Capstone Final Report – Group C3

GIANO

Ajith George, Nouriya Al Sumait, Skyla Marie Profita, Rahul Singh, Nikolas Varga, Celine Habr

Advisor: Professor Charles DiMarzio

TABLE OF CONTENTS

Abstract 3
Introduction 4
Problem Formulation 4
Design and System Analysis 5
Design Overview 5
System Flowchart 6
System Communication 7
Firmware System 9
Computer Vision System 11
ArUco Perspective Transformation 12
Mask Generation 13
Finger Tracking 16
Audio Output Design 18
Parts and Implementation 20
Velostat Force/ Pressure Sensors 20
Analog Sensor Reading 28
Haptic Motors 30
Power and Protection 32
Connectors 34
PCB and Enclosure 35
Glove Exterior Design 36
Cost Analysis 38
Timeline 39
Conclusion 40
References 41

Abstract

The GIANO team has developed a prototype wearable glove system that enables accessible, low-cost piano learning without a physical keyboard, paid lessons, or dependence on visual feedback. Traditional piano education often requires expensive equipment, fees, and access to lessons, and sight-based pattern recognition, which limits access for those without financial resources and low-vision and legally blind users. Music education supports motor coordination, memory, focus, and social-emotional development. GIANO offers a haptic feedback-driven, computer vision-assisted platform that delivers real-time instruction, feedback, and audio output across both free-play and guided learning modes.

At the center of the system is a Raspberry Pi 5 running a computer vision algorithm that generates a virtual keyboard mask, tracks hand position, and manages hand positioning logic. It communicates with three Teensy 4.0 microcontrollers: two on glove-mounted PCBs and one dedicated to audio generation. Each glove integrates seven haptic motors (one per finger and two for hand direction) along with Velostat pressure sensors and flex sensors that detect finger presses, movement, and articulation. By combining sensor data with camera-based tracking, the system provides finger-by-finger guidance in learning mode.

After a brief calibration, in learning mode, the Raspberry Pi issues data instruction sets containing data such as note, finger, and hand position goals. The glove-mounted Teensys activate the appropriate haptic motors through an I2C multiplexer and monitor sensors for user input. When a press is detected, they transmit the information to the Raspberry Pi, which verifies it against the expected note. All detected presses, in any mode, are also sent to the third Teensy, which produces audio using FM-based synthesis.

The prototype reliably detects single note presses, provides clear haptic cues, and generates accurate sound output. A key limitation appears during chord instruction: driving multiple motors simultaneously through the I2C multiplexer increases latency and reduces responsiveness.

Future work includes improving haptic control for multi-finger chords and creating a fully hands-free, visual-free setup process to enhance accessibility. Overall, GIANO demonstrates a portable and inclusive approach to music learning, lowering barriers for users of all abilities and backgrounds.

Introduction

The goal of Giano was to provide an inexpensive, accessible tool to learn piano without relying on visual cues and feedback. The product was aimed to be standalone, spatially efficient, and have a low cost of maintenance. Giano consists of 2 gloves that the user wears, including touch detection and haptic feedback mechanisms, a computer vision and central communication system on a Raspberry Pi, and a 4-octave piano representation printed on 3 sheets of paper with AruCo markers on the corners. To inform the user's hand placement relative to the mat, a camera video is sent to a computer vision unit. Velostat touch sensors were employed to confirm that the user has pressed the correct key. Flex sensors were employed to measure the user's technique and hand pose during playing.

Two modes, free play and learning mode, were developed to serve users' playing needs. These milestones required different levels of implementation across the entire system, so they also served as project development milestones.

The primary subsystems of the project are computer vision, audio output, sensors, and feedback. The sensor and feedback systems are hosted on a custom 4-layer PCB mounted to each glove, powered by Teensy 4.0 microcontrollers. The computer vision system is housed on a Raspberry Pi 5 (Raspi) single-board computer (SBC), while the audio output is housed on a third Teensy 4.0 microcontroller with an audio "hat" housing an SGTL5000 audio codec.

Problem Formulation

Learning to play the piano presents significant barriers for many beginner learners, particularly those who lack access to expensive instruments, sufficient physical space, or formal instruction. Traditional piano education often requires a full-sized piano or digital keyboard, paid lessons, and long-term access to a dedicated practice environment, which can be cost-prohibitive and impractical for many users. Additionally, most instructional methods rely heavily on visual cues, such as watching hand placement, following sheet music, or interpreting on-screen demonstrations. These requirements can make early-stage learning difficult for beginners and especially inaccessible for individuals with low vision or visual impairments.

Beyond musical skill itself, access to music education has been shown to support broader developmental benefits, including improvements in fine motor coordination, memory, concentration, and social and emotional regulation [1]. These benefits are particularly valuable for early learners, yet the same barriers that limit access to piano instruction also prevent many individuals from experiencing these positive outcomes. When music education is restricted by cost, space, or reliance on visual feedback, its cognitive and emotional benefits remain inaccessible to large portions of the population.

While alternative learning approaches exist, such as audio-only lessons or learning 'by ear,' they often lack structured, real-time guidance and feedback. Audio cues alone do not indicate which finger should be used or whether correct technique is being applied, and they offer limited support for developing proper motor coordination during early practice. As a result, learners may struggle to build confidence, form correct habits, or progress independently without constant external instruction. For users who cannot rely on visual feedback, these limitations are even more pronounced, further restricting access to meaningful music education.

The core problem addressed by GIANO was the absence of an accessible, affordable, and portable piano learning system that supports beginner learners while remaining usable by individuals who cannot depend on vision. GIANO reframes piano instruction as a multisensory experience by prioritizing tactile and auditory feedback over visual interaction. Through wearable haptic guidance and real-time audio output, the system enables users to practice finger placement, timing, and coordination without watching their hands, reading sheet music, or interacting with a physical keyboard.

From a system design perspective, this problem requires integrating sensing, processing, and feedback mechanisms into a wearable platform that can reliably detect finger presses, interpret user intent, and deliver intuitive guidance in real time. These capabilities must be achieved within constraints of comfort, simplicity, and cost to ensure broad accessibility. GIANO serves as a proof-of-concept solution demonstrating how beginner-focused and accessibility-driven design principles can be combined to lower barriers to music education and extend its cognitive, motor, and emotional benefits to a wider range of users.

Design and System Analysis

Design Overview

GIANO is a modular and multi-unit system, comprised of three primary subsystems: the gloves, the audio output unit, and the computer vision (CV) unit, all communicating and coordinating through a central computing unit (Raspberry Pi 5). Together, these systems provide auditory and tactile guidance and feedback, allowing the user to play and learn.

The glove interface, controlled by two Teensy 4.0 microcontrollers, handles primary user input and tactile feedback mechanisms. The microcontrollers each handle real-time firmware execution and serial communication with the computer vision (CV) units to receive instructions and verify performance. Flex sensors, embedded in each of the gloves, work to detect and verify hand posture, while custom pressure-sensitive Velostat sensors detect note presses along each of the fingers. The gloves operate autonomously on the firmware level, guiding setup and receiving instructional data to return sensor data back to the system controller.

The computer vision unit provides spatial context, hand and finger identification, and mapping from position to notes. Using hand and keyboard tracking, the CV system is able to assign key positions to specific finger inputs, translating hand positioning to musical intent to generate audio output. The information within this subsystem guides both the gloves through haptic feedback, interprets data, and drives the audio unit to enable a visual-free setup and play.

The audio output system was responsible for real-time sound and musical output. Centered around a third Teensy 4.0 with an audio hat attachment, MIDI note and velocity data are converted into sound through both FM and additive synthesis algorithms, with ADSR envelopes shaping note tone and dynamics. This subsystem works closely with the central unit and the resulting data from the gloves to produce immediate, piano-like audio for the player to use as guidance and feedback.

Together, these three subsystems work to form a closed-loop learning system, integrating sensors, guidance, and sound generation to support affordable and inclusive musical experiences.

System Flowchart

Below, an overall system flowchart can be seen. Figure 1 highlights the path in which communications occur, to be highlighted in later sections. MIDI files are stored, either as voice commands for setup or in the form of pre-loaded song instructions for learning mode, to be sent over serial to the RasPi 5. These instructions work hand-in-hand with the webcam and CV unit, which helps to guide and track finger/hand placement by returning coordinates of finger and hand positioning. All of this is stored and runs through logic in the RasPi, determining thresholds such as those to assign notes to keys, and what audio should be sent for output upon detecting a finger press. Serial communication between the RasPi and Teensy 4.0 microcontrollers on the gloves allows for relaying commands determined by the CV unit, as well as executing setup instructions, and reading GPIO pins to detect Velostat sensor presses. The audio output unit, upon key presses, receives MIDI output serially, and the Teensy 4.0, in combination with FM and Additive Synthesis algorithms, produces the output.

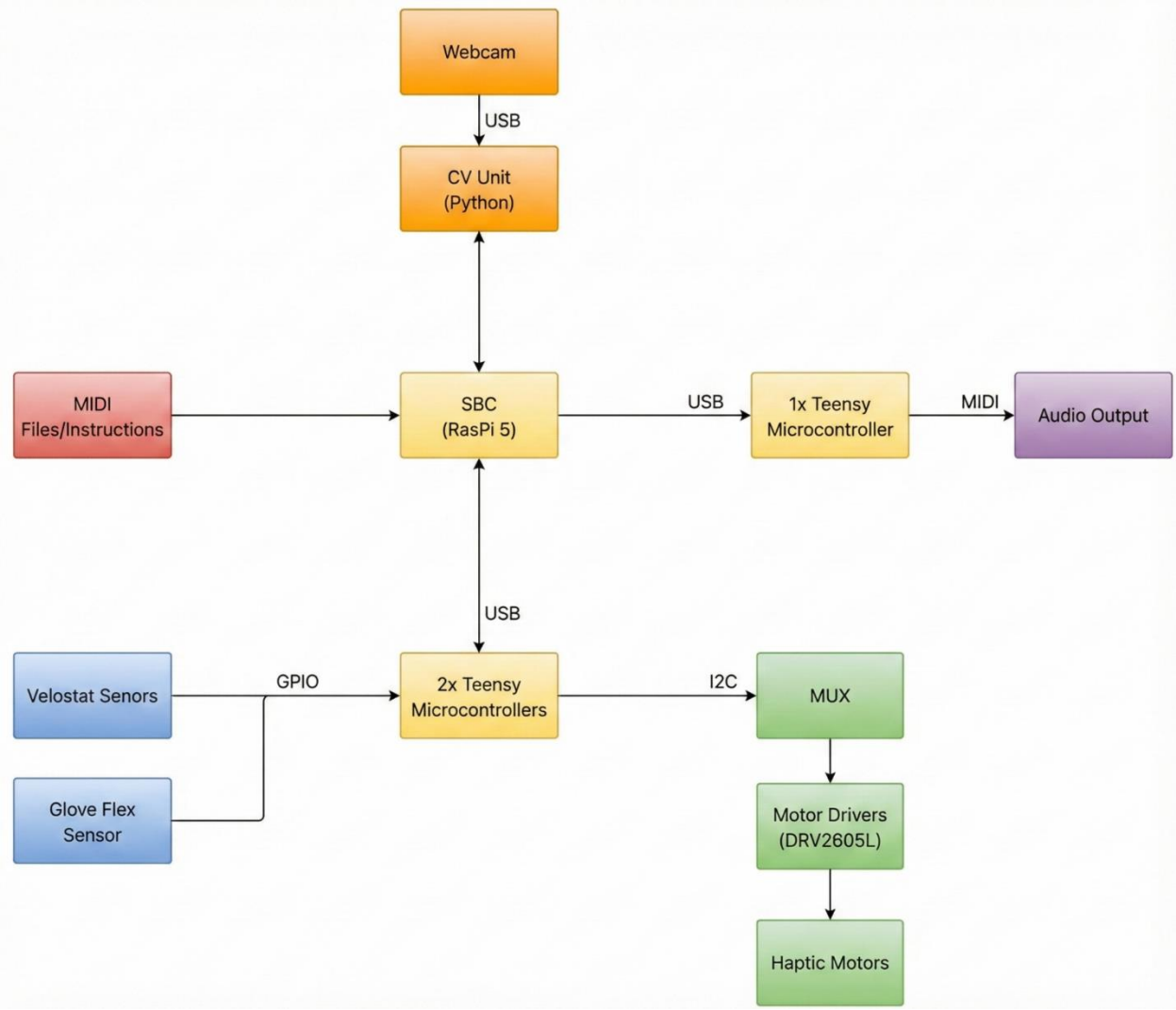

Figure 1: *Overall GIANO system flowchart.*

System Communication

With 3 microcontrollers and an SBC, communication between subsystems was a key problem identified for low-latency data transfer. To support this goal, we developed a serial-based communication system between the glove microcontrollers and the SBC. We form the connection between the microcontrollers and the SBC using a handshake protocol, where the connection is confirmed between both ends before proceeding with the piano masking process. The Python system identifies USB connections with the two microcontrollers, where the handshake protocol is completed by the glove microcontroller sending the right or left hand byte as 0xFF or 0xFE, respectively. The LeftGloveSerialManager class in Python is used to perform the handshake, and if the right-hand byte is received, it constructs and returns a RightGloveSerialManager with the connection established. This distinction is made because the left glove contains the button user interface, where voice commands are sent during the setup process. During the setup process, the

LeftGloveSerialManager in Python listens for 1-byte messages on the serial connection, then forwards these as MIDI data to the audio microcontroller. In this stage, the Velostat touch sensors are also calibrated to the user. In free play mode, the setup process ends after the mode is selected, and in learning mode, it ends after the mode and song are selected. After the flush byte is received by the LeftGloveSerialManager, the mode is sent to the right glove, which changes the loop operations. Free play mode does not require the use of the haptic motors for guiding. The setup bytes are listed in Table 1.

Command	MIDI/Serial value (Unsigned nteger value)
WELCOME_MUSIC	16
WELCOME_TEXT	17
MODE_SELECT_BUTTONS	18
SELECT_SONG	19
FREEPLAY_MODE_CONFIRM	20
LEARNING_MODE_CONFIRM	21
CALIB_VELO_NO_PRESS	22
CALIB_SOFT_PRESS	23
CALIB_HARD_PRESS	24
HOW_TO_CHANGE_MODE	25
HOW_TO_RESET_SONG	26
FLUSH	27
DEBUG	28
INVALID	29
CALIBRATING	30
CALIBRATE_SINE_WAVE	31
CALIBRATION_FAILED	32
CALIBRATION_SUCCESS	33
FIX_POSTURE	34

Table 1: *Voice command bytes*

Audio data is commonly transferred between system typologies using the Musical Instrument Digital Interface (MIDI) protocol, so we used that for communication between the SBC and audio microcontroller. In general, MIDI commands are a series of 3 bytes. The first byte is a command byte that represents what the digital instrument should do. Examples of this include the MIDI commands we use, which are NOTE ON and NOTE OFF. The second byte is the key number on an instrument. For example, 60 corresponds to the pitch middle C, or C4. The third byte is the velocity, which is used to control loudness. In our system, this corresponds to the pressure detected by the Velostat sensors. If a user were to press C4 on the piano as hard as possible, the MIDI command 128 60 127 would be sent from the Python serial listener to the audio microcontroller. Because the audio system is also used to play voice commands to tell the user how to use the system, we use MIDI values lower than 48, which is the lowest note on our piano, to transmit commands to the audio hat.

Firmware System

In this section, it is important to note that the firmware being referred to is for the Teensy 4.0s connected to each glove, which was the primary firmware system. This system was developed using C++, with Platform.io program for compilation and uploading to the glove Teensy 4.0s. The flow chart below, Figure 2, shows the progression of the system for both the learning and free play modes. Some notable aspects of the firmware system include: visual free setup of gloves and the ability to switch modes or reset a song during the active playing stage.

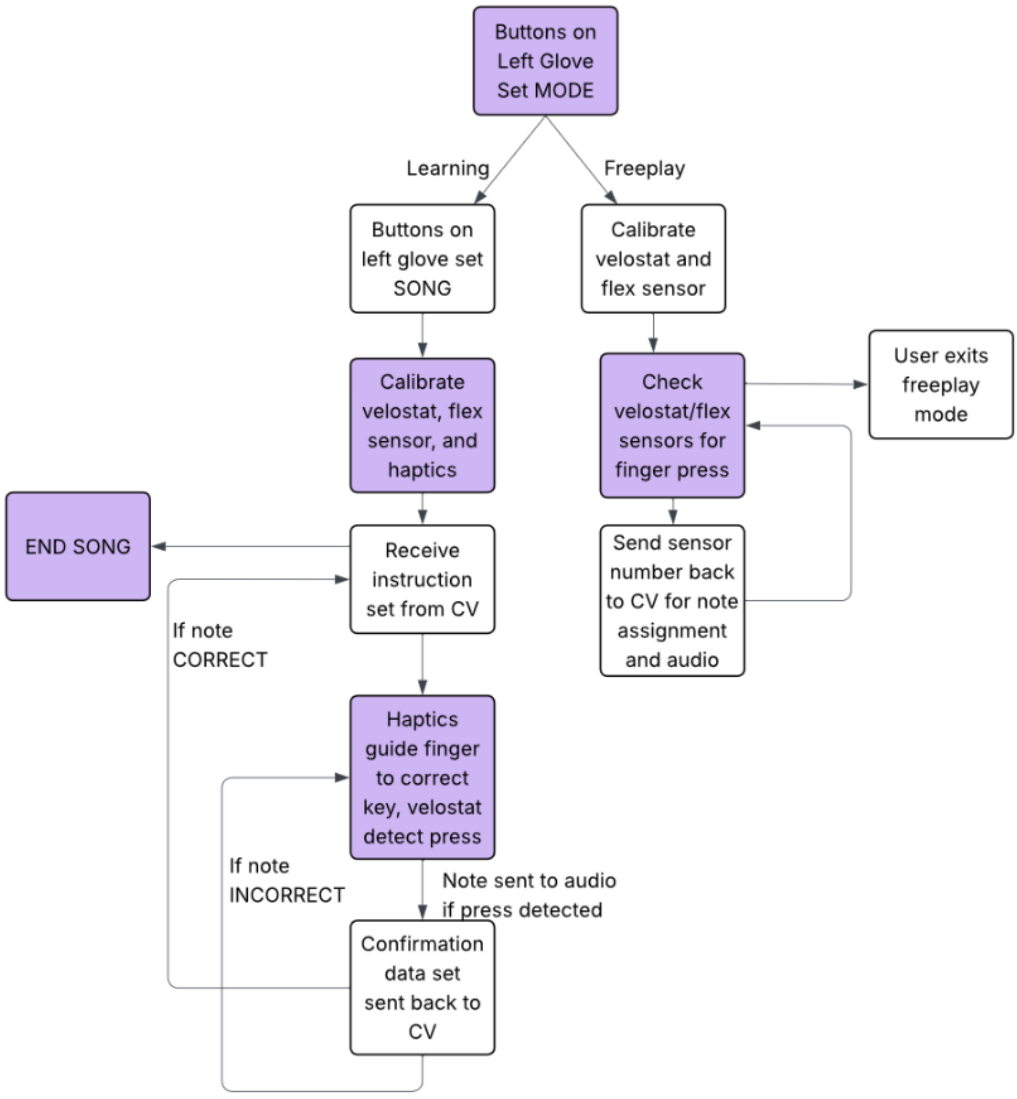

Figure 2: Firmware system flowchart for both free play and learning mode.

Upon startup of the overall GIANO system, the firmware begins to guide the mode and song selection processes. This is explicitly done through the left glove, where the buttons on the glove are activated.

After USB enumeration and a successful serial handshake with the system controller, each Teensy initializes its local peripherals, including I²C communication, sensors, and haptic drivers routing through the multiplexer. A handshake protocol with the main controller Python unit is used to reliably identify each glove as either the left or right hand before continuing execution. Once communication is established, an initial haptics calibration routine is executed to verify motor functionality across all assigned fingers, as well as to alert the user that the bootup has started in a visual-free manner.

After initialization, the firmware works with the audio output and Python-based controller unit to conduct a guided, visual-free setup sequence. In this stage, the left glove prompts the user to select an operating mode using the designated rightmost mode button. A single button press selects free play mode, while multiple presses select learning mode. This selection is transmitted to the central controller and mirrored across both gloves to ensure synchronized operation.

Once a mode is selected, the firmware performs sensor calibration. All velostat sensors are calibrated using a three-stage pressure model (open, soft press, and hard press) to dynamically establish baselines and maximum press values for each finger and user. This calibration process enables reliable detection of finger presses and releases using hysteresis thresholds, allowing for consistent performance across users and hand sizes. Flex sensor calibration is performed concurrently to establish posture reference values.

If learning mode is selected, the firmware advances to a song selection stage, controlled by the leftmost button on the left glove. The number of button presses within a fixed time window determines the selected song, which is communicated to the CV and audio subsystems. Once setup is complete, the firmware provides haptic confirmation and transitions into active gameplay.

During active operation, the firmware follows one of two execution paths depending on the selected mode. In free play mode, the system continuously monitors velostat sensors for press and release events. Detected finger presses are encoded with hand identifier, finger index, press state, and velocity, and sent in real time to the system controller for note assignment and audio synthesis.

In learning mode, the firmware operates in a guided feedback loop. Instruction sets received from the CV subsystem specify which finger should be actuated. Corresponding haptic motors are triggered to guide the user to the correct key, ensuring that octave and finger haptic strengths increase as the user comes closer to the correct position, while the velostat sensors monitor user response. Correct presses are confirmed and reported back to the CV unit, allowing progression through the song, while incorrect presses result in continued guidance until the expected input is detected.

At any point during active play, the user may toggle modes by using the mode button or reset the song using the song button on the left glove. This action triggers an interrupt-driven state change, ensuring that mode transitions occur safely without disrupting ongoing sensor reads. The ability

to recalibrate, switch modes, or restart songs without removing the gloves reinforces the system's visual-free and user-centered design.

Computer Vision System

The computer vision system was designed using the Python OpenCV library. The system follows the flowchart in Figure 3. OpenCV was used because of its large quantity of available image processing methods. Two main aspects of the computer vision system were developed: binary mask generation for the piano (identified as piano calibration in the system) and hand detection during playback.

The first step before running the project is to calibrate the camera used to remove distortion. When we began developing the system, we used a Logitech C615. Webcams are calibrated using the camera_calibration.py script. OpenCV contains a function called drawChessboardCorners(), where a chessboard paper can be used for calibration. To calculate the distortion a webcam encodes, the computer calculates the ideal position of the corners of the chessboard, then measures the error between the ideal and measured location. The distortion can be represented by equations 1 and 2. After taking around 20 pictures, the distortion coefficients k1, k2, p1, p2, and k3 are stored in a camera distortion matrix and used to correct distortion.

$$x_{distorted} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) y_{distorted} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Eqn. 1: *Radial camera distortion*

$$x_{distorted} = x + [2p_1 xy + p_2(r^2 + 2x^2)] y_{distorted} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

Eqn. 2: *Tangential distortion*

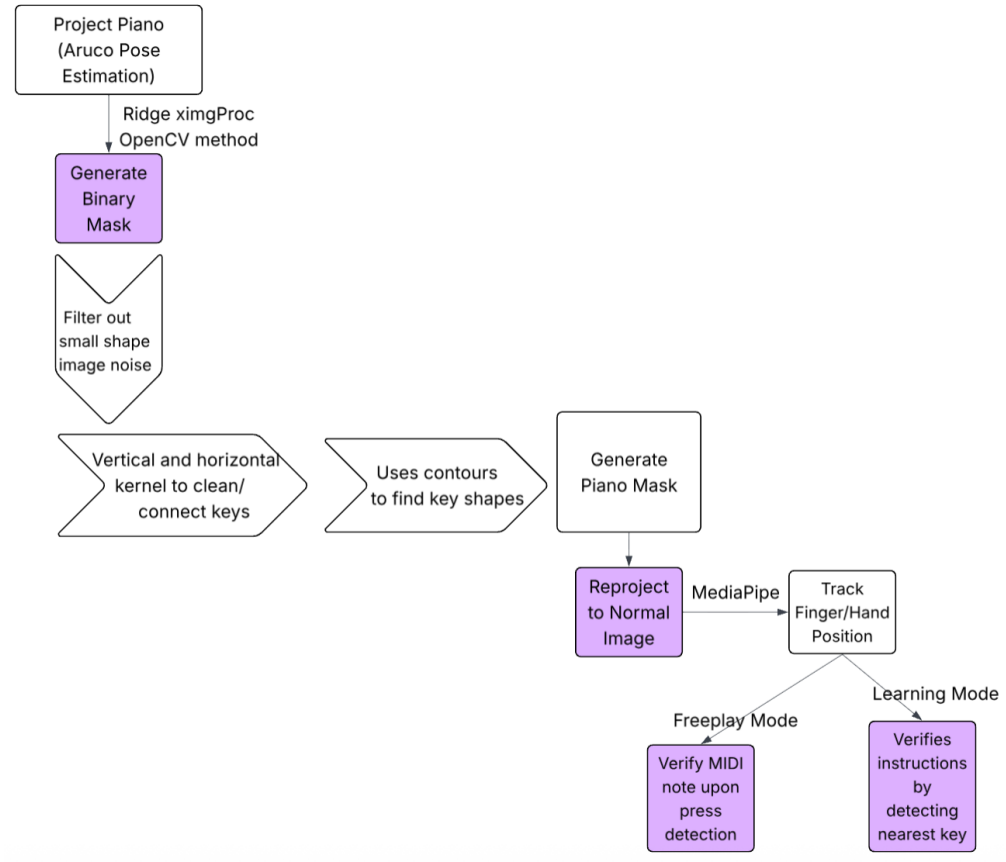

Figure 3: *CV system flowchart*

ArUco Perspective Transformation

The piano keyboard includes ArUco markers to enable a perspective shift to a "birdseye view" to simplify mask generation and detect finger distance to notes linearly. ArUco markers are "synthetic square markers composed by a wide black border and an inner binary matrix which determines its identifier (id)" [2]. OpenCV's ArUco library has a simple detectMarkers() function that identifies ArUco marker candidates through adaptive thresholding, contour extraction from the thresholded image, and code extraction from within the ArUco marker. From this process, the pose of the marker can be estimated using PnP estimation, which returns the rotation and transformation vectors that express the transform of the object from $\mathbb{R}^2$ to $\mathbb{R}^3$ [3]. This affine transformation is described by equation 3. An issue identified in the development of Giano was the amount of distortion this process was encoding on finger positions when the camera was positioned at an angle to the keyboard. This distortion was characterized by changes in the finger's z position being represented as changes in the y position in the perspective-shifted coordinate system. This was resolved by placing the camera directly above the keyboard using a table-mounted stand system.

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = T_w \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

Eqn. 3: *Affine transformation of the u,v coordinates of an object in the camera coordinate system to the world coordinate system. Tw is the estimated matrix.[3]*

Mask Generation

The first step in the CV process involves "calibrating" the piano by generating a binary mask encoding the boundaries between keys for deterministic note press detection. The first step in this process is to detect the AruCo markers. Four AruCo markers on the corners of the paper the piano is printed on are used to form the ROI for mask generation and perform a perspective transform to get a linear view of the piano. An example of this can be seen in Figure 4**.** First, median blur with a kernel of 3x3 and CLAHE are applied to enhance contrast between key boundaries and background, and widen detectable key boundaries. Next, OpenCV's implementation of the Ridge Detection Filter, followed by Otsu binary thresholding, are used to generate a binary key boundary mask where the key boundary pixels are 255, and everything else is 0.

The next step in the mask generation process is characterizing the key shapes. AruCo markers generate contours that are not key boundaries, so they are cropped from the image before the contour processing. As for preprocessing, the first step is morphological opening, which is erosion followed by dilation. This is done with a kernel of 1x3 to remove small, tall noise. We then use the OpenCV connectedComponentsWithStats() function to find all white areas with an area greater than zero. Importantly, this function returns the area of non-zero regions as one of the statistics. The masking process encodes some noise as areas that are not key boundaries but are identified as 255 on the binary mask. By turning all white regions with an area < 500 black, we discard the majority of the noise from the mask, improving key boundary detection. After this, we perform morphological losing, which can fill in "holes" in the key boundaries by connecting contiguous regions that were masked as discontinuous. We perform this with a 1x7 kernel and a 3x1 kernel, since larger horizontal kernels can connect key boundaries. The final step in generating the mask image is the OpenCV findContours() function. This finds shapes composed of inner pixels with value 0 and boundaries of pixels with value 255. The key boundaries are generally taller than they are wide, with a fairly large area. We filter contours that have bounding rectangles with $area < 0.001 * mask.size$ and those that have $2w > h$. We also filter contours with m00=0, where m00 is defined as $sum(region)$. We perform these filtering steps on each contour identified by findContours(). The remaining contours are assigned MIDI notes. As can be seen in Figure 4 **(same figure)**, the keyboard is upside down in the image. Pianos have notes going from low pitch to high pitch corresponding to left to right, so we assign MIDI notes from low to high going from right to

left. The final mask with MIDI note names on the keys is shown to the user in a pop-up window for confirmation, then stored as a dictionary for processing during playing.

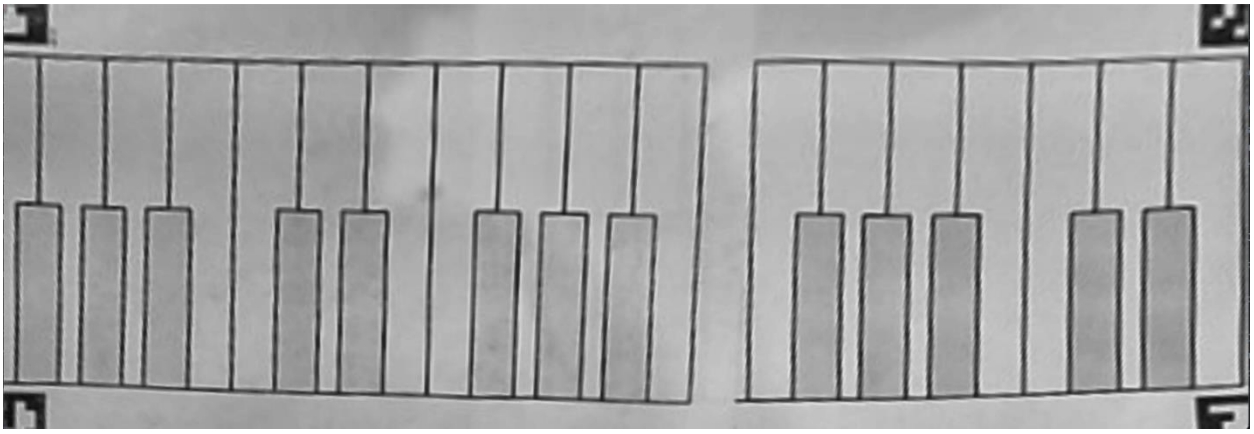

Figure 4: Transformed piano with CLAHE and median processing.

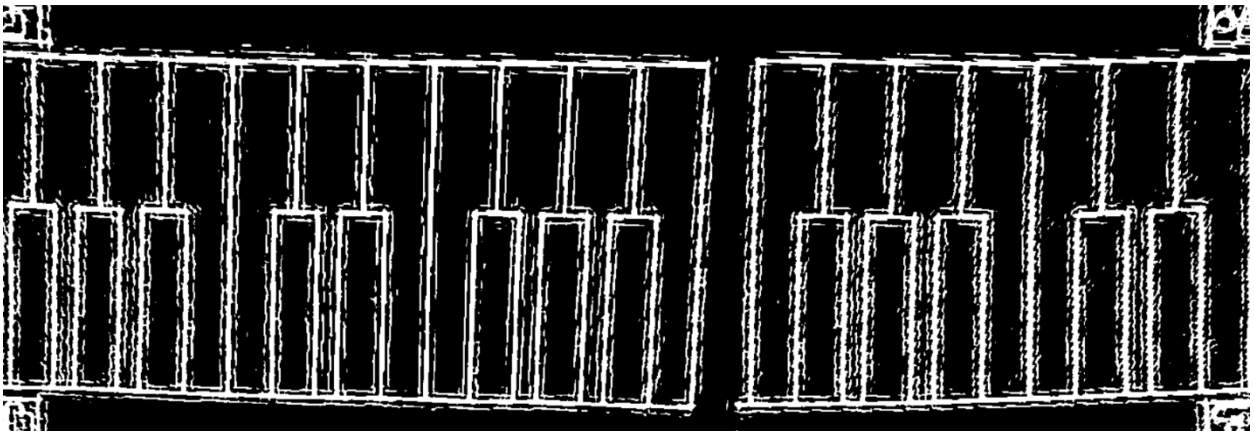

Figure 5: Ridge Otsu Mask

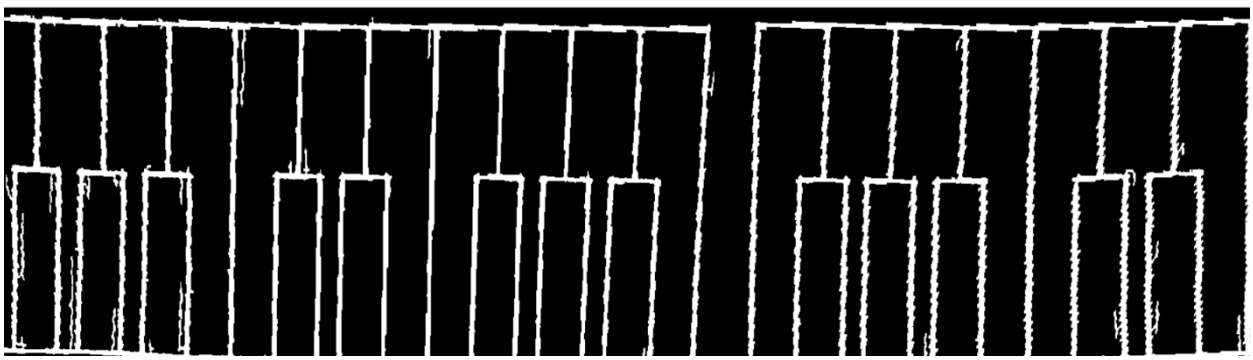

Figure 6: Cleaned masks. Notice the noise removal.

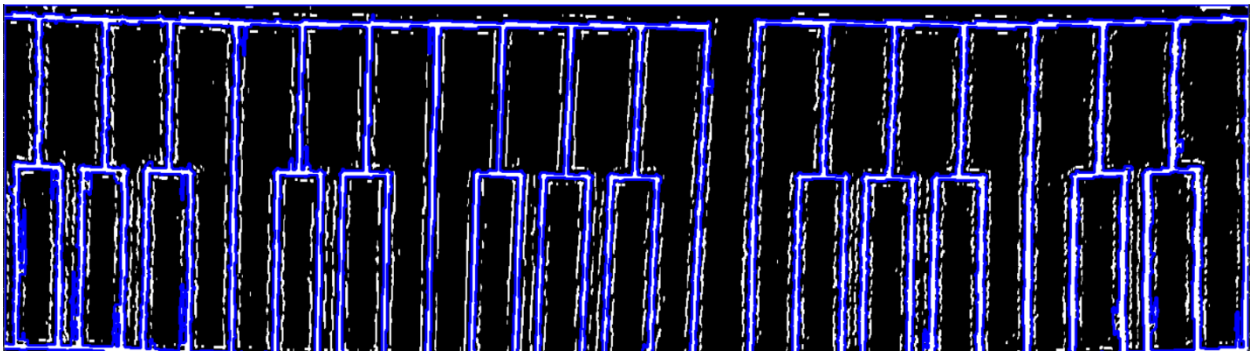

Figure 7: Generated contours

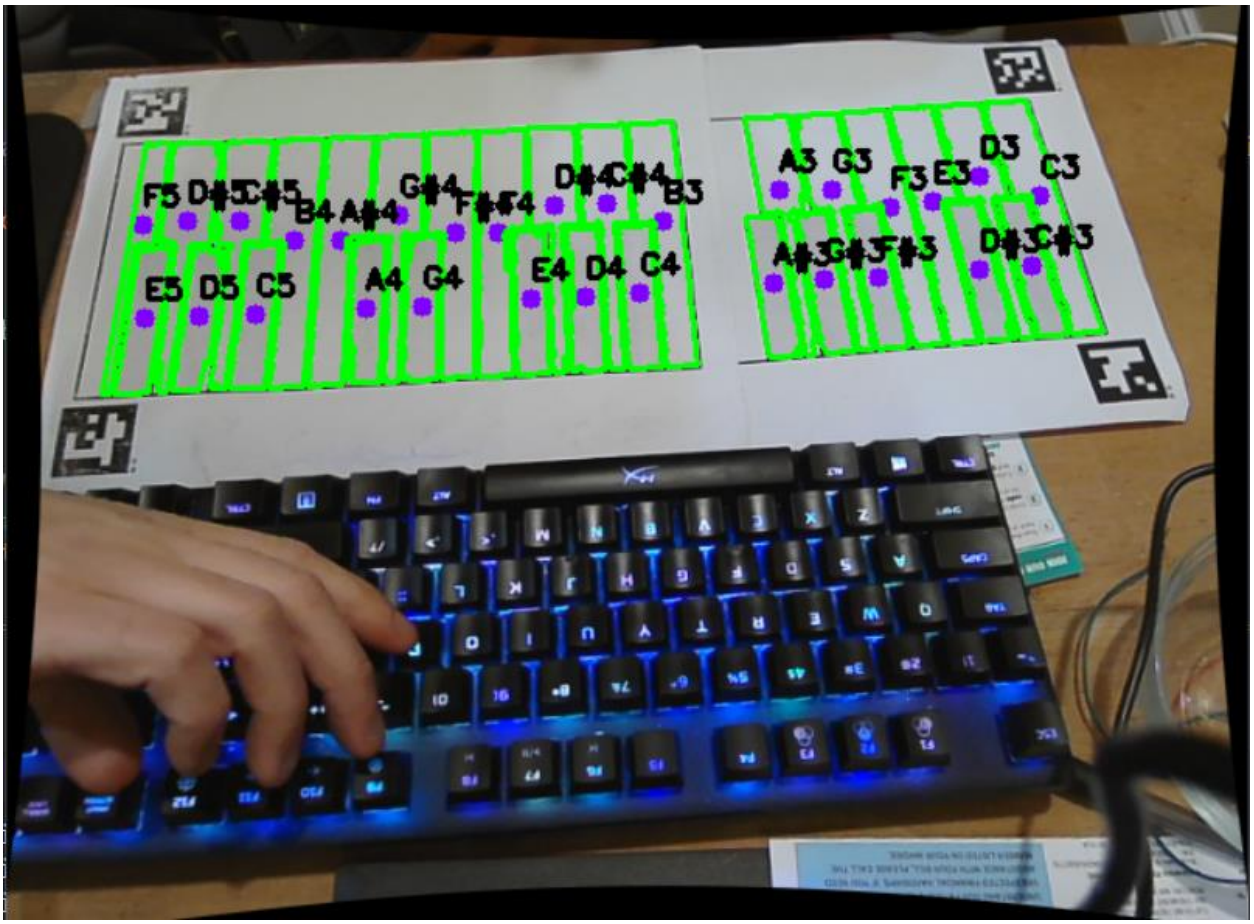

Figure 8: Final mask generation on a constrained surface. This is not the full width of the keyboard. The ArUco markers are also incorrectly placed on the viewer's left side of the keyboard.

Finger Tracking

Finger tracking is necessary to determine which note the audio system should play when a key is pressed. We first proposed using Google's MediaPipe library. MediaPipe contains a hand pose estimation model, which is composed of 2 models itself: a heavier hand detection model and a lighter-weight hand pose estimation model. This was proposed due to its ease of implementation and resistance to occlusions, which are uncommon but possible during piano playing. The model detects the (x,y) coordinates of joint keypoints on the hand, including the critical keypoints on the fingertips. During testing and implementation, it was discovered that the MediaPipe hand pose estimation model was not good at detecting gloved or heavily occluded hands. This posed a problem for our system. We instead moved towards a color detection approach, mounting colored paper with different colors on the fingers of the gloves. For detection, we first transform the image into the HSV color space, which separates color information from intensity for robustness in varying lighting conditions. We use the HSV ranges in Table 2 to define the colors. The color boundaries can be thought of as a decision surface $\in \mathbb{R}^3$, where points within the range are classified as that color. The OpenCV method inRange() is used to create a binary mask where colors within the range are 255 and colors outside the range are 0. We filter these masks by discarding those with area < 1000 pixels and > 100000 pixels. Future work would tune these ranges to more accurately reflect the printed colors and lighting conditions using a calibration process similar to that of the piano calibration process.

Color	HSV lower bound	HSV upper bound	Wrap Around
Red	[0,120,60]	[8,255,255]	[172, 120, 60]-[180,255,255]
Orange	[9,120,60]	[24,255,255]	
Yellow	[25,120,60]	[35,255,255]	
Green	[36,100,50]	[35,255,255]	
Blue	[100,100,50]	[130,255,255]	

Table 2: HSV ranges for each color.

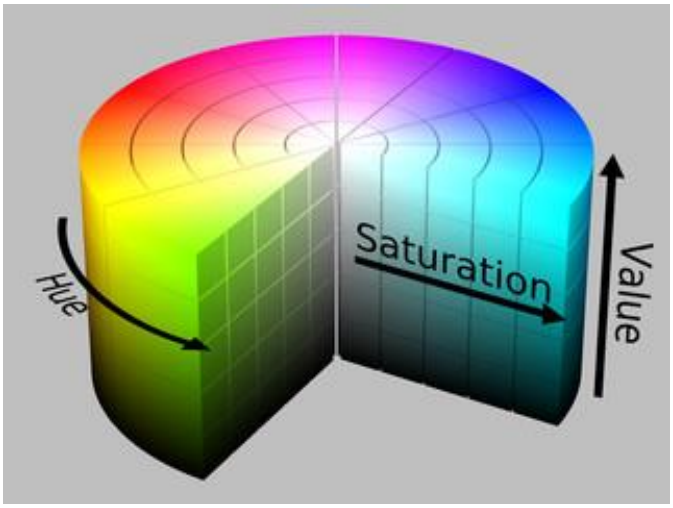

Figure 9: Visualization of the HSV color space [4].

In addition to color tracking, we use the Kalman Filter to predict the future locations of color region bounding boxes based on previous locations and velocities of the bounding boxes. This can predict finger locations for frames where the color is occluded. The Kalman filters are updated on each frame, predicting the location and size of the bounding boxes. The Kalman Filters are added to new color regions when they are detected and removed after 10 frames of missed detection. The fingertip is estimated to be the point on the contour with the highest y-value. Future work would make this more robust and accurate for tracking fingertips.

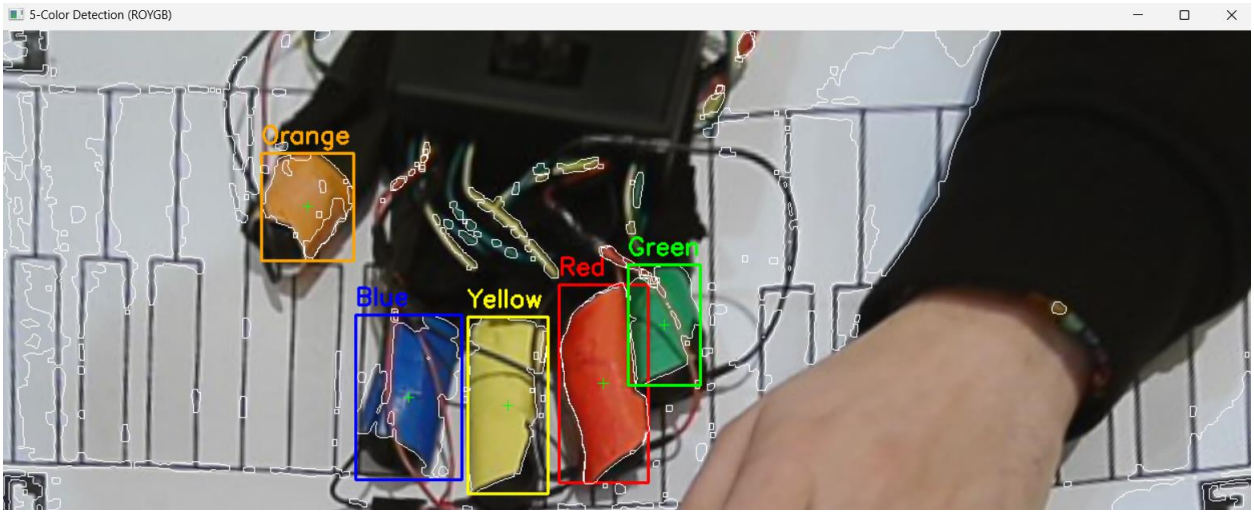

Figure 10: Color detection on fingers visualized in an OpenCV window

Audio Output Design

Using a combination of Frequency Modulation (FM) and Additive Synthesis, the entire piano audio output for Giano was made from scratch. This was programmed in C++, using the Audio Hat Teensy for output. A flowchart of the piano audio output section of the glove firmware is shown in Figure 11.

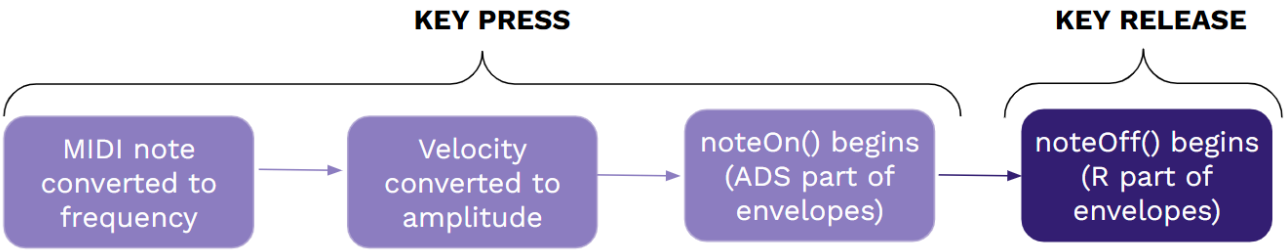

Figure 11: *Flowchart of piano audio output from a key press to a key release*

When the user presses a key, the system detects the MIDI note, which sets the carrier wave's frequency. To convert the MIDI note to a frequency, the equation 4 is used.

$$f = 440 \times 2^{\frac{n-69}{12}}$$

Eqn. 4: *MIDI note frequency conversion*

n is the MIDI note number, 69 is the MIDI note number for the A4 piano note, and each semitone, or the smallest pitch interval, is represented by the $2^{\frac{1}{12}}$ factor in the equation. This is because there are 12 keys, or semitones, within an octave.

The amplitude of the carrier wave is set by the velocity, which is determined by how hard the user presses the custom Velostat sensors mounted on the fingertips. The amplitude can be calculated as follows:

$$A(v) = \frac{\ln(1 + \frac{sv}{127})}{\ln(1+s)}$$

Eqn.5: *Amplitude formula*

s is the scaling factor, which is set to 5, and *v* is the velocity.

The modulator's frequency was set to equal the frequency of the carrier for a 1:1 ratio, and the amplitude was set to be a third of the carrier's. Three harmonics were added for a more realistic, piano-like sound using the equation for the piano's stiff-string inharmonicity model:

$$f_n = nf_0 \times \sqrt{1 + Bn^2}$$

Eqn. 6: *Piano string inharmonicity formula*

In equation 6, n refers to the integer harmonic number, which starts from $n = 2$ for the first harmonic above the fundamental. The parameter B corresponds to the inharmonicity coefficient, set to 0.0008. For higher harmonics, the amplitudes decrease rapidly, such that the amplitudes for harmonics 2, 3, and 4 are 0.1, 0.05, and 0.015, respectively.

The additive synthesis section consists of four Attack, Decay, Sustain, and Release (ADSR) envelopes that shape the audio and control the time-varying amplitudes of the harmonics. A typical ADSR curve is shown in Figure 12.

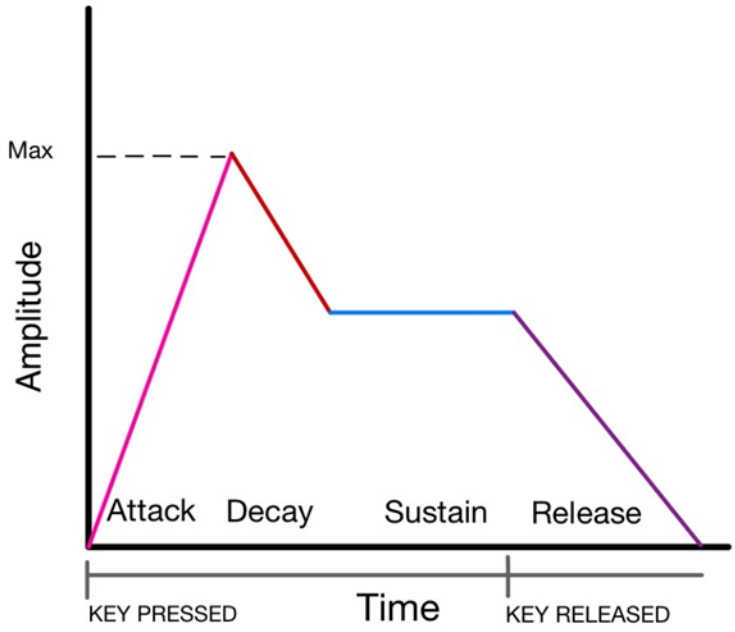

Figure 12: *ADSR Curve*

When the noteOn() function is called, the Attack part of the envelopes begins and sequentially progresses through the Decay and Sustain stages. The Attack stage of each envelope is set to 0.8 ms. The Decay stage for the fundamental harmonic is 800 ms, and that decreases by 200 ms for each higher-order harmonic. For example, the third harmonic above the fundamental decays over 200 ms. The Sustain levels for all envelopes were set to 0.9, meaning that during that stage, the signal is held at 90% of the peak amplitude.

Finally, the noteOff() function is called when the system detects a key release, triggering the Release stage of each envelope. This was set to 800 ms for all envelopes.

The audio connections, or signal routing, in the code specify how the harmonics, envelopes, and final audio output are formed. The fundamental and higher-order harmonic signals are each sent to their own ADSR envelopes for sound shaping. Then, all the envelopes are combined in a mixer and sent to the final audio output. The teensy audio mixer has a total of 4 inputs, which were fully

used in this audio output design. For any additional inputs, multiple mixers must be chained together.

The final audio output of Giano sounded like an electric piano. This was expected given the use of basic Additive Synthesis and FM, without additional effects such as timbral modulation or vibrato, which are often used to imitate an acoustic piano sound.

In practice, the Sustain stage should vary depending on the key press hold time. When this code was initially developed, it was assumed that the hold time could be taken in as a variable to dynamically change the sustain for a more piano-like sound. However, due to a limitation in the Teensy Audio Library, none of the ADS stages can be modified at the start of the envelope. Since there is no way to know the hold time before a key press, the Sustain stage was set to a fixed value for this project. However, future work can focus on experimenting with new libraries and other methods to further improve the audio.

The entire signal generation and mathematical formulas computed were implemented directly in the Teensy audio hat or the output firmware of the glove.

Parts and Implementation

Velostat Force/ Pressure Sensors

Description and Design

Velostat is a flexible, carbon-impregnated polymer sheet whose electrical resistance decreases under applied pressure. This property makes it well-suited for force and pressure sensing applications where flexibility, low profile, and robustness are required. In the GIANO system, Velostat sensors are used to detect finger press events on each glove, enabling the system to identify when and how firmly a user engages a virtual piano key.

Velostat was selected over alternative force-sensing technologies due to several key advantages. Unlike rigid force-sensitive resistors or mechanical switches, Velostat conforms easily to the curved surfaces of the hand and fingers, allowing seamless integration into a wearable glove. Additionally, it is inexpensive, lightweight, and durable, making it appropriate for an accessible and scalable assistive device. Its analog response to pressure also enables adjustable sensitivity thresholds through software, supporting reliable press detection without requiring precise finger positioning.

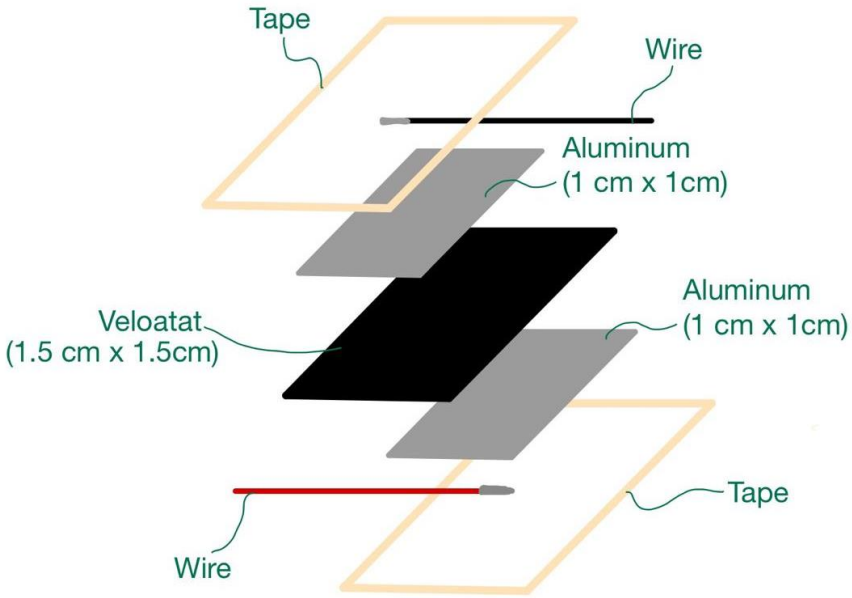

Figure 13: *Exploded view of the Velostat force sensor assembly.*

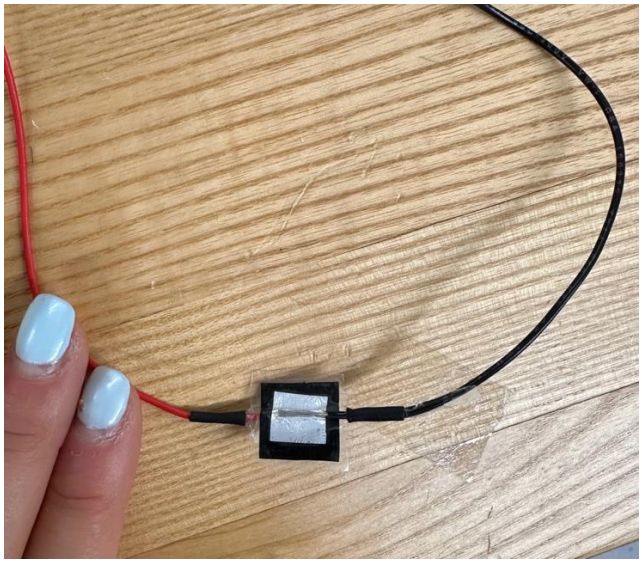

Figure 14: *Physical prototype of the final design for a custom Velostat force sensor.*

As seen in Figure 13, each Velostat sensor is constructed by layering the material between aluminum sheets acting as electrodes, with electrical connections made using lightweight conductive wiring. The sensor stack consists of adhesive tape for mechanical support, followed by a 24-gauge wire lead, an aluminum contact layer measuring 1 cm × 1 cm, the Velostat sheet measuring 1.5 cm × 1.5 cm, a second aluminum contact layer of the same dimensions, a second 24-gauge wire lead, and a final tape layer to secure the assembly. Aluminum was selected as the electrode material due to its low resistance, availability, and ease of integration. Care was taken to

ensure that only the exposed conductive portion of each wire contacted the aluminum layer, with the insulated section positioned away from the sensing surface. The use of tape provided a lightweight method for holding the layers together while preserving flexibility and comfort when embedded into the glove. The final version of the Velostat sensor is seen in Figure 14.

To interface the Velostat sensors with the microcontroller, each sensor is implemented as part of a voltage divider circuit using a fixed 1 MΩ (1000 kΩ) resistor. One terminal of the Velostat is connected in series with the resistor, and the midpoint between them is read by an analog input at the Teensy. As finger pressure varies, the changing resistance of the Velostat alters the voltage at this node, allowing the system to infer press events through analog-to-digital conversion. This approach provides a simple and reliable method for translating physical interaction into digital input suitable for real-time processing.

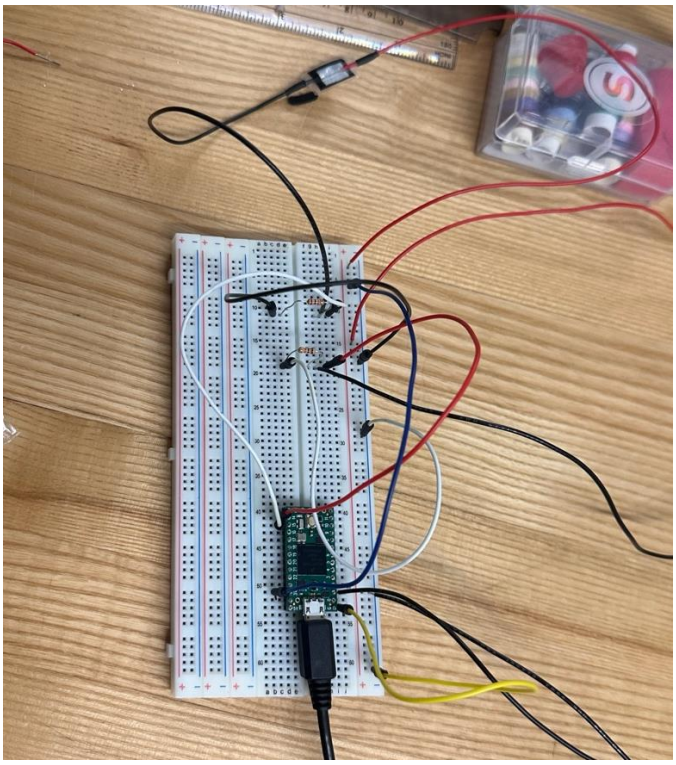

Figure 15: *Breadboard testing configuration used to validate two Velostat force sensors. Each sensor is configured in a voltage divider with a fixed resistor and connected to separate analog input channels for simultaneous press detection and comparison.*

Prior to integration onto the glove-mounted PCB, the sensing approach was validated through breadboard testing, seen in Figure 15. Individual Velostat sensors were connected in voltage divider configurations and evaluated under repeated press conditions to confirm consistent response behavior, as well as across multiple users with different hand sizes and pressure application patterns. A multimeter was used alongside the microcontroller to directly measure voltage changes across the divider, allowing verification of sensor functionality and baseline

behavior. In addition, a simple test program was written to continuously read and log analog values from the sensors, enabling real-time observation of signal changes under applied pressure before final hardware integration.

Overall, the Velostat sensing design supports GIANO's goals of comfort, affordability, and accessibility while providing dependable press detection suitable for real-time musical interaction.

Sensor Calibration

The Calibration Algorithm for the Velostat sensors is as follows: First, a sample set of 200 "Open" values was collected at a rate of 100 Hz. This was repeated for each finger, resulting in a 10-second "Open" calibration time. During this time, the user was instructed to scrunch their hands. This was done to ensure the "Open" sample set contains variance in the resistance values due to the stretching of sensors from common hand movements done while playing piano. When observing the distributions of these sample sets, it was noted that they all followed approximately normal distributions.

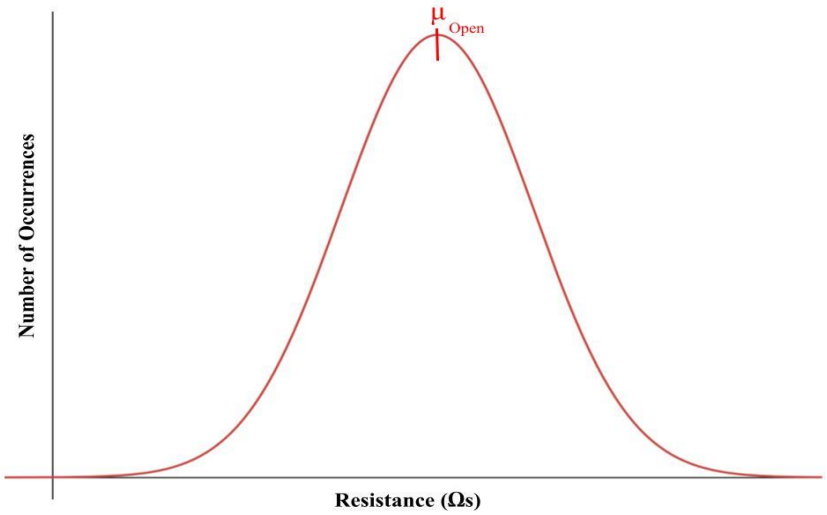

Figure 16: *Normal distribution curve*

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Eqn. 7: Probability Density Function of a Normal Distribution

This sampling process was then repeated for instructed hard presses against the piano mat,

yielding another sample set that was approximately normal. The values of these sample sets often overlapped, which led to interpretation of errors when the threshold was set too low or high.

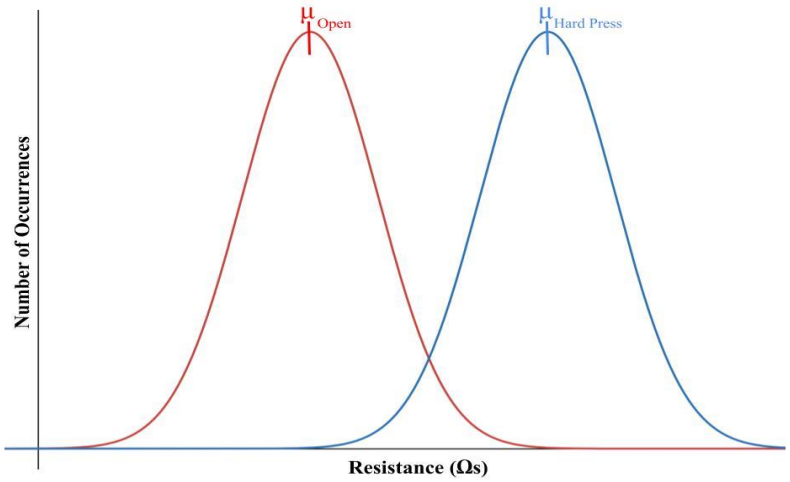

Figure 17: *Hard Press and Open Sample Sets as Distributions*

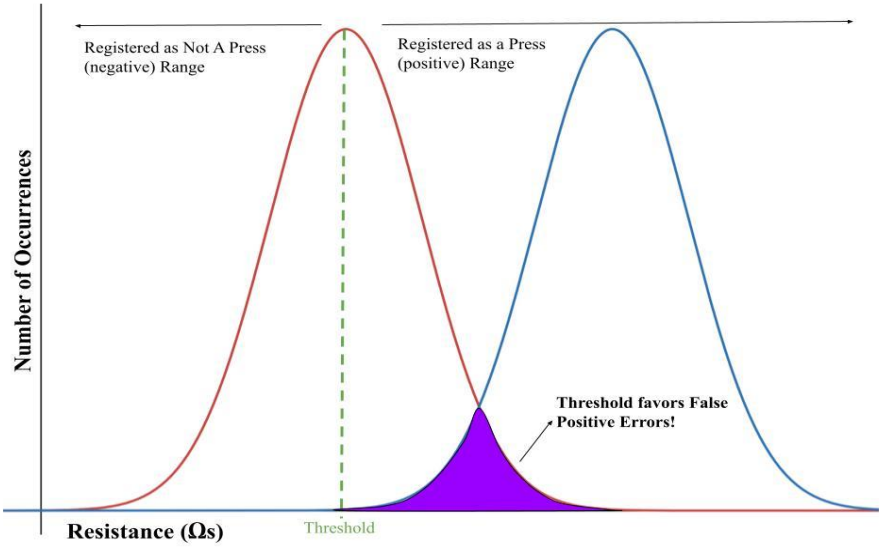

Figure 18: *Example of Threshold application leading to False Positive Errors*

It's crucial to distinguish the different outcomes that can occur when pressing or not pressing the Velostat sensors. Table 3 summarizes these in terms of True / False Positives and Negatives. A "True" outcome means the system reacted the way the user expected, while a false outcome means the opposite. Here, a Positive is considered a Value Registered, and therefore, a Note On message is sent to the Audio Unit.

User Action / Response	**Pressed a Key**	**Did not Press a Key**
Value Registered and Note Played	True Positive	False Positive
Value Registered and Note not played	False Negative	True Negative

Table 3: Declarations of False and True Positives / Negatives

From the user's perspective, these various outcomes can be variously frustrating and impact the user experience. These can be summarized by a loss function, where a low loss is seen as fitting to the user experience, while a high loss is seen as negatively detrimental to the user experience. True values work as expected and therefore have a loss of 0. A false negative would be frustrating as a user, but has low consequences as the user can try to press the key again harder. A false positive, on the other hand, could cause severe issues in Learning Mode and would be more frustrating.

$$\ell(TP) = 0 = \ell(TN) < \ell(FN) < \ell(FP)$$

Eqn.8: Loss Function for Each Outcome Compared

The loss functions indicate that for a better user experience, if the outcome must be false, false negatives should be prioritized over false positives. Therefore, the threshold of what is considered a press or not a press should lead towards preventing false presses instead of false negatives. This corresponds with setting the threshold to "cut" further into the pressed distribution.

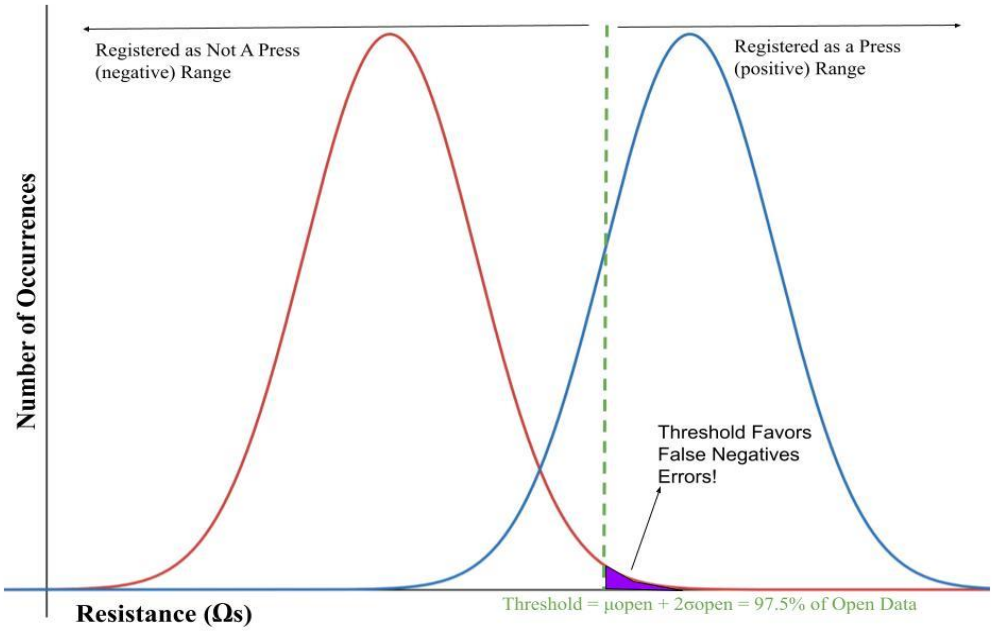

Figure 18: *Threshold Setting for Calibration Algorithm*

Since the distributions taken during the calibration sampling are approximately normal, they have the quality that 97.5% of their data falls at a point that is less than 2 standard deviations above their average. Therefore, to have a low error rate of 2.5 percentage points, the threshold of what's considered a press is set 2 standard deviations above the average value of the "Open" distribution. Values below this threshold are not considered attempted note presses and therefore have a Velocity of 0, while values above this threshold are considered attempted note presses and therefore have a velocity from 1-127.

Using the same strategy, 2 standard deviations above the hard press average were set as the max Press threshold. Resistances registered at this max press threshold were given velocity values of 127. This allows a wide range of key press hardness values and, therefore, dynamic range during playing time.

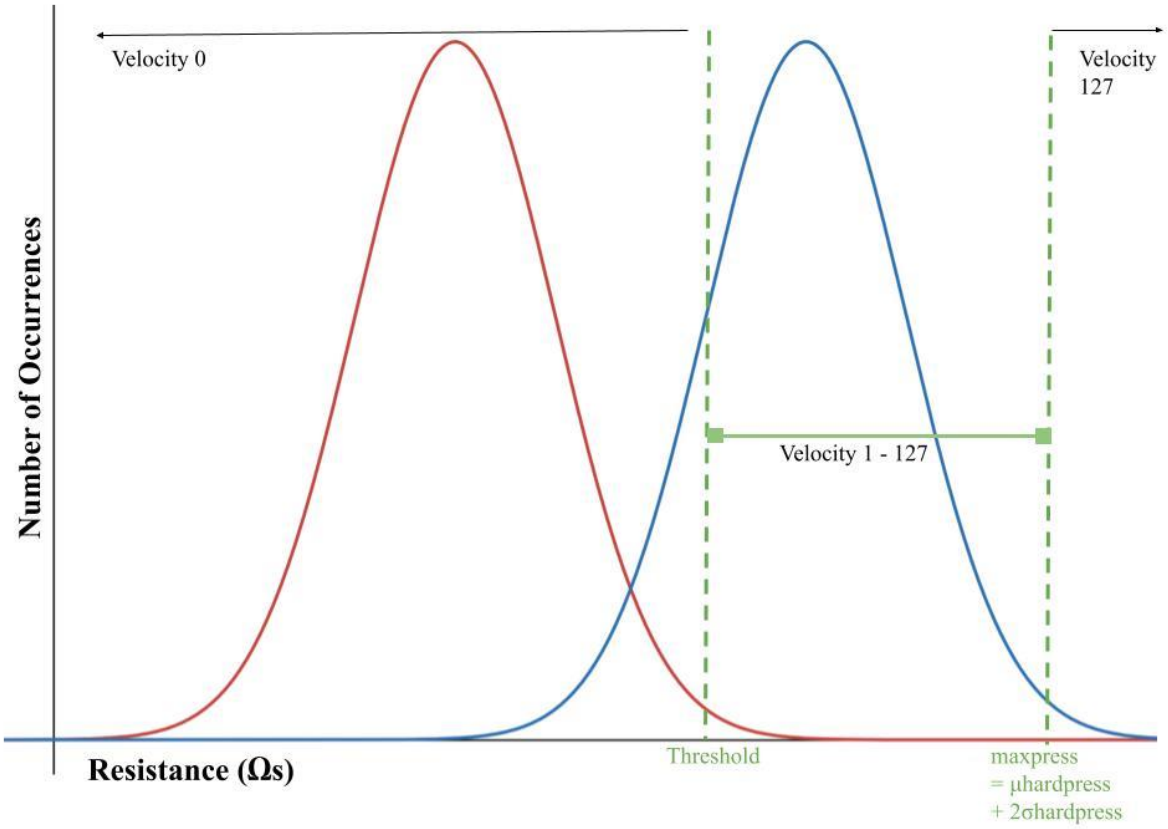

Figure 19: *Velocity Implementation using Calibration Algorithm Parameters*

Lastly, if the threshold is found to be above the mean of the hard press values for any finger, it can be concluded that something went wrong during calibration (such as pressing down during open calibration). In this case scenario, the entire calibration is repeated. A similar calibration process is done to calibrate the flex sensor, with two sample sets of bad and good postures being collected.

Challenges

Although building custom pressure sensors enabled control over sensor size and reduced costs, it also posed some challenges. For example, the initial build of the sensors was fragile, and the wires came apart from the tape, Velostat, and aluminum foil sandwich when tugged lightly. To solve this, the insulator part of the wire was superglued to the tape during assembly. However, this added extra resistance to the sensor, affecting its readings, which is why calibration was necessary. Each sensor was handmade and had a slightly different structure and amount of glue, which could not be perfectly controlled. Thus, calibration was vital.

Another issue occurred when heat shrink was added to the wires to improve insulation and further secure them from pulling out. The wires protruded rigidly from the sides, rather than bending easily for user comfort between the fingertips. This led to using less heat-shrink tubing than initially planned. In addition, caution was required during heat shrinking, as the tape began to melt, suggesting the future need for an alternative adhesive material that can sustain higher temperatures.

Analog Sensor Reading

Velostat Voltage Divider Implementation

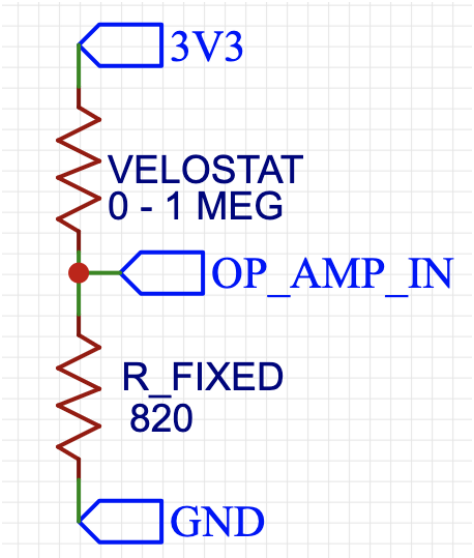

Figure 20: *Voltage divider for Velostat sensor voltage.*

The custom Velostat sensors were placed into voltage divider circuits in order to register different voltages based on pressure on the sensor. All Velostat sensors exhibited the same behavior of increasing resistance with pressure. However, the upper limit of resistance under a hard press varied between fingers, players, and sensors. Often, a "Hard-Press" could vary from thousands to millions of Ω. Due to these unique scenarios, a custom algorithm was needed to decide the baseline resistance (and hence voltage value) that would be used to distinguish between presses and non-presses.

Sensor Buffering

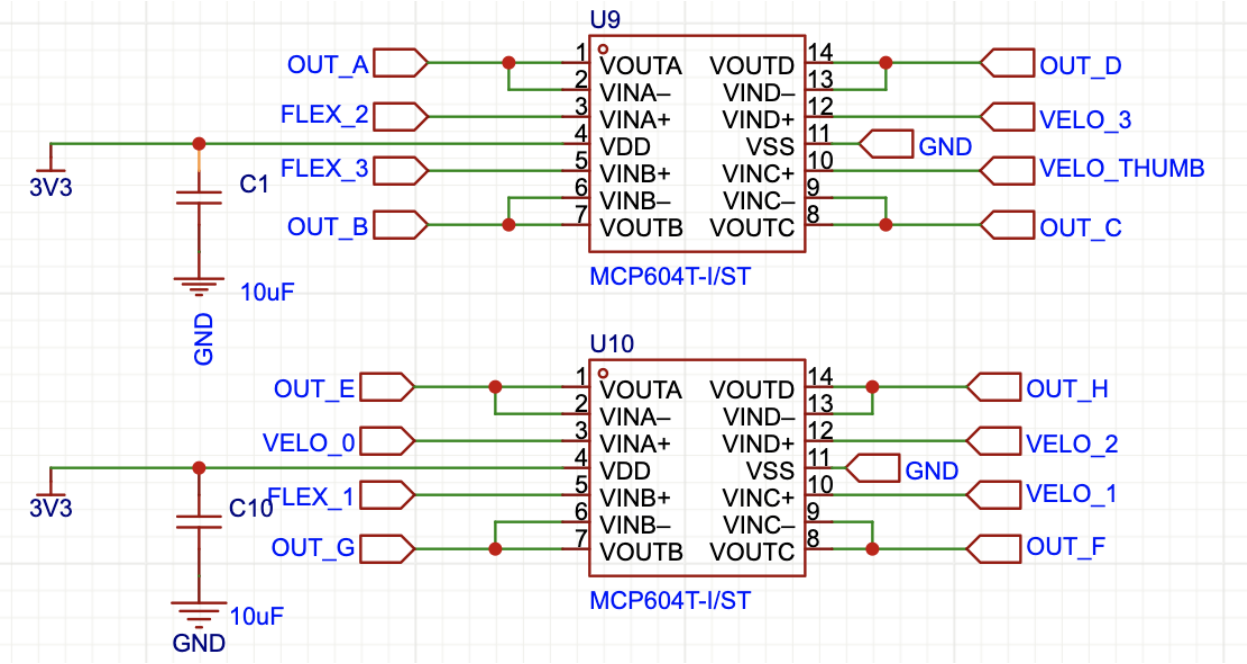

Figure 21: *Op amp buffers for analog inputs.*

High-impedance inputs can cause the sampling frequency to fall on many ADCs. To prevent this issue, the Velostat and Flex sensor voltage signals were buffered through an MCP604 Operational Amplifier configured as a voltage follower before being sent to the Teensy's Analog inputs as seen in Figure 21.

Haptic Motors

Motors

The haptic feedback of the learning mode is achieved using Adafruit's Mini Motor Disc, seen in Figure 22. These are 10mm diameter eccentric rotating vibrating motors (ERM) that operate between 2-5V and draws 40-100mA.

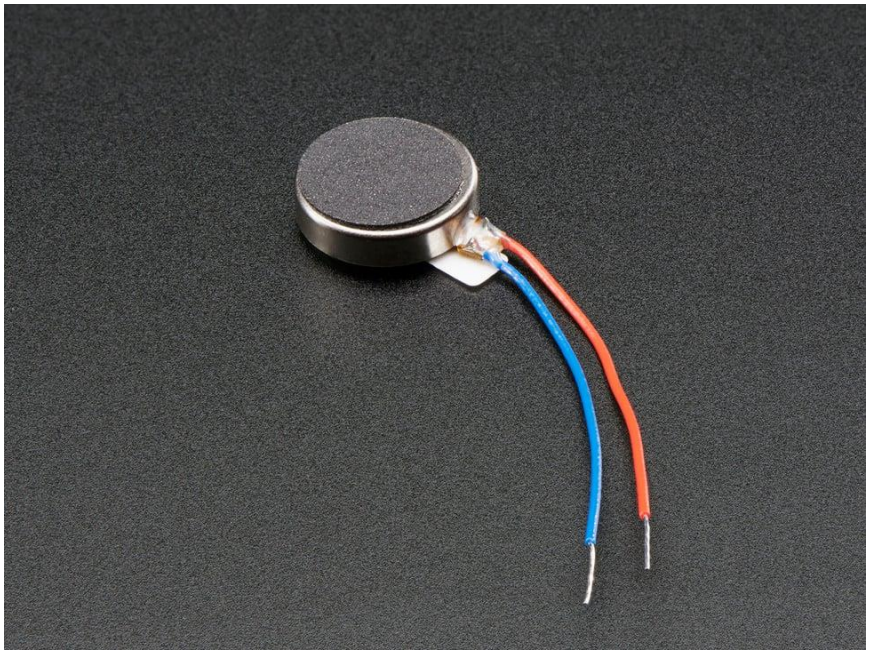

Figure 22: *Adafruit mini ERM motor disc.*

Motor Driver

Each haptic motor is driven by a DRV2605L motor driver, seen in Figure 23 with the BGA package for compact PCB layout. This driver is controlled by I2C and has a fixed address. The DRV2605L has 123 built-in haptic effects generated internally via PWM, which prevents the need for custom PWM generation. This driver can drive up to 150mA while operating at a 2.4-5.5V supply voltage.

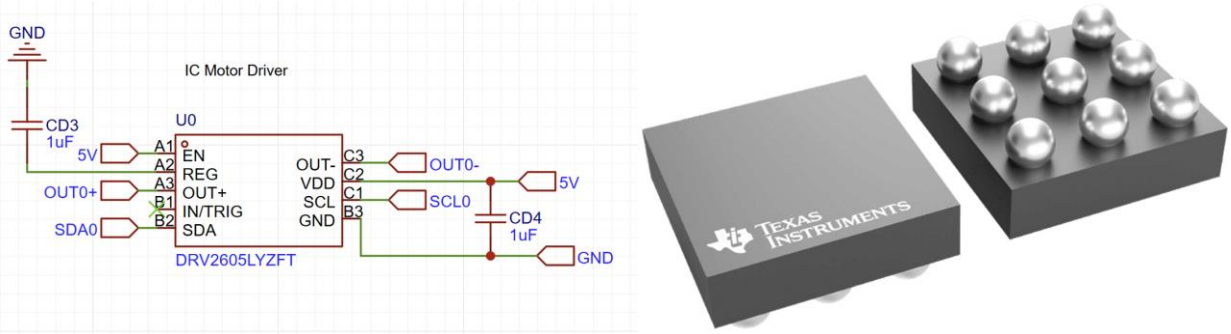

Figure 23: *Motor driver circuit and BGA package.*

I2C Mux

Since the DRV2605L has a fixed I2C address, Giano's haptic system uses a TCA9548A eight-channel I2C multiplexer, with a VQFN package, to communicate with multiple drivers, as seen in Figure 24. The Teensy 4.0 communicates between the seven motor driver circuits (channels 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40) through one I2C bus. The multiplexer's address is set via the A0, A1, and A2 pins; the resulting address is 0x77 when all are set high.

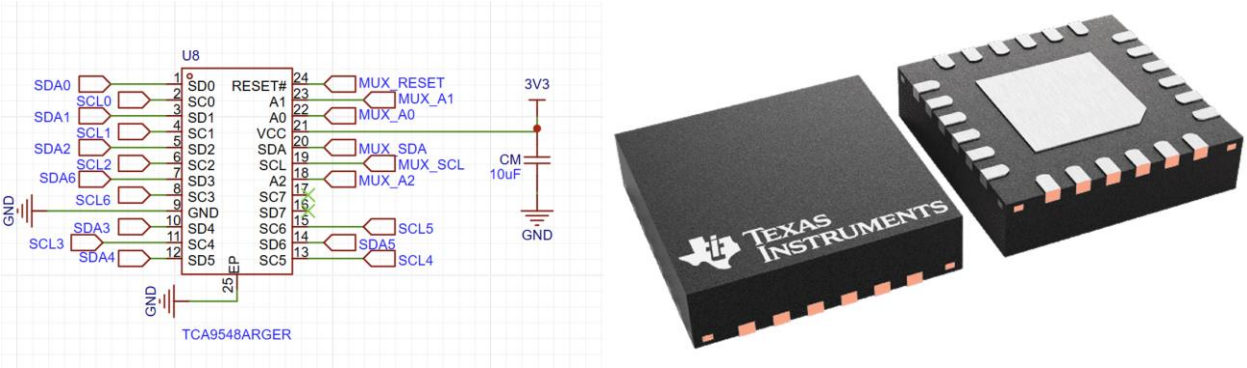

Figure 24: *I2C multiplexer and VQFN package.*

Power and Protection

Power Consumption

To ensure proper operation of components, power and current calculations were done to ensure worst-case maximum power and current requirements. Two voltage levels were used: 3.3V for logic and voltage dividers, and 5V for the microcontroller and haptic motor power. Worst-case currents were calculated for both lines, and then the total power was calculated for the device. Note the 3.3V Linear Drop Out Regulator covered in later subsections does not appear in Tables 4-6. The total worst-case power consumption of the glove was around 5W.

Component	Name	Max Current Draw (mA)	Number on Glove	Total Current Draw (mA)
I2C Multiplexer	TCA9548	0.035	1	0.035
Velostat / Flex / Button Voltage Divider Circuits	-	4.125	10	41.25
Op – Amp Buffer	MCP604	2.6	2	5.2

Table 4: 3.3 V Line Worst-Case Current Calculations

Component	Name	Max Current Draw (mA)	Number on Glove	Total Current Draw (mA)
Microcontroller	Teensy	250	1	250
Haptic Motors	-	100	7	700
Motor Drivers	DRV2605L	0.65	7	4.55
Motor Driver Pull - Ups	-	0.606	14	8.5

Table 5: 5V Line Worst-Case Current Calculations

Line	Total current (mA)	Total Power (mW)
3.3V	47.68	157.3
5V	963.03	4815.7
Total:	-	4972

Table 6: Worst Case-Current and Power per Glove

Power Supply

The primary power source of the PCB was an MTDZKJG wall adapter, as seen in Figure 25, which provides a nominal 5V 5A DC output. Based on the Power Ratings calculated in Table 6, it was chosen to meet the 4.972W / 1~A worst-case power/current of the glove.

Figure 25: *5V DC wall power supply.*

3V3 Voltage Regulator

To generate the 3.3V rail required by the resistor circuits, I2C multiplexer, and buttons, the circuit in Figure 26 is used with a TPS73633DCQR linear voltage regulator that provides up to 400mA output current.

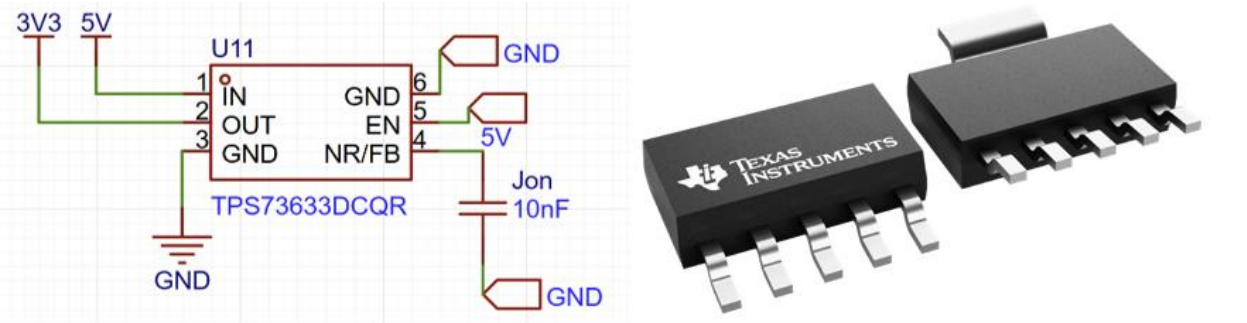

Figure 26: *Voltage regulator circuit.*

Fuse

For circuit protection, a resettable fuse (MF-SM100-2) was added to the main power input as seen in Figure 27, to provide overcurrent protection. When excess current is detected, the resistance increases to limit current. Once the fault is removed and the device has had time to cool, the fuse automatically resets, without the need for replacement.

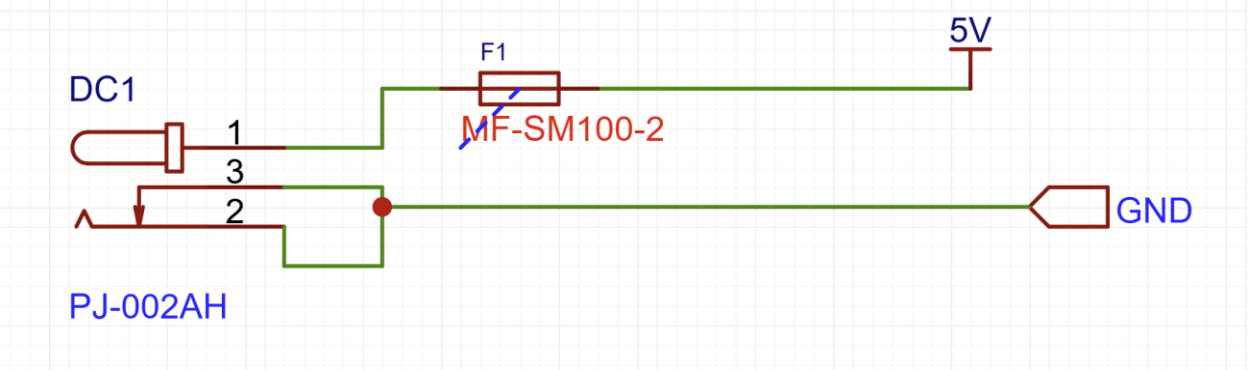

Figure 27: *Fuse protection circuit.*

Connectors

JST Connector

JST-XH connectors, shown in Figure 28, are used to connect the sensors and haptic motors to the PCB, and allow for easy removal and replacement. The PCB includes a range of connector sizes, including 2-pin, 4-pin, 5-pin, 7-pin, and 8-pin connectors. Figure 28 also shows the DC barrel jack connector for the 5V DC power input.

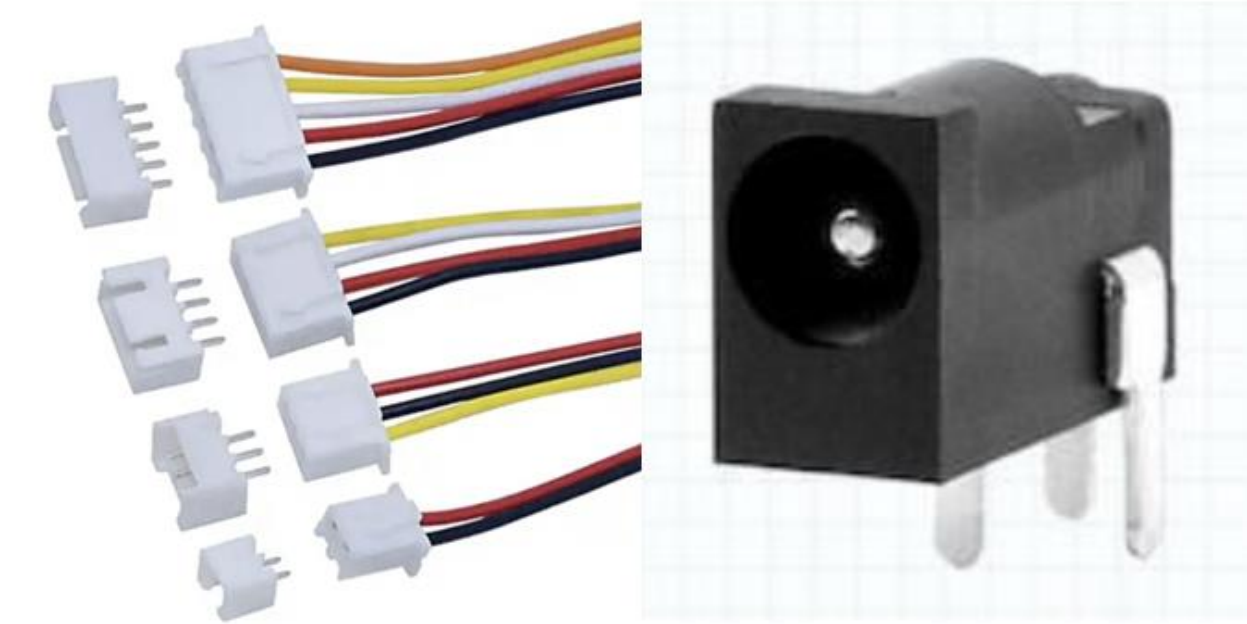

Figure 28: JST-XH connectors for sensors and haptic motors(left), and a DC barrel jack connector for power (right).

PCB and Enclosure

The PCB uses a four-layer stack-up (5V, GND, 3V3, Signal), shown in Figure 29, selected to ensure proper power integrity, simplify routing, and allow the SDA and SCL lines to be routed in parallel. The top layer includes all of the connectors, op amps, regulators, test points, buttons, the fuse, and headers. The bottom side of the PCB includes the seven motor drivers and the multiplexer. The 3D printed PCB enclosure can be seen in Figure 30 made of PLA filament, with cutouts for the connectors, Teensy, and buttons. A sliding lid was used to allow for easy access to the PCB for testing and debugging. Connector layouts allow the same PCB design to be used for both the left and right hands.

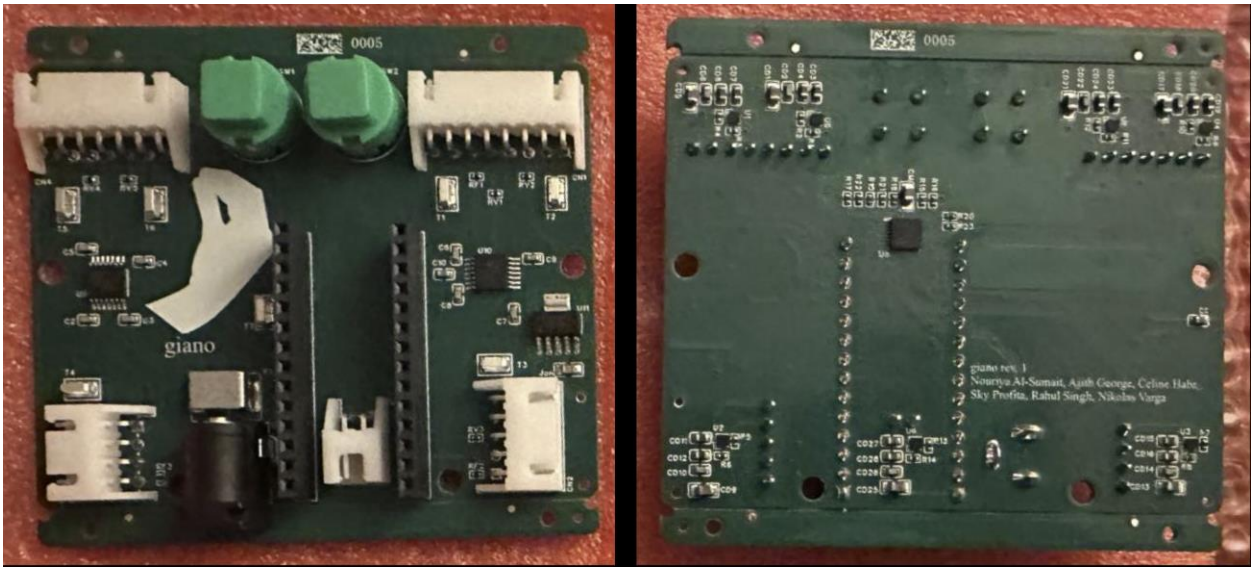

Figure 29: Top (left) and bottom (right) sides of the PCB.

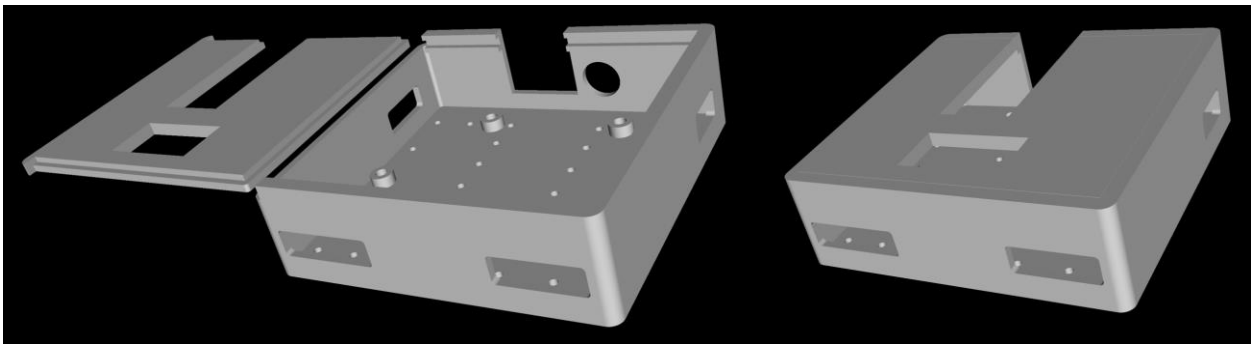

Figure 30: PCB enclosure with the sliding lid off (left) and sliding lid on (right).

Glove Exterior Design

The final glove assembly focused on securely integrating all components, including the PCB, sensors, and motors, onto the glove while maintaining accessibility if replacement is necessary.

A Velcro strip was glued to each Velostat sensor to stick to the glove fingertips easily. Especially since these sensors are handmade and may malfunction, the ease of replacing them was crucial to avoid issues with the glove's overall function. This was found to be the better option rather than gluing them to the glove directly, which can increase resistance, or sewing them, which can destroy the sensor layering.

Each haptic motor was placed into a hand-sewn pouch. These pouches were roughly 2 x 3 cm and included a Velcro strip for easy opening and closing. The pouches, including the motors, were placed slightly above the knuckles on each finger and on the left and right sides of the wrists for octave guidance.

A flex sensor was glued to the middle finger of each glove to correct hand posture when playing. In future iterations of GIANO, additional flex sensors can be mounted in this manner. An image of the palm side and back of the glove can be seen in Figures 31 and 32, respectively.

Figure 31: *GIANO glove (palm side)*

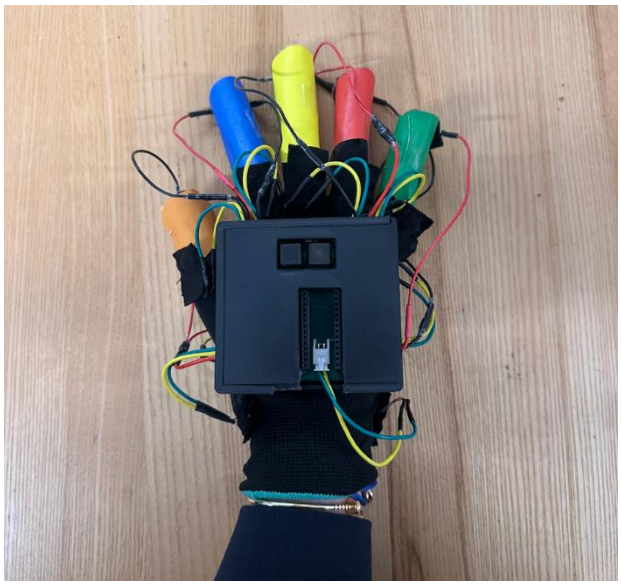

Figure 32: *GIANO glove (back-of-the-hand side)*

As shown in the images above, the color markings for CV finger identification were printed and affixed to the tops of each finger, without blocking the pressure sensors on the tips. Additionally, each PCB was placed into a custom 3D-printed case, which was attached to the glove using Velcro. These design choices ensured the protection of all electronic components, accurate sensing, and the ease of component replacement if necessary.

Cost Analysis

Part Name	Quantity	Manufacturer	Price ($)
Haptic Motors	15	Adafruit	26.40
DRVL20605L Haptic Motor Driver*	1	Adafruit	7.95
Short Flex Sensor	10	Adafruit	71.60
Teensy Audio Adapter Rev D	1	Digikey	9.80
Teensy 4.0	2	Digikey	47.60
Teensy Stackable Header Kit*	1	Digikey	1.80
Raspberry Pi 5b 8 Gig*	1	Microcenter	79.99
Adafruit TCA9548A I2C Multiplexer	1	Microcenter	10.99
SPI ADC*	2	Digikey	12.42
Velostat Sheets	2	Digikey	9.90
CMOS Op-amp*	2	Digikey	7.00
Camera	1	Amazon	21.98
PCBA	1	JLBPCB	259.59
Tariffs of PCBA	-	-	57.11
Shipping	-	Digikey	7.00
		Total	663.09

** not used in the final product*

Table 7: *Bill of Materials including price, manufacturer, quantity, and price.*

Table 7 summarizes the bill of materials (BOM) and associated costs for the GIANO prototype, including both development and projected manufacturing expenses.

The total research and development cost for the GIANO prototype was $663.09. The largest contributors to this cost were the custom PCB fabrication, the embedded computing hardware (Raspberry Pi 5 and Teensy microcontrollers), and the haptic and sensing components used for iterative testing and validation. In particular, PCB fabrication and associated tariffs accounted for a significant portion of the budget, reflecting the one-off nature of prototype manufacturing.

If GIANO were to be commercially manufactured, the estimated per-unit cost would decrease to approximately $586.78. This reduction is primarily due to the removal of development-only components, including standalone haptic motor driver boards, the Teensy stackable header kit, external ADCs, a CMOS Op-amp, and the I2C multiplexer, which are integrated directly into the final custom PCB design. As a result, fewer discrete components would be required in a production-ready system.

Additionally, bulk manufacturing would further reduce costs through volume pricing on electronic components, PCB fabrication, and assembly. These economies of scale suggest that GIANO could be produced at a lower per-unit cost in a commercial setting, improving affordability while maintaining the system's core functionality and accessibility goals.

Beyond technical feasibility, affordability is central to GIANO's mission of accessible music education. By minimizing reliance on expensive instruments, paid lessons, and proprietary hardware, GIANO demonstrates that assistive music-learning technology can be developed at a relatively low cost. The projected reduction in manufacturing cost further supports the system's scalability and potential for wider adoption, particularly for users who are financially constrained or lack access to traditional piano education resources. This cost-conscious design reinforces GIANO's goal of lowering barriers and expanding equitable access to music learning.

Timeline

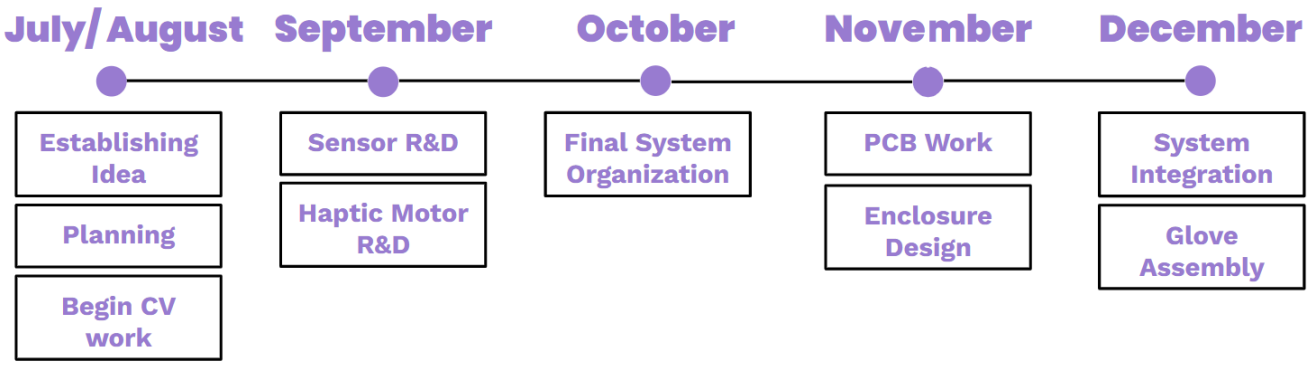

Figure 33: *Timeline for Giano*

During the summer, the project began with outlining and planning Giano. This involved narrowing down the problem Giano must address and conducting extensive research on potential components, including sensors and haptic motors, to ensure they can support the integration of free play and learning modes. Frequent team meetings were held, and CV-related work began.

In September, the focus was on research and development as well as testing to finalize components and ensure they were suitable for the gloves. In October, more hands-on work continued, including the design, build, and testing of the custom Velostat pressure sensors. This was also when the system architecture became clear and planning of the overall system integration was done. In November, the PCB was designed, ordered, and tested to ensure all components interfaced correctly and met the project's requirements. The external glove design also began, including stitching custom pouches for the haptic motors and attaching the motors and Velostat sensors using Velcro to the glove. Finally, in December, the entire system was integrated, including CV, audio output, firmware, and hardware sensors and motors, and the final PCB case for the gloves was printed. The culmination of this work was presented during Capstone presentations at the

beginning of December, when the prototype of the glove operating in free-play mode was shown to the judges and audience.

Conclusion

All in all, GIANO demonstrates the feasibility of a low-cost piano playing and learning system that removes traditional barriers to music education, including paid instruction, visual feedback, and costly equipment. With a paper piano keyboard and haptic feedback, GIANO makes piano learning and playing accessible to everyone. The free-play mode was fully functional, producing realistic piano sounds via FM and additive synthesis when a user pressed a key.

The CV unit in the design fully identified which key was pressed, detecting multiple key presses at once and accurately determining where fingers were positioned relative to a low-cost paper piano keyboard. The CV unit also guided the user to the correct key during learning mode. The setup for the gloves was visual-free, meaning all instructions and cues were delivered via sound and button presses.

With more time for this project, the learning mode could be further integrated with the gloves to teach users how to play the piano using haptic feedback. Additional work can also be done to improve each subsystem, including audio output and CV, as well as the overall integration of hardware, software, and firmware.

References

Problem formulation:

[1] E. A. Miendlarzewska and W. J. Trost, “How musical training affects cognitive development: Rhythm, reward and other modulating variables,” Frontiers in Neuroscience, vol. 7, no. 279, pp. 1–18, Jan. 2014, doi: https://doi.org/10.3389/fnins.2013.00279.

CV:

[2] S. Garrido and A. Panov, “Detection of ArUco Markers,” OpenCV Documentation, v4.13.0, https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html.

[3] Eric Marchand, Hideaki Uchiyama, and Fabien Spindler. Pose Estimation for Augmented Reality: A Hands-On Survey. *IEEE Transactions on Visualization and Computer Graphics*, 22(12):2633 – 2651, December 2016.

[4] SharkD via Wikimedia Commons