Imperial College London

Human Centred Design of Assistive and Rehabilitation Devices

Imperial College London

Department of Bioengineering

Final Report

Team Name:
Game5ense

Authors:
Byron Chan, Megan Le, Ken Mah, and Matthew Wong Sang

May 9, 2024

Acknowledgement

We extend our heartfelt gratitude to Etienne, Paschal, Jacopo, Lucille, and Alexis, whose invaluable contributions were instrumental in bringing this project to life. Without their support and expertise, this project would not have been possible!

Contents

1 Introduction

Video games transcend mere entertainment, serving as a versatile platform for narrative exploration, skill enhancement, and social connectivity. Approximately 42 million gamers face disabilities [1], with the perception of directional audio presenting a considerable challenge for those with hearing impairments. Innovative accessibility features, such as directional subtitles and visual cues, have made significant progress toward fostering an inclusive gaming landscape. Notably, professional gamers like FaZe EwOk have leveraged these audio cues to excel in competitive arenas, underscoring the advancements in inclusive gaming technologies [2].

However, these interventions, often bespoke to specific game titles, lack universal integration and over-rely on visual elements. This over-reliance can diminish the gaming experience for users with hearing disabilities, as it fails to effectively substitute the rich nature of auditory feedback.

In response to these limitations, we created an innovative solution called Game5ense. The proposed solution involves the development of a wearable device designed to translate in-game audio into haptic feedback. This approach aims to inclusively bridge the sensory divide, enabling individuals with hearing impairments to better engage with video games.

2 Design Specifications

Game5ense was engineered with interoperability, comfort, and adaptability in mind, ensuring a universally compatible and user-friendly experience for individuals with hearing impairments. The goal was to create a device which seamlessly integrates with various gaming platforms via standardised audio interfaces and software protocols, allowing for broad compatibility and future-proofing against evolving gaming technologies. Simultaneously, the design would emphasise user comfort, utilising lightweight, hypoallergenic materials to ensure it can be worn comfortably for extended periods. Lastly, for adaptability, the device would include features allowing users to customise vibration intensity, catering to personal preferences. This multifaceted approach ensures the device not only enriches gaming experiences across different platforms but also accommodates individual user needs for a truly inclusive gaming environment.

3 Device Design and Development

The device operates on the fundamental principle of converting auditory signals into tactile feedback. It comprises both software and hardware components. The software is responsible for capturing and processing the audio output from video games, identifying directional and environmental cues, and transmitting this information to the hardware. The hardware, designed based on the specifications highlighted previously, executes the delivery of haptic feedback based on the processed audio cues. The evolution of both, from concept to development, is detailed below.

3.1 Software Design

A crucial component of the device is the software that translates surround sound audio into haptic feedback through different levels of vibration. Specifically, our device captures 5.1 surround sound, which consists of two front speakers (left and right), two back speakers (left and right), a center speaker, and a subwoofer [3]. This 5.1 surround sound audio is captured from video games via the user's computer speakers in real-time, utilising third party audio jack tools, such as BlackHole for macOS [4] and Voicemeeter for Windows [5]. This allows our device to function across multiple platforms with minimal latency. Once the surround sound audio is captured, it is then processed using Python as an interpreter, alongside audio-related libraries, such as PyAudio [6].

During audio processing, the audio is filtered and thresholded so that the signals can be sent to the vibration motors at three specified levels (no vibration, medium vibration, and high vibration). The raw audio is filtered using a simple moving average filter to reduce noise and smoothen the signal. This filtered audio is then converted into three discrete levels of vibration using two thresholds. These thresholds were determined by conducting experiments to collect audio data and plot it while performing specific actions within the video game. Three different levels of vibration were used so that it is easy to differentiate between the physical sensations of the vibrations and thus learn the game-specific meanings of each level.

In the early development stage, we tested our audio processing algorithm in Minecraft [7], capturing stereo audio to start as it was easier to solve initial complications when working with only two audio channels. Once the software component was fully developed and functioned as expected, we tested it on surround sound audio using Fortnite [8], a popular free-to-play online multiplayer game which supports 5.1 surround sound. However, this experimental methodology can be similarly applied to other video games, such as Counter-Strike [9], Apex Legends [10], and Call of Duty [11].

For the lower threshold, audio data was collected in real-time while playing Fortnite, specifically during walking, sprinting, and sliding (Figure 1). From this data, the baseline threshold was determined such that audio intensities below this, resulted in no vibration. Likewise, the upper threshold was determined by collecting audio data related to the loudest in-game sounds such as sky-diving, driving and especially, shooting (Figure 2). Audio intensities above this upper threshold resulted in the highest level of vibra-

tion, whilst sounds between these thresholds, such as in-game notifications, resulted in a medium level of vibration. Notably, audio intensities also varied between different channels of the 5.1 surround sound (front versus back speakers), therefore, the thresholds were made specific to each audio channel. Additionally, testing revealed that the center and sub-woofer audio channels did not contain essential auditory information from video games, and therefore, our final prototype only captured audio from four channels of 5.1 surround sound: the two front and back speakers. This design change provided more intuitive directional audio cues from the vibration motors during gameplay.

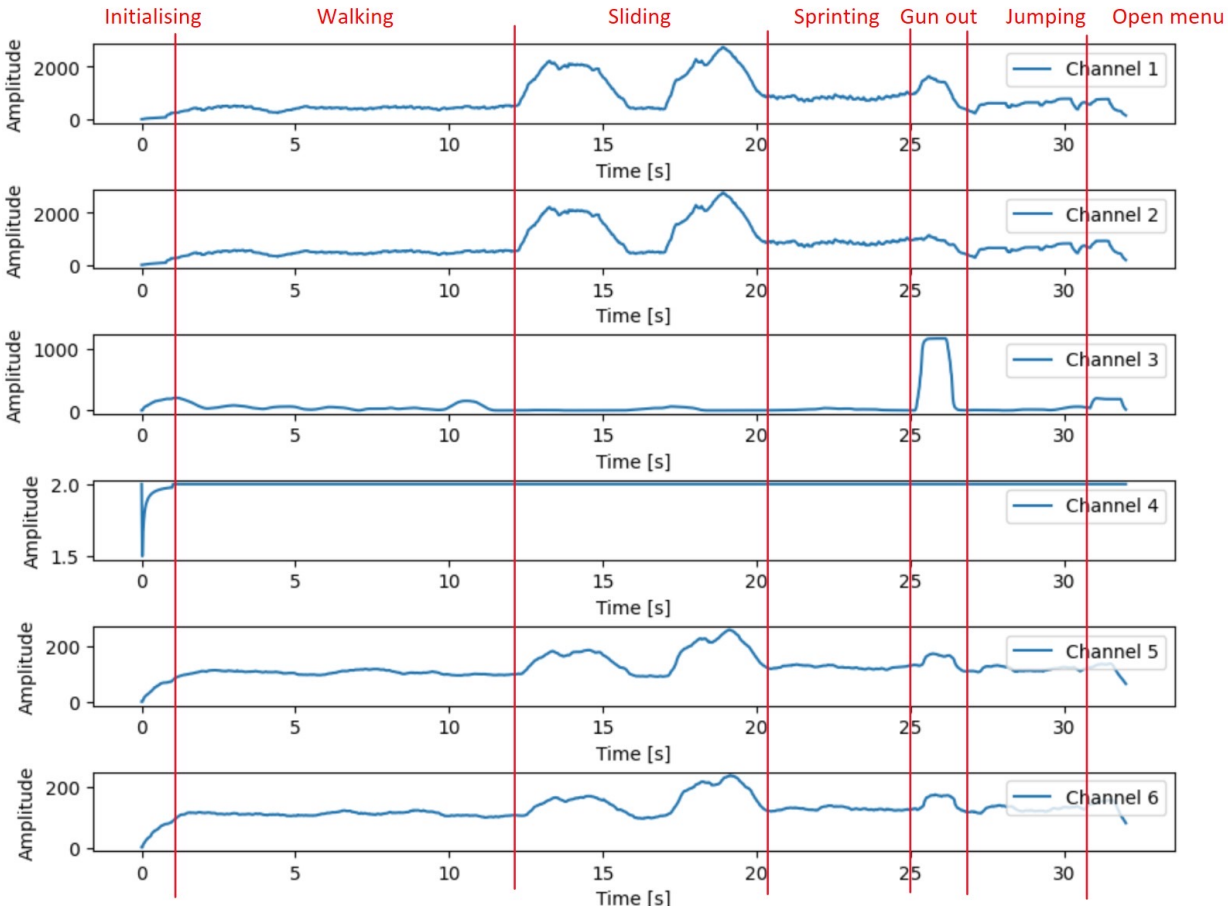


Figure 1: Audio intensities of soft in-game sounds (player movement actions) across each audio channel in 5.1 surround sound in Fortnite. Channel 1 is the front-left audio channel, Channel 2 is the front-right, Channel 3 is the center, Channel 4 is the subwoofer, Channel 5 is the back-left, and Channel 6 the is back-right.

Once the audio is fully filtered and thresholded, the audio intensities are assigned to a value between 0 and 255 pertaining to the three specified levels of vibration, which is sent as an array via serial connection to an Arduino Uno microcontroller [12]. The Arduino Uno then writes each value from the array to its corresponding vibration motor.

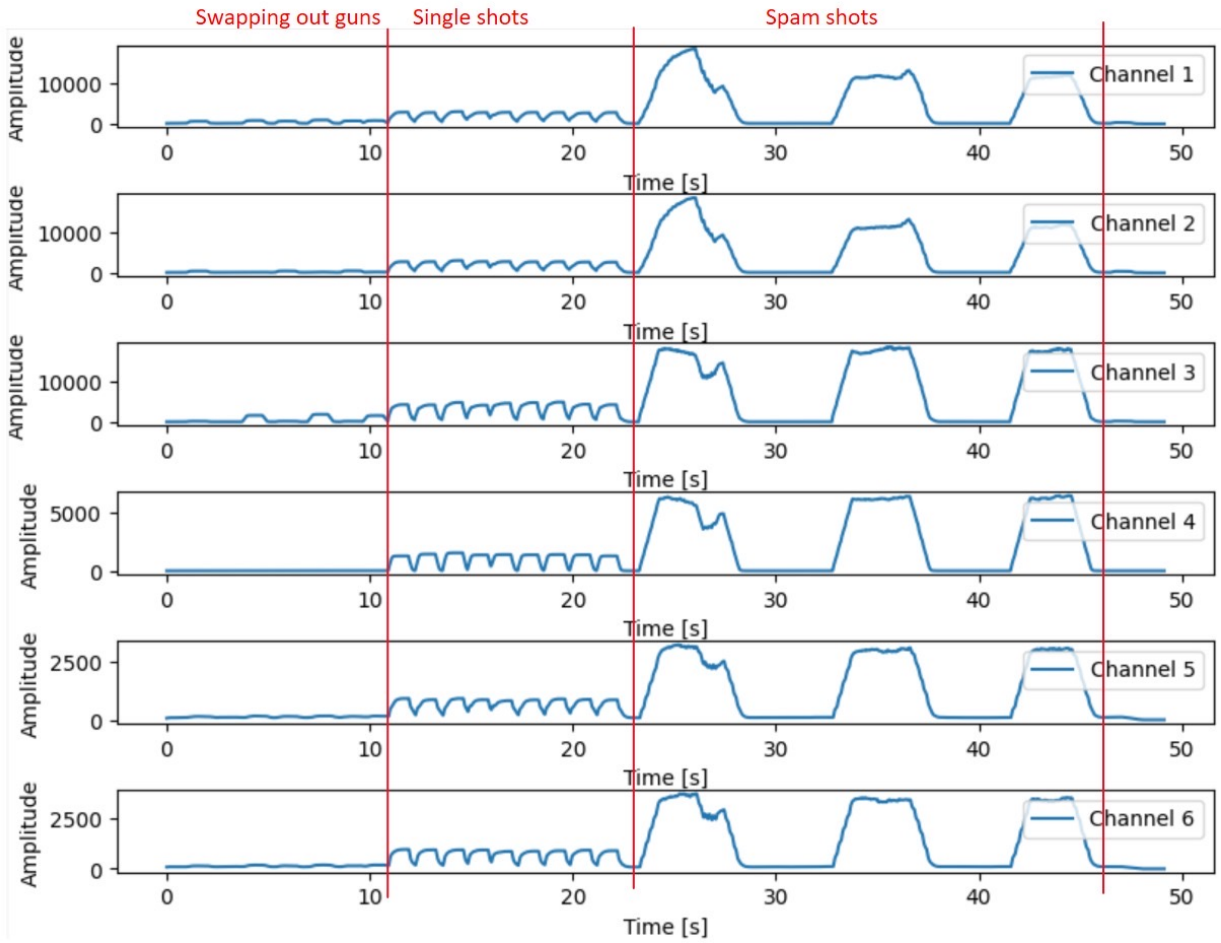


Figure 2: Audio intensities of loud in-game sounds (shooting) across each audio channel in 5.1 surround sound in Fortnite. Channel 1 is the front-left audio channel, Channel 2 is the front-right, Channel 3 is the center, Channel 4 is the subwoofer, Channel 5 is the back-left, and Channel 6 the is back-right.

The full range of values were experimented with, but larger values (>100) were found to lead to user discomfort. Therefore, in the final software implementation, values of 0 (no vibration), 75 (medium vibration) and 100 (high vibration) were used. Our audio processing algorithm can be seen in Appendix A, along with our arduino code in Appendix B.

3.2 Hardware Design

Another critical aspect is the hardware design, as it directly interacts with the user. Adhering to the design specifications, the hardware is designed to be compact, adjustable and lightweight, with minimal system delay. The hardware design of the device can be broken down into two parts: the circuit and the wearable component.

3.2.1 Circuit

Initially the circuit had a simple design, powering two vibration motors directly from the Arduino (based on stereo audio). Utilising a breadboard and connecting wires, this design allowed for a successful proof of concept demonstration. Throughout this early testing phase, light emitting diodes were frequently used in place of the vibration motors as they were easier to connect to the breadboard.

After establishing feasibility, the circuit was scaled up to support four vibration motors in parallel with each other. In this process, a handful of problems were identified with the initial design:

1. The voltage at the output pins of the Arduino varied even when they were coded to have the same value.
2. Digital pins 5 and 6 of the Arduino ran on a different clock causing the pulses from these pins to be out of sync with the other pins. This was found to be an inherent feature of the select Arduino Uno R3.
3. The voltage across each vibration motor could not be manually adjusted, which limited user friendliness.
4. There were no protective measures in the circuit (e.g. against voltage spikes).

To address these issues, three improvements were made to the previous design. Firstly, metal-oxide-semiconductor field-effect transistors (MOSFETs) to control the motors were added. MOSFETs were selected over bipolar junction transistors because of their fast-switching property. Secondly, 'flyback' diodes were added to protect against potential voltage spikes. Thirdly, potentiometers were added to allow for manual control of the voltage across the vibration motors. The final circuit design as shown in Figure 3 allows for better safety and control of the vibration motors.

For the implementation of this circuit, an Arduino Uno [12], 4 Seeed Studio 316040001 mini vibration motors [13], 4 IRF510 MOSFETs [14], 4 diodes, and 4 10k slide potentiometers were used. These components were selected primarily based on availability, although their data sheets were checked to ensure safety under the expected voltages and currents. The Arduino's digital PWM pins 3, 5, 9, and 10 were used. The fully connected and soldered circuit is shown in Figure 4.

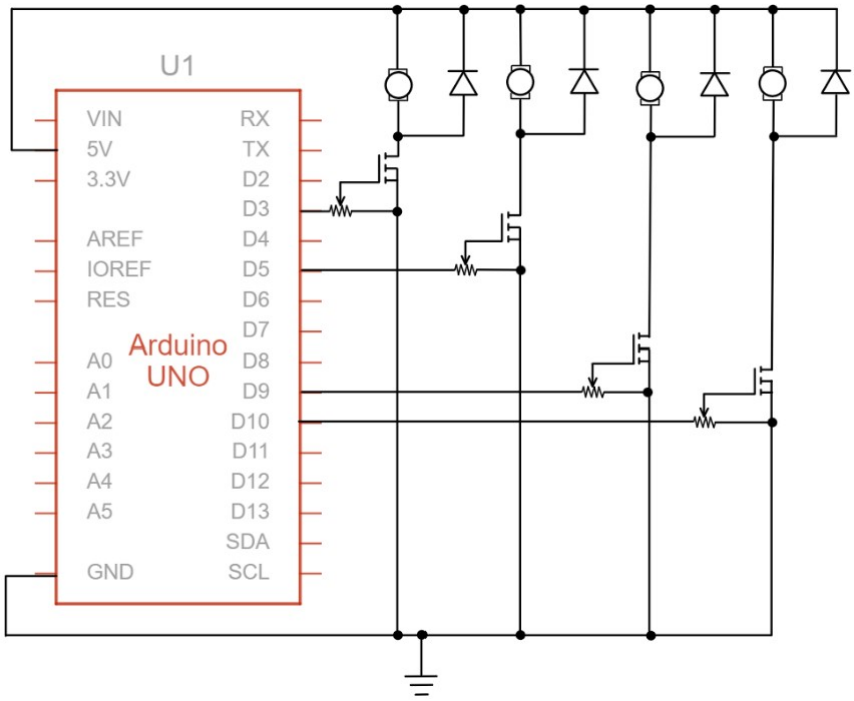


Figure 3: Final Circuit design for the device

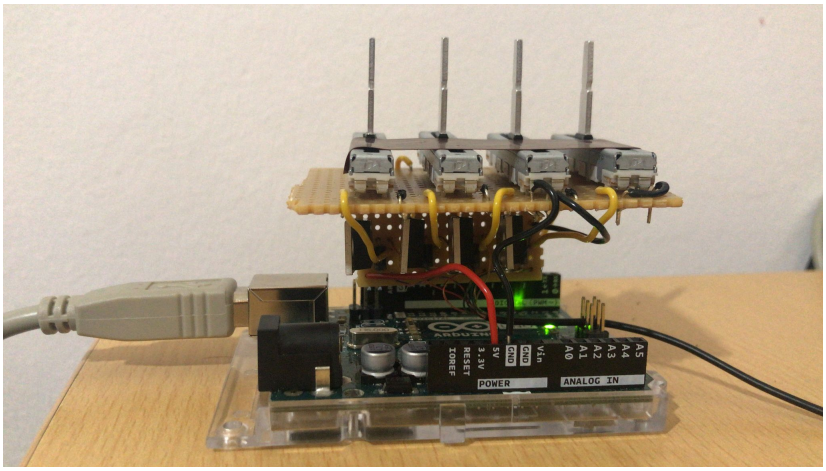


Figure 4: Circuit of the final prototype of our device.

3.2.2 Wearable Component

For the haptic feedback mechanism to be immersive and engaging, a head-worn wearable was selected. To facilitate a clean and organised connection between the circuit and the wearable, a flexible multi-conductor cable was employed. The initial prototype utilised a standard cap, leveraging its inner seam for seamless integration of vibration motors. This cap offered comfort and adjustability to accommodate various user sizes. However, the cap's design limited the spatial arrangement of the vibration motors due to its shallow fit.

Consequently, the development progressed to a second prototype employing a headband.

This iteration maintained the virtues of adjustability and comfort while providing enhanced flexibility in the positioning of the vibration motors, both horizontally and vertically around the head. Despite this improvement, the headband's elastic material introduced a challenge regarding the integration of motors, as it allowed for stretching that the wires could not accommodate.

The final prototype (Figure 5) adopted a bandana as the wearable component. This design choice retained the advantages of the previous iterations, including adjustability and comfort, while effectively addressing the challenges related to motor placement and integration. The bandana facilitated a versatile and practical approach to motor distribution and wire management, accommodating the necessary flexibility without compromising the structural integrity of the device.

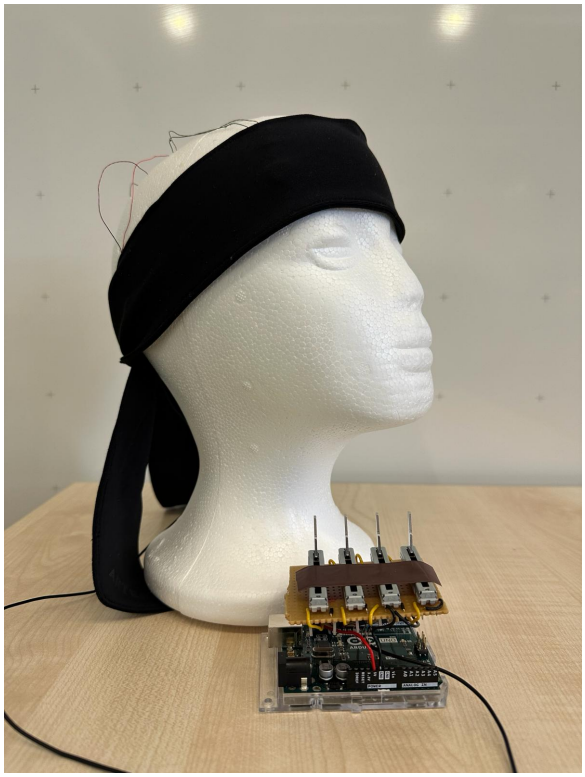


Figure 5: Final prototype of Game5ense, depicting the circuit and the wearable.

4 Limitations and Future Work

In the pursuit of advancing our device's functionality, user friendliness, and interoperability across various gaming environments, it is imperative to acknowledge and address certain limitations, including constraints related to time and budget. These factors, while reflective of the current stage of development, offer a roadmap for future enhancements. The subsequent discussion delineates specific limitations encountered in the device's design, particularly its current game-specific audio thresholding, the discrete nature of vibration feedback, and the challenges posed by wired connectivity, alongside the critical impact of time and budget constraints. Furthermore, we aim to outline strategic directions for the continued development of Game5ense, enhancing its versatility, user experience, and overall effectiveness in providing an immersive gaming experience for individuals with hearing impairments.

The first limitation to be highlighted is the game-specific, manual audio thresholding. The current device is tailored for use in Fortnite, with hardcoded thresholds. However, for Game5ense to work well across different video game genres and titles, it must be able to automatically reconfigure its thresholds for each new game. One possible solution is to implement a calibration period in which the user plays a game, and the audio data is collected and analysed in the background. Based on the distribution of audio intensities uncovered, an algorithm can automatically determine appropriate thresholds for each audio channel. An element of learning can also be implemented so that Game5ense can utilize its knowledge of previously played video games to provide a reasonable starting point for the thresholds in a new game.

Another limitation of our device, related to the previous one, is that the vibration motors are set to vibrate at only three discrete levels (no vibration, medium vibration, and high vibration). Although this was done intentionally for Fortnite, other video games may benefit from a continuous range of vibration intensities. To do this, instead of thresholding the captured surround sound audio into three discrete levels, our audio processing algorithm can be modified so that above a certain baseline range (in which no vibrations occur), the signal is instead scaled down to a range within 0 and 255 expected by the Arduino Uno.

In the contemporary landscape dominated by wireless technology, the reliance of our prototype on wired connections may constitute an additional limitation. Enhancing the device to support wireless communication between the Arduino Uno and the vibration motors represents a pivotal next step, eliminating the constraints imposed by physical wires, thereby potentially augmenting user comfort and the overall gaming experience. This transition to wireless connectivity could be facilitated through the integration of Bluetooth technology, a standard for short-range wireless communication. Transitioning to a wireless configuration necessitates the incorporation of a rechargeable power supply within the wearable component to ensure operational autonomy of the vibration motors.

Lastly, the constraints imposed by limited time and financial resources have been significant in shaping the design and development trajectory of our device. Operating within a

six-week development period and a budget of £100, these constraints influenced critical design decisions and prioritised functionality over other aspects. In the absence of these limitations, the design could be further refined through the development of a bespoke flexible printed circuit board (PCB) integrated with a Bluetooth-enabled microcontroller alongside essential components, facilitating a more streamlined and efficient design. Additionally, with extended resources, greater emphasis could be placed on the device's aesthetics, incorporating the electronics into the wearable through advanced textile engineering techniques. This would not only enhance the durability and attractiveness of the device but also position it as a desirable addition to the user's gaming setup.

5 Conclusion

Game5ense marks a key milestone in accessible gaming, ingeniously transforming in-game audio into haptic feedback to accommodate gamers with hearing impairments. Its design, emphasising interoperability, comfort, and adaptability, ensures broad compatibility with various gaming platforms, and caters to individual user preferences through customisable features. Despite facing time and budget constraints, the development of Game5ense showcases the potential for innovative solutions to foster inclusivity in gaming. As it continues to evolve, Game5ense not only enhances the gaming experience for the hearing-impaired community but also sets a benchmark for future developments in accessible gaming technology, reaffirming the commitment to making gaming a universally enjoyable experience.

References

[1] Patrick Klepek. AbleGamers' Million-Dollar Birthday Fundraiser. https://www.wired.com/story/ablegamers-million-dollar-birthday-fundraiser.

[2] Mike Stubbs. For All The Players: A History Of Accessibility In Video Games. https://www.gamingbible.com/features/games-for-all-the-players-a-history-of-accessibility-in-video-games-20200124.

[3] David Frangioni. 5.1, 7.1.2 - What Do Surround Sound Numbers Mean? https://frangionimedia.com/blog/entry/5-1-7-1-2-what-do-surround-sound-numbers-mean#:~:text=5.1%20is%20the%20most%20common,delivering%20deep%2C%20rumbling%20bass%20frequencies.

[4] Blackhole: Route Audio Between Apps. https://existential.audio/blackhole/.

[5] VB-Audio VoiceMeeter. https://vb-audio.com/Voicemeeter/.

[6] Hubert Pham. PyAudio Documentation. https://people.csail.mit.edu/hubert/pyaudio/docs/.

[7] Minecraft. https://www.minecraft.net/en-us.

[8] Fortnite. https://www.fortnite.com.

[9] Counter-Strike 2. https://www.counter-strike.net/.

[10] Apex Legends. https://www.ea.com/games/apex-legends.

[11] Call of Duty. https://www.callofduty.com/uk/en/.

[12] Arduino Uno R3. https://docs.arduino.cc/hardware/uno-rev3/.

[13] Seeed Studio 316040001 Mini Vibration Motor for Multipurpose. https://uk.rs-online.com/web/p/power-motor-robotics-development-tools/1845122?cm_mmc=UK-PLA-DS3A-_-google-_-CSS_UK_EN_PMAX_Catch%2BAll-_--_-1845122&matchtype=&gclsrc=aw.ds&gad_source=1&gclid=%2A.

[14] Power MOSFET IRF510 Datasheet. https://www.vishay.com/docs/91015/irf510.pdf.

Appendix A: Audio Processing Algorithm

The following shows our audio processing algorithm implemented in Python. The source code can be accessed at our GitHub Repository: hcard-sound-haptic-device.

```
# ======================================================================
#                    Audio Setup - Initial Imports
# ======================================================================

import pyaudio
import numpy as np
import serial
import time
import matplotlib.pyplot as plt

# ======================================================================
#           Real-Time Processing - Audio Data Processing
# ======================================================================

class RealTimeSmooth:

    def __init__(self, window_size):
        self.window_size = window_size
        self.data = np.zeros((window_size, CHANNELS), dtype=np.float32)
        self.index = 0

    def add_data(self, new_data):
        self.data[self.index % self.window_size] = new_data
        self.index += 1
        if self.index < self.window_size:
            # Not enough data yet for a full window
            return np.mean(self.data[:self.index], axis=0)
        else:
            return np.mean(self.data, axis=0)

# CALCULATE AUDIO SOUND AMPLITUDE
def amplitude(samples):
    return np.max(np.abs(samples), axis=0)

# ======================================================================
#          Stream Configuration - Define Audio Parameters
# ======================================================================

# INITIALIZE PYAUDIO
p = pyaudio.PyAudio()

FORMAT = pyaudio.paInt16  # Audio format (16-bit PCM)
CHANNELS = 6  # Number of audio channels for stereo sound
RATE = 44100  # Sample rate (samples per second)
CHUNK = 1024  # Number of frames per buffer

# SERIAL CONNECTION SET-UP
COM_PORT = '/dev/cu.usbmodem144401' # Change based on computer
arduino = serial.Serial(COM_PORT, 500000) # COMMENT OUT W/O ARDUINO
time.sleep(2) # Slight delay for connection
```

```
51
52 stream = p.open(format=FORMAT,
53                 channels=CHANNELS,
54                 rate=RATE,
55                 input=True,
56                 # input_device_index=blackhole_index,
57                 frames_per_buffer=CHUNK)
58
59 # ========================================================================
60 #                  Audio Plotting - Finding the Range
61 # ========================================================================
62
63 def plot_surround_sound(audio_list):
64     channels = np.array(audio_list).T  # Transpose to get channels as
   lists
65     time = np.arange(channels.shape[1]) * CHUNK / RATE
66     plt.figure(figsize=(10, 8))
67     for i, channel_data in enumerate(channels):
68         plt.subplot(CHANNELS, 1, i+1)
69         plt.plot(time, channel_data, label=f'Channel {i+1}')
70         plt.xlabel('Time [s]')
71         plt.ylabel('Amplitude')
72         plt.legend(loc="upper right")
73     plt.tight_layout()
74     plt.show()
75
76 # ========================================================================
77 #              Audio Stream - Capture and Process Audio
78 # ========================================================================
79
80 print("Starting audio stream...")
81 smooth = RealTimeSmooth(window_size=int(RATE/CHUNK))
82 testing = []
83
84 try:
85     while True:
86         data = stream.read(CHUNK, exception_on_overflow=False)
87         audio_data = np.frombuffer(data, dtype=np.int16).reshape(-1,
   CHANNELS)
88         # Defining and converting intensity into 0, 75, 100
89         intensity = amplitude(audio_data)
90         intensity = smooth.add_data(intensity)
91         for i, value in enumerate(intensity):
92             if i == 0 or i == 1 or i == 2:
93                 intensity[i] = 0 if value < 2000 else 75 if value <
   8000 else 100 # Front two audio channels
94             else:
95                 intensity[i] = 0 if value < 1000 else 75 if value <
   2000 else 100 # Back two audio channels
96         cmdArrayFloat = np.array(intensity, dtype=np.uint8) # Array of
   2 uint8
97         cmd_bytes = cmdArrayFloat.tobytes() # Array of 16 bytes
98         testing.append(intensity)  # Convert np.array to list for
   easier handling later
```

```
99          n = arduino.write(cmd_bytes) # Send to Arduino # COMMENT OUT W/
    O ARDUINO
100 except KeyboardInterrupt:
101     print("Stopping audio stream...")
102     stream.stop_stream()
103     stream.close()
104     p.terminate()
105     plot_surround_sound(testing)  # Plotting after stopping the stream
    with a keyboard interrupt
106     arduino.close()
```

Appendix B: Arduino Code

```
#include <string.h>
int BackRight = 11;
int FrontRight = 3;
int FrontLeft = 9;
int BackLeft = 5;
uint8_t arr[6];
uint8_t prev_arr[6] = {0};

void setup() {
  // Start serial communication at 500000 bps.
  Serial.begin(500000);
  pinMode(BackRight, OUTPUT);
  pinMode(FrontRight, OUTPUT);
  pinMode(FrontLeft, OUTPUT);
  pinMode(BackLeft, OUTPUT);
}

void loop() {
  if (Serial.available() >= 6) {
    // Read serial array
    Serial.readBytes((char*)arr, 6);

    // Compare new array with the previous one
    if (memcmp(prev_arr, arr, 6) != 0) { // If different
      // Control output
      analogWrite(FrontLeft, arr[0]);
      delay(5);
      analogWrite(FrontRight, arr[1]);
      delay(5);
      analogWrite(BackLeft, arr[4]);
      delay(5);
      analogWrite(BackRight, arr[5]);

      // Copy new array to prev_arr for the next loop iteration
      memcpy(prev_arr, arr, 6);
    }
  }
}
```

Appendix C: Additional Audio Data Plots

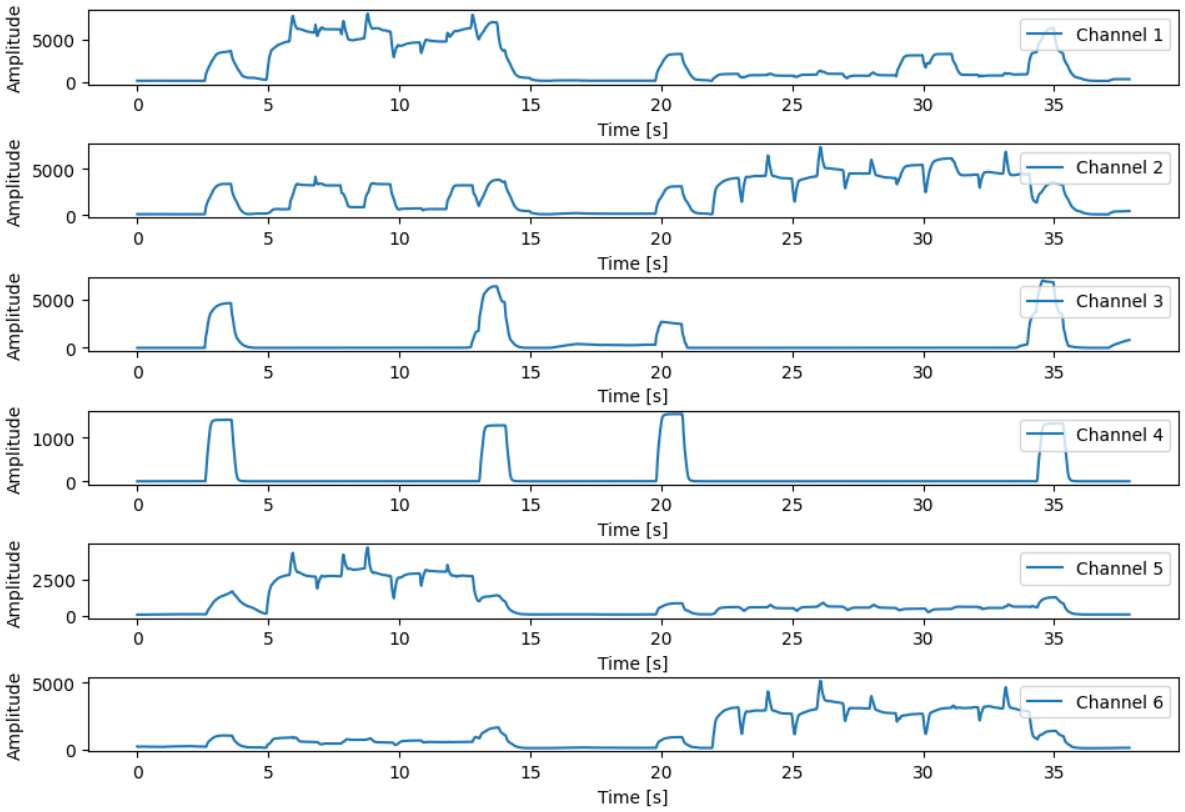


Figure 6: Audio intensities of loud in-game sounds (player getting shot from the left and right) across each audio channel in 5.1 surround sound in Fortnite. Channel 1 is the front-left audio channel, Channel 2 is the front-right, Channel 3 is the center, Channel 4 is the subwoofer, Channel 5 is the back-left, and Channel 6 the is back-right.

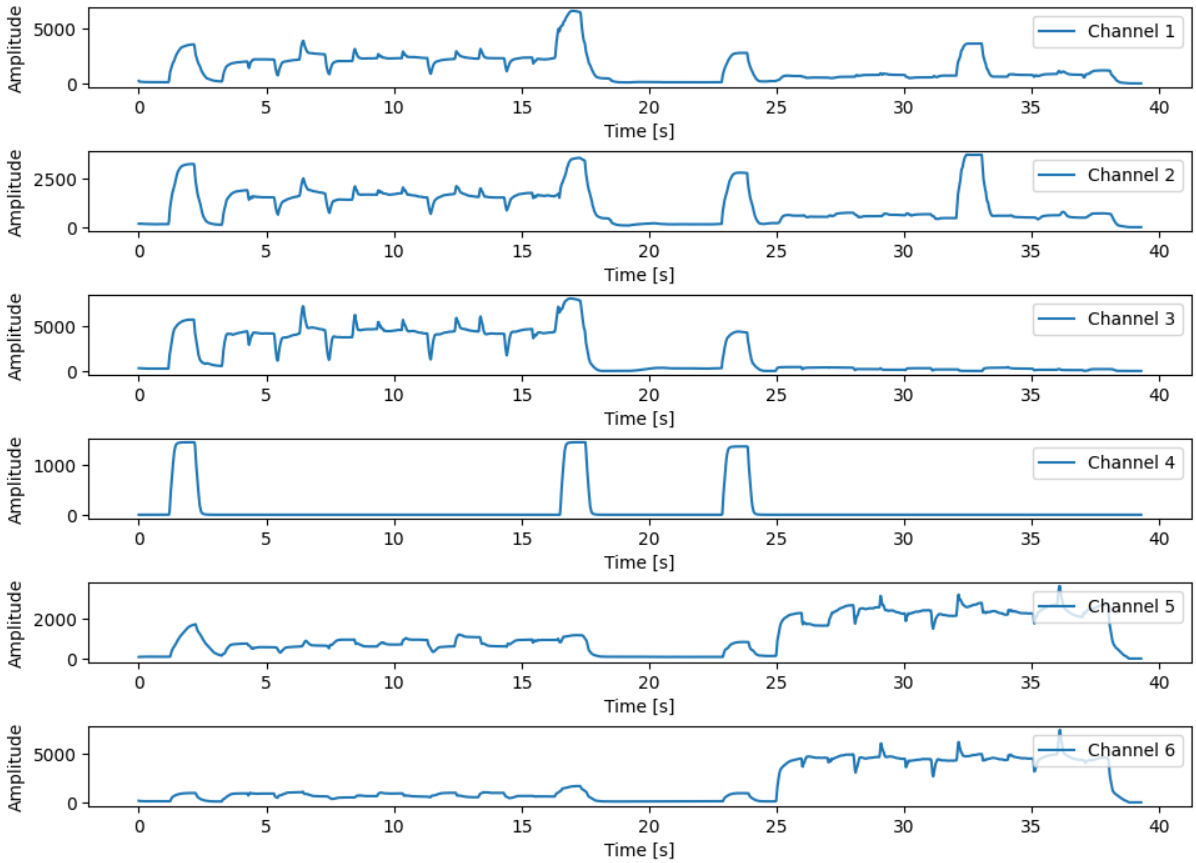


Figure 7: Audio intensities of loud in-game sounds (player getting shot from the front and back) across each audio channel in 5.1 surround sound in Fortnite. Channel 1 is the front-left audio channel, Channel 2 is the front-right, Channel 3 is the center, Channel 4 is the subwoofer, Channel 5 is the back-left, and Channel 6 the is back-right.

Appendix D: Marketing Visuals

Figure 8: Game5ense marketing poster.

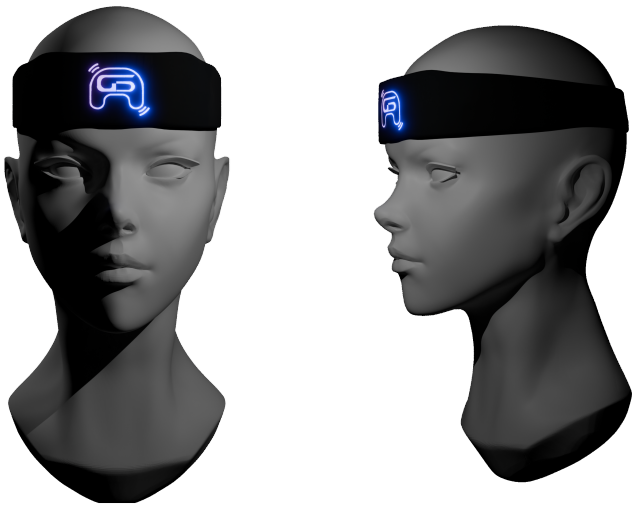
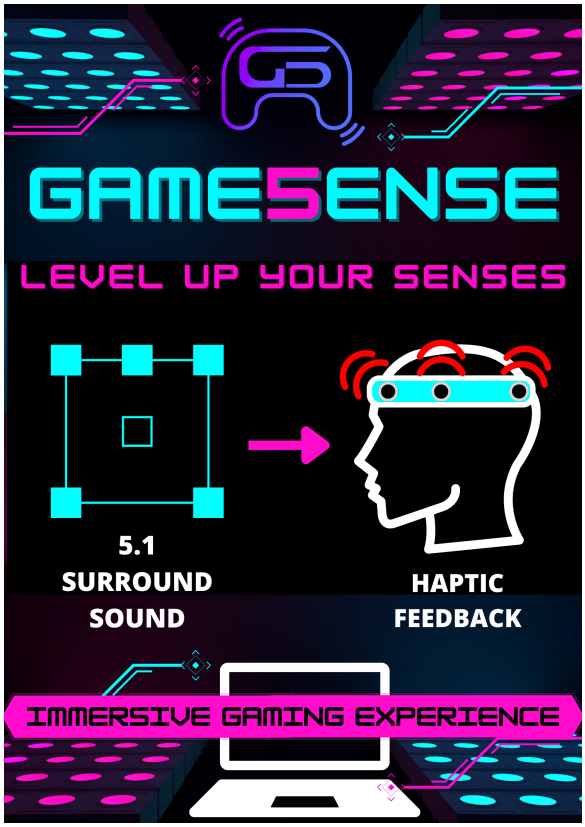
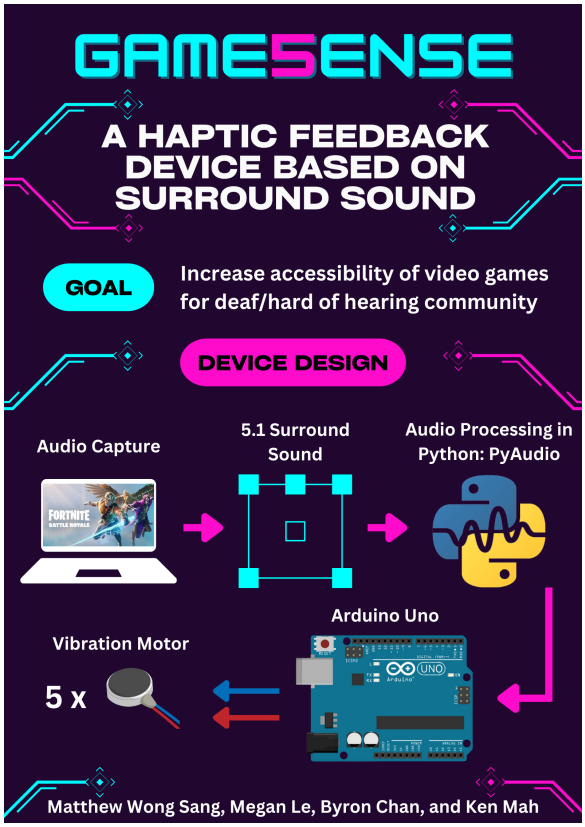


Figure 9: Visual demo of device usage.



(a) Marketing poster.



(b) Infographic poster.

Figure 10: Visual marketing posters of our device, Game5ense.