Final Report: Player Piano Project

Camden Hunt, B.S. Computer Science

Eric Golde, B.A. Computer Science

Advisor: Jeff Caley

CS 499, 2023-24

Abstract

The Player Piano project embodies the integration of technological advancement and creative expression, seeking to transform a conventional upright piano into a fully automated player piano capable of flawlessly playing MIDI files. This ambitious project employs an array of solenoids paired with a microcontroller to precisely activate piano keys in perfect alignment with digitally encoded tunes. The platform provides enthusiasts with an immersive exploration of the piano's capabilities through a customized website, offering access to a carefully curated assortment of musical compositions. The piano functionality is programmed using Java, while the website structure is developed with HTML, CSS, and JavaScript. This project, rich in electrical engineering details, posed a significant challenge, highlighting both its technical sophistication and artistic innovation.

Table of Contents

List of Figures

1. Introduction

Creating a self-playing piano perfectly captivates our group's enthusiasm due to its challenging blend of hardware and software intricacies. Beyond merely pushing the boundaries of our collective knowledge, this project promises a rich learning experience that intertwines various technical facets. A primary objective is to elevate the piano's capabilities to play compositions beyond the scope of human pianists, showcasing our interest in both hardware and software realms and their integration. This is accomplished by curating a database of preloaded songs for the user to choose from on our website, as well as creating the infrastructure to translate these requests into electrical signals to be sent to our custom-designed hardware components.
All in all, while the practical utility of this endeavor may be limited, its value as a novelty is substantial, and its academic merit is immense. The project provides a unique opportunity for comprehensive learning, driven by the fusion of hardware and software components.

1.1 Functional Objectives

There are several functional goals in mind for the project to be considered a success.

1. Design and install hardware components to automate the playback of MIDI files on a physical piano.
    a. The hardware should be easy to maintain or replace by us or others.
2. Create a web application to interface with the piano on iOS, Android, and Windows devices.
3. Curate a database of preloaded songs for the users to queue on the piano containing info such as length and number of notes as well as classifiers such as title and artist.

1.2 Education Goals

Our project serves as a multifaceted educational endeavor, providing a platform for hands-on exploration and skill development across several fields. The integration of hardware and software components offers a holistic learning experience, cultivating expertise in key areas that are not only pertinent to this project but also highly sought after in various professional fields.

1.2.1 Hardware and Software Integration

Successful implementation requires seamless collaboration between hardware and software components. This integration skill is crucial to embedded systems, where optimizing the interaction between physical devices and software is fundamental.

1.2.2 Web Development

The envisaged website for song selection and interaction introduces us to web development. Acquiring proficiency in web technologies broadens our skill set, directly applicable to careers in front-end and back-end development.

1.2.3 Hardware Design

The project necessitates a profound understanding of hardware design, enabling us to conceptualize, develop, and optimize the electronic circuits governing the piano's automation. Familiarizing ourselves with PCB design equips us with a skill set crucial for careers in electrical engineering and product development.

1.2.4 Summary

By pursuing these educational goals, our group not only aims to create a captivating musical experience but also lays a robust foundation for diverse career paths. The skills acquired in hardware and software integrations, web development, and hardware design, are transferable and applicable to a wide array of industries, positioning us for success in the evolving landscape of technology and innovation.

2. Requirements

2.1 User Stories

- As a PLU student, I want to be able to start a song on the self-playing piano, so I can enjoy a musical experience without having to play the piano.
- As a music enthusiast, I want to browse and select songs from a database on the piano's website so that I can enjoy a wide variety of music on the piano.
- As a professor at PLU, I want the hardware to be modular and well-documented so I can repair the piano if it stops working.

2.2 System Stories

The user selects a song from the front-end website. The website sends a request to the backend software to play the song using an HTTP POST request. The backend software reads the MIDI song file, and decodes and processes it. It then sends data to the microprocessor that controls the piano hardware, and any other enabled outputs. The microprocessor on the physical piano decodes that data and controls the electrical signals that cause the keys on the piano to play.

2.3 Hardware Requirements

For the hardware side, each key needs to be individually controlled. The force or velocity of each key should be controllable independently. The sustain pedal should be able to be pressed and released. The back-end control computer should control all of these functions.

2.4 Software Requirements

We decided to develop the website using vanilla HTML, CSS, and Javascript. This decision was primarily made because it would not require learning new languages, but was still difficult enough that it would serve as a beneficial educational experience. We briefly considered using Typescript or a new framework like React or Vue. Still, we decided against it as it would make the project significantly harder given our lack of experience. A small shared Javascript class served as the main communication between the website and the backend Java server. This supported many UIs that share the same communication standard. GET and POST HTTP

requests are used to retrieve static data and send execute commands, as well as WebSockets for live communication like notes being played, time updates, or statistic changes.

The entire backend, from the web server to MIDI parsing, to several programs for debugging and testing, were all written in Java. Java was chosen because it was a familiar language supported on a wide range of platforms. It can run on Windows and Linux, the primary platforms on which we were concerned with running the backend software.

For user testing, the piano backend is hosted on a server owned by Eric; it had a reverse proxy to have the URL https://piano.ericshome.xyz/. On this website, the physical piano output was disabled as its purpose was to test UI elements and interactions with multiple users.
Our database was created in JSON. This was done because JSON is easy to edit with software already being used for other parts of the code, and there is plenty of support for parsing in Java, which our backend is written in. However, as the database grows, dedicated database management systems like MySQL become increasingly appealing.

To help manage our database, a GitHub Actions bot was created that verifies and edits the JSON entries. The bot parses through each song and adds the length and note count fields, as well as ensures each song has valid paths to the required files (album art and MIDI data)

3. Software Design

3.1 Front-end Design

At the beginning of the project, our goal was to create a singular application to be run locally on a laptop, with users physically interacting with that laptop. However, as the project progressed, we shifted to a web-based approach. This change was made to minimize physical interaction with the piano, making it easier to accommodate multiple users simultaneously and allowing the piano to be placed behind a barrier or wall without losing functionality. Consequently, we needed to develop a website compatible with various platforms, including mobile devices and desktop computers.

Transitioning to a web-based platform allowed us to implement remote access and control features, enabling users to interact with the piano from different locations. This flexibility expanded the project's scope beyond a local application, offering more accessible and versatile usage scenarios. It also required us to ensure reliable performance across different networks and devices.

3.1.1 Website Development

This project might seem overwhelming to new users unfamiliar with its intricacies. To address this, we focused on designing a website that is both highly functional and remarkably simple to use. Our goal was to ensure intuitive navigation while offering robust features. We took inspiration from services like Spotify and Apple Music, incorporating original UI elements tailored specifically for the player piano to create a user-friendly experience.

When users load the website on their device, they are presented with one of two pages: either the mobile-optimized website or the desktop-optimized website. The mobile-optimized version is designed with touchscreen devices in mind, featuring large, easily-tapped icons and streamlined menus. This layout ensures that essential functions are accessible with minimal scrolling and tapping, making it effortless for users to explore and interact with the piano's features on smaller screens.

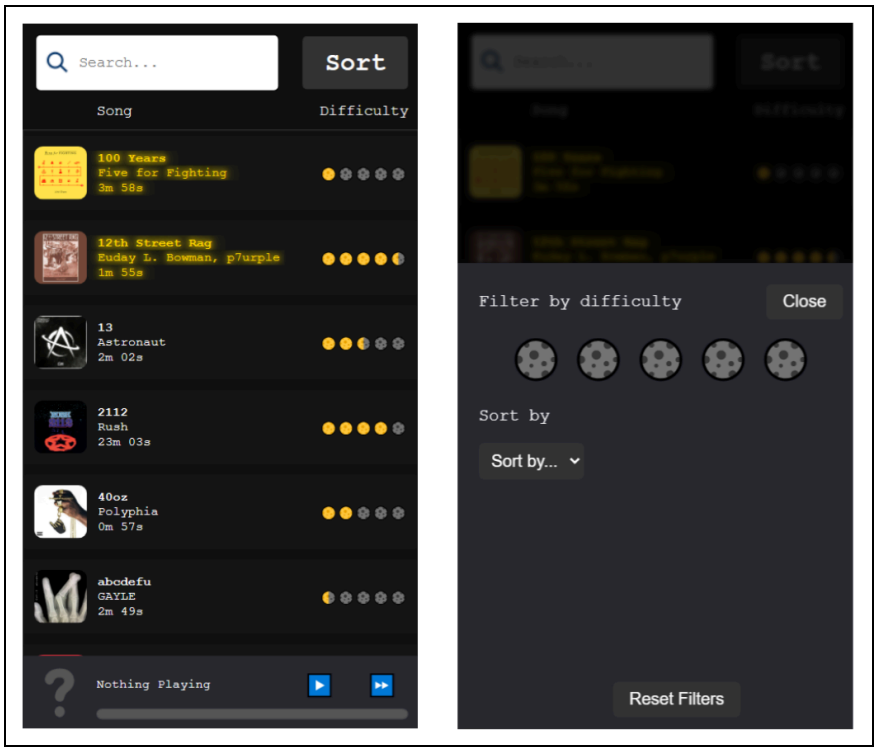

Figure 1: Mobile Website

In contrast, the desktop-optimized website takes full advantage of larger screens to provide a more detailed and expansive interface. The design maintains a clean and uncluttered appearance so every option is intuitive to find and use.

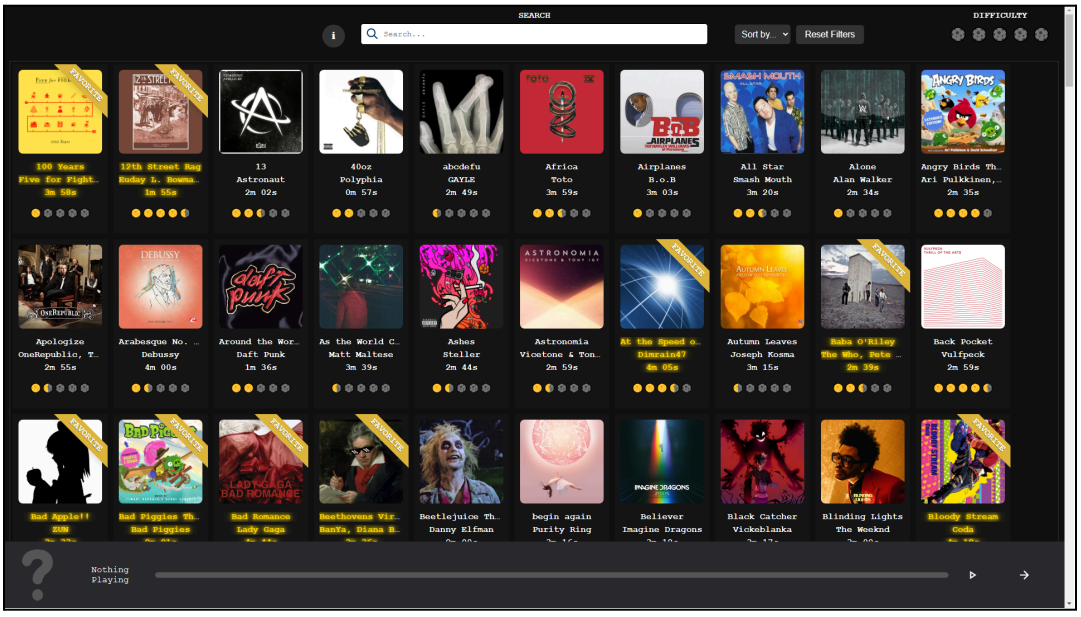

Figure 2: Desktop Website

3.1.2 Song Database

The songs are stored in a JSON list, as described in section 2.4. A critical part of the project is maintaining a comprehensive database of songs for the piano. Each song is hand-picked, so manually inputting hundreds of songs would be incredibly time-consuming. To streamline this process, GitHub Actions was used, as detailed in section 3.4.5, to automate the addition and updating of songs in our database.

name	The title of the song.
artists	An array of artists who performed or composed the song.
midiFile	The filename of the MIDI file associated with the song.
artwork	The filename of the artwork (cover art) associated with the song.
genre	An array that includes the genre(s) of the song.
tags	Additional keywords that can help users find the song through searches.
favorite	A boolean field indicating if the song is marked as a favorite.
songLengthMS	The length of the song in milliseconds.
noteCount	The number of notes in the song.

Figure 3: Database Entry

Genre, *songLengthMS*, and *noteCount* are auto-generated when the database is updated. *Genre* is autogenerated by a separate program using the Discogs API that was never fully embedded into our code base. *songLengthMS* and *noteCount* are used for calculating difficulty and are generated by the songDB subprogram detailed in section 3.4.5.

3.2 MIDI Files

MIDI files are the most common way to distribute computerized sheet music. It is a standard dating back to the early 1980s. Because almost all music is distributed in this form, it was the natural format for a project like this.

3.2.1 Sheet Music & MIDI parsing

The `SheetMusic` object is the object that represents a MIDI song in memory. It records which events and when to broadcast them. When defining what a `SheetMusic` object was, it was important to create something simple to understand, yet expandable for future use. The structure for storing notes was: `Map<Long, List<SheetMusicEvent>>`. The `Long` is used as a value of milliseconds since the start of the song. This structure allowed us to, at a specific point in time, have a list of notes or other events that happened at that time. Each inner list of `SheetMusicEvent` stores which notes change at a particular time (i.e., pressed or released). Only the changes of notes are stored, and not 88 keys every single time segment.

MIDI files define time in a few different ways. MIDI has time signature events, typically used to display the time signature, but are one way to determine timing. An alternative way is using MIDI ticks. The MIDI standard has an event for what we call `ticks_per_beat`, which is the number of ticks per 'beat'. The MIDI standard also contains a `tempo` event to track the tempo throughout a song. Unfortunately, tempo events proved incredibly hard to use, as songs are defined as a list of notes at a specific point in time. After a lot of troubleshooting, we developed two formulas to convert a MIDI event to a location in microseconds. In reality, the microseconds were converted to milliseconds, because such a high resolution was unnecessary.

```
private long midiTickToMS(float ticksPerSecond, long tick) {
    return (long) ( (1E6 / ticksPerSecond) * tick);
}

private float calcTicksPerSecond(float divisionType, int resolution, float tempoScale) {

    if(divisionType == PPQ){
        return resolution * (tempoScale / 60.0f);
    }
    else {
        float framesPerSecond = 29.97f;

        if(divisionType == SMPTE_24){ framesPerSecond = 24;}
        else if(divisionType == SMPTE_25){ framesPerSecond = 25; }
        else if(divisionType == SMPTE_30){ framesPerSecond = 30; }

        return resolution * framesPerSecond;
    }
}
```

Figure 4: Ticks Per Second Algorithm

3.2.2 Events and Note Handling

In our `SheetMusic` class, there is a map from long to list of `SheetMusicEvent`. Two classes extend `SheetMusicEvent: NoteEvent` and `SustainPedalEvent`. The `SustainPedalEvent` consists of a boolean, defining if the pedal is on or off. The `NoteEvent` is an event that stores three things: the MIDI note number, if it's on or off, and the velocity (how hard the key is pressed).

3.2.3 MIDI Cleaning

MIDI files are inherently designed for digital instruments, which lack physical constraints. However, our project operates in the physical realm, necessitating adjustments to the MIDI files to ensure playability. The first step in this process involves filtering out instruments that do not translate well to the piano, such as drums, leaving us with a file containing only piano notes. Next, any notes outside the piano's range are eliminated by simply deleting them, as notes in these ranges are rarely critical to the song's integrity, rendering a more complex solution unnecessary.

Another crucial adjustment involves accommodating the piano's physical limitations, particularly the speed at which it can play repeated notes on the same key. Digital instruments can handle rapid note sequences without issue, but the piano requires time for each key to return to its original position before it can be struck again. To address this, the duration of the first note in such sequences is shortened, ensuring sufficient time between notes for the keys to reset, thus enabling smooth and accurate playback.

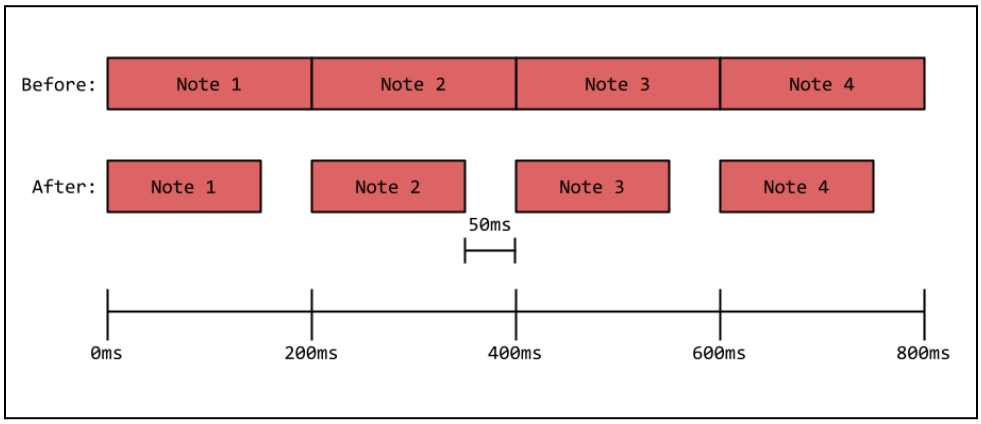

Figure 5: Note Shortening Algorithm

3.3 Event Outputs

When each event occurs, there is a set of outputs. These outputs can be configured in the outputs.json config file. This file allows us to easily change settings about each output, if it's enabled, without needing to recompile the code base.

3.3.1 Arduino Output & Note Mapping

The Arduino output module was designed to take the event data emitted by the sheet music object and convert it to the serial packet format specified in section 3.6. Due to multiple packet types for sending notes, determining which packet would result in the least amount of data sent over the wire was necessary. Instead of implementing a complex algorithm to calculate this dynamically, a simpler approach was used: generating all possible note packets and selecting the one with the least amount of data. While this method may not be the most efficient in terms of computational resources, the time it takes to generate these packets is negligible, resulting in no noticeable speed decrease.

Before transmitting these packets, we need to verify the notes sent to the piano are in the correct format. In the piano code, notes are represented by integers ranging from 0 to 87. In MIDI, however, notes are represented by integers ranging from 21 to 108, as MIDI supports notes outside the range of the piano.

3.3.2 Output Virtual Synthesizer

The Output Virtual Synthesizer output was primarily used before the physical piano was built, often in conjunction with the Output Piano GUI described in section 3.3.3. This plugin utilizes PC speakers to generate a virtual piano sound whenever a note is played. However, it is important to note that this feature is not enabled in our production environment.

3.3.3 Output Piano GUI

The Output Piano GUI is an on-screen virtual piano display that visually represents which keys are being pressed. This is vital for debugging purposes, allowing us to verify notes are being translated and played correctly. It was particularly useful during development sessions when

working on the backend code without access to the physical piano. While this feature is not included in the production environment, it can serve as a visually engaging addition if desired.

3.3.4 Output Logger

The Output Logger is a straightforward tool that prints every event to the console and color-codes the data for easy comprehension at a glance. This output is particularly useful when diagnosing issues or testing new cleaning algorithms. While this feature is seldom enabled in the production environment, it was a valuable tool during the development and testing phases.

3.4 Subprograms

When creating the backend control program with the website and multiple outputs, we quickly realized there were other features we wanted to add for debugging in order to ease the development process. One possible approach to this would be to split the codebase into multiple Java programs, each sharing specific modules (like MIDI parsing or error logging). While this idea has benefits, it introduces a variety of issues with version control. To work around this, a type of class called a `SubProgram` was created. These classes contain any feature we wanted to extend to several different parts of the project. Every subprogram contains a getter function which returns the name of the subprogram. For example, the key tester program was labeled “key-tester” and the backend server was labeled “run-server”. Each subprogram has access to the same MIDI parser and sheet music, as well as all the outputs of those programs. This allowed us to develop our GitHub Actions bot for parsing our database, the key tester program, and a program to generate key mapping configurations easily.

3.4.1 Javalin Server Backend

Javalin was used for the main HTTP server, because it was familiar to us, but was still worthwhile to learn more about. The Javalin server handles all communication between the website clients, and the main queue system. For communication between the web client and the backend server, HTTP GET and POST requests are used, while WebSockets is used to send live data back to the clients. GET and POST requests are used to verify each user separately and have information on which users queue specific songs. This approach also allows a user with administrative permissions (such as a professor or other PLU faculty) to access a panel to control

different aspects of the piano, like the volume or turn it on or off. Currently, there is a very basic session system for users, but no authentication is checked.

For our final presentation, it was important to have a seamless demo where the piano could interact with us and the presentation. Initially, our idea was to have our piano-controlling software be able to identify what point we were at in the presentation, and automatically play notes when we reached certain points in the presentation. We created a quick proof of concept Google extension that reports Google Slides data to the main controller. While this worked, we were worried that if Google updated Google Slides the day before, it would break our demo. We settled by creating a few endpoints and a straightforward mobile website with buttons that would trigger different actions.

3.4.2 Demo Website

As an aid for our presentation, a mobile application to send specific packets of data to the piano was created. The user interface is simple and intended to be used with minimal effort so that the presenters can focus on the audience. The first button demonstrates sending a singular packet of data to the piano, and the second button sends two notes. The third button plays "Fantaisie-Impromptu" by Frédéric Chopin, a piece chosen as the first of our demo songs due to its technical complexity and recognizability. The fourth button plays "Ghosts 'n Goblins" by Ayako Mori, a piece composed for the 1985 video game by the same name. This song was chosen because the piano performs songs in the ragtime style exceptionally well, and this song expresses that succinctly. The final button on the website pauses all data being sent to the piano.

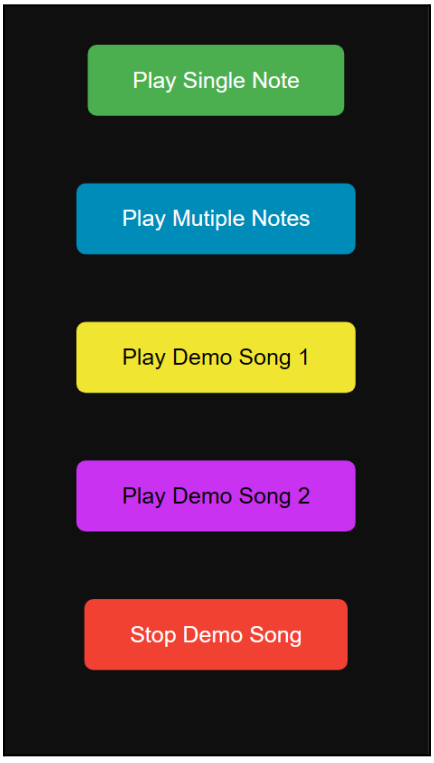

Figure 6: Demo Website

3.4.3 Key Tester

To facilitate easy calibration of the algorithm detailed in section 3.2.3, an application was created where specific packets can be sent to individual keys at preset intervals. By slowly increasing the speed at which packets were sent to a key, we were able to determine the maximum speed at which notes could reliably hit each key, and calibrate the algorithm accordingly. Unfortunately, this speed varies wildly across each key due to the age of the piano, varying anywhere between 25 milliseconds, and 130 milliseconds.

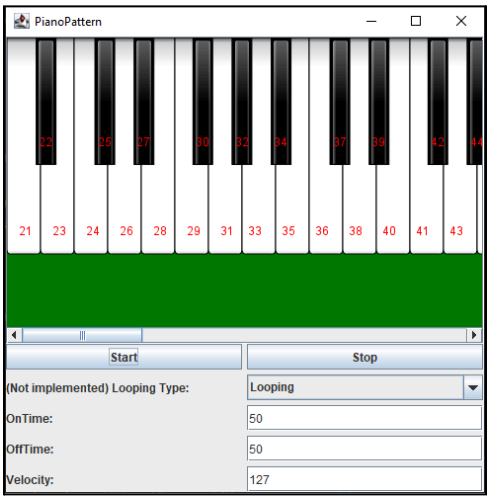

Figure 7: Key Tester Application

3.4.4 Key Config Generator

Due to the way that the solenoids are mounted, the solenoids are not wired in the exact consecutive order of the keys. For example, the black keys are controlled by different boards than the white keys, while in MIDI those keys are in normal order. This means there needs to be a mapping between the MIDI key code and the wiring order of the solenoids. To solve this, we created a Key Config Generator, which sends data to keys on the piano while prompting the user on a virtual keyboard GUI to click the key it played. After this is done for every key, the data is exported as a JSON file and placed in the config folder.

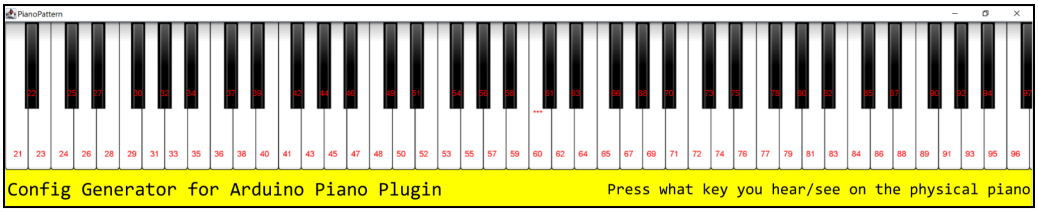

Figure 8: Key Config Generator

3.4.5 SongDBSubProgram

Adding a large amount of songs to the database proved to be a tedious and time-consuming task. To aid in this, a program was created that would add certain fields automatically. This program looks at the SongDB file, and verifies the following:

- The song name is unique and formatted correctly
- The MIDI file path is valid
- The album artwork path is valid
- Adds missing fields with default values, like genre, tags, artists, or favorite.
- Auto-generates note count and song length

In addition to this, the file organizes each entry in the database alphabetically to make it easier to find specific entries.

3.4.6 MidiIn (Unused)

This program is written for personal amusement fun and potential use as a demo. It reads MIDI input from either a physical MIDI instrument or a virtual instrument in a digital audio workspace and plays those notes on the physical piano. While this demo worked, it stayed largely unused.

3.4.7 MidiViewer (Unused)

MidiViewer is a subprogram written to verify that MIDI files are being parsed correctly. This program was never used more than a few times but may become useful in the future should any new issues arise.

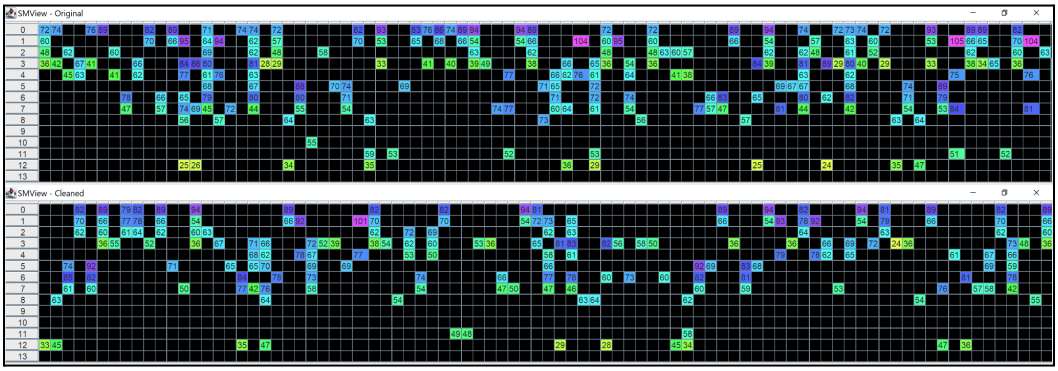

Figure 9: Midi Viewer

3.5 Arduino Software

Arduino is a framework for writing C and C++ code tailored for embedded systems. It provides a range of libraries, both built-in and user-created, designed to control various hardware components. For example, Arduino includes a built-in serial object for communicating serial data over the USB port.

To control the velocity of the solenoids, a technique called Pulse Width Modulation (PWM) was employed. PWM involves rapidly modulating the power on and off at a high frequency, effectively reducing the average power supplied to the device. PWM was used to adjust the strength of the solenoids, allowing us to vary the force with which they strike, from soft to hard impacts. This capability is crucial for replicating the dynamics and expression needed in certain songs.

The ShiftRegister-PWM-Library was used, modifying it with an alternative PWM algorithm and support for our error-checker board. The ServoTimer2 library was also used to control the servo. The Arduino framework has a built-in way to control Servos, but it interferes with the PWM functionality of the Shift Register library, so an alternative library was needed.

3.6 Packet Designs

When creating the format for the Serial Packets, it was important to create an easily expandable format. There are multiple Packet types. The packet IDs are as follows:

N	Generic **N**ote packet
B	**B**atch Note Packet
M	**M**ultiple Note Packet
S	**S**ustain Packet
P	Reset **P**ower Packet
O	**O**ff packet (Reset)

Figure 10: Packet Names

For example, the N packet is a list of notes and the corresponding velocities. If the velocity is 0, the note is turned off.

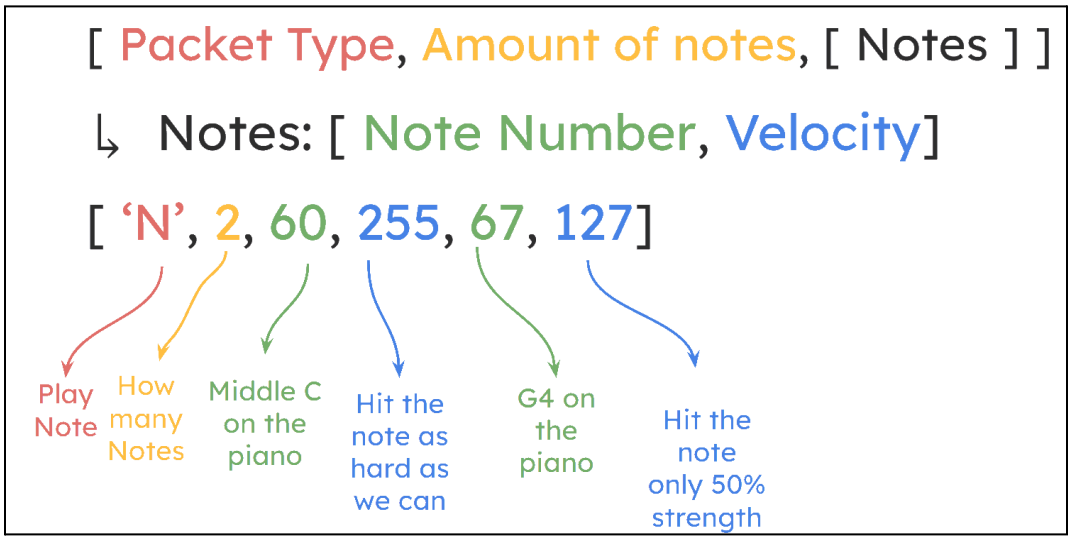

Figure 11: N Packet

To save the information sent, the M (multiple note) and B (batch) packets are used. The B packet is primarily used when a large number of continuous notes are played at once.

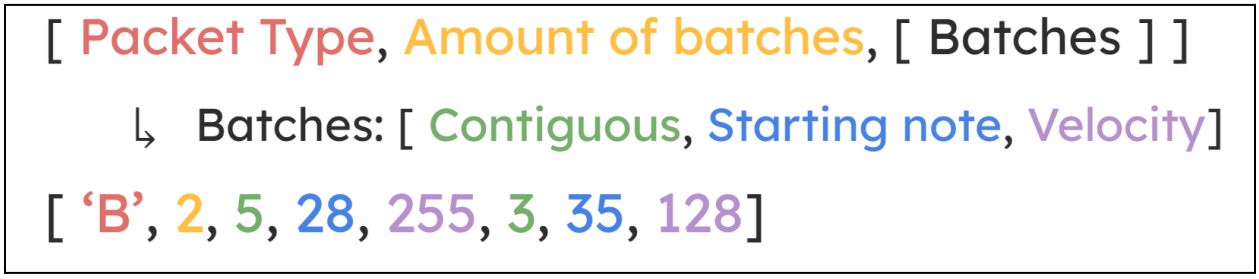

Figure 12: B Packet

The M packet is another type of packet for sending notes. This is to send multiple notes with the same velocity.

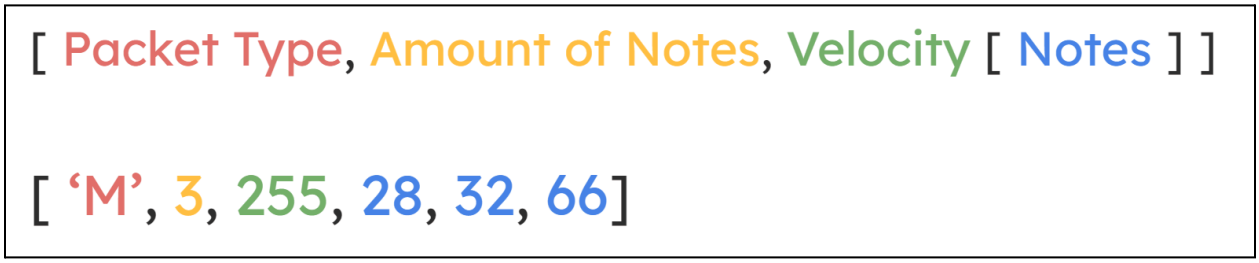

Figure 13: M Packet

The S packet contains a variable that tells how hard to push the sustain pedal down. In the MIDI standard, the pedal is either fully up or down, so it is treated as a binary.

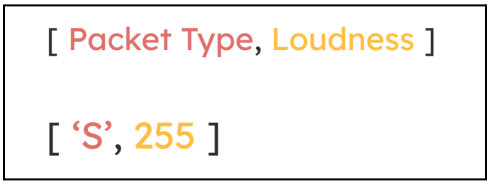

Figure 14: S Packet

The Reset power packet is a one-byte packet containing the packet header information.

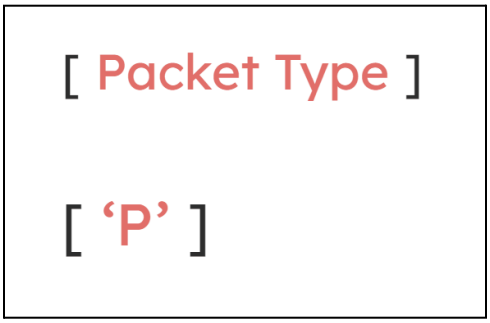

Figure 15: P Packet

The Turn all notes off packet is a one-byte packet containing the packet header information.

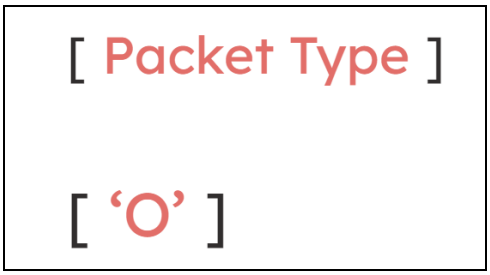

Figure 16: O Packet

4. Hardware Design

4.1 Structural Design

In designing the player piano, an important decision was made to prioritize visual engagement over traditional piano functionality. By deliberately leaving the hardware exposed, the piano transforms into a captivating kinetic sculpture, offering viewers a look at the mechanics behind each note. This approach sacrifices the instrument's conventional playability, yet it creates a compelling interactive experience that helps bridge the gap between music and engineering. This section details the design process and outcomes of inventing a player piano that invites observers to appreciate the art of its operation as much as the music it produces.

4.2 Arduino

Arduino is a prototyping platform that allows developers to write software to integrate with hardware. An Arduino is a miniature computer that has no operating system and runs code loaded onto it by an external computer. With this code, it can control a set of digital and analog pins to send electrical signals programmatically. We chose to use the Arduino Nano using the ATmega328 chip because it is inexpensive, small, and powerful enough for our purpose. During testing and development, we tried switching to the ESP32 due to its higher clock speeds but found it did not improve anything, so we returned to the Arduino Nano. It communicates with the backend software over the USB Serial standard at a 115200 baud rate.

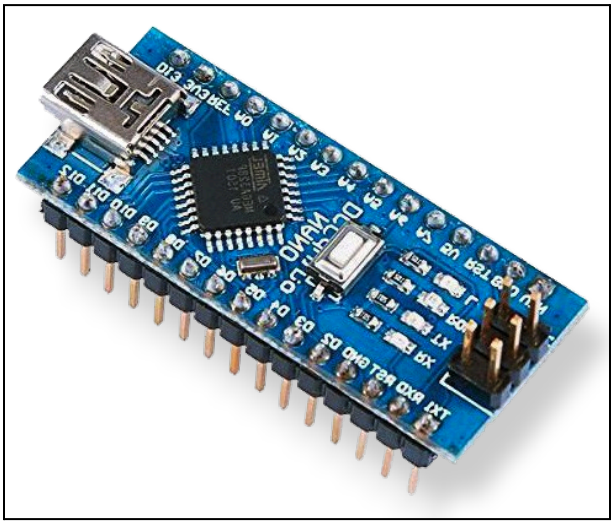

Figure 17: Arduino Nano

4.3 PCBs

A PCB (Printed Circuit Board) is a fiberglass board that connects electrical components. It provides the wires to connect different electrical pins. Due to the amount of components needed to make the project function, wiring by hand would be far too tedious. Because of this, we opted to design and get custom PCBs manufactured.

PCB creation takes several steps. Firstly, a common intermediary step is to produce a working version on a breadboard. A breadboard is a reusable plastic holder for components that enables rapid prototyping.

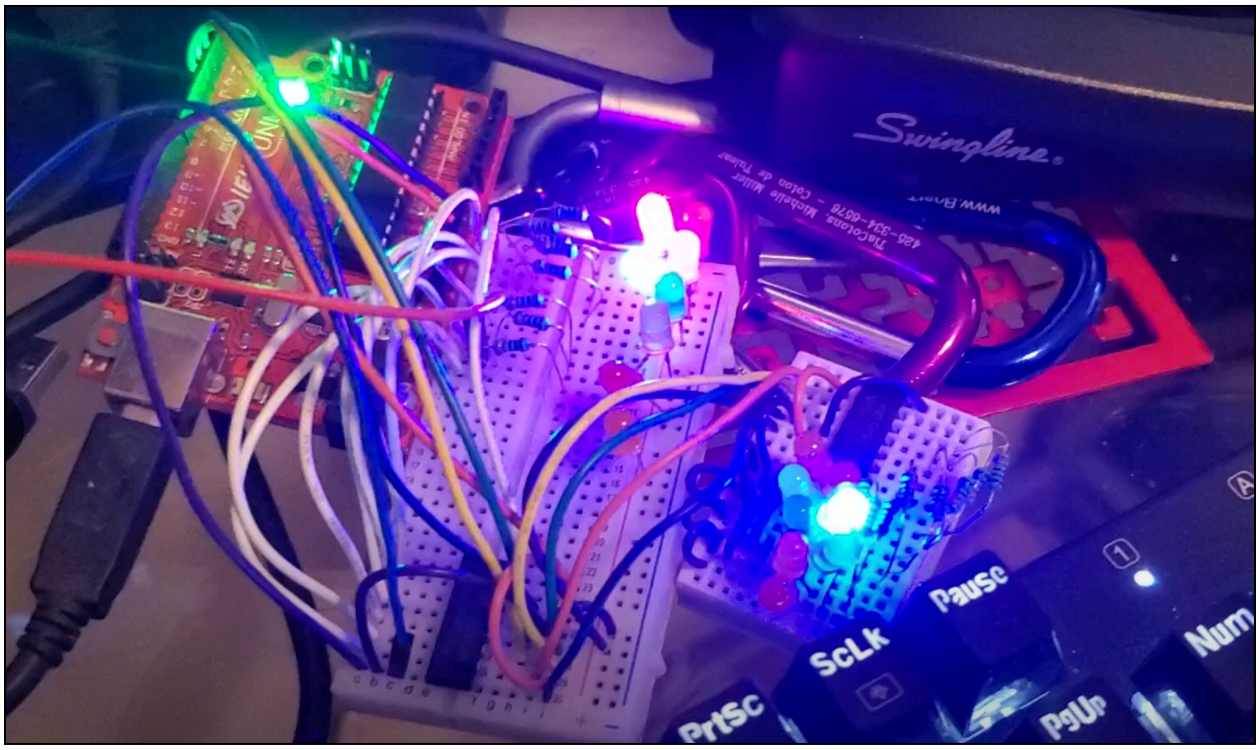

Figure 18: Breadboard

Once the breadboard prototype is tested and works, the schematic and board layout must be translated into a program. For this, EasyEDA was used because of its ease of access and abundance of online resources.

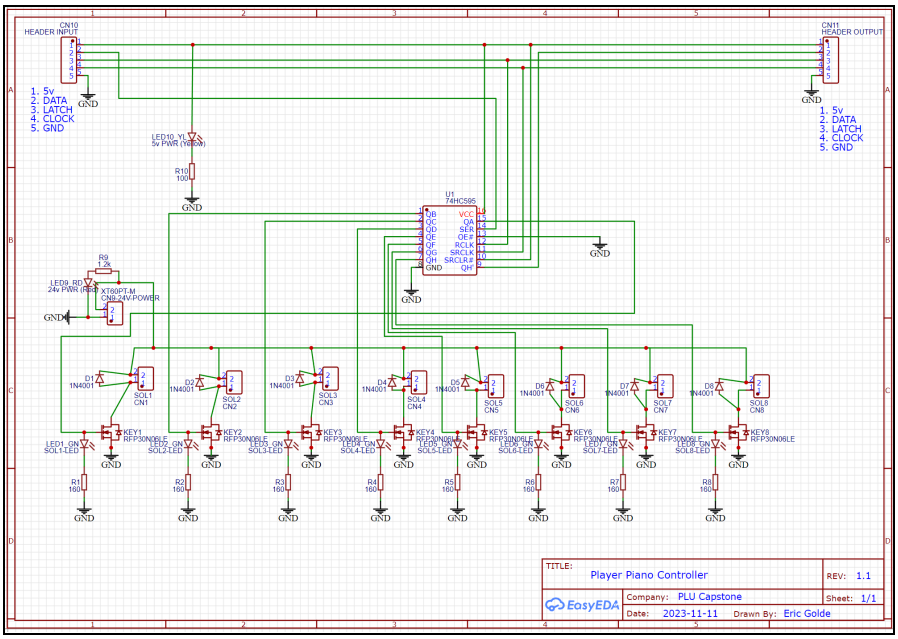

Figure 19: EasyEDA Schematic

Once the board layout is finalized, it is exported as the industry standard GERBER file. The company JLCPCB was chosen to manufacture the boards because they are a known and trustworthy company, and they have an integration into EasyEDA which makes ordering the PCBs accessible.

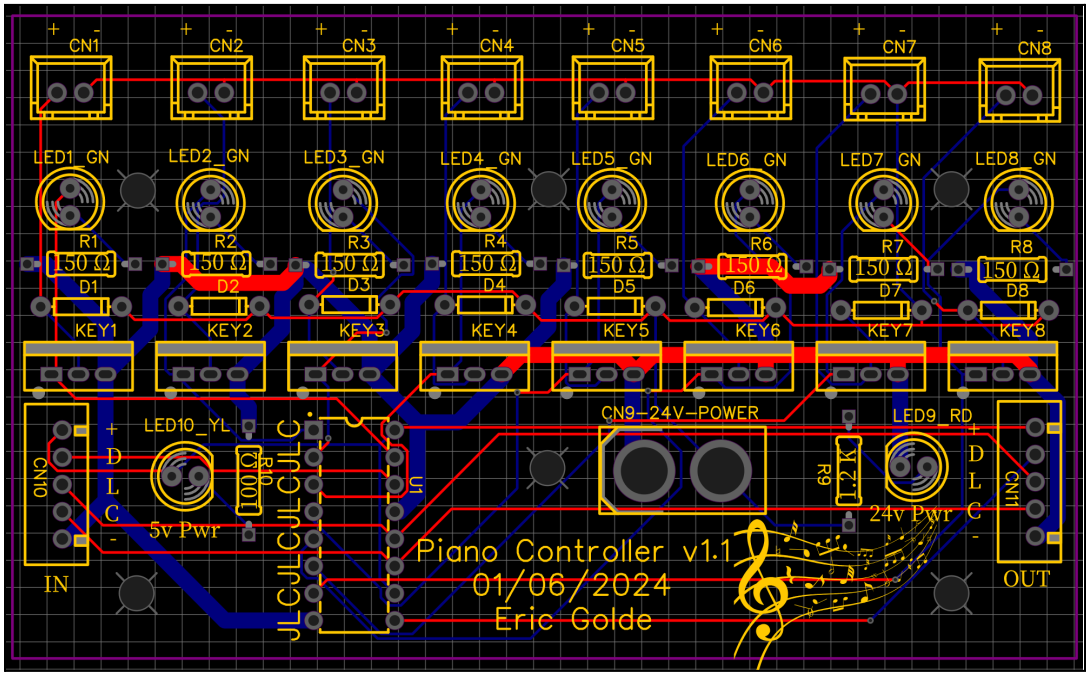

Figure 20: GERBER File

Once the PCBs arrive from overseas, all the components must be hand-soldered and tested.

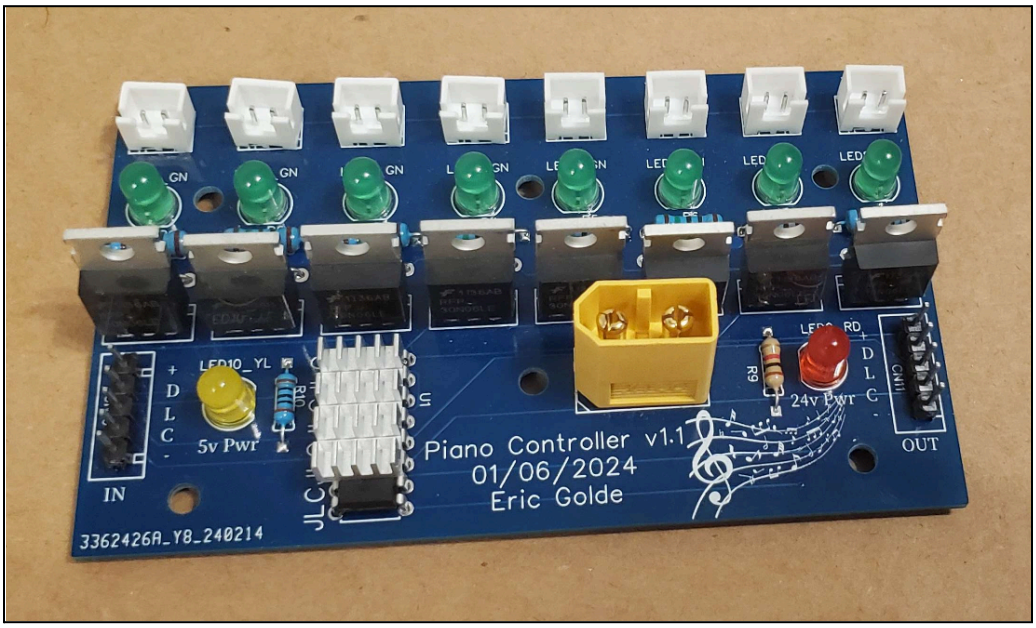

Figure 21: Finished Board

4.4 Electrical Design

An Arduino cannot individually control 88 keys due to its insufficient number of output pins. To address this limitation, we opted to multiplex the keys for several reasons. Multiplexing only requires five wires to manage virtually unlimited pins, and it utilizes the standardized 74HC595 chip. Each 74HC595 chip can handle eight outputs, and multiple chips can be daisy-chained.

We decided to use one PCB per chip, enabling us to control 8 keys per board, totaling 11 boards. Each of the eight outputs from the 74HC595 chip controls the 24-volt high-current signal that activates the solenoid, achieved using the RFP30N06LE N Channel Power MOSFET. Additionally, three types of LEDs are employed: yellow indicates 5V power to the board, red signals the 24V high-current power is active, and each green LED corresponds to an output, allowing us to see which key is powered at any given moment. These LEDs were invaluable during the debugging process.

Additional components included a 1N4001 diode to block reverse current from the solenoids, resistors to limit current to the LEDS, header, and connectors to connect wires from board to board.

4.5 Error Checker Board

The error-checker board was developed fairly late in the project, and it was hand-soldered. Issues with electrical interference were encountered causing boards to malfunction and the 74HC595 chip to overheat and crash. Through extensive testing, it was discovered that cycling the 5-volt power off and on resets the chips to their original states. The error checker board receives an extra bit of data transmitted along with the note data through each board. Once this data is received, it is sent back to an input on the Arduino, which then compares it to the bit that was sent. If they do not match, it indicates that one of the boards is malfunctioning, prompting us to reset the power pin.

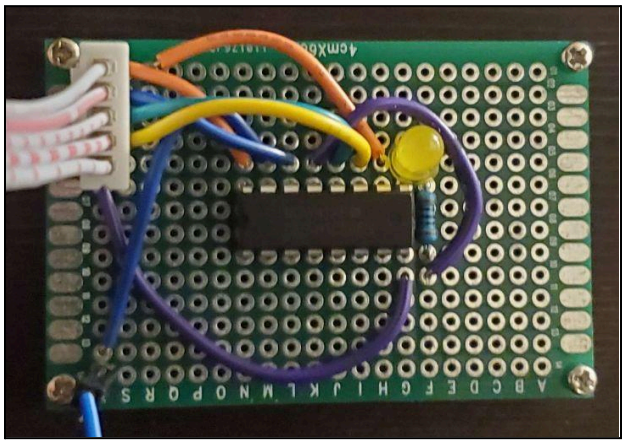

Figure 22: Error-checker Board

4.6 Solenoids

We chose to use solenoids over motors due to their quick reaction time, strong force, and compact size. A solenoid is an electromagnetic coil that moves a small iron or steel cylinder when energized and includes a spring to return it to its original position. Solenoids are commonly used in applications such as assembly lines, door locks, and pinball machines due to their reliability and efficiency.

When choosing the type of solenoid, numerous options were available. Initially, we tested the JF-0530B solenoid, a small 12V solenoid, rated with 5 Newtons of force, and appeared as one of the top results on Amazon. Its compact form factor and low noise were appealing, but it lacked the power needed to strike the key. Increasing the voltage did not improve its performance; instead, it caused the solenoid to overheat and eventually fail, emitting smoke.

We then tried the 12V variant of the TAU-1039B solenoid, which performed well but had high amperage. To address this, we switched to the 24V JF-1039B solenoid, which is significantly more powerful with a force of 25 Newtons. Using 24V, these solenoids draw less current, resulting in reduced heat. However, a minor issue was encountered: after purchasing five solenoids from one manufacturer and later buying 88 in bulk from another, we found slight differences in size specifications. This required adjustments to our 3D models to accommodate the variations.

4.6.1 Solenoid Mounter

Due to the size of the solenoids, they needed to be staggered into two rows, on both the front and back of the mounting board. A few different types of holders were produced, but the main models were the tileable holders shown below. The solenoids fit into these mounts well, and there is a small clip around the front to prevent it from rattling.

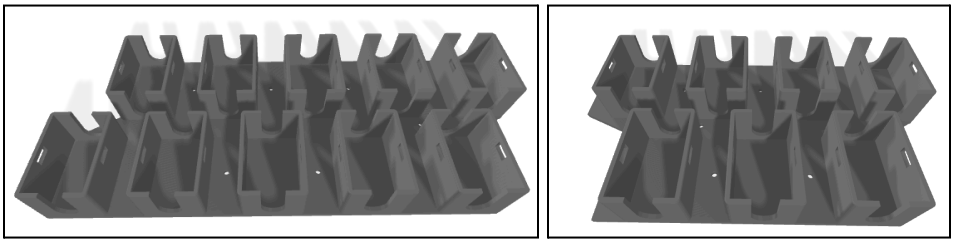

Figure 23: Main Tileable Solenoid Holders

4.6.2 Solenoid Adaptors

Each solenoid pressed the key using a length of threaded rod, which was screwed into 3D printed adapters to adapt the solenoid head to the rod. The rod was then screwed into a 3D-printed foot that struck the key. By twisting the rods, the height positioning can be adjusted to ensure the best sound and minimal clicking.

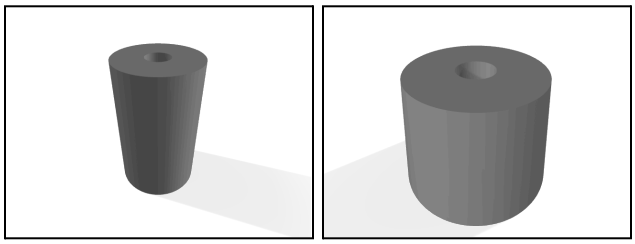

Figure 24: Solenoid Adaptors

4.7 Power Supplies

Two 24V 20A DC power supplies powered the solenoids, drawing 480 watts of power each. The PCBs are powered in alternating order to ensure a balance between the two power supplies.

Figure 25: Power Supply

4.7.1 Power Supply Holder

Brackets to hold the power supplies to the side of the piano were 3D modeled and printed.

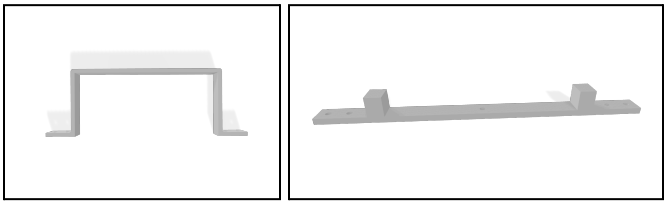

Figure 26: Power Supply Brackets

4.8 Foot Pedal Servo

For the foot pedal, we opted to use a 5V hobby servo instead of a solenoid. Since most piano pieces do not require rapid actuation of the foot pedal, speed was not a critical factor. While solenoids excel in quick response times, they lack the power to move the pedal's weight. To account for the servo's slower speed, each foot pedal event is offset by 450 milliseconds.

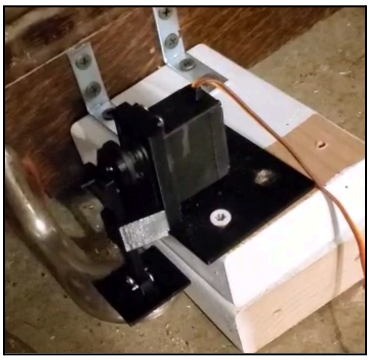

Figure 27: Foot Pedal Servo

5. Implementation

5.1 Integration

Prior to the piano becoming fully operational, various files were created to ensure that other components were functioning correctly. A suite of files was specifically designed to emulate physical components before they were fully functional. Notably, the virtual synthesizer detailed in section 3.3.2 and the piano GUI described in section 3.3.3 were instrumental in our development process.

On the website side, several extended test sessions were conducted to stress-test the website's durability during prolonged, unmoderated use. These sessions, which lasted several hours, allowed us to observe how well the website and piano performed over time. Additionally, interactive sessions were organized where users could engage with the piano. These sessions provided valuable insights into the features they found most appealing and their preferred modes of interaction.

5.2 Software Development & IDE

For the software development and IDE tools, IntelliJ was used for general coding tasks, and EasyEDA for PCB design. Fusion360 was employed for creating 3D prints, while VSCode was used for developing the website and managing the song database, enhanced with the PlatformIO extension for embedded development.

5.3 Version Control & Documentation

Throughout the project, Git was used for version control, and GitHub for repository management. Initially, we structured our repository by making each output and subprogram a separate submodule. However, as the codebase expanded, this approach became impractical due to the maintenance effort required. We then separated the SongDB into its own repository, inspired by other successful projects. This separation made the database easier to update, though it slightly increased the likelihood of merge conflicts.

We also created a repository for documentation and other essential files for anyone looking to replicate the project. This repository includes all the 3D models for components and the necessary technical documentation. Additionally, it contains the files for the circuit boards designed for the piano.

5.4 Challenges and Resolutions

5.4.1 MIDI Tempos

As described in section 3.2.1, finding the correct formula to calculate the correct MIDI timings proved challenging. After extensive trial and error and numerous online forum searches, we finally discovered a formula that works effectively.

5.4.2 Electrical Interference

We encountered significant issues with electrical interference and feedback. Despite our efforts to mitigate these problems with the error-checker board, the piano sometimes still shuts down randomly. We suspect this is caused by the rapid on-and-off switching of the solenoids (PWMing) to achieve velocity changes, though we are not entirely certain. In a future iteration, using an oscilloscope for debugging could help pinpoint the exact cause of the issue. This would allow us to design a more robust PCB with additional components to better handle the interference.

5.4.3 Overheating

We encountered unusual challenges with the 74HC595 chip overheating and failing. Although the exact cause is unclear, it is suspected to be related to the PWM driving of the solenoids. To mitigate this issue, heat sinks were added to each chip, which has proven to be mostly effective.

5.4.4 Physical Limitations of a Piano

As discussed previously in section 3.4.4, MIDI files are designed for virtual instruments without physical constraints. Our project, however, involved a real piano, which introduced several physical limitations that needed to be addressed.

The main challenge was the piano's inability to play repeated notes on the same key as quickly as a digital instrument. Digital instruments can handle rapid sequences effortlessly, but a physical

piano requires time for each key to return to its original position before it can be struck again. To resolve this, we had to modify the MIDI files to ensure smooth playback. The duration of the first note in rapid sequences was shortened, allowing sufficient time for the key to reset before the next note was played.

Additionally, the physical force required to move the piano keys and pedals presented another challenge. While solenoids generally do a good job of providing the necessary force to strike the keys, they need precise timing and sufficient power to operate reliably. We experimented with different solenoids and voltage levels, as detailed in section 4.6, to find a balance that would provide both the speed and strength required without causing overheating or failure. Addressing these physical limitations required a combination of hardware adjustments, software modifications, and extensive testing to ensure that the piano could perform accurately and reliably.

5.4.5 Logistics

Managing the logistics of a physical project of this scale presented several challenges. It was incredibly important to us to have a functional physical demo for our capstone presentation, which made the coordination of transporting the piano to campus particularly complex. The piano, weighing between 400 and 500 pounds and lacking functional wheels, required significant effort to move.

To mitigate potential issues, we decided to transport the piano to campus a few weeks before the presentations, anticipating that components might break during the move and allowing time for necessary repairs. Directly moving the piano into Xavier Hall, the presentation venue, was not feasible due to ongoing classes and the inability to leave the piano in the classrooms.

To navigate this obstacle, we first moved the piano into South Hall, mounted it on dollies, and then rolled it to Xavier Hall shortly before the presentations. This process was a substantial undertaking, requiring a team of five people and an entire hour to complete. Despite the logistical hurdles, this careful planning and teamwork ensured the piano was ready for the capstone presentation.

Figure 28: Piano in U-Haul

5.4.6 Multithreading

A significant challenge in this project was implementing multi-threading. Our backend application comprises numerous threads, with each output running on its own thread. We aimed to ensure that if one output froze, it would not halt the entire program. To achieve this, a queue system was employed to push events into another thread's queue. After extensive synchronization of threads and locks, this achieved almost complete functionality. However, a deadlock is still occasionally encountered, which has not yet been resolved.

5.4.7 Mobile and Desktop Parity

As mentioned in section 3.1, the application was originally designed to run locally on the computer attached to the piano. When the project shifted to becoming a web application, we needed to account for a variety of device sizes. This involved reworking the entire UI to be compatible with mobile devices and different screen sizes. Ensuring parity between mobile and desktop versions required careful consideration of responsive design principles to maintain functionality and usability across all devices.

Adaptive layouts were implemented that dynamically adjust to the screen size so that all features are accessible and user-friendly whether on a smartphone, tablet, or desktop computer. This included simplifying navigation for touchscreens, optimizing performance for mobile hardware, and guaranteeing that visual elements remained clear and functional across different resolutions.

5.4.8 PCB Design

Printed Circuit Board (PCB) design was a challenge in its own right. A prototype controller was built on a breadboard, but the step between prototyping and the final product proved to be challenging. The first revision of the board was almost correct but had a few flaws.

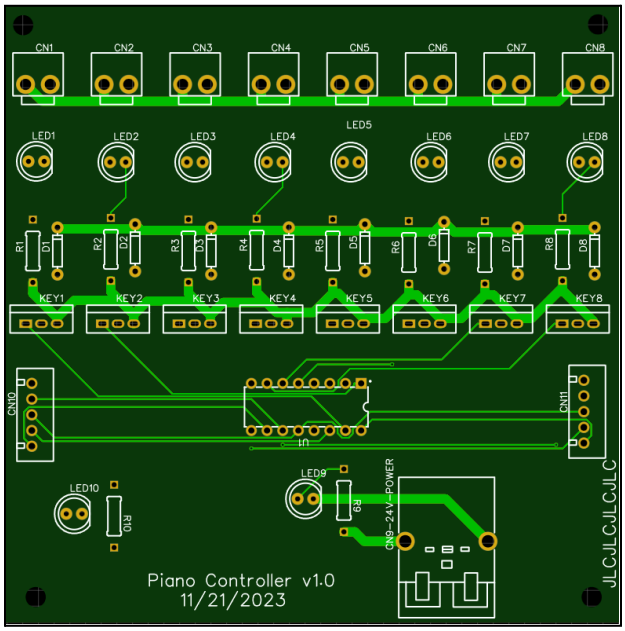

Figure 29: First PCB Revision

The wrong XT60 connector was used. There were issues with the bus sizes and connector sizes. There were issues with some traces not being fully connected. We also wanted to shrink the board down as much as possible.

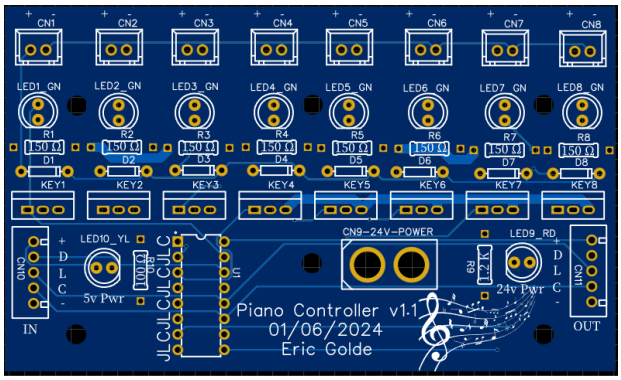

Figure 30: Final PCB Design

This is the second iteration of the board. The PCB size was reduced by half and more labels were added. The change in color from green to blue had no effect on the operation of the board, but we felt that the blue color was more attractive.

6. Ethical Considerations

Our project encompasses several different ethical considerations that must be addressed to ensure responsible and fair implementation. These considerations include the ethics of automated music and the fair use policy regarding MIDI files sourced from various websites.

6.1 Ethics of Automated Music

Automated music, such as that produced by our piano, raises important questions about the role of technology in art. While automation can democratize access to musical performances and provide education opportunities, it also risks diminishing the value of human musicianship and creativity. We must be mindful of balancing technological innovation with respect for the artistry and effort of musicians. Ensuring that the piano acts as a complementary tool rather than a replacement for live performances is crucial in maintaining this balance.

6.2 Fair Use Policy for MIDI Files

The use of MIDI files sourced from websites like Musescore.com introduces concerns related to intellectual property and fair use. MIDI files often represent transcriptions of copyrighted works, and their use must comply with applicable laws and licensing agreements. Because of this, it is essential for us to:

1. Ensure that MIDI files used are either in the public domain, licensed for educational and non-commercial use, or obtained with explicit permission from the copyright holders.
2. Follow the specific licensing terms outlined by sources like Musescore, which may include restrictions on modification, distribution, and commercial use.

7. Future Work

7.1 Difficulty Calculation

Currently, the difficulty value assigned to each song is calculated by determining the density of notes and assigning a value from 1 to 10 based on this density. While this method provides a basic measure, it is overtly simplistic. In the future, we aim to develop a more comprehensive algorithm to better approximate song difficulty. This enhanced algorithm will consider several factors, including:

- Maximum notes per second, to identify songs that exceed a predetermined ‘human limit’.
- High spikes in difficulty, to assess songs with significant increases in note density during specific portions.
- Length and number of jumps required, to account for the physical complexity of the performance rather than just the speed.

Although the difficulty is somewhat subjective, objective criteria can still be used to compare songs. For example, it is universally accepted that "Flight of the Bumblebee" is more challenging than "Hot Cross Buns". Our goal is to create a nuanced system that reflects these objective differences more effectively.

7.2 Admin Panel

We plan to develop an administrative control panel for the player piano's website, allowing users with administrative access, such as university faculty, to manage various settings. This panel will enable administrators to schedule specific times for the piano to be on or off, remove songs from the queue, and adjust the volume. The primary limitation in implementing this feature is the need to host it on PLUservers. This is essential for verifying administrative access based on PLU email credentials, ensuring that only authorized personnel can make these adjustments.

7.3 Queue Controls

We plan to enhance the website by adding features that allow users to manipulate the song queue and view its order. This will include options to rearrange, delete, or prioritize songs, providing a

more interactive and user-friendly experience. These enhancements aim to give users greater control over the playback sequence, making the player piano more versatile and responsive to user preferences.

7.4 Audio to MIDI

A potential feature for our project is an audio-to-MIDI conversion algorithm using Fast Fourier Transforms (FFTs). This complex process involves several steps:

1. **Preprocessing:** Filtering the audio signal to reduce noise and improve accuracy.
2. **FFT Analysis:** Applying FFT to convert the time-domain audio signal into the frequency domain, identifying the various frequency components.
3. **Pitch Detection:** Analyzing frequencies to detect pitches and their respective intensities.
4. **MIDI Mapping:** Translating the detected pitches and intensities into corresponding MIDI notes and velocities.
5. **Post-processing:** Refining the MIDI output to correct errors and improve musicality.

Significant progress was made in implementing this audio-to-MIDI conversion algorithm. However, the effort was ultimately abandoned in favor of focusing on a more streamlined user experience, emphasizing more necessary features to prioritize usability.

7.5 Improved Error Checking

7.5.1 Backend

The backend error checking needs improvement in terms of catching and logging errors. Occasionally, the backend code produces a fatal error without updating the debug logs. In the future, we plan to refactor our code to better catch these errors and provide the operator with essential information to prevent them from recurring.

7.5.2 Arduino

In the future, we aim to improve error detection by identifying specific errors rather than simply indicating that an error occurred. Although there is currently no straightforward method to

achieve this, each sub-board should be able to monitor data integrity issues and respond accordingly. This improvement would allow us to pinpoint faults, such as a loose wire between boards 4 and 5, by noting if board 4 reports data while board 5 does not.

7.6 Improved Documentation

7.6.1 API Documentation

Currently, our project has very minimal API documentation. We experimented with having auto-generated OpenAPI documentation, but we never reached a point where we were satisfied with the final result.

8. References

Main Website: https://piano.ericshome.xyz/

SongDB Verification Script:
https://github.com/CS499-PlayerPiano/SongDB/blob/main/.github/workflows/verify-songdb.yml

DiscogsAPI Documentation: https://www.discogs.com/developers

MIDI File Format: https://en.wikipedia.org/wiki/MIDI

Outputs.json Configuration File:
https://github.com/CS499-PlayerPiano/PianoController/blob/main/config/outputs.json

Google Slides Integration: https://github.com/CS499-PlayerPiano/GoogleSlidesEndpointTest

Error-checker Board Support: https://github.com/Simsso/ShiftRegister-PWM-Library

ServoTimer2 Arduino Library: https://github.com/nabontra/ServoTimer2

Example Project Using Submodules: https://github.com/bitfocus

Project Documentation and Downloads Repository:
https://github.com/CS499-PlayerPiano/Documentation-And-Downloads

9. Glossary

1N4001 Diode: A standard diode used to allow current to flow in one direction only, protecting circuits from reverse voltage.

74HC595: An 8-bit shift register IC used to expand the number of output pins available on a microcontroller.

Android: An open-source operating system for mobile devices developed by Google.

API (Application Programming Interface): A set of protocols and tools that allows different software applications to communicate with each other.

Apple Music: A music and video streaming service developed by Apple Inc.

Arduino Nano: A small, breadboard-friendly microcontroller board based on the ATmega328 chip.

ATmega328 Chip: Microcontroller chip used in many Arduino boards, including the Arduino Nano.

Baud Rate: The rate at which information is transferred in a communication channel, typically measured in bits per second (bps).

Bit: The smallest unit of data in computing, representing a binary value of 0 or 1.

Boolean: A data type with two possible values: true or false.

Breadboard: A reusable platform for prototyping electronic circuits without soldering.

CSS (Cascading Style Sheets): A language used to describe the style and layout of web pages.

Database: An organized collection of data that can be accessed, managed, and updated.

EasyEDA: An online tool for designing electronic circuit schematics and PCBs.

ESP32: A low-cost, low-power microcontroller with integrated Wi-Fi and Bluetooth capabilities.

Fast Fourier Transform: An algorithm to compute the discrete Fourier transform and its inverse, used in signal processing.

GERBER: A file format used to describe the printed circuit board (PCB) designs for manufacturing.

GitHub Actions: A CI/CD service provided by GitHub to automate workflows, such as building and deploying code.

GUI (Graphical User Interface): A user interface that includes graphical elements, such as windows, icons, and buttons.

High-current Signal: An electrical signal that carries a high amount of current, typically requiring specialized components to handle safely.

HTML (HyperText Markup Language): The standard language used to create and design web pages.

HTTP POST: A method used to send data to a server to create or update a resource.

IDE (Integrated Development Environment): A software application providing comprehensive facilities to programmers for software development, such as code editor, debugger, and compiler.

iOS: The operating system used for mobile devices manufactured by Apple Inc., including iPhones and iPads.

Java: A high-level, class-based programming language that is widely used for building applications.

Javalin: A lightweight web framework for Java and Kotlin designed to be simple and intuitive.

JavaScript: A programming language commonly used to create interactive effects within web browsers.

JLCPCB: A manufacturer of printed circuit boards, offering PCB fabrication and assembly services.

JSON (JavaScript Object Notation): A lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate.

LED (Light Emitting Diode): A semiconductor light source that emits light when current flows through it.

Linux: An open-source operating system based on Unix, used widely in servers, desktops, and mobile devices.

Microcontroller: A compact integrated circuit designed to govern a specific operation in an embedded system.

MIDI (Musical Instrument Digital Interface): A technical standard for digital interfaces and connectors that allows electronic musical instruments to communicate.

MOSFET (Metal-Oxide-Semiconductor Field-Effect Transistor): A type of transistor used for amplifying or switching electronic signals. It is widely used due to its high efficiency and fast switching capabilities

Multiplexer: A device that selects between several input signals and forwards the chosen input to a single output line.

Musescore: A free music notation software used to create, play, and print sheet music.

MySQL: An open-source relational database management system.

Oscilloscope: An electronic device used to measure and display the waveform of electronic signals.

PCB (Printed Circuit Board): A board used in electronics to mount and connect components in a controlled manner.

Power Supply: A device that provides electrical power to an electrical load.

PWM (Pulse Width Modulation): A technique used to vary the width of pulses in a pulse train to control the power supplied to electrical devices, such as solenoids and LEDs.

React: A JavaScript library for building user interfaces, particularly single-page applications.

Reverse Current: The flow of current in the opposite direction to the intended direction.

Reverse Proxy: A server that forwards client requests to another server, typically used for load balancing, security, and content caching.

RFP30N06LE N Channel Power MOSFET: A type of MOSFET transistor used to switch high-current loads.

Serial Packet: A format for sending data over serial communication interfaces.

Servo: A motor with a feedback mechanism used to precisely control angular position.

Solenoid: An electromagnetic coil used to generate linear motion when an electrical current passes through it.

Spotify: A digital music streaming service providing access to millions of songs and other content from artists worldwide.

Typescript: A superset of JavaScript that adds static types.

UI (User Interface): The space where interactions between humans and machines occur, encompassing input devices, screens, and other visual elements.

USB (Universal Serial Bus): A standard for connectors, cables, and protocols for connection, communication, and power supply between computers and electronic devices.

Vue: A progressive JavaScript framework used to build user interfaces and single-page applications.

WebSocket: A communication protocol providing full-duplex communication channels over a single TCP connection, often used for real-time web applications.

Windows: An operating system developed by Microsoft for personal computers, tablets, and other devices.

XT60 Connector: A type of electrical connector commonly used in RC and other electronic projects for connecting power sources.