);
    internalRook = new Rook(startPosition, col);
}
```

```
public override string GetPieceType() { return "Q"; }
```

```
        //override virtual move piece method to move internal pieces with piece
```

```
public override string MovePiece(int endPosition, List<Square> board, List<Piece> pieces)
{
```

```
    //setup string to return LA3DN data
```

```
    string data = "";
```

```
    //dereference captured piece here
```

```
    int targetPiecePtr = board[endPosition].GetPiecePointer();
```

```
    if (targetPiecePtr != -1)
```

```
    {
        data += "X" + pieces[targetPiecePtr].GetPieceType();
        pieces.RemoveAt(targetPiecePtr);
    }
```

```
    else { data += "-"; }
```

```
    data += ConvertPosToStr(endPosition);
```

```
    currentPosition = endPosition;
```

```
    internalBishop.ForceMove(endPosition);
```

```
    internalRook.ForceMove(endPosition);
```

```

    return data;
}

public override List<int> GeneratePossibleMoves(List<Square> board, List<Piece> pieces)
{
    List<int> moves = new List<int>();

    //generate moves using internal pieces and then append to main list of moves
    List<int> tmp = internalRook.GeneratePossibleMoves(board, pieces);
    for (int x = 0; x < tmp.Count(); x++) { moves.Add(tmp[x]); }
    tmp = internalBishop.GeneratePossibleMoves(board, pieces);
    for (int x = 0; x < tmp.Count(); x++) { moves.Add(tmp[x]); }

    return moves;
}
}
}

```

### Rook.cs

```

using System;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ThreeDimensionalChess
{
    class Rook : Piece
    {
        //constructor
        public Rook(int startPosition, int colour) : base(startPosition, colour) { }

        public override string GetPieceType() { return "R"; }

        //implementation of generate possible moves, doing this recursively because rooks can move to the edge
        //of the board
        public override List<int> GeneratePossibleMoves(List<Square> board, List<Piece> pieces)
        {
            List<int> moves = new List<int>();

            for (int direction = (int)Directions.Right; direction <= (int)Directions.Backwards; direction++)
            {
                List<int> tmp = new List<int>();
                //calls recursive directional move generator
                tmp = GenerateNextMove(direction, board, currentPosition, pieces);
                //appends all moves generated into moves list
                for (int x = 0; x < tmp.Count(); x++)
                {
                    //filter out when invalid moves have been reached
                    if (tmp[x] != -1) { moves.Add(tmp[x]); }
                }
            }

            return moves;
        }
    }
}

```

```

}

private List<int> GenerateNextMove(int dir, List<Square> board, int pos, List<Piece> pieces)
{
    int[] vect = ConvertPtrToVect(pos);
    //converts direction into an index that addresses the relevant part of a 3-Length Vector Array: (0&1)-> 0,
    (2&3)-> 1, (4&5) -> 2
    int arrayIndex = (dir / 2) % 3;

    //switch to transform position in direction specified - will be using base 8 things
    switch (dir)
    {
        case (int)Directions.Right:
            //moves right one square
            pos++;
            vect[arrayIndex]++;
            break;
        case (int)Directions.Left:
            //moves left one square
            pos--;
            vect[arrayIndex]--;
            break;
        case (int)Directions.Up:
            //moves up one square
            pos += 8;
            vect[arrayIndex]++;
            break;
        case (int)Directions.Down:
            //moves down one square
            pos -= 8;
            vect[arrayIndex]--;
            break;
        case (int)Directions.Forwards:
            //moves deeper on z axis by one square
            pos += 64;
            vect[arrayIndex]++;
            break;
        case (int)Directions.Backwards:
            //moves back on z axis by one square
            pos -= 64;
            vect[arrayIndex]--;
            break;
    }

    List<int> moves = new List<int>();

    //checks that piece hasn't gone off the edge of the board
    if (vect[arrayIndex] < Constants.boardDimensions && vect[arrayIndex] > -1)
    {
        //checks if there is a piece on the square
        int targetPtr = board[pos].GetPiecePointer();
        if (targetPtr != -1)
        {
            //now check if piece is friendly or enemy

```

```

    Piece target = pieces[targetPtr];
    if (target.GetColour() != colour)
    {
        //unwind recursion from here
        moves.Add(pos);
        return moves;
    } // return a -1 to be filtered out later
    else
    {
        moves.Add(-1);
        return moves;
    }
}
else
{
    moves.Add(pos);
    List<int> newMoves = new List<int>();
    newMoves = GenerateNextMove(dir, board, pos, pieces);
    //go deeper in recursion here
    for (int x = 0; x < newMoves.Count(); x++)
    {
        moves.Add(newMoves[x]);
    }
}

} // return a -1 to be filtered out later, prevent null errors
else { moves.Add(-1); }

return moves;
}

//used for queen when handling internal move
public void ForceMove(int endPosition)
{
    currentPosition = endPosition;
}
}
}

```

### ThreatSuperPiece.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ThreeDimensionalChess
{
    //the threat super piece is capable of making every possible move, but only returns the capture moves -
    //because these capture moves will give the squarePointers of threatening pieces
    class ThreatSuperPiece : Piece
    {
        public ThreatSuperPiece(int startPos, int col) : base(startPos, col) { }
    }
}

```

```
public override string GetPieceType() { return "TSP"; }

public override List<int> GeneratePossibleMoves(List<Square> board, List<Piece> pieces)
{
    List<int> moves = new List<int>();

    //process rook moves here
    for (int direction = (int)Directions.Right; direction <= (int)Directions.Backwards; direction++)
    {
        List<int> tmpR = new List<int>();
        //calls recursive directional move generator
        tmpR = GenerateNextMoveRook(direction, board, currentPosition, pieces);
        //appends all moves Generated into moves list
        for (int x = 0; x < tmpR.Count(); x++)
        {
            //filter out when invalid moves have been reached
            if (tmpR[x] != -1) { moves.Add(tmpR[x]); }
        }
    }

    //process bishop moves here
    for (int direction = 0; direction < 12; direction++)
    {
        List<int> tmpB = new List<int>();
        tmpB = GenerateNextMoveBishop(direction, board, currentPosition, pieces);

        //append Generate moves to main list
        for (int x = 0; x < tmpB.Count(); x++)
        {
            //filter out invalid moves
            if (tmpB[x] != -1) { moves.Add(tmpB[x]); }
        }
    }

    //process knight moves here
    List<int> tmpN = new List<int>();
    tmpN = GenerateKnightMoves(board, pieces);
    //append generated moves to main list
    for (int x = 0; x < tmpN.Count(); x++)
    {
        moves.Add(tmpN[x]);
    }

    //process pawn moves here - only need to check captures, from reverse
    List<int> tmpP = new List<int>();
    tmpP = GeneratePawnCaptures(board, pieces);
    for (int x = 0; x < tmpP.Count(); x++)
    {
        moves.Add(tmpP[x]);
    }

    //process King moves here
    List<int> tmpK = new List<int>();
    tmpK = GenerateKingMoves(board, pieces);
    for (int x = 0; x < tmpK.Count(); x++)
```



```
{
    moves.Add(tmpK[x]);
}

return moves;
}

public List<int> GenerateKingMoves(List<Square> board, List<Piece> pieces)
{
    List<int> moves = new List<int>();

    //process up down and across moves here
    for (int dir = (int)Directions.Right; dir <= (int)Directions.Backwards; dir++)
    {
        int pos = currentPosition;
        int[] vect = ConvertPtrToVect(pos);
        //switch transforms position
        switch (dir)
        {
            case (int)Directions.Right:
                //moves right one square
                pos++;
                vect[0]++;
                break;
            case (int)Directions.Left:
                //moves left one square
                pos--;
                vect[0]--;
                break;
            case (int)Directions.Up:
                //moves up one square
                pos += 8;
                vect[1]++;
                break;
            case (int)Directions.Down:
                //moves down one square
                pos -= 8;
                vect[1]--;
                break;
            case (int)Directions.Forwards:
                //moves deeper on z axis by one square
                pos += 64;
                vect[2]++;
                break;
            case (int)Directions.Backwards:
                //moves back on z axis by one square
                pos -= 64;
                vect[2]--;
                break;
        }

        //check if piece has gone off edge here
        if (EdgeCheck(vect)) { moves.Add(pos); }
    }
}
```

```
//process diagonal moves here
for (int dir = 0; dir < 12; dir++)
{
    int pos = currentPosition;
    int[] vect = ConvertPtrToVect(pos);
    //switch transforms position, collapse for readability, switch is in terms of 8 sided board, change later?
    switch (dir)
    {
        case 0:
            pos += 9;
            vect[0]++;
            vect[1]++;
            break;
        case 1:
            pos -= 7;
            vect[0]++;
            vect[1]--;
            break;
        case 2:
            pos -= 9;
            vect[0]--;
            vect[1]--;
            break;
        case 3:
            pos += 7;
            vect[0]--;
            vect[1]++;
            break;
        //moves from top view
        case 4:
            pos += 65;
            vect[0]++;
            vect[2]++;
            break;
        case 5:
            pos -= 63;
            vect[0]++;
            vect[2]--;
            break;
        case 6:
            pos -= 65;
            vect[0]--;
            vect[2]--;
            break;
        case 7:
            pos += 63;
            vect[0]--;
            vect[2]++;
            break;
        //moves from side view
        case 8:
            pos += 72;
            vect[1]++;
            vect[2]++;
            break;
```

```

        case 9:
            pos += 56;
            vect[1]--;
            vect[2]++;
            break;
        case 10:
            pos -= 72;
            vect[1]--;
            vect[2]--;
            break;
        case 11:
            pos -= 56;
            vect[1]++;
            vect[2]--;
            break;
    }

    //check if piece has gone off edge before adding to move of lists
    if (EdgeCheck(vect)) { moves.Add(pos); }
}

List<int> finalMoves = new List<int>();
//loop to check if pieces on squares
for (int x = 0; x < moves.Count(); x++)
{
    int targetPos = board[moves[x]].GetPiecePointer();
    if (targetPos != -1)
    {
        //only add move if its capturing
        if (pieces[targetPos].GetColour() != colour && pieces[targetPos].GetPieceType() == "K")
        {
            finalMoves.Add(moves[x]);
        }
    }
}

return finalMoves;
}

private List<int> GenerateNextMoveRook(int dir, List<Square> board, int pos, List<Piece> pieces)
{
    int[] vect = ConvertPtrToVect(pos);
    //converts direction into an index that addresses the relevant part of a 3-Length Vector Array: (0&1)-> 0,
    (2&3)-> 1, (4&5) -> 2
    int arrayIndex = (dir / 2) % 3;

    //switch to transform position in direction specified - will be using base 8 things
    switch (dir)
    {
        case (int)Directions.Right:
            //moves right one square
            pos++;
            vect[arrayIndex]++;
    }
}

```

```

        break;
    case (int)Directions.Left:
        //moves left one square
        pos--;
        vect[arrayIndex]--;
        break;
    case (int)Directions.Up:
        //moves up one square
        pos += 8;
        vect[arrayIndex]++;
        break;
    case (int)Directions.Down:
        //moves down one square
        pos -= 8;
        vect[arrayIndex]--;
        break;
    case (int)Directions.Forwards:
        //moves deeper on z axis by one square
        pos += 64;
        vect[arrayIndex]++;
        break;
    case (int)Directions.Backwards:
        //moves back on z axis by on square
        pos -= 64;
        vect[arrayIndex]--;
        break;
}

List<int> moves = new List<int>();

//checks that piece hasn't gone off the edge of the board
if (vect[arrayIndex] < Constants.boardDimensions && vect[arrayIndex] > -1)
{
    //checks if there is a piece on the square
    int targetPtr = board[pos].GetPiecePointer();
    if (targetPtr != -1)
    {
        //now check if piece is friendly or enemy
        Piece target = pieces[targetPtr];
        //also check if piece could make capturing moves
        if (target.GetColour() != colour && (target.GetPieceType() == "R" || target.GetPieceType() == "Q"))
        {
            //unwind recursion from here
            moves.Add(pos);
            return moves;
        } // return a -1 to be filtered out later
        else
        {
            moves.Add(-1);
            return moves;
        }
    }
}
else
{
    //moves.Add(pos);

```

```

        //don't add moves unless its a capture
        List<int> newMoves = new List<int>();
        newMoves = GenerateNextMoveRook(dir, board, pos, pieces);
        //go deeper in recursion here
        for (int x = 0; x < newMoves.Count(); x++)
        {
            moves.Add(newMoves[x]);
        }
    }

} // return a -1 to be filtered out later, prevent null errors
else { moves.Add(-1); }

return moves;
}

private List<int> GenerateNextMoveBishop(int dir, List<Square> board, int pos, List<Piece> pieces)
{
    int[] vect = ConvertPtrToVect(pos);

    //large switch to transform piece, going clockwise around each board, front -> top -> side
    switch (dir)
    {
        case 0:
            pos += 9;
            vect[0]++;
            vect[1]++;
            break;
        case 1:
            pos -= 7;
            vect[0]++;
            vect[1]--;
            break;
        case 2:
            pos -= 9;
            vect[0]--;
            vect[1]--;
            break;
        case 3:
            pos += 7;
            vect[0]--;
            vect[1]++;
            break;
        //moves from top view
        case 4:
            pos += 65;
            vect[0]++;
            vect[2]++;
            break;
        case 5:
            pos -= 63;
            vect[0]++;
            vect[2]--;
            break;
        case 6:

```

```

        pos -= 65;
        vect[0]--;
        vect[2]--;
        break;
    case 7:
        pos += 63;
        vect[0]--;
        vect[2]++;
        break;
    //moves from side view
    case 8:
        pos += 72;
        vect[1]++;
        vect[2]++;
        break;
    case 9:
        pos += 56;
        vect[1]--;
        vect[2]++;
        break;
    case 10:
        pos -= 72;
        vect[1]--;
        vect[2]--;
        break;
    case 11:
        pos -= 56;
        vect[1]++;
        vect[2]--;
        break;
}

List<int> moves = new List<int>();

//check that the piece hasn't gone off the board
if (vect[0] < Constants.boardDimensions && vect[0] > -1 && vect[1] < Constants.boardDimensions &&
vect[1] > -1 && vect[2] < Constants.boardDimensions && vect[2] > -1)
{
    //checks if there is a piece on the square
    int targetPtr = board[pos].GetPiecePointer();
    if (targetPtr != -1)
    {
        Piece target = pieces[targetPtr];
        //check piece could make move also
        if (target.GetColour() != colour && (target.GetPieceType() == "Q" || target.GetPieceType() == "B"))
        {
            //unwind recursion from here
            moves.Add(pos);
            return moves;
        } // return a -1 to be filtered out later
        else
        {
            moves.Add(-1);
            return moves;
        }
    }
}

```

```

    }
    else
    {
        //commented out so that only capture moves are returned
        //moves.Add(pos);
        List<int> newMoves = new List<int>();
        newMoves = GenerateNextMoveBishop(dir, board, pos, pieces);
        //go deeper in recursion here
        for (int x = 0; x < newMoves.Count(); x++)
        {
            moves.Add(newMoves[x]);
        }
    }
}
else { moves.Add(-1); }

return moves;
}

private List<int> GenerateKnightMoves(List<Square> board, List<Piece> pieces)
{
    List<int> moves = new List<int>();

    //possible moves of knight
    int[] xMoves = { 1, 1, -1, -1, 2, 2, -2, -2, 1, 1, -1, -1, 2, 2, -2, -2, 0, 0, 0, 0, 0, 0, 0, 0 };
    int[] yMoves = { 2, -2, 2, -2, 1, -1, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, -1, -1, 2, 2, -2, -2 };
    int[] zMoves = { 0, 0, 0, 0, 0, 0, 0, 0, 2, -2, 2, -2, 1, -1, 1, -1, 2, -2, 2, -2, 1, -1, 1, -1 };

    //iterate through and check moves
    for (int i = 0; i < xMoves.Length; i++)
    {
        int[] pos = ConvertPtrToVect(currentPosition);
        pos[0] += xMoves[i];
        pos[1] += yMoves[i];
        pos[2] += zMoves[i];

        //check move hasn't gone off board
        if (pos[0] > -1 && pos[0] < Constants.boardDimensions && pos[1] > -1 && pos[1] <
Constants.boardDimensions && pos[2] > -1 && pos[2] < Constants.boardDimensions)
        {
            int targetPiecePtr = board[ConvertVectToPtr(pos)].GetPiecePointer();
            //add pointer version of move to list if its valid
            //modified to only return capture moves
            if (targetPiecePtr != -1 && pieces[targetPiecePtr].GetColour() != this.colour &&
pieces[targetPiecePtr].GetPieceType() == "N")
            {
                moves.Add(ConvertVectToPtr(pos));
            }
        }
    }

    return moves;
}

```

```
private List<int> GeneratePawnCaptures(List<Square> board, List<Piece> pieces)
{
    List<int> moves = new List<int>();
    int[] vect = ConvertPtrToVect(currentPosition);

    //might need to add extra conditions e.g. for black check that vect[1] != 7
    if (colour == (int)Colours.White)
    {
        //Generate black captures here
        //front view captures:
        if (vect[0] != 0 && vect[1] != 7)
        {
            int pos = currentPosition + 8 - 1;
            moves.Add(pos);
        }
        if (vect[0] != 7 && vect[1] != 7)
        {
            int pos = currentPosition + 8 + 1;
            moves.Add(pos);
        }
        //top down captures:
        if (vect[0] != 0 && vect[2] != 7)
        {
            int pos = currentPosition + 64 - 1;
            moves.Add(pos);
        }
        if (vect[0] != 7 && vect[2] != 7)
        {
            int pos = currentPosition + 64 + 1;
            moves.Add(pos);
        }
        //side view capture
        if (vect[1] != 7 && vect[2] != 7)
        {
            int pos = currentPosition + 64 + 8;
            moves.Add(pos);
        }
    }
    else
    {
        //Generate white captures here
        //front view captures
        if (vect[0] != 0 && vect[1] != 0)
        {
            int pos = currentPosition - 8 - 1;
            moves.Add(pos);
        }
        if (vect[0] != 7 && vect[1] != 0)
        {
            int pos = currentPosition - 8 + 1;
            moves.Add(pos);
        }
        //top down captures
        if (vect[0] != 0 && vect[2] != 0)
```



```

    {
        int pos = currentPosition - 64 - 1;
        moves.Add(pos);
    }
    if (vect[0] != 7 && vect[2] != 0)
    {
        int pos = currentPosition - 64 + 1;
        moves.Add(pos);
    }
    //side view capture
    if (vect[1] != 0 && vect[2] != 0)
    {
        int pos = currentPosition - 64 - 8;
        moves.Add(pos);
    }
}

List<int> finalMoves = new List<int>();

for(int x = 0; x < moves.Count(); x++)
{
    //filter moves so its only pawns of opposite colour
    int targetPtr = board[moves[x]].GetPiecePointer();
    if(targetPtr != -1)
    {
        if (pieces[targetPtr].GetColour() != colour && pieces[targetPtr].GetPieceType() == "P")
        {
            finalMoves.Add(moves[x]);
        }
    }
}

return finalMoves;
}

private bool EdgeCheck(int[] vect)
{
    //returns true if move is safe
    bool safe = false;

    if (vect[0] > -1 && vect[0] < Constants.boardDimensions && vect[1] > -1 && vect[1] <
Constants.boardDimensions && vect[2] > -1 && vect[2] < Constants.boardDimensions) { safe = true; }
    return safe;
}
}
}

```

### SimulatedBoard.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

```

```

namespace ThreeDimensionalChess
{
    internal class SimulatedBoard
    {
        //one dimensional list used to store board - used modular arithmetic on vectors to navigate it
        private List<Square> board = new List<Square>();
        private List<Piece> pieces = new List<Piece>();

        //cloning constructor
        public SimulatedBoard(List<Square> b, List<Piece> p)
        {
            for (int x = 0; x < Constants.boardSize; x++)
            {
                //adds a new square to the board, performs mod on x to get colour, Colours enum is stored in
                //Square.cs even squares are black(with 0 also being black), odd are white
                board.Add(new Square(x % 2));
                //copy over positions
                board[x].SetPiecePointer(b[x].GetPiecePointer());
            }
            //copy over pieces
            for (int x = 0; x < p.Count(); x++)
            {
                AddPiece(p[x].GetPieceType(), p[x].GetCurrentPosition(), p[x].GetColour());
            }
        }
        public void AddPiece(string type, int pos, int colour)
        {
            Piece p = null;
            switch (type)
            {
                case "R":
                    p = new Rook(pos, colour);
                    break;
                case "P":
                    p = new Pawn(pos, colour);
                    break;
                case "K":
                    p = new King(pos, colour);
                    break;
                case "B":
                    p = new Bishop(pos, colour);
                    break;
                case "N":
                    p = new Knight(pos, colour);
                    break;
                case "Q":
                    p = new Queen(pos, colour);
                    break;
                case "TSP":
                    p = new ThreatSuperPiece(pos, colour);
                    break;
            }
        }
    }
}

```

```

        pieces.Add(p);

        board[pos].SetPiecePointer(pieces.Count() - 1);
    }

    public bool LegalMove(int endPos, int piecePtr, int currentPlayer)
    {
        int targetPiecePtr = board[endPos].GetPiecePointer();

        board[pieces[piecePtr].GetCurrentPosition()].SetPiecePointer(-1);
        string moveData = pieces[piecePtr].MovePiece(endPos, board, pieces);
        //added logic here to prevent ptr misalignment - if a piece is taken and its pointer is less than the current
        //piece, we need to offset current piece ptr by one
        if (moveData.Contains("X"))
        {
            if (targetPiecePtr < piecePtr) { piecePtr--; }
            for (int x = targetPiecePtr; x < pieces.Count(); x++)
            {
                board[pieces[x].GetCurrentPosition()].DecrementPiecePointer();
            }
        }
        board[endPos].SetPiecePointer(piecePtr);

        //use check threat method with piece index as currentplayer (points at their king)
        //takes negation of return value, since checkthreat returns true for threat but we want to know if move is
        //safe
        bool ret = !CheckThreat(pieces[currentPlayer].GetCurrentPosition(), currentPlayer);
        return ret;
    }

    private bool CheckThreat(int squarePtr, int currentPlayer)
    {
        bool threat = false;
        ThreatSuperPiece tmp = new ThreatSuperPiece(squarePtr, currentPlayer);
        //see if there are ANY pieces threatening current square
        List<int> threatMoves = tmp.GeneratePossibleMoves(board, pieces);
        if (threatMoves.Count() > 0) { threat = true; }

        return threat;
    }
}

```

### DatabaseHandler.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data.SQLite;

namespace ThreeDimensionalChess
{

```

```

class DatabaseHandler
{
    //constructor
    public DatabaseHandler()
    {
        //create tables, first init only
        //CreateTables();
    }

    private void CreateTables()
    {
        //create db connection
        SQLiteConnection dbConnection = new SQLiteConnection("Data Source=database.db");
        dbConnection.Open();
        SQLiteCommand comm = dbConnection.CreateCommand();
        //enter command string
        comm.CommandText =
            @"
CREATE TABLE player (
    playerID INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    whiteLosses INTEGER NOT NULL,
    blackLosses INTEGER NOT NULL,
    whiteDraws INTEGER NOT NULL,
    blackDraws INTEGER NOT NULL,
    whiteWins INTEGER NOT NULL,
    blackWins INTEGER NOT NULL,
    date DATE NOT NULL
);

CREATE TABLE game (
    gameID INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    moveList TEXT NOT NULL,
    gamestate INTEGER NOT NULL,
    lastAccessed DATE NOT NULL,
    whitePlayerID INTEGER NOT NULL,
    blackPlayerID INTEGER NOT NULL,
    undoMoves BOOLEAN NOT NULL
);

INSERT INTO player (name, whiteLosses, blackLosses, whiteDraws, blackDraws, whiteWins, blackWins, date)
VALUES ('testPlayer1', 0, 0, 0, 0, 0, 0, $date);

INSERT INTO player (name, whiteLosses, blackLosses, whiteDraws, blackDraws, whiteWins, blackWins, date)
VALUES ('testPlayer2', 0, 0, 0, 0, 0, 0, $date);";
        comm.Parameters.AddWithValue("$date", DateTime.Today);
        comm.ExecuteNonQuery();
        dbConnection.Close();
    }

    public int AddPlayer(string name)

```

```

{
    SQLiteConnection dbConnection = new SQLiteConnection("Data Source=database.db");
    dbConnection.Open();
    SQLiteCommand comm = dbConnection.CreateCommand();

    //enter command text
    comm.CommandText =
        @"
INSERT INTO player (name, whiteLosses, blackLosses, whiteDraws, blackDraws, whiteWins, blackWins, date)
VALUES ($name, 0, 0, 0, 0, 0, 0, $date);";
    comm.Parameters.AddWithValue("$name", name);
    comm.Parameters.AddWithValue("$date", DateTime.Today);

    comm.ExecuteNonQuery();

    //once player is created need to return its ID to higher level
    comm.CommandText = "SELECT * FROM player ORDER BY playerId DESC LIMIT 1;";
    SQLiteDataReader reader = comm.ExecuteReader();
    reader.Read();
    int ret = reader.GetInt32(0);
    reader.Close();
    dbConnection.Close();

    return ret;
}

public List<Player> GetPlayers()
{
    //set up query
    List<Player> ret = new List<Player>();
    SQLiteConnection dbConnection = new SQLiteConnection("Data Source=database.db");
    dbConnection.Open();
    SQLiteCommand comm = dbConnection.CreateCommand();

    comm.CommandText = "SELECT * FROM player";
    SQLiteDataReader reader = comm.ExecuteReader();

    //read the db into the list
    while (reader.Read())
    {
        int ID = reader.GetInt32(0);
        string name = reader.GetString(1);
        int whiteLosses = reader.GetInt32(2);
        int blackLosses = reader.GetInt32(3);
        int whiteDraws = reader.GetInt32(4);
        int blackDraws = reader.GetInt32(5);
        int whiteWins = reader.GetInt32(6);
        int blackWins = reader.GetInt32(7);
        DateTime joinDate = reader.GetDateTime(8);
        Player tmp = new Player(ID, name, whiteLosses, blackLosses, whiteDraws, blackDraws, whiteWins,
blackWins, joinDate);
        ret.Add(tmp);
    }
    reader.Close();
    dbConnection.Close();
}

```

```

        return ret;
    }

    public Player GetPlayer(int inp)
    {
        SQLiteConnection dbConnection = new SQLiteConnection("Data Source=database.db");
        dbConnection.Open();
        SQLiteCommand comm = dbConnection.CreateCommand();

        comm.CommandText = "SELECT * FROM player WHERE playerId=$input;";
        comm.Parameters.AddWithValue("$input", inp);
        SQLiteDataReader reader = comm.ExecuteReader();

        reader.Read();
        int ID = reader.GetInt32(0);
        string name = reader.GetString(1);
        int whiteLosses = reader.GetInt32(2);
        int blackLosses = reader.GetInt32(3);
        int whiteDraws = reader.GetInt32(4);
        int blackDraws = reader.GetInt32(5);
        int whiteWins = reader.GetInt32(6);
        int blackWins = reader.GetInt32(7);
        DateTime joinDate = reader.GetDateTime(8);
        Player ret = new Player(ID, name, whiteLosses, blackLosses, whiteDraws, blackDraws, whiteWins,
blackWins, joinDate);
        reader.Close();
        dbConnection.Close();

        return ret;
    }

    public void UpdatePlayer(Player p)
    {
        SQLiteConnection dbConnection = new SQLiteConnection("Data Source=database.db");
        dbConnection.Open();
        SQLiteCommand comm = dbConnection.CreateCommand();

        comm.CommandText = @"
UPDATE player
SET whiteLosses=$WHL, blackLosses=$BLL, whiteDraws=$WHD, blackDraws=$BLD, whiteWins=$WHW,
blackWins=$BLW
WHERE playerId=$ID;";

        //load parameters from player obj
        comm.Parameters.AddWithValue("$WHL", p.GetWhiteLosses());
        comm.Parameters.AddWithValue("$BLL", p.GetBlackLosses());
        comm.Parameters.AddWithValue("$WHD", p.GetWhiteDraws());
        comm.Parameters.AddWithValue("$BLD", p.GetBlackDraws());
        comm.Parameters.AddWithValue("$WHW", p.GetWhiteWins());
        comm.Parameters.AddWithValue("$BLW", p.GetBlackWins());
        comm.Parameters.AddWithValue("$ID", p.GetID());

        comm.ExecuteNonQuery();
        dbConnection.Close();
    }

```

```

    }

    //returns false if unsuccessful, true if successful
    public bool DeletePlayer(int inp)
    {
        SQLiteConnection dbConnection = new SQLiteConnection("Data Source=database.db");
        dbConnection.Open();
        SQLiteCommand comm = dbConnection.CreateCommand();

        //delete player from db, make sure to delete any games referencing them too
        comm.CommandText = @"
DELETE FROM player WHERE playerId=$input;

DELETE FROM game WHERE whitePlayerID=$input OR blackPlayerID=$input;
";
        comm.Parameters.AddWithValue("$input", inp);

        bool ret = true;
        try
        {
            comm.ExecuteNonQuery();
        }
        catch(Exception e)
        {
            //probably expecting an SQLLogic or SQLArgument exception here
            Console.WriteLine("Experienced Error: " + e.Message);
            ret = false;
        }
        dbConnection.Close();

        return ret;
    }

    public int CreateGame(string name, bool undo, int whiteID, int blackID)
    {
        SQLiteConnection dbConnection = new SQLiteConnection("Data Source=database.db");
        dbConnection.Open();
        SQLiteCommand comm = dbConnection.CreateCommand();

        comm.CommandText = @"
INSERT INTO GAME (name, moveList, gamestate, lastAccessed, whitePlayerID, blackPlayerID, undoMoves)
VALUES ($name, $empty, $state, $date, $white, $black, $undo);";

        //insert params
        comm.Parameters.AddWithValue("$name", name);
        comm.Parameters.AddWithValue("$empty", "");
        comm.Parameters.AddWithValue("$state", (int)Gamestates.Ongoing);
        comm.Parameters.AddWithValue("$date", DateTime.Today);
        comm.Parameters.AddWithValue("$white", whiteID);
        comm.Parameters.AddWithValue("$black", blackID);
        comm.Parameters.AddWithValue("$undo", undo);

        comm.ExecuteNonQuery();

        //once game is created need to return its ID to higher level
    }

```

```

        comm.CommandText = "SELECT * FROM game ORDER BY gameId DESC LIMIT 1;";
        SQLiteDataReader reader = comm.ExecuteReader();
        reader.Read();
        int ret = reader.GetInt32(0);
        reader.Close();
        dbConnection.Close();
        return ret;
    }

    public void UpdateGame(string moves, int state, int ID)
    {
        SQLiteConnection dbConnection = new SQLiteConnection("Data Source=database.db");
        dbConnection.Open();
        SQLiteCommand comm = dbConnection.CreateCommand();

        comm.CommandText = @"
UPDATE game
SET moveList=$moves, gamestate=$state, lastAccessed=$date
WHERE gameId=$ID";

        //load params into query
        comm.Parameters.AddWithValue("$moves", moves);
        comm.Parameters.AddWithValue("$state", state);
        comm.Parameters.AddWithValue("$date", DateTime.Today);
        comm.Parameters.AddWithValue("$ID", ID);

        comm.ExecuteNonQuery();
        dbConnection.Close();
    }

    public List<GameInfo> GetGames()
    {
        List<GameInfo> ret = new List<GameInfo>();
        SQLiteConnection dbConnection = new SQLiteConnection("Data Source=database.db");
        dbConnection.Open();
        SQLiteCommand comm = dbConnection.CreateCommand();

        comm.CommandText = "SELECT * FROM game";
        SQLiteDataReader reader = comm.ExecuteReader();

        //iterate through all records
        while (reader.Read())
        {
            int ID = reader.GetInt32(0);
            string name = reader.GetString(1);
            string moveListRepr = reader.GetString(2);
            int gamestate = reader.GetInt32(3);
            DateTime lastAccessed = reader.GetDateTime(4);
            int whiteID = reader.GetInt32(5);
            int blackID = reader.GetInt32(6);
            bool undoMoves = reader.GetBoolean(7);
            GameInfo tmp = new GameInfo(ID, name, moveListRepr, gamestate, lastAccessed, whiteID,
            blackID, undoMoves);
            ret.Add(tmp);
        }
    }

```



```
        reader.Close();
        dbConnection.Close();

        return ret;
    }

    public GameInfo GetGame(int inp)
    {
        SQLiteConnection dbConnection = new SQLiteConnection("Data Source=database.db");
        dbConnection.Open();
        SQLiteCommand comm = dbConnection.CreateCommand();

        comm.CommandText = "SELECT * FROM game WHERE gameId=$input;";
        comm.Parameters.AddWithValue("$input", inp);
        SQLiteDataReader reader = comm.ExecuteReader();

        //pull values
        reader.Read();
        int ID = reader.GetInt32(0);
        string name = reader.GetString(1);
        string moveListRepr = reader.GetString(2);
        int gamestate = reader.GetInt32(3);
        DateTime lastAccessed = reader.GetDateTime(4);
        int whiteID = reader.GetInt32(5);
        int blackID = reader.GetInt32(6);
        bool undoMoves = reader.GetBoolean(7);
        GameInfo tmp = new GameInfo(ID, name, moveListRepr, gamestate, lastAccessed, whiteID, blackID,
undoMoves);
        reader.Close();
        dbConnection.Close();

        return tmp;
    }

    public bool DeleteGame(int inp)
    {
        SQLiteConnection dbConnection = new SQLiteConnection("Data Source=database.db");
        dbConnection.Open();
        SQLiteCommand comm = dbConnection.CreateCommand();

        comm.CommandText = "DELETE FROM game WHERE gameId=$input;";
        comm.Parameters.AddWithValue("$input", inp);

        bool ret = true;
        try
        {
            comm.ExecuteNonQuery();
        }
        catch (Exception e)
        {
            //probably expecting an SQLLogic or SQLArgument exception here
            Console.WriteLine("Experienced Error: " + e.Message);
            ret = false;
        }
        dbConnection.Close();
    }
}
```

```

        return ret;
    }
}

```

### GameInfo.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ThreeDimensionalChess
{
    class GameInfo
    {
        private int gameId;
        private string name;
        private List<string> moves;
        private int gamestate;
        private DateTime lastAccessed;
        private int whitePlayerID;
        private int blackPlayerID;
        private bool undoMoves;
        public GameInfo(int gameIdInp, string nameInp, string movesInp, int gamestateInp, DateTime
lastAccessedInp, int whitePlayerInp, int blackPlayerInp, bool undoMovesInp)
        {
            gameId = gameIdInp;
            name = nameInp;
            moves = new List<string>(movesInp.Split(','));
            gamestate = gamestateInp;
            //if last move is an unfinished promotion, remove it, set state accordingly
            if (moves[moves.Count() - 1].Contains("=") && moves[moves.Count() - 1].Split('=')[1].Length == 0) {
moves.RemoveAt(moves.Count() - 1); gamestate = (int)Gamestates.Ongoing; }
            lastAccessed = lastAccessedInp;
            whitePlayerID = whitePlayerInp;
            blackPlayerID = blackPlayerInp;
            undoMoves = undoMovesInp;
        }

        public int GetGameID() { return gameId; }
        public string GetName() { return name; }
        public string GetWhitePlayerName()
        {
            DatabaseHandler db = new DatabaseHandler();
            string ret = db.GetPlayer(whitePlayerID).GetName();
            return ret;
        }
        public string GetBlackPlayerName()
        {
            DatabaseHandler db = new DatabaseHandler();
            string ret = db.GetPlayer(blackPlayerID).GetName();

```

```

    return ret;
}
public bool GetUndoMoves() { return undoMoves; }
public List<string> GetMoves() { return moves; }
public string GetGamestate()
{
    //return a string, better for UI rep
    string ret = "Ongoing";
    switch (gamestate)
    {
        case (int)Gamestates.WhiteW:
            ret = "White W";
            break;
        case (int)Gamestates.BlackW:
            ret = "Black W";
            break;
        case (int)Gamestates.Stalemate:
            ret = "Draw";
            break;
    }
    return ret;
}
public int GetGamestateForComparison()
{
    // return a -1 if game is ongoing in anyway
    if(gamestate == (int)Gamestates.Ongoing || gamestate == (int)Gamestates.PendingPromo) return -1;
    return gamestate;
}
public string GetLastMove()
{
    string tmp = moves[moves.Count()-1];
    if (tmp == "") { tmp = "No moves have been made"; }
    return tmp;
}
public int GetGamestateAsInt() { return gamestate; }
public DateTime GetLastAccessed() { return lastAccessed; }
public int GetWhitePlayerID() { return whitePlayerID; }
public int GetBlackPlayerID() { return blackPlayerID; }
}
}

```

### Player.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ThreeDimensionalChess
{
    class Player
    {
        private int ID;
    }
}

```

```
private string name;
private int whiteLosses;
private int blackLosses;
private int whiteDraws;
private int blackDraws;
private int whiteWins;
private int blackWins;
private DateTime joinDate;
private int colour;
DatabaseHandler db;

public Player(int IDInp, string nameInp, int whiteLossesInp, int blackLossesInp, int whiteDrawsInp, int
blackDrawsInp, int whiteWinsInp, int blackWinsInp, DateTime joinDateInp)
{
    ID = IDInp;
    name = nameInp;
    whiteLosses = whiteLossesInp;
    blackLosses = blackLossesInp;
    whiteDraws = whiteDrawsInp;
    blackDraws = blackDrawsInp;
    whiteWins = whiteWinsInp;
    blackWins = blackWinsInp;
    joinDate = joinDateInp;
    DatabaseHandler db = new DatabaseHandler();
}

// write methods
public void AddWhiteWin() { whiteWins++;}
public void AddBlackWin() { blackWins++;}
public void AddWhiteLoss() { whiteLosses++;}
public void AddBlackLoss() { blackLosses++;}
public void AddWhiteDraw() { whiteDraws++;}
public void AddBlackDraw() { blackDraws++;}

// get methods
public int GetColour() { return colour; }
public void SetColour(int col) { colour = col; }
public string GetName() { return name; }
public int GetWhiteLosses() { return whiteLosses; }
public int GetBlackLosses() { return blackLosses; }
public int GetDraws() { return whiteDraws+blackDraws; }
public int GetWhiteDraws() { return whiteDraws; }
public int GetBlackDraws() { return blackDraws; }
public int GetWhiteWins() { return whiteWins; }
public int GetBlackWins() { return blackWins; }
public int GetTotalWins() { return blackWins + whiteWins; }
public int GetTotalLosses() { return blackLosses + whiteLosses; }
public int GetID() { return ID; }

public double GetWinrate()
{
    double WR;
    WR = (double)(whiteWins + blackWins) / (double)(whiteLosses + blackLosses + whiteWins + blackWins +
whiteDraws + blackDraws);
    WR *= 100;
}
```

```

        if (double.IsNaN(WR))
        {
            WR = 0;
            if(whiteWins + blackWins > 0) { WR = 100; }
        }
        return WR;
    }

    public DateOnly GetJoinDate()
    {
        return DateOnly.FromDateTime(joinDate);
    }

    public double GetWhiteWinrate()
    {
        double WR;
        WR = (double)whiteWins / (double)(whiteLosses + whiteDraws + whiteWins);
        WR *= 100;
        //eliminate divide by 0 problems
        if (double.IsNaN(WR))
        {
            WR = 0;
            if(whiteWins > 0) { WR = 100; }
        }
        return WR;
    }

    public double GetBlackWinrate()
    {
        double WR;
        WR = (double)blackWins / (double)(blackLosses + blackDraws + blackWins);
        WR *= 100;
        //eliminate divide by 0 problems
        if (double.IsNaN(WR))
        {
            WR = 0;
            if(blackWins > 0) { WR = 100; }
        }
        return WR;
    }

    public int GetTotalGames() { return whiteLosses + whiteWins + blackWins + blackLosses + whiteDraws +
    blackDraws; }
    }
}

```

### Sorter.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ThreeDimensionalChess

```

```
{
    class Sorter
    {

public Sorter() {}

public List<Player> MergeSortString(List<Player> list)
{
    //exit recursion
    if (list.Count() <= 1) { return list; }

    //split list into two
    int midpoint = list.Count() / 2;
    List<Player> left = new List<Player>();
    List<Player> right = new List<Player>();
    for(int x = 0; x < midpoint; x++)
    {
        left.Add(list[x]);
    }
    for(int x = midpoint; x < list.Count(); x++)
    {
        right.Add(list[x]);
    }
    //recurse with each half
    left = MergeSortString(left);
    right = MergeSortString(right);
    List<Player> sorted = MergeString(left, right);
    return sorted;
}

private List<Player> MergeString(List<Player> left, List<Player> right)
{
    List<Player> ret = new List<Player>();
    // combine and sort elements
    while(left.Count() > 0 || right.Count() > 0)
    {
        //check both lists have elements to compare
        if(left.Count() > 0 && right.Count() > 0)
        {
            //compare and reorder elements
            if (left[0].GetName()[0] <= right[0].GetName()[0])
            {
                ret.Add(left[0]);
                left.RemoveAt(0);
            }
            else
            {
                ret.Add(right[0]);
                right.RemoveAt(0);
            }
        }
        //otherwise add remaining contents of list
        else if(left.Count() > 0)
        {
            ret.Add(left[0]);
            left.RemoveAt(0);
        }
    }
}
```

```

    }
    else
    {
        ret.Add(right[0]);
        right.RemoveAt(0);
    }
}
return ret;
}

public List<Player> MergeSortDate(List<Player> list)
{
    //exit recursion
    if (list.Count() <= 1) { return list; }

    //split list into two
    int midpoint = list.Count() / 2;
    List<Player> left = new List<Player>();
    List<Player> right = new List<Player>();
    for (int x = 0; x < midpoint; x++)
    {
        left.Add(list[x]);
    }
    for (int x = midpoint; x < list.Count(); x++)
    {
        right.Add(list[x]);
    }
    //recurse with each half
    left = MergeSortDate(left);
    right = MergeSortDate(right);
    List<Player> ret = MergeDate(left, right);
    return ret;
}

private List<Player> MergeDate(List<Player> left, List<Player> right)
{
    List<Player> ret = new List<Player>();
    // combine and sort elements
    while (left.Count() > 0 || right.Count() > 0)
    {
        //check both lists have elements to compare
        if (left.Count() > 0 && right.Count() > 0)
        {
            //compare and reorder elements
            if (left[0].GetJoinDate() <= right[0].GetJoinDate())
            {
                ret.Add(left[0]);
                left.RemoveAt(0);
            }
            else
            {
                ret.Add(right[0]);
                right.RemoveAt(0);
            }
        }
        //otherwise add remaining contents of list
    }
}

```

```

        else if (left.Count() > 0)
        {
            ret.Add(left[0]);
            left.RemoveAt(0);
        }
        else
        {
            ret.Add(right[0]);
            right.RemoveAt(0);
        }
    }
    return ret;
}

public List<Player> MergeSortWins(List<Player> list)
{
    //exit recursion
    if (list.Count() <= 1) { return list; }

    //split list into two
    int midpoint = list.Count() / 2;
    List<Player> left = new List<Player>();
    List<Player> right = new List<Player>();
    for (int x = 0; x < midpoint; x++)
    {
        left.Add(list[x]);
    }
    for (int x = midpoint; x < list.Count(); x++)
    {
        right.Add(list[x]);
    }
    //recurse with each half
    left = MergeSortWins(left);
    right = MergeSortWins(right);
    List<Player> ret = MergeWins(left, right);
    return ret;
}

private List<Player> MergeWins(List<Player> left, List<Player> right)
{
    List<Player> ret = new List<Player>();
    // combine and sort elements
    while (left.Count() > 0 || right.Count() > 0)
    {
        //check both lists have elements to compare
        if (left.Count() > 0 && right.Count() > 0)
        {
            //compare and reorder elements
            if (left[0].GetWinrate() <= right[0].GetWinrate())
            {
                ret.Add(left[0]);
                left.RemoveAt(0);
            }
            else
            {

```



```

        ret.Add(right[0]);
        right.RemoveAt(0);
    }
    //otherwise add remaining contents of list
    else if (left.Count() > 0)
    {
        ret.Add(left[0]);
        left.RemoveAt(0);
    }
    else
    {
        ret.Add(right[0]);
        right.RemoveAt(0);
    }
}
return ret;
}

public List<Player> MergeSortBlackWR(List<Player> list)
{
    //exit recursion
    if (list.Count() <= 1) { return list; }

    //split list into two
    int midpoint = list.Count() / 2;
    List<Player> left = new List<Player>();
    List<Player> right = new List<Player>();
    for (int x = 0; x < midpoint; x++)
    {
        left.Add(list[x]);
    }
    for (int x = midpoint; x < list.Count(); x++)
    {
        right.Add(list[x]);
    }
    //recurse with each half
    left = MergeSortBlackWR(left);
    right = MergeSortBlackWR(right);
    List<Player> ret = MergeBlackWR(left, right);
    return ret;
}

private List<Player> MergeBlackWR(List<Player> left, List<Player> right)
{
    List<Player> ret = new List<Player>();
    // combine and sort elements
    while (left.Count() > 0 || right.Count() > 0)
    {
        //check both lists have elements to compare
        if (left.Count() > 0 && right.Count() > 0)
        {
            //compare and reorder elements
            if (left[0].GetBlackWinrate() <= right[0].GetBlackWinrate())
            {
                ret.Add(left[0]);
            }

```

```

        left.RemoveAt(0);
    }
    else
    {
        ret.Add(right[0]);
        right.RemoveAt(0);
    }
} //otherwise add remaining contents of list
else if (left.Count() > 0)
{
    ret.Add(left[0]);
    left.RemoveAt(0);
}
else
{
    ret.Add(right[0]);
    right.RemoveAt(0);
}
}
return ret;
}

public List<Player> MergeSortWhiteWR(List<Player> list)
{
    //exit recursion
    if (list.Count() <= 1) { return list; }

    //split list into two
    int midpoint = list.Count() / 2;
    List<Player> left = new List<Player>();
    List<Player> right = new List<Player>();
    for (int x = 0; x < midpoint; x++)
    {
        left.Add(list[x]);
    }
    for (int x = midpoint; x < list.Count(); x++)
    {
        right.Add(list[x]);
    }
    //recurse with each half
    left = MergeSortWhiteWR(left);
    right = MergeSortWhiteWR(right);
    List<Player> ret = MergeWhiteWR(left, right);
    return ret;
}

private List<Player> MergeWhiteWR(List<Player> left, List<Player> right)
{
    List<Player> ret = new List<Player>();
    // combine and sort elements
    while (left.Count() > 0 || right.Count() > 0)
    {
        //check both lists have elemnts to compare
        if (left.Count() > 0 && right.Count() > 0)
        {

```

```

        //compare and reorder elemnts
        if (left[0].GetWhiteWinrate() <= right[0].GetWhiteWinrate())
        {
            ret.Add(left[0]);
            left.RemoveAt(0);
        }
        else
        {
            ret.Add(right[0]);
            right.RemoveAt(0);
        }
    } //otherwise add remaining contents of list
    else if (left.Count() > 0)
    {
        ret.Add(left[0]);
        left.RemoveAt(0);
    }
    else
    {
        ret.Add(right[0]);
        right.RemoveAt(0);
    }
}
return ret;
}

//equivalent sorts copied for GameInfo lists here
public List<GameInfo> MergeSortName(List<GameInfo> list)
{
    //exit recursion
    if (list.Count() <= 1) { return list; }

    //split list into two
    int midpoint = list.Count() / 2;
    List<GameInfo> left = new List<GameInfo>();
    List<GameInfo> right = new List<GameInfo>();
    for (int x = 0; x < midpoint; x++)
    {
        left.Add(list[x]);
    }
    for (int x = midpoint; x < list.Count(); x++)
    {
        right.Add(list[x]);
    }
    //recurse with each half
    left = MergeSortName(left);
    right = MergeSortName(right);
    List<GameInfo> sorted = MergeName(left, right);
    return sorted;
}

private List<GameInfo> MergeName(List<GameInfo> left, List<GameInfo> right)
{
    List<GameInfo> ret = new List<GameInfo>();
    // combine and sort elements

```

```

while (left.Count() > 0 || right.Count() > 0)
{
    //check both lists have elemnts to compare
    if (left.Count() > 0 && right.Count() > 0)
    {
        //compare and reorder elemnts
        if (left[0].GetName()[0] <= right[0].GetName()[0])
        {
            ret.Add(left[0]);
            left.RemoveAt(0);
        }
        else
        {
            ret.Add(right[0]);
            right.RemoveAt(0);
        }
    }
    //otherwise add remaining contents of list
    else if (left.Count() > 0)
    {
        ret.Add(left[0]);
        left.RemoveAt(0);
    }
    else
    {
        ret.Add(right[0]);
        right.RemoveAt(0);
    }
}
return ret;
}

public List<GameInfo> MergeSortDate(List<GameInfo> list)
{
    //exit recursion
    if (list.Count() <= 1) { return list; }

    //split list into two
    int midpoint = list.Count() / 2;
    List<GameInfo> left = new List<GameInfo>();
    List<GameInfo> right = new List<GameInfo>();
    for (int x = 0; x < midpoint; x++)
    {
        left.Add(list[x]);
    }
    for (int x = midpoint; x < list.Count(); x++)
    {
        right.Add(list[x]);
    }
    //recurse with each half
    left = MergeSortDate(left);
    right = MergeSortDate(right);
    List<GameInfo> ret = MergeDate(left, right);
    return ret;
}

```

```
private List<GameInfo> MergeDate(List<GameInfo> left, List<GameInfo> right)
{
    List<GameInfo> ret = new List<GameInfo>();
    // combine and sort elements
    while (left.Count() > 0 || right.Count() > 0)
    {
        //check both lists have elements to compare
        if (left.Count() > 0 && right.Count() > 0)
        {
            //compare and reorder elements
            if (left[0].GetLastAccessed() <= right[0].GetLastAccessed())
            {
                ret.Add(left[0]);
                left.RemoveAt(0);
            }
            else
            {
                ret.Add(right[0]);
                right.RemoveAt(0);
            }
        }
        //otherwise add remaining contents of list
        else if (left.Count() > 0)
        {
            ret.Add(left[0]);
            left.RemoveAt(0);
        }
        else
        {
            ret.Add(right[0]);
            right.RemoveAt(0);
        }
    }
    return ret;
}

public List<GameInfo> MergeSortState(List<GameInfo> list)
{
    //exit recursion
    if (list.Count() <= 1) { return list; }

    //split list into two
    int midpoint = list.Count() / 2;
    List<GameInfo> left = new List<GameInfo>();
    List<GameInfo> right = new List<GameInfo>();
    for (int x = 0; x < midpoint; x++)
    {
        left.Add(list[x]);
    }
    for (int x = midpoint; x < list.Count(); x++)
    {
        right.Add(list[x]);
    }
    //recurse with each half
    left = MergeSortState(left);
    right = MergeSortState(right);
}
```

```
List<GameInfo> sorted = MergeState(left, right);
return sorted;
}

private List<GameInfo> MergeState(List<GameInfo> left, List<GameInfo> right)
{
    List<GameInfo> ret = new List<GameInfo>();
    // combine and sort elements
    while (left.Count() > 0 || right.Count() > 0)
    {
        //check both lists have elements to compare
        if (left.Count() > 0 && right.Count() > 0)
        {
            //compare and reorder elements
            if (left[0].GetGamestateForComparison() <= right[0].GetGamestateForComparison())
            {
                ret.Add(left[0]);
                left.RemoveAt(0);
            }
            else
            {
                ret.Add(right[0]);
                right.RemoveAt(0);
            }
        }
        //otherwise add remaining contents of list
        else if (left.Count() > 0)
        {
            ret.Add(left[0]);
            left.RemoveAt(0);
        }
        else
        {
            ret.Add(right[0]);
            right.RemoveAt(0);
        }
    }
    return ret;
}

public List<GameInfo> MergeSortWhitePlayerName(List<GameInfo> list)
{
    //exit recursion
    if (list.Count() <= 1) { return list; }

    //split list into two
    int midpoint = list.Count() / 2;
    List<GameInfo> left = new List<GameInfo>();
    List<GameInfo> right = new List<GameInfo>();
    for (int x = 0; x < midpoint; x++)
    {
        left.Add(list[x]);
    }
    for (int x = midpoint; x < list.Count(); x++)
    {
        right.Add(list[x]);
    }
}
```

```

    }
    //recurse with each half
    left = MergeSortWhitePlayerName(left);
    right = MergeSortWhitePlayerName(right);
    List<GameInfo> sorted = MergeWhitePlayerName(left, right);
    return sorted;
}

private List<GameInfo> MergeWhitePlayerName(List<GameInfo> left, List<GameInfo> right)
{
    List<GameInfo> ret = new List<GameInfo>();
    // combine and sort elements
    while (left.Count() > 0 || right.Count() > 0)
    {
        //check both lists have elements to compare
        if (left.Count() > 0 && right.Count() > 0)
        {
            //compare and reorder elements
            if (left[0].GetWhitePlayerName()[0] <= right[0].GetWhitePlayerName()[0])
            {
                ret.Add(left[0]);
                left.RemoveAt(0);
            }
            else
            {
                ret.Add(right[0]);
                right.RemoveAt(0);
            }
        }
        //otherwise add remaining contents of list
        else if (left.Count() > 0)
        {
            ret.Add(left[0]);
            left.RemoveAt(0);
        }
        else
        {
            ret.Add(right[0]);
            right.RemoveAt(0);
        }
    }
    return ret;
}

public List<GameInfo> MergeSortBlackPlayerName(List<GameInfo> list)
{
    //exit recursion
    if (list.Count() <= 1) { return list; }

    //split list into two
    int midpoint = list.Count() / 2;
    List<GameInfo> left = new List<GameInfo>();
    List<GameInfo> right = new List<GameInfo>();
    for (int x = 0; x < midpoint; x++)
    {
        left.Add(list[x]);
    }

```

```

    }
    for (int x = midpoint; x < list.Count(); x++)
    {
        right.Add(list[x]);
    }
    //recurse with each half
    left = MergeSortBlackPlayerName(left);
    right = MergeSortBlackPlayerName(right);
    List<GameInfo> sorted = MergeBlackPlayerName(left, right);
    return sorted;
}

private List<GameInfo> MergeBlackPlayerName(List<GameInfo> left, List<GameInfo> right)
{
    List<GameInfo> ret = new List<GameInfo>();
    // combine and sort elements
    while (left.Count() > 0 || right.Count() > 0)
    {
        //check both lists have elements to compare
        if (left.Count() > 0 && right.Count() > 0)
        {
            //compare and reorder elements
            if (left[0].GetBlackPlayerName()[0] <= right[0].GetBlackPlayerName()[0])
            {
                ret.Add(left[0]);
                left.RemoveAt(0);
            }
            else
            {
                ret.Add(right[0]);
                right.RemoveAt(0);
            }
        }
        //otherwise add remaining contents of list
        else if (left.Count() > 0)
        {
            ret.Add(left[0]);
            left.RemoveAt(0);
        }
        else
        {
            ret.Add(right[0]);
            right.RemoveAt(0);
        }
    }
    return ret;
}

public List<Player> Reverse(List<Player> list)
{
    //reverse a list simply, using a stack
    Stack<Player> stck = new Stack<Player>();
    for(int x = 0; x < list.Count(); x++)
    {
        stck.Push(list[x]);
    }
}

```



```

    List<Player> ret = new List<Player>();
    for (int x = 0; x < list.Count(); x++)
    {
        ret.Add(stck.Pop());
    }
    return ret;
}

public List<GameInfo> Reverse(List<GameInfo> list)
{
    //reverse a list simply, using a stack
    Stack<GameInfo> stck = new Stack<GameInfo>();
    for (int x = 0; x < list.Count(); x++)
    {
        stck.Push(list[x]);
    }
    List<GameInfo> ret = new List<GameInfo>();
    for (int x = 0; x < list.Count(); x++)
    {
        ret.Add(stck.Pop());
    }
    return ret;
}
}
}

```

## Data Structures Implementation

### List.cs

```

using System;
using System.Collections;

namespace ThreeDimensionalChess
{
    class List<T>
    {
        //head node poiting to nothing by default
        private ListNode<T> head = null;

        public List() { }
        public List(T inp)
        {
            Add(inp);
        }
        //constructor that takes an array as input
        public List(T[] arr)
        {
            foreach (T o in arr) { Add(o); }
        }

        //adds to end of list
        public void Add(T inp)
        {

```

```

ListNode<T> tmp = head;
//checks if head is null, if it is it starts the list there with a new node
if (head != null)
{
    //moves through to end of list
    while (tmp.next != null)
    {
        tmp = tmp.next;
    }
    //sets next reference to a new node
    tmp.next = new ListNode<T>(inp);
}
else { head = new ListNode<T>(inp); }

}

//ALLOWS THE LIST TO BE ACCESSED BY INDEXERS
public T this[int i]
{
    //uses private methods
    get { return RetrieveAt(i, head); }
    set { SetAt(i, head, value); }
}

//private method for retrieving data, adjusted to use while loops like SetAt()
private T RetrieveAt(int i, ListNode<T> node)
{
    if (i < 0) { throw new IndexOutOfRangeException(); }
    ListNode<T> tmp = node;
    while (i > 0)
    {
        //if tmp points to null then that means index has run out of bounds
        if (tmp == null) { throw new IndexOutOfRangeException(); }
        tmp = tmp.next;
        i--;
    }
    return tmp.GetData();
}

//private set method
private void SetAt(int i, ListNode<T> node, T inp)
{
    if (i < 0) { throw new IndexOutOfRangeException(); }
    ListNode<T> tmp = node;
    //runs while i >= 0, fine to do this here since using ints
    while (i > 0)
    {
        //if tmp is empty then that means index has run out of bounds
        if (tmp == null) { throw new IndexOutOfRangeException(); }
        tmp = tmp.next;
        i--;
    }
    tmp.SetData(inp);
}

```

```

public T RemoveAt(int i)
{
    ListNode<T> tmp = head;
    T ret = default;
    if (i == 0)
    {
        if (tmp == null) { throw new ArgumentOutOfRangeException(); }
        ret = head.GetData();
        head = tmp.next;
    }
    else
    {
        i--;
        //RETRIEVES THE LIST ITEM BEFORE THE ONE TO REMOVE
        while (i > 0)
        {
            //if tmp is empty that means index is out of bounds
            if (tmp == null) { throw new ArgumentOutOfRangeException(); }
            tmp = tmp.next;
            i--;
            if (tmp.next == null) { throw new ArgumentOutOfRangeException(); }
        }
        //dereferences next node and sets reference to next node over
        ret = tmp.next.GetData();
        tmp.next = tmp.next.next;
    }

    return ret;
}

//wanted to have these two options for count but functionality's different, so:
public int Count()
{
    return Length(head);
}

public int Count(T inp)
{
    return Search(head, inp);
}

public bool Contains(T inp)
{
    //checks if it contains
    bool ret = false;
    if (this.Count(inp) > 0)
    {
        ret = true;
    }
    return ret;
}

//recursive, looks a bit messy but it's built on the Length method so it's alright really
private int Search(ListNode<T> node, T inp)

```

```

{
    //if node == null list is empty or somehow got out of bounds or is missing a node, most normally an
    empty list or the end of one so returns 0
    if (node != null)
    {
        if (node.next != null)
        {
            //only increments return, via recursion, when data == input
            if (node.GetData().Equals(inp))
            {
                return 1 + Search(node.next, inp);
            }
            else { return Search(node.next, inp); }
        }
        else
        {
            //if we are here, next node is null, therefore recursion unspools
            if (node.GetData().Equals(inp))
            {
                return 1;
            }
            else { return 0; }
        }
    }
    else { return 0; }
}

private int Length(ListNode<T> node)
{
    //if node == null, the list is empty
    if (node != null)
    {
        return 1 + Length(node.next);
    }
    else { return 0; }
}

public void AddFront(T inp)
{
    //creates new node of input value
    ListNode<T> a = new ListNode<T>(inp);
    //points new node towards current head so it isn't lost when head now points at our new node
    a.next = head;
    head = a;
}

//use this for comparisons
public string ConvertToString()
{
    string tmpstr = "";
    for (int x = 0; x < Count(); x++)
    {
        tmpstr = tmpstr + Convert.ToString(RetrieveAt(x, head));
        //if not the last record add a comma, can be used to split list later !!!
        if (x != Count() - 1) { tmpstr = tmpstr + ","; }
    }
}

```

```

    }
    return tmpstr;
}

public void Insert(int i, T inp)
{
    if (i < 0 || i > this.Count()) { throw new ArgumentOutOfRangeException(); }

    //needs to handle insert at 0 differently
    if (i == 0)
    {
        this.AddFront(inp);
    }
    else
    {
        //needs to get node before where to insert
        i--;
        ListNode<T> tmp = head;
        while (i > 0)
        {
            tmp = tmp.next;
            i--;
        }
        //swaps things around without accidentally derefencing
        ListNode<T> n = new ListNode<T>(inp);
        n.next = tmp.next;
        tmp.next = n;
    }
}
}
}

```

### ListNode.cs

```

using System;

namespace ThreeDimensionalChess
{
    class ListNode<T>
    {
        private T data;
        public ListNode<T> next = null;

        //constructor
        public ListNode(T inp)
        {
            data = inp;
        }

        //get set methods
        public T GetData() { return data; }
        public void SetData(T inp) { data = inp; }
    }
}

```

Stack.cs

```
using System;
using System.Text;

namespace ThreeDimensionalChess
{
    class Stack<T>
    {
        //using a List as the stack
        List<T> stack = new List<T>();

        //constructors, in case of array items are pushed to stack 0 onwards
        public Stack() { }
        public Stack(T inp)
        {
            this.Push(inp);
        }
        public Stack(T[] inp)
        {
            foreach (T a in inp) { Push(a); }
        }

        //adds an item to the top of the stack
        public void Push(T inp)
        {
            stack.Add(inp);
        }

        //gets top item without removing it
        public T Peek()
        {
            T ret = default;
            //checks stack isn't empty
            if (stack.Count() > 0)
            {
                ret = stack[stack.Count() - 1];
            }
            return ret;
        }

        //returns top item on the stack and removes it
        public T Pop()
        {
            T ret = default;
            //checks stack isn't empty
            if (stack.Count() > 0)
            {
                ret = stack.RemoveAt(stack.Count() - 1);
            }
            return ret;
        }

        public bool Contains(T inp)
```

```
{
    //stacks are so similar to lists you can just do stuff like this??
    return stack.Contains(inp);
}























//returns number of items in the stack
public int Count()
{
    return stack.Count();
}

public bool IsEmpty()
{
    bool empty = false;
    if (Count() == 0)
    {
        empty = true;
    }
    return empty;
}

public string ConvertToString()
{
    return stack.ConvertToString();
}











public Stack<T> Clone()
{
    Stack<T> ret = new Stack<T>();
    for(int i = 0; i < stack.Count(); i++)
    {
        ret.Push(stack[i]);
    }
    return ret;
}
}
}
```

### File Structure

 resources	11/12/2023 08:27	File folder	
 Bishop.cs	04/12/2023 09:07	C# Source File	5 KB
 Board.cs	05/02/2024 11:12	C# Source File	19 KB
 Chess.cs	28/02/2024 11:40	C# Source File	19 KB
 DatabaseHandler.cs	28/02/2024 08:50	C# Source File	14 KB
 GameInfo.cs	27/02/2024 11:05	C# Source File	4 KB
 King.cs	04/12/2023 09:07	C# Source File	6 KB
 Knight.cs	04/12/2023 09:07	C# Source File	2 KB
 List.cs	26/02/2024 11:29	C# Source File	8 KB
 ListNode.cs	28/11/2023 10:37	C# Source File	1 KB
 Pawn.cs	04/12/2023 09:07	C# Source File	9 KB
 Piece.cs	26/02/2024 11:06	C# Source File	7 KB
 Player.cs	07/12/2023 09:44	C# Source File	4 KB
 Queen.cs	04/12/2023 09:07	C# Source File	3 KB
 Rook.cs	04/12/2023 09:07	C# Source File	5 KB
 SimulatedBoard.cs	04/12/2023 09:07	C# Source File	4 KB
 Sorter.cs	04/12/2023 09:07	C# Source File	22 KB
 Square.cs	04/12/2023 09:07	C# Source File	3 KB
 Stack.cs	04/12/2023 11:06	C# Source File	3 KB
 ThreatSuperPiece.cs	04/12/2023 09:07	C# Source File	21 KB
 ThreeDimensionalChess.csproj	11/12/2023 08:27	C# Project file	3 KB
 UserInterface.cs	27/02/2024 15:36	C# Source File	81 KB

The files follow a loose organisation, they are all in one folder, but each file is only one class and all their relevant methods or constants. Then there is a resources subfolder which contains the textures for the pieces and any audio files that are needed.

When the project is compiled to the Binaries folder, the resources folder is also copied so that project's executable can continue to access the resources locally. Furthermore, upon its first run, the executable produces a database.db file which will remain in the binaries folder and be accessed later.

 resources	28/02/2024 10:44	File folder	
 runtimes	27/02/2024 11:29	File folder	
 database.db	28/02/2024 10:45	Data Base File	20 KB
 Raylib-cs.dll	05/11/2023 13:19	Application extens...	87 KB
 System.Data.SQLite.dll	10/06/2023 21:02	Application extens...	418 KB
 ThreeDimensionalChess.deps.json	27/02/2024 11:29	JSON File	5 KB
 ThreeDimensionalChess.dll	28/02/2024 10:44	Application extens...	74 KB
 ThreeDimensionalChess.exe	28/02/2024 10:44	Application	147 KB
 ThreeDimensionalChess.pdb	28/02/2024 10:44	Program Debug D...	57 KB
 ThreeDimensionalChess.runtimeconfig...	27/02/2024 11:29	JSON File	1 KB

## Evaluation and Testing



## Test Plan

To carry out testing and ensure that all requirements have been fulfilled, we need to test our features with three different kinds of input:

- Normal Cases – And see that the intended output is reached.
- Boundary Cases – And see that the intended output is reached.
- Erroneous Cases – And make sure the program handles invalid inputs correctly.

## Menu Tests

Test ID	Purpose	Testing Method	Type	Expected Outcome
1	Checks that menu buttons take user to the right page.	Click on buttons on menu	Normal	Players – Players Table Play – New/Load Game Choice New Game – New Game Screen Load Game – Games Table Exit – Closes program
2	Check player creation works	From Players Table, click on Add Player button and enter information	Normal	New player is present in players table after creation
3	Check player creation works via alternative route	From New Game screen, click on plus button beside drop down to create a player and enter information	Normal	New player is present in players table after creation
4	Check game creation works correctly	Enter all necessary information in New Game screen, create game then quit to menu and navigate to Load Game screen	Normal	Created game is present with correct information in the games table on the Load Game screen
5	Check game deletion works correctly	Click on a game's row and then click on the delete button	Normal	Game is deleted from database and therefore no longer visible in Load Game, games table
6	Check player deletion works correctly	(Have a game created with the player to be deleted as one of the players) Navigate to Players table and click on a row of the player, then click the delete button	Normal	Player is deleted from database and therefore no longer visible in Players table. Any games containing the player are removed from the database and therefore not visible in the games table in the Load Game screen

7	Check undo moves choice works	From the new game menu, uncheck undo moves and then create the game	Normal	When the new game is created there should be no undo move button under the move list
---	-------------------------------	---	--------	--

### Game Tests

Test ID	Purpose	Testing Method	Type	Expected Outcome
8	Check that board is represented correctly in 2D	Step through all layers of the board from all view directions	Normal	Squares are coloured correctly, pieces are represented in expected positions
9	Check that the constraints of the board work correctly	Try to step out the bottom and top of the board, using the arrow buttons	Erroneous	Nothing changes
10	Check that board is represented correctly in 3D	Click on 3D button in the game's UI	Normal	The 3D representation of the board is shown, pieces in correct position
11	Check that rook generates moves correctly	Click on a rook in the 2D UI to select it, then click on the 3D button to see the highlighted squares	Normal	Rook only generates possible moves, excluding self-check, friendly captures and physically impossible moves
12	Check that the bishop generates moves correctly	Click on a bishop in the 2D UI to select it, then click on the 3D button to see the highlighted squares	Normal	Bishop only generates possible moves, excluding self-check, friendly captures and physically impossible moves
13	Check that the Queen generates moves correctly	Click on a queen in the 2D UI to select it, then click on the 3D button to see the highlighted squares	Normal	Queen only generates possible moves, excluding self-check, friendly captures and physically impossible moves
14	Check that the Knight generates moves correctly	Click on a knight in the 2D UI to select it, then click on the 3D button to see the highlighted squares	Normal	Knight only generates possible moves, excluding self-check, friendly captures and physically impossible moves
15	Check that the King generates moves correctly	Click on a King in the 2D UI to select it, then click on the 3D button to see the highlighted squares	Normal	King only generates possible moves, excluding self-check, friendly captures and physically impossible moves

16	Check that a White Pawn generates moves correctly	Click on a white pawn in the 2D UI to select it, then click on the 3D button to see the highlighted squares	Normal	The White Pawn only generates possible moves, excluding self-check, friendly captures and physically impossible moves
17	Check that a Black Pawn generates moves correctly	Click on a black pawn in the 2D UI to select it, then click on the 3D button to see the highlighted squares	Normal	The Black Pawn only generates possible moves, excluding self-check, friendly captures and physically impossible moves
18	Check that moving a piece works correctly and makes a sound	After having clicked on a friendly piece, click on a blue highlighted square to move the piece	Normal	The image of the piece moves to the new square and the list of moves on the UI is updated. A sound is made with each move.
19	Check that moving a piece works correctly	After having clicked on an enemy piece, click on a red highlighted square	Erroneous	Squares are no longer highlighted
20	Check that moving a piece works correctly	After having clicked on any piece, click on a non-highlighted square	Erroneous	Squares are no longer highlighted
21	Check that capturing works correctly and makes a different sound	After having clicked on a friendly piece, click on an enemy piece that can be captured (highlighted in blue)	Normal	The image of the piece moves to the clicked-on square, the captured piece is no longer present, the list of moves on the UI is updated with capture notation. The capture sound plays.
22	Checks that check works	After having clicked on a friendly piece, click on square that would make a move that puts the enemy King in check	Normal	The image of the piece moves to the new square and the list of moves on the UI is updated with correct notation for Check
23	Check that checkmate works	Make a series of moves, with the objective of a King being checkmated (thereby control both pieces by one user)	Normal	No more input on the board works after the checkmating move, the winning player's name is displayed, the move list is updated with the move and correct notation for checkmate. In the database, visible from the load game table, the game

				outcome is displayed in the correct column
24	Check that stalemate works	Make a series of moves, with objective of reaching stalemate (control both pieces as one user)	Normal	No more input works after final move, the stalemate text is displayed, the most list is updated with the correct notation for stalemate. In the database, visible from the load game table, the outcome of the game is correctly recorded as stalemate
25	Check pause menu functionality	Click on buttons in pause menu	Normal	Resume – Game is resumed Exit to Menu – User is returned to Main Menu screen Exit to Desktop – Program closes
26	Check forfeit/agree to draw functionality	Click on buttons to confirm forfeit	Normal	User is returned to game screen, no more input can be taken on board, the move list is updated with Black#, White# or Stalemate to indicate the agreed outcome. In the database, visible from the load game table, the game's outcome is correctly recorded
27	Check undoing of moves works correctly	Make a move as per above tests, then click on the undo move button	Normal	The image of the piece is returned to the square where it was before, the move list is updated, the counter reduced by one and the move deleted, the database reflects this.
28	Check undoing of a capture move works correctly	Make a capture as per above tests, then click on the undo move button	Normal	The image of the piece is returned to where it was before, the captured piece is restored on the end square. The move list is updated, the counter reduced by one and the latest move is deleted, the database reflects this
29	Check undoing of a promotion works correctly	Move a pawn onto the last rank and select a piece to promote it to. Then click the undo move button	Normal	The move is undone, meaning that the piece is demoted to a pawn and moved back to the square it moved from. Database

				and move list reflect this as above
30	Check undoing of an incomplete promotion works correctly	Move a pawn onto the last rank but do not select a piece for it to promote to. Then click the undo move button	Boundary	The pawn moves back to the square it started on with no side effects. Database and move list reflect this as above
31	Check loading of an ongoing game works correctly	From the load game menu, click on a game's row then click on the load game button. Then click on the confirm button	Normal	The game is loaded, pieces are in the same position as before, the move list is filled in with the list of moves
32	Check undo moves works on loaded games and writes to database correctly	After having loaded a game (test 30) click on the undo move button	Normal	The last move is undone and satisfies test 27, 28 or 29
33	Check loading of a completed game works correctly	From the load game menu, click on the row of a completed game, then click load game, then click confirm	Normal	The game loads, the last move which finished the game is automatically undone, the move list reflects this and a new entry in the database, as visible from load game table, is present from this point of the game.
34	Check pawn promotion works	Move a pawn to the final rank, select a piece to promote it to	Normal	The pawn is replaced by the select piece, the move list is updated with the correct promotion notation
35	Check loading a game with an incomplete promotion works	Move a pawn to the final rank but don't select a piece for it to promote to. Exit to menu and then navigate to attempt to load the game	Boundary	The game should load as if the pawn hasn't been moved yet

Table Tests

Test ID	Purpose	Testing Method	Type	Expected Outcome
36	Check sorting of players based on name works	From the Players menu, click on the name column header. After demonstrating that, click on it again to demonstrate	Normal	The same list but sorted in order of name. After the second click the same list but reversed

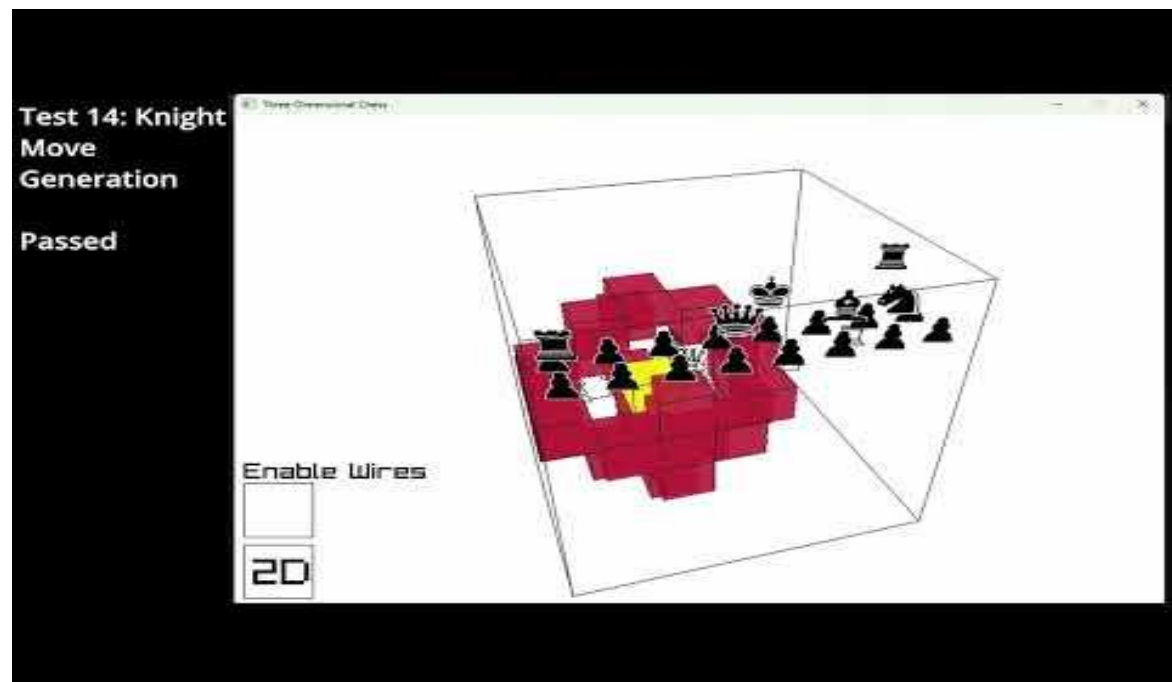
		reverse functionality		
37	Check sorting of players based on date created works	From the Players menu, click on the date column header. After demonstrating that, click on it again to demonstrate reverse functionality	Normal	The same list but sorted in order of date created. After the second click the same list but reversed
38	Check sorting of players based on win-rate works	From the Players menu, click on the W/L/D column header. After demonstrating that, click on it again to demonstrate reverse functionality	Normal	The same list but sorted in order of win-rate. After the second click the same list but reversed
39	Check sorting of players based on white win-rate works	From the Players menu, click on the White WR column header. After demonstrating that, click on it again to demonstrate reverse functionality	Normal	The same list but sorted in order of white win-rate. After the second click the same list but reversed
40	Check sorting of players based on black win-rate works	From the Players menu, click on the Black WR column header. After demonstrating that, click on it again to demonstrate reverse functionality	Normal	The same list but sorted in order of black win-rate. After the second click the same list but reversed
41	Check sorting of games based on name works	From the load game menu, click on the name column header. After	Normal	The same list but sorted in order of name. After the second click the same list but reversed

		demonstrating that, click on it again to demonstrate reverse functionality		
42	Check sorting of games based on game-state works	From the load game menu, click on the state column header. After demonstrating that, click on it again to demonstrate reverse functionality	Normal	The same list but sorted in order of game-state. After the second click the same list but reversed
43	Check sorting of games based on date created works	From the load game menu, click on the date column header. After demonstrating that, click on it again to demonstrate reverse functionality	Normal	The same list but sorted in order of date created. After the second click the same list but reversed
44	Check sorting of games based on player name works	From the load game menu, click on the White or Black column header. After demonstrating that, click on it again to demonstrate reverse functionality	Normal	The same list but sorted in order of the names of the selected column. After the second click the same list but reversed

### Test Results

### Testing Video

<https://youtu.be/Ex04t4At9BE>



Test ID	Outcome	Timestamp	Additional Comments
1	Passed	0:00-0:08	
2	Passed	0:09-0:20	
3	Passed	0:21-0:30	
4	Passed	0:31-0:46	Here we see the process of creating a game and then the evidence of it in the Load Game table
5	Passed	0:47-0:55	Here I also refresh the Load Game table to further evidence the game has been correctly deleted
6	Passed	9:39-9:49	Any game including the deleted player has also been deleted; this list can be seen prior in the vide at 7:55
7	Passed	9:04-9:10	Note absence of undo moves button
8	Passed	0:57-1:18	Note the coordinate indicator at the bottom of the viewport
9	Passed	1:18-1:24	It's hard to tell but those are clicks on the greyed out triangles
10	Passed	1:25-1:38	Also demonstrated cube border functionality
11	Passed	1:39-1:54	Demonstrates the functionality of move highlighting too, based on player's turn. Shown in 3D for clarity.
12	Passed	2:06-2:23	Shown in 3D for clarity.
13	Passed	2:24-2:52	Shown in 3D for clarity.
14	Passed	2:53-3:18	Shown in 3D for clarity.
15	Passed	3:19-3:44	Shown in 3D for clarity. Also shows edge cases including adjacent kings.
16	Passed	3:45-4:02	Shown in 3D for clarity. Also demonstrates capturing moves.
17	Passed	4:03-4:17	Shown in 3D for clarity. Also demonstrates capturing moves.
18	Passed	1:55-2:05	Is also demonstrated many times throughout the video
19	Passed	4:19-4:22	
20			
21	Passed	4:23-4:25	Note also that the capture sound can be heard



22	Passed	4:26-4:34	Note the check notation at the top of the move list. Also demonstrates that check can be properly cancelled by capturing the checking piece
23	Passed	5:25-5:30	The minute of video prior to this is gameplay to achieve the test scenario
24	Passed	6:41-6:52	The minute of video prior to this is gameplay to achieve the test scenario
25	Passed	7:00-7:04	It appears that in this section the Return to Menu button is not tested due to an unfortunate video cut, therefore I would like to clarify that it is also demonstrated at 7:28.
26	Passed	7:06-7:31	Note at the end the games and their outcomes: pause menu – Black W forfeit 2 – White W agree to draw – Draw
27	Passed	8:05-8:08	This is also demonstrated many times throughout the video
28	Passed	7:32-7:39	
29	Passed	7:43-7:46	
30	Passed	7:47-7:51	
31	Passed	7:52-8:03	Again, uses the 3D representation to demonstrate that the positions of all pieces are correct
32	Passed	8:04-8:17	
33	Passed	5:30-5:42	Last move is undone, shown in move list and also demoed by the use of the 3D representation
34	Passed	7:40-7:43	Note the correct notation for promotion is also shown at the top of the move list
35	Passed	8:38-9:03	Start timestamp includes the setup which is more important for this test than others (where time of setup has been excluded)
36	Passed	9:11-9:16	
37	Passed	9:16-9:20	
38	Passed	9:20-9:23	
39	Passed	9:23-9:25	
40	Passed	9:25-9:28	
41	Passed	9:28-9:31	
42	Passed	9:31-9:33	
43	Passed	9:33-9:35	
44	Passed	9:35-9:38	

### Evaluation Plan

In order to assess whether the design and brief of the project has been fulfilled, I will inspect different aspects of the project. In order, this will be:

1. An evaluation of whether the requirements have been met. This should all be covered by the test too.
2. Interview with an end user. I will be gathering feedback from the same potential users that influenced the design and they will assess to what extent the project has met their expectations.
3. A summary of what I think went well.
4. A summary of potential improvements, given more time or a different approach.

### Evaluation of Requirements

## Board and UI

1.0 - Store an 8x8x8 three dimensional board (512 cells total)

1.01a - Adds squares from left to right, bottom to top, then near to far.

1.01b - Square 0 is black(A1), square one is white(B1): Each odd square is white (Alternate the starting square colour of each row)

1.2 - Pieces are represented on the board by icons, they are limited to one square

1.5 - There should be a list of past moves represented in a list from the most recent move to the first move

1.51 - The move representation should use chess notation, with the last 8 letters of the alphabet to represent the coordinates in the third dimension

These requirements are trivial and constantly demonstrated in all interactions with the board.

1.1 - Ability to look at board from front(z coordinate is constant), side(x coordinate is constant), top(y coordinate is constant)

1.11a - User can view each layer of the board from any of the angles(front, side, top)

1.11b - Each layer is an 8x8 representation of the board

1.11c - User can move up or down a layer by a pair of opposite arrow buttons

These requirements are demonstrated as fulfilled by [test 8](#) which was passed.

1.3 - When the user clicks on a friendly piece, squares that the piece could possibly move to should be tinted in blue (In accordance with piece moving rules and 2.03)

1.4 - When the user clicks on an enemy piece, squares that the piece could possibly move to should be tinted in red(In accordance with piece moving rules and 2.03)

This requirement is fulfilled and most clearly visible in the move generation [tests \(11 – 17\)](#), as well as being present all throughout the video.

1.31a - If the user then click on one of the highlighted blue squares, the piece should be moved to that square

1.31b - The icon representing the piece should overwrite what is currently on the cell that is being moved to

1.31c - A sound should play when a piece is moved onto a square:

A hollow wooden sound for moving onto an empty square

A solid wooden sound for taking an opponent's piece

1.6 - After each move, the square a piece started on and the square that a piece finished in, should be shaded in yellow

1.61 - Reset this for each move so only one pair of squares is shaded yellow at a time

Whilst some of these widely demonstrated throughout the testing video, it would make sense to link them under [tests 18 – 21](#) where the fulfilment of these requirements is demonstrated purposefully.

1.7 - There should be a button which changes the UI to a screen showing a 3D representation of the board

1.71 - Any piece selected on the 2D view will have its possible moves shown on the 3D representation

1.72 - Include a toggle to show the outlines of every square otherwise only draw outlines of possible move squares

1.73 - Show the square moved to and from in yellow

1.74 - Have a button to return the view to the 2D representation of the board

These results of fulfilling these requirements are explicitly shown in [test 10](#) but are also very present throughout the rest of the demonstration.

### Pieces

2.0 - There should be a King piece

2.01 - The King may move one square in any direction, this includes diagonally by one square on one plane

These requirements are fulfilled in the implementation of the King piece and tested in [test 15](#).

2.02 - If a piece of the opposite colour may move onto the King's square (at the end of the opponent's turn) then the King is in check

2.02a - If the King is in check there are multiple resolutions:

The King may move to a square that is not threatened by an opponent's piece

The piece threatening the King can be taken, provided this doesn't put the King in check

A piece may move to block the line of cells towards the king, intercepting the line of check

2.03 - A player may not put their own King in check, either by moving the King or another of their pieces

2.04 - If the King cannot escape check by any of the listed resolutions, it is checkmate and the player of the trapped King has lost, the other player winning

2.05 - The UI notifies a player if their King is in check with a line of texts

These auxiliary requirements are crucial to the game however we don't see them as much due to them being near to the end of the game, therefore they have been targeted by specific tests: [22](#) and [23](#)

2.1 - There should be a Pawn piece

2.11a - A white Pawn can move up the board or forwards on the z axis by one cell

2.11b - A black Pawn can move down the board or backwards on the z axis by one cell (from white's perspective)

2.12a - It may not move over another piece, it must stop on the cell before it

2.12b - It may not finish its move on the same square as a piece of the same colour

2.13 - A Pawn can take an enemy piece by moving one cell diagonally (on one plane) and finishing its turn on the cell of the piece of the opposite colour

2.13a - A Pawn can only move diagonally to take a piece

2.13b - This move must still be towards the promotion rank, so white pawns cannot capture diagonally one deep and one down; black pawns cannot capture diagonally one deep and one up.

As above with the King, even though the method of move generation is the most complex in the specification and the algorithm for generating them is made of many conditions. The method for testing is the same, simply twice for each colour in [tests 16 and 17](#).

2.14 - If a Pawn reaches the end of the board, row **8z** for any white pawn and row **1s** for any black pawn, it is promoted. The player chooses any piece other than a King or Pawn for it to become.

Like pawn move generation this requires a lot of conditions and boundary conditions, but it was fulfilled successfully as shown in [test 34](#).

2.2 - There should be a Rook piece

2.21a- The Rook can move any number of cells in a straight line (parallel with the x, y or z axis)

2.21b - It may take a piece of the opposite colour by finishing its move on that piece's square

- 2.21c - It may not move over pieces, it can only move up to that piece
- 2.21d - It may not finish its turn on the same square as a piece of the same colour
- 2.21e - It may not move over the edges of the board, it can only move up to the edge of the board

2.3- There should be a Bishop piece

- 2.31a - The Bishop can move any number of cells diagonally on one plane
- 2.31b - It may take a piece of the opposite colour by finishing its move on that piece's square
- 2.31c - It may not move over pieces, it can only move up to that piece
- 2.31d - It may not finish its turn on the same square as a piece of the same colour
- 2.31e - It may not move over the edges of the board, it can only move up to the edge of the board

2.4 - There should be a Knight piece

- 2.41a - The Knight can move two cells on one axis and then one cell on a different axis
- 2.41b - It may move any direction on this axis, for example, up or down on Y-Axis
- 2.41c - It may take a piece of the opposite colour by finishing its move on that piece's square
- 2.41d - It may move over pieces of any colour
- 2.41e - It may not end its move on a piece of the same colour
- 2.41f - It may not move over the edges of the board, moves that end off the board are not valid

2.5 - There should be a Queen piece

- 2.51a- The Queen may move as a combination of the Rook and the Bishop
  - 2.21a- The Queen can move any number of cells in a straight line (parallel with the x, y or z axis)
  - 2.31- The Queen can move any number of cells diagonally on one plane
  - Note: See 2.21 and 2.31 for full moving rules of relevant pieces
- 2.51b - It may take a piece of the opposite colour by finishing its move on that piece's square
- 2.51c - It may not move over pieces of any colour
- 2.51d - It may not end its move on a piece of the same colour
- 2.51e - It may not move over the edges of the board, moves that end off the board are not valid

The other pieces function more in line with each other and thus were much easier to implement, these requirements have been fulfilled as proven in [tests 11, 12, 14, 13](#) (in order of requirement blocks).

2.6 - At the start of the game, set the board out as follows(Taking origin as (0,0,0):

2.6a - White: Starting in the bottom left, closest corner (0,0,0). Place a white rook. To its right place a Knight (1,0,0), Bishop(2,0,0), Queen(3,0,0), King(4,0,0), Bishop(5,0,0), Knight(6,0,0), Rook(7,0,0)  
For the rows (x, 1, 0), (x, 1, 1): every square should be filled by a pawn.

2.6b - Black: Starting in the top left, farthest corner (0,7,7). Place a black rook. To its right place a Knight(1,7,7), Bishop(2,7,7), Queen(3,7,7), King(4,7,7), Bishop(5,7,7), Knight(6,7,7), Rook(7,7,7)  
For the rows (x,7,6) and (x,6,7): every square should be filled by a black pawn so that the non-pawn pieces are completely encapsulated by pawns.

Although this requirement went through several variations in the research phase, it was quite simple to fulfil with the interfaces I implemented: [test 4](#).

### Game Rules

3.0 - White always moves first

3.1 - As per 2.02, check must always be resolved

3.2 - If check cannot be resolved, the player who cannot move their King and it is under threat has lost and the other player has won

3.21 - This is also the end of the game and no more moves can be made after it

3.4 - When the game ends it should automatically be saved to the database

As per above mention of [test 23](#), this has been fulfilled.

3.3 - If a player cannot move on their turn and their King is not in check, it is a stalemate, a.k.a. a draw. This is the end of the game, and no more moves can be made after it.

This alternative outcome to the game has been fully implemented, rendering the game fully complete, as shown in [test 24](#).

### Game and Player Storage

4.0 - A game can be saved into a database

4.01 - When a game is saved it should include the move history

4.02 - Saved games can be loaded from the Load Game menu (See 7.12)

4.02a - The pieces positions are loaded and applied to the board

4.02b - The move history is loaded into the Move History List in the UI

4.03 - Any extra optional rules should be stored and subsequently loaded when the game is loaded (See [6: Optional Rules](#) for optional rules)

4.04 - Which players took part in the game and which colour each player was

4.05 - The outcome of the game, this will be "White Win", "Black Win", "Stalemate" or "Ongoing"

4.06 - The date the game begins should also be recorded

Demonstration of the game database can mostly be seen in [tests 31, 32, 33 and 35](#).

4.1 - Players can be stored in a database

4.11 - The number of games won, lost and drawn by each player should be recorded

4.11a - The total number of games played should be recorded

4.11b - Record the number of games won as white and the number of games won as black

4.11c - Record the number of games played as each colour

4.12 - The name of the player should be recorded

4.13 - The date the player is registered should be recorded

These requirements were slightly harder to design tests for, the best way to show the state of the player database is in the table shown in the Players menu, throughout demonstration this is accessed many times but specifically in [test 2 \(player creation\)](#) and [test 6 \(player deletion\)](#).

4.2 - At the end of each turn the current state of the game should be saved to the database

This requirement is fulfilled, as shown in [test 31](#), and its boundary conditions are covered in [tests 32-35](#).

### Players

5.0 - Chess has two players, one controls the white pieces and the other controls the black pieces

5.1 - The game must keep track of who's turn it is

- 5.11 - Players can only move one piece on their turn and then it is the other player's turn
- 5.12 - The UI should display who's turn it is
- 5.2 - When the game is started each player may select a player profile from the list, fetched from 4.1
- 5.21 - The player should have the option to make a new player profile

Test 1 (game creation) demonstrate the player selection; the relevant tests for the player list are shown above. Throughout the video we can see the player's names used in messages. In regard to only being able to move on your turn, this requirement is fulfilled and demonstrated in test 19.

### Optional Rules

6.0 - Before starting a new game, the players will be asked whether they want to use any optional rules

A simple requirement to implement but required implementation on multiple levels: database, interface and the main game class. Demonstrated in test 7.

- 6.1 - Optional Rule 1: Players can rewind the moves of the game
  - 6.11 - Next to the list of moves in the UI (See 1.7) there should be a rewind button
  - 6.12 - This button should undo the last move, returning the board to the state it was in before the last move was made
  - 6.13 - Any moves made after having rewinded should overwrite the move list and delete any entries later than it, to avoid creating problems when exporting/importing games

These requirements have been fulfilled and demonstrated, including boundary cases, in tests 27, 28, 29, 30, 32, 33 and 35.

### Menus

7.0 - Upon starting the program the user should be faced with a main menu, composed of large, simple buttons

- 7.01 - The first option on this menu should be "Play" - This will take the user to another menu, detailed in 7.1
- 7.02 - The second option on this menu should be "Players" - This will take the user to another menu, detailed in 7.2
- 7.03 - The third and final option on this menu should be "Exit" - This will close the program
- 7.1 - The "Play" menu should have two large, simple buttons, those being "New Game" and "Load Game" which will take the user to their respective menus (detailed in 7.11 and 7.12)
  - 7.11 - The "New Game" menu will have multiple buttons (Please see the UI Mockup for arrangement and size)
    - 7.11a - On the left, there should be a dropdown list of established player profile for the White player to select from
    - 7.11b - Next to the dropdown menu, have a button to create a new player profile which opens a popup to go through it
    - 7.11c - Have the same options mirrored for black on the other side of the UI
    - 7.11e - Have a tickbox for whether to use Rewinds (Optional Rule 1; See 6.1)
    - 7.11f - Have an entry box at the top to name the match, otherwise generate a random string
    - 7.11g - Have a large, simple "Start" button at the centre base of the UI, which starts the game
  - 7.12 - The "Load Game" menu will have multiple buttons (Please see the UI Mockup for arrangement and size)

7.12a - This menu should be mainly composed of a large list of games stored in the database (see 4.0) displayed as a table

7.12b - This list should be able to be sorted using navigation panels at the top of the menu element, this criteria should be sort by "Name", "Date", "Outcome", "White Player Name" and "Black Player Name"

7.12d - Upon selecting a game to open, the user should be taken to intermediary screen where it shows the game settings and current move. The user can then confirm this is the game they intend to load and upon clicking on a large, simple "Start" button the game is loaded

7.12e - If a completed game is loaded, automatically enable rewind moves and change the name so that if it's saved again, it isn't saved back onto the completed game but onto a new position in the database

7.2 - The "Players" menu should show a list of all players displayed as a table

7.21 - Using a navigation panel at the top of the table, the list can be sorted according to "Name", "Date Registered", "Overall Wins", "Winrate as Black", "Winrate as White"

7.22 - There should be a button that takes the user to a new player creation screen, labelled accordingly

These requirements are all demonstrated under [tests 1-7](#). The menus are also used throughout the demonstration video as being a core part of the project. I'm happy with how quick and detailed they are since they are a core part of using the project.

7.3 - Whilst in a game, there should be a menu icon on the bottom left corner of the UI, clicking this element will pause the game. Making other elements uninteractable and displaying a menu with large simple buttons.

7.31 - The first button should be a "Resume" button which returns the game to its unpaused state

7.32 - There should be another button, labelled "Quit to menu", which saves the game and returns the user to the Main Menu

7.33 - There should be another button, labelled "Quit to desktop", which saves the game and closes the program

7.34 - There should be two buttons, one labelled "White Forfeit" and one labelled "Black Forfeit"

7.34a - Clicking on this button should take the user to a screen where they confirm their decision

7.34b - Once their decision has been confirmed, end the game and save the outcome depending on which player forfeited

7.35 - There should be another button, labelled "Agree to draw"

7.35a - Clicking on this button will take the user to a screen where they confirm their decisions

7.35b - After confirming their decision, end the game as a draw and save the outcome

[Test 25](#) demonstrates the menu functionality and [test 26](#) demonstrates the agreed ending functionality.

### End User Assessment

In order to assess the project, I have interviewed Ciaran Brightley-Davies again, since the group he represents (chess players) are target end users.

Q: “Do you think the program is sufficiently easy to use, as you mentioned you were concerned in earlier interviews?”

A: “For the most part, yes. The menus make sense and are easy to navigate quickly. Moving pieces around and moving up and down layers of the board is easy enough once you get some practice with it. It can be a bit hard to figure out where you are in the cube though, the coordinate line underneath the board is useful but I also didn’t really understand it at first. I think the main issue is when you lose track of what direction you’re looking at the board from, sure you can infer that from the coordinate line, but it would probably be more helpful to highlight the direction button that you last selected.”

Q: “Are you satisfied with the level of personal competition?”

A: “I think it’s just right for a chess alternative such as this. It lends itself to a more casual audience so not having leaderboards and rankings be the front and centre makes sense, but it’s neatly tucked away in the players menu. I really like that you can sort separately for your win-rate as black or white. Of course, online chess has many more stats, I think you limited yourself with your table size there. Maybe a more in-depth analytics page for each player could be nice, if you were to add to it.”

### Potential Improvements

A collection of ideas and improvements that could be made to the program given time and/or a revisit to the problem. These are inspired by the feedback gathered and self-evaluation.

#### Improvement 1:

Rework the 2D representation slightly:

- Highlight the current view direction
- Use the coordinate line to show the coordinates of a piece
- Annotate the ranks and columns to improve understanding

An issue with allowing users to play the project was always losing themselves spatially. Having spent half a long time working on the project I had gotten used to the board in ways that I wasn’t aware a first-time user would miss. Therefore, more adjustments need to be made, that I overlooked, to facilitate ease of use – some of these were highlighted by Ciaran Brightley-Davies (end-user interviewee) and others arose in demonstration/testing sessions.

#### Improvement 2:

Because full games proved to be exceedingly long, it would be a good idea to have an alternative mode. Something akin to a puzzle-builder, where a user can place pieces however they like for whatever purpose. They could create interesting positions, puzzles, art or simply play around with the limits of the program. This would provide some extra functionality.

The main difficulty with this would be creating new UI and input elements to handle all this, as well as adding extra attributes to the database and extra constructors for the board. Given the time, it is an avenue I would’ve liked to have explored but was too great an investment for a small feature for me to include in the project.

#### Improvement 3:

Another improvement, this one mostly reliant on time to complete the solution, is alternate players. By this I mean:



- Playing online against other people
- Playing against an AI opponent

The first one is outside the scope of my skills unfortunately, so would require lots more time and research. It may also require centralised hardware, such as a server, to run games and store game data. This would be nice for driving a competitive environment too.

The second is also outside the scope of my abilities but it's something I would be interested in implementing in the future. I think it would be very interesting to see how a computer plays this game though I'd be wary of creating an AI too powerful to the point that its unfun. So, this idea brings several challenges about difficulty and implementation that would've have made the project too bloated in terms of skills and resources.

### Project Summary

In conclusion, I'm very happy with the state of the project, especially as there were certain stages where the challenges seemed nigh insurmountable. Despite this and the lengthy development time, all the requirements have been met and testing went well.

Having spent a good amount of time on the research, I had a good idea of what it should look like and most of the rules – some had to be worked out later in design (such as starting positions). A lot of the requirements were informed by research and feedback from the potential end-user base; particularly, the importance of ease of use was made greatly apparent. With clear goals in mind and a set of robust requirements the design phase was also easy.

Most of the design require structuring my Object Oriented model to handle events and requirements correctly. I dug into this fully with my class and sequence diagrams which showed that the chain of events could be very neat and allowed me to avoid messier solutions such as involving timing. In designing my model to be responsive rather than asynchronous it also simplified the later implementation of the user interface.

Before working on the implementation of my solution, I was convinced that the user interface was going to be the most difficult as it required many moving parts to represent and allow interaction with the game. However, thanks to the design of my model this was not the case. Other features proved more challenging, such as calculating checkmate efficiently or not allowing self-check moves. With the application of high-level concepts, such as recursion, these challenges were also resolved.

Finally, despite some possible improvements and concerns raised in feedback I have fulfilled the solution to the problem I set out at the start: a 3D chess client which is accessible by casual and competitive players alike.