MUSIK:
MIDI-Utilized System Interface for Keyboards

Brandon Cheng, *ECE/CS*, Tommy Chu, *ECE*, Jeremy Hui, *ECE/CS*,
Damon Lin, *ECE/CS*, Edward Pena, *ECE*, Ian Rosenheim, *MAE*, and Jouan Yu, *MAE*

***Abstract*—MUSIK is a project designed to enrich the piano-playing experience by integrating visual cues and intelligent functionalities that augment the conventional auditory experience. This enhancement is achieved through responsive lighting and a 1:1 scale display that mirrors a complete 88-key piano keyboard. Utilizing a MIDI (Musical Instrument Digital Interface) file, our display guides you by showing which notes to play and the precise timing for each.**

CONTENTS

I. INTRODUCTION

THE path to learning and practicing a musical instrument can be arduous or frustrating due to its need for meticulousness. Although traditional, learning music through a teacher can often be costly, intimidating, and a huge time commitment. With MUSIK, we take a novel approach that aims to make music accessible to all. MUSIK is a real-time, live feedback computer system that utilizes visual and auditory cues to teach students how to play the piano. With MUSIK, practicing on the keyboard becomes a gamified, interactive, and competitive experience, and learning to play the piano has never been more fun!

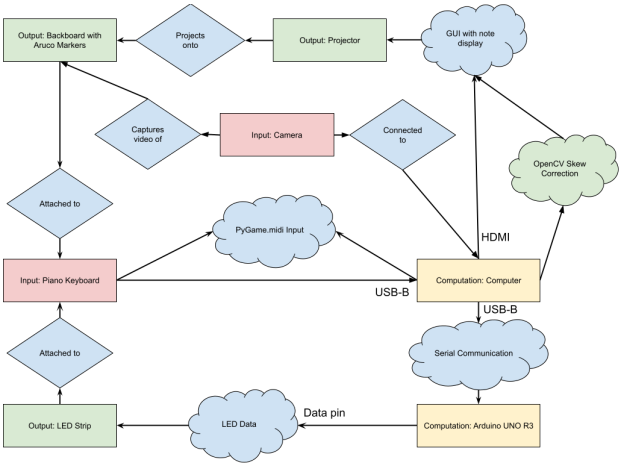

Fig. 1. Project Flowchart

A. Digital Element

The core implementation of MUSIK is programming-intensive and aimed at fulfilling our objectives. The project's code base is divided into two main platforms: high-level scripting in Python, which facilitates the integration of external libraries for enhanced functionality, and low-level microcontroller programming for the Arduino UNO R3 (see yellow computation elements in Fig. 1). The Python scripts run on a local laptop or computer, simplifying development with access to tools such as Git, pip, code editors, and various ports for external connections. Meanwhile, the Arduino programming is carried out in the Arduino IDE, which is used for compiling and uploading to the microcontroller.

B. Theoretical Element

When handling MUSIK's physical components, such as the LED strip and the projector's backboard stand, it is crucial to ensure that the design specifications are theoretically sound before actual implementation. This approach minimizes material and time expenditures for potential revisions, which is particularly vital given tight deadlines. Electronically, calculations have been performed to select an appropriate power supply that can reliably power the LED strip under maximum load. Mechanically, the backboard stand has been engineered to support the weight of the backboard and to facilitate straightforward 3D printing.

Fig. 2. System Setup

C. Physical Prototype Element

In order to present a clear idea of the implementation of this user experience, it is necessary to create a physical prototype to illustrate how these ideas will work in practicality. Many design considerations must be made in how the parts will be positioned to ensure a clear viewing experience. This is especially true for the projector, which will be elevated and positioned behind the user in a spot that does not cast the user's shadow. Supplementary parts will need to be assembled for this component, including the backboard stand designed with CAD and attaching the backboard. The connections between the keyboard, Arduino UNO R3, LED strip, computer, and projector must also all be organized in a way that is non-intrusive to the user.

II. Equipment and Methodology

A. MIDI Acquisition

The core objective of MUSIK is the ability to interpret MIDI data for a secondary function, forming the project's cornerstone. The MIDI protocol conveys information as state changes within the instrument. It assigns numerical values to musical notes, enabling identification of the corresponding note for each event. For instance, engaging and disengaging middle C on a piano triggers two distinct messages for the press and release, assigned the note number 60. Moreover, MIDI tracks additional details such as pedal events. On a piano, the pedal, operated by the foot, allows a note to sustain its tone even after the corresponding key has been lifted. The MIDI protocol thus provides a precise depiction of the musical performance.

1) MIDI from a Piano Keyboard: The pygame library is employed to capture MIDI data from our piano keyboard. Once the keyboard is connected to the computer via a USB port, the MIDI module within pygame is used to identify MIDI devices and monitor the port for incoming MIDI data. This process is encapsulated within a function that executes in a continuous loop. Remarkably, the MIDI data retrieved from the piano keyboard is highly responsive and experiences no buffering, even during stress tests, likely due to the high-performance hardware of the computer running the script.

Event type	Value
159	Note pressed
143	Note released
191/190/189	Pedal event

Fig. 3. MIDI Event Types

When a MIDI event is detected, the data is stored in a two-element Python list with the following structure:

$$event = [data, \text{timestamp}]$$

$$data = [\text{event type}, \text{note number}, \text{velocity}, 0]$$

The timestamp variable at $event[1]$ represents the time in milliseconds of the event. This timestamp, which is tracked internally by the MIDI module, is initialized to zero when the MIDI device object is created, not when the MIDI device is powered on. The $data$ variable at $event[0]$ is a four-element Python list containing the actual information of the MIDI event as numerical values at its indices. Through trial and error, we have determined the correspondence between events and their numerical representations in Fig. 3.

In events involving a key press or release, $data[1]$ contains the corresponding MIDI note number (indicating which note was pressed), and $data[2]$ contains the note's velocity (measuring how hard the key was pressed on a scale from 0 to 255, excluding 64). The note velocity returns as 64 when a key is released.

For pedal events, three distinct events are recorded when the event type is set to 189, 190, or 191 with $data[1] = 64$. $data[2]$ returns as 127 when the pedal is pressed and to 0 when it is released.

2) Reading From a MIDI File: For data to be retrieved from any song (as a MIDI file) and transformed into tiles that can be visualized by the user, the Mido library was used, according to an implementation found in StackOverflow [7]. The MIDI file is first opened and the data is inserted into a dictionary as either a 'note_on' or 'note_off' message, depending on whether a note is recorded to start or end. These messages are originally stored in the MIDI file as the time difference between each message, and so they must be transformed to a time corresponding to the start of the MIDI file to correctly set the position in the visualizer. Additionally, if a 'note_on' message has a velocity of 0, it is set to a 'note_off' message (in order to not show the note being played). Finally, the dictionary is transformed into a nested list, the time signature of the file is added, and this output list is returned to be used by the visualizer. As such, the format of each note is:

$$[type, note, time, channel]$$

B. MIDI to Tiles Display

Once the MIDI file from the Piano Keyboard and the MIDI file of a chosen song is prepared, this data must be shown to the user. This is done in the form of the MIDI visualizer program running on pygame, which works by first getting the dictionary of notes from a given MIDI file using the program

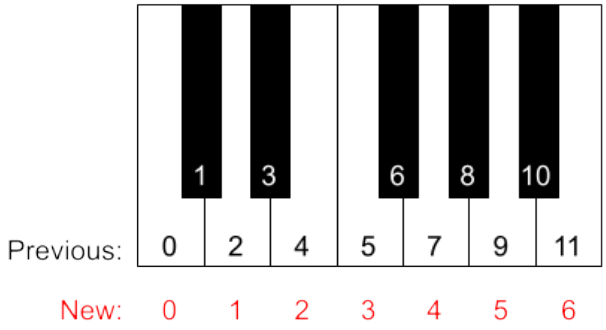

Fig. 4. Note mappings from a 13-note octave to an 8-note octave.

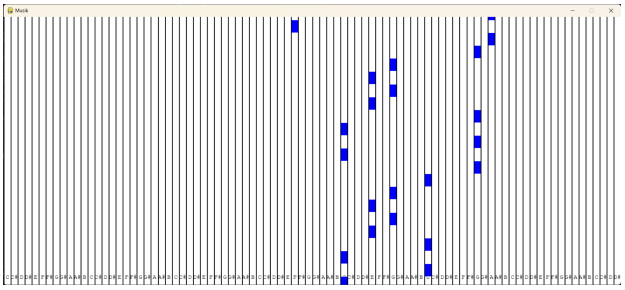

Fig. 5. MIDI visualizer showing notes from the dictionary of notes.

described in the previous section. The user is first shown the title screen, and on pressing the play button, begins converting the notes in the MIDI dictionary into tiles.

During the conversion of note numbers in the dictionary into column numbers, considerations must made be made as to which column each note corresponds. For the purposes of this project, only the white notes are used, as the alignment of black notes requires further adjustments in the offset of the notes. The formula used for computing the column number is shown below (where c represents the note number in the dictionary):

$$\text{col} = ((\text{c} - 21)//12) * 7 + \text{twelve\_to\_white\_only}((\text{c} - 21)$$

This formula is necessary since the first 20 notes are unplayable by 88-note keyboards, and to adjust each note to be in a 0-indexed array, this offset must be applied so that the first note (A0) may be displayed correctly. Additionally, the floor division on the note number ($//12$) is necessary for the program to decide which octave the key belongs in, and to transfer the key from a 13-note octave with both white and black keys (12 distinct notes), to an 8-note octave with only white keys (7 distinct notes), multiplication by 7 must be applied. Finally, the particular note is computed by using modulus on 12, then applying the twelve_to_white_only function, which maps each note number (from 0 to 11) to a particular column in an octave (from 0 to 7) as shown in Fig. 4.

While the program is running, the program automatically draws a set of equally distanced lines based on the size of the screen, which the tiles will drop between and makes resizing the piano simple to do in the future. The screen is continuously refreshed by drawing rectangles to the desired screen at the corresponding coordinates. The coordinates are found from the tiles list, which is kept in the format of:

$$[\text{color,x-position,y-position}]$$

With this format, we add notes (with a default blue color) to the x-position of $2 + \text{row\_w} * \text{col}$, where row_w is the row width. Multiplying the row width with the column allows the program to correctly position the tile in the correct column position, with a two-pixel offset to ensure that the tile does not overlap with the vertical lines. The y-position is computed by taking the time from the MIDI dictionary and multiplying it by -300. This causes notes that appear later in the song to be shown higher, as coordinates are measured at $(0, 0)$ at the top left corner, and increase in the x-coordinate moving to the right of the screen and in the y-coordinate moving to the bottom of the screen. While most of the notes will initially appear outside the screen, the tile positions will be updated by performing $\text{t}[2] = \text{t}[2] + \text{note\_speed}$ at each frame, causing the notes to move toward the bottom of the screen.

The final result is shown in Fig. 5.

C. Projector Backboard

The backboard is attached to the piano via a stand that attaches to the middle of the keyboard near where the sheet music usually goes. A lightweight, sturdy, and easily manufacturable stand was designed using Computer Aided Design (CAD) and iterated to create a sleek yet effective design, holding up the backboard via a friction fit, allowing for easy attachment and detachment. The projector projects on the backboard, which allows the user to see the notes falling onto the keys they need to press in real-time.

D. Projector Alignment

When setting up the projector and piano, we wanted a way to ensure that the projected display would line up with the correct keys on the piano. To solve this issue, we developed a system to align the projector with markers placed on the display. The goal of this system is to detect where the projector is currently projecting its image related to the desired position on the board and apply a processing step to the output feed to make it line up properly with the piano.

To do this, we decided to use Affine transformations, which preserve straight lines and planes while achieving the desired placement of the projected image on the board by applying a combination of scaling, translation, and stretching transformations.

Affine transformations can be represented using the following 2x3 matrix, where the A matrix represents scaling and shearing, and the B matrix represents translation.

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}, B = \begin{bmatrix} b_{00} \\ b_{10} \end{bmatrix}$$

$$M^* = \begin{bmatrix} A & B \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & b_{00} \\ a_{10} & a_{11} & b_{10} \end{bmatrix}$$

Fig. 6. Calibration image projected onto the screen with three ArUco markers.

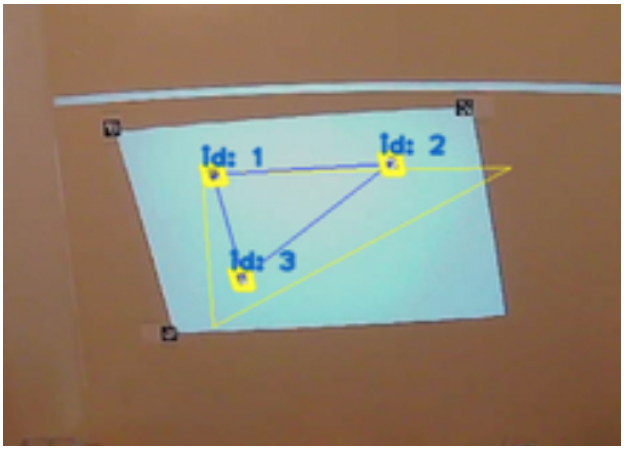

Fig. 7. ArUco marker detection and calculation of the triangles.

$$M = \begin{bmatrix} & M^* & \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & b_{00} \\ a_{10} & a_{11} & b_{10} \\ 0 & 0 & 1 \end{bmatrix}$$

To calculate the transformation required to align the projected image with the piano, we use the following steps:

1) Place three unique ArUco markers outlining the top left, top right, and bottom left corners of the desired position on the physical display screen.
2) Project a calibration image with three different unique ArUco markers onto the screen. This projected image will have no transformation applied to it, so the resultant projection will be just based on the projector's placement and likely will be skewed relative to the display screen. This can be seen in Fig 6.
3) Use a webcam to detect the positions of the 6 unique ArUco markers and record their positions in the *camera space*, which is a space of 2D coordinates that are relative to the image produced by the camera. This can be seen in Fig 7.
4) Calculate the Affine transformation matrix to convert from the *default space* to the *camera space*. The *default space* is the space of 2D coordinates relative to the calibration image. We can do this since we have two triangles: one triangle in the *default space* which is defined by the known positions of the ArUco markers on the calibration image before being projected. The other triangle is in the *camera space*, defined by the detected positions of those same ArUco markers after being projected onto the wall. By calculating the transformation matrix between these triangles, we effectively create a model for how the projector & camera system transforms an input image into a captured image from the camera.
5) Calculate the Affine transformation matrix between the projected triangle from the projected ArUco markers and the real triangle from the real ArUco markers on the board. This Affine transformation represents the shifting and skewing that is required to move the projected triangle to line up with the real markers. However, a key thing to note is that this transformation takes place in the *camera space*, which is why we had to calculate the first transformation which will be used to first bring the calibration image into the camera space.
6) Calculate the inverse transform to go from the *camera space* to the *default space*. This can be done by taking the inverse of the first transformation matrix.
7) Matrix multiply the three transformation matrices together to get the final transformation matrix. When this combined matrix is applied to the calibration image, it is analogous to applying the three transformations one by one. First, the image would be transformed into the *camera space*. Then, the image would be translated, rotated, and stretched accordingly to move from its original projected spot to line up with the real-life ArUco markers. Lastly, the image would be translated back to the *default space* by using the inverse of the first transformation matrix.

With this system, we can calculate corrections to align the projected image and its desired position for a calibration image, which can be used to transform any fullscreen display output to the correct position.

One of the issues we encountered while developing this system was that we did not account for the *camera space* and *default space*. In our initial attempt, we calculated the transformation just off the measurements of the ArUco markers from the camera feed. This led to seemingly plausible results, but the translations were always off. After debugging and determining that the transformation was correctly aligning the two triangles, we realized that this was only valid from the perspective of the camera, and we figured out that we needed to add steps to transform between the two coordinate spaces.

Another issue we encountered after this process was that the resultant image after translation lined up properly, but it got extremely pixelated. We realized that this was because we were applying the three transformations separately, one at a time to the input image. When we transformed to the camera space, this caused the image to get shrunk from its default resolution of 1920 x 1080 down to fit within the camera space, which was on a low resolution of 600 x 400, causing a large loss in detail. Then, after performing the rest of the transformations, the low-detail image would be stretched back to the resolution of the calibration image, and the pixelation would be very visible.

Our first attempt to fix this was simply increasing the resolution that the camera recorded at, which resulted in a large improvement. However, the best results were obtained when

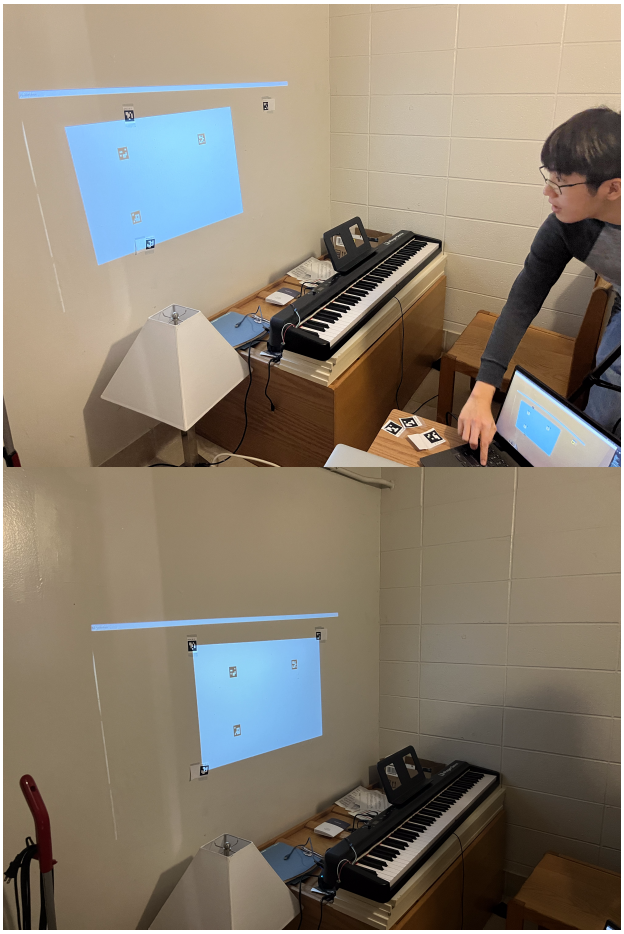

Fig. 8. Before and after alignment with two different arrangements of the real markers.

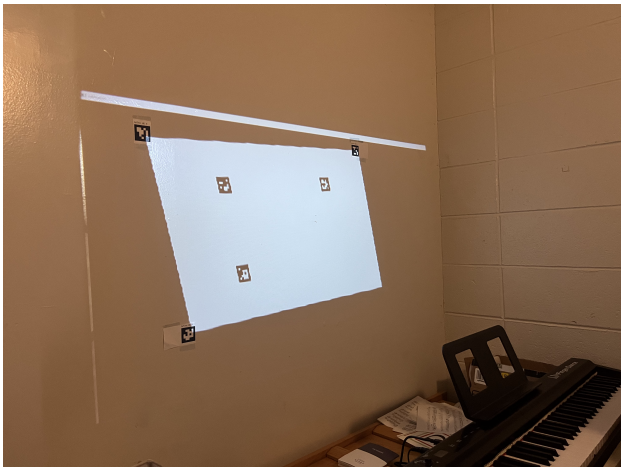

Fig. 9. Alignment with different positioning for real markers

we realized we could multiply the transformation matrices together before applying them to the image, and then applying the combined matrix all at once, which would not stretch or shrink the image more than it needed to. This eventually gave us the best results, which can be seen in Fig 8 and 9.

E. Microcontroller Communication Protocol

Once the pipeline for MIDI data from the piano keyboard is established, a method is required to relay this information to the Arduino UNO R3 microcontroller, which in turn controls the LED strip that responds as the user plays.

Since the Arduino is connected to the computer via a USB port, we can transmit information using serial communication with Python's pySerial library. The Python script opens the serial port and writes information for the Arduino to read. A limitation of serial communication is that the Arduino's serial port can be opened by only one process at a time. This means that while the Python script is running and using the port, it is not possible to upload new code to the Arduino or view outputs on the serial monitor. To address this, we terminate the Python process before uploading the code. Additionally, to enable access to the serial monitor for debugging Arduino code, we implemented a separate thread in the Python script that operates independently and reads from the serial monitor, thus facilitating easier development and debugging of the Arduino.

During the development of the pipeline between the Python program on our computer and the Arduino code, we encountered several challenges. The Arduino, by default, has a timeout that waits for 1 second after receiving a serial message, leading to a delay in the LED strip's response to keyboard input. We addressed this by setting the timeout to 1 ms. Another issue was maintaining message integrity; an overloaded serial buffer could result in out-of-order messages. Initially, we used character strings terminated by a newline character `\n` for serial communication, which caused issues when the Arduino prematurely inserted line breaks. This led to unreliable decoding of serial data on the Arduino, resulting in incorrect LED behavior.

Here's an example of our initial naive serial protocol. It shows two different cases of a MIDI response of Note 21 (a low-pitched A) is 1 velocity (very quiet) and Note 100 (a high-pitched E) at velocity 255 (very loud).

```
"21 1\n"
"100 255\n"
```

To address the issue of message integrity, we standardized the length of serial messages to a constant, smaller size. Previously, character string messages varied in length from 4 bytes to 7 bytes as seen above, with each character taking up 1 byte. These strings required encoding before writing to the serial port and decoding after reading from the serial port, which significantly hampered the Arduino's performance. Operating at a mere 16 MHz clock rate, the Arduino would lag and crash when overloaded with messages, as serial reading is a resource-intensive task.

Our new encoding format utilizes just 2 bytes of information, and we've limited the rate of serial messages to allow the Arduino ample time to clear the serial buffer. This is achieved by incorporating a 10 ms delay in the Python code following each message sent to the serial port. Moreover, to ensure further integrity, the Arduino is programmed to read from its serial port only when there are more than 2 bytes of

Dimension	Length [mm]
White key width	330.0/14 ≈ 23.57
Black key width	13
C#, F# offset	-5
A#, D# offset	-7
G# offset	-8

Fig. 10. Various Piano Key Dimensions

information available. The message format is as follows:

$$message = specifier << 8 + modifier$$

The specifier, occupying 1 byte, determines the type of event to be sent, while the modifier, also 1 byte, assigns the associated value. The message is encoded as a two-byte value, with a bit shift for the specifier, allowing the Arduino to read it into its $buffer$ array as $buffer[0]$ and $buffer[1]$ for the specifier and modifier, respectively. When reading the value, a note press is indicated by the specifier representing the note number, ranging from 12 to 119, covering the full piano range and beyond the standard 88 keys. The modifier represents the note velocity, which, through experimentation, cannot be 64, as this value is reserved for encoding a note release. For pedal events, the specifier is set to 189, with the modifier at 0 or 127 to indicate if the pedal is pressed or released. This system allows the specifier to represent various events within the 0-255 range, excluding the ranges dedicated to note and pedal events.

Reserving a specifier of 0 for no message, specifiers 1, 2, and 3 are used to alter the red, green, and blue LED colors, enhancing the visual feedback for new key presses. By recording the keyboard's state and integrating it with the music21 library, chords can be identified. The LED strip then responds with yellow for major chords and blue for minor, reflecting the chord's emotional tone.

F. LED Strip Interpolation

The LED strip we are utilizing boasts a density of 144 LEDs per meter. This ensures that each piano key is associated with at least one LED. Originally 2 meters in length, trimming the strip to the appropriate size results in 177 LEDs covering the span of the keyboard keys. However, since the piano keys are not perfectly aligned with the LEDs, we must use programming to interpolate which LEDs correspond to each key.

To achieve this, we determine the left and right boundaries of each key on the physical keyboard, starting from the left edge of the first key. This requires measuring the necessary dimensions for accurate alignment, as shown in Fig. 10.

To determine the number of white keys to the left of a given note, the Arduino code executes a loop from the left-most note number to the current note's number. Initially, the Arduino requests the user to press the keyboard's leftmost key to establish the starting note upon startup. With the left-most note number identified, a loop is run from this starting note to the current note, checking if each note is white and incrementing a counter for the white keys. A note's color is determined by the remainder when its number is divided by 12, as the pattern of notes repeats every 12 notes, forming an octave.

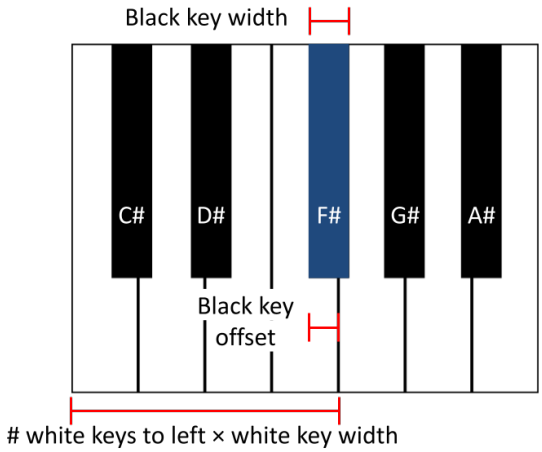

Fig. 11. Piano Key Bound Calculation

For a white key, multiplying the count of white keys to its left by the width of a white key gives the left boundary of the note, and adding the width of a white key to this boundary gives the right boundary. For a black key, the left boundary is adjusted left or right depending on the specific note. As illustrated in Fig. 11, C# and F# are slightly left-shifted, D# and A# are right-shifted, and G# is centered between its adjacent white keys. The position of a black key within the octave is also determined by the note number's remainder after division by 12. The right boundary of a black key is found by adding the width of a black key to its left boundary.

Each LED is addressed incrementally, starting from 0. The leftmost and rightmost index bounds are determined by dividing the key bounds by the width of each LED. After calculating these indices, all LEDs within this range are updated. When a note is pressed, the LEDs illuminate according to a global color variable (red, blue, green), with brightness scaled to 100% white based on the velocity, making louder notes shine brighter. Conversely, when a note is released, the LEDs either switch off or dim. The Arduino records pedal actions, maintaining the current state. If the pedal is depressed upon note release, the corresponding LED dims using a 0x0A0A0A bit mask on the RGB value, with 0% white to simulate a gradual note fade. Furthermore, releasing the pedal triggers all LEDs on the strip to turn off.

1) Power Requirements: The LED strip operates at 5 Volts. Although the Arduino UNO R3 has a 5 V output pin, its maximum output is 500 mA. Given that each LED can draw up to 60 mA and there are 177 LEDs in total, the maximum current draw if all LEDs are fully lit at the same time would be 10.62 Amperes. Therefore, we have chosen a 5 V 10 A power supply, which provides sufficient wattage for this project.

Considering the actual usage of the LED strip control, a more realistic estimate of the current draw is approximately 12.55% of the maximum, or 1.33 amps, since we have programmed the strip to operate at 32/255 of its full brightness.

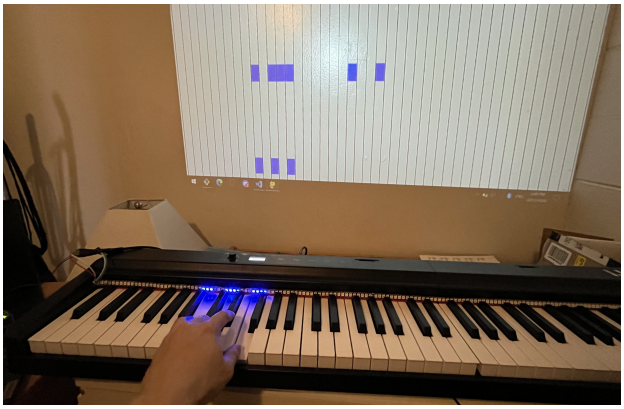

Fig. 12. MIDI Visualizer with Reactive LED strip

III. Results and Discussion

Due to time constraints, a few features as well as the overarching connection between the different components were not implemented. Although the different components of this project (MIDI visualizer, projector alignment, LED strip display) are not fully connected, considerable progress has been made on each subsystem through our parallelized workflow. Furthermore, MUSIK's codebase was developed solely in Python, which would facilitate the integration of the three distinct components in the future.

For the MIDI visualizer, the notes can be read from the MIDI file and move toward the bottom at a given speed, based on the dictionary that is passed in. Additionally, upon clicking each tile (which would be equivalent to playing the notes on the keyboard), the timestamp would be printed out using the Python time library, based on the UNIX timestamp. This will be useful in a more finalized version of this project, where the timestamp of the note when it reaches the bottom of the screen can be compared to when it is played on the keyboard. In the future, this will be incorporated along with the projector alignment for proper key display.

The LED strip display functionality of MUSIK is fully connected to the piano keyboard, yielding a responsive reaction to the player as they play the piano keyboard. In addition, the Arduino microcontroller can utilize the full red-green-blue functionality of the LED strip by changing colors, which is currently used to react to major or minor chords. This component is the most memorable component of MUSIK so far.

A. Future Work and Improvements

With more time, the following may be integrated:

1 **Interconnectivity:** The MIDI visualizer should communicate with the Python program managing the MIDI inputs to ascertain whether the user's inputs correspond to descending musical notes. These two portions should interact with the Arduino and LED strip to demonstrate whether the user's inputs were on time. The auto-alignment feature should align the projection with the display surface and return a transformation matrix. This matrix will be passed to another program that handles displaying an augmented projection feed.

2 **MIDI Visualizer:** With more time, the MIDI visualizer needs to support black keys, displaying and detecting keys other than the white keys. Moreover, incorporating interface elements, such as adding songs, adjusting speed, and pausing the song, would immerse the user further. While not necessary nor detracting from the overall experience for the user, finger-to-note mapping perfects and completes the MIDI visualizer.

3 **Electronic Footprint:** Currently, the project encapsulates numerous hardware components such as the webcam, the projector, the LED strip, and the computer. These components all house their electronic components separately and require unsharable cables and power sources. Thus, figuring out a way to house most, if not all, of the electronic components within one module, would drastically decrease the electronic footprint MUSIK has.

4 **AI Instructor:** Although ambitious, the objective of implementing an AI instructor is not out of the realm of possibility. The instructor should provide feedback on the user's performance, taking in data from their MIDI inputs. The only limitation comes from the need for an excessive amount of training data.

5 **Perspective Transforms:** During the testing of the auto-alignment feature, another method of orienting the image was through a perspective transform as opposed to an affine transform. On paper, the perspective transform remedies the issue where the aligned projected image skews relative to the display screen. Therefore, with more research and trials, attempting to implement perspective transformations would be an objective.

IV. Conclusions

MUSIK provides an important step towards engineering design and development as it reveals a potential solution that assists individuals in learning piano, especially those who cannot afford a teacher. It facilitates high customizability, resulting in individuals being more willing to learn piano as they can play the songs that they enjoy, and elevates their experience throughout the process into something enjoyable through the use of colorful lights, encouraging users to return to practice more. This project also extends the functionality of hardware that already exists on piano keyboards but is likely not utilized fully by most people. Our design has the potential to impact a large community, helping beginners learn the fundamentals and more advanced players enjoy a new experience in playing piano.

Acknowledgements

We would like to thank Dean Antoine, Megan Holleran, our design advisor Girish Sankrithi, our alumni mentor Maxwell Legrand, and the rest of the design advisor team for guiding this project to where it is today.

REFERENCES

[1] "Adafruit NeoPixel." *Adafruit NeoPixel - Arduino Reference, Arduino, https://www.arduino.cc/reference/en/libraries/adafruit-neopixel/*

[2] dmitchelldm74. "Dmitchelldm74/Tiles-Pygame: A Game Somewhat like Piano Tiles, Made in Pygame." *GitHub, github.com/dmitchelldm74/Tiles-pygame.*

[3] "How to Add a Visualizer to Your Piano (after Effects Tutorial)." *YouTube, HalcyonMusic, 26 Aug. 2023, www.youtube.com/watch?v=5ZXz5cORqrA.*

[4] "Music21.Chord¶." *Music21.Chord - Music21 Documentation, web.mit.edu/music21/doc/moduleReference/moduleChord.html.*

[5] "Piano Key Frequencies." *Wikipedia, Wikimedia Foundation, 20 Mar. 2024, en.wikipedia.org/wiki/Piano_key_frequencies.*

[6] "Pygame Documentation." *Pygame Front Page - Pygame v2.6.0 Documentation, www.pygame.org/docs/.*

[7] "Python Mido How to Get [Note, Starttime, Stoptime, Track] in a List?" Stack Overflow, 1 May 1966, stackoverflow.com/questions/63105201/python-mido-how-to-get-note-starttime-stoptime-track-in-a-list.

[8] Vandenneucker, Dominique. "MIDI Tutorial." *Arpege Music - Dominique Vandenneucker, www.cs.cmu.edu/ music/cmsip/readings/MIDI%20tutorial%20for%20programmers.html.*