A Musical Chessboard

Alejandro Mayagoitia
alejandro.mayagoitia@yale.edu

Advisor: Scott Petersen
scott.petersen@yale.edu

A Senior Thesis as a partial fulfillment of requirements for the Bachelor of Science in Electrical Engineering and Computer Science

Department of Computer Science
Yale University
May 1, 2025

Acknowledgements

This project would not have been possible without the help of several people, including—but not limited to—the following:

My parents, Francisco and Isabel, for raising me to question and learn from the world around me.

My brother, Adrian, for building paths I followed, and always helping and making me laugh.

Professor Scott Petersen, for advising this project and introducing me to many of the tools I used to create it.

To Larry Wilen and all the CEID staff, for providing materials and advice twenty-four hours a day.

To MakeHaven, for their laser cutter.

To Silliman College, for being my home over the last four years and funding this project.

To Nora Hylton, for making life fun and staying by my side through many long nights.

To Hale, for helping me escape two nights a week.

To my family.

To my friends.

Thank you!

Contents

A Musical Chessboard

Alejandro Mayagoitia

Abstract

This project presents a novel interactive system that transforms the traditional game of chess into a dynamic auditory experience. By integrating a custom-built chessboard, the system detects real-time piece positions. Piece positions are detected by using photoresistors and LEDs to distinguish the color of every piece on the board—black or white. Using the color data, it is possible to infer each piece's type by comparing the current board state to a previous, known board state. Doing this analysis after every turn allows the system to know the state of the game at any time. The board state is further analyzed using Stockfish, a chess AI that provides win percentages and other game information. This data is processed through a music generation engine, resulting in a responsive musical accompaniment that reflects the evolving dynamics of the match. In addition to tempo and volume, the system maps piece location to harmonic structure and adjusts filter parameters based on game progression. These parameters make each match that is played musically distinct, keeping players engaged and adding replay value to the game. The music currently generated is inspired by Steve Reich's Piano Phase, although the same system could be used to create other pieces. Drawing inspiration from historical works like John Cage's *Reunions*, this implementation advances the concept by incorporating real-time strategic analysis to influence musical output. The fusion of tangible gameplay with algorithmic composition offers a multisensory experience, enhancing player engagement and exploring new intersections between game design, artificial intelligence, and generative art.

1 Introduction

Sounds have the power to transform ordinary experiences into moving and memorable events. A film may play a dark, somber tone when the villain is first introduced, or a cheery chord may ring when a player earns an achievement in a videogame. These sound design choices work because we associate emotions with different sounds even when they are not the central focus of a piece of art. Through sounds, visual and interactive media like film or videogames can add a layer of immersion for their audience, intensifying triumphant moments and deepening the impact of defeat.

Like films and videogames, board games have intense highs and depressing lows. But unlike videogames, it can be hard to feel the ebbs and flows of board games without being fully invested in the match. And, unfortunately, it can be harder to stay focused on board games than other interactive media. Board games often last several hours, fatiguing players as they strategize and play to win. Additionally, most board games are very technologically simple, so they have fewer bells and whistles to grasp players' attentions. Notably, board games cannot accompany the game action with sound, leaving a gap in their immersive potential.

If board games could successfully incorporate sound to immerse their players, they could become more appealing to newcomers and more engaging for audiences. This raises the question: Can a dynamic, game-state-driven soundtrack enhance the experience of playing a board game like chess?

This paper presents a chessboard design that detects the state of the game in real time, keeping track of each piece's location over the course of a standard chess game. In this instance, the design is tailored for music generation, but the same system could be used to fulfill other purposes. Since the system keeps track of the current game state, it could be used to stream a physical game online or project it for a wider audience to see, for example. For the purposes of this project, however, music generation is the main priority. That choice impacted which game state properties the system records and analyzes.

Several game state properties of the current match can be used to influence the musical events being generated. In particular, the music-making algorithm uses win-draw-loss probabilities, the physical location of each piece, and the number of turns to generate a soundtrack for the current match. This way, the music for each match can evolve over time in a way that can immerse audiences and players. Great moves may flip a player's expected outcome from a loss to a win, which should be reflected in the music they hear. Likewise, as the number of turns drags on, stakes may feel higher due to the time investment both players have put in. That time investment should be reflected with an increase in the soundtrack's intensity. By combining these parameters to create a product that is pleasing to the ear, the whole design brings new life to a game created over a thousand years ago.

2 Background

Chessboards that make music are not a new idea. The earliest major example of such an invention is Lowell Cross's work for *Reunions*, a 1968 performance by John Cage, Marcel Duchamp, and Teeny Duchamp [1]. Cross's design is purely analog and is essentially a giant filter. 16 inputs are connected to the board, all of which are attenuated according to the resistance values of a photoresistor array. Each photoresistor in the array is located at one of the 64 squares on the chessboard, so they can detect pieces based on the absence or presence of ambient light. The sonic events generated evolve with the game in the sense that different moves will affect the state of the filter, causing different audio outputs. This creates a few restrictions for the design that affects the music it generates. The overall structure of the music generated does not account for the individual significance of each move; a checkmate could be sonically indistinguishable from moving a pawn. Additionally, the photoresistors only react to the absence of ambient light, so any shadow induces a response from the filter. Therefore, the soundtrack that is generated is closer to background noise than an intentional piece meant to immerse the players and audience.

Since 1968, new versions of *Reunions* have been created, most notably an online version meant to be played at home on a computer or tablet [2] and a DIY kit to build a copy of the original board [3]. In addition to these innovations, there is a small community of hobbyists who have created soundtracks meant to score historical chess games, like the match between IBM's Deep Blue computer and world chess champion Gary Kasparov [4]. Similarly to the work of Cross, Cage and the Duchamps, these hobbyists try to connect chess and music. Unlike the generative music of *Reunions*, the music these hobbyists write is created strictly after the match is long over. Because of that, hobbyists can write music that is more fitting to each moment in the game, with changes in the score reflecting the events of the match. Unfortunately, this comes at the cost of the music feeling separate from the game itself, as they cannot exist at the same time.

This project aims to connect these two approaches. The music generated is relevant to the action in the match, and like *Reunions*, the board can generate sonic events in real-time as the game plays out. To achieve both of these things, the system uses a board that can detect occupied squares in a similar fashion to the chessboard Cross made for *Reunions*. But unlike Cross's board, this board is not fully analog. By using a microcontroller, it is possible to take the photoresistor data and infer the exact location of each piece. This information can be used to actually analyze the state of the game to figure out who is winning and the significance of each move. There are several AI tools that can analyze game states and determine the quality of possible moves very well, such as Stockfish [5]. A board that can accurately determine the game state at any given point and query a chess AI like Stockfish could determine the quality of each move being made in real-time. That data could then be used to create music fitting for each move of the game as it is being

played, rather than modifying existing inputs as in *Reunions*.

3 Methodology

The system consists of two main parts:

1 The piece-detecting chessboard

2 The music generation algorithm

Each of these components has a distinct structure and methodology behind their design, so they will be covered in two separate sections.

3.1 The Chessboard

The chessboard must first physically detect the location of each piece, and after that, it must turn that information into a format usable by the music algorithm.

3.1.1 Detecting Each Piece - Hardware

Like Cross's original chessboard for *Reunions*, this chessboard detects occupied and unoccupied squares using photoresistors. In order to go further than Cross's original design and identify the location of each piece, this chessboard uses the photoresistors on each square to detect each piece's color, black or white. This is done by placing an RGB LED alongside each square's photoresistor. The circuit under each square looks like this:

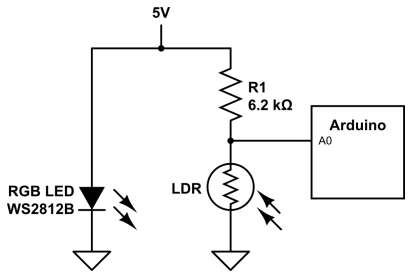

Figure 3.1: Basic circuit to detect color

The RGB LED shines red, then green, and then blue towards the chess piece. Afterwards, according to the color of the chess piece, each color of light emitted by the LED is absorbed or reflected at a different intensity. For example, if the chess piece is red, the red light emitted by the LED will be reflected, while the blue and green light are absorbed. These differences in the reflected light can be detected by the photoresistor. Then, by comparing the readings of the photoresistor to baseline values, the color of a material can

be detected. The baseline values used to detect each color are found by taking the average of many readings. In this case, the system took 100 readings for white pieces and 100 readings for black pieces for every photoresistor. To get the baseline values, the average of all these readings is calculated and stored for each photoresistor.

Every square has two holes in it, one for the photoresistor, and the other for the RGB LED. Therefore, from the top, the whole board looks like Figure 3.2 below.

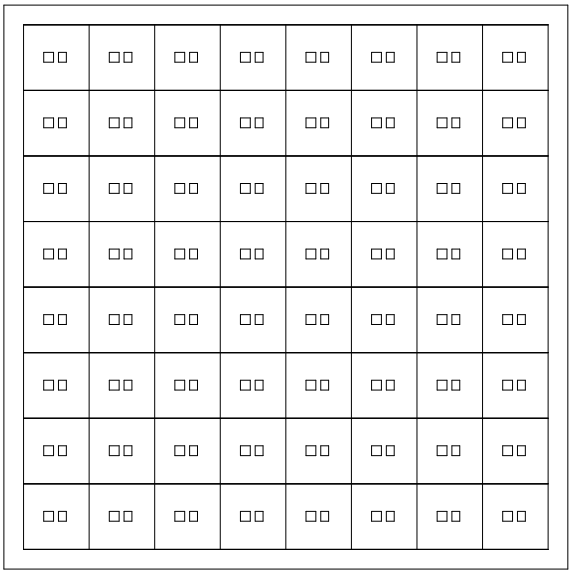

Figure 3.2: Top view of chessboard

The Arduino Mega 2560 Rev3, which is the microcontroller used in this design, only has 16 analog inputs, but there are 64 photoresistors that must be read [6]. In order to read all 64 photoresisors given the limited number of analog inputs, multiplexers are used. Instead of connecting each photoresistor to an analog input on the Arduino, each photoresistor connects to an analog input on a multiplexer, and the Arduino's analog input pins connect to the multiplexer's output pin. To read a photoresistor value, the microcontroller must communicate to the multiplexer which photoresistor is being read. This is done using a digital signal. Afterwards, the multiplexer outputs the voltage value at the photoresistor selected by the Arduino.

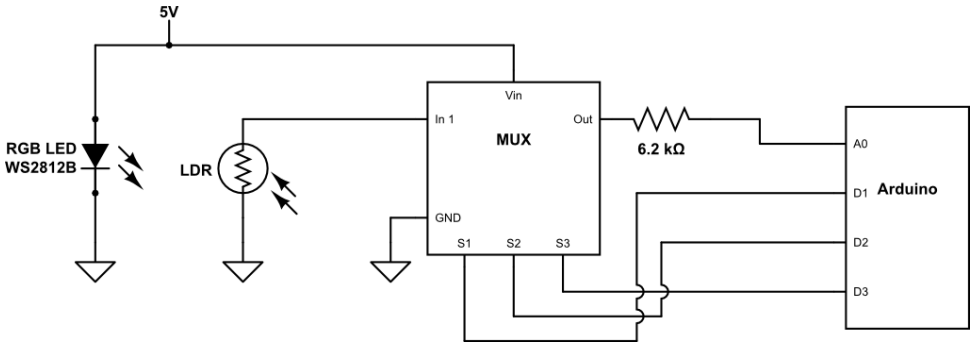

Figure 3.3: A single photoresistor connected to a multiplexer and communicating with the Arduino.

For this design, 8 multiplexers were used, each with 8 input pins. Each multiplexer corresponds to a row of 8 squares on the chessboard.

The last notable piece of hardware used in this design is a button used by players to indicate the end of their turn. Since the game state only changes at the end of each turn, this button is an easy way to know when the board must be read. All hardware components and their connections can be represented at a high level with the following diagram:

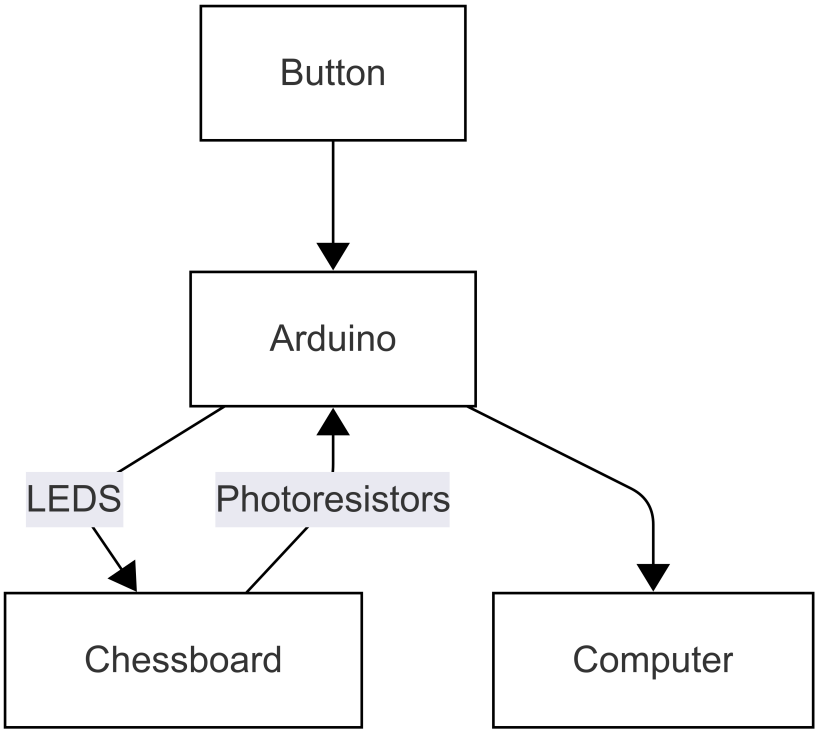

Figure 3.4: "Big picture" diagram of hardware setup

3.1.2 Detecting Each Piece - Software

There are 3 components that make up the software for this project:

1 The Arduino code, which outputs a matrix of detected colors on each square

2 A Python script that infers and analyzes the game state

3 A Max program to sonify the game state information

The Arduino Code

The Arduino code uses the photoresistors and LEDs to generate an output matrix. This output matrix represents open squares and occupied squares, including the piece color on each occupied square. Open squares are represented by a 0, black pieces are represented with a -1, and white pieces are represented by a 1. The row closest to the white player is treated as the first row, and the column to the white player's left is treated as the first column. So, for example, the output matrix for the initial board state in chess would look like this:

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{matrix}$$

Figure 3.5: Matrix representation of the initial board state

The white player moving a single pawn forward would generate the following matrix:

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{matrix}$$

Figure 3.6: White moves the pawn at e2 forward one square

If black retaliated by moving a knight, this would be the result:

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 & -1 & -1 & -1 & -1 \end{matrix}$$

Figure 3.7: Black responds with a knight move

And so as the match continues, the Arduino will update its output matrix accordingly.

On their own, these matrices do not convey much information about the game state. They tell us which player occupies each square, but that information alone is not enough to understand the state of the game. For example, the output matrix is not enough to understand a random game state in the middle of a match.

$$\begin{matrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 \\ -1 & 0 & -1 & -1 & 0 & 0 & -1 & -1 \\ 0 & -1 & 0 & 0 & -1 & -1 & -1 & 0 \end{matrix}$$

Figure 3.8: A possible mid-game output matrix

The matrix shown in Figure 3.8 could be a board state detected by the Arduino mid-game. From this matrix alone, it is impossible to conclude which piece is located where. It is impossible to know whether the furthest black piece up the board (located at row 3, column 4 or square d3 in chess terms) is a pawn that worked its way up the battlefield or a queen that shot across half the board. All one can say with certainty is that this piece is a black piece. The piece's actual identity is ambiguous.

The Python Code

The ambiguity disappears when considering that chess has a known starting position. Because the game has an unambiguous initial position and only one piece moves at a

time, we can track the matrix changes after each turn to identify each piece. Tracking the game changes is easily implemented using matrix subtraction. To show this, consider the first move shown in Figures 3.5-3.6.

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{matrix}$$

Matrix A

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & \mathbf{0} & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{matrix}$$

Matrix B

Figure 3.9: Matrix comparison before and after White's first move

Subtracting matrix A from matrix B clearly illustrates piece movement.

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{-1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

Figure 3.10: Matrix $B - A$

Looking at Figures 3.9 and 3.10, it is easy to see that a -1 in $B - A$ represents a piece's previous location while a 1 represents its new location. Therefore, if we know the initial game state looks like this:

$$
\begin{array}{cccccccc}
R & N & B & Q & K & B & N & R \\
P & P & P & P & P & P & P & P \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
p & p & p & p & p & p & p & p \\
r & n & b & q & k & b & n & r
\end{array}
$$

Figure 3.11: Initial Chessboard State with Capital Letters for White and Lowercase for Black

We can conclude that the board state after the move in Figure 3.6 would look like this:

$$
\begin{array}{cccccccc}
R & N & B & Q & K & B & N & R \\
P & P & P & P & 0 & P & P & P \\
0 & 0 & 0 & 0 & P & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
p & p & p & p & p & p & p & p \\
r & n & b & q & k & b & n & r
\end{array}
$$

Figure 3.12: Game state after moving the pawn in front of white's king

Similar logic applies for moving black pieces. The only in difference is that, for black pieces, the new location is marked by a -1 in $B - A$ and the old location is marked by a 1. This change is due to the assignment of -1 to black pieces and 1 to white pieces. Using this method, the system can track simple movement of one piece to any empty square. But some chess movements include multiple pieces. Namely, a player could capture an opponent's piece, perform an en passant capture, or castle their own king. All of these moves can be identified with the same matrix subtraction method, although there are slight changes in the implementation of each.

The Arduino output for a simple piece capture is shown below:

1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	**1**	0	0	0
0	0	0	**-1**	0	0	0	0
0	0	0	0	0	0	0	0
−1	−1	−1	0	−1	−1	−1	−1
−1	−1	−1	−1	−1	−1	−1	−1

Matrix A

1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	**0**	0	0	0
0	0	0	**1**	0	0	0	0
0	0	0	0	0	0	0	0
−1	−1	−1	0	−1	−1	−1	−1
−1	−1	−1	−1	−1	−1	−1	−1

Matrix B

Figure 3.13: Arduino output when white captures a piece (bold numbers used to highlight movement)

Subtracting these two matrices creates the following matrix:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	**-1**	0	0	0
0	0	0	**2**	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 3.14: Matrix $B - A$ showing white's capture of a black piece

When white captures a black piece, the white piece moves to the position of the black piece and the black piece leaves the board. That movement is represented in Figure 3.13. When subtracting the two matrices associated with the board, -1 is subtracted from 1 at the new location. The result is that the white piece's new location contains a 2 while the old contains a -1, as seen in Figure 3.14. The signs are flipped when black captures a white piece.

Whenever any piece is captured, the capturing piece takes the board position of the captured piece — except with an en passant capture. An en passant capture is a type of capture performed by a pawn. It gets different treatment because the capturing piece moves *behind* the captured piece's location rather than taking the same location. The Arduino output for such a capture is shown below:

1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	**-1**	**1**	0	0	0
0	0	0	0	0	0	0	0
−1	−1	−1	0	−1	−1	−1	−1
−1	−1	−1	−1	−1	−1	−1	−1

Matrix A

1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	**0**	**0**	0	0	0
0	0	0	**1**	0	0	0	0
−1	−1	−1	0	−1	−1	−1	−1
−1	−1	−1	−1	−1	−1	−1	−1

Matrix B

Figure 3.15: Arduino output of an en passant capture by white (bold numbers highlight movement)

Subtracting matrix A from matrix B creates the following matrix:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	**-1**	0	0	0
0	0	0	**1**	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 3.16: Matrix subtraction showing white's en passant capture (bold numbers show correct movement)

The result of the matrix subtraction in the en passant case looks much different than during normal capture moves. In normal capture moves, the capturing piece's new location is denoted with a 2 or -2. But the en passant case for white has no 2. Instead, there are two 1s and a single -1. At first, this seems like there are two destinations for the white piece, which is impossible. However, the confusion is dispelled when one considers that pawns only move forward. Consequently, the destination of the capturing pawn cannot be on or behind its current location. This means that we can distinguish which of the two 1s in $B - A$ corresponds to the new destination, as it must be the square that is furthest forward—toward higher index rows for white pieces and toward lower index rows for black pieces.

The last major movement type is castling. Castling allows the current player to protect their king using a rook, and it is the only move in chess where players move two pieces at once. Below is a representation of white performing a "kingside" castle:

R	0	0	0	**K**	0	0	**R**
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
r	0	0	0	k	0	0	r

Before Castling

R	0	0	0	**0**	**R**	**K**	**0**
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
r	0	0	0	k	0	0	r

After Kingside Castling

Figure 3.17: Kingside castling for white, showing only the king and rook

Below is what the Arduino output for such a move would look like:

1	0	0	0	**1**	0	0	**1**
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-1	0	0	0	-1	0	0	-1

Matrix A

1	0	0	0	**0**	**1**	**1**	**0**
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-1	0	0	0	-1	0	0	-1

Matrix B

Figure 3.18: Kingside castling using 1 for white and -1 for black

After subtracting A from B, you get the following matrix:

0	0	0	0	**-1**	**1**	**1**	**-1**
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 3.19: Matrix result of $B - A$, highlighting the change due to kingside castling

At a first glance, this looks confusing to interpret. There are two "from" positions and two "to" positions, and there is no way of telling how each piece moved from the matrix subtraction alone. But castling has very specific rules; it only ever involves the king and rook, and the pieces only ever move to and from the same locations. Therefore, if two pieces ever move, it is possible to know how those pieces moved because it must be castling.

It is also possible to castle on the other side of the king than what is shown in Figure 3.18. This is known as "queenside" castling. Recognizing and dealing with queenside castling does not change any of the program logic, so it will not be illustrated here.
With the heuristics described here, it is possible to keep track of every piece's location on the board. To completely store the game state, the python program stores all of the following data after every turn:

1 The color of the pieces on every square, as outputted by the Arduino program.

2 The identity of every piece on every square, as shown in Figure 3.11 and 3.12.

3 The total number of moves made

4 The castling availability (castling can only be done once per game, using rooks and kings that have not moved)

5 How many moves have passed since the last piece was captured

The final step of the python program is to analyze the game state. This is done by querying Stockfish, an AI tool focused on analyzing chess games. To query the API, the python program uses chess-api.com., which returns win probabilities for each player [7].

3.2 The Music Algorithm

The ultimate goal of the board is to generate music to accompany the game being played. To that end, I created a patch using Max 9, a music programming language. This patch generates a piece based on the game information gathered by the python program and Arduino. The generated piece is based on Steve Reich's "Piano Phase." Reich's original piece is for two pianos. Both pianos begin with the same repeating pattern at the same time. As the piece continues, the pianos begin playing at slightly different tempos, and get "out of phase" The uneven change in tempo creates interesting and satisfying patterns to listen to.

This piece felt fitting to accompany a game of chess because the piece follows a similar arch to a chess match. Both players start in the same place but slowly drift apart from each other while trying to reach the same goal.

The generated piece begins with two instruments playing the same repeating pattern from "Piano Phase." One of the two instruments corresponds to the black player and the other corresponds to the white player. Like the original Steve Reich piece, the point is that the two patterns will get out of sync, but for this piece, the tempo of each patterns is tied to each player's win probabilities. As the win probabilities differ, the black player's

instrument begins slowing down, getting out of sync with the white player. Along with the tempo, the volume of each player's signal is tied to their win probabilities, with the winning player receiving an amplitude boost and the losing player receiving an amplitude attenuation.

Lastly, the timbre of each player's instrument varies based on the moves they make. Every column represents one of the first 8 harmonics of a fundamental frequency. As pieces move up the columns for each player, the amplitude of each harmonic is raised or lowered depending on the location of the furthest piece up the board on that column. Additionally, at the start of the game, there is a low-pass filter with a cutoff frequency of 1000Hz on each player's signal. As the game goes on, this cutoff frequency is increased exponentially based on the number of moves that have been made.

Because all of these values change with each move, the music algorithm just continuously listens for output from the python program by reading a text file.

4 Parts List

Item	Quantity	Unit Price
Photoresistors	64	∼$0.27
RGB LEDs[8]	64	∼ $0.14
Arduino Mega 2560 Rev3[6]	1	$60.00
Ribbon Cable	30 ft	∼$10.00
8-to-1 Analog Multiplexers[9]	8	$7.99
Breadboards/Protoboards	4	$10.00
Acrylic for the top	–	∼$45.00
Wood for the base	–	∼$25.00

Table 4.1: Component list and unit prices for the build.

5 Related Work

Piece-detecting are not a new concept. Big chess tournaments have been using electronic boards that detect pieces for several years, helping them stream matches to an online audience. One of the biggest manufacturers of these boards is Digital Game Technology (DGT), who hold a patent for their board's design [10]. Their board uses electromagnetic induction to detect pieces. The board is fitted with electronics that react to coils hidden inside each piece. These interactions are then used to detect piece locations, similarly to how this board design uses color detection. Their systems are quite sleek and elegant, although they are typically a few hundred dollars.

Like piece-detecting chessboards, algorithmic music composition also has an established history. John Cage, besides making *Reunions*, one of the original inspirations for this project, also pioneered other forms of generative and stochastic music composition [11]. One of his most famous pieces is the *Music of Changes*, which uses a Chinese classic text typically used for divination to inform musical events. These early experiments highlight the potential for systems that respond to external or unpredictable inputs, an idea that underpins the music generation component of the musical chessboard.

6 Conclusion

This project set out to explore whether a dynamic, game-state-driven soundtrack could enhance the experience of playing a traditional board game like chess. By designing a chessboard capable of detecting piece locations and colors using photoresistors and RGB LEDs, and by integrating this information with a music generation system, the project demonstrates that it is possible to sonify the flow of a match in real time.

Unlike previous systems such as Cross's custom chessboard built for *Reunions*, this system actively reacts to in-game events. Rather than passively responding to changes in ambient lighting, this system analyzes the significance of each move. This helps the musical component react to each move as well as the evolving drama of the match with changes in tempo and timbre.

The key contribution of this work lies in its successful merging piece-detecting chessboards with generative music composition into a unified system that enhances the emotional engagement of board game play. By bridging the gap between sound design, physical computing, and strategic gameplay, the musical chessboard offers a new way to experience a centuries-old game, suggesting broader possibilities for using sound to enrich physical interactive experiences.

Knowing the result of this build, however, the piece-detecting method should change for future iterations. Color detection using photoresistors and LEDs is not reliable enough to use this design in more serious contexts. Depending on the nature of the ambient light, the photoresistors used were prone to detecting false positives—indicating a square was covered when it was in fact empty. These failures were not common enough to discredit the project as a proof of concept, but they hindered the user experience.

7 Future Work

Although the board works, there are ways to improve it. The most obvious pitfall of the board is in its presentation. Unfortunately, there was not enough wood available to complete the full frame. This leaves much of the wiring exposed, which adds an eye-catching aesthetic, but also makes the board susceptible to whatever environment it is placed in. Additionally, the user experience could be improved. At the moment, the board is not very user-friendly. Creating a GUI would make the board more pleasant to use, and it would make it easy to stop any piece-detection failures.

Piece-detection failures happen whenever the board reads a move incorrectly. Generally, the board reads moves accurately more often than not, but failures have long-term consequences. When any move is read incorrectly, the error propagates to all following moves. One error may disrupt the entire piece, which makes catching piece-detection errors important. Having some sort of GUI to help the players see what the board detects would help players catch detection errors quickly.

To improve on this design, the presentation, user interface, and piece-detecting methods would need improvement. Future iterations may not use photoresistors for piece-detection, or they may have a GUI allowing users to set the board to calibration mode, music mode, or display mode. Whatever changes are made, it is a certainty that this design could be improved by refining its existing features and adding some new functionality.

Bibliography

[1] Lowell Cross. “Reunion: John Cage, Marcel Duchamp, Electronic Music and Chess”. In: *Leonardo Music Journal* (1999). URL: https://www.johncage.org/blog/cross_reunion.pdf.

[2] Ian Miguel Chris Jeferson. *Reunions Essay*. 2013. URL: https://johncage.org/reunion/digital_reunion_essay.html.

[3] Accesed on 03-07-2025. URL: https://sonicartefacts.com/product/sonic-chessboard/.

[4] Archil Leon. URL: https://www.youtube.com/watch?v=kyEEEkoGDgo.

[5] Alex Turk Shiva Maharaj Nick Polson. “Chess AI: Competing Paradigms for Machine Intelligence”. In: *Entropy* (2022). URL: https://doi.org/10.3390/e24040550.

[6] Arduino. *Arduino Mega 2560 Rev3 User Manual*. Accessed on 03-20-2025. URL: https://docs.arduino.cc/resources/datasheets/A000067-datasheet.pdf.

[7] ChrisC. *Chess-API*. URL: https://chess-api.com.

[8] *WS2812B Intelligent Control LED Integrated Light Source*. Worldsemi. URL: https://cdn-shop.adafruit.com/datasheets/WS2812B.pdf.

[9] *CD405xB CMOS Single 8-Channel Analog Multiplexer or Demultiplexer With Logic-Level Conversion*. Texas Instruments. URL: https://www.ti.com/lit/ds/symlink/cd4051b.pdf?ts=1745428525201&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FCD4051B.

[10] Bernard Johan Bulsink. *Device for detecting playing pieces on a board*. DGT Projects BV, 2001.

[11] James W. Pritchett. “From Choice to Chance: John Cage’s Concerto for Prepared Piano”. In: *Perspectives of New Music* (1988).