

Sonic Interaction Design for Websites

A Framework Designed for Simplifying
User Interaction Sounds

Master Thesis
2022

Johannes E Lechner, BSc

Contents

1. Introduction	I
2. Sound and Interface Design	2
2.1. Sound and Interface Design in Other Media and Systems	5
2.1.1. In Games	5
2.1.1.1. Counter-Strike: Global Offensive	6
2.1.1.2. Grand Theft Auto V	8
2.1.2. In Mobile Operating Systems	9
2.1.3. In Desktop Operating Systems	11
3. The Framework	17
3.1. The Need for a Framework	17
3.2. Development	18
3.2.1. The Backend	18
3.2.1.1. Different Types of Buttons	18
3.2.1.2. Asynchronously Loading Audio Files	20
3.2.1.3. Playing Audio	23
3.2.1.4. Audio Playing Issues	29
3.2.1.5. Mixer and Mute	31
3.2.1.6. Saving Settings	33
3.2.1.7. Volume Groups	36
3.2.1.8. Settings Groups	39

3.2.1.9. Audio Context Startup and Settings for Site Owners	41
3.2.2. Enhancements Through TypeScript	43
3.2.3. The Front-End	46
3.2.3.1. Usage with CSS Frameworks	46
3.2.3.2. Mixer Styling	49
4. Conclusion	51
5. Table of Figures	52
6. Bibliography	53

Glossary

CMS	Content Management System a system to manage creation and modification of website contents like pages or posts
TTFB	Time To First Byte the time it takes for the first byte of a website to arrive after clicking a link
UWP	Universal Windows Platform a computing platform created by Microsoft to develop apps on Windows 10, 11, Xbox One and other Microsoft OS and devices
Win32	Windows API UWP's predecessor for developing Windows-only programs

Abstract

Many digital user interfaces are supported by sound. Especially computer games, operating systems like Android, and professional control panels benefit from this. Websites, however, are largely unaffected by this.

This thesis is about designing and implementing a web framework to coordinate user interface sounds and make them manageable. While still customizable, the main goal was to provide an easy-to-work-with framework with predefined sounds and plug-and-play capabilities.

The theoretical part of the thesis builds a foundation for the decisions made during development and shows how different media deal with sonic interaction design.

1. Introduction

As a regular mass transit user, I have often noticed the widespread landscape of people's notification sounds. Computer games, regardless of genre, also tend to have a lot of sonic feedback, whether in menus or directly in the game. Finally, operating systems on computers also feature sonic feedback; in all three cases, it seems to enhance the experience for the user. On websites, this is, however, primarily non-existent. This thesis aims to open up the possibility of adding interaction sounds to any website by creating a framework that developers can easily add to any existing web project. There are two central questions that this thesis will answer:

Can websites also use sonic interaction design?

Can a user- and developer-friendly framework be created to accomplish this?

The thesis will answer the two questions above by being split into three parts: The first chapter will reference other literature to explain the importance of sound in interface design. Secondly, other media and operating systems will be analyzed and compared to figure out how to approach the framework. Thirdly, I will explain the framework itself with subsections focusing on the individual parts that make the framework function. Finally, in the conclusion, I will summarise the most significant issues again and explain what should be done differently by following projects.

2. Sound and Interface Design

Auditory feedback plays a vital role in our daily lives. Feedback like shutting a door or the sizzling sound of food in a pan indicates valuable information without taking a deeper look at what is happening. This feedback already plays a significant role in product design, where companies consider many factors to get a distinct sound design when interacting with an object [1]. Rocchesso et al. [1] say this is especially true in areas that: “... create products with high functional densities, strong design identities, or which address demanding markets such as car design.” They furthermore state that “prominent industries that have benefited from it [sound design] include the automobile and cosmetics industries, but lower profile applications have arisen in diverse other areas, from kitchen appliances to toys and office equipment. “However, in the digital world, there is no natural auditory (and haptic) feedback and no properties such as material type and force that would create sound on interaction. Instead, digital interfaces can have any desirable audio feedback, including none at all. Though the last option is not advisable, as Stephen Brewster and Murray Crease [2] found out through an experiment:

“The results have shown that by indicating menu and item slip errors in a salient way the usability of menus can be significantly improved. Sound was shown to be a very effective method of providing this feedback. The workload analysis showed that the overall workload, and in particular effort expended, was reduced significantly when using the sonically-enhanced menus. Workload was reduced because the sonic enhancements meant that participants needed to expend less effort to notice and recover from

ments meant that participants needed to expend less effort to notice and recover from menu and item slips. This, however, was not at the expense of making the menus more annoying to use. This, along with previous results from sonifying other graphical widgets (Brewster et al. 1994, Brewster et al. 1995a, Brewster 1998), indicates that earcons can provide a significant qualitative improvement in a user's experience with a system."

Earcons, as mentioned above by Brewster and Crease, are sound icons that either request a reaction or create auditory feedback on interaction. According to Hereford and Winn [3], these "can tell you immediately if you have made an error, such as a wrong key-stroke, or can tell you the status of the system, such as the rich tone that is made when a disk is put into the drive on a Macintosh computer." When writing the article in 1994, Hereford and Winn [3] stated that sound in interfaces was "limited almost exclusively to [...] arbitrary tones and beeps" which, to a neophyte, are not incredibly informative. This is still the case today as, e.g., smartphone notification sounds show. The browser Opera GX for example, as seen in "Figure 2.1: Screenshot of Opera GX Sound Options", has numerous options to enable clicks and typing sounds in the browser's interface. And while real-world objects such as a typewriter inspired some sounds, the tab closing sound, for example, seems to have no real-world pendant.

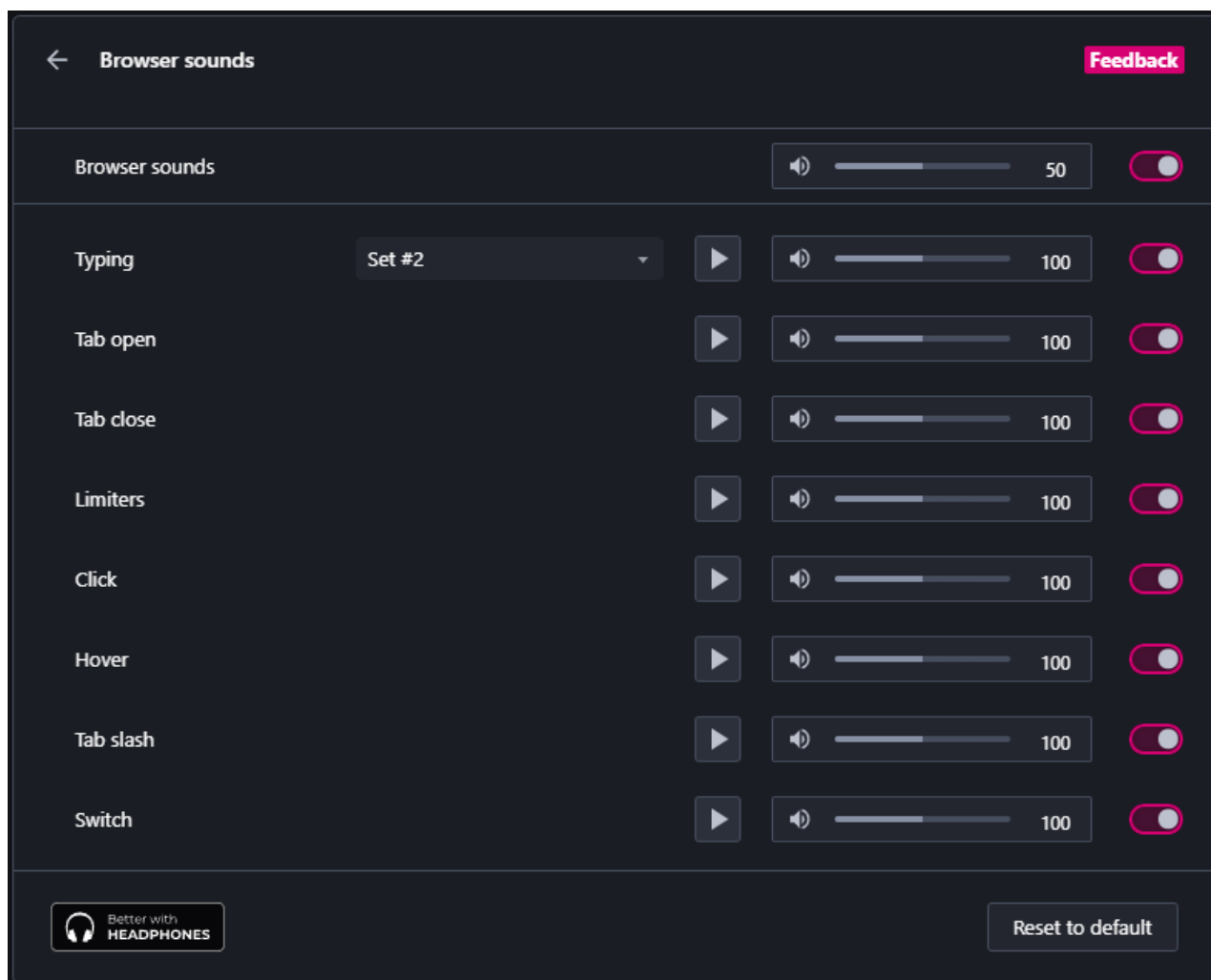


Figure 2.1: Screenshot of Opera GX Sound Options

2.1. Sound and Interface Design in Other Media and Systems

Before developing a web framework to quickly implement interaction sounds, a reference of how to design the framework and its interaction, must be established. For this, other media and systems are analyzed in the following sections. It is assumed that the same principles that apply to similar systems, such as operating systems, also apply to websites, as usability is handled similarly.

2.1.1. Games

The following games have been selected based on three factors: First, a leak from Steam in 2018 where the top played games could be estimated [4]. The data has been compared to current statistics using SteamDB [5] as seen in “Figure 2.2: Screenshot of SteamDB most played games as of 18. August 2022”. Although very similar to other days, the screenshot below explicitly shows stats from 18. August 2022. Lastly, my preferences and ownership of the listed games have influenced the selection.



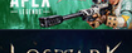



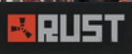










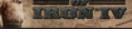
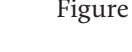

1.		Counter-Strike: Global Offensive	743,547	987,884	1,308,963	+
2.		Dota 2	582,032	653,605	1,295,114	+
3.		Apex Legends	221,840	440,668	511,676	+
4.		Lost Ark	221,120	221,120	1,325,305	+
5.		PUBG: BATTLEGROUNDS	177,349	370,162	3,257,248	+
6.		Grand Theft Auto V	123,192	159,213	364,548	+
7.		Team Fortress 2	104,668	104,668	151,253	+
8.		Rust	100,416	101,003	245,243	+
9.		FIFA 22	78,842	79,669	108,295	+
10.		Football Manager 2022	62,363	67,824	88,767	+
11.		ARK: Survival Evolved	60,357	69,935	248,405	+
12.		Wallpaper Engine	54,073	81,148	106,210	+
13.		War Thunder	52,191	57,875	76,204	+
14.		Destiny 2	49,537	60,455	292,513	+
15.		MONSTER HUNTER RISE	41,543	88,574	231,360	+
16.		Sid Meier's Civilization VI	40,841	50,116	162,657	+
17.		MIR4	38,968	40,700	97,173	+
18.		Dead by Daylight	38,326	44,402	105,093	+
19.		Warframe	37,484	49,080	189,837	+
20.		Hearts of Iron IV	36,178	38,713	68,019	+

Figure 2.2: Screenshot of SteamDB most played games as of 18. August 2022

2.1.1.1. Counter-Strike: Global Offensive

In August 2018, Counter-Strike: Global Offensive (CS:GO) received a new User Interface [6]. The so-called “Panorama UI” modernized CS:GO’s interface and introduced a new sound design. Valve introduced a hierarchy of interaction sounds. These range from very soft typewriter-like sounds when hovering over settings menu items to more pregnant versions when hovering over more significant menu items, to deep, airy explosion mimicking sounds when opening and closing main windows. In addition to

that, all sounds feature a very soft but noticeable reverb effect. For users on headphones, audio is locational, meaning menus on the left also generate sound from the left. The last effect, however, requires a setting to be enabled in the audio options seen below.

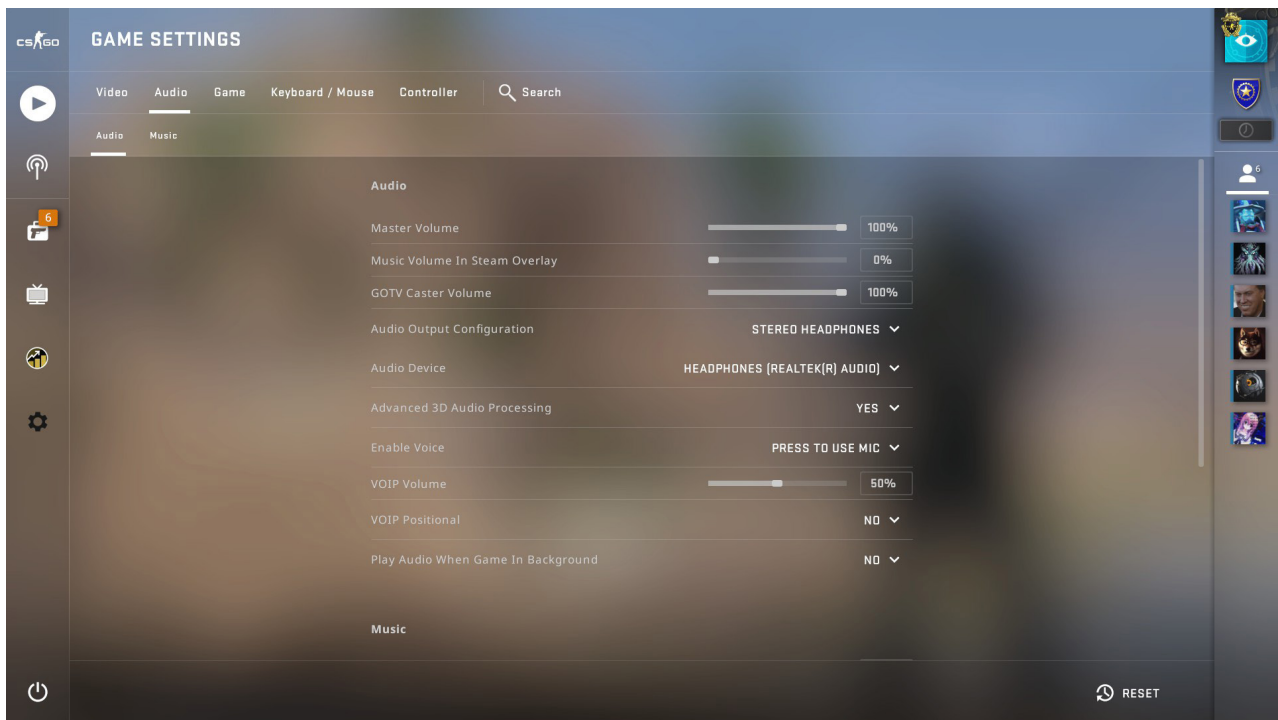


Figure 2.3: Screenshot of Counter-Strike: Global Offensive Sound Settings

Surprisingly CS:GO only offers three non-music volume sliders, one of which is a master audio slider. Most settings are related to positional audio and voice chat, which for a game like CS:GO are the most likely the most critical settings. But other sound settings lack entirely. For example, volumes for sounds in matches like footsteps and gunshots are not adjustable, as changing these settings can arguably lead to considerable advantages in competitive games. However, interaction sounds in the menu are also not editable, which would not modify gameplay [7].

2.1.1.2. Grand Theft Auto V

Grand Theft Auto V (GTAV) has a very similar modern user interface. The menu, as seen in , is split into tabs in the top row and submenus on the left. While the interface appears normal through visual inspection, the sonic feedback shows that GTAV was built for consoles and not PCs. There are no hover sounds, but clicks or navigating with controllers or a keyboard's arrow keys do generate sound. There is, however, no hierarchy detectable between menus and options. The exception is the category row on top, which produces the same sound when clicking on the tabs. The content, however, has to load every time. After finishing loading a few milliseconds later, it produces another, different sound. Like CS:GO, GTAV does not have any settings for interaction sounds and does not feature much customizability for other sound options. There are three volume sliders, a few options for multichannel audio setup, and settings for music. Both games show the volume sliders as horizontal bars rather than vertical ones. This is most likely due to space constraints but seems unintuitive as other digital applications such as DAWs and hardware all feature up-down sliders, which users can intuitively map to increasing or decreasing volume [8].



Figure 2.4: Screenshot of Grand Theft Auto V Sound Settings

2.1.2. In Mobile Operating Systems

Android 11, specifically EMUI 11, has a lot of customizability built-in. First, a quick toggle for different sound modes can be seen. This toggle changes all sound settings like a master channel's mute button. As seen in “Figure 2.5: Screenshot of EMUI 11 “Sound & vibration” Settings”, Android also has four different volume sliders for different types of audio: “Ringer, Notifications”, “Alarms”, “Music, Videos, Games”, and “Calls”. Button presses like touch sounds fall under the first category. The sub-menu “More settings” reveals on-off switches for these sounds [9]. Different manufacturers change the layout of these settings. Samsung, for example, does not have a “More settings” menu and uses multiple sub-menus [10]. The functions, however, are generally very similar among Android flavors.

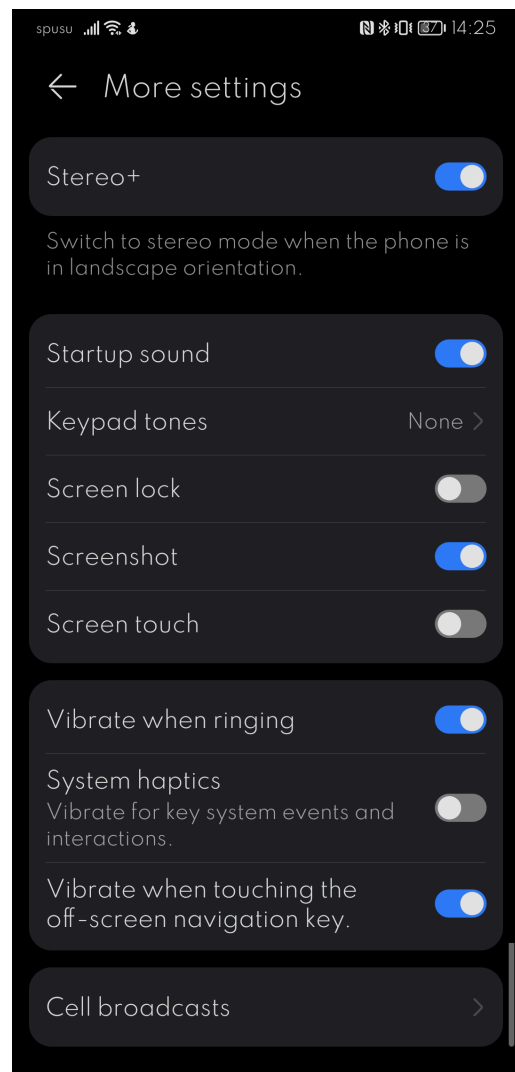
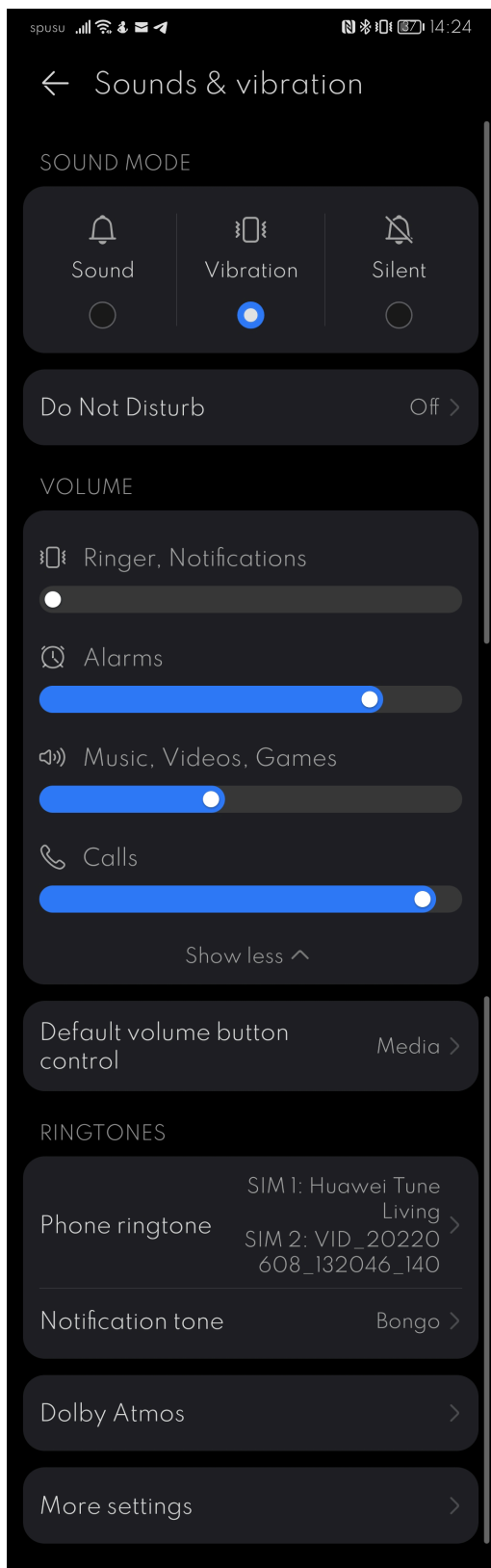


Figure 2.6: Screenshot of EMUI 11 “More settings” Sound Settings

Figure 2.5: Screenshot of EMUI 11 “Sound & vibration” Settings

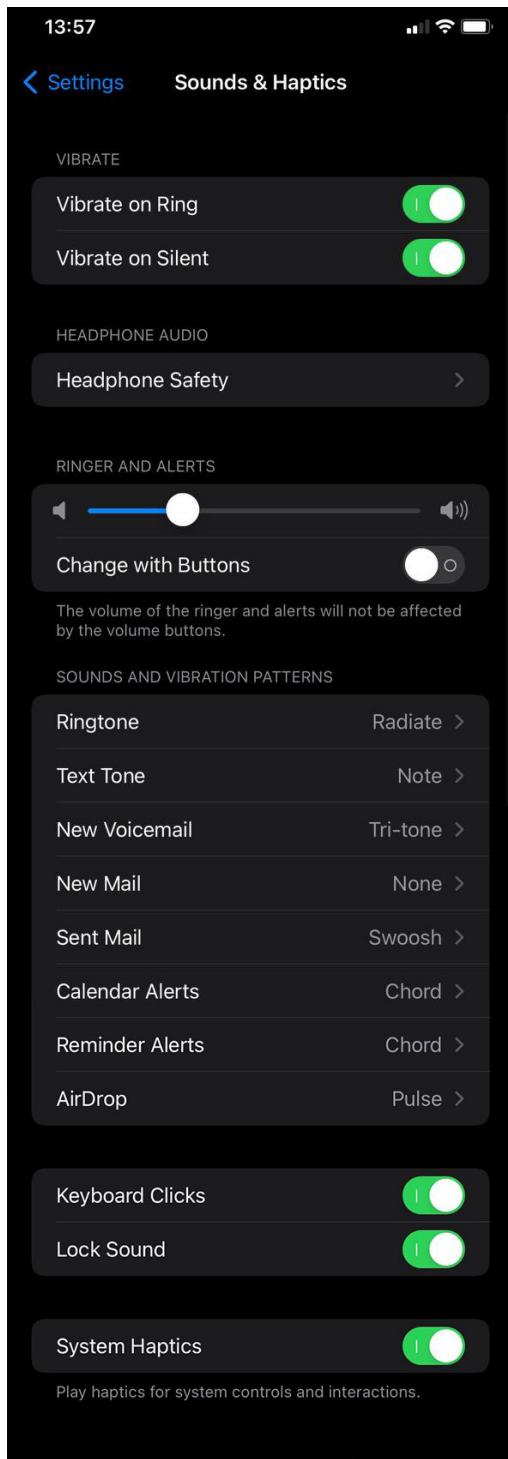


Figure 2.7: Screenshot of iOS 15 “Sound & Haptics” Settings

Apple’s iOS 15 offers much less customization than Android. For example, only one volume slider controls the ringer and alerts. Apple, however, has integrated options to change sounds for many interactions, like sent mail tones. In Android, while having a settable default notification sound in the system settings, this is generally handled on a per-app basis. Keyboard clicks, for example, are set in the installed keyboard, but Apple has an on-off switch for this in its settings.

2.1.3. In Desktop Operating Systems

Apple’s macOS 12 shows a very minimalistic approach to sound settings (see “Figure 2.8: Screenshot of macOS 12 Sound Settings”). For example, several presets that affect notification tones are available for the user to choose, but only two volume sliders: one for master volume, which affects all applications, and one for notification sounds. Apple also included three switches

for interaction sounds. Compared to iOS, there are more switches but overall fewer options [11].

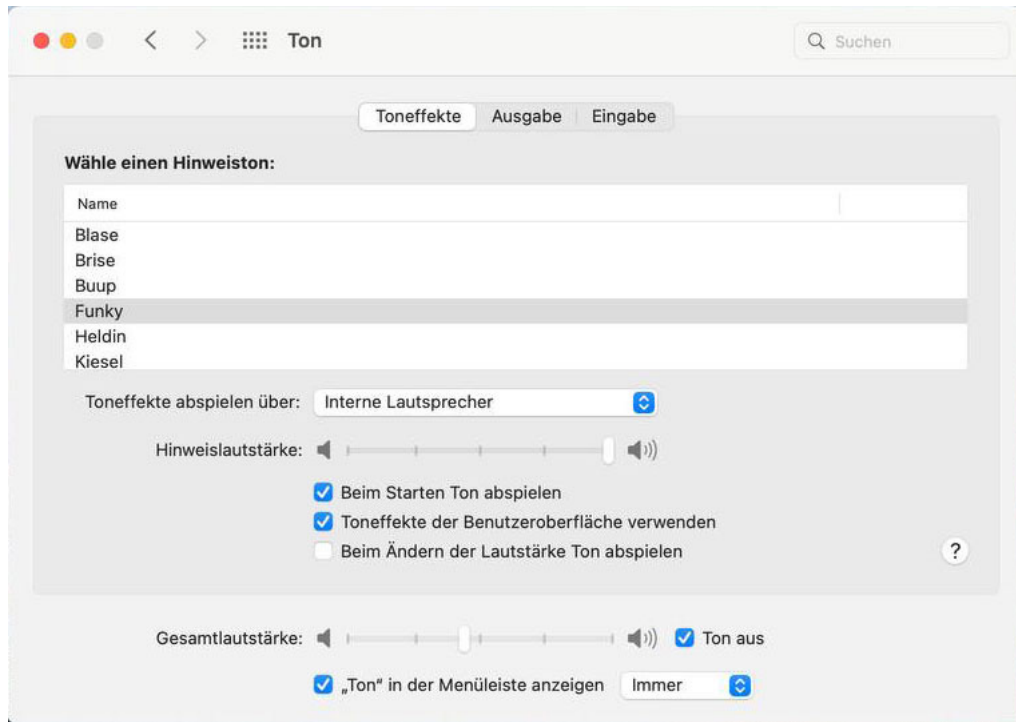


Figure 2.8: Screenshot of macOS 12 Sound Settings

Windows 11's sound settings appear limited at first sight as well. There is only a volume slider and the option to combine the left and right channels. Windows, however, has a few sub-menus presenting the real strength of its sound settings. As seen in 1, the volume mixer allows users to tamper with different apps' volume settings from a centralized location and set sound in- and outputs per app. Before Windows 11, users would open the volume mixer from the tray, but this feature has been moved into the new UWP Settings app [12]. The "More sound settings" button opens up an older variant of sound settings from before Microsoft started upgrading their apps to UWP. Users can still change individual sounds in this window, as seen in 1 [13].

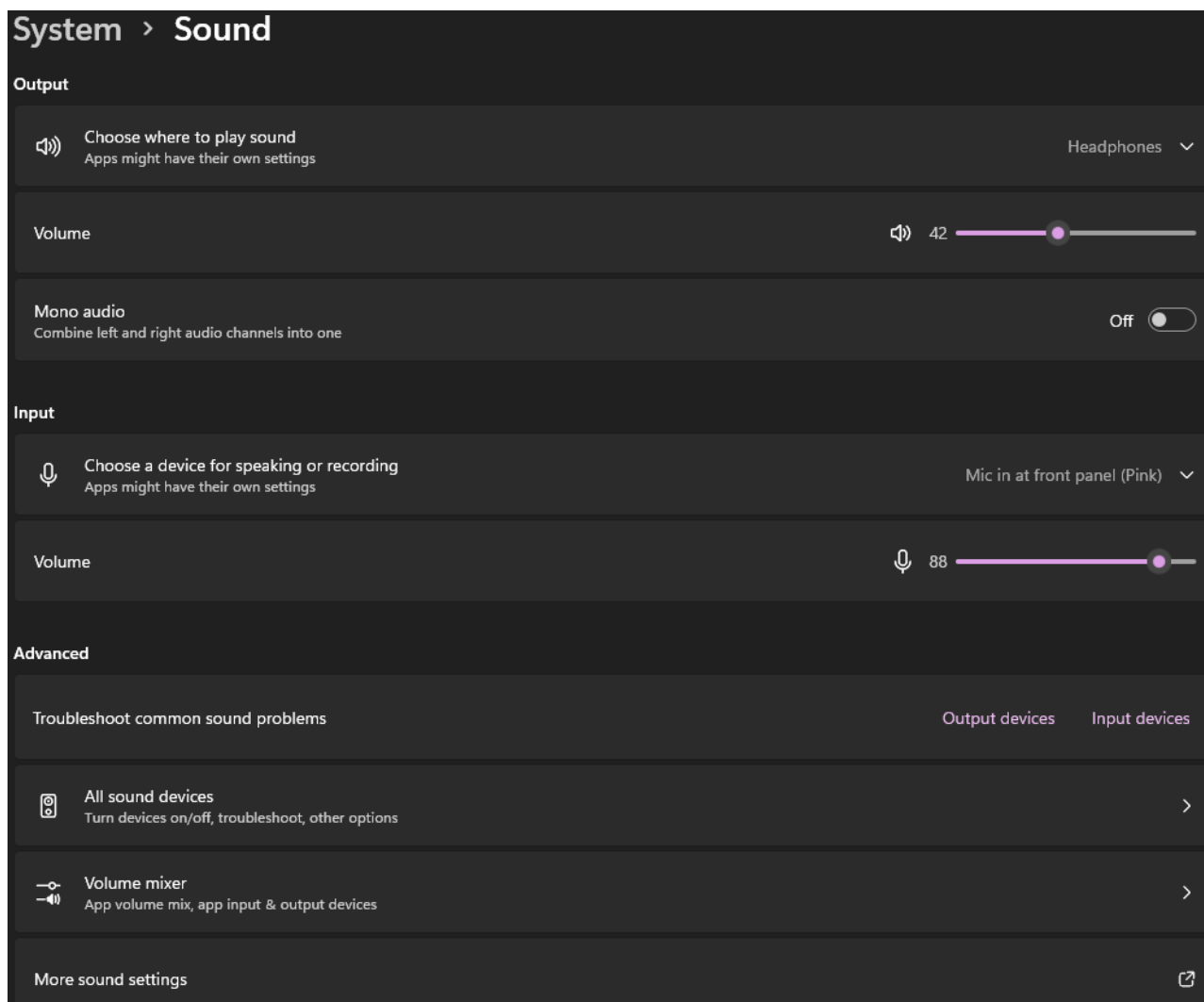


Figure 2.9: Screenshot of Windows 11 Sound Settings

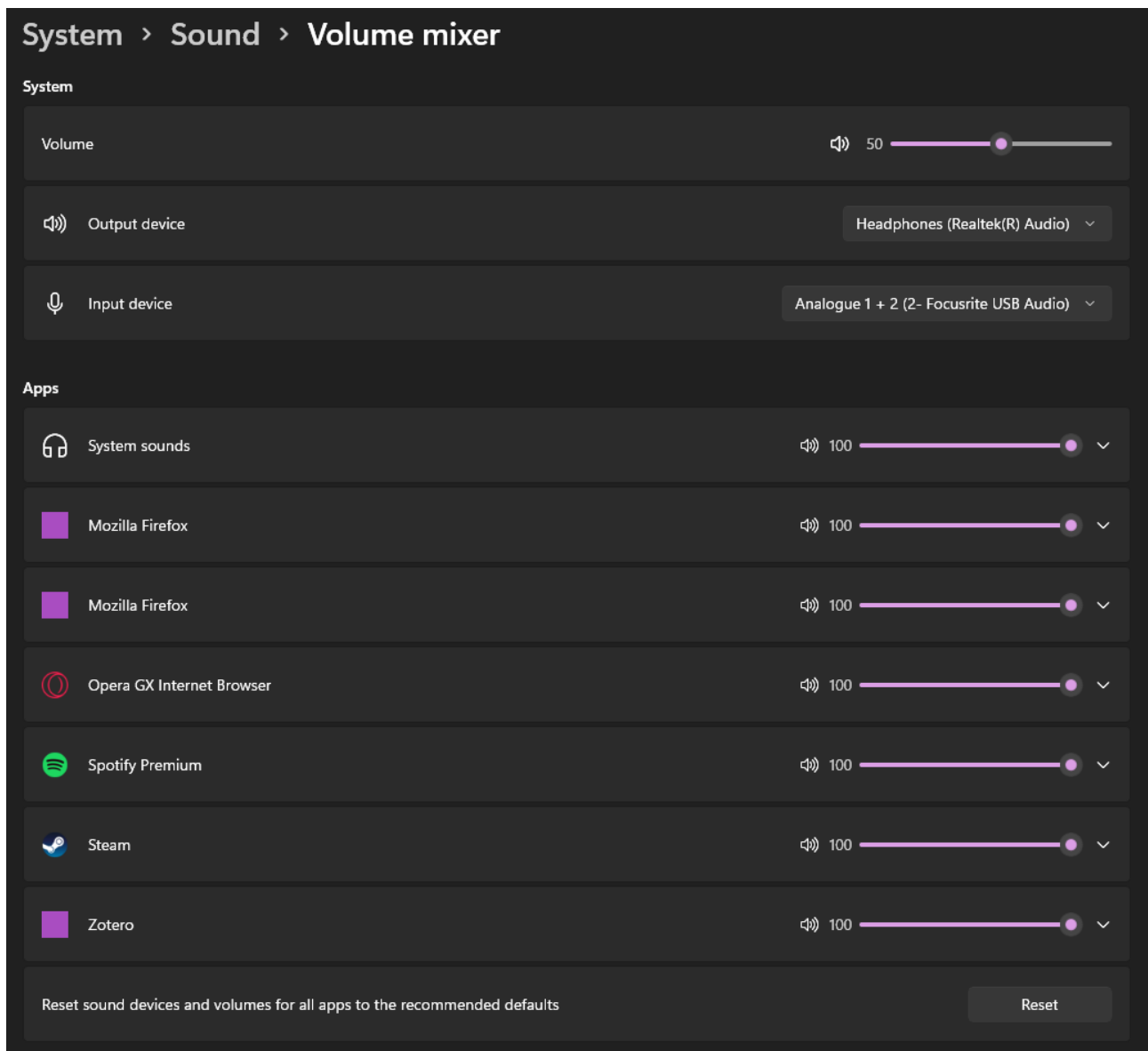


Figure 2.10: Screenshot of Windows 11 Sound Mixer

t

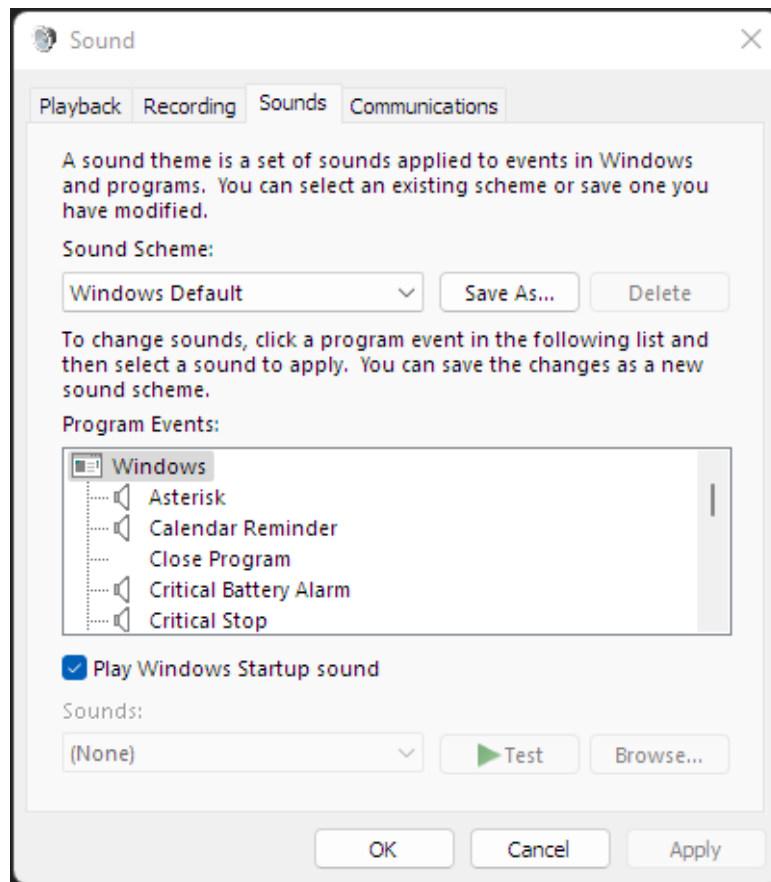


Figure 2.11: Screenshot of Windows 11 Win32 Sound Options

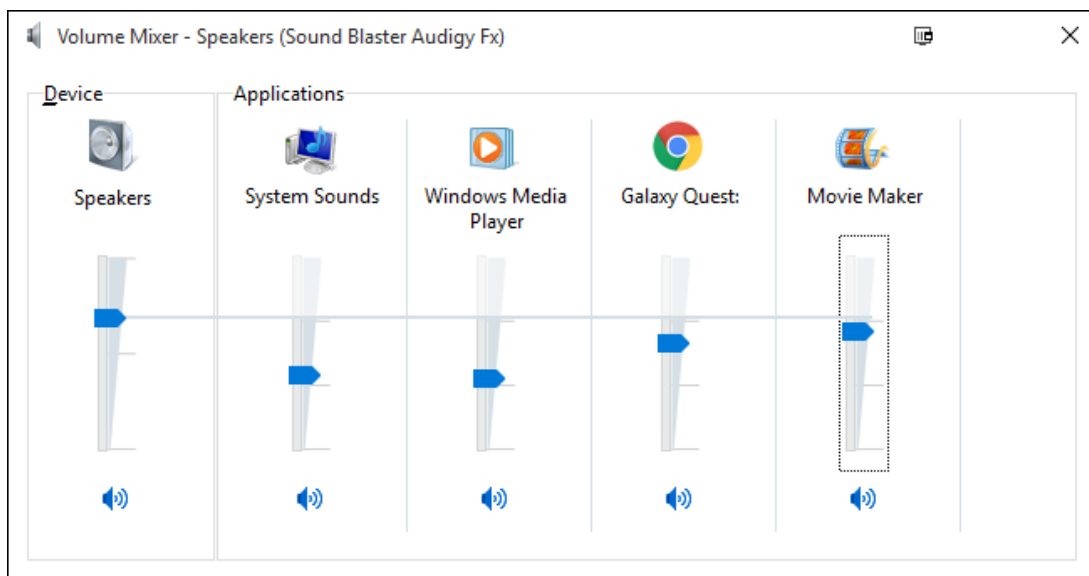


Figure 2.12: Screenshot of Windows 11 Win32 Sound Options [13]

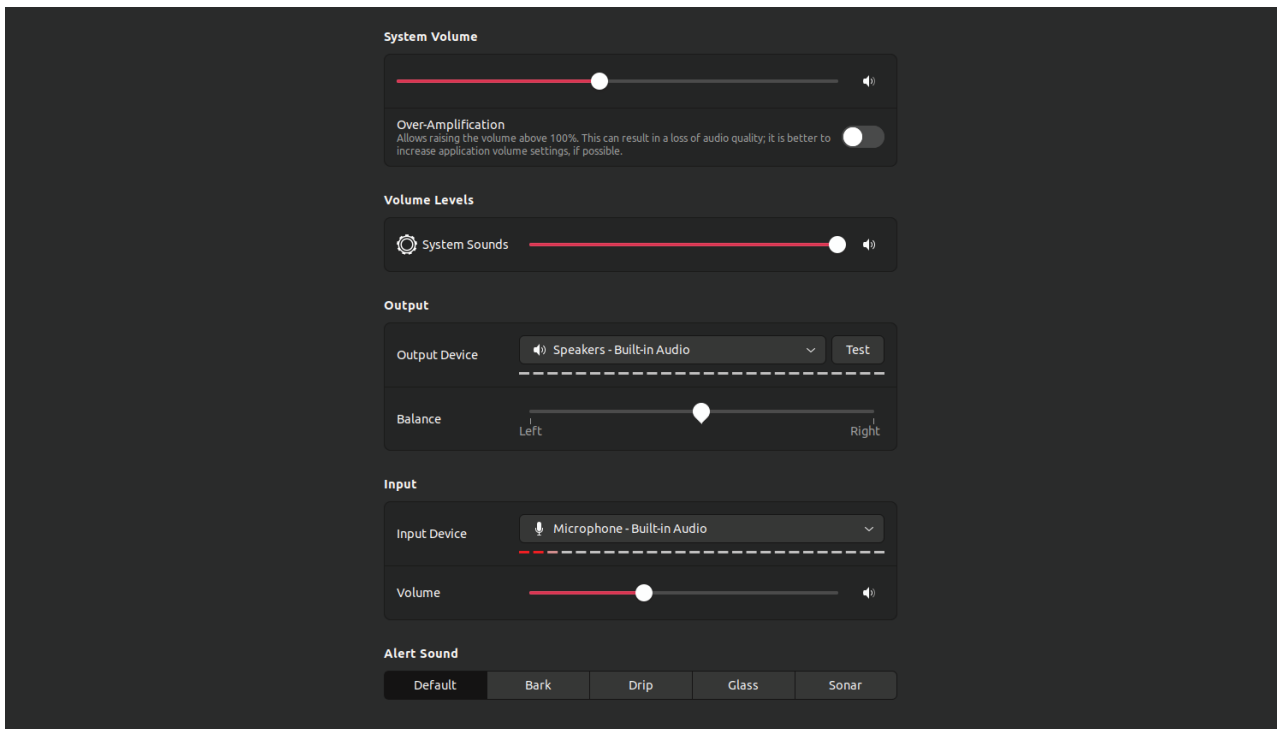


Figure 2.13: Screenshot of Ubuntu 22 LTS Sound Settings

Ubuntu 22 LTS has very similar options to macOS. There is a system volume slider and a system sounds volume slider; the latter controls various sounds, like plugging and unplugging USB devices. The only other option focusing on interaction sounds is the alert sound selector, which lets the user choose between five different types of sounds [14]. This settings menu resembles macOS' settings very much. It has to be stated, though, that this is the default ubuntu configuration with GNOME as a desktop environment. GNOME has many resemblances to macOS and works great without much customization. However, there are other options like KDE's Plasma Desktop Environment, which has a more Windows-inspired approach, giving the user many customization options [15].

3. The Framework

3.1. The Need for a Framework

For the entire history of the web, sound was usually not a part of it. But with the introduction of HTML 5 and the audio tag in 2008, users could load audio files into websites, and browsers would render a simple control UI [16]. The Web Audio API expanded upon this concept and allowed users to build various web applications with similar features to Digital Audio Workstations. The problem, however, is that smaller developers and smaller web projects usually do not have the time and resources to add audio enhancements to websites. Richard MacManus [17] states: ‘At their best, frameworks make it easy for developers to create sophisticated web apps.’ Thus, a small framework for basic Sonic Interaction Design could help many developers quickly add a new component to their websites.

3.2. Development

3.2.1. The Backend

Before starting with anything else, the Web Audio API requires an `AudioContext` to be created. The `AudioContext` is the central processing unit of everything the Web Audio API offers. It is responsible for controlling and generating nodes and decoding and playing audio. A single `AudioContext` is sufficient for a website to handle multiple streams at once [18].

3.2.1.1. Different Types of Buttons

Hyperlinks are what defined the internet from the very first time it was used. These links are usually located inside anchor tags (`<a>`) and either styled as underlined text changing color on hovering over them (default browser behavior) or styled like buttons. The two other commonly used options for interaction are the input tag and the button tag. In CSS frameworks, these three types usually fall under one category. Generally, they are styled as buttons regardless of whether the element is a button tag or an anchor tag. W3.CSS 4, for example, defines `w3-btn` as well as `w3-button` classes, which are generally used to style any link or button. These can be further styled with a handful of color classes such as `w3-blue` and other effects such as border and shadow classes [19]. Bootstrap 5 has a very similar `btn` class, but instead of dedicated color classes, button classes serving specific semantic purposes are available. With the default settings, coloring a button yellow would require the `btn-warning` class, for example [20]. Nevertheless, the semantic differences of these buttons can not only be used to style them visually but also to have different kinds of audio feedback.

In this project, the developer has the option to define classes that correspond to different audio cues. However, to be as open as possible to any CSS framework or none at all, the default behavior is not to apply any sounds to any elements. Therefore, the developer has to assign so-called `bsound` classes manually. With the default settings, these classes are `bsoundDefault`, `bsoundImportant`, `bsoundHover`, and `bsoundHold`. The name implies the use case, but for clarification, here are the intended uses for each type:

- **`bsoundDefault`:**
A short impulse for any actions that require generic sound feedback on interacting; the default trigger is a click.
- **`bsoundImportant`:**
A short impulse for actions with higher priority than the generic sounds; the default trigger is a click.
- **`bsoundHover`:**
A short sound that fades in and out; the default trigger is hovering over objects.
- **`bsoundHold`:**
A brief and continuously repeated sound; the default trigger is pressing and holding down on an element (intended for sliders). For this to work correctly, the sound has to be as short as possible and seamlessly loopable. That implies that the sound wave has to be cut at zero-crossings and only in whole-numbered multiples of one cycle. Additionally, it is recommended to use WAV files as MP3 files often contain silence, thus making the sample not seamlessly loopable anymore [21].

These settings only apply to the default behavior. The framework can be used outside the box or customized to the developer's preferences. Developers could also add more types of sound by copying existing code and editing a few lines. Below is an example of the default sound types applied to a Bootstrap 5 website.

```
var defaultSoundSelector = ".btn-close, .btn-primary, .btn-secondary,  
    .nav-item";  
var importantSoundSelector = ".btn-warning, .btn-success,  
    .btn-danger";  
var hoverSoundSelector = ".bsoundHover";  
var holdSoundSelector = "input[type='range']";
```

3.2.1.2. Asynchronously Loading Audio Files

Developers can acquire audio in the Web Audio API in several ways. The MDN lists the following four methods:

- generating it directly in JavaScript by an audio node
- reading it from raw PCM data (e.g., WAV files)
- generating it from HTML media elements exposed in the DOM
- getting a live feed from a WebRTC MediaStream (e.g., webcams, microphones)

[22]

As this project works with audio files, only two options are viable. Usually, for accessibility, longer audio files are exposed in the DOM and loaded into the AudioContext via `audioContext.createMediaElementSource()`. Shorter audio files like button clicks get loaded into buffers via `audioContext.createBufferSource()`. Therefore music would be loaded via the first option, while sounds used in this framework rely on

the second option. Loading and playing a sample requires three functions and a list of sounds present on the server to be used in the framework. Developers can specify these sounds simply as an array like in the following:

```
const audioUrls = [  
  "audio/click.mp3",  
  "audio/click2.mp3",  
  "audio/woosh.mp3",  
  "audio/loop.wav"  
];
```

To avoid modifications to the code, developers must list sounds in a specific order. The default order is `bsoundDefault`, `bsoundImportant`, `bsoundHover`, and `bsoundHold`. Developers can change the order by switching the index in the play function shown below and changing the order in the `audioUrls` constant. The play function requires two other asynchronous functions to be functional. The `audioFilesToBuffers(urls)` function loops through the array of audio files and produces an array of `AudioBuffers` for use by the Web Audio API. This function is called once on load and is visible below.

```

async function audioFilesToBuffers(urls) {
  const audioBuffers = [] as AudioBuffer[];
  for (let url of urls) {
    const sample = await singleAudioFileToBuffer(url)
      as AudioBuffer;
    audioBuffers.push(sample);
  }
  return audioBuffers;
}

```

As seen above, the `singleAudioFileToBuffer(url)` function is called inside `audioFilesToBuffers(urls)` and uses the 2015 introduced fetch API to get an audio file from the specified URL. This API requires using the `async` and `await` keywords, suspending operation until `fetch()` delivers a response. The response is then converted into an `ArrayBuffer`; basically, an array of bytes similar to a WAV files PCM stream of data. As the last step, the array buffer is converted to a Web Audio API compatible `AudioBuffer` and returned.

```

async function singleAudioFileToBuffer(url) {
  const response = await fetch(url);
  const arrayBuffer = await response.arrayBuffer();
  const audioBuffer = await audioContext.decodeAudioData(
    arrayBuffer
  );
  return audioBuffer;
}

```

When the browser calls the function `audioFilesToBuffers(urls)` on load, an arrow function deals with the response where the array of samples is saved to a variable previously defined. Inside this function, the event listeners for the different types of buttons also get created.

```
let audioBuffers = audioFilesToBuffers(audioUrls).then((response) => {  
    samples = response;  
    ...  
})
```

3.2.1.3. Playing Audio

Since playing audio should happen on interaction, event listeners are required. These require the previously saved samples array where all Web Audio API AudioBuffers are stored. The individual buttons get collected by the methods explained in 3.2.1.1 on page 18. The variables storing the DOM elements are now iterated through to apply different event listeners depending on the button type. Below are two examples. The first one shows the event listener setup for default buttons, and the second one is for hold buttons.

```
for (var i = 0; i < buttonsDefault.length; i++) {  
    buttonsDefault.item(i).addEventListener('click', function () {  
        startAudioContext();  
        playSimpleSample(samples[indexDef], gainNodeDefault);  
    }, false)  
}
```

```

for (var i = 0; i < buttonsHold.length; i++) {
    buttonsHold.item(i).addEventListener('mousedown', function () {
        startAudioContext();
        playHoldSample(samples[indexHold], gainNodeHold, 0.1, 0);
    })
}

document.body.addEventListener('mouseup', () => stopHoldSample());

```

In the two examples above the function `startAudioContext()` appears. This includes a statement which is a requirement for the Web Audio API to function properly and will be further explained in 3.2.1.9 on page 41. The event listeners used to trigger the different interactions generally worked out of the box as intended. However, checking the `mousedown` and `mouseup` events on sliders proved to be an issue while testing the framework. Initially, two event listeners, `mousedown` and `mouseup`, were added to all hold buttons. However, as users could move the mouse outside the object while still changing the values, the framework needed another solution. Replacing the `mouseup` event with `mouseout` was unreliable as sometimes the sound simply would not stop playing and on a second entry would start again, resulting in a layered sound. The solution was to add an event listener onto the website's body to detect the `mouseup` event. After testing, this seemed to be a reliable solution. Interaction with the elements could now trigger all types of interaction sounds, but since hold buttons also required a second event, the play function had to be modified. The default play function is visible here:

```
function playSimpleSample(audioBuffer, gainChannel) {
    const source = audioContext.createBufferSource();
    source.buffer = audioBuffer;
    source.connect(gainChannel).connect(gainNodeMaster).connect(
        audioContext.destination
    );
    source.start(0);
}
```

First, the `AudioContext` creates a new sound source, an `AudioBuffer`. Next, one element from the previously generated samples array is passed as an argument and then passed into the new `AudioBuffer`. Then, before connecting to the destination (the destination is comparable to virtual cables connecting to speakers), different `GainNodes` get connected. The following section will explain these in more detail, but they typically work like volume sliders. Finally, the `start` function is called with the parameter `0`, meaning playback should start immediately. This parameter was passed as an argument in earlier versions of the framework. Regardless, this is unnecessary as a delayed start of interaction sounds would defeat the purpose of interaction sounds.

When writing the play function(s), there were two options for how it/they would be called. The first option was to create a single function with multiple arguments passing at least the correct `AudioBuffer` from the sounds array and the correct `GainNode`. However, hold buttons would require a customized version nonetheless. Thus having modularity and the hold buttons in mind, creating a function for every type of sound was the more logical way to go. This way, developers can easily add effects for single button types like reverb at a later point without creating many type checks inside the

play function. Splitting up the functions had the additional benefit of being less code for a single play and, therefore, easier to understand and most likely faster as there are fewer lines of code to go through before playing the sound. The AudioBuffer argument stayed, however, to have the ability to easily change the sounds either via user interaction or a developer changing the index.

```
var indexDef = 0;
var indexImp = 1;
var indexHov = 2;
var indexHold = 3;
```

Contrary to other play functions, the hold function had to use two events. This is because instead of just playing the sound once, the hold buttons needed a second event to stop the looping audio again. The modified play function can be visible below.

```
function playHoldSample(audioBuffer, gainChannel, rampTime, delayTime) {
    holdSampleRampTime = rampTime;
    if (holdSampleSource !== undefined) {
        stopHoldSample();
    }
    holdSampleSource = audioContext.createBufferSource();
    holdSampleSource.buffer = audioBuffer;
    holdSampleSource.connect(gainNodeFadeInOut).connect(
        gainChannel).connect(gainNodeMaster).connect(
        audioContext.destination);
    gainNodeFadeInOut.gain.setTargetAtTime(1, 0, delayTime + rampTime);
}
```



```
holdSampleSource.start(delayTime);  
holdSampleSource.loop = true;  
holdSampleIsPlaying = true;  
}
```

Since two functions now access the same sound source, the `AudioBuffer` can no longer be isolated inside a single function. Therefore new variables are created in the first few lines.

```
var holdSampleSource = undefined;  
var holdSampleIsPlaying = false;  
var holdSampleRampTime = 0.1;
```

When the page gets loaded, the `holdSampleSource` variable is set to `undefined`. The sound source does not exist yet. On `mousedown` the `playHoldSample()` function gets called. When writing this function, the stop function was not yet called reliably. The solution was to check if the sound source was `undefined`. If not, it immediately stops playback, mutes audio, and sets the `holdSampleSource` to `undefined`. The last step is likely unnecessary but was added just to be safe. Also, developers could then always check the state of whether or not the slider was playing sounds or not instead of adding another event listener. After this check, the sound source connects to the audio context like any other sound type. The difference is that the `gain.setTargetAtTime()` function sets a different `GainNode`'s volume - much like an ADSR curve, or in this case, merely the attack. In this case, the time argument is still available, as a delayed start could be useful for fade-ins. Finally, looping is enabled, and the sample starts to play.

```

function stopHoldSample() {
    if (holdSampleSource !== undefined) {
        if (holdSampleIsPlaying) {
            gainNodeFadeInOut.gain.setTargetAtTime(
                0, 0, holdSampleRampTime
            );
            holdSampleSource.stop(holdSampleRampTime + 0.1);
            holdSampleIsPlaying = false;
        } else {
            holdSampleSource.stop(0);
            holdSampleIsPlaying = false;
        }
    }
}

```

For safety, the stop function checks if the sound source exists. If it does exist, the `setTargetAtTime()` function starts a fade-out, and playback is stopped after a delay. In earlier versions, the sample source was deleted after the stop function, either directly or within a timeout function. However, this is already done automatically by the Web Audio API. After being played, it deletes and garbage-collects `AudioBufferSourceNodes`, whether they ended playback on their own or were force-stopped via `stop()`.

3.2.1.4. Audio Playing Issues

WordPress is a very popular Content Management System which, according to wordpress.org, 43% of the web uses [23]. The developers of WordPress built upon the ability to navigate between pages and posts. While single-page websites are possible and do exist, it is not the intended use of WordPress.

While this framework's approach generally works very well, handling multiple pages was a big issue that was not solvable during development. Although many frontend frameworks provide a great single-page experience, many sites rely on numerous pages, with many running on WordPress and similar CMS. Adding a custom JavaScript file to WordPress is possible, and WordPress functions for PHP are available to do so. Nevertheless, on each page change, the JavaScript file gets loaded again. Because the Web Audio API relies on an active AudioContext before anything is functional, this is a problem. Changing pages initializes that AudioContext again. Pressing, for example, a link in the navigation bar generates a sound. However, clicking this link also changes the page the visitor is viewing. So the sound starts playing but gets cut off as soon as the browser has received the response from the web server and starts displaying the new page. The time until the browser gets the response usually only takes around 100ms. This TTFB depends on many factors, including page size, server and client location, server and client network, and more. But even assuming an audio length of only half a second, a website's TTFB is generally shorter, resulting in the audio being cut mid-play. To clarify, after the TTFB, the site is fully loaded yet. It just means the browser is starting to load the website from nothing and replaces the previous one resulting at the end of audio playback.

Due to time constraints and simply other functions being more critical, there have not been any significant attempts to circumvent this. Though there have been a few ideas:

- Saving the AudioContext

Working with LocalStorage is explained at a later point. The idea, however, was to save the AudioContext to it. Unfortunately, but unsurprisingly, the same problem still exists, except now, after loading the page, the AudioContext is retrieved from LocalStorage instead of being generated. This issue, however, resulted in the idea of using Web Workers as this technology enables background execution of JavaScript files, which without ever having worked with it, at least sounds like a promising solution.

- Delaying site loading

In JavaScript, developers can prevent the default click behavior of links. But a timer could start loading the site after the audio sample finishes. The issue is that this makes the website feel unresponsive for the user and generates extra time where clicking on other buttons is possible. This, in turn, leads to the browser cutting off audio yet again.

- Having a container page

Instead of reloading the Web Audio API on multiple pages, a container page could load the JavaScript file where an iframe shows the website content. However, it has not been tested as part of this project if the Web Audio API can use buttons in iframes with the Web Audio API.

- Back-End implementations

JavaScript can not only be used on the Front-End but also as a Back-End via node.js. However, the most likely issue with this idea is that the Web Audio API was built to be used in the user's browser, not on a server. Therefore, saving AudioContexts to the Back-End is most likely not a possibility.

- No multipage websites

Although not a desirable solution to the problem, this is how it has been dealt with in this project. It will also be the solution for developers that want to use this framework without having to do many modifications. Again, while not ideal, the problem does not exist if the site is only a single-page site.

3.2.1.5. Mixer and Mute

At a very early point in development, it became clear that users should be given control over the sounds played through the framework. While desktop browsers generally provide options to mute individual tabs, adding settings directly to the site was a much cleaner solution. At first, a simple GainNode (a virtual fader) was added to allow users to change the site's volume without having to modify system sounds. However, since there are multiple types of sound, numerous GainNodes made sense. There is also an added benefit of having an extra mixer channel dedicated to hover sounds, which section 3.2.1.9 on page 41 will describe in more detail. Just like an AudioContext, GainNodes are one of the first things to be created, as much of the following code relies on them.

```
const gainNodeMaster = audioContext.createGain();  
gainNodeMaster.gain.value = 0.8;
```

The script above creates such a GainNode; in this case the master GainNode. Additionally, another GainNode has to be generated for every category of sound. All these GainNodes get set to a default volume of 0.8 or 80%. This value is based on the settings of the DAW FL Studio, where the default volume for virtual instruments is 78% and 80% for mixer channels. The 78% stem from the MIDI standard, where 128 values are possible for volume information. Volume sliders often get assigned to a value of 100 per default which corresponds to 78.125%. Because websites do not have constraints like MIDI's 7 bits of information, the value is rounded to a "better looking" 80%.

Working with GainNodes, however, requires some kind of user interface to change the volume. Thus HTML code is necessary for the functionality of this framework. It is possible to embed this in the JavaScript file, but keeping ease of use in mind, it ultimately made more sense for the HTML file to store all the HTML. Developers can already create and style the volume settings before ever touching this framework's JavaScript file. Additionally, changing color classes, for example, on all elements, is simplified when everything styleable is present inside the same file. In addition to volume sliders, users can directly mute all sounds or specific types of sounds. The JavaScript file stores two values other than the GainNode's gain value: the slider's float value and a boolean for the mute button. The GainNode's gain value is now calculated on any value change by multiplying the slider value with either zero or one, depending on whether the user toggled the mute button on or off. The calculation can be seen here, and the assignment of the event listener after that.

```
gainNode.gain.setTargetAtTime(volume * Number(!muted), 0, 0);
```

```

if (!!slider) {
    slider.addEventListener('input', function () {
        if (!!slider) volume = Number(slider.value);
        if (!!sliderValue) sliderValue.innerHTML = String(volume);
        volume.updateLocalStorage();
    });
}

```

In the case shown above, the `gainNode.setTargetAtTime()` function is preferred over directly setting `gainNode.gain.value` to a specific value as the function has higher priority than assigning the value directly. The variable `masterVolSliderExists` is a boolean value calculated when gathering all event listener input triggers on value change.

3.2.1.6. Saving Settings

While a big issue was the inability to make multipage websites work correctly, another case was re-opening the site. For example, setting the volume of a sound category would be saved while visiting the site. But upon reloading, the browser reset everything to the default 80% by reloading the JavaScript file. A well-working solution for this is LocalStorage. LocalStorage is a simple JavaScript object that gets saved to a user's browser. Reloading pages and clearing the cache does not clear LocalStorage thus, settings saved to LocalStorage would generally be safe from auto-cleaning browser extensions and users troubleshooting pages.

Saving to LocalStorage is done by adding variables to the LocalStorage object. Here is an example using the master volume:

```
localStorage.volMaster = 0.8;  
localStorage.volMasterMuted = false;
```

When this is first run, LocalStorage is empty. However, by assigning a value to a non-existent variable inside the LocalStorage object, the variable is also created. However, checking if the variable exists is still necessary because settings should also be read from and not only saved to LocalStorage. The example below shows how this project does this.

```
volMaster = localStorage.volMaster === undefined ? 0.8 :  
    parseFloat(localStorage.volMaster);  
volMasterMuted = localStorage.volMasterMuted === undefined ? false :  
    parseStringToBoolean(localStorage.volMasterMuted);
```

In both cases above, the conditional or ternary operator is an elegant and fast way to check if the value exists. In essence, it is a shorter version of an if-else statement. The first section performs the if statement - in this case `localStorage.volMaster === undefined`. After the question mark and separated by the colon, two values get returned following the result of the condition; the first one if the condition is true, and the second one if it is false. Since a value gets returned, developers can assign the result to a variable. The statement could also be written as a function like the following:


```

volMasterMuted = getFromLocalStorage();

function getFromLocalStorage() {
    var value;
    if (localStorage.volMasterMuted === undefined) {
        value = 0.8;
    } else {
        value = localStorage.volMasterMuted;
    }
    return value;
}

```

In both cases, it is necessary to parse the data because variables saved to LocalStorage are stored as strings. The function `parseFloat` is already available in JavaScript. For booleans, however, there is no built-in function as JavaScript has a different system to deal with any type of data to convert it into a boolean. MDN writes the following [24]: “In JavaScript, a truthy value is a value that is considered true when encountered in a Boolean context. All values are truthy unless they are defined as falsy. That is, all values are truthy except false, 0, -0, on, "", null, undefined, and NaN. “

As the MDN wrote, only empty strings (“”) are falsy. Thus, booleans returned inside strings are always `true`. The following custom function circumvents this issue:

```

function parseStringToBoolean(s) {
    if (s === "true") return true;
    else return false;
}

```

The function above could, in theory, be passed a different string, and it would return **false**. However, since the only usage is getting the information on whether a single type of audio was muted or not, and changing that value requires users to tamper with LocalStorage via the console, the decision was made that this is not an issue.

3.2.1.7. Volume Groups

Saving volume settings quickly became a lot of work because the same lines of code had to be written with minor variations for each type of sound, all being a potential source of error. As prevention, a custom class was created to store volume data and easily access it. The class is visible here:

```
class VolumeGroup {  
    volume;  
    muted;  
    type;  
  
    constructor(volume, muted, type) {  
        this.volume = volume;  
        this.muted = muted;  
        this.type = type;  
    }  
  
    get Volume() { return this.volume; }  
    set Volume(v) { this.volume = v; }
```

```

get Muted() { return this.muted; }
set Muted(v) { this.muted = v;
getCurrentVol() { return this.volume * Number(!this.muted) }
...
}

```

An individual **VolumeGroup** object always corresponds to a volume slider and a mute button. Currently, the project uses five **VolumeGroup** objects for the sound types previously described. Getters and setters have been defined for the volume and the mute variable, which get triggered on the “input” event in case of the sliders and the “change” event in case of the mute buttons. The event listeners will be described further in the next section when **SettingsGroup** objects have been introduced. A **getCurrentVol()** function is also present. However, this is currently not used anymore as the calculations for the GainNodes are done in the next function:

```

class VolumeGroup {
...
updateLocalStorage() {
    switch (this.type) {
        case "master":
            localStorage.volMaster = this.volume;
            localStorage.volMasterMuted = this.muted;
            gainNodeMaster.gain.setTargetAtTime(
                this.volume * Number(!this.muted), 0, 0);
            break;
...

```

This function updates LocalStorage and the corresponding GainNode. The volume gets directly sent to LocalStorage, but the GainNode has to handle both the mute and the volume variables. Multiplying the volume with either `0` or `1`, depending on whether muted is `true` or `false`, makes it possible to use it in a single GainNode. Luckily, in JavaScript, this does not rely on any custom functions. As mentioned before, all data types are either truthy or falsy. While this can be used for numbers and strings as logical operators, another option is to turn it the other way around, converting booleans to different data types. In the case of this project, the boolean value of muted gets turned into a number. This number is `0` for the boolean `false` and `1` for `true`. First, the muted value has to be negated, though, as a muted signal should correspond to a value of zero. This negation was preferred over a more complicated variable name such as `notMuted` or `volumeEnabled` as these versions are less understandable.

Creating a `VolumeGroup` is now easily done by the following statement:

```
var volMaster = new VolumeGroup(0.8, false, "master");
```

This also replaces the ternary operator statement from the section before with the use of getters and setters:

```
volMaster.Volume = localStorage.volMaster === undefined ? 0.8 :  
    parseFloat(localStorage.volMaster);
```

3.2.1.8. Settings Groups

Similar to `VolumeGroup` objects, settings are condensed into one object. The custom `SettingsGroup` class can be seen below:

```
class SettingsGroup {  
    slider  
    sliderValue  
    mute  
  
    constructor() {  
        this.slider = undefined;  
        this.sliderValue = undefined;  
        this.mute = undefined;  
    }  
  
    get Slider() { return this.slider; }  
    get SliderValue() { return this.sliderValue; }  
    get Mute() { return this.mute; }  
    set Slider(v) { this.slider = v; }  
    set SliderValue(v) { this.sliderValue = v; }  
    set Mute(v) { this.mute = v; }  
}
```

Contrary to `VolumeGroup` objects, `SettingsGroup` objects do not get assigned in the constructor. While the volume settings can exist without the means to modify them, `SettingsGroup` objects refer to HTML elements in the DOM; thus, an undefined state

has to be possible if a developer chooses not to implement, for example, mute buttons. Getters and setters exist for all variables. While setters generally only get called once, getters are called whenever the settings change. Creating, for example, `settingsMaster` can look like this:

```
var settingsMaster = new SettingsGroup();
settingsMaster.Slider = document.getElementById('volSliderMaster');
settingsMaster.SliderValue = document.getElementById(
    'volSliderMasterValue');
settingsMaster.Mute = document.getElementById('volMuteMaster');
```

Here is the function responsible for linking a `VolumeGroup` to a `SettingsGroup`:

```
function addVolumeSettingsEventListeners(settingsGroup, volumeGroup) {
    if (!!settingsGroup.Slider) {
        settingsGroup.Slider.addEventListener('input', function () {
            if (!!settingsGroup.Slider) volumeGroup.Volume = Number(
                settingsGroup.Slider.value);
            if (!!settingsGroup.SliderValue) settingsGroup.
                SliderValue.innerHTML = String(volumeGroup.Volume);
            volumeGroup.updateLocalStorage();
        });
    }

    ...
}
```

```

if (!!settingsGroup.Mute) {
    settingsGroup.Mute.addEventListener('change', function () {
        if (this.checked) volumeGroup.Muted = true;
        else volumeGroup.Muted = false;
        volumeGroup.updateLocalStorage();
    });
}
}

```

As seen above, event listeners will only get applied if the if statements are **true**. The double quotation mark is a double negation. It relies on the concept of truthy and falsy values again, as one quotation mark negates the input and changes the datatype to boolean. The second one negates again, resulting in two negations (or no negation). This is used to check if the element even exists. Suppose it does not, `settingsGroup.Slider` returns `undefined`, which is a falsy value, thus resulting in **false** after the double negation. The event listeners, therefore, do not get applied if the element does not exist. Inside the event listener, there is another check if the elements exist, though at least the slider should exist at that point. After moving the project to TypeScript, this, however, was marked as an error. Another check was implemented to get rid of this error.

3.2.1.9. Audio Context Startup and Settings for Site Owners

A significant limitation of this framework is that browsers block audio before user interaction with the website. So when loading a site with the Web Audio API enabled, the `AudioContext` gets disabled until a user has clicked somewhere on the site. This is, in many cases, a welcome feature, for example, when opening YouTube tabs in the back-

ground. For this framework, however, it is a problem. When freshly loading a site, pressing buttons can have a noticeable delay, and hover sounds do not work at all. Thus a notice is necessary which catches the first interaction with a user before the main site with all types of interaction sounds is working. This notice can be any shape or form as long as the user has to click once before visiting the site's content. For example, with W3.CSS, this could look like the following:

```

<!-- Trigger/Open the Modal -->
<button id="openModal" onclick="document.getElementById(
    'w3-modal').style.display='block'" class="w3-button w3-hide">Open
    Modal </button>
<!-- The Modal -->
<div id="w3-modal" class="w3-modal">
    <div class="w3-modal-content">
        <div class="w3-container">
            <span onclick="document.getElementById('w3-modal').style.
                display='none'" class="w3-button w3-indigo
                w3-display-topright">&times;</span>
            <p>Unfortunately, browsers block a site's sound until a
            user has interacted with it.<br>Thus, for full functionality of this
            site, this popup is a requirement.<br><br>You can change sound settings
            at any time by visiting the settings menu.</p>
        </div>
    </div>
</div>

```


The `w3-hide` class on the button element sets its display property to none. Thus users will not see it on the website. In JavaScript, however, it can be referenced via its id and open with the `click()` function. Since only hover sounds are majorly affected by browsers disabling sounds, a small extra feature was developed:

```
if (hoverSoundsAcitvated) {  
    if (!volHover.Muted) {  
        if (!!modalButton) modalButton.click();  
    }  
}
```

The constant `hoverSoundsActivated` is set at the beginning of the settings file and completely disables hover sounds and the popup. If hover sounds are activated, then a check is performed if they are muted. Then, the modal informing users about the browser's sound blocking is shown if they are not muted.

3.2.2. Enhancements Through TypeScript

While developing the framework, many minor errors occurred, which individually would be far too insignificant to write about in this thesis. However, combined, they made up a considerable amount of time spent with the framework. While restructuring code numerous times was helping with clarifying what each function and each section actually does to be able to re-write it, many errors stemmed from other issues like wrong types being set in functions. These errors were not easily solvable before running the code on the website, and even then, they could have been from anywhere as many

functions are interconnected. Adding TypeScript to the project was a huge help in figuring out these issues. Types in classes were predominantly already working out of the box, and for all other instances, only minor adjustments had to be made to show and eliminate all type errors. For example, parsing from boolean to string was overlooked entirely, but TypeScript helped figure this issue out before even testing it.

```
type BooleanString = "true" | "false";

function parseStringToBoolean(s: BooleanString): boolean {
    if (s === "true") return true;
    else return false;
}
```

As visible above, a new type was declared to convert booleans inside of quotes into real booleans. Another instance where TypeScript had a significant impact on bug fixing was the `SettingsGroup` class, as there was the possibility of HTML elements not being added. TypeScript allows multiple types to be assigned to a single variable. This helped solve issues where functions or variables returned something other than an `HTMLInputElement` or `undefined`.

```
class SettingsGroup {
    private slider: HTMLInputElement | undefined
    private sliderValue: HTMLElement | undefined
    private mute: HTMLInputElement | undefined

    ...
}
```

```

constructor() {
    this.slider = undefined;
    this.sliderValue = undefined;
    this.mute = undefined;
}

public get Slider(): HTMLInputElement | undefined {
    return this.slider;
}

public get SliderValue(): HTMLElement | undefined {
    return this.sliderValue;
}

public get Mute(): HTMLInputElement | undefined {
    return this.mute;
}

public set Slider(v: HTMLInputElement | undefined) {
    this.slider = v;
}

public set SliderValue(v: HTMLElement | undefined) {
    this.sliderValue = v;
}

public set Mute(v: HTMLInputElement | undefined) {
    this.mute = v;
}
}

```

3.2.3. The Front-End

3.2.3.1. Usage with CSS Frameworks

As explained before, one of the essential features of this framework was its flexibility to be used with virtually any other technology except multipage websites. Therefore, a small site was created showcasing all framework features three times with three different CSS frameworks demonstrating the before-mentioned flexibility.

The first choice was W3.CSS. Unlike the other two frameworks, this was purely a personal choice. I have often chosen W3.CSS in past projects because building basic layouts was always a joy and was accomplished quickly.

The second choice was Bootstrap, as, according to GitHub, it is the most popular CSS framework available. Furthermore, as of August 2022, it is the ninth most starred repository on GitHub [25] and by far the most starred CSS framework [26]. These statistics greatly influenced the decision to include a version for Bootstrap and the third and final framework.

Because Bootstrap and W3.CSS are both frameworks with many pre-built components, for the third implementation, I was looking for something different. Tailwind CSS is a newer approach to CSS frameworks providing classes to build components instead of pre-built components. This approach is popular: As of August 2022, Tailwind CSS is the second most starred CSS framework on GitHub [26].

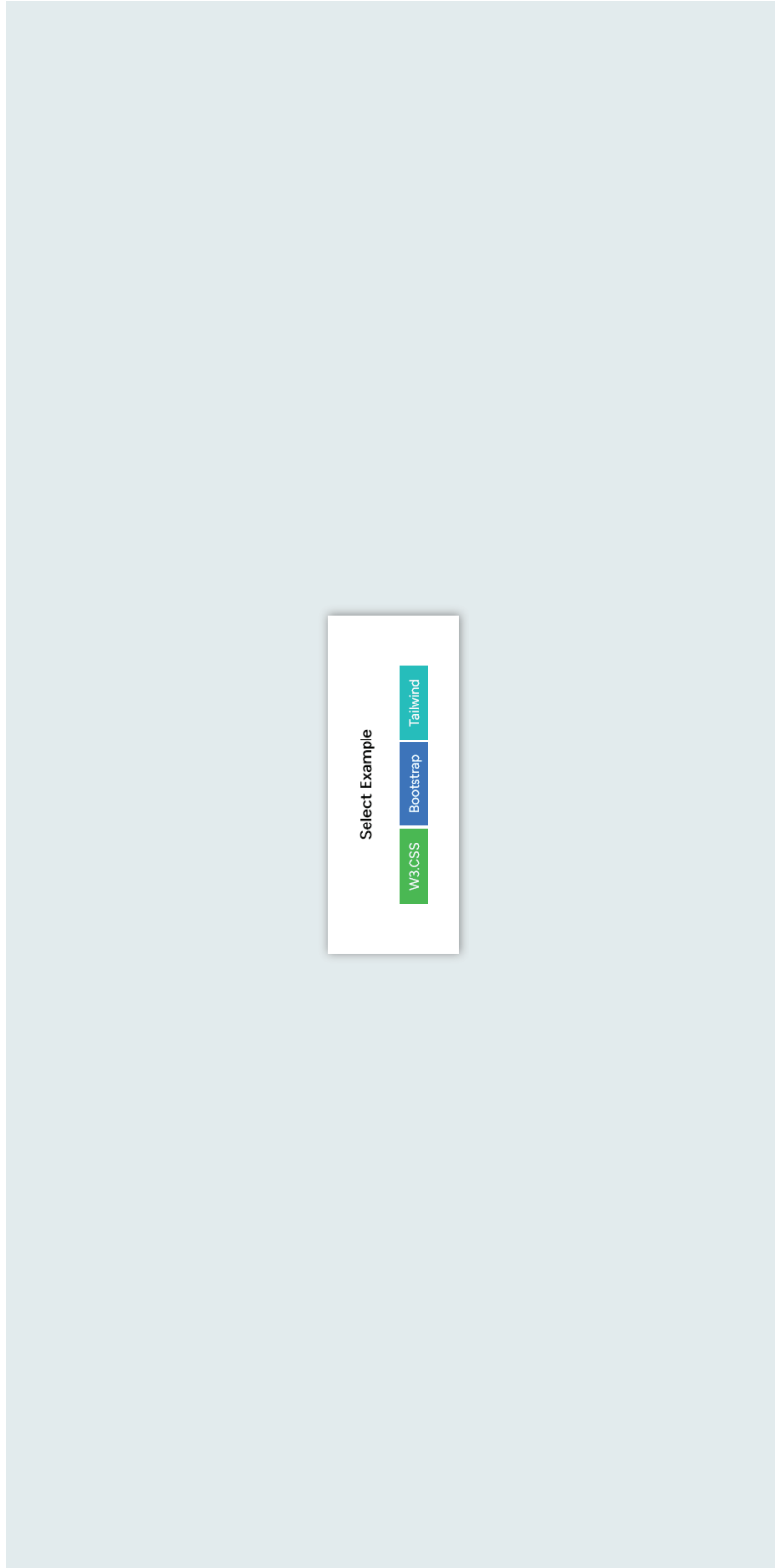


Figure 3.1: Screenshot of CSS Framework Selection

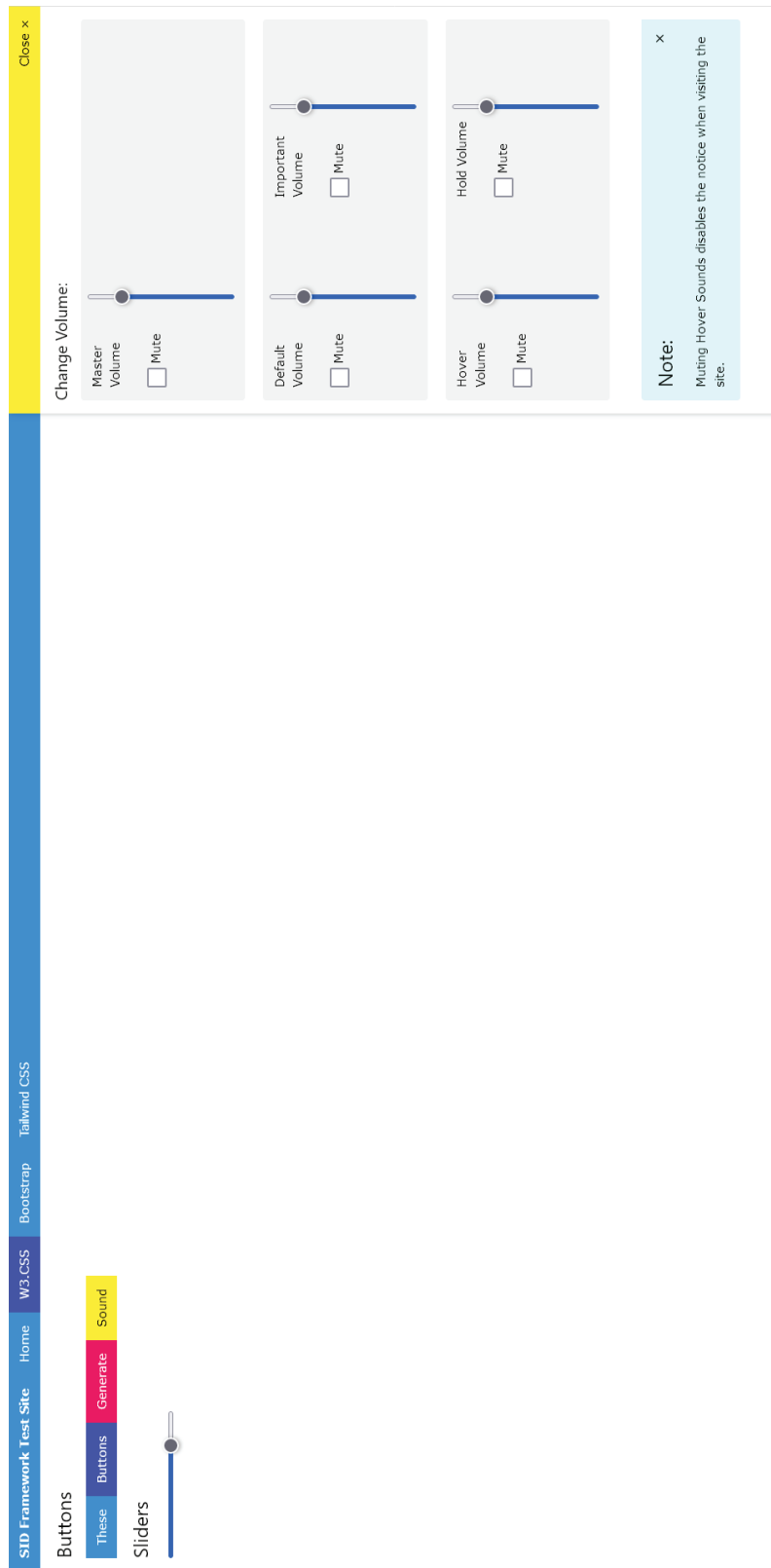


Figure 3.2: Screenshot of W3.CSS Implementation

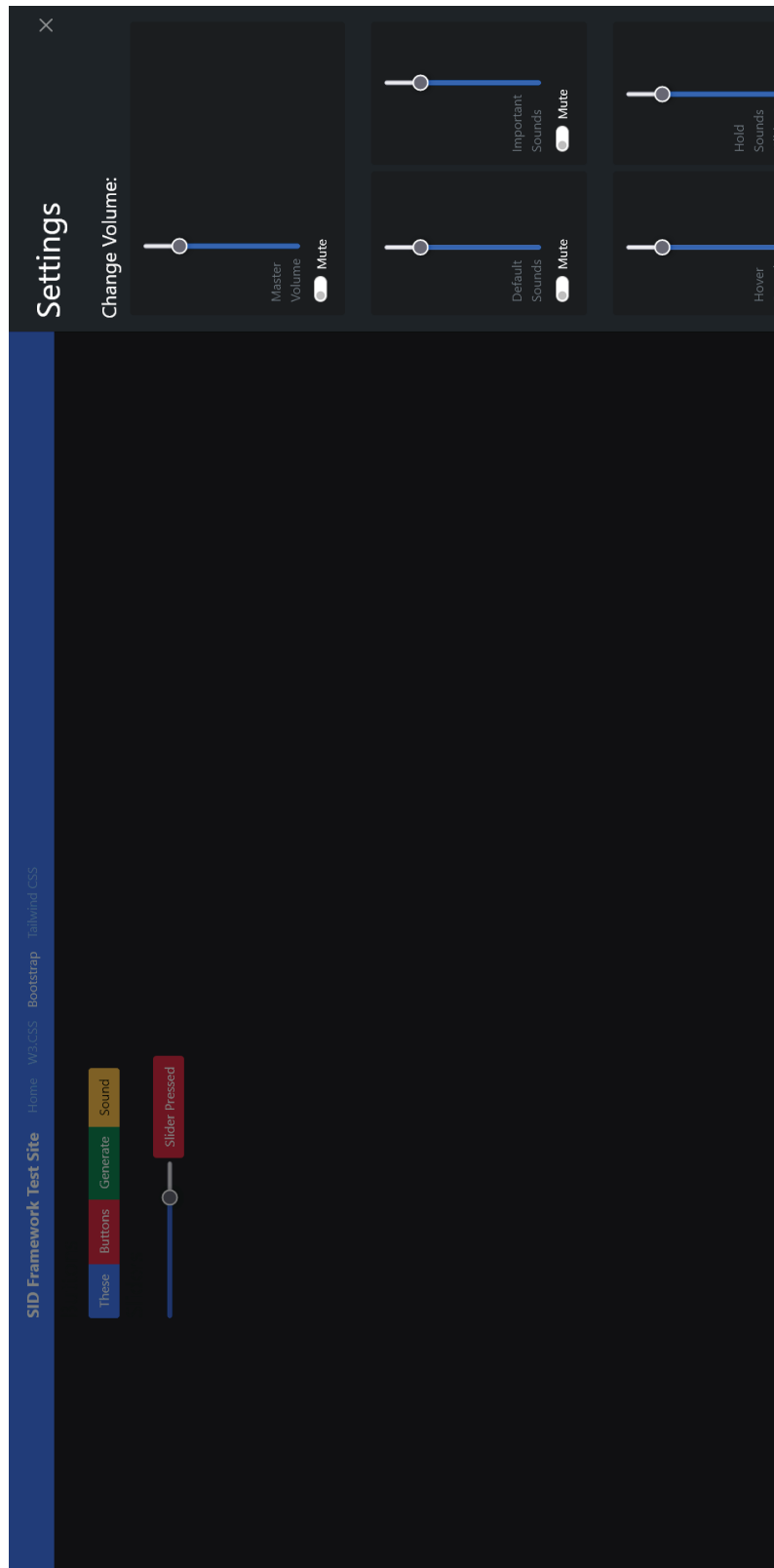


Figure 3.3: Screenshot of Bootstrap 5 Implementation

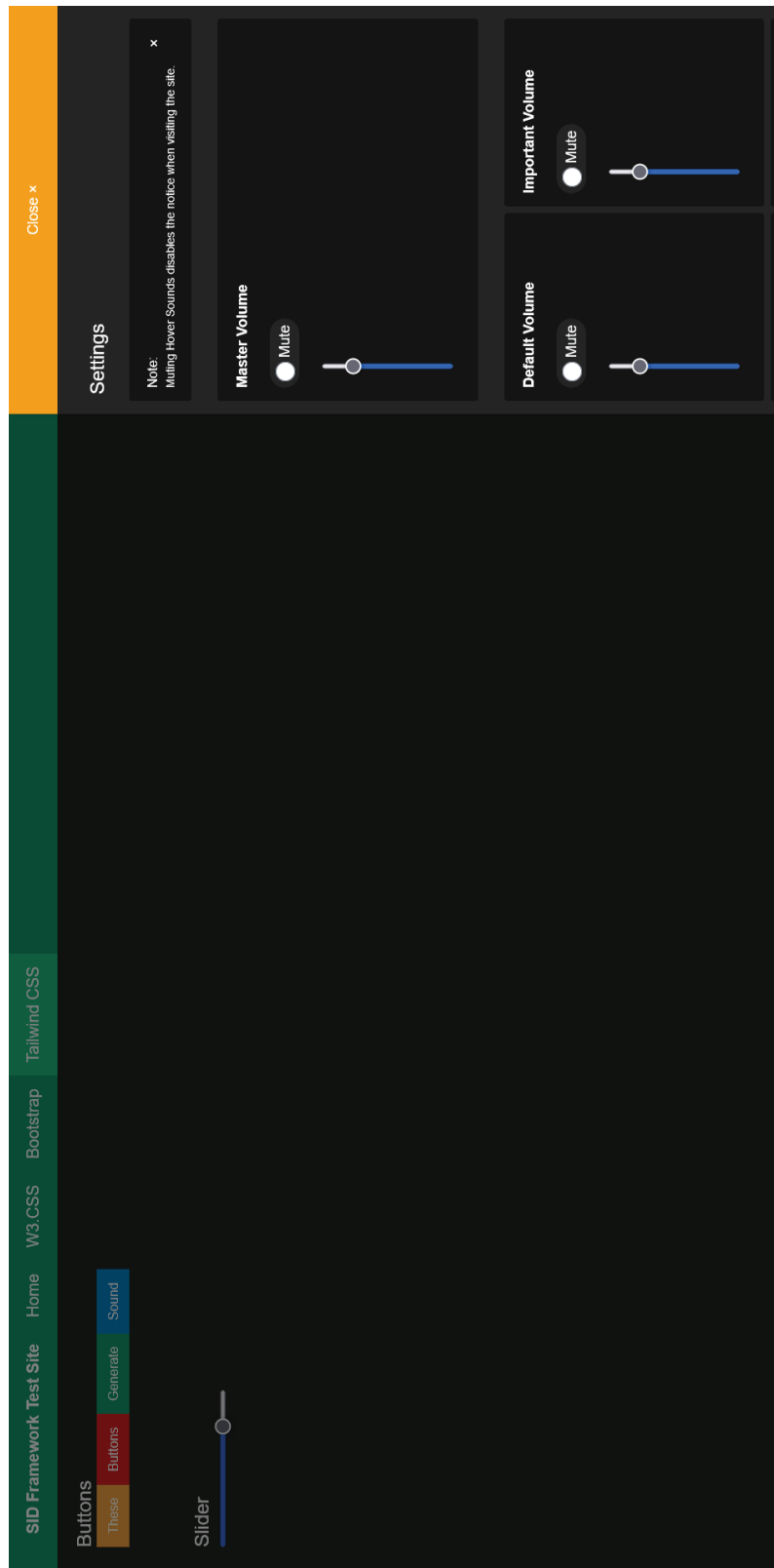


Figure 3.4: Screenshot of Tailwind CSS Implementation

3.2.3.2. Mixer Styling

The HTML part simply consists of a slider and a checkbox. CSS Frameworks can style these in any way. However, the elements have to have unique ids assigned. Per default configuration, these are called volSlider or volMute with an ending depending on the type. For the master channel, this would be volSliderMaster and volMuteMaster. While setting the ids is sufficient, a better approach would be to add labels for users to understand what the individual intractable object changes. Below are two examples of how these sliders could look. The left image shows default browser styling without any CSS. The right image shows a version with Bootstrap 5 styling.



Figure 3.5: Screenshot of Default Styling for Inputs of Type Range and Checkbox

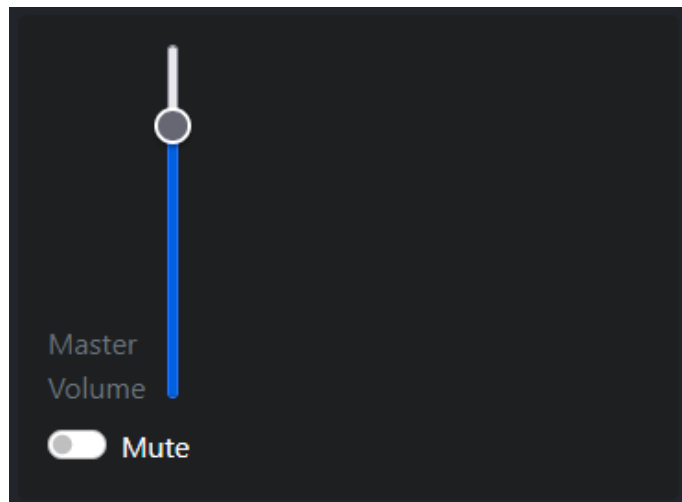


Figure 3.6: Screenshot of Bootstrap 5 Styling for Inputs of Type Range and Checkbox

```

<div>
  <input type="range" min="0" max="1" step="0.01"
    value="0.8" orient="vertical" id="masterVolSlider" />
  <br>
  <label for="masterVolSlider">Master Volume</label>
  <br><br>
  <label for="muteMasterVol">Mute</label>
  <input type="checkbox" role="switch" id="muteMasterVol" />
</div>

<div>
  <label for="masterVolSlider" class="form-label text-secondary">
    Master<br>Volume
  </label>
  <input type="range" min="0" max="1" step="0.01" value="0.8"
    orient="vertical" id="masterVolSlider" />
  <br>
  <div class="form-check form-switch">
    <input class="form-check-input" type="checkbox" role="switch"
      id="masterVolMute" />
    <label class="form-check-label" for="masterVolMute">Mute
  </label>
  </div>
</div>

```

4. Conclusion

Interaction sounds on websites are currently not yet widely used. Therefore, this thesis tried to answer the following two questions:

Can websites also use sonic interaction design.

Can a user- and developer-friendly framework be created to accomplish this?

The theory shows that sound can add much value to interfaces, which is certainly also true for websites. Moreover, I believe this project has accomplished this, but I am also convinced that there is much more potential for sound on websites.

As more and more programs get moved to the cloud, and entire operating systems (e.g., ChromeOS) rely on web apps, website interfaces play a more vital role. Currently, however, browsers disabling sound output until a user interaction has occurred limits how sound can be used. For example, I can imagine administrative panels for home servers that send notifications enriched by sonic feedback. Such a panel could be left running in the background to catch necessary alerts - but unfortunately, restarting the browser requires users to interact with the site again, taking its toll on productivity. Another issue is multipage support. If AudioContexts were saved per site, sounds could still be played while loading other pages.

Following this research, the next logical step for me would be to look at accessibility which I have not touched on yet. I believe that sonic feedback could provide helpful information for visually impaired people. First, however, research is necessary to discover how to incorporate this into existing technologies like screen-readers and if this is even necessary, as there are other options like Braille readers.

5. Table of Figures

2.1:	Screenshot of Opera GX Sound Options	4
2.2:	Screenshot of SteamDB most played games as of 18. August 2022	6
2.3:	Screenshot of Counter-Strike: Global Offensive Sound Settings	7
2.4:	Screenshot of Grand Theft Auto V Sound Settings	9
2.6:	Screenshot of EMUI 11 “More settings” Sound Settings	10
2.5:	Screenshot of EMUI 11 “Sound & vibration” Settings	10
2.7:	Screenshot of iOS 15 “Sound & Haptics” Settings	11
2.8:	Screenshot of macOS 12 Sound Settings	12
2.9:	Screenshot of Windows 11 Sound Settings	13
2.10:	Screenshot of Windows 11 Sound Mixer	14
2.11:	Screenshot of Windows 11 Win32 Sound Options	15
2.12:	Screenshot of Windows 11 Win32 Sound Options [13]	15
2.13:	Screenshot of Ubuntu 22 LTS Sound Settings	16
3.1:	Screenshot of CSS Framework Selection	47
3.2:	Screenshot of W3.CSS Implementation	47
3.3:	Screenshot of Bootstrap 5 Implementation	48
3.4:	Screenshot of Tailwind CSS Implementation	48
3.5:	Screenshot of Default Styling for Inputs of Type Range and Checkbox	49
3.6:	Screenshot of Bootstrap 5 Styling for Inputs of Type Range and Checkbox	49

6. Bibliography

- [1] D. Rocchesso et al., “Sonic interaction design: sound, information and experience,” in CHI '08 Extended Abstracts on Human Factors in Computing Systems, New York, NY, USA, Apr. 2008, pp. 3969–3972. doi: 10.1145/1358628.1358969.
- [2] S. Brewster and M. Crease, “Correcting menu usability problems with sound,” *Behav. Inf. Technol.*, vol. 18, May 1999, doi: 10.1080/014492999119066.
- [3] J. Hereford and W. Winn, “Non-Speech Sound in Human-Computer Interaction: A Review and Design Guidelines,” *J. Educ. Comput. Res.*, vol. 11, no. 3, pp. 211–233, Oct. 1994, doi: 10.2190/MKD9-Wo5T-YJ9Y-8iNM.
- [4] K. Orland, “Valve leaks Steam game player counts; we have the numbers,” *Ars Technica*, Jun. 07, 2018. <https://arstechnica.com/gaming/2018/07/steam-data-leak-reveals-precise-player-count-for-thousands-of-games/> (accessed Aug. 18, 2022).
- [5] SteamDB, “Steam Charts and Stats · Most Played Games on Steam,” SteamDB. <https://steamdb.info/graph/> (accessed Aug. 18, 2022).
- [6] Valve Corporation, “A New Horizon,” *Counter-Strike: Global Offensive Blog*, Aug. 01, 2018. <https://blog.counter-strike.net/index.php/2018/08/20738/> (accessed Aug. 18, 2022).

- [7] Valve Corporation, “Counter-Strike: Global Offensive.” Valve Corporation, Bellevue, Washington, US, Aug. 21, 2012.
- [8] Rockstar Games, Inc., “Grand Theft Auto V.” Rockstar Games, Inc., New York City, US, Sep. 17, 2013.
- [9] Huawei Technologies Co., Ltd., “EMUI 12.” Huawei Technologies Co., Ltd., Shenzhen, China, Aug. 26, 2021.
- [10] Samsung Electronics Co., Ltd., “One UI 4.” Samsung Electronics Co., Ltd., Seoul, South Korea, Nov. 2021.
- [11] Apple Inc., “macOS 12 Monterey.” Apple Inc., Cupertino, California, U.S., Oct. 25, 2021.
- [12] W. Glenn, “How to Adjust the Volume for Individual Apps in Windows,” How-To Geek. <https://www.howtogeek.com/244963/how-to-adjust-the-volume-for-individual-apps-in-windows/> (accessed Aug. 19, 2022).
- [13] Microsoft Corporation, “Windows 11.” Microsoft Corporation, Albuquerque, New Mexico, U.S., Oct. 2021.
- [14] Canonical Foundation and Ubuntu Community, “Ubuntu 22.04 LTS.” Canonical Foundation, London, UK, Apr. 21, 2022.

- [15] Canonical Foundation, “Ubuntu PC operating system,” Ubuntu. <https://ubuntu.com/desktop> (accessed Aug. 20, 2022).
- [16] M. Smith, “HTML 5 Publication Notes,” W3C, Jun. 10, 2008. <https://www.w3.org/TR/2008/NOTE-html5-pubnotes-20080610/> (accessed Jul. 27, 2022).
- [17] R. MacManus, “Web Frameworks: Why You Don’t Always Need Them,” The New Stack, Feb. 15, 2021. <https://thenewstack.io/case-against-web-frameworks/> (accessed Jul. 27, 2022).
- [18] MDN contributors, “AudioContext - Web APIs | MDN,” MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/API/AudioContext> (accessed Jul. 27, 2022).
- [19] W3Schools, “W3.CSS Buttons.” https://www.w3schools.com/w3css/w3css_buttons.asp (accessed Jul. 27, 2022).
- [20] M. Otto, J. Thornton, and Bootstrap, “Buttons.” <https://getbootstrap.com/docs/5.0/components/buttons/> (accessed Jul. 27, 2022).
- [21] M. Taylor, “LAME Technical FAQ,” LAME Technical FAQ. <https://lame.sourceforge.io/tech-FAQ.txt> (accessed Jul. 28, 2022).
- [22] MDN contributors, “Basic concepts behind Web Audio API - Web APIs | MDN,” MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Basic_concepts_behind_Web_Audio_API (accessed Jul. 27, 2022).

- [23] WordPress.org, “Blog Tool, Publishing Platform, and CMS,” WordPress.org. <https://wordpress.org/> (accessed Jul. 29, 2022).

- [24] MDN contributors, “Truthy - MDN Web Docs Glossary: Definitions of Web-related terms | MDN.” <https://developer.mozilla.org/en-US/docs/Glossary/Truthy> (accessed Aug. 10, 2022).

- [25] GitHub, Inc, “GitHub Search Top Repositories.” <https://github.com/search?o=desc&q=stars%3A%3E1&s=stars&type=Repositories> (accessed Aug. 12, 2022).

- [26] GitHub, Inc, “css-framework · GitHub Topics · GitHub.” <https://github.com/topics/css-framework?o=desc&s=stars> (accessed Aug. 12, 2022).