**Simple Chess**

Prepared by: Petros Vasileios Antoniou

Supervised by: Prof. E. Vagianou

ITC 4918 – Software Development Capstone Project

Spring Semester 2020

Table of Contents

[1. Introduction 2](#_Toc40203825)

[1.1 Aims and Objectives 2](#_Toc40203826)

[1.2 Background Research 2](#_Toc40203827)

[1.3 Development Resources 4](#_Toc40203828)

[2. System Analysis 5](#_Toc40203829)

[2.1 Product Perspective 5](#_Toc40203830)

[2.2 Product Features 5](#_Toc40203831)

[2.3 User Classes and Characteristics 5](#_Toc40203832)

[2.4 Operating Environment: 6](#_Toc40203833)

[2.5 Design and Implementation Constrains 7](#_Toc40203834)

[2.6 Assumptions and Dependencies 7](#_Toc40203835)

[2.7 Interface Requirements 7](#_Toc40203836)

[3 System Design 9](#_Toc40203837)

[3.1 System Features 10](#_Toc40203838)

[3.2 Use Cases 21](#_Toc40203839)

[3.3 Interactivity and Associations 43](#_Toc40203840)

[4 System Documentation 48](#_Toc40203841)

[4.1 Technical Documentation 48](#_Toc40203842)

[4.2 User Documentation 48](#_Toc40203843)

[5 Conclusions 50](#_Toc40203844)

[References 51](#_Toc40203845)

[Appendices 53](#_Toc40203846)

[A. Challenges 53](#_Toc40203847)

[B. Screenshots 53](#_Toc40203848)

[C. Testing 59](#_Toc40203849)

[D. Supporting content 59](#_Toc40203850)

[E. Deployment Instructions 60](#_Toc40203851)

# Introduction

This project is a game in which the users can play chess against an AI of different difficulties. More specifically, there is a chess component that stores the data of all pieces on the board and, when the pieces move, it registers the piece that moves and whether the piece ate an opponent’s piece. Then, the moves of all pieces are calculated, and if a side’s king is in check, the moves of that side’s pieces are limited. If a move results in a check mate, the side that succeeded in the check mate is the victor and the game ends. The AI moves according to decision making algorithms, and more specifically Minimax and some of its variations. For the first few moves, the moves of the AI will be based on standard opening moves. The game will be built using Unity mainly for the graphics when showing the board, pieces, the movement of the pieces and for showing user interfaces, while the chess component will calculate the moves and then move the pieces.

During the last few years, games have become an integral part of our lives. Computer games help people pass their time and provide a hobby, mobile games give people something to do when traveling or when they do not have access to a computer, like in an outside area, and there are people who play competitively to become the best or to make a living, like with the case of e-sports. In the case of chess, there are many people that do not know how to play. Even though there are websites that show the movements of each piece, some people prefer to play a simple, easy game and learn that way.

This chess game is a way to help both those that do not know how to play as well as the people that want to challenge themselves and improve. For the ones that want to learn how to play, there is a tutorial showing how each piece can move, how the player wins through check mate and special moves like castling and the promotion of a pawn. For those that know how to play already, the game provides different difficulties allowing the users to challenge themselves and get better at the game.

## 1.1 Aims and Objectives

The purpose of Simple Chess is to provide a user friendly and easy to use environment where users can play games of chess. People that do not know how to play chess will have the opportunity to learn and experiment with the game through tutorials and a practice mode. People that already know how to play can play through various difficulties to become better at the game.

## 1.2 Background Research

The background research started by searching on several forums and on Google. The first two pieces of information were a thread on Stack Overflow [1] and the site based on Tom Kerrigan’s Simple Chess Program (TSCP) [2].

On Stack Overflow, a question was posted concerning the creation of a chess engine. As the person that posted the question pointed out, they were looking for answers for creating a simple chess engine. The first and best answer to that question lists several resources that can be used for reference. The first resource listed is the Chess Programming Wiki [3]. This site has many different parts related to writing a chess engine. Within these parts, the Basic part will be very useful. More specifically, the Board Representation is a page that contains ways to represent and use a chess board. Move Generation is also an important part of a chess engine and the Wiki has a page on that topic. Also, the Wiki has a page centered on how to search for the next move, a page on evaluating the states and an opening book for the first moves of the game.

Another resource that is listed is a simple introduction on GameDev [4]. This page does not have as much information as the previously mentioned Wiki. It has come very basic information on board representation, move generation, search techniques and board evaluation.

The answer lists other resources as well, including an analysis of MTD(f), a search algorithm, a web page on validation of move generation and others.

TSCP Chess is another good starting point. It is a site created bases on Tom Kerrigan’s Simple Chess Program. This site contains a more in-depth analysis on the different components of a chess engine. The first component is the move generation. This page explains some difficulties on move generation and explains how the move generation can be done. Another page explains how making a move can be done. It explains that making a move involves a from and to position, storing parameters and checking for rules. It contains a page on board evaluation and how it can be used with examples and pages on searching for the best move to make, the importance of depth and alpha beta pruning for optimization.

The next step of the research was the design of a chess engine. Geeks for Geeks [5] contains an article on a basic first look on the design of an engine with different components. The first component that the article lists is a spot. It is a block of the 8x8 grid of the chess board. Another component is the piece. It is the basic component of a chess engine. It holds information like the type of piece it is (Rook, Knight, etc.). The board is another component. It is a combination of 64 Spots and all active Pieces. Other components include the Player, the Move and the Game.

An answer on Stack Exchange [6] offers insight on the logic of the chess engine. It explains the basic flow of the game. According to that answer, the first step is a player choosing a piece to move, which then checks its valid moves. Aside of the movement, it is essential to check for possible captures. If that player was in check and the move does not remove it, it must be undone. If the move exposes a check, it must also be undone. If the piece that moved captured a piece, that piece must be removed. If the piece moving is a pawn and it reaches the final rank, it is promoted. If the move was a castling, both the king and the rook must move. Lastly, if that move resulted in a stalemate or check mate, the game is over. This logic is valid and a great starting point.

An article [7] on Peter Ellis Jones shows steps toward efficient legal moves generation. The article describes the difficulties when checking for valid moves. Such difficulties include checking for checks, checking that a move does not result, etc.

The algorithm is also very important and the research behind it is split into two parts. The first part of the research focuses on what algorithm is optimal for a chess engine. On Quora [8] and Stack Overflow [9] there are threads where people asked what the best algorithm for chess is. According to the answers, the best algorithm is minimax, a brute force algorithm. Also, Rune Djurhuus, a chess grand master argues that minimax is the basic algorithm to search for moves in his Chess Algorithms paper [10]. He explains that it is optimal and that using alpha beta pruning the algorithm can search deeply and with great efficiency. Another thread on Stack Overflow [11] is a question of what happens if the minimizer plays sub-optimally. The minimizer is the player, where the maximizer is the AI opponent in the minimax algorithm. On that thread, the first answer explains how the evaluation is processed. According to that answer, the algorithm checks all possible moves and gives the move where the maximizer will gain the most and lose the least. However, the move that is returned assumes that the minimizer plays optimally. So, according to the answer “Thus, by definition, if you don't play optimal, your opponent has at least one path that will give him a higher score than his best score if you played optimal”. Finally, Free Code Camp [12] has an article that explains how a simple chess AI is created step-by-step. It follows four steps: move generation, board evaluation, minimax and alpha beta pruning. The first step shows how moves are generated and how the board can be represented and visualized, the second step shows how points are distributed to pieces and how the board is evaluated, the third step introduces minimax as a search algorithm and the fourth and final step explains how alpha beta pruning makes the minimax algorithm more efficient.

The second part of the research focuses on the openings. The search focused on openings databases. Some of these included Chess Base and Chess Tempo. However, Chess Tempo [13] gives the ability to make moves and see what openings can be used. So, Chess Tempo is used to look for the various opening moves. Also, using Chess Tempo it is possible to see the most frequent moves given the state of the chessboard, as well as the percentages showing the win rate of the white and black side that have played the moves.

## 1.3 Development Resources

Unity is the main tool used for the development of the application. Using Unity, it is very easy to create a user interface. The image of the pieces, the valid moves, the movement of each piece and the general UI can all be easily created. Unity API also has many advantages. Coroutines are very flexible and are used for the piece movement and the move of the AI and the Unity Scene system is used to create different levels and use game objects that interact with each other.

C# is the programming language used for development. C# is the language supported by Unity and has its own individual features like using threads and structs.

Due to Unity’s JSON Utility not being able to deserialize JSON files with a depth more than 7 (see Appendix A), Newtonsoft Json is used. It is a popular framework for working with JSON files and is available in the Unity Asset Store so there is no need to use any third-party sources to download it.

# System Analysis

## 2.1 Product Perspective

The project is a standalone application that consists of two parts:

* Chess Board: The chess board is a class that stores all information regarding the current game. Some of the information that is included is pieces positions, castling information and whether a king is in check. When a piece moves, the valid moves for both sides are calculated and if a side has no more valid moves, it is game over. Also, this class allows a pawn to be promoted and returns the chess board’s current state.
* The graphics: The graphics mainly includes the objects that the user interacts with. These will be buttons with an image on top representing the specific piece. When the user presses on a button, the valid moves of a pawn are created on top of the tiles where that piece can move.

## 2.2 Product Features

* Movement of Chess Pieces
* Valid Move Generation
* Tutorial Scene
* Practice Scene
* Choice of Difficulty
* Board Representation as 2D Pieces
* Valid Move Representation as Green Tiles
* Game Conclusion (Win/Loss/Draw)
* Save and Load Game

## 2.3 User Classes and Characteristics

Even though there is one user class, the chess players, it can be divided into two user modes, the users that do not know the rules of chess and use the application to learn and the users that already know and use the application to become better.

New Chess Players:

Needs:

* A tutorial that explains the rules of chess
* A practice mode where the user moves both the white and black pieces and can observe the valid moves of these pieces.

Social Characteristics:

* Age: 12 – 100 years old
* Gender: Male or Female
* Knowledge Background: None
* Skillset: Basic knowledge of how to use the internet to download the application
* Profession: None

Players that already know how to play chess:

Needs:

* A practice mode where the user can move both the white and black pieces to test different strategies
* A game mode versus an AI with different difficulties.

Social Characteristics:

* 15 – 100 years old
* Gender: Male or Female
* Knowledge Background: Rules of chess
* Skillset: Basic knowledge of how to use the internet to download the application
* Profession: None

## 2.4 Operating Environment:

Operating System: Windows

Platform Used: Unity (with C# as the preferred language)

External Data File Format: JSON

## 2.5 Design and Implementation Constrains

The project does not have any constrains, since Unity offers the possibility to build the project and create an executable file. However, the Openings, Save and Tutorials JSON files are in a specific folder so that they can easily be found. The game will not work if they are moved.

## 2.6 Assumptions and Dependencies

As this is a local version of the game, the only dependency is that the game is run in Windows since the built version is compatible with Windows only.

## 2.7 Interface Requirements

User Interface

Menus

* Main Menu: This interface is shown at the start of the game and allows the user to select how to proceed. There are 5 options: Play the tutorial, view the game instructions, play in practice mode, play against the AI and exit the game.
* Tutorial: In this game mode the user can see the rules of chess and press the Next or Previous button to navigate through the tutorials. There is also the Back button that allows the user to return to the Main Menu.
* Instructions: This interface allows the user to see how the application works. It consists of images and an accompanying text that explains the image. The user can press on the Next or Previous buttons to navigate through the different tutorials r the Back button to return to the Main Menu.
* Practice: In this game mode, there is the Undo and the Back button. Pressing Undo will undo the last move and Back will return to the Main Menu.
* Against AI: Pressing this button will open the Difficulties Menu.
* Difficulties: Here the user can choose the difficulty of the AI or press back to go back to the Main Menu.
* Exit: Pressing this button will close the application.

Chess Board:

* Pieces: The pieces on the board are buttons with an image on them. The image changes according to the type of the piece (Queen, Rook, etc.). The user can press on a piece and that piece’s valid moves will appear.
* Valid Moves: After pressing on one of the piece’s valid moves, the piece will move to that specific tile.
* Promotion: When a pawn reaches the last row, it can be promoted. When that happens, a small panel will appear on the right with the 4 types the pawn can promote to. Then, the user can press on one of the four buttons on that panel to promote the piece.

Hardware Interface

* Windows: The application will be released for the Windows OS.
* Standard Peripherals

# System Design

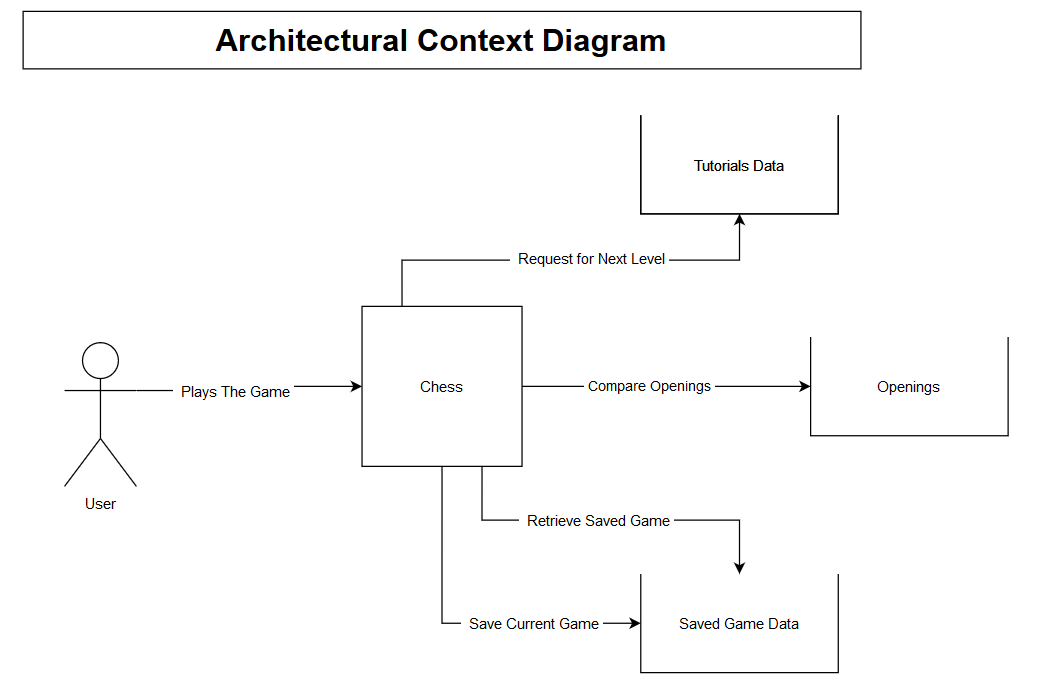
The design of the application is based on Pieces. They are structures that contain information like type (rook, queen, etc.), color, coordinates and an id. By playing the game, the position of the pieces is changed. These pieces are used in 2 different ways:

* The Board 2D Array: This is an 8 by 8 array that holds 64 pieces. It represents the chess board. The pieces in this array represent an active piece with a type and color, or an empty space with no color and an asterisk (“\*”) for the type. The pieces on this array are used mainly as a reference when calculating the valid moves of the other pieces.
* The Pieces Array: This array holds the crucial information of a piece like its valid moves, its coordinates on the board and the points the specific piece is worth.

When calculating the moves of a piece, the Board array is used to check where it can move. Then, its moves are stored on that piece in the Pieces array. All active pieces on the Board array have an id that links them to the corresponding piece on the Pieces array. If a piece is eaten, it is removed from the Board array and its id becomes -1 on the Pieces array. When the board is saved, it is compressed to a byte array with its necessary information. Then, a board object can be created from that array.

When creating the interactable graphics board, a new array is created called uPieces. This array contains Unity Pieces, which are buttons with an image on them. That image changes according to the type of piece the uPiece represents. The elements on the uPieces array are mapped one to one to those of the Pieces array so that their coordinates are used to represent them. So, the Pieces array of the Board class acts as the link between Board and the Graphics parts. Finally, when the used presses on a uPiece that can move, green tiles appear on top of the background tiles. These green tiles show where that piece can move and use the list of valid moves to create coordinates.

Below is the Architectural Context Diagram (ACD):



## System Features

The Board Class

The Board class holds the information regarding the chess board like information on the pieces, the checks and the castling conditions. It has 3constructors to create a starting board, a cope of a pre-existing board or a copy from a byte array.

Constructors

Board

This is the simple constructor that creates a new chess board with the pieces in their starting positions.

Input: None

Output: Chess board, board object

Board (Board object)

This is the constructor that takes as parameter a Board object and copies it.

Input: Chess board, board object

Output: Chess board, board object

Board (byte array)

This is the constructor that takes as parameter a byte array and creates a Board object using the information on that array.

Input: Chess board, byte array

Output: Chess board, board object

Move

This process is called with four integers as parameters. These integers are the from and to coordinates. Then the piece on the from coordinates moves to the to coordinates. If there is a piece already there, it is eaten.

Input: Coordinates, integers

Output: None

Move Generation

This process is called to calculate all valid moves that the pieces can make. Through several other processes, it assigns a list of valid moves to every active piece. These processes are:

Moves of Pieces

This process is called to calculate and return a list containing the valid moves a piece can make and takes as arguments that piece’s coordinates.

Input: Coordinates, pair of integers

Output: Valid moves, list of integers

Can Move

This process is called for all active pieces. It takes as parameters the coordinates of a piece and calculates if that piece can move and where. If the piece is between the king and a slider for example, it either returns no valid moves or a list of limited moves.

Input: Coordinates, integers

Output: Valid moves, list of integers

Number of Checkers

This process takes as a parameter the id of a king and calculates and returns how many pieces are currently checking that king.

Input: King id, integer

Output: Number of checkers, integer

Check

This process is called to re-calculate the valid moves of all pieces of a color. If a king is in check, all active pieces of that king’s color will have their valid moves limited. If the piece checking the king is a slider, the pieces can move either between the checker and the king or to eat the checker. If the checker is not a slider, the pieces can move only to eat the checker.

Input: King id, integer

Output: None

Moves of King

This process is called to calculate the moves of a king. It takes as a parameter the id of the king and returns a list of that king’s valid moves.

Input: King id, integer

Output: Valid moves, list of integers

Check if Tile

This process is called when calculating the valid moves of a king. It takes as parameters the id of the king and two coordinates and returns true if the king can move to the coordinates specified or false if it cannot.

Input: King id, coordinates, integers

Output: Can move to tile, boolean

Final State

This process is called to compress the current chess board to an array of bytes. The information regards the position of the pieces and information on castling and promotion. Then, it returns that byte array.

Input: None

Output: Chess board, byte array

Promote

This process is called when a piece reaches its last row. It takes as parameters the piece’s id and the type and points that it will be worth. Then, that pawn’s type and points change to that of the parameters. This process is only called from outside of the class by the Graphics.

Input: Pawn id, piece type, point, integers

Output: None

The Graphics

The Graphics is the part that creates the UI and handles the user’s actions. It contains the features that are used when the user interacts with the board and in the Versus AI mode, the Graphics part calls the algorithm that calculates the AI’s next move. The Graphics are also used to read the Openings JSON and return a move if the user plays according to an opening strategy, to read and write to the Save JSON and to create the tutorial boards from the Tutorials JSON.

Mono

Mono is a C# script that uses objects from the Board class and creates, manipulates and destroys uPieces. It is the main script in the Versus AI Mode.

Update Graphics

This process is called when a white or black piece moves. It moves the uPiece corresponding to the Board piece that moved and calculates possible castlings, pawn promotions and victory, defeat or possible draw.

Input: Coordinates, integers

Output: None

Move

This process is called when the player presses on a valid move to move a uPiece. First, Update Graphics is called and then it checks for possible opening moves from the current state. If none are correct, it proceeds to start calculating the AI’s next move.

Input: Coordinates, integers

Output: None

Undo

This process is called when the user presses the Undo button. It changes the current chess board to the state it was before the last two moves. If there have been less than two moves before pressing the Undo button, the game is reset to is original state.

Input: None

Output: Chess board, byte array

Easy Mode / Hard Mode

These processes start after the user has made their move and there are no possible openings. Then, the algorithm to calculate the AI’s next move starts. When it is calculated, the array returned is passed to Update Graphics.

Input: None

Output: Chess board, Board object

Output: Max Depth, integer

Output: Max Player, boolean

Output: Alpha and Beta values, integers

Output: Coordinates, integers

Openings Move

This process is called after the user has made their move and there is at least a valid opening move. Then, this process takes that move and passes it to Update Graphics

Input: Coordinates, integers

Output: Coordinates, integers

Get Moves

This process is called to search for and return a list of the valid moves of a piece. I takes as parameters the coordinates of the board and searches based on those

Input: Coordinates, integers

Output: Valid moves, list of integers

Get ID

This process is called to search for and return the id of the Board piece specified by the coordinates passed as parameters

Input: Coordinates, integers

Output: Id, integer

Save

This process saves the current save of the chess board to the Save JSON file.

Input: None

Output: Chess board, byte array

Load

This process takes the saved state of the game and creates a new Board object based on that.

Input: Chess board, byte array

Output: None

Lost

This process is called when one side loses. It takes as a parameter a boolean which stands for which side lost.

Input: Side lost, boolean

Output: None

Promote

This process promotes a pawn. It is called when the user chooses the piece to promote the pawn to. The parameter is an integer array with two elements. The first is the pawn’s new piece type and the second the pawn’s new points that it is worth.

Input: New piece info, integer array

Output: None

Draw

This process is called when the game ends in a draw. It takes as a parameter a string which is the explanation of the draw.

Input: Draw reason, string

Output: None

Threefold Repetition

This process is called by Update Graphics every time a move is made. It is used to compare all previous states of the game and the current one. If there are 3 states that are identical, the game ends in a draw because of threefold repetition.

Input: Move history, list of byte arrays

Output: Match, bool

Tutorial

This is a C# script that manages the tutorial stages. It has multiple similar processes to Mono, so these similarities will be listed below:

* Start
* Update Graphics
* Move
* Get Moves
* Get Id
* Promote

Tutorials Initialize

This process creates the list of tutorial stages. It reads the data from the Tutorials JSON, puts them in a list and returns it.

Input: Tutorials, JSON

Output: Tutorials, list of TutorialLevel objects

Next / Previous Tutorial

This process is used to change the chess board and the explanation text of the tutorial. It creates a Board object from a byte array from a TutorialLevel object and changes the chess board and explanation text to the previous or next one.

Input: None

Output: Chess board, Board object

PVP

This is a C# script that is used in the Practice Mode. It is very similar with the Mono script and does not have unique features.

Diff

This is a C# script used when the user chooses a difficulty. It is note destroyed when the scenes change.

Set Diff

This process is a setter and, based on the user’s choice, a difficulty is set.

Input: Difficulty, integer

Output: None

Get Diff

This process is called by Mono to get the difficulty chosen. This process returns the difficulty previously set.

Input: None

Output: Difficulty, integer

Openings

This is a C# script that reads the data from the Openings JSON file.

Next Move

This is a process that uses takes as a parameter the last move the user made and compares it to the data in the JSON file. If an opening is found, the to and from coordinates are returned in an integer array. Otherwise, an integer array is returned with only the element -1 in it.

Input: White side’s move, integer array

Output: Black side’s move, integer array

Equals Arrays

This process compares the elements of the two integer arrays passed as parameters and returns true if these elements are equal or false if they are not.

Input: Arrays to compare, integer arrays

Output: Equal, boolean

Moving Through

This is a C# script that is used to navigate through the Instructions Panels.

Change Instructions

This process is called to show the instructions panel determined by the index specified. The panels are in an array, so the index shows which element, which Instruction panel to show.

Input: Index, integer

Output: None

Unity Piece

This is a C# script that all uPieces have. It has an I and J coordinates that show where on the chess board the uPiece is.

Move

This process moves the uPiece. When called, it takes to and from coordinates as parameters and moves the uPiece to the specified position.

Input: Coordinates, integers

Output: None

On Piece

This is the C# script that controls the uPieces button action.

Start (Unity’s default method)

This process begins when the uPieces are created and is called automatically. It uses the Get Id process of the main script (Mono, PVP or Tutorial) using the coordinates of the uPiece this component is currently on.

Input: Piece id, integer

Output: None

Moves

This process is activated when the user presses on a uPiece. Using the Get Moves process of the main script, a list of all valid moves of the current uPiece is generated. Then, if there are already valid moves tiles for a uPiece, they are removed, and new valid moves tiles are created on top of the tiles the current uPiece can move to. Else, new valid moves tiles are created without removing anything. Finally, for all valid moves generated, their I and J coordinates are set.

Input: Valid moves, integers

Output: None

Green Button

This is a C# script used after the user presses on a uPiece to show its valid moves. When the valid moves are shown they have this script on them. When the user presses on a green button, the uPiece is moved to that tile.

Set I / Set J

These processes are simple setters that set the coordinates of the valid moves tiles.

Input: I or J coordinate, integer

Output: None

Move

This process is called when the user presses on a valid moves tile. It calls the Move process of the Unity Piece script on the uPiece and removes all green tiles.

Input: None

Output: Coordinates, integers

Minimax / Alpha Beta

This is the algorithm that calculates and returns the next move of the AI. It is used when the opening moves no longer cover the current state of the game.

Get Best Move

This process is used to move through all possible moves both the white and black side can make. After every move, the Evaluate is used to evaluate and return a score that indicates how good the current state is. This process is recursive and is called until the depth is reached.

Input: Chess board, board object

Input: Max Depth, integer

Input: Max Player, boolean

Input: Alpha, Beta, integers

Output: Next move, array of integers

Evaluate

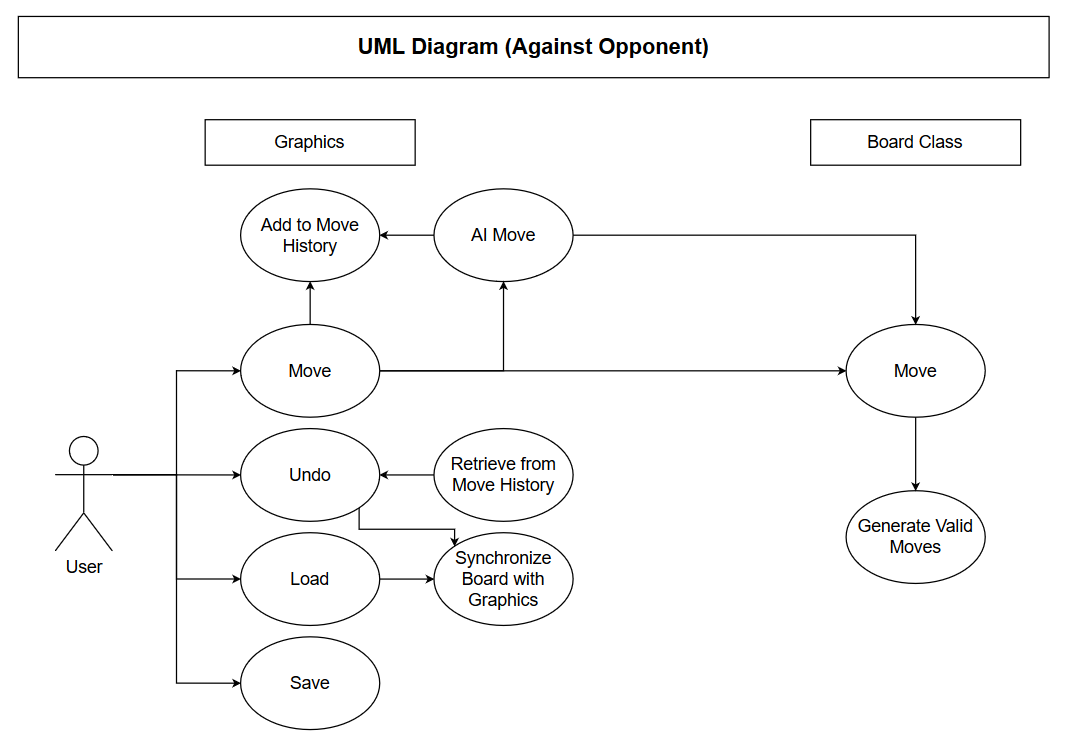
This process is called every time the Get Best Move process reaches the max depth. When this happens, this process uses the chess board passed as parameter and evaluates it, returning its score.

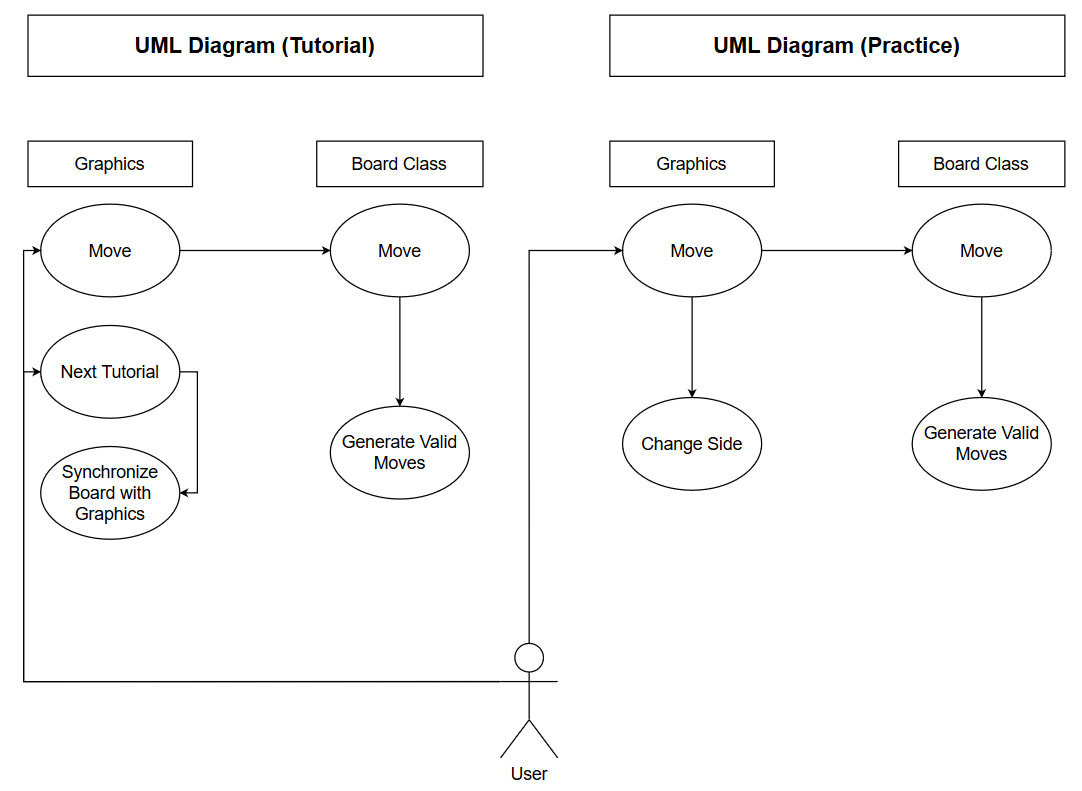
Input: Chess board, board object

Output: Score, integer

## Use Cases

The Use Cases bellow will provide a scenario of the usage of each feature as well as possible exceptions.





**Use Cases of the Board Class**

Use Case: Board

Primary Actor: The Board class

Goal in Context: Create a new Board object

Preconditions: None

Trigger: A new chess board is created

Scenario:

* User: Presses the VS AI button and chooses a difficulty
* Mono: Starts the new scene and creates a new Board object
* Board class: Creates a new Board object with each piece in its original position

Exceptions:

* Certain pieces are in the wrong positions:
  + Priority: High, must be fixed
  + When Available: First Increment
  + Frequency of Use: Small
* Pieces array and board 2d array are not correctly linked:
  + Priority: Very high, must be fixed
  + When Available: First Increment
  + Frequency of Use: Interaction between pieces array and board 2d array happens every move

Open Issues: None

Use Case: Board (Board object)

Primary Actor: The Board class

Goal in Context: Create a new Board object identical to the one passed as a parameter

Precondition: The Board object passed has no errors

Trigger: AI’s move is being calculated and the current chess board is being deep copied by the Minimax algorithm

Scenario:

* User: Plays their move
* Mono: Starts calculating the AI’s move
* Minimax: Starts the recursion to calculate the best next move
* Board class: Current chess board is being deep copied in every iteration

Exceptions: None

Open Issues: None

Use Case: Board (byte array)

Primary Actor: Board class

Goal in Context: Create a Board object by decompressing a byte array

Preconditions: The byte array’s length is correct, the byte array holds the correct elements

Trigger: User presses the Load or Undo button, the user starts the tutorial, the user changes tutorials

Scenario:

* User: Presses Load
* Mono: Reads the byte array from the Save JSON file
* Board class: Decompresses the byte array and uses it to construct a new Board object

Exceptions:

* Byte array holds either more or less elements than needed to construct a Board object:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: High, many times in one chess game
* Elements of byte array are incorrect:
  + Priority: Medium, will be fixed
  + When available: Second increment
  + Frequency of Use: High, many times in one game

Open Issues: None

Use Case: Board class: Move

Primary Actor: User

Goal in Context: Change the Coordinates of a piece in the board 2d array and pass the change to the pieces array

Precondition: From coordinates point to an active piece

Trigger: User moves a piece by pressing on a valid move tile

Scenario:

* User: Presses on valid move tile
* Main script: Calls Move with from and to coordinates
* The Board class: Coordinates are changed

Exceptions:

* Coordinates out of bounds (greater than 8 or less than 0):
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: High, many times in one game
* Coordinates do not point to an active piece in the Board 2d array:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: High, many times in one game
* Castling cannot be done when it should / can be done when it should not:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: High, many times in one game
* New coordinates not registered correctly in Pieces array
  + Priority: Very high, must be fixed immediately
  + When available: First increment
  + Frequency of Use: High, many times in one game

Open Issues: None

Use Case: Moves of Pieces

Primary Actor: Chess Piece (except king)

Goal in Context: Assign a list of valid moves to a piece depending on its piece type and position

Precondition: The piece is active

Trigger: The user plays a new game or makes a move, or the AI makes a move

Scenario:

* User: Starts new game VS AI
* Mono: Creates Board object
* Board class: Creates new chess board
* Board Pieces: Have list of moves assigned

Exceptions:

* Piece has different type than expected (e.g. rook instead of bishop):
  + Priority: Hing, must be fixed
  + When available: First increment
  + Frequency of Use: High, many times in one game
* Valid moves are different than expected:
  + Priority: Medium, will be fixed
  + When available: Second increment
  + Frequency of Use: High, many times in one game

Open Issues: None

Use Case: Can Move

Primary Actor: Board piece

Goal in Context: To calculate if by moving, the piece could leave the king in check

Preconditions: The king is not already in check

Trigger: The user plays a new game or makes a move, or the AI makes a move

Scenario:

* User: Makes a move
* Board class: Generates moves of pieces
* Board Piece: If it moves, the king will be in check. So, a list with no valid moves is returned

Exceptions: None

Open Issues: None

Use Case: Number of Checkers

Primary Actor: Board King

Goal in Context: To calculate how many enemy pieces have the king checked (max possible is 2)

Preconditions: The king is already in check by at least one enemy piece

Trigger: A piece puts the enemy king in check

Scenario:

* User: Plays VS AI
* User: Checks enemy king with rook
* Board: Calculates that there is only one checker

Exceptions: None

Open Issues: None

Use Case: Check

Primary Actor: Board King

Goal in Context: Calculate moves of pieces when a king is in check

Preconditions: The king is in check, there is at least one active piece except the king

Trigger: King is checked

Scenario:

* User: Plays VS AI
* User: Checks enemy king with rook
* Board: Calculates that there is only one checker
* Board Pieces: Valid moves assigned so that each piece can move if it can either block the rook or eat it

Exceptions: None

Open Issues: None

Use Case: Moves of King

Primary Actor: Board King

Goal in Context: Calculate the moves the king can make

Preconditions: None

Trigger: The user plays a new game or makes a move, or the AI makes a move

Scenario:

* User: Makes a move
* Board class: Calculates moves of pieces
* Board class: Calculates moves of king

Exceptions:

* Castling cannot be done when it should / can be done when it should not
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: High, many times in one game

Open Issues: None

Use Case: Check if Tile

Primary Actor: Board King

Goal in Context: See if the king can move to a specific tile

Preconditions: None

Scenario:

* User: Makes a move
* Board class: Calculates moves of pieces
* Board class: Calculates moves of king
* Board King: For every possible move, check if it is possible

Exceptions:

* Move of a king leads to king being checked:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: Very high, many times in one game

Open Issues: None

Use Case: Final State

Primary Actor: Board class

Goal in Context: Compress the current chess board in a byte array

Preconditions: None

Trigger: The user makes a move and the move history is updated or the user saves the game

Scenario:

* User: Clicks the Save button
* Board class: Creates a byte array that expresses the current chess board
* Board class: Returns the byte array to the main script to save the game

Exceptions:

* Castling or position information not correctly passed to the array
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: High, many times in one chess game
* Byte array does not hold the correct number of elements:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: High, many times in one chess game

Open Issues: None

Use Case: Promotion

Primary Actor: Board Piece

Goal in Context: To change the type and points worth of a pawn after it reaches its last row

Preconditions: None

Trigger: The user moves a pawn to the last row and chooses which piece to promote the pawn to

Scenario:

* User: Moves a pawn to its last row
* User: Chooses the piece that the pawn will be promoted to from the promotion panel
* Board class: Changes the type and points of the promoted pawn

Exceptions:

* The array passed has wrong type or points
  + Priority: Medium, will be fixed
  + When available: Second or last increment
  + Frequency of Use: Low, a few times in a game

Open Issues: None

**Use Cases of Mono**

Use Case: Update Graphics

Primary Actor: uPieces

Goal in Context: To move a uPiece according to the coordinates specified and check if the game has ended

Preconditions: None

Trigger: The user presses on a green tile to move a uPiece

Scenario:

* User: Presses on a uPiece to see its valid moves
* User: Presses on a green tile to move the piece to that tile
* uPiece: Changes its coordinates to the new coordinates

Exceptions:

* Coordinates are out of bounds of the chess board:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: Very high, every time the user or the AI moves
* The uPiece that is moving has been destroyed:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: Very high, every time the user or the AI moves
* The Board piece that is moving is not active:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: Very high, every time the user or the AI moves

Open Issues: None

Use Case: Move

Primary Actor: User

Goal in Context: To get the to and from coordinates of the uPiece’s movement

Preconditions: The green tiles are working properly

Trigger: The user presses on a green tile

Scenario:

* User: Presses on a uPiece to see its valid moves
* User: Presses on a green tile to move the piece to that tile
* User: Presse on a green tile
* uPiece: Returns to and from coordinates

Exceptions:

* The original coordinates of uPiece and Board 2d array do not match:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: Very high, every time the user or the AI moves
* The coordinates returned are outside the bounds of the chess board:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: Very high, every time the user or the AI moves

Open Issues: None

Use Case: Undo

Primary Actor: Board class

Goal in Context: To return the current chess board to a state before the last 2 moves were made

Preconditions: There are enough elements in the move history list to return the state to what it was 2 moves before, the AI is not calculating its next move

Trigger: The user presses the undo button

Scenario:

* User: Presses Undo
* Board class: Is returned to what it was 2 moves ago
* uPieces: Are mapped to Board class’s Pieces array to represent the new chess board

Exceptions:

* A piece that was eaten is not returned to the chess board:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: Very high, every time the user or the AI moves
* A previous promotion is still in effect, even though the pawn did not reach the last row:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: Very high, every time the user or the AI moves
* Castling can be done when it should not / cannot be done when it should:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: Very high, every time the user or the AI moves
* King that was in check is still in check, even though the move that resulted to check was undone:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: Very high, every time the user or the AI moves

Open Issues: None

Use Case: Easy Mode / Hard Mode

Primary Actor: Minimax

Goal in Context: Return next move of AI depending on the difficulty the user chose

Preconditions: Difficulty must have been set, the game has not ended (victory, defeat or draw), Minimax returns the correct move

Trigger: The user has made their move and the main script starts to calculate the AI’s next move

Scenario:

* User: Plays VS AI
* User: Makes a move
* Main Script: Calls Minimax to calculate the AI’s next move
* Minimax: Gets current chess board and return the best next move
* Main Script: Calls Update Graphics with the returned move

Exceptions: None

Open Issues: None

Use Case: Opening Moves

Primary Actor: Openings

Goal in Context: Read the Openings JSON file, compare the chess board with the possible openings and return a move if an opening is found or -1 if not

Preconditions: The Openings JSON file is found and read properly

Trigger: After the user has made their move, the main script will compare board with possible openings

Scenario:

* User: Plays VS AI
* User: Makes a move
* Main Script: Calls Openings to compare the chess board
* Openings: Returns a move or -1

Exceptions: None

Open Issues: None

Use Case: Get Moves / Get ID

Primary Actor: Board class

Goal in Context: Use the Board object in the main script to get the moves / id of a Board Piece

Preconditions: The Board object is working correctly, the coordinates passed as parameters are correct

Trigger: The user clicks on a uPiece which gets the list of moves of the Board piece the uPiece is mapped to and showing these moves as green tiles

Scenario:

* User: Clicks on a uPiece
* Main Script: Gets id of the clicked uPiece
* uPiece: Gets the valid moves of the Board piece it is mapped to

Exceptions: None

Open Issues: None

Use Case: Save

Primary Actor: Save JSON file

Goal in Context: Save the current chess board as a byte array on the Save JSON file

Preconditions: Save JSON file is found, Board class’s Final State works properly

Trigger: When the user presses the Save Button

Scenario:

* User: Presses the Save button
* Main Script: Calls Final State of the Board class to get the byte array
* Board class: Returns the byte array of the current chess board
* Main Script: Saves the byte array to the Save JSON file

Exceptions: None

Open Issues: None

Use Case: Load

Primary Actor: Main Script

Goal in Context: Get the byte array from the Save JSON file and use it to recreate the chess board

Preconditions: Save JSON File is found, Board class’s Board (byte array) constructor works properly

Trigger: When the user presses the Load Button

Scenario:

* User: Presses the Load Button
* Main Script: Gets the byte array from the Save JSON file
* Main Script: Passes the byte array to the Board class constructor
* Board class: creates a new Board object based on the byte array
* Main Script: Changes the chess board to match the new Board object

Exceptions: None

Open Issues: None

Use Case: Lost

Primary Actor: Main Script

Goal in Context: If a side has lost, end the game and state the victor

Preconditions: Board class’s variables on what side has lost are working properly

Trigger: A side, white or black, has no more valid move on any of its active pieces

Scenario:

* User: Moves a piece so that the enemy cannot make any moves, resulting in check mate
* Board class: Calculates enemy’s moves but no piece has any moves
* Main Script: Prevents the user from making any more moves and ends the game with a message saying that the user has won

Exceptions:

* The user can make a move after the game has ended:
  + Priority: Medium, will be fixed
  + When Available: Last Increment
  + Frequency of Use: Low, once a game if the user does not stop the game before it ends

Open Issues: None

Use Case: Promote

Primary Actor: Board class

Goal in Context: Change the type and points worth of a piece after a pawn reaches its last row

Preconditions: The promotion panel works properly

Trigger: The user presses on a piece on the promotion panel

Scenario:

* User: Moves a pawn to its last row
* User: Chooses a promotion in the promotion panel
* Main Script: Gets the new type and points and sends them with the pawn’s id to the Board class
* Board class: Changes the pawn’s type and points according to the parameters passed

Exceptions:

* Type passed from promotion panel is wrong:
  + Priority: High, must be fixed
  + When Available: First or second increment
  + Frequency of Use: Medium, may be done a few times in a game

Open Issues: None

Use Case: Draw

Primary Actor: Main Script

Goal in Context: To end the game if it ends in a draw and state the draw’s explanation

Preconditions: Board class’s variables on what side has lost are working properly

Trigger: A move is made that results in a draw. The reason can be stalemate, 50 move rule, threefold repetition or that only the kings remain as pieces

Scenario:

* User: Makes a move that causes a draw to take place
* Main Script: Checks and identifies the draw condition
* Main Script: Ends the game with a draw and the explanation

Exceptions:

* Threefold repetition is not recognized:
  + Priority: High, must be fixed
  + When available: First increment
  + Frequency of Use: Very low, will happen very rarely in a game

Open Issues: None

Use Case: Threefold Repetition

Primary Actor: Main Script

Goal in Context: To compare two integer arrays and return true if the elements in the arrays match or false if they do not

Preconditions: Both arrays have at least 88 elements in them.

Trigger: After a move is made, Threefold Repetition is called by Update Graphics to compare all the elements of the move history list and see if the same state has existed 3 times in total

Scenario:

* User: Makes a move
* Main Script: Moves the uPiece
* Main Script: Calls Threefold Repetition to see if the current state has existed at least 2 times before

Exceptions: None

Open Issues: None

**Use Cases of Tutorial**

Use Case: Tutorials Initialize

Primary Actor: Tutorials JSON file

Goal in Context: To create an object containing the tutorial stages

Preconditions: The tutorials JSON file is correctly structured meaning it is in a tree format where every white’s move has a sub-object containing the black’s move which has a sub-object with the next white’ move etc., the Tutorials JSON file is found

Trigger: The user presses the button to play the tutorial in the main menu

Scenario:

* User: Clicks on Tutorial
* Main Script: Reads the elements of the Tutorials JSON file
* Main Script: Creates a list of objects containing the data read

Exceptions: None

Open Issues: None

Use Case: Next / Previous Tutorial

Primary Actor: Main Script

Goal in Context: Change the chess board and explanation text

Preconditions: There is a next / previous element in the list of tutorial objects

Trigger: The user presses on the Next / Previous button to change the tutorial

Scenario:

* User: Presses the Next button
* Main Script: Gets the next element in the tutorial objects list
* Main Script: Changes the chess board and the explanation text to match those of the next tutorial object

Exceptions: None

Open Issues: None

**Use Cases of Diff**

Use Case: Set / Get Diff

Primary Actor: The Diff script

Goal in Context: To set the difficulty the user selected and use it in the VS AI mode

Preconditions: None

Trigger: The user selects a difficulty, which sets the difficulty selected, or the VS AI mode starts, and the main script gets and uses the difficulty

Scenario:

* User: Presses VS AI
* User: Selects the Easy Mode
* Diff: Set the current mode to easy
* Main Script: Gets the mode selected

Exceptions:

* Diff not found when the VS AI mode starts:
  + Priority: Very high, must be fixed
  + When available: First increment
  + Frequency of Use: Very high, every time the VS AI mode starts

Open Issues: None

**Use Cases of Openings**

Use Case: Next Move

Primary Actor: Openings

Goal in Context: Compare the user’s move with the openings to find a move that exists in an opening and return it. If one is not found, -1 is returned

Preconditions: Openings JSON file is found, integer array passed has 4 elements

Trigger: The user makes a move in the VS AI mode in high difficulty

Scenario:

* User: Chooses to play VS AI in hard difficulty
* User: Makes a move
* Main Script: Send the move as an integer array
* Openings: Compares the array passed as parameter and returns the result

Exceptions:

* The JSON file has a wrongly structured move, resulting in an error in comparing the move passed and possible openings
  + Priority: High, must be fixed
  + When Available: First increment
  + Frequency of Use: Very high, it is used every time the user moves for the first few moves in the hard difficulty

Open Issues: None

Use Case: Equal Arrays

Primary Actor: Openings

Goal in Context: To compare the elements of two integer arrays

Preconditions: Both arrays have 4 elements

Trigger: Two arrays are passed when Next Move needs to compare the move with all possible openings

Scenario:

* User: Makes a move in the VS AI mode in hard difficulty
* Main Script: Passes the move to Next Move
* Next Move: Uses Equal Arrays to compare the integer array passed with possible openings

Exceptions: None

Open Issues: None

**Use Cases of Moving Though**

Use Case: Change Instructions

Primary Actor: Instructions Panels

Goal in Context: To change between instructions panels based on the index passed as parameter

Preconditions: All instruction panels have been loaded in an array so that the index can specify the element of that array that will be shown

Trigger: The user presses on the Next or Previous button to change between instruction panels

Scenario:

* User: Presses on the Instructions button in the main menu
* User: Presses Next or Previous to change between instructions panels

Exceptions:

* Pressing on Next or Previous passes a wrong index, resulting in the wrong panel
  + Priority: Medium, will be fixed
  + When available: Last increment
  + Frequency: High, instructions panels will be used very often so that the users will know how the application works

Open Issues: None

**Use Cases of Unity Piece**

Use Case: Move

Primary Actor: uPiece

Goal in Context: To move the uPiece to a different position in the chess board

Preconditions: The uPiece is active and not destroyed, the coordinates passed as parameters are correct

Trigger: The user presses on a uPiece which shows its valid moves and then presses on a green tile

Scenario:

* User: Presses on a uPiece
* uPiece: Shows its valid moves with green tiles
* User: Presses on a green tile

Exceptions:

* The uPiece moves to a different position that expected:
  + Priority: Very high, must be fixed
  + When Available: First increment
  + Frequency of Use: Very high, it is used constantly, perhaps more than once per move

Open Issues: None

**Use Cases of On Piece**

Use Case: Moves

Primary Actor: uPiece

Goal in Context: To show what are the valid moves that the selected piece can make as green tiles

Preconditions: Show Moves script is found and used, Green Tiles as objects are working properly

Trigger: The user presses on a uPiece

Scenario:

* User: Presses on a uPiece
* uPiece: Gets list of valid moves from main script
* On Piece: Creates a green tile for every valid move

Exceptions:

* The coordinates on the green tiles are wrong, so if the user presses on the green tile a wrong move will be made:
  + Priority: Very high, must be fixed
  + When Available: First increment
  + Frequency of Use: Very high, it is used constantly, perhaps more than once per move
* The number of green tiles in not the same as the number of valid moves:
  + Priority: Very high, must be fixed
  + When Available: First increment
  + Frequency of Use: Very high, it is used constantly, perhaps more than once per move

Open Issues: None

**Use Cases of Green Button**

Use Case: Set I / Set J

Primary Actor: Green Tile

Goal in Context: To set the I and J coordinates of the green tile

Preconditions: None

Trigger: The user presses on a uPiece

Scenario:

* User: Presses on a uPiece
* uPiece: Gets list of valid moves from main script
* On Piece: Greatest a green tile for every valid move
* User: Presses on a green tile
* Green Button: Gets the I and J coordinates of the valid move

Exceptions: None

Open Issues: None

Use Case: Move

Primary Actor: uPiece

Goal in Context: to move the specified uPiece to the green tile’s position

Preconditions: The green tile has the correct I and J coordinates, the uPiece has the correct coordinates

Trigger: User presses on a green tile

Scenario:

* User: Presses on a uPiece
* uPiece: Shows valid moves as green tiles
* User: Presses on a green tile
* uPiece: Uses current coordinates and green tile’s coordinates to move

Exceptions: None

Open Issues: None

**Use Cases of Minimax**

Use Case: Get Best Move

Primary Actor: Minimax

Goal in Context: To make all possible moves of both sides and, after evaluating them, to return the best move that the AI can make

Preconditions: The Board class is deep copied, the initial Alpha and Beta values are correct

Trigger: The user makes a move in the VS AI mode

Scenario:

* User: Makes a move
* Main Script: Sends the current Board object with the other parameters to Minimax
* Minimax: Makes and evaluates all possible moves
* Minimax: Returns the best move

Exceptions:

* If in an iteration the game is over the Minimax will not stop the current path but continue:
  + Priority: Very high, must be fixed
  + When Available: First increment
  + Frequency of Use: High, used only in VS AI but used every turn
* If the game has already ended and Get Best Move is called, it will not go over any moves and return an empty array:
  + Priority: Very high, must be fixed
  + When Available: First increment
  + Frequency of Use: High, used only in VS AI but used every turn

Open Issues: None

Use Case: Evaluate

Primary Actor: Minimax

Goal in Context: To assign a score indicating the state of the chess board

Preconditions: None

Trigger: Get Best Move reaches a terminal node or a game over

Scenario:

* User: Makes a move in VS AI mode
* Main Script: Uses Get Best Move
* Get Best Move: Goes over all possible moves until the max depth is reached
* Get Best Move: Uses Evaluate to get a score of the current chess board
* Evaluate: Returns the score

Exceptions: None

Open Issues: None

## Interactivity and Associations

Data Set

This project does not utilize an SQL database and as a result an ERD cannot be used. Instead, a data set diagram will be used explaining the relation of different data.

To begin with, it is important to describe the basic data form that is used throughout the board class, and that is the Piece struct. As mentioned earlier, it consist of an id, i and j coordinates, a type, an amount of points, a color and a list of moves. This struct is used both by the board and the pieces arrays in the Board class. The board 2d array uses 64 of them to represent the chess board, while the pieces array uses 32. The pieces on both arrays are linked, meaning a piece with a specific id in the board 2d array will be in the position specified by its coordinates on both arrays. The valid moves a piece can make are only stored in the pieces array but when calculating what moves a piece can make, the board 2d array is used as reference. The board has both active pieces and empty tiles, which have -1 as their id, ‘\*’ as the type and none as their color.

Another kind data that is frequently used is the uPieces array. This is an array of game objects that represent the pieces of both players. Each element in this array represents a different piece with that piece’s graphics. This array is mapped one to one with the pieces array in the Board class. The id of the elements in the pieces array is their index in that array and the elements of the U Pieces array have their ids set to map their specific piece. The data also include the i and j coordinates of that specific piece. For example, the white rook that starts on the bottom left of the chess board has the same id in the pieces and the U Pieces arrays and the i and j match to present the same tile both in the Board class’s board and the Graphics board. This is done so that it is easy to identify the moving piece and manipulate its position, as well as to link the board of the Board class with the graphics board. Also, when the user presses on a piece to move it is easy to identify it by its id, and then move the corresponding piece in the pieces array.

The byte array is a data form that compresses the current chess board. This data form exists to express the chess board as a sequence of numbers. This makes it easy to write the current board to a file as well as to store it in a list when a move is made.

The JSON files have their specific data structure as well:

The Save file only contains a byte array which is either read or overwritten.

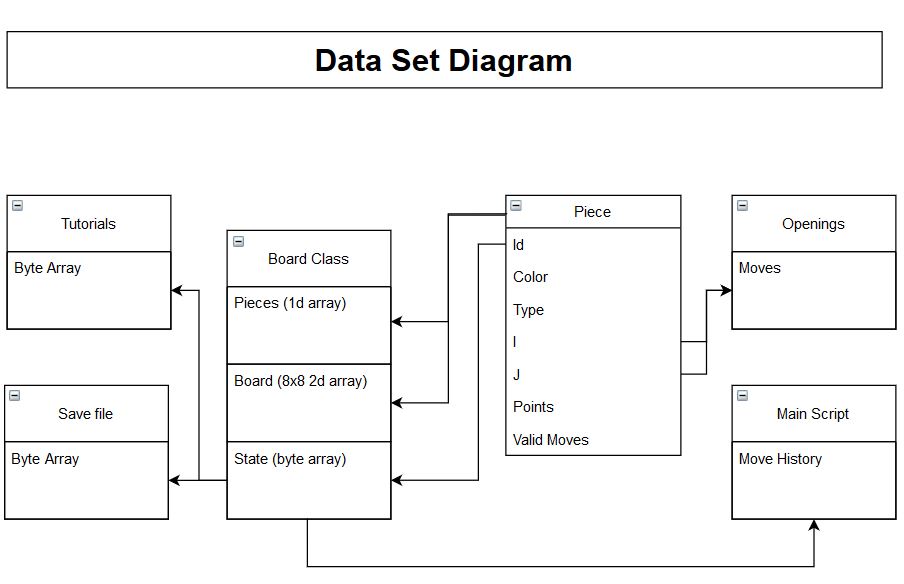
The Openings file contains two objects:

* Main Category: This object contains an integer array that is the move of the white side and a list of Sub Categories
* Sub Categories: This object contains an integer array that is the move of the black side in order to continue with the current opening and next, which is a list of Main Categories

This structure allows the openings to create a tree like relationship with a move leading to then next node. Also, this structure ensures that after a move by the white side is made, only the openings that contain the current state of the game will be examined.

The Tutorials file has a list of Levels. Each Level has a byte array, which is used to create a board and then display it, a text that explains which rule the current configuration of the board covers and a boolean. This boolean is true if a piece can be promoted. If a user that does not know how to play chess plays the tutorials, they might make a move that leads to a promotion. However, without any explanation it will be confusing. So, a promotion is only possible in the object which cover this topic.

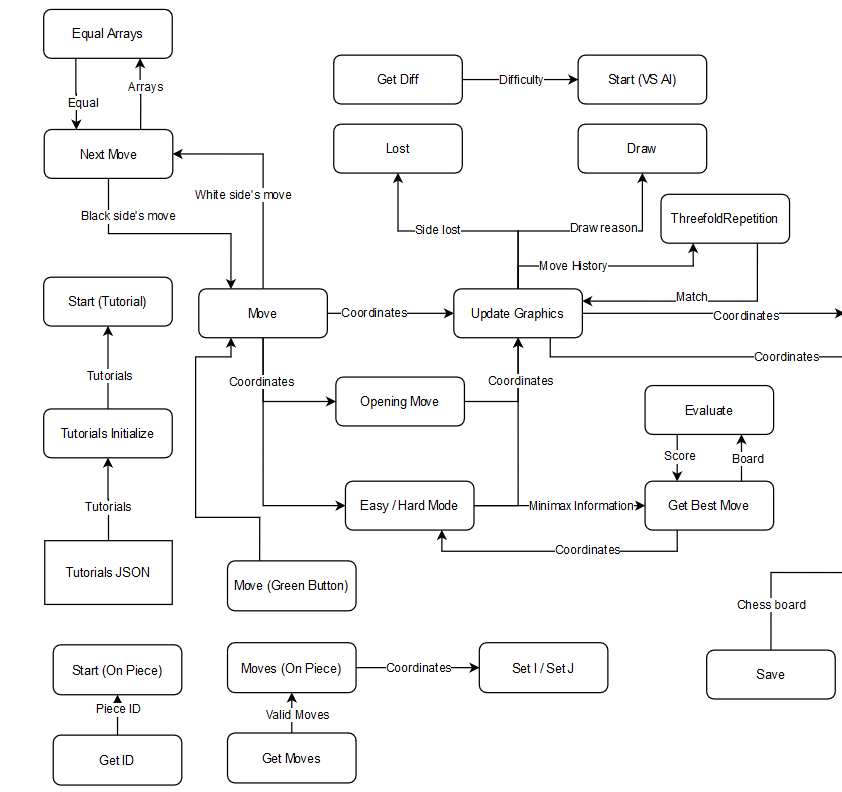
Data Set Diagram

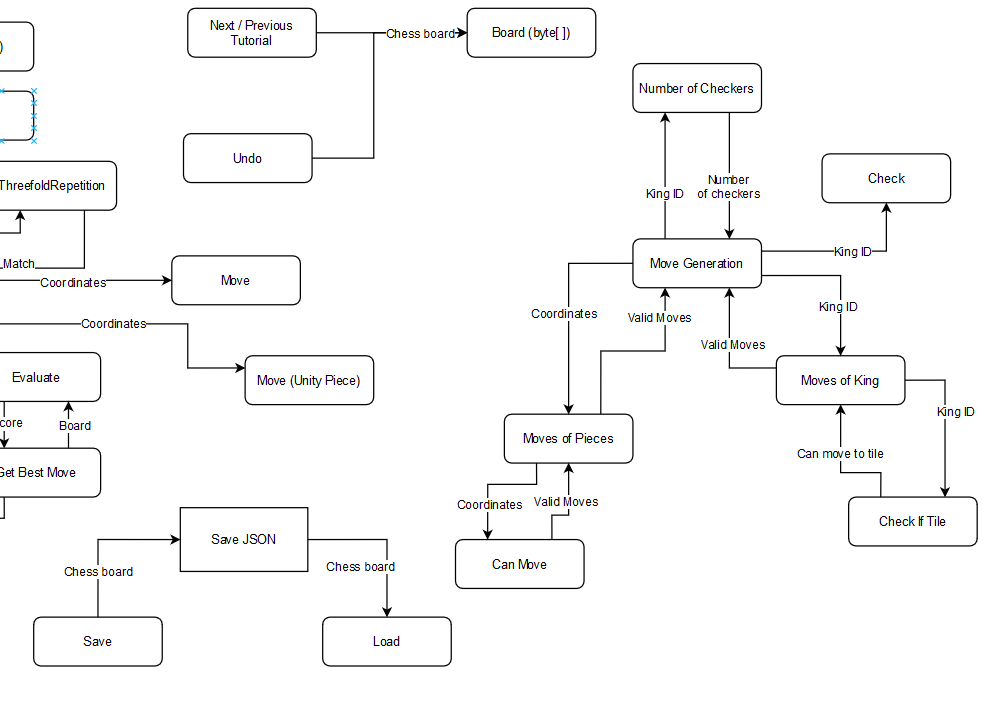


The above diagram shows how the data is connected throughout the application. The Board class contains the data on the board 2d array and the pieces array. They are both arrays of Pieces. The State is the byte array. It takes the id of every piece on the board and where there are no pieces it is set to -1 (it is a signed byte array). The Tutorials and Save files also use the state byte array. The Openings files uses a set of coordinates. These are from and to coordinates and use the I and J coordinates of the Pieces. Finally, the move history also uses the State byte array. The move history is a list of States.

Data Flow

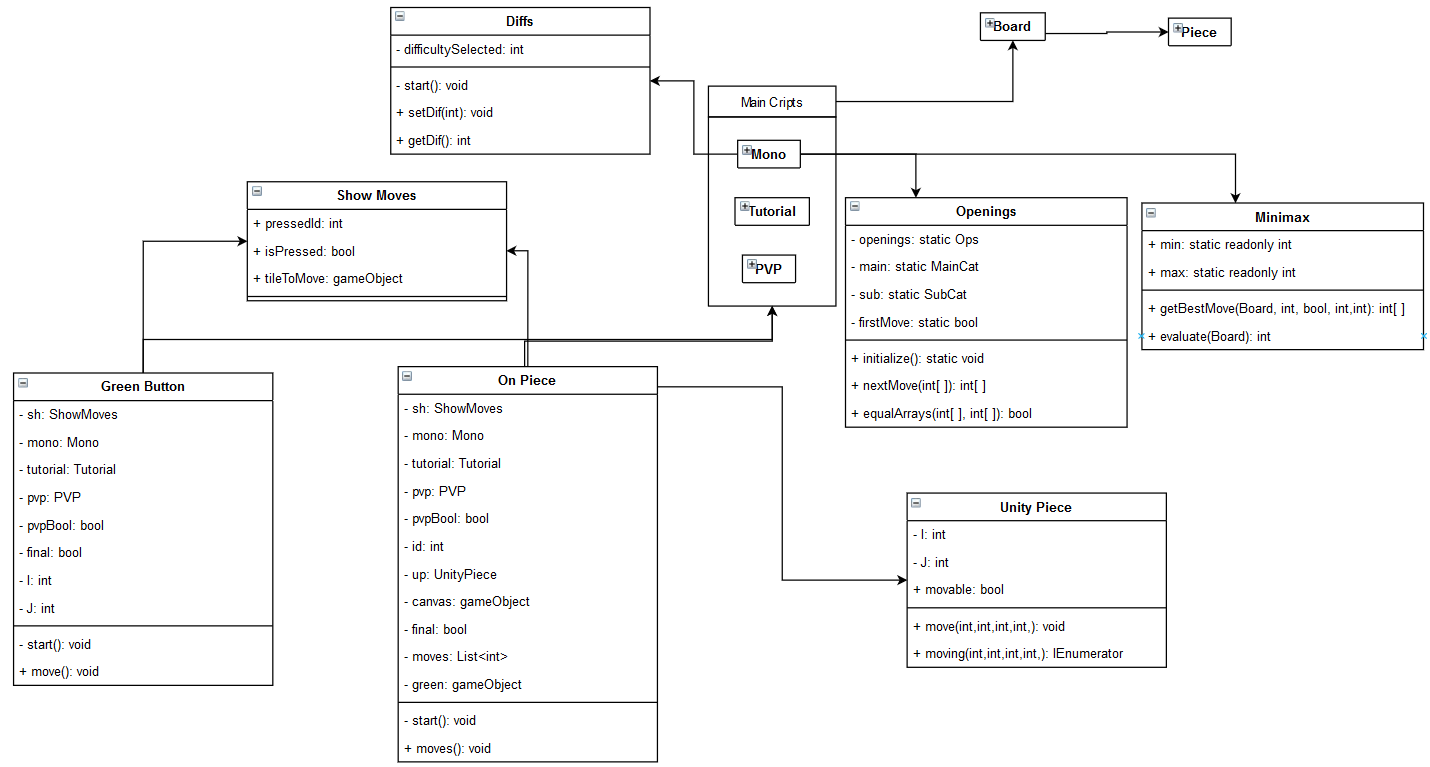
The Data Flow Diagram will be split in two images.





The reason it is split in two is that the original image was too big and the resulting resolution made the image blurry.

Class Model



Above is the Class Model of the application.

# System Documentation

## Technical Documentation

When the application is built, a folder will be created with a dll file, a crash handler and the executable file. The application starts when the executable file is run. The dll and the crash handler are automatically generated by Unity. Also, there will be 2 extra folders, also generated by Unity. The first will be a folder containing further dll files and the second will contain the application data. In that second folder is the Resources folder. Unity makes it easy to search for files by automatically navigating to that Resources folder. This folder will contain, among other files, the Save, Openings and Tutorials JSON files.

## User Documentation

The user documentation exists in the game in the form of the Instructions. It is easier for the user to look at the instructions of how the application works rather than having to download a manual. Also, the instructions have images of how the in-game interface looks like with explanations to make it easy to understand.

Detailed User Documentation

Main Menu

There are 5 options in the main menu:

* Instructions: Pressing this will open the Instructions panels. There are three panels explaining how the game is played and the user can move between Instructions by pressing the Next or Previous buttons. There is also a Back button that will return to the main menu.
* Tutorial: Pressing this will start the Tutorial mode. There, the user can play each of the Tutorial stages. By pressing the Next or Previous buttons the user can change the stage. Each of the tutorials has a chess board and a text underneath that explains a rule of chess. The user can play with the chess board to experiment. There is also a Back button that will return to the main menu.
* Practice: Pressing this will start the Practice mode. Here the user can play as both sides, changing a side after every move. The user can experiment with different openings and tactics in this mode. Pressing the Undo button will undo the last move and pressing Back will return to the main menu.
* VS AI: Pressing this will show the difficulties panel. Here the user can choose a difficulty (Easy, Medium or Hard) or press Back to return to the main menu. If the user chooses a difficulty, the VS AI mode will start. In this mode, the user can play against the AI. There are 4 buttons that give the user different options. Pressing Undo will undo the last two moves on the board, the last move of the black and the white side. Pressing Save will save the current game to the Save JSON file and pressing Load will load the currently saved game. By default, the Save JSON file contains a byte array that represents a chess board with the starting configuration. The user cannot press any of these 3 buttons while the AI is calculating its next move. The final option is Back, which will return to the main menu.
* Exit: Pressing this will quit the application.

When playing, the user can press on a piece to see its valid moves. In the VS AI mode, this will only happen when the user presses on a white piece and the AI is not calculating its next move. In the Practice mode, this will happen when the user presses on a piece that belongs to the side that can move. In the Tutorial, this will happen every time the user presses on a white piece. If the user has previously pressed on a piece and its valid moves are shown, the user can either press on that piece again, which will hide its valid moves or press on a new piece, which will hide the old piece’s valid moves and show the new piece’s ones.

When a pawn is promoted, the promotion panel is shown. In the VS AI mode, the AI will not make a move until the user has selected a promotion. In the other game modes, the user will note be able to make a move while the promotion is in progress. The promotion panel shows 4 pieces, a rook, a knight, a bishop and a queen. These are the pieces that the pawn can be promoted to. When the user chooses a promotion out of the 4, the game will continue.

When the game is over in the VS AI or Practice modes a text will appear stating which side won or, if the result is a draw the reason for the draw, and the Back button will be the only one left.

# Conclusions

All in all, the finished application has achieved all the requirements. Simple Chess contains both a tutorial stage that lets the user learn the rules of the game and a level where the user can play against different difficulties. All components of the software are working properly, and all features interact with each other correctly. During the development, many tests were conducted. These tests took place in a different Scene and a test was conducted every time a feature was completed. Afterwards, the testing scene was used to test the interactions between different features. First the main menu, then setting the difficulty and getting it. Then moving a piece and sending the same move from the main script to the Board class. The final tests were the ones with the algorithms. Through these tests certain errors were observed and corrected. There are also other similar software like The Chess Lv.100[14] and Decode Chess [15]. However, there are some differences. For example, The Chess Lv. 100 has 100 different levels of strength, which is a very useful tool for improving. It lacks in tutorials, however, and is not meant for users that do not know how to play chess. Decode Chess is another useful tool. Its strength lies in the ability to analyze a game, showing different moves and how these moves can affect the game. However, like before it is not meant for users that want to play a simple, casual game, and neither is it meant for users that want to learn. As stated in the application’s official website: “DecodeChess was created for chess enthusiasts who want to improve their skills and win every chess move”. On the other hand, Simple Chess is very useful for users that want to casually play or users that want to learn to play chess. Even so, there are possible enhancements that can be added in the future. The first is the expansion of the opening moves. Right now, the opening moves take into consideration only the most popular moves. However, the Openings file could easily hold the data of all possible openings. The second enhancement that can be added is online gameplay. The current version of the application is local, and a user can only practice or play against an AI. However, this could change by adding an online option and letting users play among themselves.

# References

1. Question on stack overflow: <https://stackoverflow.com/questions/494721/what-are-some-good-resources-for-writing-a-chess-engine>
2. TSCP chess: <https://sites.google.com/site/tscpchess/home>
3. The Chess Programming Wiki: <https://www.chessprogramming.org/Main_Page>
4. Laramée, F. D. (2000, May 17). Chess Programming Part I: Getting Started. Retrieved 2020, from <https://www.gamedev.net/reference/articles/article1014.asp>
5. Geeks for Geeks article by user shashipk11: <https://www.geeksforgeeks.org/design-a-chess-game/>
6. Asked on Stack Exchange by user SummerCode: <https://codereview.stackexchange.com/questions/71790/design-a-chess-game-using-object-oriented-principles>
7. Jones, P. E. (2017, February 09). Generating Legal Chess Moves Efficiently. Retrieved 2020, from <https://peterellisjones.com/posts/generating-legal-chess-moves-efficiently/>
8. Question on Quora by Eric Nichols: <https://www.quora.com/What-is-the-Algorithm-behind-a-computer-Chess-game>
9. Stack Overflow question by user anarhikos: <https://stackoverflow.com/questions/2026262/currently-known-best-algorithms-for-computer-chess>
10. Djurhuus, R. (2017, September 20). Chess Algorithms Theory and Practice. Retrieved 2020, from https://www.uio.no/studier/emner/matnat/ifi/INF4130/h17/undervisningsmateriale/chess-algorithms-theory-and-practice\_ver2017.pdf
11. Stack Overflow question by user Gurgeh: https://stackoverflow.com/questions/6305619/minimax-what-happens-if-min-plays-not-optimal
12. Hartikka, L. (2017, March 30). A step-by-step guide to building a simple chess AI. Retrieved 2020, from <https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/>
13. Link to Chess Tempo: <https://old.chesstempo.com/game-database.html>
14. Link to The Chess Lv. 100 on play store: <https://play.google.com/store/apps/details?id=jp.co.unbalance.android.chessunbcc&hl=en_US>
15. Link to Decode Chess website: <https://decodechess.com/ref/28/?campaign=/chess-software-windows-10/>

Newtonsoft Framework official website: <https://www.newtonsoft.com/json>

Newtonsoft on the Unity Asset Store: <https://assetstore.unity.com/packages/tools/input-management/json-net-for-unity-11347>

Images Used:

Images of the chess pieces found on Clip Art: <http://clipart-library.com/clipart/pcqrGKzLi.htm>

Background image found on Wallpaper Stream: <https://wallpaperstream.com/collection/black-and-white/Chess-Board-Black-and-White-Wallpaper>

Hourglass icon found in Vecteezy.com: <https://www.vecteezy.com/vector-art/582498-hourglass-icon-vector-illustration>

The logo of the application was found on ya-webdesign: <https://i.ya-webdesign.com/images/chess-clipart-chess-logo-10.jpg>

Sounds Used:

Found on Bensounds: <https://www.bensound.com>

Main Theme (The Lounge): <https://www.bensound.com/royalty-free-music/track/the-lounge>

Game Theme (Dreams): <https://www.bensound.com/royalty-free-music/track/dreams-chill-out>

Found on Sound Bible: <http://soundbible.com>

Victory Theme: <http://soundbible.com/1823-Winning-Triumphal-Fanfare.html>

Found on Zap Splat: <https://www.zapsplat.com>

Defeat Theme: <https://www.zapsplat.com/music/game-lose-negative-tone-1/>

Chess Piece Move: <https://www.zapsplat.com/music/apple-computer-mouse-click-3/>

Menu Button Theme: <https://www.zapsplat.com/music/compass-click-close-4/>

# Appendices

## Challenges

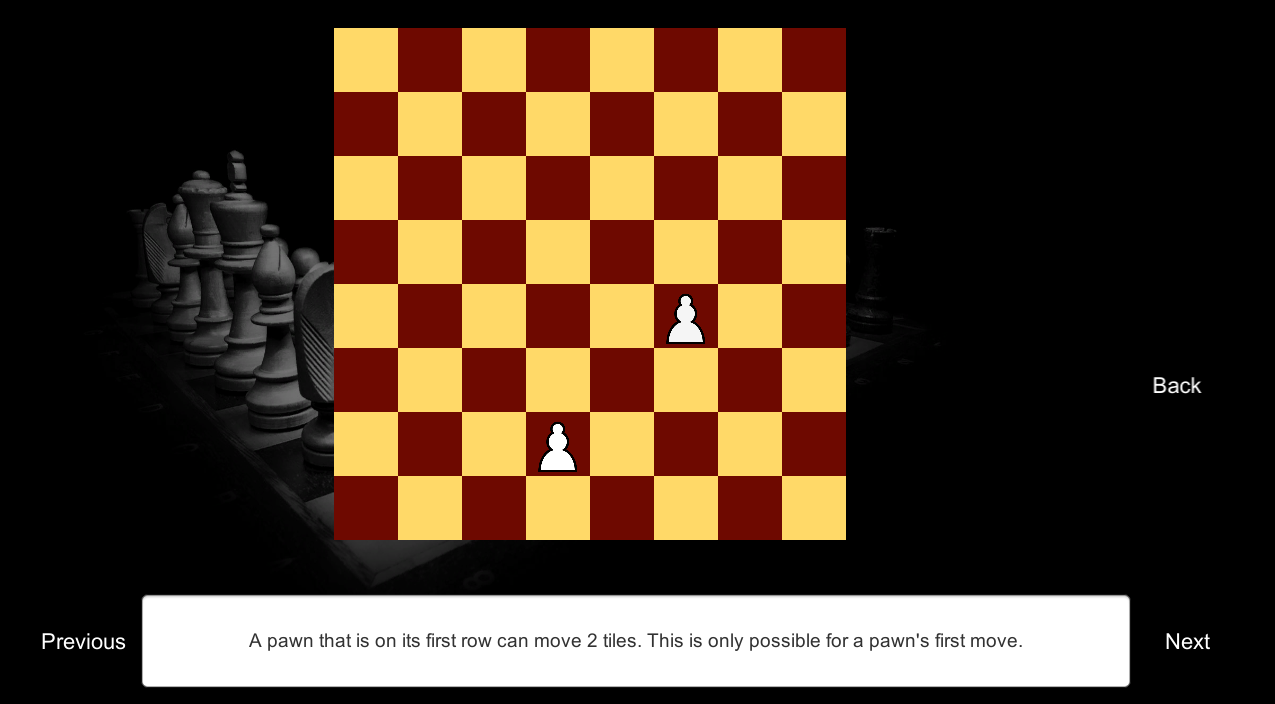
Moves that lead to the same side’s check

One big challenge was to determine what moves of a side would result in that side’s king being checked. For example, if the white king is in tile b2 and there is a black bishop in e5 that is a check. But if there is a piece between them, there is no check. However, if the piece between the white king and the black bishop is a white bishop, it can still be moved to c3. If the piece is a pawn in d4, it can eat the black bishop. This means that a piece can move freely, it can not move at all, or in certain situations it can make a limited number of moves. To solve this problem an enumerator is used with three values: none, limited or full. Also, the feature Can Move was changed to return a list of integers. It calculates if the piece can move and if it the criteria is correct (e.g. white bishop between white king and black bishop), then the limited moves are put in the list of moves. Then the list is returned to the Moves of Pieces feature like before. The first element in that list corresponds to a value of that enumerator. If it is none, the piece cannot move. If it is full, then Moves of Pieces will continue and calculate all moves that specific piece can make. However, if it is limited, the Moves of Pieces feature will return the rest of the elements of that list.

Another challenge come with Unity’s JSON Utility. This is a Unity feature that allows the serialization and deserialization of JSON files. However, as a precaution, JSON Utility does not read a JSON file where the object length is greater than 7. In most cases that is not a problem, but the openings JSON file is structured like a tree. It has initial nodes and every node expands to multiple other nodes. This creates a tree with different levels of depth, some of which exceed 7. In order to properly deserialize the JSON file the Newtonsoft JSON asset is used. Originally Newtonsoft is a popular framework for parsing JSON files in .Net but it also exists in the Unity Asses Store as an asset. After being used for the openings, it became the feature that parses all JSON files to have a single JSON parser rather than using two at the same time.

## Screenshots

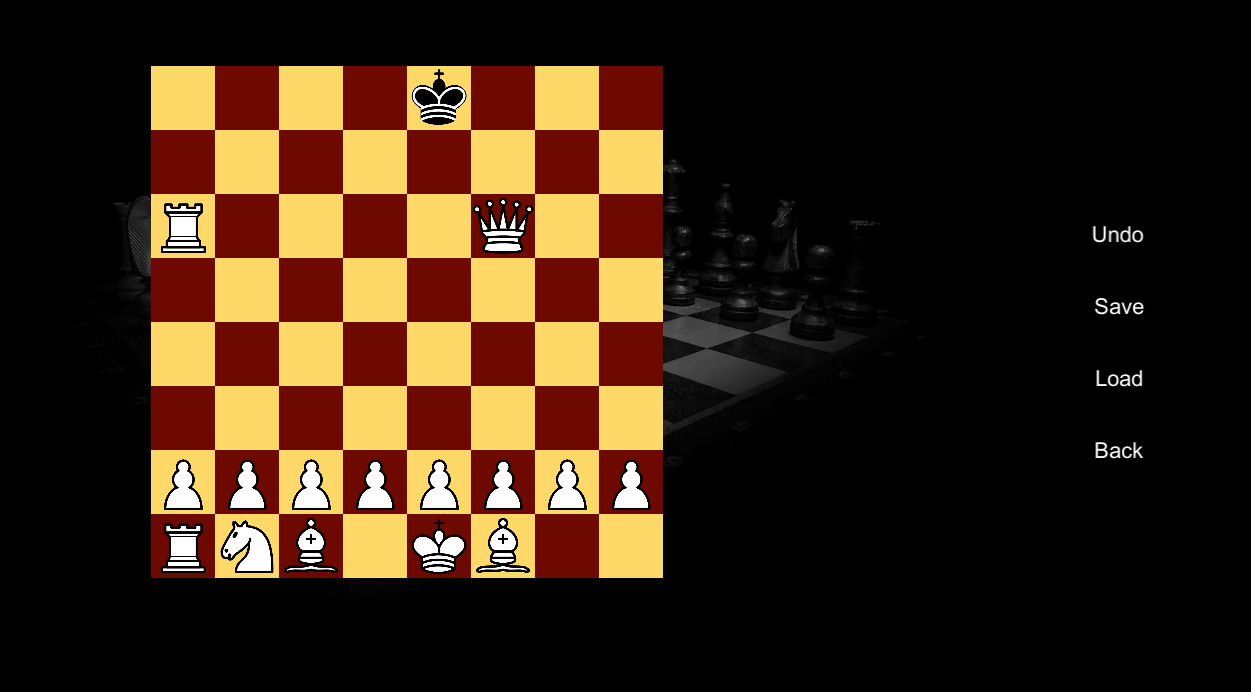
In the Tutorial mode the chess board changes according to the level. Pressing on Next or Back changes the toturial.



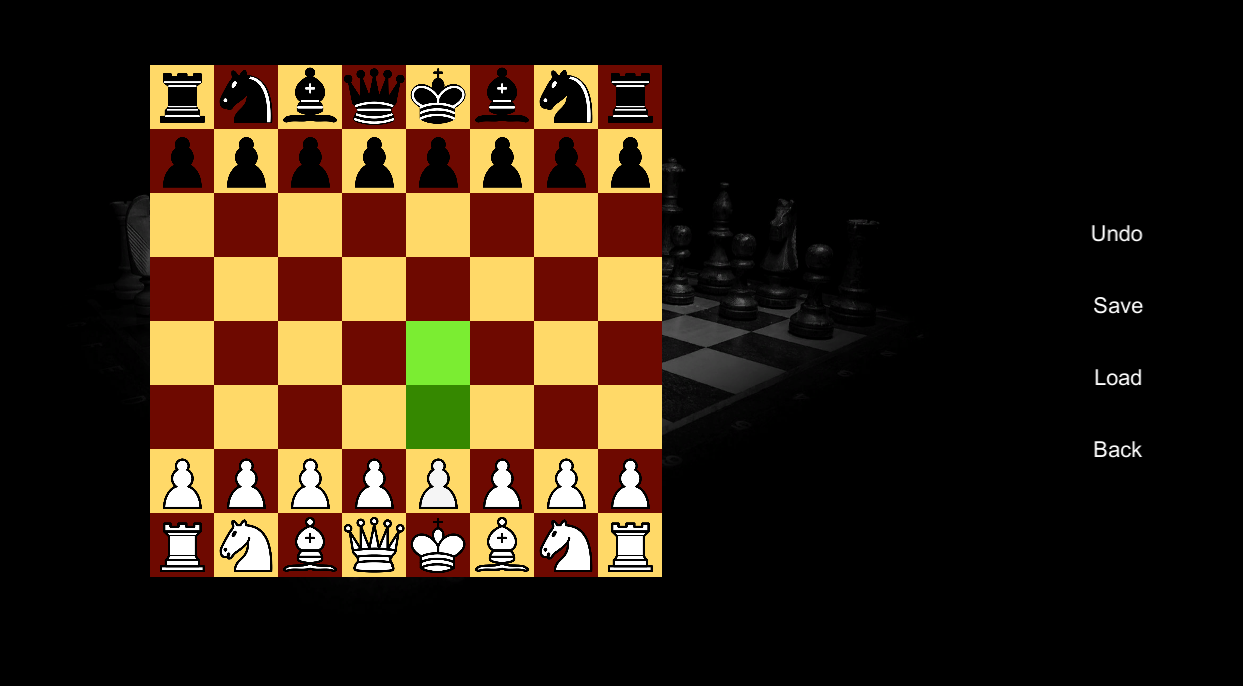
In the VS AI mode, the original chess board is created



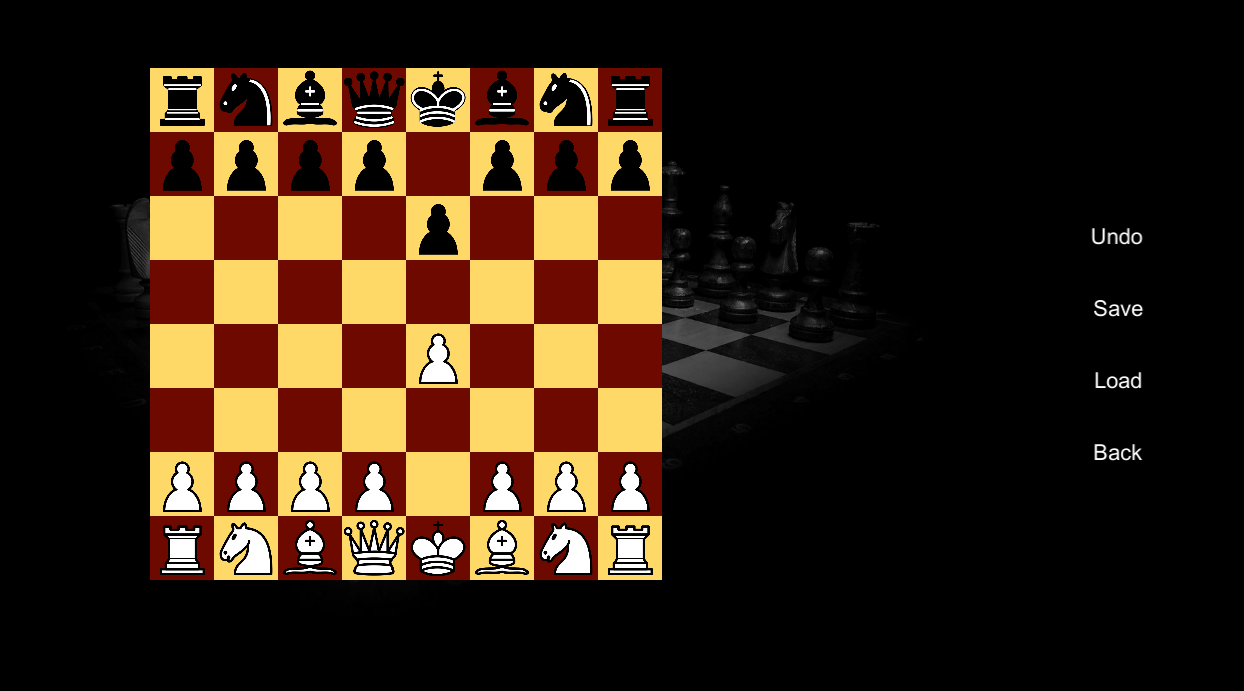
The user can load a saved chess board which changes the board



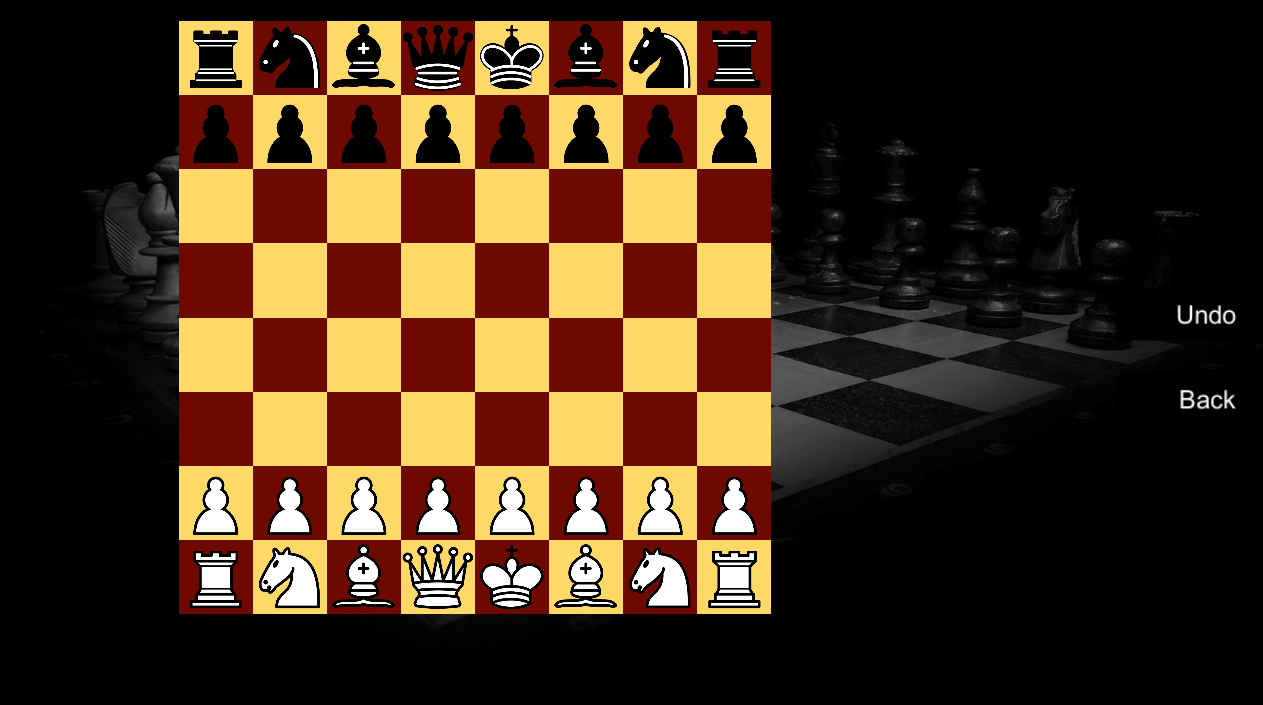
Clicking on a piece on the board shows its valid moves. Then, by pressing on a green tile, that piece moves

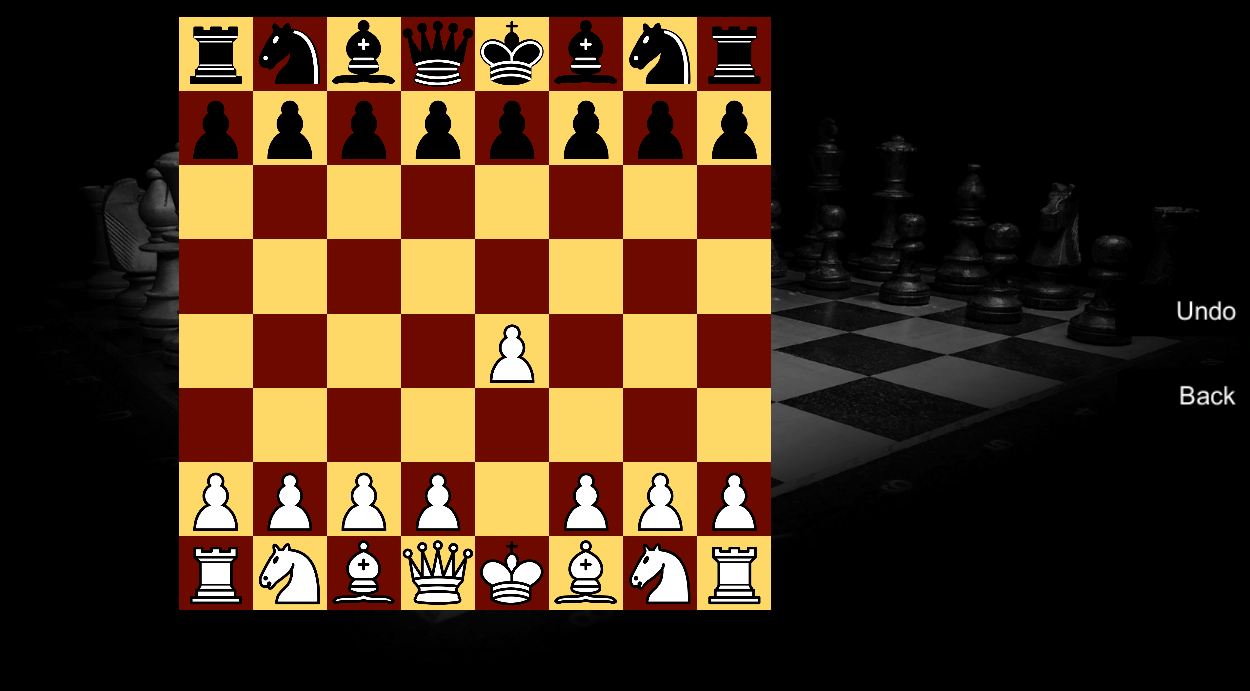


And after the user makes their move, the AI calculates and plays



In the Practice mode the user plays as both the white and black side, in turns

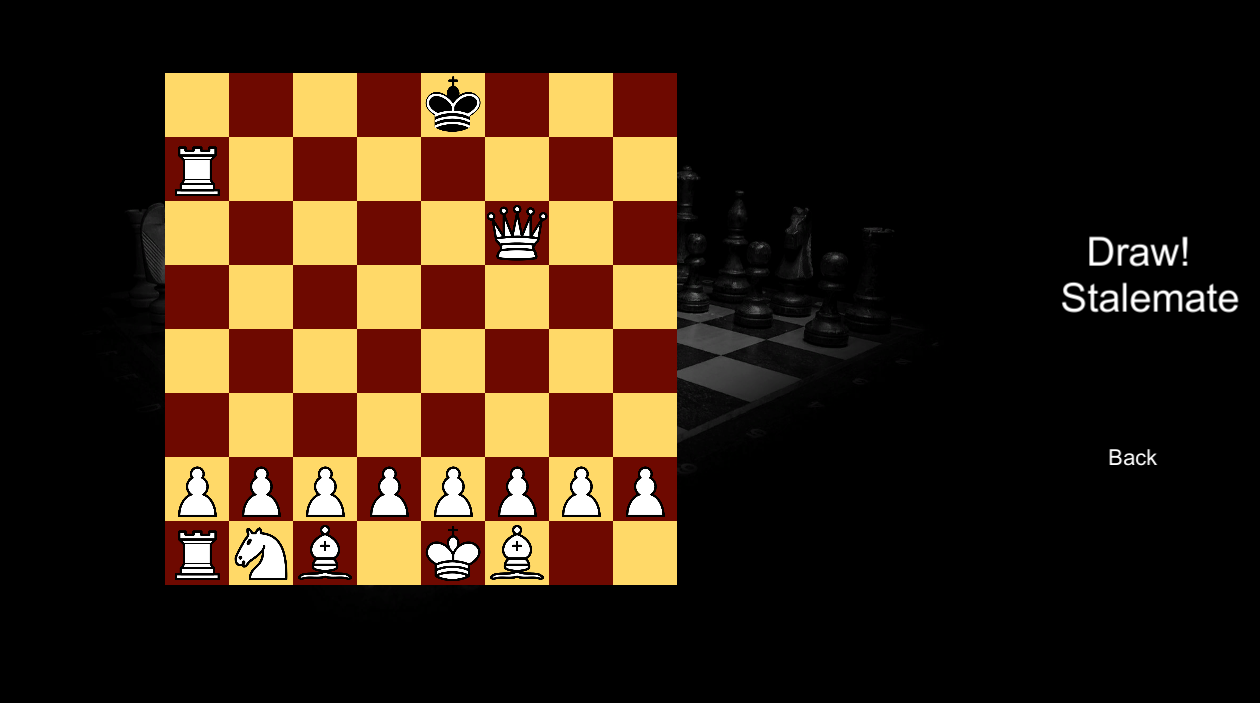


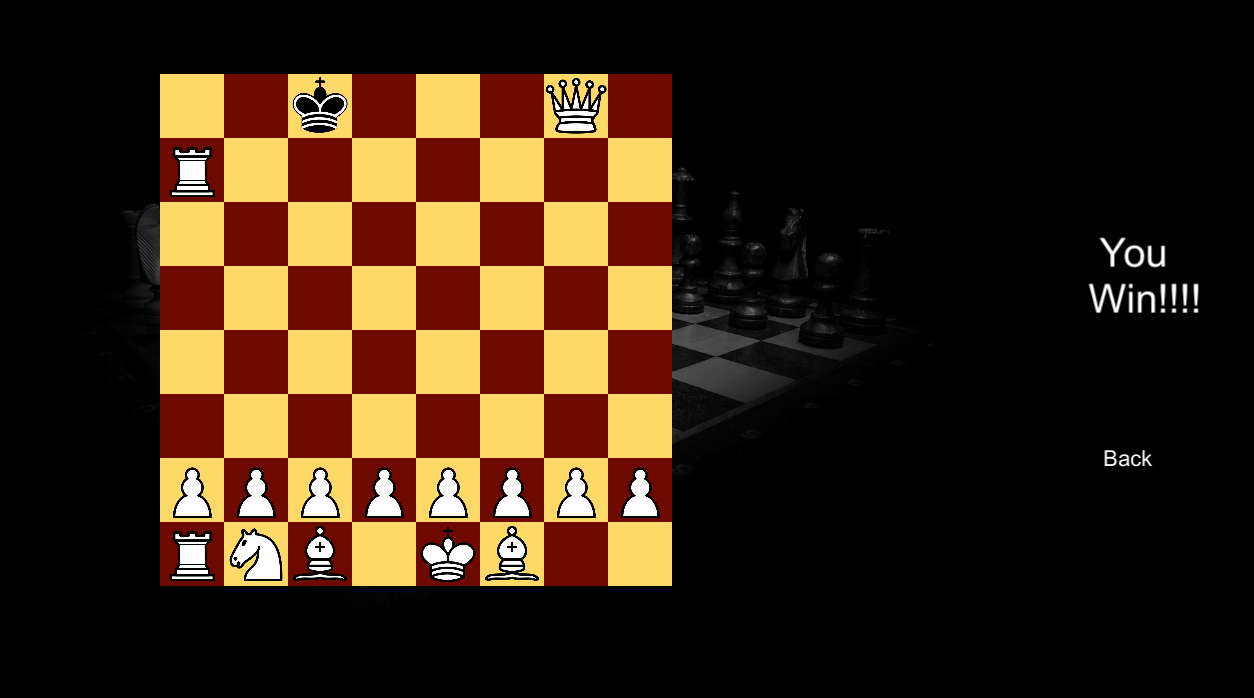


The Instructions show how the game is played



Finally, when a side wins or when it is a draw, the game ends





## Testing

The application is built with Bottom-up integration. First the important individual classes were created (Board and the Main Scripts). Then smaller components were created, like the On Piece and the Unity Piece ones. After finishing a component, extensive testing was carried out to ensure there are no error or exceptions. After all the small components were complete, the integration of these single units began. The testing was conducted with capabilities outside of the moves. These were the ability to move any piece anywhere on the board and the ability to create a lot of Board objects at once. The integration testing was complete after the different Scenes were working properly. Then the validation testing started. The testing of the menus and the different interfaces was first. Then the complete, continuous flow from the main menu to all modes. The final testing, the System Testing then began. Through this phase the tests were centered around the JSON files and possible problems with deserializing them.

## D. Supporting content

Key Terminology:

* Board
  + Pieces Array: It refers to the 32-element array that contains the Piece structures.
  + Board 2D Array: It refers to the 64-element array that represents the chess board.
* Graphics:
  + uPieces: It refers to the array of 32 objects. These objects contain 2 important components:
    - Unity Piece and
    - On Piece
  + Main Script: It refers to the script that handles a level. In the VS AI mode, the script is Mono. In the Practice mode, the script is the PVP. Finally, in the Tutorial mode, the script is the Tutorials.
  + Scene: In Unity, Scenes are used to create different levels. So, the Main Menu, Tutorial, Practice and VS AI are all different scenes.

Other Features Used:

* Coroutines: In Unity, coroutines act like simple methods with IEnumerator as a return type. As stated in the Unity Manual: “A coroutine is like a function that has the ability to pause execution and return control to Unity but then to continue where it left off on the following frame”. More on Coroutines can be found here: <https://docs.unity3d.com/Manual/Coroutines.html>
* Calculating the black side’s next move is done by using C#’s Threads. Normally, if a simple function or coroutine is used the game will freeze until the algorithm has returned the next move, so multi-threading is used to calculate that move without stopping other processes. So, in the case of an AI that might take more than a second to finish, a Thread would help. More on Threading in Unity can be found here: <https://docs.unity3d.com/Manual/JobSystemMultithreading.html>
* Game Object: In Unity, a game object is an object that exists in the game scene. It can have several components on it. The uPIeces array holds game objects, which all have the Unity Piece and the On Piece component on them. More on game objects can be found here: <https://docs.unity3d.com/ScriptReference/GameObject.html>
* Start: In Unity, start is a function shared by all game objects by default, although the developer can decide not to use it. It starts its activation when the game object first enters the scene. More on the Start function can be found here: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html>
* Struct: The Pieces are structs and not classes. The difference is that structs are value type, meaning that the original value is used. On the other hand, classes are reference type, meaning that when the variable is used, a reference to the class is used. More on structs can be found here: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct>

## Deployment Instructions

When downloaded, the application will come inside a folder, along with other files and folders. The application will start by clicking on the executable file Simple Chess.exe. The other 2 files are UnityCrashHandler64.exe and UnityPlayer.dll. These files are automatically generated by Unity when building the project. The 2 folders are MonoBleedingEdge and Simple Chess\_Data and they are both generated automatically. The Simple Chess\_Data folder also contains the Resources folder which is where the JSON files are located.