Data-Driven Soundtracks for Chess

Stefan Höller

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im November 2019

Declaration

Contents

Abstract

Chess is a complex game that most people have played and basically understand, but very few truly master. This complexity has been extensively studied, resulting in very large amounts of collected data that have led to nearly unbeatable virtual opponents. Nevertheless, few players can truly appreciate the dynamics of individual piece movements and their associated strategies. Sounds can provide the necessary context for players to understand the game easier.

A game of chess can be utilized to generate audio. Thanks to the development of many audio programming languages, it is possible to synthesize audio using chess as input in real-time. The project created for this thesis analyzes already played chess games and uses the calculated metrics as an input for controlling a soundtrack. The soundtrack aims to be subtle and enjoyable over longer periods of time, as chess games tend to be minutes or even hours long. The thesis determines promising chess metrics which provide better context for inexperienced chess player. Inexperienced chess players tend to wrongly evaluate a dangerous situation and subsequently make erroneous moves. To compensate this lack of knowledge the soundtrack provides the necessary information to signal the danger of the situations to the players.

The sound design was evaluated by three experienced but amateur chess players in a heuristic evaluation, which revealed issues in the implementation. A metric list ranks the metrics for their suitability in the sound design. A chess metric evaluation illustrates their average development throughout nearly a thousand chess games. Furthermore, a real-time analysis evaluates the chess parameters' versatility in a real-time scenario. The concept for a chess soundtrack for inexperienced players has the potential to provide new insights, as well as being applicable for learning chess.

Kurzfassung

Schach ist ein komplexes Spiel, welches die meisten Menschen schon einmal gespielt haben und im Grunde verstehen, aber nur sehr wenige wirklich beherrschen. Diese Komplexität wurde eingehend untersucht, was in einer sehr großen Menge an gesammelten Daten resultierte und zu fast unschlagbaren virtuellen Gegnern geführt hat. Dennoch können nur wenige Spieler die Dynamik der einzelnen Züge und die damit verbundenen Strategien wirklich schätzen. Töne können den notwendigen Kontext für diese Spieler bieten, um das Spiel besser verstehen zu können.

Eine Schachpartie kann zum Erzeugen von Tönen verwendet werden. Dank der Entwicklung vieler Audio-Programmiersprachen ist es möglich, Töne mit Schach als Eingabe in Echtzeit wiederzugeben. Das für diese Arbeit erstellte Projekt analysiert bereits gespielte Schachpartien und verwendet die berechneten Metriken als Eingabe für die Steuerung des Soundtracks. Der Soundtrack soll subtil und angenehm über einen längeren Zeitraum sein, da Schachpartien in der Regel Minuten oder sogar Stunden dauern können. Die Arbeit ermittelt vielversprechende Schachmetriken, die einen besseren Kontext für unerfahrene Schachspieler bieten. Gelegenheitsspieler neigen dazu, eine gefährliche Situation falsch einzuschätzen und anschließend fehlerhafte Züge auszuführen. Um diesen Mangel an Wissen auszugleichen, liefert der Soundtrack die notwendigen Informationen, um den Spielern die Gefahr der Situation zu signalisieren.

Das Sounddesign wurde von drei erfahrenen Amateurschachspielern in einer heuristischen Bewertung analysiert, die geholfen haben Probleme bezüglich der Implementierung aufzuzeigen. Eine Liste von Schachmetriken reiht die Metriken nach deren Tauglichkeit im Sounddesign. Eine Bewertung der Schachmetriken veranschaulicht deren durchschnittliche Entwicklung in fast 1000 Schachpartien. Darüber hinaus bewertet eine Echtzeitanalyse die Verwendbarkeit der Schachparameter in einem Echtzeit-Szenario. Das Konzept eines Schach-Soundtracks für unerfahrene Spieler hat das Potenzial, neue Einblicke in das Spiel zu bieten, als auch für das Lernen von Schach anwendbar zu sein.

Chapter 1

Introduction

This thesis describes a data-driven soundtrack for chess, the chess-related principles involved for determining a set of chess metrics suitable for such a soundtrack, as well as the background for creating an adaptive soundtrack in detail.

1.1 Motivation

In a game of chess, two players try to anticipate each other's next moves to checkmate before the other player seizes the opportunity. This does not only include the knowledge of the possible moves by all pieces, but even more the right utilization of strategies and avoidance of errors. Unless players practice the game, it is almost impossible to comprehend what is happening on the chessboard. Therefore, most strategies behind the game remain hidden to casual players. Hence, to clarify the meaning for these strategies, sound can highlight these dynamics. Generating sound based on the data of a chess game was developed as an art installation and even as software system to introduce non-musicians to musical composition. Each project triggers sounds when moving chess pieces. As a result, this can be used to create an impression of the dynamics of chess games. Despite adding a new level of perception, these systems do not try to provide new context to the game itself, but rather emphasize on sound generation. Analyzing a chess game and incorporating sounds which interpret the player's performance could increase the insights into the game's strategies. To enable players to identify the current situation properly, the system requires unique sounds which transmit certain meanings.

1.2 Problem Statement

A soundtrack for chess giving new insights for inexperienced players has the novel problem of determining a set of chess metrics providing the right information in any given situation. As for chess analysis, chess engines utilize a large set of parameters to evaluate chess and compute the best moves for any position. Chess engines aim to surpass the best chess players and since the 90s chess engines have had little difficulty in defeating chess grand masters. Therefore chess engines are a huge source for useful metrics for this thesis. Chess engines evaluate metrics such as threats on the board, material, development, etc. The problem of creating this soundtrack is the determination of the right set of metrics suitable for providing enough context for inexperienced players, yet not overcomplicate the sounds by utilizing too complex metrics that are not comprehensible.

For this reason this thesis aims to analyze chess, evaluate the metrics suitable for controlling music and develop an audio component that can play an adaptive soundtrack based on chess game data sets. The soundtrack should accompany chess games which can range from relaxing to alerting sounds. This emphasizes the situation in which the players are.

1.3 Document Structure

Chapter 2 explains the necessary concepts and principles utilized in the thesis. These principles include attacks, guards, threats, exchanges, pins, blunders, etc. After the explanation of those principles, an overview of the development of artificial chess players is provided, ranging from the mechanical *Turk* to modern chess engines such as *lz0*. Apart from the historical overview, a detailed view on the concepts and algorithms behind chess engines is presented. Additionally, it provides a description of the chess engine utilized in the thesis project.

Chapter 3 switches from the context of chess to audio synthesis. The chapter presents the state of the art of audio synthesis, explains why *FMOD* was chosen as the audio framework utilized in the project and provides a detailed description of *FMOD*'s functionality. After that, related works are presented, such as John Cage's *Reunion* as well as diverse follow up projects, i.e., *Reunion2012* and *Music for 32 Chess Pieces*. These projects all use a chessboard to control the generation of sounds or music.

After Chapter 2 and Chapter 3 provide the necessary background information on the topic at hand, Chapter 4 proceeds to explain the concept for a data-driven soundtrack. The chapter explains the requirements the implementation needs to meet, as well as possible limitations. Finally, two different soundtrack designs are presented, the first using music tracks and the second utilizing basic music notes to create a subtle soundscape. Each of those sound designs explains the soundtracks structure, chess inputs as well as audio outputs.

Chapter 5 describes the architecture and concrete implementation of the second soundtrack design from every angle. It provides details on the chess analysis, the sound design created in *FMOD* and the program consuming both an analyzed chess game and the *FMOD* soundtrack to control it via the game's data.

Chapter 6 proceeds to evaluate the sound design and implementation by conducting a heuristic evaluation and a concrete chess metrics evaluation. The heuristic evaluation utilized the knowledge of three experienced but amateur chess players to discover problems within the soundtrack's implementation. The evaluators provided substantial feedback, proving the soundtracks concept while expressing its potential for improvements. The metric evaluation focuses on the visualization of chess metrics and providing information for how and when to use the analyzed metrics.

Chapter 7 concludes the thesis elaborating the evaluations findings as well as future works similar to John Cage's *Reunion* with the extension of providing context for inexperienced players. Lastly, a conclusion is drawn over the thesis.

Chapter 2

Chess

This chapter starts by explaining the most important principles and factors of chess for this thesis. A number of moves are explained which should enable the reader to grasp the ideas and concepts discussed in later chapters. After the introduction to the game itself, a historic overview of chess programs and engines is illustrated. This overview contains the major milestones in pursuing the creation of an artificial chess player. The chapter goes on to describe principles and concepts for modern chess engines, the open-source engine *Stockfish* and standards for chess programming.

2.1 Principles and Metrics

In this section general ideas and principles are discussed that are important for later sections in the thesis. The most basic rules of chess are omitted, as most readers will be familiar with the pieces, their basic movement and captures.

2.1.1 Move

A half-move or *ply* is the movement of a chess piece on the chessboard by a player. A full-move incorporates two plies, including a white ply and a black ply. After a full-move it is the white player's turn again. Throughout this thesis, a half-move is called *ply* and a full-move simply *move*.

2.1.2 Attacks

An *attack* [4, p. 324] is a move which can capture an opponent's piece. Often an attack forces the opponent to react and defend the piece. In Figure 2.1 (b) the white knight on `c3` attacks the pawn at `d5`.

Threats

A *threat* is an advantageous move that would be played if it were not the opponent's turn, or rather will be played if the move is not prevented beforehand [4, p. 346]. Mostly, this includes possible captures that are either undefended or a higher valued piece than the attacker [42]. A threat is an attack, but an attack is not necessarily a threat, as this depends on the involved pieces' values. In chess, the players try to build up threats, as they can gain material and therefore a significant advantage. A threat can be countered by either moving the threatened piece to a safe square, moving a less valuable piece to block the threat, capturing the attacker, or defending the piece when the piece's value is lower or equal the attacking piece's value. For example on Figure 2.1 (a) the white pawn on `e4` threatens to capture the more valuable black knight on `d5`.

Attacker

An *attacker* or *attacking piece* attacks one or multiple of the opponent's pieces. Attackers are shown on Figure 2.1 (a) on squares `c3`, `d5`, and `e4`.

Defense

Defense [4, p. 327] is a response or parry to an attack. It tries to either prevent an attack to be converted to a capture by attacking the attacker or capture the attacker in the following move after the capture took place.

Guard

A *guard* or *guarding piece* protects another of its own pieces by being able to move to the pieces square after it was captured by the opponent and in turn capturing the attacker. Guards are illustrated on Figure 2.1 (a) in which the black queen protects the knight on `d5`, but is rather inadequate as a recapture of a pawn would not compensate the loss of a knight. The white bishop on `d2` protects both the rook on `c1` and the queen on `f4` in Figure 2.1 (d).

Undefended Piece

An *undefended* or *hanging* piece [4, p. 333] is a piece or pawn that is not guarded by another piece. These pieces are open to attacks and material can be lost easily. The white queen on `c3` in Figure 2.1 (c) is neither threatened nor protected from other pieces and therefore undefended.

Exchange

An *exchange* [4, p. 331] is a two ply sequence that involves capturing an opponent's piece, while knowing the opponent will recapture. An exchange often times involves two pieces of equal value being exchanged, i.e., a white queen captures a guarded black queen and the guarded piece recaptures the white queen. The result is the removal of two equally valued pieces from the chessboard. An exchange can be won by capturing a higher valued piece, while giving up a lower valued piece. Figure 2.1 (a) shows an initial position for an exchange to take place. The black knight captures the white knight (`6...Nxc3`) and the white pawn on `b2` completes the exchange by capturing the black knight on `c3` (`7.bxc3`). Thus, both players removed a knight from the board in two plies.

Pin

A *pin* [4, p. 338] is an attack made by a sliding piece—queen, rook, or bishop—targeting an opponent's piece. But moving the attacked piece may be ill-advised, as it would reveal a more valuable piece to the attacker. An *absolute pin* occurs, if a king is concealed by an attacked piece, the attacked piece must not move, as moving the king into check is illegal. A *relative pin* involves a piece more valuable than the attacked piece. Figure 2.1 (b) display both a relative and absolute pin from white. The bishop on `a6` pins the pawn on `b7`. Moving the pawn would threaten the rook on `c8`. The white queen on `a4` imposes an absolute pin on the black knight on `c6`. The black knight must not be moved as it would reveal the black king to the white queen, which is illegal.

Skewer

A *skewer* [4, p. 343] is an attack along a line made by a sliding piece. It involves two valuable pieces. Moving the attacked piece to a safe square, would open the other valuable piece to the attack. In Figure 2.1 (c) an absolute skewer from the white bishop on `e5` to the black king on `f6` is created. The black king has to move either by capturing the white bishop or moving to a safe square and in turn reveal the queen to the bishop.

Fork

A *fork* [4, p. 332] is an attack by a piece attacking two or more of the opponent's pieces simultaneously. The defender has to chose which piece is more valuable and save it. It is not possible to save both pieces as only one can be moved, except when the forking piece can be captured. In Figure 2.1 (d) the black knight on `d3` attacks the white rook on `c3` and the white queen on `f4` simultaneously.

2.1.3 Castling

Castling [4, p. 326] is a move in which a king and a rook essentially switch places. The move serves to move the king away from the center and enables the rook to be involved in the game.

2.1.4 Blunder

A *blunder* [4, p. 325] is a move which worsens your situation significantly. It eliminates your advantage or turns a good position in the game into a lost position.

2.1.5 Material

Material [4, p. 335] compares the number and values of pawns and pieces for both players on the chessboard. If the two players have the exact same pawns and pieces on the board, they have the same material. A material advantage is defined by having more pieces or more valuable pieces than the other player on the board.

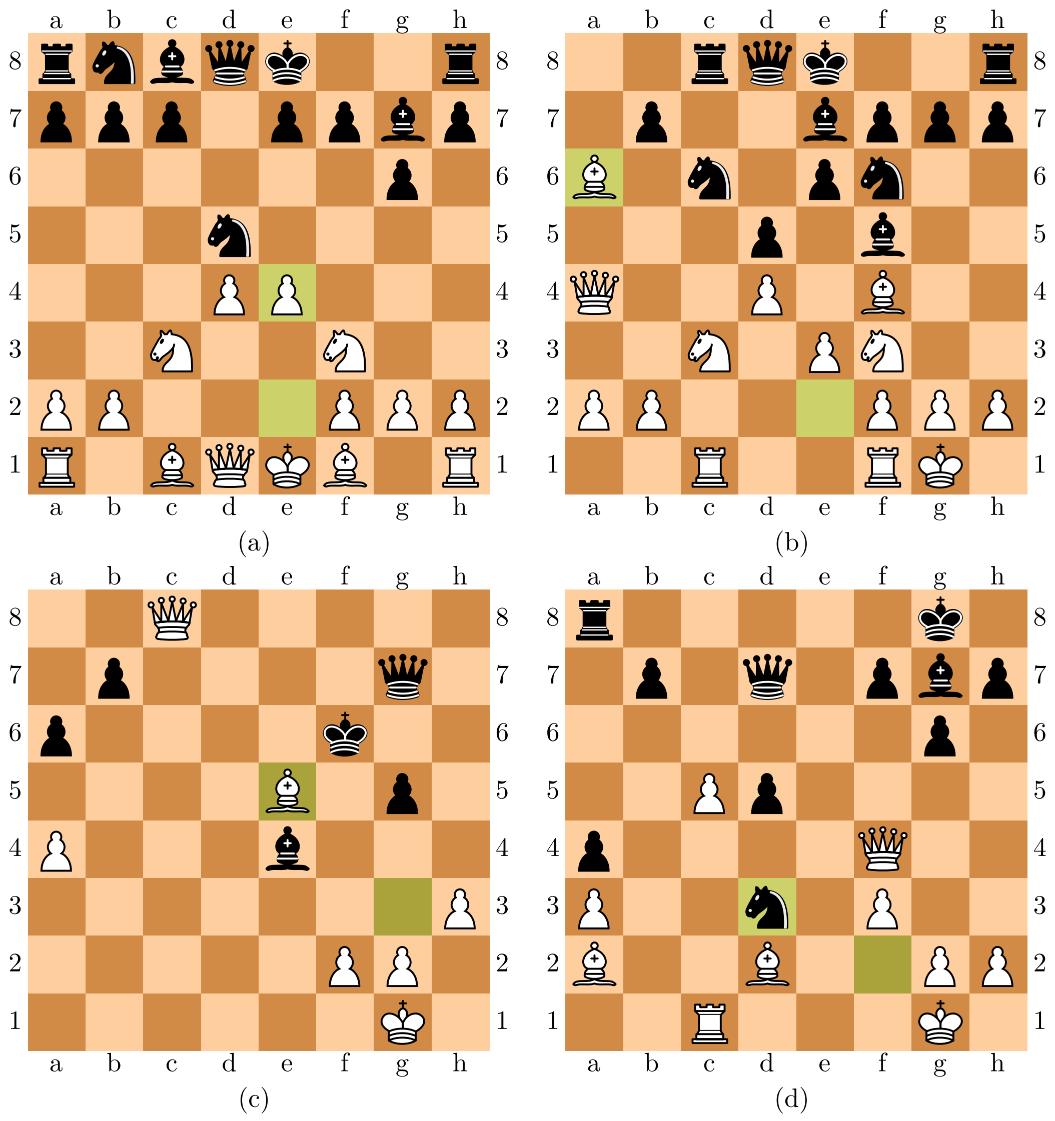


Figure 2.1: (a) Exchange of Knights at move `6...Nxc3`, `7.Bxc3`, Kramnik vs Leko 2001, (b) Relative and absolute pin at move `13.Bxa6`, Kramnik vs Anand, 2007, (c) Absolute skewer at move `51.Be5+`, Short vs Vaganian, 1989, (d) Fork at move `34...Nd3`, Tissir vs Dreev, 2007.

2.1.6 Mobility

Mobility [4, p. 336] is often defined as the number of possible moves a player has at hand. It describes the rate of pieces moving freely on the chessboard. The higher the number the better.

2.1.7 Initiative

Initiative [4, p. 334] is the ability to develop or having already established threats on the board. Thus, creating a situation in which the opponent cannot create threats as they have to react to threats. At the beginning of the game the white player has initiative, as white moves first. While developing pieces, white has the advantage of positioning its pieces first and creating the threats black has to defend [5, p. 25].

2.1.8 Pawn Structure

The *pawn structure* [4, p. 336] consists of the various positions of pawns on the chessboard. It defines a player's territory. An effective pawn structure has the ability to protect and guard other pieces, as well as allow pieces to move past it. It also holds the opponent's pieces at bay, as the pawns attack key squares on the board.

2.1.9 Passed Pawn

A *passed pawn* [4, p. 337] has either no opponent's pawn in front of it or non of the opponent's pawns is able to capture the pawn. A passed pawn can move towards the end of the chessboard to be promoted. A passed pawn is an advantage in the end game, as it can lead to an decisive advantage.

2.1.10 Development

Pieces are developed when they are moved to their most advantageous position and therefore can unfold their potential [4, p. 327]. Important actions in development are moving bishops, knights and the queen from their starting position, castling and connecting the two rooks together. When a player is ahead in development this player can try to use this advantage and attack the opponent.

2.1.11 Imbalances

Imbalances [29, pp. 3–28] are a significant difference in two respective positions. Two similar positions are compared to each other and it is determined whether a player has an upper hand on a specific criterion. The criteria can include: superior minor piece, pawn structure, space, material, control of a key file, control of a hole/weak square, lead in development, initiative, king safety, etc.

2.1.12 Phases of Chess

Chess can be divided into three phases, opening, middle game and end game. The boundaries between the phases are not set in stone, as chess is highly versatile. Nevertheless, the following definitions try to make the roles of the phases clear.

Opening

The *opening* [23, pp. 135–141] is the initial phase in which the players try to bring the more valuable pieces into play. This also requires the pawns to move to make way for the pieces to move past them. Often times pawns are developed towards the center to have influence on important center squares. The king should be castled either kingside or queenside for protection. The castling enable the rooks to be connected and in the center. Standard openings can help the players to accelerate the opening phase. After the development of pieces has more or less been completed, the middle game starts.

Middle Game

The *middle game* [23, pp. 142–152] follows the opening. Most pieces have been developed and maybe few captures have taken place. In this phase the players devote their concentration on to creating threats, counterattacks, and the objective of gaining material. The players should follow a plan and work on its implementation. Occupying crucial squares or files is important to impose threats. On top of that the players should not disregard the safety of their kings. Hiding them behind pawns and defending possible threats is important for the king safety. In contrast to the opening, the middle game does not offer clear guidelines and requires the skill and creativity from the players to defend their pieces, while gaining material.

End Game

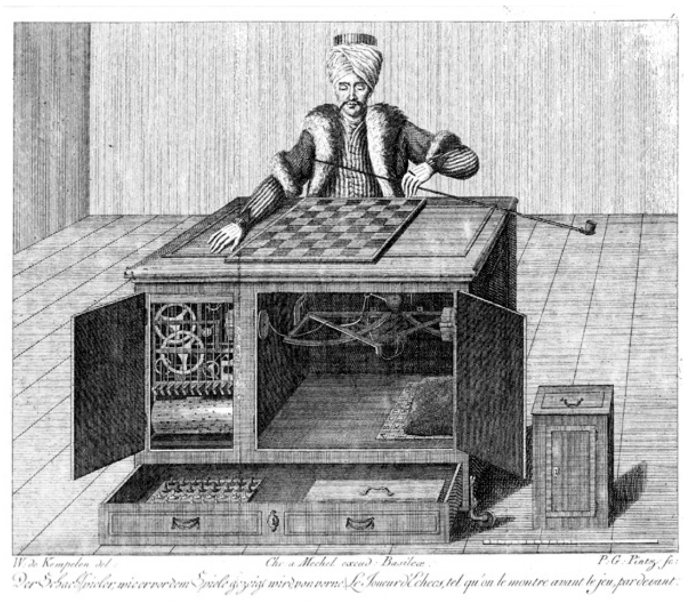
After most pieces on the board have been exchanged, the *end game* [23, pp. 152–156] starts. Typically, the chessboard hosts a small number of pawns, one or two pieces—such as a rook, or a bishop—and the king. In the end game players often try to promote a pawn to a queen to gain a decisive advantage. The king was previously heavily guarded and hidden behind their own line of pawns, and now is encouraged to move around the board. The king can escape enemy attacks or prevent passed pawns from being promoted. The game ends when the king cannot avoid being checkmated or a draw or stalemate occurs.

2.2 Chess Engines Overview

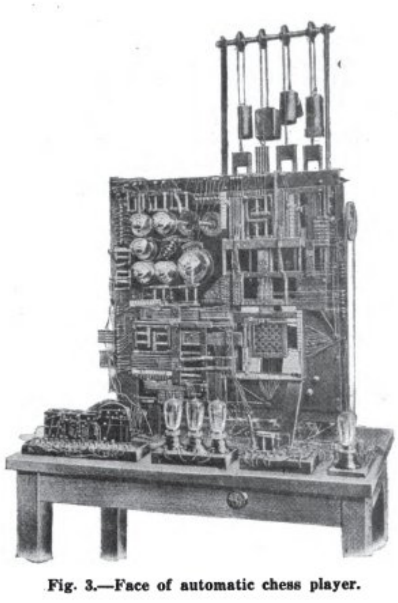
In this section the historic timeline of creating an artificial chess player and program is explored. This includes the major milestones and advances made in that area since the inception of the *Turk* in year the 1770.

2.2.1 First Mechanical Chess Players

The idea of a machine capable of playing and winning chess against humans first manifested itself in 1770. Wolfgang von Kempelen constructed a chess player called the *Turk* [7, pp. 154–163]. The *Turk* consists of a cabinet housing a complex mechanism, a chessboard fixed onto the surface of the cabinet and a full scale model of the *Turk* itself with a torso, a nodding head, moving arms and a smoking pipe, as seen in Figure 2.2 (a). Before the *Turk* starts playing against a visitor, the exhibitor opens every door and drawer to reveal gearwheels, barrels and pulleys to the audience. Except for a space large enough to fit a fully-grown adult, the operator, into the machine. The operator sits on a sliding seat and by moving the seat forward a set of dummy machinery folds down concealing the operator [31, Ch. 11]. The operators needs to conceal themselves in a similar fashion until every door of the cabinet is opened. After the initial inspection by the audience, the operator assumes an upright position and the game begins. The operator uses a second chessboard to keep record of the game and a series of levers to



(a)



(b)

Figure 2.2: (a) The *Turk*: Copper engraving depicting the *Turk*'s front side showing its interior. [40], 1783, (b) *El Ajedrecista*: face of automatic chess player [32].

control the *Turk*'s arm to pick pieces and make moves. The magnets on the underside of the exterior chessboard signal to the operator which piece was moved. Writer Edgar Allen Poe [25], mechanical engineer Robert Willis [36] and Joseph F. von Racknitz [27] to name a few who attempted to uncover the secrets behind the *Turk*, all suspected a hidden human chess player, but made wrong assumptions regarding the mechanics. The inner workings of the *Turk* were revealed by Silar W. Mitchell [16], son of the *Turk*'s last owner, after the device was lost in a fire in 1854. The *Turk* was the first attempt at creating an artificial chess player. Ultimately a failed attempt as it required a human chess player to make its moves.

The significant development happened in 1914, when Leonardo Torres y Quevedo accomplished a more authentic approach to the mechanic chess player. The machine called *El Ajedrecista*, which translates to English as *The Chess Player*, is depicted in Figure 2.2 (b). Torres constructed a real machine capable of checkmating a black king—the black king played by the human spectator—by only using a white rook and white king [32]. A double slide mechanism can move chess pieces to any square by clamping the pieces. When the black king is moved, the machine first checks the legality of the move. The machine signals an illegal move by turning up a light bulb. After a legal move by the spectator, the machine determines a move and moves either rook or king by one square. This procedure is repeated until the black king is checkmate. While the machine by Torres y Quevedo accomplished solving a particular end game scenario, it was not a complete chess player.

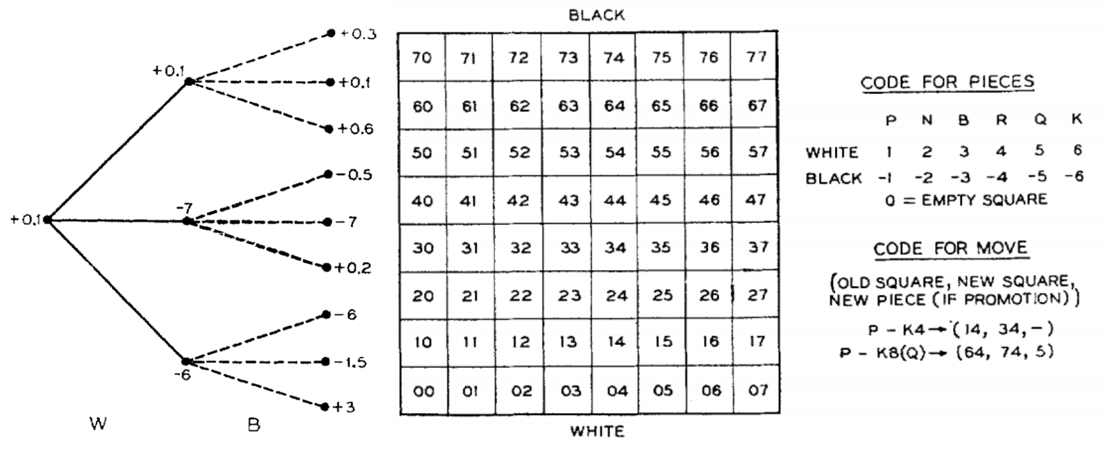
2.2.2 Shannon's Chess Engine Definition

In 1950 mathematician and engineer Claude E. Shannon described a routine for general purpose computers capable of playing a tolerable good chess game [28]. Shannon suggests an *evaluation function* $f(P)$ evaluating a position P to determine a won, drawn or lost position. Theoretically a machine could evaluate every possible move for a position and then every move by the opponent and so on. Creating a move tree that ends in every variation possible. By looking at the end results and step back to the current position, it is possible to determine if the position is a win, draw, or loss. Hence, making it a perfect chess player. But as Shannon also remarks, a position offers on average 30 legal moves. A full move consists of 10^3 possible moves. By assuming a conservative average game length of 40 moves, the move tree would have 10^{120} variations to calculate, making this endeavor quite impossible as to calculate the first move in the game can take 10^{90} years, using a machine which calculates one variation per micro-microsecond.

An evaluation function $f(P)$ is an approximation and cannot be complete, as it is based on generalizations of certain principles from empirical data. The suggested principles include the number and kind of black and white pieces (*material*), *pawn structure*, *mobility*, etc. These heuristics are then determined for both players and subtracted. Consequently, a positive evaluation is an advantage for the white player and a negative one is an advantage for the black player. The more the evaluation deviates from 0 the greater the advantage for a player. The evaluation function should only be applied to relatively quiescent positions as evaluating a position where a Queen captured another defended Queen makes little sense, when the remaining Queen will be captured in the counter move and therefore completing the exchange.

Furthermore, a chess program should be able to determine the player's next best move. Not only by applying the evaluation function to the current possible moves and select the best one, but to look ahead in a tree of moves with a computable move depth. Generally, the white player tries to maximize and the black player tries to minimize the evaluation function to determine the best possible move. This results in a procedure of analyzing maximizing every white move and minimizing every black move within the search tree, alternating the two functions for every step. Every variation in this tree is completely evaluated and the variation best for the respective player is chosen. Then the program steps back from the end result to the next playable move and plays it. This leads to playing the best evaluated move by a certain depth and not the best immediate move which loses its advantage in the long term. Figure 2.3 (a) is a simple depiction of the search tree process. If white chooses the upper branch, black can minimize the evaluation to $+0.1$. For the second move it is -7 and the third one -6. White maximizes and therefore chooses the upper branch for its best option. As remarked before, the search tree grows exponentially for every additional move. Therefore Shannon suggests another strategy to only evaluate moves which seem promising. These moves can be selected by heuristics, but determining the right heuristics is a difficult process. He names the former procedure *type-A* and the latter one *type-B*.

The theoretical paper describes a chessboard representation as an eight by eight square list. The squares can be numbered according to Figure 2.3 (b). The squares can take a state between -6 to $+6$. Each piece is a number within that range, negative for black, positive for white and 0 is an empty square. The number λ is $+1$ or -1 indicates



(a) (b)

Figure 2.3: (a) A tree of moves. The first three branches are white moves, the following are black moves [28]. (b) Board representation used by a computer program [28].

White's or Black's turn, respectively. A move is represented by two squares, the original and the final square of the moving piece. A third indicator is added in case of pawn promotion. Hence, the move consists of (a, b, c). a and b are squares and c the promotion.

The principles of evaluation functions, search trees and board representations introduced by Shannon still hold up to modern chess engine standards, even though the implementations have drastically improved over the following decades.

2.2.3 Chess Engines Pursuit to Gain Strength

After decades of further advancing algorithms and evaluation functions for chess programs led to the inception of chess programs such as *Kaissa* or *Chess* in the 1970s. These programs ran on supercomputers and were able to compete in computer chess tournaments, but not on a grand master level. Nevertheless, in 1997 the ultimate breakthrough for computer chess occurred when the chess computer *Deep Blue* [14] defeated the—at the time undisputed—russian grand master Garry Kasparov. Significant advances in computing power, as well as improved implementations of Shannon's approach to solving computer chess, enabled this milestone.

Since then chess engines—as envisioned by Shannon—asserted their strength and can compete on human grand master level. Notable examples are the open-source engine *Stockfish*[1] which is currently one of the world's strongest traditional chess engines, the commercial chess engine *Kommodo*[2] and *Houdini*[3]—also being commercial, etc. These three engines have similar strengths but vary in their implementation, e.g., *Stockfish* relying on search depth and *Kommodo* on a more advanced evaluation function.

In 2017 the *DeepMind* team devised a completely new approach for a computer chess program. *AlphaZero* [30] uses reinforcement learning from chess games as well as self-

[1]https://stockfishchess.org/
[2]https://komodochess.com/
[3]http://www.cruxis.com/chess/houdini.htm

play. It completely refrains from using heuristics and finely tuned weights—as utilized by traditional chess engines—and only provides the program with the necessary rule set for chess. The deep neural network used in *AlphaZero* learns move probabilities by self play and utilizes them for its search. Instead of using the alpha-beta search used in traditional chess engines, it works with a *Monte Carlo tree search* (MCTS) to devise the next move. This new approach led to the defeat of *Stockfish*, validating this new method as a very potent chess program. The open-source chess engine *Leela Chess Zero*[4] (lc0) is based on the works of *AlphaZero* and has a comparable strength to *Stockfish*.

2.3 Chess Engine Principles

The following algorithms and principles are commonly used in traditional chess engines. Every chess engine has its own implementations and variations of these concepts, but the described principles are utilized in almost every major chess engine, apart from engines based on neural networks.

2.3.1 Board Representation

The array of squares board representation suggested by Shannon—as described in Section 2.2.2—has the advantage of being very light on memory usage. Nevertheless, the concrete implementation of an eight by eight array was never used as a ten by twelve array solved the problems of knights moving out of bounds [11]. Each element in the array represents a square on the board or the squares off limits. These squares each hold a designated value for a white or black piece.

Another approach is the usage of bitboards. A bitboard consists of multiple 64 bit words—each representing a piece type [1, Ch. 3]. A single square is assigned to one bit in a 64 bit long word, thus storing all 64 squares of a chessboard. A word is meant for a concrete piece type, such as white pawns, black pawns, white bishop, etc. Setting a bit to 1 means that the square is occupied by the piece type specified by the word and 0 if that piece type is not situated on that particular square. Therefore, a bitboard consists of at least twelve words, one word for each piece type and their respective color. Additional words can be defined such as all white pieces and all black pieces. The advantage compared to Shannon's approach is the low count of instructions needed to compute complex situations. This is accomplished by Boolean operations such as logical *and*, logical *or* or *not* on the bit maps [11, Ch. 3]. To compute the possible captures for a white knight, a bitmap containing every position the knight on that particular square could move to is fetched. Secondly, a bitmap containing all white piece positions is retrieved and *negates* that map, so that when both maps are combined with an *and* instruction, only the squares remain which the white knight could move to. To finally calculate the possible captures, a bitmap for all black pieces is fetched and the two bitmaps are combined with an *and* instruction, resulting in a bitmap containing all possible captures for the particular white knight. The operations required three fetches from memory, one *not* and two *and* instructions.

Bitboards require more memory than the square array, but compute complex operations using fewer instructions, thus improving the performance of chess engines.

[4]https://lczero.org/

2.3.2 Static Evaluation Function

An evaluation function [11] is used in a look-ahead procedure to determine a variations worth and subsequently choose the next best move. The evaluation function is mostly applied in quiescent positions, meaning the position is relatively stable and exchanges have been completed. The evaluation itself is a comparison of a multitude of factors between the black and white players. Each factor influences the evaluation's accuracy, as well as its performance. Chess engines either have a simple and fast evaluation to facilitate a deep look-ahead search or a complex evaluation and a shallow search. Both—the evaluation function and look-ahead procedure—have to be in a delicate balance to create a well performing chess engine. Over the last decades a long list of suitable factors for chess engines has accumulated. These factors are e.g., material balance, mobility, pawn structure, control of center squares, king safety, passed pawns, rooks on open files, pins [11, 13, 28]. Chess engines can solely rely on a fast material balance check or incorporate as many factors as possible. These factors have different implementations and weights associated with them, as the implementations rely on assertions.

2.3.3 Look-Ahead Procedure

The next best move to make can be determined by evaluating every move to a defined depth, as described in Section 2.2.2. This creates a tree composed of moves and every level alternating between white and black moves. At each node the algorithm chooses the best move for the moving player by applying an evaluation function. An advantage for the white player can be defined to be a large number and a low number for a black player's advantage. Consequently, a white player's move needs to be maximized and a black player's move minimized. This leads the algorithm to alternate maximize and minimize for every node it traverses deeper into the tree, which is why the algorithm is called *minimax* [11]. For example for a white move with a four ply deep analysis the top down order would be maximize, minimize, maximize, minimize. There are two different approaches to traversing the tree: depth-first and breadth-first. The former explores a branch to is lowest node, before moving to the next one. The latter approach first explores every node on the first level, before moving to the second and then third level, etc. Normally a depth-first approach is chosen.

Figure 2.4 demonstrates a possible game tree four plies deep. Normally, a node would have around 30 moves available, but this example uses one to three sub-nodes. White moves are symbolized as squares and black moves are circles. First the tree is traversed down to the lowest level following the path of the following nodes: 1, 2, 3, 4. Node 5 and 6 have an evaluation of both +1. To select which move is best suitable both nodes are compared. As Node 4 is a black move it tries to minimize the evaluation, resulting in +1. Next up is node 7 where 0 and +3 are minimized, choosing node 8. The comparison between node 4 and 7 is maximized as node 3 is a white move. White chooses node 4 as it yields the higher evaluation of +1. This procedure is strictly followed until every node is evaluated and compared to each other from the bottom up and one move remains. In the end node 1 maximizes node 2 and 22 with the values +1 and −4, thus choosing node 2. The next move is `P-K4` (`d4` in SAN notation).

As Shannon [28] mentioned, the growth of the tree is exponential and therefore cannot be completely evaluated. An optimization is needed to evaluate a tree to a

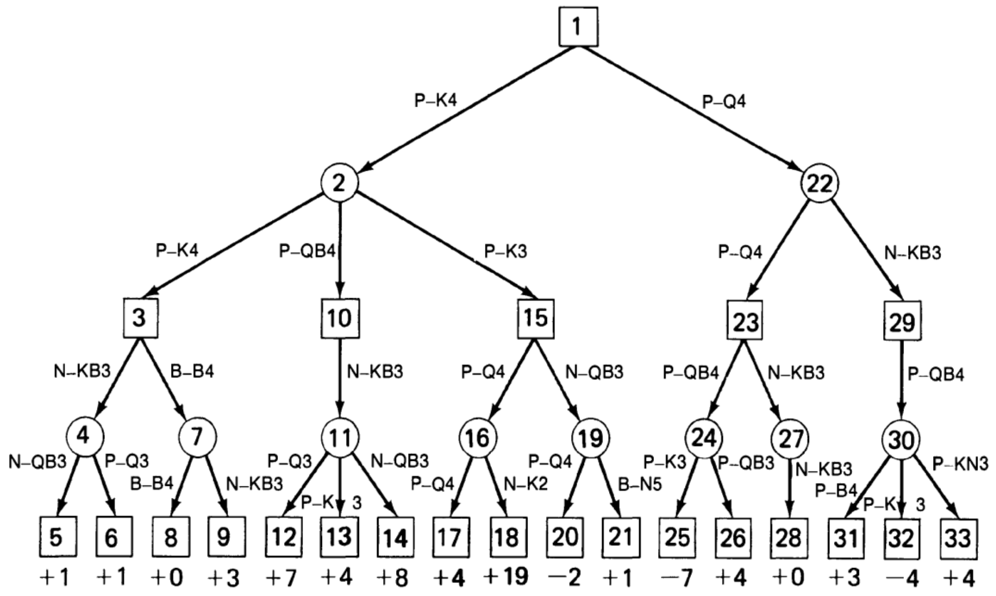


Figure 2.4: Sample game tree for the opening position in chess [11].

certain depth, which yields appropriate results. α-β pruning offers a solution to this problem. α as minimum value for the maximizing player and β as maximum value for the minimizing player. Following the example of Figure 2.4 we evaluate the tree depth-first. By first looking at node 5, 6 and then minimizing at node 4 we get again an evaluation of +1. Then proceeding to node 8 we get a 0 evaluation. The evaluation of node 9 is not required, as node 3 maximizes and it chooses node 4 over node 7, because it has the higher evaluation. This is because node 7 minimizes and chooses at least an evaluation that is lower than +1 and node 3 maximizes therefore choosing node 4. Hence, it is possible to stop evaluating the hole tree and prune branches, while not altering the result of the procedure. In a real scenario with 30 possible moves at every node huge parts of the tree do not need to be evaluated, therefore enabling a deeper search.

2.3.4 Opening Book

Chess openings constitute a problem for chess engines, as pieces have not been removed from the game and are far from being removed any time soon. This initial phase spans a very broad and deep search tree. The program has to make immense efforts to compute good moves. To avoid this a database of standard positions is composed and evaluated. The chess engine can rely on these positions and immediately respond with a suitable next move. The problem of the approach is that a chess game eventually deviates from the positions stored in the opening book. Thus, the chess engine falls back to its look-ahead search. Often times this switch makes the engine perform worse for a couple of moves, as the opening book and the evaluation function favour different factors. The chess engine then adjusts the pieces to match its play style and consequently losing tempo or advantages. This problem can be countered by constructing an opening book matching the engine's style.

2.3.5 End Game

Applying *minimax* search onto end games seems inadequate as the moves required to win a game are very specific and may require a plan of 20 moves to accomplish a win. To mitigate this problem, end games can be precalculated and stored up to a certain number of remaining pieces on the chessboard. An end game tablebase [2, 56] is generated by backward analyzing the search tree from a checkmating position. Then these end game positions can be searched within a tablebase to retrieve the best move, without searching for it.

2.4 Stockfish

Stockfish[5] [41] is an open-source engine in the traditional sense, which is a fork of the chess engine *Glaurung.* [6] *Stockfish* introduced significant improvements to *Glaurung*, which led to combining forces of Romstad and Costalba by solely focusing on the development of *Stockfish.* Currently, *Stockfish* is one of the strongest traditional chess engines [43]. It relies on the principles discussed in Section 2.3, but their implementations are more advanced. The chessboard is represented by utilizing bitboards. Its evaluation function includes factors such as material, imbalances, mobility, threats, passed pawns, space control, initiative, king safety, etc. The search uses a minmax procedure with α-β pruning and other optimizations. The engine is written in *C++*, which makes it run on a broad range of operating systems, e.g., *Windows*, *Linux*, *macOS*. *Stockfish* supports the UCI protocol, which enables the communication between chess engines and chess user interfaces. The advanced evaluation and search, as well as recognition for being one of the strongest chess engines available, make *Stockfish* a well suited pick for being utilized in the master thesis.

2.5 Universal Chess Interface

The *Universal Chess Interface* [54] (UCI) was developed in 2004 to enable communications between chess engines and graphical user interfaces (GUI). GUIs can send individual moves in LAN or FEN notation. User interfaces can define a maximum threshold for the move calculation, e.g. depth in plies, move time, number of nodes, mate in a number of moves. After the move calculation the engine sends back the calculated next move, the evaluated score, the explored depth, time needed, nodes traversed, etc. UCI is broadly adopted by most chess engines, as it makes GUIs and chess engines interchangeable and not dependent on each other.

[5] https://stockfishchess.org/

[6] *Glaurung* was created by Tord Romstad in 2004 and Marco Costalba forked *Glaurung* 2.1 to create *Stockfish.*

2.6 Chess Notations

The *Standard Algebraic Notation* [45] (SAN) is used to describe the moves in a chess game. The rows, called ranks, are numbered from 1 to 8 and the columns, called files, are labeled from `a` to `h`. A move consists of a move number, which is only incremented when both white and black have taken their turn, the two half moves for white and black, which in turn are comprised of a letter for the piece moved and a destination square. The piece letters are `K` for king, `Q` for queen, `R` for rook, `B` for bishop and `N` for knight. The pawn is not a piece and therefore does not does not have its own letter. When a pawn is moved the letter is just omitted. For example the first move for white and black could look like this: `1. Nc3 f5`. The moves mean that white has moved their knight from `b2` to `c3` and black has moved their pawn from `f7` to `f5`. A capture is indicated using an `x`, i.e., `Nxc3`—knight captures a piece or pawn on the square `c3`. Promotion of pawns appends the letter of the piece the pawn is promoted to, i.e., d8Q. Kingside castling and queenside castling are indicated as `0-0` and `0-0-0`, respectively. Checks add `+` and checkmates `++` or `#` at the end of a move. An extended notation to SAN is the *Long Algebraik Notation* (LAN). It adds the origin square to the notation, i.e., `Nb2-c3`.

An alternative to SAN is the *Forsyth-Edwards Notation* [53] (FEN). In contrast to SAN's minimal notation, FEN represents the whole chess board using a line of text. The lettering for the pieces is almost the same, with the difference of pawns written as `P`. White pawns as uppercase and black pawns as lowercase letters. A rank on the chessboard consists of eight letters with the exception that consecutive empty squares are written as a number. Ranks are separated using a `/` symbol. This notation can be processed easily as a move always stores the whole board. An example of FEN is:

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1.
```

The FEN notation is easy to use to define positions other than the standard position.

The *Universal Chess Interface* (UCI) protocol uses yet another version of LAN. It refrains from using letters for pieces and just uses the their origin and destination squares. An example `e2e4`. UCI is described in Section 2.5.

The *Portable Game Notation* [53] (PGN) is used for computer programs to process and record chess games. It uses SAN to describe the moves. Most chess games are available in the PGN format. Making it very useful when working with chess game data. Program 2.1 shows a chess game in the PGN format.

2.7 Centipawn

Pawns are a unit of relative value of pieces to the pawn. Pawns can express an advantage of a player over the other player. To calculate an advantage, the pieces have been assigned to de facto standard values, even though countless derivations emerged over time. The values for queen, rook, bishop, knight and pawn are 9, 5, 3, 3, 1, respectively [28]. Modern chess engines make use of *Centipawn*—1/100 of a pawn. This allows a more granular evaluation by the engines.

Program 2.1: Bobby Fischer and Boris Spassky draw in round 29 of their match in 1992 [53].

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 a6 4. Ba4 Nf6 5. O-O Be7
6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8 10. d4 Nbd7
11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15. Nb1 h6
16. Bh4 c5 17. dxe5 Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6
21. Nc4 Nxc4 22. Bxc4 Nb6 23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+
26. Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5 hxg5 29. b3 Ke6 30. a3 Kd6
31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2 Bf5 35. Ra7 g6
36. Ra6+ Kc5 37. Ke1 Nf4 38. g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5
41. Ra6 Nf2 42. g4 Bd3 43. Re6 1/2-1/2
```

2.8 Score

Apart from the determined next move, a chess engine such as *Stockfish* returns a score after the evaluation was completed. The score usually expresses an advantage of a player in Centipawn. A positive score evaluation is usually an advantage for the white player—a negative evaluation for black. For example a +300 centipawn score is an advantage of a bishop or three pawns for the white player.

2.9 Elo Rating

The *Elo* rating [4, pp. 329–330]—named after its inventor Arpad Elo—is a system to determine a player's skill level, based on the games they played to date. The rating itself is numerical. It gives more recent matches more weight in the calculation. Two players with the same rating are expected to score equal wins and losses. The chance of scoring against another player is computed by the difference of the two ratings.

2.10 Summary

The chess principles described do not represent a complete picture of the tactics, strategies and move principles involved, but rather give the reader the essential understanding of those principles used in later chapters, such as Chapter 4. The historic overview of chess engines highlights the most significant developments in the pursuit of creating an artificial chess player, which is capable of having a human-like behaviour. The theoretical paper written by Shannon already depicts the principles used in today's chess engines, although more than 40 years should pass before *Deep Blue* is able to defeat the world champion. Shannon among many others clearly demonstrates the effort and foresight that was put into finding the right algorithms, well before the necessary processing power was available. Surely, those algorithms have been advanced significantly and are far more capable than before, but those principles still hold true. The chess engine's principles, *Stockfish*, UCI, SAN, centipawn, score and *Elo* rating are utilized in the later chapters. Next, Chapter 3 presents basics for adaptive soundtracks, which are the other major topic area for this thesis.

Chapter 3

Adaptive Soundtracks

Adaptive soundtracks are continuous audio streams manipulated by human interaction or their reaction to the environment, e.g., a video game's soundtrack reacts to a player's actions, a real time composition of music while performing it simultaneously, a sonification of data as the input changes over time. The underlying principle of adaptive audio is to generate music or sounds by adapting and reflecting on their inputs. This chapter explores the state of the art for audio programming languages and audio tools usable for audio synthesis, a number of approaches to compose music for the audio synthesis, as well as projects that utilize these approaches for audio synthesis. At the end of the chapter the related work—which also serves as inspiration—for the thesis is discussed. *Reunion* by John Cage can be highlighted as it is one of the first music performances that try to make a chess game sound.

3.1 Audio Programming Languages

Since the development of *Music I* [17]—one of the first programming languages for computer music—in 1957, it and its successors have laid the foundation for decades of research in audio synthesis. The open-source audio programming language *Csound*[1] [57, 62] is a direct descendent of the *Music N* program family. *Csound* uses *unit generators* [17], also called *opcodes*, which are the fundamental building blocks for audio synthesis, to generate sounds. Basic unit generators include oscillator, add, multiply and output [17]. The *oscillator* is the basic element for creating sounds. Oscillators consist of three inputs: the stored function, amplitude and frequency. The *stored function* defines the waveform. The *amplitude* controls the loudness of the sound. The *frequency* sets the pitch of the sound. Combining these three inputs creates a sound with a specific loudness, pitch and texture. *Add* and *multiply* combine two inputs by adding and multiplying inputs, respectively. *Output* unit generators are used to send the resulting signal through a digital to analog converter to speakers. These and other unit generators make it possible to create sound effects such as chorus, reverb, vibrato, etc [57]. Thus, enabling the user to depict real world sounds, such as instruments or synthesize new sounds completely averted from the real world. *Csound* splits a program into two parts: an orchestra and a score [57]. An orchestra includes all instruments needed for the mu-

[1] https://csound.com/

sic. The instruments are created by defining, modifying and combining unit generators to achieve the desired sound. The instruments can subsequently be utilized in a score to create the actual music. The score depicts the actual written notes and rhythms to create melodies and a music piece as a whole. After over 30 years of active development, the language is primarily used in a real-time context, although it was not intended for that. *Csound* is written in *C*, nevertheless, its API supports a number of programming languages such as *Python* and *Java* [57]. Many of the later developed audio programming languages utilize the principles that originated in the *Music N* program family.

RTcmix[2] [55] is an open-source real-time digital signal processing and synthesis language, written in *C/C++*. Heavily inspired by *Music N* programs, *RTcmix* utilizes similar concepts such as, unit generators, instruments and scores. *RTcmix* uses a *C*-like syntax to enable the usage of conditional statements and loops. The language can also be used in *Perl* or *Python*.

Contrary to *Csound*, the functional programming language *Nyquist*[3] [10] refrains from the use of the orchestra and score concept. *Nyquist* combines both into one system. Consequently, *Nyquist* programs are flexible and easy to use. *Nyquist* programs can be adapted on-the-fly, which makes them interactive for the user.

SuperCollider[4] [18] is an open-source object-oriented language similar to *Smalltalk*. It lends concepts such as unit generators from *Music N* programs. *SuperCollider* is designed to be used for live improvisations and as a result of this intention it excludes the orchestra and score concept in exchange for improved flexibility. The synthesis engine of *SuperCollider* is a server that runs independently from the *SuperCollider* programming language. Both components communicate using the *Open Sound Control* (OSC) protocol [49]. Hence, the *SuperCollider* server can be controlled by any program.

ChucK[5] [34, 35] is an open-source concurrent audio programming language for real-time audio synthesis, composition and performance. It can be programmed on-the-fly similarly to *Nyquist*. *ChucK* makes use of unit generators, similar to *Music N* programs. The language introduces synchronized threads, called *shreds*, to enable the parallel execution of multiple programs based on a shared time.

In contrast to the previously mentioned text-based audio programming languages, *Max*[6] [26] is a graphical programming environment for developing real-time music software applications. *Max* is intended for live performances by musicians. Therefore, its concepts are highly abstracted. *Max* can be considered to be an descendent of the *Music N* programs, as it defines patches that function similar to unit generators. These patches are constructed by connecting boxes together using lines in a graphical user interface, as opposed to lines of code in other music languages.

Open Sound Control[7] [37] (OSC) is a commonly used protocol for communication from and to sound synthesizers. The following synthesizers support OSC: *Csound*, *RTcmix*, *Nyquist*, *SuperCollider*, *ChucK* and *Max*.

---

[2] http://rtcmix.org/

[3] https://www.cs.cmu.edu/~music/nyquist/

[4] https://supercollider.github.io/

[5] https://chuck.cs.princeton.edu/

[6] https://cycling74.com/

[7] http://opensoundcontrol.org/introduction-osc

3.2 FMOD

FMOD Studio[8] [47] is a commercial audio engine designed for arranging adaptive soundtracks mostly for video games. The soundtracks are constructed in a graphical user interface. Compared to the previously mentioned audio programming languages in Section 3.1, *FMOD* refrains from generating its own audio, but rather relies on the usage of audio tracks. In *FMOD Studio events* are used to define soundtracks, a detailed description follows. Events can be combined and built to *banks*, in which the soundtrack is stored as a whole. Every audio asset and every event that the—in *FMOD Studio* defined—soundtrack consists of are incorporated into the bank. Banks are exposed through an API to game engines such as *Unity* or *Unreal Engine*. Additionally, *FMOD* provides an *FMOD Studio* API for triggering and manipulating the predefined events and a *FMOD Studio* Low Level API for controlling simple sounds within a programming language [46]. Both APIs—*FMOD Studio* API and *FMOD Studio* Low Level API—can be consumed in *C++* programs and work in real-time.

Events arrange audio assets in the desired configuration to create sound. These assets can range from short samples to long and complex music tracks. Assets are integrated into events as instruments by adding them to audio tracks. *Audio tracks* can assign their output to either the master track or another audio track, which is useful for combining multiple audio tracks to add the same behaviour and effects. The audio tracks build the backbone to an event in *FMOD*, as these audio tracks are the output to the loudspeaker. Audio tracks can be manipulated by chaining multiple effects in a row. *FMOD Studio* already provides a multitude of effects, such as Chorus, Compressor, Distortion, Delay, Gain, Multiband Equalizer, Pitch Shifter, Reverb, etc. The effects can be automated which makes them controllable by parameters. *Parameters* play an essential role to make adaptive soundtracks, as they are one of the few factors which can be manipulated by the API. Parameters are decimal numbers that operate in a user-defined number range. In the context of sound effects the parameters assigned to the different effects control the effect's extent. A two-dimensional curve is applied to define the effect's extent corresponding to the parameter's value range. For example, parameters are able to control the volume of an audio track by a parameter with a range from 0.0 to 1.0 and a linear curve which decreases the volume as the parameter continuously increases its value from 0.0 to 1.0. Hence, the effect would be lowering the audio track's volume as the parameter's value rises.

FMOD uses a timeline to organize instruments and audio tracks in the soundtrack. The timeline is played through using a playback position. Audio tracks can be segmented into sections by using *destination markers*. A *transition marker* leads the playback position to jump to a different predefined destination marker. *Loop regions* let the playback position jump to the beginning of the loop region when reaching its end. Therefore, loop regions repeat a customizable area of audio tracks. Another important usage of parameters is the control of markers and regions on the timeline. Markers and regions can be activated and deactivated when the applied parameters reach their predefined value ranges. This concept is similar to conditionals in programming languages, as both cases only arise when its criteria are met. This enables the soundtrack to jump and loop

[8] https://www.fmod.com/

according to the parameters, which again are controllable through the API.

3.3 Choosing the Appropriate Audio Program

The extensive research behind audio synthesis over the decades led to the creation of capable programming languages. Many of which are open-source, actively developed and offer an API, such as *Csound*, *RTcmix*, *Nyquist*, *SuperCollider* and *ChucK*. *Max* offers a GUI, but is closed-source and there are costs involved. *RTcmix*'s lack of documentation [55] makes it difficult to acquire the required knowledge. *Csound*, *SuperCollider* and *ChucK* are good options as all three have real-time capabilities, are well-documented, open-source and being further developed. On the other hand, *FMOD* is a commercial, closed-source, actively developed and through an API exposable sound engine, which on top offers a GUI. *FMOD* does not charge money for non-commercial projects. Due to the fact of its relatively intuitive design approach through their user interface, the use of pre-existing audio assets and free usage for non-commercial projects make *FMOD* stand out. On top of that, *FMOD* allows quick prototyping of adaptive soundtrack which is necessary for this thesis's project as multiple sound designs are tested. Consequently, *FMOD* is chosen for developing the sound component of this thesis.

3.4 Controlling Adaptive Soundtracks

In the previous sections audio programming languages and tools are explored, as well as their techniques to create and manipulate sounds. Either the sounds are generated by using unit generators or predefined concrete sound samples. Yet, the sounds need to be arranged into a sequence of sounds or notes to create a musical piece. The following paragraphs explore the different ways for music to be created:

- A *score* is a traditional piece of music that defines notes and rhythm to be played throughout the piece. The notes define the pitch, loudness and length and they can be played in a sequence or simultaneously. The notes then lead to rhythms, chords and melodies within the musical piece.
- The broad field of *algorithmic composition* [22] relies on algorithms—a strict sequence of instructions—to generate the musical structure. The algorithm receives a set of input values and based on those values the algorithm determines the occurrence of specific notes. For example algorithms can use a simple set of rules which leads to complex sounds over time, or even imitate specific music genres. Often times algorithmic composition uses probabilities to introduce variation to the generation process. The following list explores a number of different approaches to algorithmic composition:
  - *Markov models* determine the next state on the bases of their current state and a set of probabilities. Therefore, Markov models disregard the previous states and rely solely on the current state to calculate the next state. This allows the imitation of musical genres, as the current note defines the next note to be played.
  - *Generative grammars* consists of a set of rules which generate the content

based on the rules. The grammar provides rhythmic as well as tonal structures. Generative grammars can be used to imitate specific styles of music.

  - *Transition networks* consist of interconnected nodes. The networks can follow a certain path within the network and jump to other sections of the network when certain conditions are fulfilled. This leads to the creation of music similar to that of generative grammars.
  - *Chaos theory* describes complex systems with unpredictable or chaotic outcome. Chaos theory introduces the usage of L-systems as well as fractals which allows the usage of fractional noise.
  - *Genetic algorithms* evolve over time based on specified criteria. Applied to music composition the music changes continuously and does not stay the same.
  - *Cellular automata* consist of an n-dimensional grid, in which the cells can assume a state. The cell states are defined by simple rules which alter the cells over time. These simple rules can lead to complex compositions.
  - *Artificial Intelligence* (AI) includes a broad field of algorithms that aims to learn, solve problems and make decisions in a human way. Machine learning is an approach to train models on large data sets for a very specific problem without instructions provided on how to solve them. Machine learning is suitable to reproduce authentic music pieces that mimic the data set it is trained upon.

- *Data-driven composition* [44] uses data to control the sequence of instructions defined in a program. The program maps sounds and effects to the corresponding data, which is mostly structured as lines or tables. The program retrieves the data sequentially, which creates a loop that reacts to the data accordingly. The data adopts the role of a score and the program that of the orchestra.
- *Interactive composition* [6] or *real-time composition* utilizes a system of functions that is altered by human interaction over time. Interactive composition either adjusts the function parameters to influence the musical outcome [6] or fundamentally changes and replaces the functions by applying new algorithms or instruments to the synthesis as the music is performed [8]. The latter is called *live coding* and enables the programmers to add, modify and remove code, while the program is executed [8].

To demonstrate the capabilities of algorithmic, data-driven and interactive composition, three projects utilizing those composition approaches are described.

Application to Polyphonic Music Generation and Transcription[9] by Boulanger-Lewandowski, Bengio and Vincent creates algorithmic compositions that are capable of generating music based on a *recurring neural network* (RNN) [3]. The project compares multiple music data sets, as well as varying implementations for RNNs. The models are trained on several music data sets, such as classical piano, folk tunes, or chorales from Bach. The most promising models are capable of producing polyphonic music, in which chords and complex melodies are played. The musical output is generated as a piano-roll representation, which uses time stamps and every time stamp has a binary value for

[9] http://www-etud.iro.umontreal.ca/~boulanni/icml2012

whole note range. The result are authentic and pleasing polyphonic music sequences.

The song *Two Trains*[10] [48] by Brian Foo applies a data-driven approach to generate a song based on the median household income of the area surrounding the New York City subway line 2 and its stations. The song follows a train driving from Brooklyn through Manhattan to the Bronx. The dynamics of the song are bound to the income of the area, as the income increases so does the quantity of instruments and their loudness. This results in the usage of three instruments in the poorest area and up to 30 in the wealthiest areas. The project uses *Python* to calculate the distance between stations, assigns instruments to stations and generates a sequence of sounds. This sound sequence is sent to *ChucK* for synthesis. The result is a song that transports the mean income of an area through the medium of music.

The group *Slub*[11] [8] is a live coding music duo that creates techno music by writing and adapting their programs while performing. The two group members McLean and Ward have their own approaches to creating music, while McLean mostly writes his code in a command line, Ward depends on his own music interfaces. The interfaces control their own music programs, which explore musical ideas ranging from chordal progressions to a musical version of dancing people. Despite the music being created by two different people on their laptops, the music matches as both inputs are synchronized over a network protocol. Lastly, the synchronized music inputs are sent to a synthesizer which outputs the music. The result is live performed electronic music that is ever changing as the two musicians alter the music code and its parameters over time.

The presented projects are capable of creating authentic music that can be pleasant to the ears, depending on the listeners taste. Boulanger-Lewandowski, Bengio and Vincent created a RNN that learns to create and mimic the music provided by the data set without human intervention. Although humans did not have any say on how the music is created, the RNN generates authentic music. *Two Trains* maps specific data to the audio synthesis and therefore does not have the same objective as the previous project—an algorithm creating human-like music. *Two Trains* focuses on the context of the data and tries to transport the data's semantics to the listeners ears through sound. Contrary to the previously described projects, *Slub* does not create static music, but rather creates techno music on the fly by programming as the music is performed to an audience. It uses live programming as an instrument to play.

3.5 Audio Synthesis for Chess

Up to this point, the techniques of general audio synthesis, the concrete approaches for controlling synthesizers and their associated projects have been discussed. To complete the scope of this analysis of adaptive soundtracks, the subject matter of chess needs to be included.

3.5.1 Reunion

The music performance *Reunion* [9] from 1968 conceived by John Cage combines a game of chess with the ambience of music. *Reunion* was performed in the Ryerson Theatre

[10] https://datadrivendj.com/tracks/subway/
[11] http://slub.org/

in Toronto. The aim of *Reunion* was to build "an electronic chessboard that would select and spatially distribute sounds around a concert audience as a game unfolded" [9]. To implement such a chessboard, Lowell Cross mounted 64 photoresistors—one per square—inside a chessboard. The photoresistors react to the exposure to light and allow the passing of a signal depending on their default configuration. While the four ranks (1, 2, 7 and 8) covered by the initial position of pieces are turned off when the pieces are placed on them, the other ranks (4, 5, 6 and 7) are turned off when the squares are empty. The movement of pieces on the chessboard changes the state of the photoresistors to either turn the different music inputs on or off. This configuration ensures that a game starts silent. 16 music inputs are coupled to eight outputs through semi-random combinations of inputs and outputs on the chessboard. 60 of the 128 possible combinations can be utilized, which leads every input to be coupled with exactly four of the eight available outputs. The electronic music inputs are composed by David Behrman, Gordon Mumma, David Tudar and Lowell Cross. The outputs are eight loudspeakers distributed in the theatre which should give the impression of moving sound.

The first game of the night was Duchamp against Cage, as seen in Figure 3.1. Duchamp removed his king side knight from the game to even the odds for Cage—Duchamp was a chess master and Cage his pupil. Nevertheless, Duchamp won decisively in less than half an hour. The second game was played by Cage and Duchamp's wife and lasted for hours. Some moves in the two games created a panning sound effect when a piece was moved over a series of squares. This allowed music inputs to wander, i.e., from the back of the theatre to the front. While a hand moved a piece, this hovering hand additionally influenced the photoresistors activity. As the games progressed and pieces were removed from the board, the sound scenery also toned down. *Reunion* was not received well by critics, as the first game was too short to unfold the sound dynamics and the second game was dragged out for too long and not much happened for the audience to be entertaining.

3.5.2 Reunion2012

Based on Cage's performance, *Reunion2012* [33] modernizes and expands on *Reunion. Reunion2012* again uses a physical chessboard which acts as a controller and eight speakers for sound output. The chessboard incorporates 64 hall effect sensors to recognize magnets within the pieces. The sensors' signals are sent from an *Arduino* board to a computer. The computer runs a chess engine and *Max.* Eight loudspeakers are equally spaced around the chessboard to give surround sound. *Reunion2012* can be broken up to two kinds of performances. A concert version and an interactive version. The former mixes the live music input of eight musicians—four inputs for white and four for black—similar to Cage's *Reunion.* The concert version further develops Cage's approach by determining the number and kind of outputs being played by the movement on the ranks and the piece's color, and the movement on the files to decide on which speakers the music should be heard. The interactive version generates music from samples. It substitutes the musicians input determination process with the application of a number of effects. The piece types are associated with their own leitmotif. These samples get transformed using effects such as speed and pitch. Other effects get applied randomly or by playing a certain move such as castling.

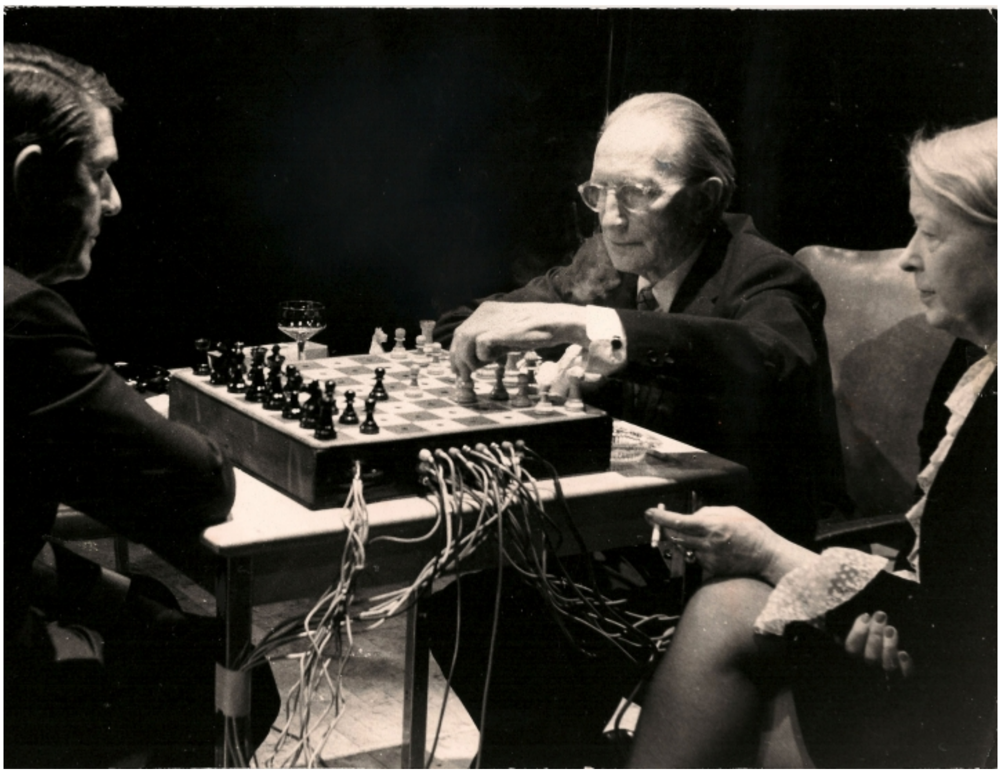


Figure 3.1: Duchamp (white) moves a modified chessboard, while Cage and Duchamp's wife watch [38].

3.5.3 Music for 32 Chess Pieces

Music for 32 Chess Pieces [24] is a software program designed for non-musicians to improvise music by moving chess pieces and adjusting parameters in the program using a GUI. It consists of a game server that handles the game's states and piece relationships. The software can load plugins for mapping parameters (attack or support relationship, movement speed, etc.) to influence the music (duration, tempo, phrasing). These plugins can completely change the generated sound. The game server controls the audio synthesizer such as *ChucK*, *SuperCollider* or *Max* via OSC.

3.6 Summary

Creating sound or music using chess as an input is a rather understudied field. However, *Reunion*, *Reunion2012* and *Music for 32 Chess Pieces* demonstrate how chess can be utilized as an input method. *Reunion* only takes piece movements into account and activates the music input semi-random [9]. *Reunion2012*'s interactive version has a solid architecture, but falls apart when it comes to analyzing the chess game itself, as it only takes piece positions and specific moves into account [33]. *Music for 32 Chess Pieces* on the other hand can make use of the different piece relationships and manipulate the sound accordingly [24].

Modern implementations process chess variables to a certain degree to influence the music. The parameters used by *Music for 32 Chess Pieces* such as attack and support relationship of pieces or player's move speed [24] are quite basic. There are more parameters that can be harnessed for audio synthesis such as: king safety, center control, the development of pieces in the game, threats to pieces, captures, forks, pins, skewers, castling, etc. Mapping these parameters to audio synthesis could change the sound's dynamics dramatically.

There are plenty of options for audio programming languages and tools that can be utilized for creating audio or music for chess games. While *Reunion2012* and *Music for 32 Chess Pieces* offer insights to a modern architecture for audio synthesis for chess, there is little or none focus on how to convey meaning of a chess game through the audio. This offers the opportunity to research the semantics behind chess moves and parameters as well as the transformation to audio. Hence, providing meaningful feedback to the chess players.

Chapter 4

Data-Driven Soundtracks for Chess

This chapter explores the general ideas behind the thesis and defines the requirements the concept has to fulfill. The requirements are a detailed list which describe the intended behavior of chess metrics and principles for the usage in an adaptive soundtrack, as well as the soundtrack itself. Following the requirements is a list of limitations the concept has. These limitations restrict the scope of the thesis, to set boundaries. Lastly, concrete soundtrack designs are discussed by detailing the metrics, the sounds and effects, as well as their interplay.

The concept behind a data-driven soundtrack for chess is to utilize the millions of chess games[1] available for an analysis to define not necessarily a complete depiction of chess through its metrics, but a reasonable set of chess parameters that can provide useful information for inexperienced chess players. Therefore, complex and high level metrics used by advanced players such as piece development or imbalances on the chessboard are dismissed. This narrows the broad set of possible metrics and helps to keep the focus on the task at hand: the development of a soundtrack that supports chess players in assessing the situation in the game. Which means to hint which player has the upper hand in the game or create suspense as the number of threats on the board is increasing as the game progresses. The calculated metrics are subsequently consumed by an adaptive soundtrack. The soundtrack contains predefined events and loops that can be manipulated by the metrics. These manipulations contain additions and subtractions of audio samples, triggering certain sound effects such as distortion or low pass filters as well as changing the intensity of provided examples according to the chess metrics. Listening to the adaptive soundtrack should enable inexperienced chess players to adapt their positions according to the sound. The soundtrack generally hints the player on their performance and does not make suggestions. This way the players have to come up with their own moves, but are assisted to identify their current situation.

4.1 Requirements

The following list defines the requirements for the concept:

[1]https://database.lichess.org/ - The open-source chess server *Lichess* provides online chess matches, analysis as well as learning tools. Additionally, *Lichess* is one of many online sources providing already played chess games in a readable text format.

- **The metrics used represent chess reasonably:** The chosen set of chess metrics does not need to depict chess as a whole, but includes metrics important for communicating basic ideas in chess [61]. Depicting every detail of chess completely in a computer program is not preferable, as new evaluation terms do not necessarily improve the evaluation. Additionally, a complete and exact evaluation is probably not achievable as the rules of chess are inherently too complex [28].
- **Metrics change according to the positions on the chessboard:** The set of metrics used reflect changes on the chessboard. The metrics are to be chosen in a way to capture the changes in the game. Which means, a minimal set of metric should be able to capture a maximum of context on the board.
- **Metrics fit the use case to provide information for inexperienced chess players:** The metrics can be digested from inexperienced players when confronted with the information, the metrics provide.
- **Metrics can be utilised in a real-time scenario:** The metrics require little time for calculation. Performance will be a major factor in future developments of the project, as live evaluations of chess games will come into focus.
- **The soundtrack changes its sound according to the metrics:** A significant change in a metric is reflected by the soundtrack. The tonality, sound effects and their intensity is regulated by the metrics and can either change analog to the metric itself or when passing predefined thresholds. Consequently, the soundtrack only changes when required and in the required intensity.
- **The soundtrack's sounds and sound effects can be distinguished from another, to enable the listener to associate the effect with the changes occurring on the chessboard:** This should enable the listener to learn the meaning of the effects and react accordingly. Using the soundtrack repeatedly should increase the soundtracks meaning and intent to the listener, hence have a learning curve.
- **The soundtrack can recognize and communicate critical situations based on the chess metrics:** The soundtrack utilizes appropriate sounds to transport the intent behind the metric. The listener should hear when they are in a critical situation, making it easier to defend or win a position.
- **The soundtrack creates suspense for the listener:** Critical situations, as mentioned in the previous list item, should also create tension which can make the listener uneasy, without being too provoking.
- **The soundtrack is pleasant to listen to repeatedly, without straining its listeners:** Chess games have a broad range of time available for players. This can range from less than 15 minutes per player in a Blitz to 90 minutes per player plus additional 30 minutes when reaching move 40 in a tournament game [45]. As a result the soundtrack must be adequate and without elements which already annoy the listener after a short time.
- **The soundtrack reacts to the metrics immediately:** This does not mean the soundtrack changes immediately to the listeners ear, but rather that the chess metrics can be communicated to the soundtrack without delay. Changes in tonality or sound effects can intentionally fade in slowly to improve the quality of the soundtrack.

4.2 Limitations

It is not possible to capture the players' intentions and plans, as the program does not have insights into the players' train of thought. Often players try to achieve a advantageous position in a number of moves, which cannot be easily anticipated. The two options are either a search by a chess engine or an opening book. Typically, chess engines evaluate a search tree by searching a certain number of moves ahead, also called search depth, and the most promising tree branch returns the next best move to make [11, pp. 61–65]. Secondly, an opening book stores multiple standard openings [28, p. 273; 11, pp. 77–79]. It contains a limited number of moves considered to be best suited in the early game. The opening book is applied as long as the player plays moves contained in the opening book. The openings can be anticipated, but only if the player has knowledge of the opening. An inexperienced player won't follow standard openings and therefore an opening book will be of little use. Both methods can explore possible moves and variations but not the actual moves played by the player.

A second limitation is to recognize future possible threats. It is difficult to anticipate future threats, as specific movements have to happen for this to occur. This would include looking ahead and evaluating every possible move to a certain depth, not unlike calculating the best move. This could be used when the evaluation of the best move and possible threats is done in the same process, as both would evaluate a search tree. As opposed to when using a chess engine such as *Stockfish* and additionally evaluating possible threats would result in an increase of processing time.

Completely sound metrics are not achievable as assertions and decisions on the calculation process have to be made by humans [28, pp. 261–262]. The metrics are approximations based on the analysis of numerous chess games, resulting in empirical evidence. Therefore these metrics do not hold true for every situation and can be countered by certain examples. A chess engine's evaluation function relies on decisions the author makes and their evaluation for the same position can vary from version to version. The decisions are often derived from multiple publications and forum posts from within the chess community. This makes chess metrics rely on empirical evidence, human consensus and a lot of fine-tuning over time.

Furthermore, sound can only transport a limited number of sounds and effects before it gets an obscure mixture of sounds. As the user should be able to identify the individual sounds and effects used in the soundtrack, it constraints the number of sounds available.

Lastly, the soundtrack is deterministic and sounds the same when repeated for the same game. The sounds are tuned to the metrics used for the soundtrack. Consequently if the exact same position occurs in two different games, the soundtrack will behave similar, with a few exemptions for situational metrics that react to the move itself. The soundtrack will always have the same basic mood and the same effects, but their sequence and combination will vastly differ. This makes the soundtrack recognizable and a learning effect can arise, as the player learns the purpose behind each sound.

4.3 First Soundtrack Design

The first concept for an adaptive soundtrack uses music to accompany chess games. The idea is to assign a distinct music track for each player. Only one of the two tracks is played at a time as it signals which player has the upper hand. A change in the lead is followed by a change to the other music track. The soundtrack loops over predefined regions in the tracks and switches between the regions seamlessly. The regions are defined for different intensity levels in the game. As the game reaches more critical situations, the music adds melodies.

4.3.1 Samples

The two music tracks are provided by the sound engine *FMOD*—described in Section 3.2—itself. The orchestra track has a heavy metal sound, while the electronic track consists out of electronic elements, as the name suggests. Both tracks have a similar structure, tempo, tonality and layers. The audio tracks are written in the key of A minor. The tracks have a 4/4 time signature and a tempo of 135 beats per minute. The tracks consist of 32 measures and can be divided into four eight measures long parts. These parts have a different sound and chord associated with them. The parts are from now on called A, B, C and D and can be described as follows:

- Part A starts neutrally with an A minor chord.
- Part B changes to the C major chord and gets more intense as it changes the rhythm.
- Part C changes from C major to A minor again and functions as a bridge to Part D.
- Part D changes from A minor to E minor and is the most intense of all four Parts.

The tracks also have four layers which add new instruments to the sound. Layer 1 is a base track using the most prominent instrument as well as a little use of drums. While the other layers add percussive elements, bass and additional melodies.

4.3.2 Audio Tracks

The sound design uses two different audio tracks. One is electronic and the other a heavy metal track. Each track consists of four layers, they share the same speed of 135 BPM and can be divided into the same sections. For each of the two tracks three loop regions were defined for low, medium and high intensity. Each loop region is four bars long.

4.3.3 Leading Player

A player is leading when the score is evaluated in their favour. The chess engine *Stockfish* offers an evaluation, as described in Section 2.4, which can be a positive or negative number of centipawn and signal an advantage for the white or black player, respectively. A change in lead is communicated to the players by a change in tune. The soundtrack plays the electronic track for a white leading player and the orchestra track for a black leading player. Hence, the players can easily identify a score evaluation favouring the other player. For controlling the adaptive soundtrack a `Leading` parameter is defined accepting 0.0 for white's advantage, 1.0 for black.

4.3.4 Intensity

An `Intensity` parameter is introduced to communicate the significance of the player's advantage in centipawn resulting from the *Stockfish* evaluation. The intensity is arranged into three categories: low, medium and high intensity. The soundtrack uses parts A, B and D from the samples to signal the intensity and repeats the current part as long as the intensity level is sustained. The intensity is calculated by dividing the absolute score by 1000. Therefore, the intensity level ranges from 0.0 to 1.0, making it necessary to ignore score evaluations above 1000 centipawn. The resulting `Intensity` parameter controls which loop region gets played. It ranges from:

- 0.0 to 0.1 (0 to 100 centipawn) for the low intensity loop using the neutral Part A,
- 0.11 to 0.3 (110 to 300 centipawn) for the medium intensity loop using Part B and
- 0.31 to 1.0 (310 to 1000 centipawn) for the high intensity loop using Part D.

4.3.5 Possible Moves

The possible legal moves available for a player in a turn inform the player on their opportunities on the chess board. A high number of possible moves tells the players that they have relative freedom on choosing their next move. On the other hand a low number of possible moves can signal a dangerous situation as the player has to react to threats on the board. Often this means that the player's king is check and they have to counter the threat immediately. This shows the quality of communicating critical situations even if it doesn't come much to use as players try to avoid these situations in the first place. The soundtrack utilizes this metric using the parameter `Possible Moves`. The parameter defines a threshold for the number of possible moves lower than five. If true, it triggers a low pass filter to dampen the soundtrack. The sound makes the lack of mobility quite apparent to the players.

4.3.6 Is Check

The `Is Check` parameter tests if the player's king is check. If this is the case a heartbeat track gets activated and plays as long as the king is threatened. A checked king is rather obvious to every type of player, but it enriches the soundtrack to reflect a distressing situation.

4.3.7 Move Category

In chess, a move can be described as good, neutral, inaccuracy, mistake and blunder. These categories are determined by comparing the scores of the actual move and the best available move according to a chess engine, also called the best move score difference. The larger the centipawn difference, the graver becomes the mistake. According to *Chess.com* [50, 52] and *Lichess* [51] moves can be defined as follows:

- An inaccuracy is a weaker move than the best available move and can be defined as a 30 centipawn (*Chess.com*) or 50 centipawn (*Lichess*) difference compared to the best move.

- A mistake is a bad move, which affects the position immediately. A move is recognized as a mistake starting with a 90 centipawn (*Chess.com*) or 100 centipawn (*Lichess*) difference.
- A blunder is a bad move which results in a loss of material or the game in one or two moves. The threshold for a blunder is either 200 centipawn (*Chess.com*) or 300 centipawn (*Lichess*).

The thresholds defined by *Lichess* are better suited for the use case of evaluating inexperienced players, since the thresholds are more forgiving.

In the context of the adaptive soundtrack, the `move category` parameter gives an immediate feedback to the players to recognize a mistake being made and its significance. Inaccuracies, mistakes and blunders are signaled right after a piece moved. A short sound effect is triggered which fades out after a few seconds. With increasing significance the volume of the sound effect rises too. This makes blunders easier to make out than inaccuracies.

4.3.8 Attackers Count

An attacker is a piece which could capture one or more pieces from the other player and therefore threaten them. The higher the number of attackers, the greater the number of threatened pieces which leads to an increasingly dangerous situation. The `Attackers Count` parameter calculates how many of the opponent's pieces could capture one of the current player's pieces. If the count is greater than four it activates one of the four original layers from the audio tracks. This audio layer is normally deactivated. As the count rises it gains more volume. The audio layer itself tries to increase the tension in the music.

4.4 Second Soundtrack Design

The second implementation of an adaptive soundtrack is a soundscape, which refrains from using rhythm or melody. It utilizes continuous sound samples by using different synthesizers created with *Logic Pro X*'s plugin *Alchemy*[2]. The soundscape adds and removes single tone samples according to the chess metrics. The samples can be faded in and out and sound effects enrich the soundscape. It aims to be pleasant to the players over longer periods of time, because compared to the first sound design, described in Section 4.3, the soundscape does not use musical elements which wear off or become annoying over time.

4.4.1 Samples

The *Alchemy* plugin for *Logic Pro X* is a synthesizer containing over 3000 sounds. Its sound-generating engines allow the creation of dynamic sounds that slightly change over time. This gives the sounds a refreshing quality. For the soundscape 63 single tone samples were recorded in Logic Pro X using the Alchemy synthesizer. Each sample has a length between 50 and 55 seconds. Three sounds were chosen for the soundscape. *Aquifer*

[2]https://www.apple.com/lae/logic-pro/plugins-and-sounds/

Blockage can be described as an eerie, droning sound, making its listener uneasy. The samples span over one octave ranging from *c3* to *c4*, resulting in 13 samples. *Hemisphere* can be described as neutral and aloof. *Warm Glistening* has a more prominent and narrow sound as opposed to the other sounds which have a broader range of sounds incorporated in them. This makes Warm Glistening perfectly suitable to highlight important factors. Both Hemisphere and Warm Glistening span over two octaves ranging from *c3* to *c5*, resulting in 25 samples. The samples are key agnostic as every half step tone is recorded from the twelve tone equal temperament, which is the basis for every kind of western music.

4.4.2 Base Layers

The base layers consist of five different notes, *d3*, *g3*, *c4*, *f4* and *a#4*. The notes are each a fourth apart, which results in a neutral sound when played simultaneously. One of the notes played continuously throughout the game. But the note can change depending on the score evaluation from *Stockfish*. *C4* is assigned as neutral note for a score evaluation between -99 to $+99$ centipawn. A score ranging from $+100$ to $+299$ centipawn symbolizes a moderate advantage of the white player over the black player and therefore changes to the higher note *f4*. An evaluation greater or equal to $+300$ centipawn signals a significant advantage for the white player and the soundtrack starts to play the even higher note *a#4*. The values for a black advantage are exactly mirrored, meaning a score from -299 to -100 centipawn enables the lower note *g3* and an evaluation lower or equal to -300 centipawn triggers the lowest note *d3*. As long as the evaluation stays in one of those defined ranges the base note does continue to play. This way the players should always be able to tell who is in the lead.

4.4.3 Fluctuation

A significant change in the *Stockfish* evaluation after a move has been made, can signal the impact of a move in the game without interpreting the meaning. This score change can mean the win or loss of important material, a mistake or blunder, a brilliant defense of a major threat, etc. Such outstanding moves should be highlighted. This change is communicated to the players by alternating base tones. This effect fades the base tones in and out as long as the position created by this significant move has not changed. The `fluctuating_score` parameter controls this effect. A neutral score evaluation, according to the definition in Section 4.4.2, switches between the notes *g3*, *c4*, *f4*. A white favouring score switches between the notes *f4* and *a#4*. A black favouring score switches between the notes *d3* and *g3*. The alternating effects create a sense for change without losing the quality of communicating the leading player while three base notes fading in and out.

4.4.4 Unopposed Threats

`Unopposed threats` is a good indicator on the current state of the game. An unopposed threat increases the chance of a piece being captured as the opponent can safely capture the piece without risking their own piece. Adding additional notes for every unopposed threat illustrate the suspension in the game. The notes start with e3 and each additional note is a half step above the previously added note. The effect consists of six layers, making it capable of representing up to six unopposed threats.

4.4.5 Mistake

The effect defined uses similar move categories as described in Section 4.3.7. It activates a low note using the eerie Aquifer Blockage sound. The note is played as an immediate reaction to the player's mistake. The note fades out after a few seconds to give way to other sound elements to be heard. A mistake triggers an *f#3* and a blunder triggers an even lower *c#3*. Inaccuracies are not utilized as they represent a too insignificant centipawn loss for an inexperienced player to offer additional value.

4.4.6 Is Capture

A capture is a significant event in chess, as decreasing the number of chess pieces progresses the game towards the end game. As a capture is not a continuous state but an irregular event, an immediate but short reaction is best suited—similar to mistake's effect from Section 4.4.5. Consequently, capturing a chess piece triggers a distortion effect. The effect lasts a few seconds before fading out. The effect should make the soundtrack become louder and alert the chess player.

4.4.7 Possible Moves

This implementation functions in a similar way to the first implementation described in Section 4.3.5. It keeps track of the possible legal moves for the current player and enables a low-pass filter if the possible moves drop below or equal to five moves. The effect signals the player that they are running out of options to play.

4.4.8 Attack/Defense Relation

The *attack/defense relation* analyzes every piece involved in an attack, including attackers, guards and the attacked piece. The pieces' corresponding centipawn values are used to calculate the relation between attackers and guards in an attack. Then those relations are aggregated per player and the white and black relation are compared to each other. Therefore, the relation gives information on which side has the better attacks on the board in the regards for their offensive and defensive pieces. In the sound design's context the base notes wander from one player to another depending on the attack/defense relation. The base notes are panned to the left channel when the black player is in the lead, the opposite happens when the white player has the advantage in the attack/defense relation. The base notes are played on both channels when the parameter is around zero.

4.5 Summary

The first sound design is used to explore the capabilities of *FMOD* and chess parameters. The usage of premade audio samples enables an immediate implementation of the design without spending valuable time on creating samples. After a demonstration to an audience, a conclusion was reached to discard the melodic and rhythmic samples as they led to a overstimulation of senses and are rather distracting from chess itself. The verdict led to the development of a much more sensible soundtrack design.

This second soundtrack design aims to be much more subtle compared to the first design. The design allows the chess player to focus on the game at hand. The soundtrack hints on the player's performances and signals critical situations to create a more dramatic sound when necessary. The sound design's implementation is documented in Chapter 5 and evaluated in great detail in Chapter 6.

The defined requirements help to evaluate the sound design for the added value it can provide to chess players. The grade of compliance with the requirements will further be discussed in Chapter 6.

Chapter 5

Implementation

The second soundtrack design, as described in Section 4.4, is a product of researching possible chess metrics in relevant lecture and the experimentation and exploration of capabilities within the software implementation. The software implemented for this thesis realizes the goals for enabling an in-depth chess metric analysis, as well as creating an data-driven soundtrack which conveys important information for inexperienced chess players. Hence, the software is split into a number of different programs that are built on each other and have their own objectives.

5.1 Architecture

The two main applications for the realisation of the data-driven soundtrack are the *Chess Analysis Tool* and the *Chess Music Tool*. The *Chess Analysis Tool*[1]—written in *Python*—takes chess games as input, iterates over the game move by move and writes the newly calculated chess metrics into a *comma-separated values* (CSV) file for later processing by the *Chess Music Tool*. PGN files—containing meta data, as well as the moves of a complete chess game—provide the every necessary information for a complete analysis. Each move is analyzed sequentially, beginning with the first move. The *Python* chess library *python-chess*[2] is incorporated for the purpose of analyzing a chess game. *python-chess* offers a chessboard representation, move validation, move generation, communication with chess engines via UCI, as well as other functions useful for an in-depth analysis. The chess engine *Stockfish*—as described in Section 2.4 is utilized to generate an objective score evaluation of a move. Some of the computed information by the program includes: *Stockfish*'s score, the best possible move, mistakes, threats, attackers, guards, material, etc. Lastly the *Chess Analysis Tool* saves the information in a CSV file, as well as generates a *Scalable Vector Graphics* (SVG) file for every chessboard position in the game. The generation of the CSV and SVG files for a game is specifically the task of the *Analysis CSV* program.

Then the *Chess Music Tool* reads the newly generated CSV and SVG files for controlling the soundtrack. The *Chess Music Tool* is written in *C++* and its main purpose is

[1]The *Chess Analysis Tool* is based on Professor Stöckl's initial analysis program for the visualisation of chess games: https://medium.com/@andreasstckl/chessviz-graphs-of-chess-games-7ebd4f85a9b9.

[2]https://python-chess.readthedocs.io/

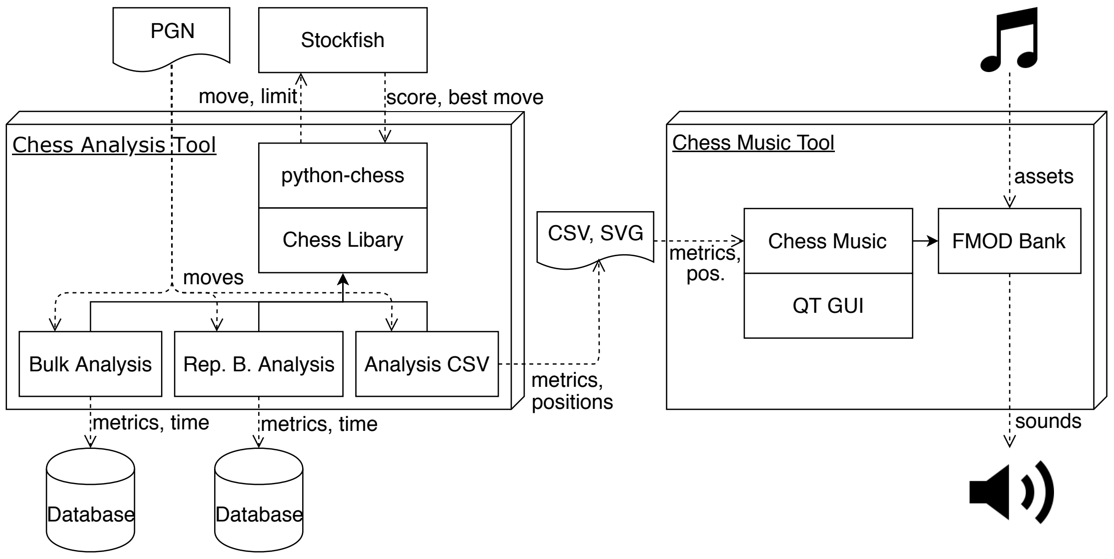


Figure 5.1: Architecture of the implemented software.

to actuate the *FMOD* bank and depict the current situation including the chess parameters in an *Qt*[3] user interface. As described in Section 3.2, *FMOD* exports soundtracks to a standalone bank that includes every asset and their usage within the soundtrack. This bank parameters are exposed through an *C++* API. The data from the generated CSV files are mapped to the corresponding *FMOD* parameters. When browsing through each chess move the soundtrack reacts immediately. The user interface—created with the usage of the frontend framework *Qt*—provides visual context to the soundtrack's reactions. It displays the corresponding SVG which depicts the current chessboard position in the game. Furthermore, the chess metrics from the CSV file are shown in as text in the UI. The user interface provides controls to switch moves back and forth, as well as an autoplay checkbox to continuously draw moves in a fixed interval, until the game ends.

Apart from the two main programs, two other programs are created to enable a chess metric analysis—described in Chapter 6. *Bulk analysis* utilizes a similar approach to analyzing a chess game, but not only for one game but a multitude of chess games. The program takes a single PGN file that stores multiple chess games, analyzes each game and saves their analyzed metrics to a database. The data includes the chess metrics, the time it takes to compute those metrics, as well as the moves associated game data.

Repeated Bulk Analysis is a variation of the *Bulk Analysis* program as it analysis different input parameters for the score evaluation by *Stockfish*. *Stockfish* can take different limiting factors to their evaluation function, such as the maximum *time* to compute the evaluation, the maximum *depth* the search tree is traversed to, the maximum number of tree *nodes* it takes into account, etc. The input parameters for the evaluation are 10 ms, 20 ms, 50 ms, 100 ms, 200 ms, 500 ms, 1000 ms, 2000 ms, as well as 17 depth-levels and 20 depth-levels. Additionally to this broad input parameter analysis, each move is evaluated five times to compute the disparity of the chess engine's score calculation

[3]https://www.qt.io/

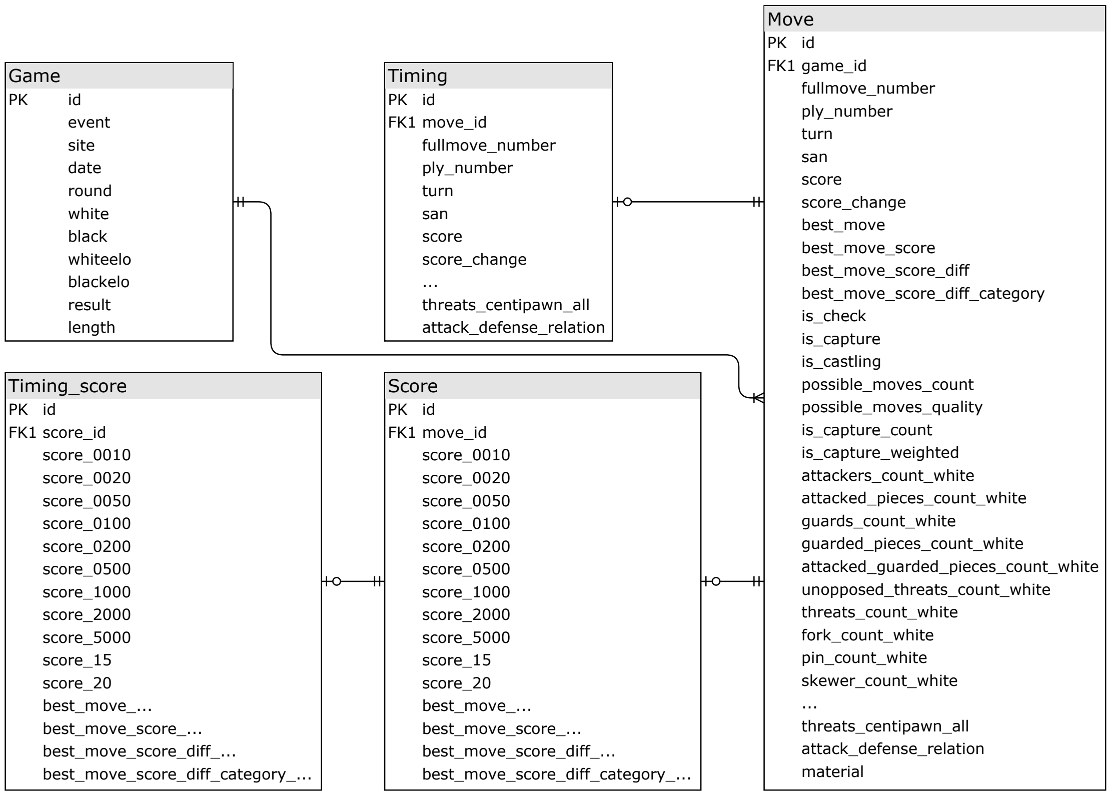


Figure 5.2: Entity relationship model of the SQLite database.

for each input parameter. This makes it possible to determine the best suitable input parameter for *Stockfish*'s evaluation function. More details on the *Stockfish* evaluation can be read in Chapter 6.

Figure 5.1 depicts the software's architecture including the four programs *Bulk Analysis*, *Repeated Bulk Analysis*, *Analysis CSV* and *Chess Music Tool*. The three programs in the *Chess Analysis Tool* block have a PGN input, use the chess analysis functions defined in the *Chess Library*. The *Chess Library* in turn utilizes the board representation, as well as chess engine support provided by *python-chess*. As mentioned before *Bulk Analysis* and *Repeated Bulk Analysis* store their output into a database. *Analysis CSV* on the other hand generates a CSV storing the metrics as well as an SVG depicting the chessboard for every move. Within the *Chess Music Tool* block the program consumes the generated CSV and SVG files and display their data in the *Qt* user interface. The *Chess Music Tool* program then proceeds to communicate with the prebuilt *FMOD* bank through the *FMOD Studio* API. Lastly, the *FMOD* bank stores the sound assets and plays the according sounds of the soundtrack.

5.1.1 Database

The *SQLite*[4] database—as despicted in Figure 5.2—stores game meta data, the game's moves, the moves associated chess metrics and the computing duration for the metrics. This results in a database scheme consisting of the following tables: `Game`, `Move`, `Timing`, `Score` and `Timing_score`. The relationships between the tables are:

- A game stores many moves, but a move belongs to only one game.
- A move has zero or one timing and a timing belongs to only one move.
- A move has zero or one score and a score belongs to only one move.
- A score has zero or one score timing and a score timing belongs to only one score.

The tables `Timing` and `Timing_score` have the same attributes as `Move` and `Score`, but instead of saving the computed values both store the time needed to compute the attributes in seconds. This allows an evaluation for the qualification of metrics in a real-time scenario. In the `Move` table attributes may have postfixes that signify the attribution to a player. The postfixes are `white`, `black` or `all` and affiliate the attribute to the white player, black player or both players, respectively. If non of the postfixes are attached, the attribute is move specific, such as `fullmove_number`, `score` or `best move`. An exception of these patterns are `material` and `attack_defense_relation`.

5.2 Details on the Chess Analysis Tool

A closer look upon the code reveals the mechanics and complexity involved in the implementation. In this section the *Chess Analysis Tool*'s code is inspected and described in greater detail than in Section 5.1. First the initial operations for loading a chess game from a PGN file, the retrieval of a connection to a chess engine and the structure of the main program are described. Then the definition and computation of metrics are discussed.

5.2.1 Open a Chess Game

To begin the analysis process the *python-chess* library loads a chess game from a PGN file into a game variable, as seen in Program 5.1. The variable `board` is a stack of moves, in which a move represents a state of the chessboard. The `board` variable is very capable, as it provides functionality, such as board validation, move generation, attack retrieval, the manipulation of the move stack, etc [60]. `board` is used throughout the chess analysis to compute the chess metrics.

5.2.2 Connect to Stockfish

Next, the chess engine *Stockfish* is used to evaluate and retrieve the score of a move. Program 5.2 connects to the chess engine via UCI[5]. The function returns an engine object that is useful for evaluating moves and compute the best possible move in a given situation.

[4]https://www.sqlite.org/

[5]*Universal Chess Interface* enables the communication between a chess engine and a chess interface, as described in Section 2.5.

Program 5.1: Open a chess game.

```python
import chess
filename = "kasparov_karpov_1986"
pgn = open("pgn/" + filename + ".pgn")
act_game = chess.pgn.read_game(pgn)
board = act_game.board()
```

Program 5.2: Connect to *Stockfish*.

```python
import chess.engine
# Connect program with the chess engine Stockfish via UCI
def connect_to_stockfish():
    return chess.engine.SimpleEngine.popen_uci("engine/stockfish_10_x64.exe
    ")
```

5.2.3 Main Program for Chess Analysis

Program 5.3 displays the instructions required to read a chess game's PGN file, analyze its every move and generate the output as CSV and SVG files. First of the program imports *python-chess* for game manipulation, *IPython* for generating SVG files and the projects own chess related methods and utility methods from `chess_analysis.py` and `chess_io.py`, respectively. The main method initializes the time limit for *Stockfish*'s evaluation, retrieves an chess `engine` object for the communication between the project and *Stockfish*, sets up a folder structure for the games output, reads a chess game from a PGN file and sets up the `counts` list, in which every calculated metric is stored for every move.

The `for` loop iterates over each move of a chess game and provides the `ply_number` and `mv` (move) for the current move. The `ply_number` is a half move, meaning the movement of a piece by a single player—see Section 2.1.1 for more details on moves. An important detail for the move evaluation is that `compute_score()` computes the score for the current move and the best possible move, which is not played yet. For this specific reason the score needs to be evaluated two times in the first run of the loop. First to calculate the best possible move and second to evaluate the move's score after the move has been applied to the `board`. The calculation of most metrics is omitted from Program 5.3 due to a lack of space in this document, nevertheless every chess metric is computed within the `for` loop and appended to the `counts` list. At the end of each move evaluation the best move and score are stored for the next iteration as both variables are required for the computation of the metrics. A SVG representation of the current board position is stored to the games output folder and is named by using the current move's ply number. When the loop has finished, the game is stored to a CSV file for the use of the soundtrack.

Program 5.3: Main method for `analysis_csv.py`.

```python
import chess
from IPython.display import SVG
from lib import chess_analysis, chess_io

def main():
    time = 0.100  # time limit for Stockfish evaluation
    engine = chess_analysis.connect_to_stockfish()
    filename = "kasparov_karpov_1986"
    chess_io.init_folder_structure(filename)  # prepare folders for output
    act_game = chess_analysis.read_game(chess_io.open_pgn(filename))
    board = act_game.board() # Get the intial board of the game
    counts = {  # saves a list for every metric
        "score": [],  # stores the scores calculated by Stockfish
        "best_move": [],  # stores the best move in SAN
        ...
    }
    prev_score, score_change, score = 0
    best_move, next_best_move = None
    # Iterate through every moves and play them on a board.
    for ply_number, mv in enumerate(act_game.mainline_moves(), start=1):
        fullmove_number = board.fullmove_number
        if best_move is None:  # calculate best move for first turn
            not_needed, best_move = chess_analysis.compute_score(
                                                    engine, board, time)
        best_move_score = chess_analysis.compute_best_move_score(
                                         engine, board, best_move, time)
        board.push(mv)  # apply move
        score, next_best_move = chess_analysis.compute_score(
                                                    engine, board, time)
        counts["score"].append(score)  # append parameters to the arrays
        counts["best_move"].append(best_move)
        best_move = next_best_move
        prev_score = score
        chess_io.export_board_svg(board, filename, len(ply_number), mv)
    chess_io.write_dict_to_csv(filename, counts)
    engine.quit()
main()
```

5.2.4 Computing Score and Best Move

Stockfish provides a very effective and efficient move evaluation as a score. The score is an accumulation of differently weighted values, e.g., material, threats, pawn structure, king safety. The score signals an advantage for a player over the other player. A positive

Program 5.4: Compute score and best move.

```python
def compute_score(engine, board, time):
    play = engine.play(board=board, limit=chess.engine.Limit(time=time),
                                                                   info=Info.ALL)
    score = play.info.get('score').white().score()
    if score is None:
        score = 0
    return score, play.move
```

Program 5.5: Compute best move's score.

```python
def compute_best_move_score(engine, board, move, time):
    board.push(move)
    info = engine.analyse(board=board, limit=chess.engine.Limit(time=time),
                                                                   info=Info.ALL)
    board.pop()
    score = info.get('score').white().score()
    if score is None:
        score = 0
    return score
```

score means an advantage for White and a negative score an advantage for Black. More details on the evaluation functions used by chess engines are described in Section 2.3.

For the score calculation the *Stockfish* engine is used, as seen in Program 5.4. The engine plays a move on the given chess board for a specified time frame. The `engine.play()` function returns a detailed list containing information on the analyzed move. Both the score and best move are extracted from this list. The score is returned as a centipawn value and the best move as a move object, in which the piece's initial square (from square) and target square (to position) are stored.

The best move is further analyzed to evaluate its score. Program 5.5 calculates the score for the previously generated best move from Program 5.4. The `engine.analyse()` function works similar to `engine.play()` with the notable difference of not calculating the best possible move. `engine.analyse()` requires the move to be pushed onto the board before the evaluation. Afterwards the move is to be removed again as the best possible move is not actually played.

5.2.5 Categorize Difference between Best and Actual Move

The difference between the best possible move and the player's actual move is calculated pretty easily. To determine the meaning of this difference, it is necessary to divide it into five categories: blunders as 4, mistakes as 3, inaccuracies as 2, neutral moves as 1 and good moves as 0. A blunder—as described in Section 2.1.4 is a common term used in chess for a move that gives up a player's advantage or even leads to an imminent

Program 5.6: Categorize difference between best and actual move.

```python
def categorize_best_move_score_diff(best_move_score_diff):
    category = 1
    if best_move_score_diff >= 300:
        category = 4
    elif best_move_score_diff >= 100:
        category = 3
    elif best_move_score_diff >= 50:
        category = 2
    elif best_move == actual_move:
        category = 0
    return category
```

defeat. The other categories are more or less derived from chess computers to give a more granular evaluation of a move's performance. A difference greater than 300 centipawn is considered to be a blunder. A difference between 100 and 299 centipawn is categorized as mistake. An inaccuracy occurs between 50 and 99 centipawn. A value below 50 is considered as a normal move. Lastly, if a move matches the best possible move it is a good move. The centipawn boundaries are taken from lichess [51]. This scheme is depicted in Program 5.6.

5.2.6 Analyzing Captures

It is important to not only signal if a piece or pawn has been captured but additionally convey the captures significance to the game. The function

$$c(p_{\text{aa}}, p_{\text{ab}}, m_{\text{c}}, m_{\text{p}}) = \begin{cases} v(p_{\text{ab}}) & \text{if } m_{\text{c}} \text{ is capture} \wedge p_{\text{ab}} \text{ is undefended,} \\ v(p_{\text{ab}}) - v(p_{\text{aa}}) & \text{if } m_{\text{c}} \text{ is capture} \wedge (p_{\text{ab}} \text{ is defended} \\ & \vee\ m_{\text{p}} \text{ is capture} \wedge s_{\text{to}}(m_{\text{p}}) = s_{\text{to}}(m_{\text{c}})), \\ 0 & \text{otherwise,} \end{cases} \tag{5.1}$$

calculates if a capture has led to winning material, losing material or an exchange of pieces. $c()$ is the weighted capture function, p_{aa} is an attacking piece, p_{ab} is an attacked piece, m_{c} is the current move, m_{p} is the previous move, $s_{\text{to}}(m_{\text{c}})$ is the square the current move moves its piece to, $s_{\text{to}}(m_{\text{p}})$ is the same but for the previous move and $v()$ is the centipawn value function. The attacker piece's centipawn value is only subtracted from the attacked piece's value if the attacked piece is either guarded or the move leads to an exchange of pieces. An exchange can occur when to consecutive plies have the same destination square (`to_square`). Thus, an exchange can be determined by the same two destination squares of the current and previous ply. A positive centipawn result is a clear gain of material, while a negative value loses material and should be avoided at all times. Therefore, not every capture of pieces is beneficial for an attacker and the involved pieces' values and their surrounding pieces should be taken into account.

Program 5.7: Compute the associated weight to a capture.

```python
def compute_is_capture_weighted(board, mv):
    value = 0
    if board.is_capture(mv):
        value = get_piece_centipawn(board, mv.to_square, True)
        if len(board.attackers(not board.turn, mv.to_square)) > 0: #guarded
            value-=get_piece_centipawn(board, mv.from_square, False)
        else:
            prev_mv = board.pop()
            if board.is_capture(prev_mv)
                and prev_mv.to_square==mv.to_square: #exchange
                value-=get_piece_centipawn(board, prev_mv.to_square, False)
            board.push(prev_mv)
    return value
```

Program 5.8: Detect attacks and returns a list of attack moves.

```python
def compute_attack_moves(board, color):
    pieces = board.piece_map()
    attack_moves = list()
    for square, piece in pieces.items():
        attackers = [i for i in board.attackers(not piece.color, square) if
                        board.piece_at(i).color == color]
        for a in attackers:
            attack_moves.append(chess.Move(a, square))
    return attack_moves
```

Program 5.7 implements the calculations and its cases from Equation 5.1 to return a weighted centipawn value for the capture.

5.2.7 Detect Attacks

Program 5.8 searches for attacking moves that attack the other player's pieces. The function iterates through every piece on the board. The `board.attackers()` detects attackers of the given color for a certain square. The attacking and the attacked squares are combined to a move and added to a list. After iterating over every square a list containing every attack move is returned.

5.2.8 Detect Guards

Program 5.9 works in a similar way to the determination of attack moves in Program 5.8. It iterates every piece on the board. The interesting fact of the functionality of `board.attackers()` is that the function does not check if the given square is occupied

Program 5.9: Detect guards for pieces and return a list of guard moves.

```python
def compute_guard_moves(board, color):
    pieces = board.piece_map()
    guard_moves = list()
    for square, piece in pieces.items():
        if piece.color == color:
            guards = board.attackers(color, square)
            for guard in guards:
                guard_moves.append(chess.Move(guard, square))
    return guard_moves
```

Program 5.10: Detect unopposed threats and return them as a list threatened pieces.

```python
def compute_unopposed_threats(attacked_pieces, guarded_pieces):
    unopposed_threats = set()
    for attacked_piece in attacked_pieces:
        if attacked_piece not in guarded_pieces:
            unopposed_threats.add(attacked_piece)
    return list(unopposed_threats)
```

by a white piece, black piece or a piece at all. It only returns the pieces of the given color that are capable of moving to the square. Consequently, the `board.attackers()` function can be utilized for both generating attackers, as well as guards.

5.2.9 Detect Unopposed Threats

Unopposed threats are attacking the opponent's hanging pieces—refer to Section 2.1.2 for a definition. Unopposed threats are always represent a safe way to win material, as the attacking piece cannot be captured in a counter move. Program 5.10 defines a function which takes `attacked_pieces` and `guarded_pieces` generated by Program 5.8 and Program 5.9. Each attacked piece is iterated over and compared to its existence in the guarded pieces list. If the attacked piece is not guarded, the attack counts as a unopposed threat.

5.2.10 Determine the Number of Possible Moves

Determining the number of legal moves a player has available in a given position is a simple solution to the mobility metric—as described in Section 2.1.6. The *python-chess* library provides a built-in function to retrieve the legal moves from the board at the current position. The Program 5.11 counts the number of legal moves.

Program 5.11: Function to count the number of legal moves.

```python
def compute_move_count(board):
    return len([i for i in board.legal_moves])
```

Program 5.12: Compute a list of threat moves for the given attack moves and attacked guarded pieces.

```python
def compute_threat_moves_weighted(board, attack_moves,
    attacked_guarded_squares):
    threat_moves = list()
    for attack_move in attack_moves:
        if attack_move.to_square in attacked_guarded_squares:
            if get_piece_centipawn(board, attack_move.to_square, True)
                >get_piece_centipawn(board, attack_move.from_square, True):
                threat_moves.append(attack_move)
        else:
            threat_moves.append(attack_move)

    return threat_moves
```

5.2.11 Determine the Number of Threats

Threats—as described in Section 2.1.2—attack either undefended pieces or pieces of higher centipawn value than the attacking piece. These threats lead to a clear gain of material. Program 5.12 iterates over a list of attack moves and compares its existence in a list of attacked guarded squares. If the attacked piece is guarded the centipawn value of the attacked piece is required to be higher than that of the attacking piece to count as a threat. If the attack is undefended, it is immediately counted as threat. This evaluation of threats gives a more complete evaluation of the situation than solely relying on unopposed threats.

5.2.12 Parameter Definitions

For the chess analysis a broad list of parameters has been implemented to test them on their usage in a soundtrack. These parameters range from a simple move's number to complex fork calculations. The following parameters are implemented within the program:

- `fullmove_number` is the number of the current move,
- `ply_number` is the number of the current half move,
- `turn` signals the player to make a move, with White being True and Black being False,
- `san` is the algebraic notation for the move,
- `lan` is the long algebraic notation for the move,

- `score` is the calculated score by *Stockfish*,
- `score_change` describes the difference between the scores of the current and previous move,
- `score_change_category` takes the `score_change` and divides it by the average `score_change`.
- `move_count` counts the number of possible moves,
- `best_move` is the best possible move described in SAN,
- `best_move_score` is the best move's score,
- `best_move_score_diff` is the score difference between the calculated best move and the actual move,
- `best_move_score_diff_category` is the category for the calculated difference—as described in Section 5.2.5,
- `is_check` determines if the opposed king is checked by the move,
- `is_capture` determines if the move actually captures a piece,
- `is_castling` is true if the king has been castled,
- `possible_moves_count` counts the number of possible moves available in the current turn,
- `possible_moves_quality` analyzes the possible move scores, ranks them and divides the number of score improving moves by the number of possible moves—this reveals the changes of improving the score in the current player's favor,
- `captures` is a list of possible captures,
- `is_capture_count` is the number of possible captures,
- `is_capture_weighted` is the weighted centipawn value for a capture—as described in Section 5.2.6,
- `attackers` contains the squares on which the opponent's pieces reside, that could capture a piece,
- `attackers_count` counts the number of attackers,
- `attacked_pieces` the squares of the pieces attacked by `attackers`,
- `attacked_pieces_count` counts the number of attacked pieces,
- `guards` are the squares on which the pieces reside, that defend attacked pieces,
- `guards_count` counts the number of guards,
- `guarded_pieces` are the squares on which the pieces reside, that are defended by `guards`,
- `guarded_pieces_count` counts the number of guarded pieces,
- `attacked_guarded_pieces` are the squares on which the pieces reside, that are both attacked, as well as guarded by other pieces.
- `attacked_guarded_pieces_count` counts the number of `threatened_guarded-_pieces`,
- `unopposed_threats` are the squares on which the residing pieces are threatened but not defended,
- `unopposed_threats_count` counts the number of unopposed threats,
- `threats_count` counts the threats that enable a gain in material,

- `forking_pieces` are the pieces that attack two or more opponent's pieces,
- `fork_count` is the number of forks on the board,
- `pin_count` is the number of pins,
- `skewer_count` is the number of skewers,
- `attackers_centipawn` multiplies attacking pieces with their respective centipawn value,
- `attacked_pieces_centipawn` multiplies attacked pieces with their respective centipawn value,
- `guard_centipawn` mutliplies guarding pieces with their respective centipawn value,
- `guarded_pieces_centipawn` multiplies the guarded pieces with their respective centipawn value,
- `attacked_guarded_pieces_centipawn` multiplies the `attacked_guarded_pieces` with their respective centipawn value,
- `unopposed_threats_centipawn` multiplies the unopposed threats with their respective centipawn value,
- `threats_centipawn` captures how threatened a player is by their opponent. It summarizes the centipawn values of the threatened pieces and subtracts the centipawn value of the least valuable attacker if the threatened piece is defended. The attacker is only considered when the threatened piece is defended, as only then it puts itself in the danger of being captured. The least valuable attacker is substracted because only one attacker can eventually make the capture. This leads to the definition:

$$t(m_\mathrm{n}) = \sum_{i=1}^{n} \left[\begin{cases} v(p_{\mathrm{ab,\ i}}) & \text{if } p_\mathrm{ab} \text{ is undefended,} \\ v(p_{\mathrm{ab,\ i}}) - \min_{j=1}^{o} v(p_{\mathrm{aa},j}) & \text{if } p_\mathrm{ab} \text{ is defended} \\ & \wedge\ v(p_\mathrm{ab}) > v(p_\mathrm{aa}), \\ 0 & \text{otherwise} \end{cases} \right], \tag{5.2}$$

$$\mathit{threats} = t(m_\mathrm{w}) - t(m_\mathrm{b}), \tag{5.3}$$

  here $t()$ is the function to calculate the centipawn value for the threats. m_n are the attacking moves, $v()$ is the function to retrieve a piece's centipawn value, p_aa is an attacker piece, p_ab is an attacked piece. *threats* is the centipawn result for the calculation, m_w are White's attack moves, while m_b are Black's attack moves.
- `attack_defense_relation` determines every piece involved in an attack and uses their centipawn value to calculate the relation to each other. An attack involves the attacked piece, its attackers, as well as its guards. Attackers and guards are included per attacked piece, which means a piece can be considered more than once. A guarding piece that defends two attacked pieces is taken into account two times. Therefore, the metric rewards effective piece placement. The metric is defined as follows:

$$ad(m_\mathrm{n}) = \sum_{i=1}^{n} \left[v(p_{\mathrm{gb,\,i}}) + \sum_{j=1}^{o} v(p_{\mathrm{ga,\,j}}) - v(p_{\mathrm{ab,\,i}}) - \sum_{k=1}^{p} v(p_{\mathrm{aa,\,k}}) \right], \tag{5.4}$$

$$\mathit{attack/defense} = ad(m_\mathrm{w}) - ad(m_\mathrm{b}), \tag{5.5}$$

here $ad()$ is the function to calculate the attack/defense relation to a centipawn value. m_n are the attacking moves, $v()$ is the function to retrieve the piece's centipawn value, p_{aa} is an attacker piece, p_{ab} is an attacked piece, p_{ga} is a guard piece and p_{gb} is a guarded piece. *attack/defense* calculates the centipawn result and compares the relations of both White and Black to another. m_{w} are White's attack moves, while m_{b} are Black's attack moves.

- `material` compares the remaining chess pieces of each player to another—as described in Section 2.1.5. The metric is calculated by using the pieces' centipawn values and is defined as:

$$m = (P_w - P_b) + 300(N_w - N_b) + 300(B_w - B_b) + 500(R_w - R_b) + 900(Q_w - Q_b), \quad (5.6)$$

  whereas P_w, N_w, B_w, R_w and Q_w are the number of remaining white pawns, knights, bishops, rooks and queens, respectively [28]. Black pieces are annotated with the letter b, e.g., P_b.
- `pawnending` checks if only kings and pawns are left on the board,
- `rookending` checks if only kings, rooks and possible pawns are left on the board.

The 24 parameters from the list ranging from `attackers` to `threats_centipawn` are stored for both the white and black player with the postfixes `_white` and `_black`. This allows to have the data available for both players at any time.

Additionally, these parameters sum up the respective black and white parameters and therefore give easy access to the complete situation on the board independent of the individual players:

- `attackers_count_all`,
- `attacked_pieces_count_all`,
- `guards_count_all`,
- `guarded_pieces_count_all`,
- `attacked_guarded_pieces_count_all`,
- `unopposed_threats_count_all`,
- `threats_count_all`,
- `fork_count_all`,
- `pin_count_all`,
- `skewer_count_all`,
- `attacked_pieces_centipawn_all`,
- `guarded_pieces_centipawn_all`,
- `attacked_guarded_pieces_centipawn_all`,
- `unopposed_threats_centipawn_all`.

`threats_centipawn_all` subtracts their White and Black values to provide a comparison between the player's threat situation.

Many of the discussed parameters are required to calculate the metrics utilized by the soundtrack. Nonetheless, each parameter is stored separately for an easier adaption of the soundtrack without necessarily adapting the analysis program.

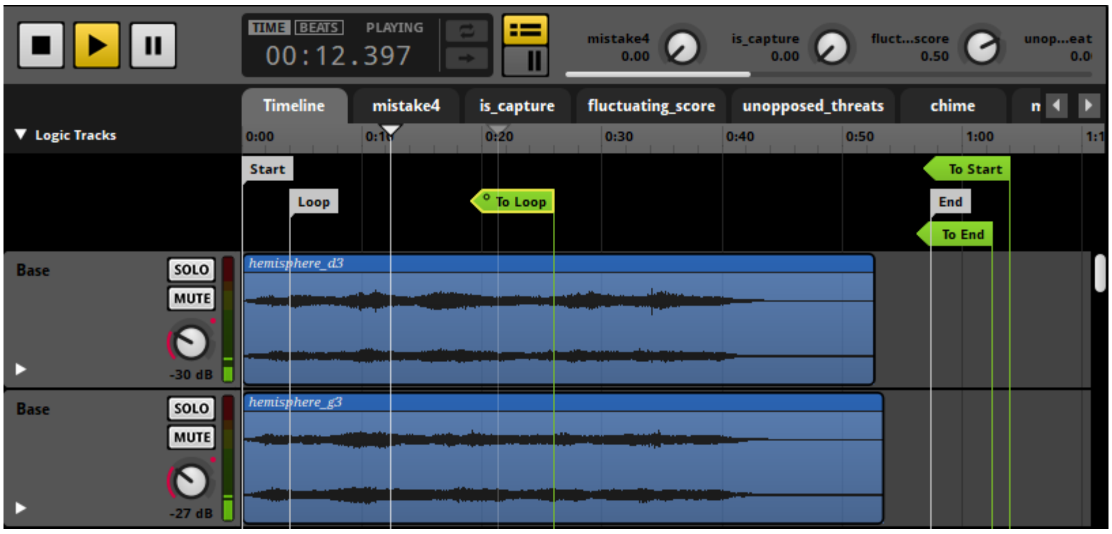


Figure 5.3: *FMOD* timeline depicts the setup of the destination markers and transition markers.

5.3 Details on the FMOD Studio Event

The *FMOD Studio* event is designed to be enjoyable for longer periods of time, as chess games can last between a few minutes to over an hour. For this reason melodies and rhythm has been omitted from the soundtrack design—as mentioned in Section 4.4.

The event's timeline—as shown in Figure 5.3—is constructed as follows: The *start* destination marker tags the beginning of the soundtrack at second 0, the *loop* destination marker marks the beginning of the main sound loop at second 4 and ends with a transition marker at second 26 back to the *loop*. When the chess game has ended the playback position leaves the main *loop* and plays the last seconds of the samples until it reaches the *end* destination marker at second 58 which loops back to the *end* marker when its transition marker is reached at second 62. The *end* loop goes on indefinitely, except the user goes back a couple of moves, in which the *end* loop is left again only to reach a transition marker at second 64 to lead back to the start of the soundtrack. The `game_phase` parameter determines which of the two loops are being played. The parameters values range from 0.0 to +1.0 and values between 0.0 to +0.5 activates the main *loop*, while values between +0.51 and +1.0 activate the *end* loop.

The soundtrack has five base layers with notes ranging from a low *d3* to a higher *a#4*, as described in Section 4.4.2. The five notes are evenly spaces fourths. The `fluctuating-_score` parameter has a value range of −1.0 to +1.0 and it determines which of the base layers is played. The base layers volume control is automated in such a way that the specific layers volume only increases when the `fluctuating_score` reached a certain range. Each note has a highest volume range of 25, except for *d3* and *a#4* which are positioned on the extremes—meaning 0.0 and +1.0. Those two notes have a range of 12.5. This leads to a smooth transition between the notes when needed.

The distribution is also beneficial when looping between the notes. The constant transitioning between the notes comes into play when the change in *Stockfish*'s evaluation between the previous and the current move is above the average score change. Consequently, the `fluctuating_score` can then be continuously increased and decreased to transition between the desired notes back and forth.

The effect for unopposed threats consists of six layers each a chromatic step above the layer before. The number of audible threat layers increases equally to the *unopposed_threats* parameter which has a value range between 0.0 and +10.0. The more threats are on the chess board, the more threat layers are activated and more dissonances are created. The dissonances should convey a feeling of unease and clearly state that something is wrong in the current situation.

Blunders, mistakes and inaccuracies are mapped to their own layers and parameters `mistake4`, `mistake3` and `mistake2`, respectively. The parameters value range is between 0.0 and +1.0. The layers have a volume automation, in which the automation's curve slowly rises to 4.00 dB at a parameter value of +0.5 and then maintains the volume until it dramatically drops before it reaches +1.0. For the effect one of the mistake parameters is set to +1.0 and the parameter immediately starts decreasing its value continuously, until it reaches 0.0 again. This leads to the volume automation curve being played in reverse, as one of mistake parameters is changed to +1.0 its corresponding layer's audio sample is almost immediately audible due to the sharp increase in volume. After that it maintains the volume for a few seconds before it slowly fades out again.

`possibe_moves` maps to the possible moves of the current player. Its value ranges from 0.0 to +100.0. When the possible moves are lower than +5.0 a low-pass filter is activated to dampen the sound. It signals the player that they do not have many moves at their disposal and their situation is probably very critical.

The base layers' outputs are not directly connected to the master output, but to the *Attack/Defense Submix*. The submix is utilized by the two parameters `is_capture` and `attack_defense`. `is_capture` controls a similar automation curve as the mistake parameters, but it controls a distortion effect, which is activated when the parameter is set to +1.0 and then slowly decreases its value to fade the effect out.

As the *attack/defense relation* gives an evaluation of which player has the better setup of possible attacks and guarding pieces the corresponding `attack_defense` parameter pans the audio output of the *Attack/Defense Submix* between the left and right channel. A positive parameter value emphasizes the right channel, while a negative value moves the base sounds to the left channel.

Lastly, the `chime` parameter activates a short beat whenever an effect has changed. The parameter ranges between 0.0 to +1000.0 and its sound is a nested event positioned at +900.0. Whenever the parameter is set to +900.0 the audio sample is played a single time. The sample only contains one beat for signifying a change in the scenery.

5.4 Details on the Chess Music Tool

5.4.1 Loading a Game into Chess Music Tool

In Program 5.13 the context switched from the *Chess Analysis Tool* to the *C++ Chess Music Tool* program. As a first step it is necessary to read in the CSV file produced by *Chess Analysis Tool*. On start up the program opens a dialog and requires the user to select a game's folder. A game includes a `game.csv` file and a `images` folder containing

Program 5.13: Load CSV file.

```
QString folder = "";
while (folder == "") {
  folder = QFileDialog::getExistingDirectory(this, tr("Open Game folder"), ".");
}
CSVReader reader(folder.toStdString());
m_game = reader.loadGame();
```

Program 5.14: Load *FMOD* bank and event.

```
FMOD::Studio::Bank* musicBank = NULL;
FMOD_RESULT result = system->loadBankFile(
    Common_MediaPath("Chess.bank"),
    FMOD_STUDIO_LOAD_BANK_NORMAL,
    &musicBank);

FMOD::Studio::EventDescription* eventDescription = NULL;
ERRCHECK(system->getEvent("event:/Soundscape 2", &eventDescription));

FMOD::Studio::EventInstance* eventInstance = NULL;
ERRCHECK(eventDescription->createInstance(&eventInstance));
```

a SVG file for every move. The SVG file's names are ascending numbers starting with 0. The `CSVReader` loads the game and creates a `Game` object. The `Game` object `m_game` stores the parameters of every move in a separate move list.

5.4.2 Load FMOD Bank

A bank stores the events defined in *FMOD Studio*. Every event contains their assets, effects, their arrangement, as well as the exposable parameters. The bank is the essential part to enable the program to play the desired sounds. Therefore, the `Chess.bank` is loaded into the program, as seen in Program 5.14. The *Chess Music Tool* utilizes the `Soundscape 2` event from the bank and an event instance is required to manipulate the soundtrack.

5.4.3 Initialize FMOD Parameters

The parameters defined in *FMOD Studio* can directly be manipulated by using the *FMOD Studio* API. Every time the parameter has to be changed the event instance's `setParameterValue()` method must be invoked. The first parameter of the method is the parameter's name in *FMOD Studio* as a string. The second parameter is the float value ranging from the—in *FMOD Studio* defined—value range. The Program 5.15 defines the initial value for the possible moves parameter in *FMOD* to be 20—as are the number of moves available to a player when a chess game begins.

Program 5.15: Set initial *FMOD* parameters.

```
m_fmod_possibleMoves = 20.0f;
ERRCHECK(m_eventInstance->setParameterValue(POSSIBLE_MOVES_STR, m_fmod_possibleMoves
    ));
```

5.4.4 Fluctuating Score

The function `fmodLoop`, described in Program 5.16, controls the behaviour of the fluctuating score for the *FMOD* event. The function is run in its own thread and therefore the parameter can be updated continuously. The variable `m_fmod_fluctuating_score` gradually increases from 0 to 1 by a step of 0.01 for every 150 ms. When 1 is reached, it decreases to 0 at which point it starts increasing again. `m_fluctuatingScore` is set to true when `score_shift_category` is greater than 1.0 and therefore enables the fluctuating effect. `m_fmod_fluctuating_score` is increased or decreased depending on the boolean evaluation of `m_waveDirection`.

If the `score_shift_category` is lower than 1.0 the score is not fluctuating and therefore the loop plays the same base note for as long the score is not significantly changed. Then the score itself determines which note is to be played. The conversion of the score to the *FMOD* parameter's value range is achieved by calculating $score/1000 - 0.5$. This results in the score being converted into a range between -1.0 and 0.0. Scores between -100 (-0.6) and $+100$ (-0.4) centipawn activate the neutral note of *c4*. A score between -300 (-0.8) and -100 (-0.6) activates the deeper note *g3* to signal a black player's advantage. Scores lower than -300 (-0.2) centipawn activate the even deeper note *d3*. In the opposite direction a score between $+100$ (-0.4) and $+300$ (-0.2) centipawn activate a higher *f4* note associated with a white player's advantage and a score over $+300$ (-0.2) centipawn activates the highest note of *a#4*.

The thread is started at startup of the program and continues to loop until the program is terminated. Every time a move is played the `m_fluctuatingScore` variable gets updated and the fluctuation effect is changed if the new value fits into other criteria. The threat running the `fmodLoop` function is initialized as described in Program 5.17.

5.5 Summary

The *Chess Analysis Tool* conducts an in-depth analysis on a broad set of chess parameters. It generates a CSV file for the usage in the soundtrack, as well as a database for an in-depth parameter evaluation for Chapter 6. The *FMOD Studio* event sets up a dynamic soundscape and exposes the according parameters—as the sound design defined it in Section 4.4—through the *FMOD Studio* API. The *Chess Music Tool* in turn consumes the *FMOD Studio* event and depicts a visual representation of the board and the parameters in a GUI. Additionally, the GUI offers the required controls to traverse a whole chess game.

Program 5.16: Function fmodLoop.

```
void FMODSoundscapeController::fmodLoop(QString name) {
  while (true) {
    if (m_aborted) return;
    if (m_fluctuatingScore) {
      if (m_fmod_fluctuating_score >= 1.0 || m_fmod_fluctuating_score <= 0)
              m_waveDiretion = !m_waveDirection;
      if (m_waveDirection)
        m_fmod_fluctuating_score += 0.01;
      else
        m_fmod_fluctuating_score -= 0.01;
    }
    else {
      if (m_score < -400)
        m_fmod_fluctuating_score = -0.9;
      else if (m_score > 400)
        m_fmod_fluctuating_score = -0.1;
      else
        m_fmod_fluctuating_score = (m_score / 1000) - 0.5;
    }
    ERRCHECK(m_eventInstance->setParameterValue(FLUCTUATING_SCORE_STR,
    m_fmod_fluctuating_score));

    ERRCHECK(m_system->update());
    QThread::msleep(150);
  }
}
```

Program 5.17: Looping through a chess game.

```
QFuture<void> fluctuating_thread = QtConcurrent::run(this, &FMODSoundscapeController
    ::fmodLoop, QString("A"));
```

Chapter 6

Evaluation

After Chapter 4 provided a detail description of the concept behind a data-driven soundtrack offering context on chess moves for inexperienced players and Chapter 5 examined concrete implementation details for the soundtrack, this chapter evaluates the design in four different evaluations. Firstly, in the heuristic evaluation three evaluators identify usability problems of the soundtrack's implementation. The chess metric ranking asks the evaluators to assess the metrics' suitability in the context of providing useful information for inexperienced chess players. The chess metric evaluation visualizes the metrics utilized in the second soundtrack design. Lastly, the real-time evaluation determines the metrics usefulness in a real-time context.

6.1 Heuristic Evaluation

Heuristic evaluation [21, Ch. 5] is a usability engineering method for identifying usability problems in an user interface. The evaluation defines a set of rules (heuristics), which the user interface is required to comply. A limited number of evaluators test the user interface for violations of the heuristics. These usability issues—referencing the violated heuristic—are collected from the different evaluators and compiled into a list. The issue list provides the developers with concrete starting points to implement solutions resolving the violations. The project for the thesis refrains from evaluating its graphical user interface, instead the soundtrack is evaluated, as the audio component conveys the information to the chess players. The *Chess Music Tool*'s user interface is used in the heuristic evaluation to provide the evaluators the necessary context for the soundtrack, as already played chess games are used for the evaluation and therefore the evaluators need visual aid to see the chess moves.

6.1.1 Method

Heuristic evaluations utilize the knowledge of domain experts to evaluate an user interface [58]. This process does not require a fully implemented program or prototype, but the prototype can even consist of paper mock-ups, because the heuristic evaluation method rather evaluates concepts than a concrete implementation. The heuristics used in an evaluation are a list of principles adjusted for the use case. Nielson suggests to invite three to five evaluators to inspect the interface [58]. Nielson conducted a series of

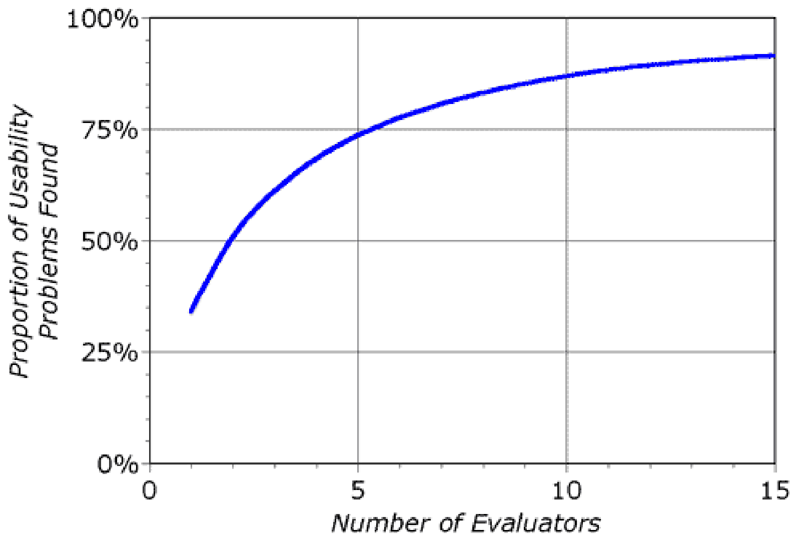


Figure 6.1: A curve comparing the number of evaluators to the proportion of usability problems found [39].

heuristic evaluations on six projects and concluded that a single evaluator can identify around 35 percent of usability problems [21]. Evaluators tend to find different problems and therefore complement each other. Three evaluators provide a valid evaluation and five evaluators a more complete picture. Figure 6.1 displays a curve of the efficiency for the number of different evaluators involved in a heuristic evaluation. The number of evaluators can be changed depending on the necessity of finding as many problems as possible or the cost. Experts in the evaluated domain tend to find more problems and consequently are beneficial for a successful evaluation.

The evaluation of the user interface is usually performed alone by an evaluator with the exception of an observer providing necessary information and taking notes during the evaluation. It is important not to have two evaluators inspecting the user interface in the same room simultaneously, as they influence each others thought process. During the evaluation the user interface is inspected mulitple times. The first time to get the evaluator acquainted to the user interface. After the user is familiar with the interface, further inspections aim to find usability issues. Evaluators use the list of heuristics to identify the concrete problems. The observer usually writes down the discovered usability issues. Additionally, evaluators can ask observers questions about the user interface for clarification. After every evaluation is completed, evaluators can gather together for a debriefing to discuss their findings and brainstorm possible solutions to the problems.

Next, a complete list of the usability problems is created and the evaluators are asked to rate them for the problem's severity [20]. The severity ranges from zero to four, zero being no problem and four a usability catastrophe. If an independent view on the problems is desired, the debriefing session should not be conducted yet, as it can change an evaluator's opinion on the issue. Asking only one evaluator to rate the issues does not provide a reliable rating. More effective is a rating by at least three or all of the involved evaluators and subsequently calculate the mean rating for the issues. The mean provides a reliable assessment of the issues. Lastly, the list of issues can be ordered by the mean severity rating to determine which issues are to be resolved first.

6.1.2 Adjusted Approach

For the evaluation a list of heuristics is derived from multiple pre-existing lists [19, pp. 156–158; 15, Ch. 7; 12, Appx. A]. The heuristics are heavily altered and new heuristics are added to fit the use case, because the original lists are designed for user interfaces or video games, but not for soundtracks directly. This results in a list of the nine heuristics specified in Table 6.1. The heuristics range from chess-related principles, such as the correct use of terminology or the chess metrics validity and significance, to soundtrack-related heuristics, such as the audio being able to convey semantics to the listeners. The heuristics for the soundtrack either solely focus on the sounds and audio effects or the heuristics discuss the soundtracks relation to the chess game.

Three experienced amateur chess players[1] have been invited to take part in the heuristic evaluation. Two of them have an additional programming background and one even knowledge in sound design. The evaluations took place on several separate occasions in cafés in Linz or Hagenberg. After a brief introduction between the observer and evaluator, the thesis' concept and soundtrack design was presented. Next, the list of heuristics was explained to the evaluators, enabling them to already look out for the aspects in question. The evaluators could inspect the soundtrack for two predetermined chess games. After one or two program walkthroughs the evaluators gave feedback to the chess metrics and the sound design. For the purpose of gathering the feedback into a document, the audio of the evaluation sessions was recorded. Subsequently, those recordings were transcribed and a list of issues was compiled. Following the last evaluation, the complete list of issues was sent out to the evaluators via email for rating the issues' severities. All three evaluators rated the issues, thus a reliable mean rating could be calculated.

The mean severity rating suggests which issues should resolved first. The severity rating is constructed similarly to the definition by Nielsen [20, 59]—a scale ranging from zero to four:

0 = Not a problem: The issue is not a problem at all.

1 = Negligible problem: The issue does not impact the experience. There is no need to fix this issue unless extra time is available.

2 = Minor problem: The issue has a small impact and fixing this should be given a low priority.

3 = Major problem: The issue has a strong impact and is important to fix, so it should be given high a priority.

4 = Catastrophe: The issue is must be fixed, before the program can be used.

[1]The chess players do not compete professionally, but have profound knowledge of chess, play in clubs or compete in amateur tournaments.

No.	*Title*	*Description*
1	Terminology of metrics is understandable for chess players.	The terminology used in the program is comprehensible for chess players. The terminology is commonly known by chess players.
2	Metrics are significant and valid.	The chess parameters presented in the program are useful and valid and therefore can be considered for the program.
3	The audio itself conveys a certain meaning.	When the sounds are considered detached from the game, they still have a meaning. The sounds create different atmospheres, e.g., alarming, suspenseful, mysterious, eerie, bustling, energetic, dynamic, uncomfortable, warm, welcoming, inviting.
4	The audio effects reflect the chess game and its defined metrics.	As a chess game is played through step by step, the soundtrack reacts to the metrics accordingly. The audio effects are fitting for the metric which triggers them.
5	The audio effects can be distinguished from another.	The user can recognize the audio effects even when triggered simultaneously. The effects are unique enough to be perceived as distinct effects.
6	The audio representation creates suspense when required (Arc of suspense, Spannungsbogen).	As a game progresses, thrilling and exciting situations occur. The soundtrack is designed in a way to create suspension when a critical moment takes place and relaxes when this is resolved.
7	The audio is enjoyable to replay.	The soundtrack retains its pleasing listening experience over the course of multiple games.
8	Audio representation supports the game.	The soundtrack retains its pleasing listening experience over the course of multiple games.
9	The sound design elicits a fitting emotional response.	The sounds trigger an expected feeling or atmosphere to the situations presented in the chess games.

Table 6.1: The list of heuristics used for the heuristic evaluation.

6.1.3 Result

The result of the heuristic evaluation is a list of twelve issues associated to the heuristics, discovered and rated for their severity by the three evaluators. The mean severity ratings are decimal numbers and for their concrete assignment to a severity category the mean ratings are either rounded up or down. Consequently, three of the problems are negligible as their rounded mean severity rating suggests. Three issues are minor problems, five are major problems and one issue is a catastrophe. The issues are described in the following list, ranked by their mean severity rating in an descending order. In addition to the quoted severity rating, the issue also references the heuristic it violates. *S.r.* and *h.* are abbreviations for the severity rating and heuristics, respectively. The issues are as follows:

3.67 s.r. / 5th h.: As the base tone is ever present, additional tones and sound effects are drowned out as the game progresses. Therefore, the effects lose their weight associated to them and aimed semantics cannot be transported as intended by the soundtrack's design. To solve this problem the base notes' volume can be lowered or the base notes are not always present. This should lead to an reduction of the noise level and a better recognition of the reactionary metrics, such as mistakes or captures.

3.33 s.r. / 4th h.: The chess metrics causing the different sound effects cannot be identified. Players have no possibility to match tones and sound effects to the actual chess metrics and its context remains unclear to the players. To tackle this problem a calibration phase can be introduced, in which the different sounds and effects are implicitly explained. At the beginning of the game fundamental tones and sound effects such as high notes for a leading white player can be activated whenever white is to move. This can lead a player to associate the corresponding tone to themselves. As the game progresses additional tones and effects can be added. Another approach can be to introduce different modes to the soundtrack that work with fewer metrics, resulting in easier identifiable context behind the sounds and effects.

3.33 s.r. / 5th h.: As a chess game progresses more tones and sound effects are added to the soundscape, resulting in an increase in volume but at the same time hamper the ability to perceive the individual effects and their associated metrics. The soundscape could focus on the most critical metric and depict it more prominent, but the determination of the most critical metric in a given situation seems not to be easily achievable.

3.33 s.r. / 8th h.: The soundtrack appears flat and conveys little information. As the soundtrack mostly works with increasing and decreasing the volume of different notes, it appears to be rather static and the changes in the metrics are not transported to the listener as intended. The base notes suggesting the leading player are not broad enough and therefore the low notes of a decisively leading black player are still to near to the neutral base note. The solution to this problem is to fine-tune the soundtrack over many iterations until a satisfactory dynamic range is reached.

3.00 s.r. / 6th h.: Suspenseful situations can be perceived through the soundtrack, but do not occur at the appropriate move or in the correct intensity. The suspense is created by dissonances in the soundtrack and those dissonances are mapped to the number of unopposed threats. Unopposed threats can be adjusted to include defended threats to capture a more complete situation on the board. Additionally the score evaluation can be incorporated into the effect. Dissonances can then be activated as the score diverges more to the extremes.

2.67 s.r. / 2nd h.: Unopposed threats do not capture the entirety of possible threats. Defended pieces can also be threatened and these threats can be as critical for the game as undefended threats. As suggested in the previous issue, the inclusion of defended threats can provide better context. Forks, pins and skewers are also threats which cannot be easily ignored by the threatened player and therefore would enhance the threat evaluation significantly. Nevertheless the use case for the soundtrack should be considered when expanding the scope of the chess metrics. The metrics should remain comprehensible to inexperienced chess players.

2.33 s.r. / 2nd h.: Captures and threats do not differentiate between the associated pieces as the pieces are only counted. Therefore, it is impossible to discern two different pieces, e.g., a threatened queen or bishop. The two pieces have completely different values associated to them in chess and a threatened queen must clearly be more emphasized than a threatened bishop. To solve this problem the centipawn values of the pieces can be taken into account and therefore regard the weight of situation.

2.33 s.r. / 2nd h.: Unopposed threats consider the threats of both colors – both white and black – simultaneously. This makes it hard for players to determine their own threats from of the soundscape. A solution to this would be to make the metric player-centric not chessboard-centric. This enables the players to hear their own unopposed threats instead of from both sides.

1.67 s.r. / 2nd h.: Most metrics such as score, score change and capture only react to the player's moves and do not anticipate possible outcomes. A possible new metric could analyze the possible moves for the current player and anticipate the options available to the player. Ranking the possible moves by their score can display the ratio of improving or worsening the player's current situation and score evaluation.

1.33 s.r. / 3rd h.: The soundtrack does not signal when a chess game ends. An easy solution is to simply stop after the game's last move or play distinct ending sounds.

1.00 s.r. / 1st h.: The terminology of parameter "score shift" does not quite fit, as it captures the change in the score evaluation from move to move. The name of the parameter can be changed to the more appropriate name "score change".

1.00 s.r. / 2nd h.: The program only highlights the mistake made by the current move. Previously made mistakes are completely dismissed. A possible solution would be to include a history that keeps record of the player's inaccuracies, mistakes and blunders. This could help to better capture the course of the game. Nevertheless, the score evaluation is sufficing as it already captures the situation as a whole.

The issues and their proposed solutions are useful for further developments of the soundtrack. They point out if the sound design's metrics provide the necessary information to convey the semantic content behind any given situation for an inexperienced player, as well as how a soundscape can be modified to be better suited for chess game dynamics.

Besides the heuristic evaluations, a lot of chess metrics were discussed and suggested to be used by the program during the evaluation meetings. The proposed metrics as well as the already implemented metrics are compiled to a list and ranked by the evaluators for their suitability in thesis' use case. The metric ranking is discussed in the next Section 6.2.

6.1.4 General Feedback

Furthermore, general feedback regarding the project and implementation was provided by the evaluators. The feedback suggests that the soundscape is the right approach for a sonification of chess. The soundscape sounds pleasant and is enjoyable over longer periods of time. The soundtrack can translate exciting moves from the chessboard to audio, but the sound effect's meaning remains concealed. Nevertheless, the effects are able to raise the player's attention to specific moves, despite the lack of concrete information. An explanation of the sound effects meaning can lead a player to be able to differentiate the sound effects from each other. Alternatively, a calibration of sound effects during the first few moves of a game could solve the problem of the sound effects hidden meaning. There is a lack in dynamics as the soundscape mostly remains the same throughout a game. Nevertheless, the soundscape can elicit a feeling of alertness as threats are on the board. The metrics utilized in the sound design are suitable for inexperienced players, but fail to capture the big picture in the game. The evaluators emphasize the importance of threats on the board. The threats determine the danger in a given situation. Therefore the soundtrack should capture not only unopposed threats, but also defended threats, pins, skewers and forsk. In the end the evaluation and feedback suggests that the soundtrack has much potential and the concept behind the soundtrack is worth pursuing.

6.2 Chess Metric Ranking

While the heuristic evaluation in Section 6.1 evaluated the soundtrack's issues and their severity, this evaluation asked the evaluators to rank the existing metrics in addition to their own suggested metrics. The aim in rating the metrics is to determine the best suitable metrics for conveying the current situation on the chessboard for inexperienced players, including fundamental metrics for learning chess—whether by signalling the leading player or threats are on board. This evaluation helps to determine and incorporate the most promising metrics—yielding relevant and understandable information for beginners—into the soundtrack. For this purpose, the evaluators give each metric a rating. The rating is defined by a linear scale, where 1 is a metric not suitable for beginners and 7 is a fundamental metric, which provides much needed context for inexperienced players. The steps are a fluent transition between the two extremes. Every metric is rated based on the following statement: The metric is essential for conveying fundamental information of a move or situation on the chessboard for inexperienced chess players.

The final rating for a metric is calculated by taking the mean of the individual evaluator ratings. The evaluated metrics are either described in Section 2.1 or in Section 5.2.12. The Table 6.2 lists the evaluated metrics by the rating in descending order. In regard of the metrics utilized by the second soundtrack design, it becomes apparent that four of the soundtrack's metrics are placed within the top five places in the list—score change, score, mistake and unopposed threats. Is capture and attack/defense relation get a rating of 4.33 and 4.00, placing them midfield. The number of possible moves receives a rating of 2.67 taking the 27th place in a list of 37 evaluated metrics. This ranking demonstrates that the sound design already utilizes important and useful

Metric	*Rating*	*Metric*	*Rating*
Score change	7.00	Material	4.00
Score	6.33	Impact field	4.00
Mistake	6.33	Is capture weighted	3.67
Threats centipawn	6.33	Impact weight	3.67
Unopposed threats	6.00	Half/fully open files	3.67
Fork	6.00	Mobility	3.33
Pin	6.00	Imbalances	3.00
Imminent mate threat	6.00	No. possible moves	2.67
Skewer	5.67	No. possible captures	2.67
Development	5.67	Attackers	2.67
King safety	5.33	Guards	2.67
Pawn structure	5.33	Guarded pieces	2.67
Is check	5.00	Guarded threats	2.67
Possible moves quality	5.00	Clock/time	2.67
Initiative	5.00	Openings	2.33
Best move	4.33	King-/Queenside play	2.33
Is capture	4.33	Move repetition	2.33
Attacked pieces	4.33	Is castling	1.67
Attack/defense relation	4.00		

Table 6.2: The list of ranked metrics in descending order.

metrics for inexperienced chess players. Furthermore, the list provides new and useful metrics for a possible future sound design.

6.3 Chess Metric Evaluation

The chess metric evaluation visualizes metrics used by the second sound design, as well as metrics deemed to be valuable by the evaluators from the heuristic evaluation. The graphs focus on the distribution of values or a metric's development over the course of chess games. The evaluation uses 989 chess games retrieved from the chess server *Lichess*. The *Bulk Analysis* program processes those 989 chess games and stores the analyzed metrics into a database. This database is the foundation for the metric evaluation as graphs are generated. The graphs are written in *Python* by using the data visualization tools *matplotlib*[2] and *seaborn*[3].

All 989 chess games are online matches between two people played on the chess platform *Lichess* and took place on 31 Dezember 2018. The players' strengths range from an *Elo* rating of 800 to 2610, while the mean *Elo* rating for white players and black players is 1550 and 1556, respectively. This ranks the average player of the data set above beginners and below professional players. The two distribution graphs in Figure 6.2 display the length distribution of the 989 games. The shortest game is 2

[2]https://matplotlib.org/
[3]https://seaborn.pydata.org/

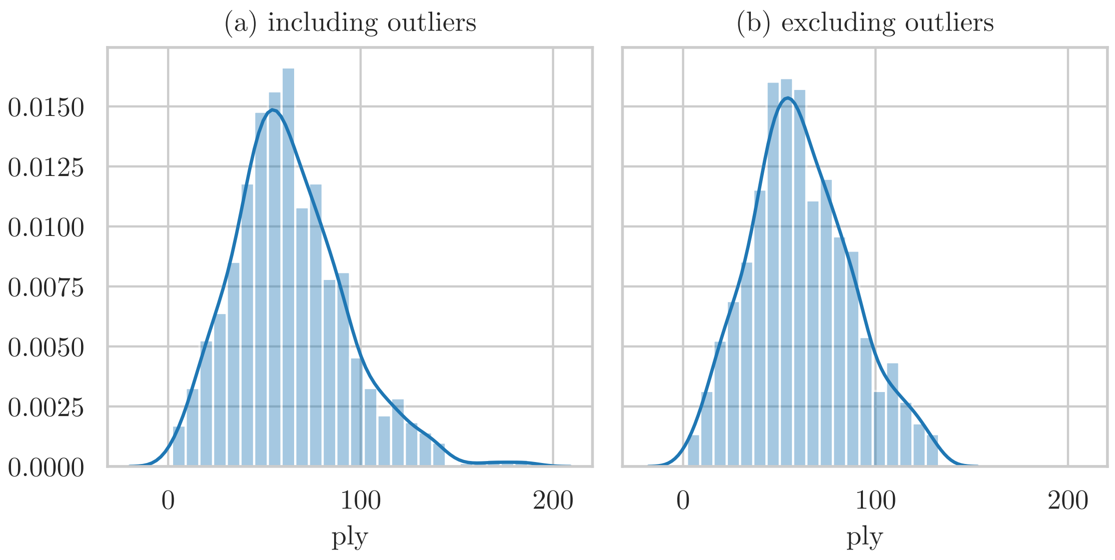


Figure 6.2: Distribution of games according to their length, (a) includes outliers, while (b) excludes outliers.

plies, while the longest game takes 187 plies to complete. The original data includes games that are uncommonly long, as displayed in Figure 6.2 (a). After removing the outliers the games are reduced from the original 989 to 968 and their length ranges from 2 to 133 plies. The graph in Figure 6.2 (b) depicts a more compact distribution of games according to their length and therefore enable the following graphs to be more comprehensible. The graphs will only consider moves up to the 133$^{\text{th}}$ ply, as games longer than 133 plies are outliers. 50 percent of the cleaned games are between 43 and 78 plies long.

6.3.1 Score and Score Change

The line graph in Figure 6.3 depicts the mean development of the score and score change, while the count of plies gives a hint on how many games the mean score and score change are derived from as the games progress. The average score change between the current and the previous move begins at 76.51 centipawn after the first two plies. In opening phase the score alternates for every ply as the players make their moves and change the evaluation. The change itself steadily increases by a small amount until ply 51 when it increased from the initial 76.51 to 303.75 centipawn. From this point onwards, the score abandons the pattern of alternating the score up and down by a predictable centipawn value, but begins to change its value randomly. The score seems to favour the white player during the first 51 plies, but as the games progress the mean score converges to the extremes in both directions. The score's change in pattern can be explained by the players acquiring significant advantages as their individual games progress and the number of games available for analysis drops with increasing plies. The loss in available games amplifies the extreme score evaluations. The outliers in the score and score change are not removed as the outliers demonstrate the variance of advantages

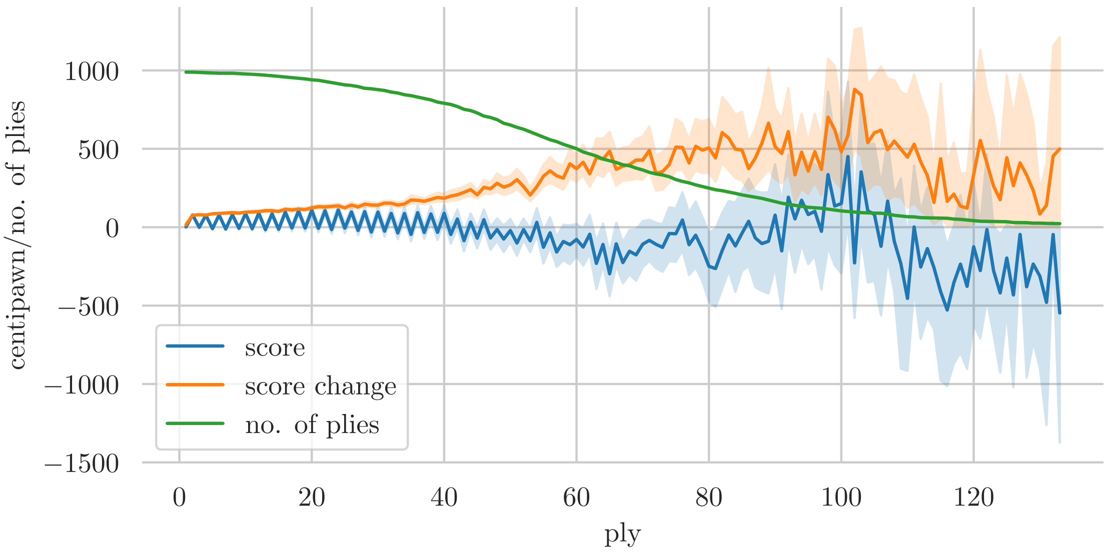


Figure 6.3: A comparison between the score and score change mean development in chess games.

a player can achieve within a game. Even though the mean score uses a predictable pattern in the opening phase, it develops a more chaotic behaviour towards the end. For the inclusion of the soundtrack this behaviour is highly interesting as it promises a diverse progression of individual games. The score does not develop the same way in games. The score change can determine a move's impact on the game. Therefore the median of all score change values is calculated which is 64 centipawn. This threshold can be utilized to determine a move's importance to the game and instruct the soundtrack to signal the moves significance when the threshold is exceeded by a move.

6.3.2 Threats

The line graph in Figure 6.4 compares the average development of the different kinds of threats over the course of the evaluated games. For this analysis the mean number of threats is calculated for total number of threats, unopposed threats, defended threats, pins, skewers and forks. The threats are calculated as defined in Section 5.2.11. Pins, skewers and forks have their own definition not directly dependant on the threats. Nevertheless, pins, skewers and forks are important to force the opponent to react and therefore are a threat in the context of the chess game, but not strictly in the sense of the threat definition in this thesis. This fact makes it possible for the number pins and skewers being higher than the total number of threats at certain plies. The number of threats rises steadily until it reaches its peak in the middle game. As the game progresses into the end game the number of threats significantly decreases. The unopposed threats make up most of the total number of threats, as unopposed threats are more easily achievable compared to defended threats. Defended threats are more relevant in the opening phase, as the number of pieces on the board is high. Between ply 20 and 50 the defended threats retain the same level of occurrence, while the number of unopposed

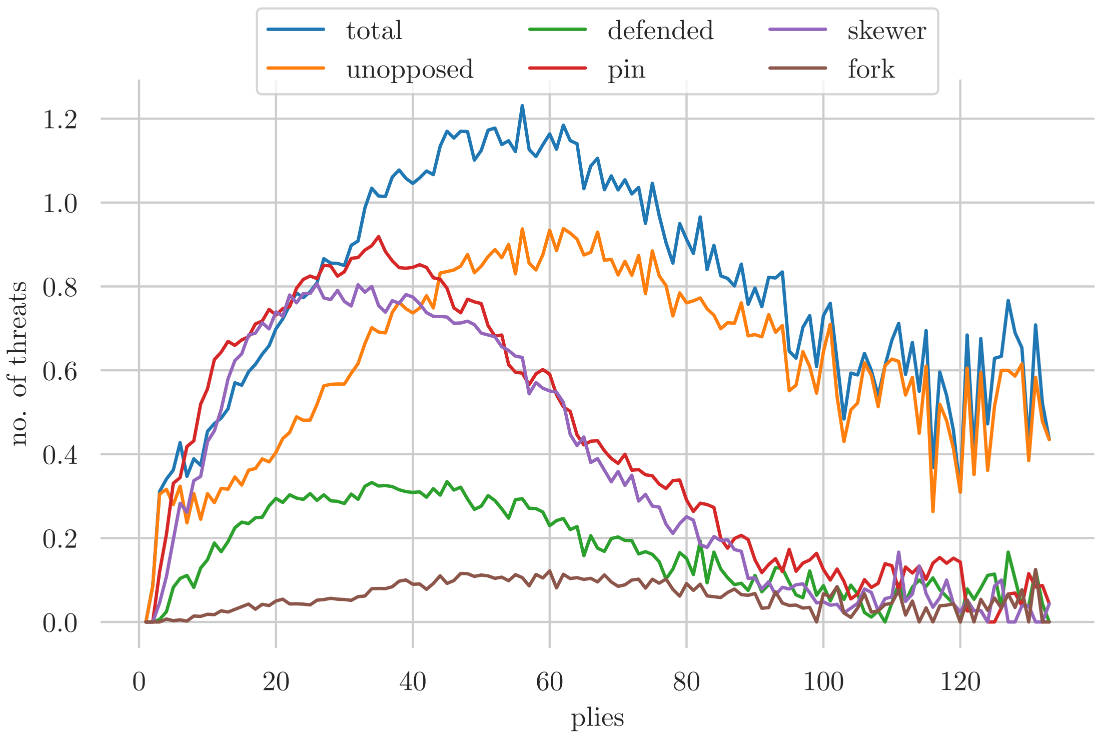


Figure 6.4: A comparison between the total number of threats, unopposed threats, defended threats, pins, skewers and forks and their mean development in chess games.

threats still rises. After ply 50 the undefended threats steadily decrease and converge to zero. The number of pins shares a similar curve to skewers. Both increase sharply in the beginning of the game and hit their peak between ply 20 and 40, after which both values steadily decrease. Forks do not occur often in any phase of the game, but are always relevant, because if a player creates a fork the player is almost guaranteed a capture. From the graph can be concluded that a player can build up threats in any game phase. Defended threats do not occur often in the end game for the lack of available defenders. Pins, skewers and forks also do not play a huge role in the end game. Despite the improbability of pins, skewers and forks appearing in the end game, they cannot be ignored as the three kinds of threats threaten multiple pieces at once and every piece needs to be preserved in the end game.

6.3.3 Mistakes

Blunders, mistakes and inaccuracies can often worsen the situation for a player significantly and those moves can be a decisive factor in a game. For this reason, the occurrence of those poor moves is interesting to analyze. First, the outliers need to be removed as a difference between the best possible move and the actual move of over 10.000 centipawn seems large to be taken into consideration. Before removing the outliers the values range from 0 to 15.320 centipawn and after that the adjusted values range from 0 to 215 centipawn, which drastically reduces the range of values. The box

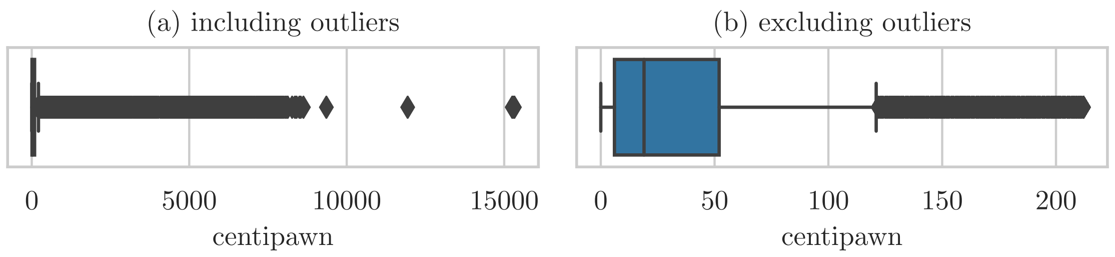


Figure 6.5: The distribution of difference between the best move's score and the actual score.

plot in Figure 6.5 (a) illustrates why the outlies have been removed, 50 percent of the values are contained within 8 and 91 centipawn and therefore rendering most values unrecognizable on an x-axis ranging up to 16.000 centipawn. The box plot in Figure 6.5 (b) on the other hand depicts the adjusted values within a range of 215 centipawn, which results in a balanced distribution of mistakes.

Only games of similar length are considered for an occurrence comparison of the different types of mistakes in Figure 6.6. This analysis shows when the different moves occur in the game and when each category is most relevant. The 151 games utilized for the mistake analysis have a length of 80 to 100 plies. This measure prevents shorter games from distorting the distribution, as games of lower length only contain mistakes at a lower ply number. For this analysis four different graphs are generated as *kernel density estimation* (KDE), each for another type of move (normal move, inaccuracy, mistake and blunder). The moves are categorized according to the *Lichess* definition as described in Section 4.3.7. According to the graph in Figure 6.6 (a), most normal moves occur throughout the whole game but become concentrated towards the end game and most of the normal moves have a centipawn value between 0 and 10. In Figure 6.6 (b) inaccuracies appear from the beginning of the game until their decline either before the middle game ends or at the beginning of the end game. The inaccuracies' centipawn values are evenly spread throughout the game, but concentrate at around 60 centipawn. Mistakes, as depicted in Figure 6.6 (c), most likely appear throughout the middle game, but also occur in the opening and end game. Most mistakes have a value ranging between 100 and 150 centipawn. Lastly, blunders do not immediately occur in the beginning but concentrate more throughout the middle game and end game, as displayed in Figure 6.6 (d). While games approach the end game blunders become more severe, as the broadening of the graph shows. Whereas at the beginning of games blunders are about 400 centipawn, in the end game blunders range more between 300 and 2000 centipawn. The decline in inaccuracies at the end of games indicates that other categories occur more frequent, such as normal moves, mistakes or blunders. Therefore the importance of inaccuracies diminishes as games progress. The general distribution of move types for the 898 games is the following: 64.28 percent are normal moves, 12.4 percent inaccuracies, 13.1 percent mistakes and 10.22 percent blunders. From the normal moves 55.5 percent equal the best possible move, according to the *Stockfish* evaluation.

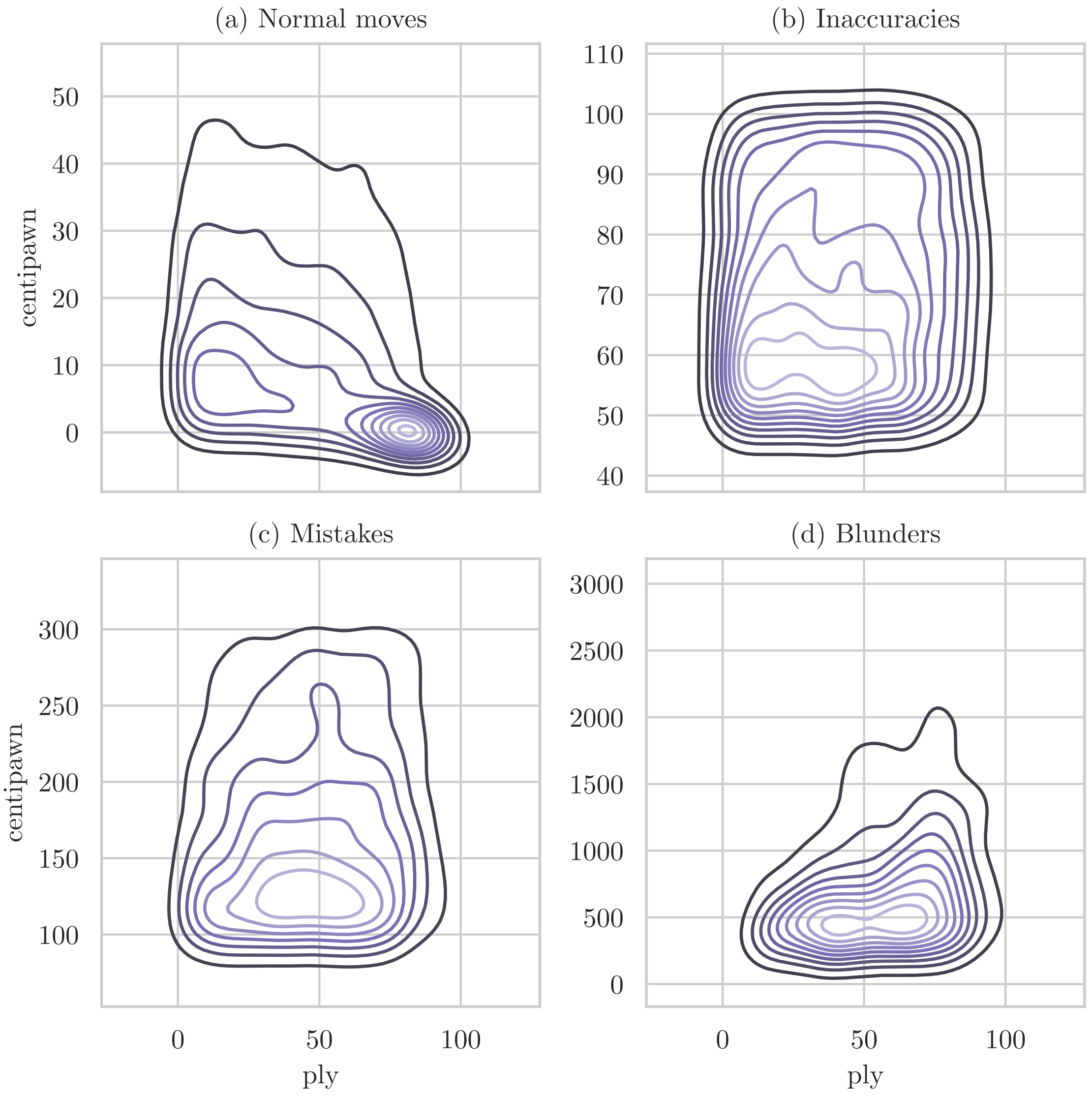


Figure 6.6: Kernel densities for different move categories in games with length ranging from 80 to 100 plies.

6.3.4 Captures

The line graph in Figure 6.7 displays the probability of a capture for the certain plies and it shows how the probability for captures develops over all analyzed games. The number of average captures starts at zero percent, as no piece can be captured within the first move. The following 20 plies the number of average captures increases drastically and reaches its peak of 33.96 percent at ply 34. The captures maintain a capture rate of average 30.9 percent captures starting from ply 20 to ply 54. From ply 54 onwards the probability of a capture decreases continuously, due to the fact that the number of remaining pieces diminishes per capture. The probability for captures rises while pieces are still in development and threats are created and reaches its peak when the game

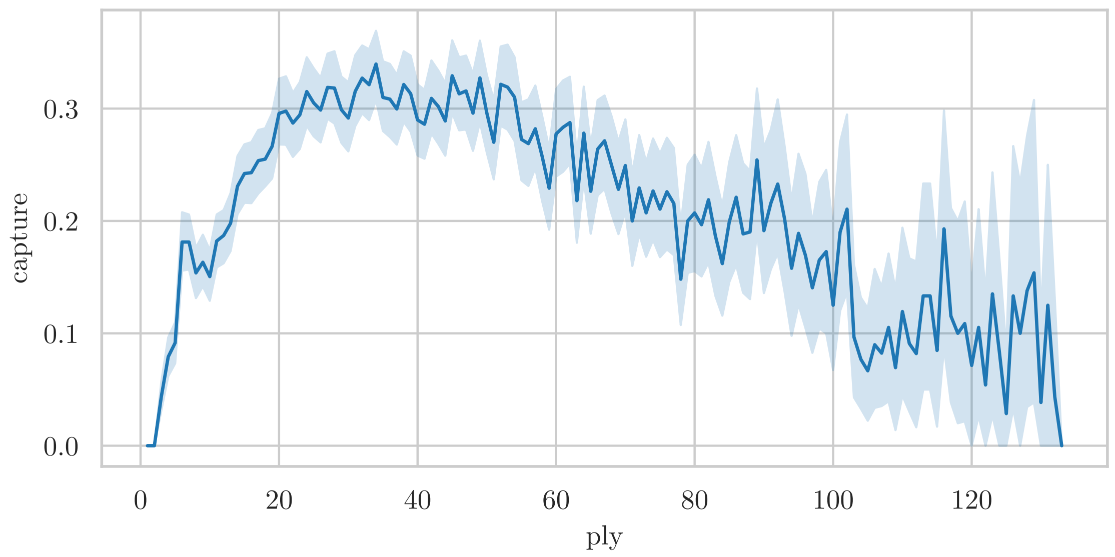


Figure 6.7: A line graph displaying the mean occurrence of captures throughout the analyzed chess games.

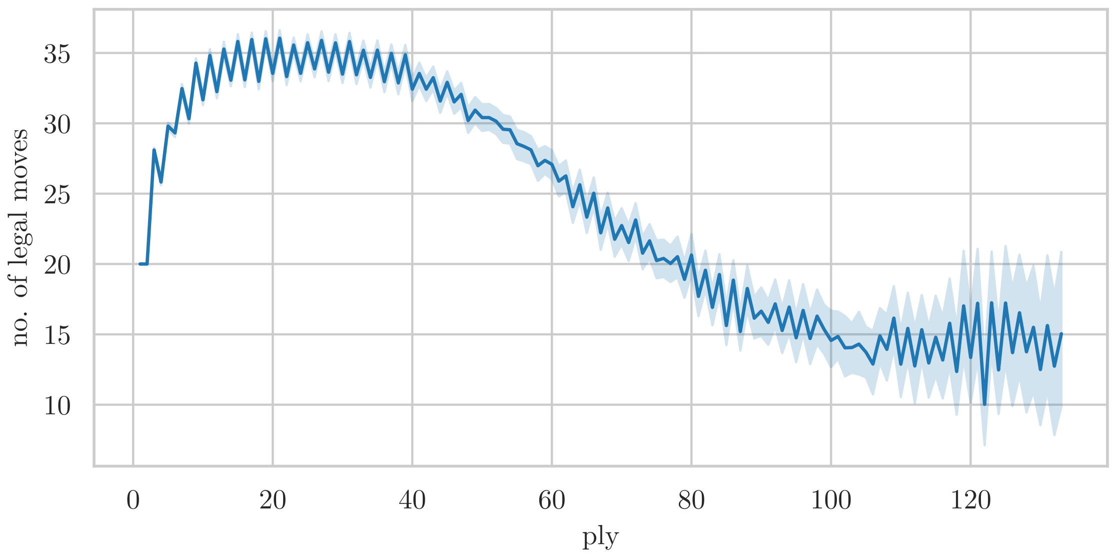


Figure 6.8: A line graph displaying the mean development of legal moves throughout the analyzed chess games.

enters the middle game and threats are converted to captures. After the middle game not many pieces are left to be removed from the board and players try to preserve as many pieces, while capturing the opponent's pieces. This leads the decreasing capture rate during the end game.

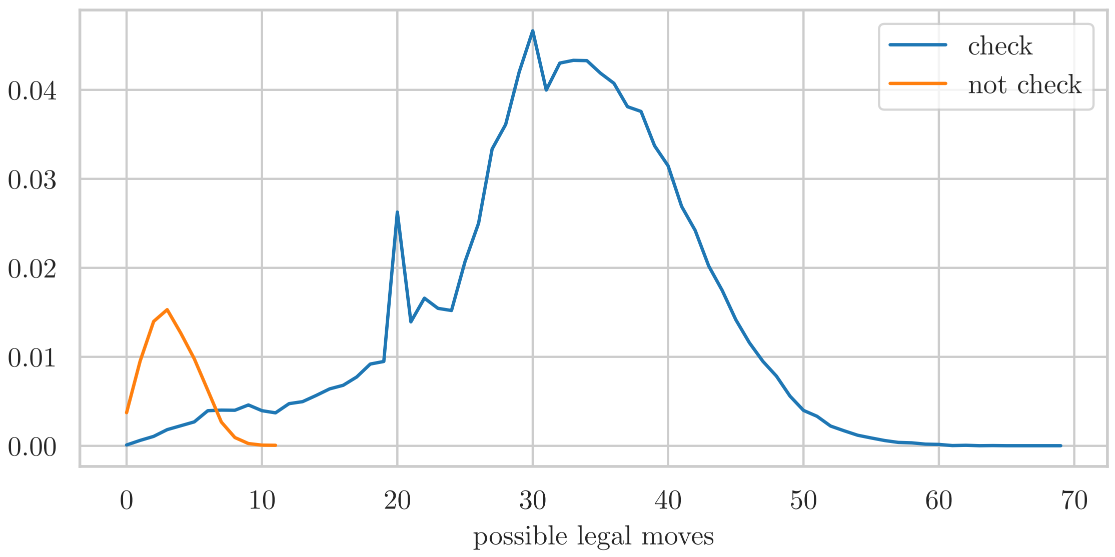


Figure 6.9: A line graph depicting the probability of the number of possible legal moves occurring in a game, while splitting the moves depending on the king being in check or not.

6.3.5 Legal Moves

Legal moves are the number of moves available to the players during their turn that comply the rules of chess. The number of legal moves is also related to the mobility—as described in Section 2.1.6—and defines how freely a player can move around the chessboard. Figure 6.8 depicts the mean legal moves available per ply. The number of legal moves starts for both players at 20 legal moves and increases sharply after a few moves. From ply 8 to 52 the average number of legal moves is above 30 and reaches its peak of 36.05 legal moves at ply 21. This peak signals that around ply 21 the players have developed their pieces and reached their maximum mobility. After that, pieces are captured, the game enters the middle game and the number of legal moves gradually decreases until a legal move count below the initial 20 moves is reached at ply 82 onwards. For the remaining moves the average number of legal moves fluctuates between 19.55 and 10.02. The number of legal moves can diverge from the mean, as the minimum is 1 and the maximum 69 legal moves. The overall mean of legal moves is 29.65, while the median is at 32 moves.

Having one legal move generally means that the player is in check, but the player also can have a larger number of legal moves available, despite their king being in check. The number of legal moves available to the player when the king is checked ranges between 1 and 11. Figure 6.9 depicts the probabilities of each number of possible legal moves ranging from 1 to 69. 92.48 percent of all moves do not check the king, meaning 7.52 percent of the analyzed moves do. 50 percent of the values for not checking legal moves reside between 27 and 38 legal moves, their mean is 31.79 and their median being at 33, while 50 percent of the values for checking legal moves are located between 2 and 5 legal moves, their mean is 3.33 and their median being at 3.

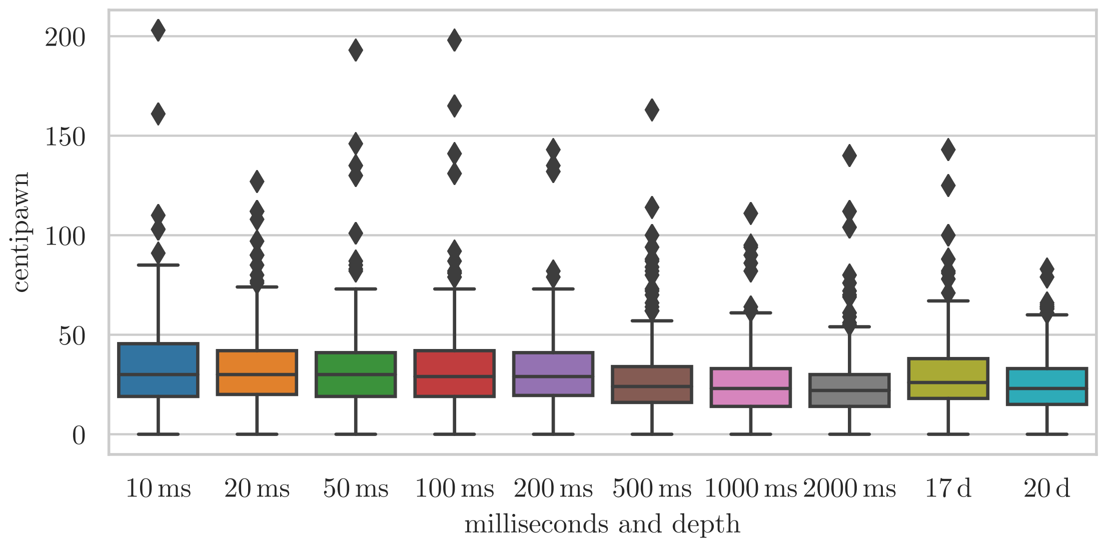


Figure 6.10: The distribution of score evaluation differences for various time and depth limits.

6.4 Real-Time Evaluation

The analyzed metrics are undergone a real-time analysis to evaluate their versatility in a real-time context. The parameters suitable for a live evaluation could be used in a chess installation. *Repeated Bulk Analysis*, as discussed in Section 5.1, repeatedly evaluates the score for the same move five times to generate comparable minimum and maximum values. The score's range for each move is then analyzed to determine the best suitable input parameters for a narrow mean score. Thus, discover the most reliable input parameters. Four chess games were used as a data source for *Repeated Bulk Analysis*, resulting in 1555 analyzed plies. For each ply nine score evaluation were conducted by using nine different input parameters. The score evaluations were limited to 10, 20, 50, 100, 200, 500, 1000, and 2000 milliseconds, as well as 17 and 20 depth-levels. The box plots in Figure 6.10 display the variation created when the score evaluation is limited by the corresponding input parameter. The mean dispersion of a score evaluation for 10 milliseconds is 34.0 centipawn, while 2 seconds yield a better result of 23.55 centipawn, as described in Table 6.3. Taking the mean value as well as the standard deviation into account 2000 milliseconds and 20 depth-level yield the best results. Yet to have a consistently timed evaluation, a time limit is more suitable, because a depth evaluation takes a different amount of time for each execution. For example the 20 depth-level evaluation's execution time can range from 1.64 seconds to 6.85 seconds, rendering it highly unreliable in a real-time context.

Parameters such as the score and best move can only be computed by *Stockfish*. Therefore, those two parameters and the parameters derived from score and best move take at least the amount of time a single evaluation needs. The *Stockfish* evaluation dependent parameters are:

- `score`,

	10 ms	*50 ms*	*100 ms*	*200 ms*	*500 ms*	*1000 ms*	*2000 ms*	*17 d*	*20 d*
count	311.00	311.00	311.00	311.00	311.00	311.00	311.00	311.00	311.00
mean	34.00	31.57	32.37	30.91	27.15	25.10	23.55	28.92	24.60
std	23.37	22.47	23.52	19.59	19.44	17.01	17.20	19.15	16.00
min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
25 %	19.00	19.00	19.00	19.50	16.00	14.00	14.00	18.00	15.00
50 %	30.00	30.00	29.00	29.00	24.00	23.00	22.00	26.00	23.00
75 %	45.50	41.00	42.00	41.00	34.00	33.00	30.00	38.00	33.00
max	203.00	193.00	198.00	143.00	163.00	111.00	140.00	143.00	83.00

Table 6.3: Statisical analysis of the score evaluation for the corresponding input parameters.

- `score_change`,
- `score_change_category`,
- `best_move`,
- `best_move_score`,
- `best_move_score_diff`,
- `best_move_score_diff_category` and
- `possible_moves_quality`.

As `possible_moves_quality` analyzes every legal move for its score, it takes $timelimit *$ *no. of possible moves* to compute. Every other parameter can be calculated independent from a chess engine and without taking up much computation time.

6.5 Summary

The different evaluation methods reveal the strengths and weaknesses of the second soundtrack design. The issues discovered in the heuristic evaluation indicate that further developments are required to create a soundtrack, in which inexperienced players can understand the game more easily and can properly assess the current situation. The chess metric evaluation provides a prioritized list of metrics that defines a reasonable order, in which further metric analysis should be conducted. Furthermore, on the basis of the real-time analysis can be concluded that every *Stockfish* dependant metric takes up at least the time predefined by the evaluation's limit. Other metrics can be computed immediately, without significant delay. A detailed discussion of the evaluation is described in Chapter 7.

Chapter 7

Conclusion

This chapter discusses the results from the evaluation, provides insights to further developments of the soundtrack and summarizes some problems faced during the work on the thesis, as well as possible improvements to the concept.

7.1 Evaluation Results

The heuristic evaluation sheds light on the issues the current soundtrack implementation has and provides useful feedback for its improvement. The ranking of metrics for their versatility in a sonification of chess for inexperienced players provides a useful list of promising metrics. The chess metric evaluation provides a necessary overview on how metrics develop throughout chess games and how value ranges for those metrics can be defined. Finally, the real-time analysis determines how the *Stockfish* evaluation is best utilized and what parameters are suited for a real-time analysis of chess games. The following list provides interesting findings:

- Implementing the soundtrack as a soundscape is a good way to add sounds to a game of chess as well as to transport a move's meaning. The soundscape sets a calming mood that can also elicit a feeling of alertness, when a player is a dangerous situation. The soundtrack is pleasant to listen to over longer periods of time precisely because it refrains from using melodies or rhythm.
- Although the soundscape fits the use case, the soundtrack lacks dynamics as it does not change much throughout the game. The chess metrics are permanently mapped to specific events and sound effects in the soundtrack. This is one of the main reasons the base theme of the soundtrack stays the same. The individual chess games certainly arrange the soundtrack differently based on the metrics which provides another feel to every game, but overall the soundtrack does not change much from game to game. To compensate this, the soundtrack could apply different musical themes determined in the opening phase of the game, or randomly choose a set of sounds to correspond to the metrics. Additionally, a different set of metrics could be chosen to change the input.
- The soundtrack mostly utilizes metrics which are suitable for inexperienced players, but metrics such as unopposed threats should be expanded to also include defended threats. Pins, skewers and forks are important leverage to gain initiative

and force the opponent to concentrate on defending their own pieces, while the player can develop more threats. Nevertheless, inexperienced players should not be overwhelmed with a multitude of complicated metrics but should be guided by fundamental metrics that a beginner can comprehend and find a solution to the problems hinted by the soundtrack.

- Defining and implementing a concrete metric is not an easy task, as most metrics are derived from empirical values and even chess engines have their own varying implementations for metrics. The chess metric evaluation in Section 6.3 uncovers the development of some chess metrics and gives information on their relevance:
  - The score provides the most complete and objective form of evaluation, as it is computed by a highly regarded chess engine. Individual chess games vary so much in their evaluation that the mean score development over all games cannot provide predictability for the score in the later stages of the game. Nevertheless, a fact is the score evaluation becomes more extreme as games progress.
  - On the contrary, the median score change provides an important threshold for the determination of important chess moves, as it can activate the *fluctuating score* effect from the second soundtrack design.
  - A combination of unopposed and defended threats provide improved context for the players as defended threats can have significant impact on a game. The evaluation also provides the information when and how numerous threats occur in a game. The mean development for the different threat types is useful to the soundtrack design as the number of expected threats is known for the individual game phases now.
  - Inaccuracies, mistakes and blunders can occur in every phase of a chess game. Inaccuracies appear more prominently in the opening, mistakes in the middle game, while blunders appear more often in the end game. Additionally, blunders become more severe in later stages of chess games. The soundtrack could reflect that behaviour.
  - The number of possible captures rapidly rises within the first moves and reaches as well as maintains this high value throughout the opening. After the opening, the number of possible captures decreases as attacks are converted to captures and as a result the number of remaining pieces declines continuously. The number of captures could be useful for the determination of the degree of development, as more possible captures correlates with the optimal position for a piece.
  - The number of legal moves has a similar progression as the number of captures. The number of legal moves can be associated to the status of a king being in check. When a king is checked, the number of legal moves significantly decreases. Therefore, the number of legal moves does not only serve as basis for the mobility metric, but implicitly reflects on a king being checked.
- Most of the implemented metrics do not take up significant computation time and are usable for a real-time scenario, whereas the score evaluation depends on the predefined time limit. There is a delicate balance required for choosing

a time limit which does not take up too much time to compute—for being able to reflect the chess game in a short time frame—and choosing a time limit large enough to provide a reliable score evaluation. A time limit between 500 ms and 2 s seems reasonable to fulfill both requirements. Generally speaking, the larger the evaluation's time limit, the better the evaluation's reliability. In classic chess games players tend to play more slowly as they are given more time to make their moves, as opposed to a blitz game using a time control of five minutes. This impacts the size for the time limit, as in a blitz game multiple moves can occur in rapid succession and therefore smaller time limits are preferred. There are not many score evaluation dependant metrics and most importantly every threat on the board can be calculated immediately and independent from *Stockfish*.

The takeaways present valuable guidelines for further developments and give insights to the issues associated with developing a soundtrack for chess. The chess metric evaluation can be expanded, but already provides valuable observations for the development of metrics over the course of a multitude of games, as well as the value range a metric can assume.

7.2 Future Prospects

The implementation for the data-driven soundtrack for chess clearly has shortcomings as evaluations suggest. The soundtrack's issues can be ironed out and solutions to those problems have been suggested. More refined input metrics for the soundtrack can provide necessary context for inexperienced players and improved dynamics in the soundtrack's implementation allow easier recognition of the situation. When considering the evaluation results, the data-driven soundtrack for chess is a proof of concept which utilizes the right ideas but requires refinement. Those refinements concern both the metrics and the sound implementation themselves. An in-depth analysis of additional chess metrics is required to determine the most useful metrics for the soundtrack. The chess metric ranking from Section 6.2 provides a list ordered by their suitability for the soundtrack and subsequently should be processed in the suggested order. Imminent mate threats, piece development, king safety or pawn structure could be analyzed in a next step and implemented in the soundtrack. Furthermore, the soundtrack could be enhanced to enable the activation and deactivation of certain metrics to provide the opportunity for detecting the proper mix of metrics.

As only a limited number of parameters are dependent on the time-consuming score evaluation many of the important parameters can be computed immediately after a move was played. Therefore further developments focused on a real-time implementation are feasible and allow a realization of a chessboard providing immediate feedback through the medium of sound. This could include a physical chessboard on which real chess can be played on. The chessboard could be capable of recognizing and analyzing moves in real-time, with the constraint being the time limit for a chess engine's evaluation. Defining a time limit between 500 ms and 2 s would provide a reliable score evaluation, depending on the computer's processing power. The analyzed parameters could then be sent to an audio synthesizer or sound engine, similar to a predefined *FMOD* event. Creating and controlling the soundtrack with the use of audio programming languages—

such as *Csound*, *SuperCollider* or *ChucK*—are also possible. The audio component could react to the given metrics and send its sounds to speakers. As the soundscape in the second sound design proved to be a promising implementation, this approach can be refined for a better depiction of chess for inexperienced players, as well as optimized for better dynamics. The chessboard could be placed within a room that houses multiple loudspeakers which are evenly distributed within a room, similar to Cage's *Reunion*. This enhanced concept would be a worthy approach to a sonification of chess games and it would be able to provide meaningful audio feedback for beginners.

7.3 Conclusion

Chess is a complex game in which a lot of strategy is involved. Inexperienced players often do not understand the full scope of the game and easily overlook game deciding threats. Therefore a data-driven soundtrack was developed to interpret chess games and play sounds reflecting the game. Few projects are dedicated to add a soundtrack to chess, most notably John Cage. Cage's *Reunion* aimed to add a soundtrack to the movements on the chessboard without minding their implied significance in the game. Contrary to Cage's approach, this soundtrack aims to provide meaningful feedback for players in various ways through the medium of sound. The soundtrack suggests the player with the an advantage in the game, signals threats on the board, addresses mistakes by the players, hints when a king is in check and more. The implementation is aimed to enable an in-depth analysis of chess metrics and determine their suitability to provide new insights into chess for inexperienced players. Unlike *Reunion*, the project does not process chess games in real-time. Nevertheless, the evaluation proves the possible implementation of a soundtrack played in real-time and controlled by a physical chessboard. It provides the necessary guidelines to implement a soundtrack suitable for inexperienced players.

Additionally, it can be highlighted that the usage of a soundscape was well received by the evaluators and they showed great interest in the development of the soundtrack. The evaluators identified significant potential for the soundtrack and reassured the validity behind the concept. Of course the soundtrack's implementation is not perfect, it utilizes suitable metrics, yet it does not provide a complete picture of the game. The soundtrack itself appears flat in its sounds. A challenge in the project was to create a enjoyable soundtrack. The first sound designed relied on the usage of instrumental music tracks, which annoyed listeners within in a short time frame. The utilization of a calm and subtle soundscape that plays certain notes throughout the whole game, as well as changes the notes depending on the evaluation, marked a great milestone. The soundscape enabled the listeners to concentrate on the chess game itself and not be completely distracted by the soundtrack.

Eventually, the soundtrack provides some insights into a game of chess for inexperienced players and assists players in assessing the current situation properly. Most of the metrics utilized in the second soundtrack design proved to be valuable and useful for the soundtrack. Further development especially in the sound design can introduce more dynamics to the soundtrack. A development of a physical chessboard analyzing the game while playing a soundtrack which reflects on the analyzed metrics in real-time seems feasible and the thesis provides valuable findings for such an implementation.

Appendix A

Technical Details

A.1 Project Directory

The thesis project consists of three main sections, as presented in Figure A.1. Folder `ChessAnalysisTool` contains the tools to analyze chess games, including the three programs *Analysis CSV*, *Bulk Analysis* and *Repeated Bulk Analysis*—described in Section 5.1. The *Analysis CSV* program is implemented in the file `analysis_csv.py`, which in turn is stored in the folder `csv_program`. The database-related *Python* programs *Bulk Analysis* and *Repeated Bulk Analysis* are implemented as `bulk_analysis.py` and `repeated_bulk_analysis.py`, respectively. Both programs are located in folder `db_program`. `engine` contains the *Stockfish* chess engine which is used for evaluating chess games. The `graphs` folder includes multiple programs used to generate the plots for the chess metric evaluation described in Section 6.3. The `lib` folder contains the base functionality written in *Python* for the different analysis programs. The `models` folder contains the object models for the database. The `output` folder contains all analyzed games including a move by move analysis stored as CSV file, a line graph generated for the game scores and images of a chess board for every move in a game as SVG files. Generated databases and graphs are also stored in `output`. `pgn` contains all chess game files that serve as input for the *Python* programs.

`ChessFMODStudio` contains the *FMOD Studio* project itself and all related data. All audio tracks which serve as base for *FMOD* are located in `Assets`. `Metadata` stores the meta data for the project e.g., the link to the assets within the project, parameter presets or the folder to the bank's location. In `Build` reside the banks generated by *FMOD*, ready to be used by a *C++* program.

`ChessMusicTool` contains the *C++* application for controlling the soundtrack by the provided chess metric data. The application contains four folders. The `ChessMusicGUI` folder is comprised of the main program, the `common` folder containing utility methods for accessing the *FMOD Studio* API and the `games` folder for input files. `lowlevel` and `studio` contain the necessary `.dll` files to run the program. `media` holds *FMOD* banks for soundtrack consumption.

```
/
├── ChessAnalysisTool
│   ├── csv_program
│   ├── db_program
│   ├── engine
│   ├── graphs
│   ├── lib
│   ├── models
│   ├── output
│   │   ├── db
│   │   ├── graph
│   │   └── kasparov_karpov_1986
│   │       └── images
│   └── pgn
├── ChessFMODStudio
│   ├── Assets
│   ├── Build
│   └── Metadata
└── ChessMusicTool
    ├── ChessMusicGUI
    │   ├── common
    │   └── games
    │       └── kasparov_karpov_1986
    ├── lowlevel
    ├── media
    └── studio
```

Figure A.1: Directory of the project.

Appendix B

CD-ROM/DVD Contents

Format: CD-ROM, Single Layer, ISO9660-Format

B.1 PDF-Files

Path: /

Höller_Stefan_2019.pdf	Master thesis

Path: /evaluation

chess_metrics.pdf . . .	A list of metrics ranked by the evaluators as described in Section 6.2
handout_de.pdf	The handout given to the evaluators in German
handout_en.pdf	The handout given to the evaluators in English
heuristic_problems.pdf	A list of problems found by the evaluators during the heuristic evaluation as described in Section 6.1
results_a.pdf	The completed handout by evaluator A
results_b.pdf	The completed handout by evaluator B
results_c.pdf	The completed handout by evaluator C

Path: /online_sources

*.pdf	Online references saved as Portable Document Format files
*.jpg, *.png	Online references saved as raster graphics

B.2 Project

Path: /project/bin

ChessMusicTool.zip . .	Executable program for the data-driven soundtrack

Path: /project/other

*.mp3	Audio recordings of the second soundtrack design for different chess games.

Path: /project/src

ChessAnalysisTool.zip	Project files for the *Chess Analysis Tool*. Archive structured as described in Appendix A.1
ChessFMODStudio.zip	Project files for second sound design's *FMOD* project. Archive structured as described in Appendix A.1
ChessMusicTool.zip	Project files for the *Chess Music Tool*. Archive structured as described in Appendix A.1

B.3 Miscellaneous

Path: /images

*.pdf, *.pdf_tex	Original Portable Document Format files
*.svg	Original Scalable Vector Graphics files
*.jpg, *.png	Original raster graphics

References

Literature

[1] George M Adel'son-Vel'skii et al. "Programming a computer to play chess". *Russian Mathematical Surveys* 25.2 (1970), pp. 221–262 (cit. on p. 12).

[2] Richard Bellman. "On the application of dynamic programing to the determination of optimal play in chess and checkers". In: *Proceedings of the National Academy of Sciences of the United States of America.* Vol. 53. 2. National Academy of Sciences, 1965, pp. 244–247 (cit. on p. 15).

[3] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. "Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription". In: *Proceedings of the 29th International Conference on International Conference on Machine Learning.* Edinburgh, Scotland, UK: Omnipress, June 2012, pp. 1881–1888 (cit. on p. 23).

[4] Graham Burgess. *The Mammoth Book of CHESS.* 2nd ed. London: Constable & Robinson Ltd, 2009 (cit. on pp. 3–7, 17).

[5] José R. Capablanca. *Chess Fundamentals.* Algebraic edition. New York: Harcourt, Brace and Company, 1934 (cit. on p. 7).

[6] Joel Chadabe. "Interactive Composing: An Overview". *Computer Music Journal* 8.1 (1984), pp. 22–27 (cit. on p. 23).

[7] William Clark, Jan Golinski, and Simon Schaffer. "Enlightened Automata". In: *The Sciences in Enlightened Europe.* University of Chicago Press, 1999. Chap. 5, pp. 126–165 (cit. on p. 8).

[8] Nick Collins et al. "Live coding in laptop performance". *Organised Sound* 8.3 (2003), pp. 321–330 (cit. on pp. 23, 24).

[9] Lowell Cross. "Reunion: John Cage, Marcel Duchamp, Electronic Music and Chess". *Leonardo Music Journal* 9 (1999), pp. 35–42 (cit. on pp. 24–26).

[10] Roger B. Dannenberg. "The Implementation of Nyquist, a Sound Synthesis Language". *Computer Music Journal* 21.3 (1997), pp. 71–82 (cit. on p. 20).

[11] Peter W. Frey. *Chess Skill in a Man and Machine.* 2nd ed. Springer, 1983 (cit. on pp. 12–14, 30).

[12] Andreas Friedl. “Integration of Mobile Devices into a Floor-Based Game to Increase Player Dynamics”. Masterarbeit. Hagenberg, Austria: University of Applied Sciences Upper Austria School of Informatics, Communications and Media, Sept. 2015 (cit. on p. 58).

[13] Richard D. Greenblatt, Donald E. Eastlake, and Stephen D. Crocker. “The Greenblatt Chess Program”. In: *AFIPS ’67 (Fall) Proceedings of the November 14-16, 1967, fall joint computer conference* (Anaheim). Washington D.C.: Thompson Books, Nov. 1967, pp. 801–810 (cit. on p. 13).

[14] Feng-hsiung Hsu. *Behind Deep Blue. Building the Computer that defeated the World Chess Champion.* Princeton University Press, 2002 (cit. on p. 11).

[15] Katherine Isbister and Noah Schaffer. *Game usability. Advice from the Experts for Advancing the Player Experience.* Elsevier, 2008 (cit. on p. 58).

[16] G.M. Levitt. *The Turk, Chess Automaton.* McFarland, Incorporated, Publishers, 2000 (cit. on p. 9).

[17] Max V. Mathews. *The Technology of Computer Music.* MIT Press, 1969 (cit. on p. 19).

[18] James McCartney. “Rethinking the Computer Music Language: SuperCollider”. *Computer Music Journal* 26.4 (2002), pp. 61–68 (cit. on p. 20).

[19] Jakob Nielsen. “Enhancing the explanatory power of usability heuristics”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 1994, pp. 152–158 (cit. on p. 58).

[20] Jakob Nielsen. “Reliability of Severity Estimates for Usability Problems Found by Heuristic Evaluation”. In: *Posters and Short Talks of the 1992 SIGCHI Conference on Human Factors in Computing Systems.* CHI ’92. Monterey, California: ACM, 1992, pp. 129–130 (cit. on pp. 57, 58).

[21] Jakob Nielsen. *Usability Engineering.* Mountain View: Morgan Kaufmann, 1994 (cit. on pp. 56, 57).

[22] Gerhard Nierhaus. *Algorithmic Composition. Paradigms of Automated Music Generation.* Vienna: Springer, 2009 (cit. on p. 22).

[23] John Nunn. *Learn Chess. A Gold-Medal Winner Explains How to Play and Win at Chess.* London: Gambit Publications, 2010 (cit. on pp. 7, 8).

[24] Dale E. Parson. “Chess-Based Composition and Improvisation for Non-Musicians”. In: *Proceedings of the International Conference on New Interfaces for Musical Expression.* Pittsburgh, PA, United States: nime.org, 2009, pp. 157–158 (cit. on pp. 26, 27).

[25] Edgar Allan Poe. “Poe’s Works”. In: ed. by John H. Ingram. Vol. 3. A & C Black, 1899. Chap. Maelzel’s Chess-Player, pp. 286–311 (cit. on p. 9).

[26] Miller Puckette. “Combining Event and Signal Processing in the MAX Graphical Programming Environment”. *Computer Music Journal* 15.3 (1991), pp. 68–77 (cit. on p. 20).

[27] Joseph F von Racknitz. *Ueber den Schachspieler des Herrn von Kempelen.* Müler, 1784 (cit. on p. 9).

[28] Claude E. Shannon. “Programming a Computer for Playing Chess”. *Philosophical Magazine Series 7* 41.314 (Mar. 1950), pp. 256–275 (cit. on pp. 10, 11, 13, 16, 29, 30, 50).

[29] Jeremy Silman. *How to Reassess Your Chess. Chess mastery through chess imbalances.* 4th ed. Los Angeles: Siles Press, 2010 (cit. on p. 7).

[30] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. *Science* 362.6419 (2018), pp. 1140–1144 (cit. on p. 11).

[31] Tom Standage. *Digitale Bildverarbeitung. The Life and Times of the Famous Eighteenth-Century Chess-Playing Machine.* 1st ed. New York: Walker Publishing Company, Inc, 2002 (cit. on p. 8).

[32] “Torres and his remarkable automatic devices”. *Scientific American* 80.2079 (Nov. 1915), pp. 296–298 (cit. on p. 9).

[33] Anders Tveit et al. “Reunion2012: A Novel Interface for Sound Producing Actions Through the Game of Chess”. In: *Proceedings of the International Conference on New Interfaces for Musical Expression.* Goldsmiths, University of London, 2014, pp. 561–564 (cit. on pp. 25, 26).

[34] Ge Wang. “The Chuck Audio Programming Language ‘A Strongly-timed and On-the-fly Environ/Mentality’ ”. PhD thesis. Princeton, NJ, USA, 2008 (cit. on p. 20).

[35] Ge Wang and Perry R. Cook. “ChucK: A Concurrent, On-the-fly, Audio Programming Language”. In: *Proceedings of the 2003 International Computer Music Conference.* International Computer Music Association, 2003 (cit. on p. 20).

[36] Robert Willis. *An Attempt to Analyse the Automaton Chess Player of Mr. De Kempelen.* Booth, 1821 (cit. on p. 9).

[37] Matthew Wright, Adrian Freed, et al. “Open SoundControl: A New Protocol for Communicating with Sound Synthesizers”. In: *Proceedings of the 1997 International Computer Music Conference.* International Computer Music Association, 1997 (cit. on p. 20).

Audio-visual media

[38] Shigeko Kubota. *Marcel Duchamp and John Cage.* Duchamp (white) moves a modified chessboard, while Cage and Duchamp’s wife watch. The chessboard is fitted with 64 photoresistors to register piece movements for control sounds. 1968. URL: https://hyperallergic.com/424124/marcel-duchamp-john-cage-reunion-chess-toronto/ (visited on 11/09/2019) (cit. on p. 26).

[39] Jakob Nielsen. *How to Conduct a Heuristic Evaluation.* Curve showing the proportion of usability problems in an interface found by heuristic evaluation using various numbers of evaluators. The curve represents the average of six case studies of heuristic evaluation. Nov. 1994. URL: https://media.nngroup.com/media/editor/2012/10/30/heur_eval_finding_curve.gif (cit. on p. 57).

[40] Karl Gottlieb von Windisch. *The Turkish Chess Player*. Copper engraving from the book: Karl Gottlieb von Windisch, Briefe über den Schachspieler des Hrn. von Kempelen, nebst drei Kupferstichen die diese berühmte Maschine vorstellen. 1783. URL: https://commons.wikimedia.org/w/index.php?title=File:Tuerkischer_schachspieler_windisch4.jpg&oldid=325912949 (visited on 10/22/2019) (cit. on p. 9).

Online sources

[41] *About - Stockfish*. URL: https://stockfishchess.org/about/ (visited on 11/04/2019) (cit. on p. 15).

[42] Dominick Blanchette. *Difference Between Attacked Piece and Threatened Piece*. URL: https://web.archive.org/web/20190528213030/http://chesslessons4beginners.com/rules/lesson-3-attack-threat.htm (visited on 05/28/2019) (cit. on p. 3).

[43] *CCRL 40/4 Rating List - All engines, best versions only*. URL: http://www.computerchess.org.uk/ccrl/404/ (visited on 11/04/2019) (cit. on p. 15).

[44] *Data-Driven Programming*. URL: https://en.wikipedia.org/w/index.php?title=Data-driven_programming&oldid=898745506 (visited on 11/07/2019) (cit. on p. 23).

[45] FIDE. *FIDE Laws of Chess*. 2018. URL: http://arbiters.europechess.org/wp-content/uploads/2019/05/Arbiters-Manual-Laws.pdf (visited on 10/16/2019) (cit. on pp. 16, 29).

[46] *Firelight Technologies FMOD Studio API*. Mar. 2019. URL: https://www.fmod.com/resources/documentation-api (visited on 03/14/2019) (cit. on p. 21).

[47] *FMOD Studio User Manual 1.10*. Mar. 2019. URL: https://www.fmod.com/resources/documentation-studio?page=welcome-to-fmod-studio.html (visited on 03/14/2019) (cit. on p. 21).

[48] Brain Foo. *Two Trains. Sonification of Income Inequality on the NYC Subway*. URL: https://datadrivendj.com/tracks/subway/ (visited on 11/07/2019) (cit. on p. 24).

[49] Adrian Freed and Matt Wright. *Open Sound Control*. URL: http://opensoundcontrol.org/introduction-osc (visited on 01/11/2019) (cit. on p. 20).

[50] *How does the Game Report Analysis work?* URL: https://support.chess.com/article/364-how-does-the-game-report-analysis-work (visited on 10/18/2019) (cit. on p. 32).

[51] Ilyps. *Help me understand the lichess computer*. June 2015. URL: https://www.reddit.com/r/chess/comments/38yw7o/help_me_understand_the_lichess_computer/cryyomi?utm_source=share&utm_medium=web2x (visited on 10/18/2019) (cit. on pp. 32, 44).

[52] *In the computer analysis, what's the difference between inaccuracy, mistake, and blunder?* URL: https://web.archive.org/web/20150922072729/https://support.chess.com/customer/portal/articles/1444907-in-the-computer-analysis-what-s-the-difference-between-inaccuracy-mistake-and-blunder- (visited on 10/18/2019) (cit. on p. 32).

[53] Interested readers of the Internet newsgroup rec.games.chess. *Portable Game Notation Specification and Implementation Guide.* Mar. 1994. URL: https://web.archive.org/web/20190411152024/https://www.thechessdrum.net/PGN_Reference.txt (visited on 04/11/2019) (cit. on pp. 16, 17).

[54] Stefan-Meyer Kahlen. *Description of the universal chess interface (UCI).* Apr. 2004. URL: https://web.archive.org/web/20190924182927/http://wbec-ridderkerk.nl/html/UCIProtocol.html (visited on 09/24/2019) (cit. on p. 15).

[55] Paul Lansky and Brad Garton. *RTcmix.* URL: http://rtcmix.org/ (visited on 08/21/2019) (cit. on pp. 20, 22).

[56] *Lomonosov Endgame Tablebases.* 2012. URL: https://chessok.com/?page_id=27966 (visited on 10/28/2019) (cit. on p. 15).

[57] Iain McCurdy and Joachim Heintz. *Csound.* 2015. URL: http://openweb.flossmanuals.net/files/csound.pdf (visited on 11/20/2019) (cit. on pp. 19, 20).

[58] Jakob Nielsen. *How to Conduct a Heuristic Evaluation.* 1995. URL: https://www.nngroup.com/articles/how-to-conduct-a-heuristic-evaluation/ (visited on 11/20/2019) (cit. on p. 56).

[59] Jakob Nielsen. *Severity Ratings for Usability Problems.* Nov. 1994. URL: https://www.nngroup.com/articles/how-to-rate-the-severity-of-usability-problems/ (visited on 11/14/2019) (cit. on p. 58).

[60] *python-chess documentation.* Nov. 2019. URL: https://python-chess.readthedocs.io/en/latest/index.html (visited on 11/12/2019) (cit. on p. 40).

[61] Tord Romstad. *The Art of Evaluation (long).* Aug. 2007. URL: http://www.talkchess.com/forum3/viewtopic.php?p=135133#p135133 (visited on 10/11/2019) (cit. on p. 29).

[62] Barry Vercoe. *The Canonical Csound Reference Manual.* Version 6.06. 2015. URL: http://www.csounds.com/manual/html/ (visited on 11/20/2019) (cit. on p. 19).