AuO: Audio Recorder and Editor on the Web

by

Luo Qian

S.B., Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

© Massachusetts Institute of Technology 2016. All rights reserved.

Author .................................................................

Department of Electrical Engineering and Computer Science

August 5, 2016

Certified by ............................................................

Professor Eric Klopfer

Professor; Director, MIT Scheller Teacher Education Program

Thesis Supervisor

Accepted by ............................................................

Dr. Christopher J. Terman

Chairman, Department Committee on Master of Engineering Theses
AuO: Audio Recorder and Editor on the Web

by

Luo Qian

Submitted to the Department of Electrical Engineering and Computer Science
on August 5, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In grade school, students learn that human beings possess five main senses: sight, hearing, taste, smell, and touch. Of these, we currently can experience only the first two through a computer, and by extension, the Internet. Although web technologies have supported visual feedback since the earliest of times, only recently have browser provided web developers with audio technologies capable of recording, editing, and playing audio without resorting to external-to-browser plug-ins, such as ActionScript and Java. This presents a ripe opportunity for web applications to leverage the power of the web and deliver immersive auditory experience that appeal to users’ ears.

This thesis presents AuO, an online audio recording and editing application. Inspired by the needs of StarLogo Nova, AuO uses only HTML, CSS, and JavaScript to allow web developers to provide their users with the ability to record audio from their browsers, load audio from the Internet or users’ machines, make edits to the audio, play back the audio as a preview, and save the audio to either the Internet or users’ local filesystems.

Designed with both web developers and users in mind, and implemented with the aim of delivering speedy and smooth experiences, AuO’s development process, from conception to future work, documents the creation of the first integrable native-to-browser web audio recorder and editor.

Thesis Supervisor: Professor Eric Klopfer
Title: Professor; Director, MIT Scheller Teacher Education Program
Acknowledgments

“The journey of a thousand miles begins with a single step.” – Laozi

This thesis not only presents the terminal work for my master’s project in the Scheller Teacher Education Program, but also the journey that brought me here this far, and will carry me forward. I cannot sufficiently express my gratitude to all the people who have helped me in reaching this point in my life, and especially to the key people who deserve special mentions for journeying with me so far in my life.

First off, many thanks to Daniel Wendel, the lead developer of StarLogo Nova and my project supervisor, for giving me the opportunity to work with him. He entrusted a large bulk of StarLogo Nova to two master’s students, and let us run amok with it, allowing us to fully explore the field. As a result, I have learned so much more than I initially thought I would through this experience. Additionally, Daniel’s patience and willingness to guide me when I needed help has made my experience through this project all the smoother. Daniel, thank you for your guidance throughout my project and thesis.

I would also like to thank Professor Eric Klopfer, my faculty supervisor, for allowing Daniel to persuade him to take me on as an MEng student, and taking the time to also guide me and provide feedback on my project.

Additionally, in the vein of celebrating the journey, I also want to express my gratitude to several people who, perhaps unknowingly, have made major contributions to the path that has led me here. To my role model, Jessica F. Liu, thank you for inspiring me to achieve, work hard, and aspire to attend MIT by leading the way yourself. To my roommate, Benjamin F. Reynolds, thank you for your kindness and friendship since September, 2012; you can always brighten an otherwise gloomy day with your cheerful attitude, and you never complain about my cooking, even when I stress...
cook. To Diane Zhou, who has more than once listened to me complain about all the craziness that abounds in the MEng life and encouraged me afterward, thank you for staying patient with me and inviting me to socialize when I otherwise would have not. To Mrs. Barbara Burnett and Mr. Vern Williams, thank you for encouraging me to try hard and explore, even when I do not know the direction. To Mr. Michael Stueben, thank you for not only teaching me for four year straight in high school, but also encouraging me to learn for knowledge’s sake, to seek resources above and beyond, to ask for help when I have lost my way, and to challenge the status quo. To Professor Charles Leiserson, thank you for showing me that research contains more than just writing code and meeting benchmarks, and challenging me to write using a wider variety of verbs other than the forms of the verb, “to be,” and I hope that this thesis makes a reasonable attempt at completing that challenge. To Professor Saman Amarasinghe, thank you for demonstrating that composure in the face of uncertainty can make the difference between success and failure while working with you and the stellar staff of 6.172. To Professor Rob Miller and Dr. Max Goldman, thank you for allowing me to learn by teaching alongside you and fantastic staff teams in the many semesters of 6.005 that you gave me the honor of staffing.

Finally, I want to thank my loving parents, Xuan Luo and Gefei Qian, who carried me through my formal education in the United States. By having me experience schooling in a variety of environments, you have given me the chance to see opportunities in modern education and the motivation to pursue them. By giving me independence and responsibilities starting from a young age, you allowed me opportunities to learn how to manage my life. And even though I did not choose a career in medicine, law, or business, you have given me the hope and courage to undauntedly test new boundaries by electing to believe in and encourage me.
Contents

1 Introduction .................................................. 17
   1.1 Motivation ............................................. 18
   1.2 Constraints Overview ................................ 18
       1.2.1 Project Metrics ................................. 19
   1.3 Contributions Overview .............................. 21
   1.4 Thesis Organization ................................. 22

2 Background .................................................... 23
   2.1 Previous Work ......................................... 24
       2.1.1 Online Audio Recording and Editing Applications . 24
       2.1.2 Native Browser APIs ............................ 25

3 Design .......................................................... 27
   3.1 High-Level Design Goals ............................. 28
       3.1.1 User Interface .................................. 28
       3.1.2 Client API ...................................... 29
       3.1.3 Integration ..................................... 29
   3.2 Initial Design (v0.1) ................................. 30
       3.2.1 Recording ...................................... 30
       3.2.2 Playback ....................................... 31
       3.2.3 Saving ......................................... 32
       3.2.4 Controls UI .................................... 33
       3.2.5 Overlaying ..................................... 33
       3.2.6 Waveform Displaying ......................... 34
4.12 Loading ......................................................... 74
  4.12.1 File Retrieval ............................................. 74
  4.12.2 Data Retrieval ............................................. 75
4.13 Tagging .......................................................... 75
  4.13.1 Updating the Tags UI ...................................... 75
  4.13.2 User Interactions .......................................... 77
  4.13.3 Client Interactions and the API .......................... 78
4.14 Hotkeys .......................................................... 78
4.15 Modals ........................................................... 79
4.16 Implementation Summary ....................................... 80

5 Testing ............................................................. 83
  5.1 Automated Testing ............................................. 84
    5.1.1 Environment .............................................. 84
    5.1.2 Code Style ............................................... 84
    5.1.3 Code Organization ....................................... 84
    5.1.4 Server Package ......................................... 85
    5.1.5 Client Package .......................................... 88
    5.1.6 Tests Package ........................................... 89
    5.1.7 Limitations ............................................... 90
    5.1.8 Validation Procedure .................................... 91
    5.1.9 Automated Testing Summary ............................. 91
  5.2 Client and User Testing ....................................... 92
    5.2.1 Key Contributors ........................................ 93
    5.2.2 WAV Support ............................................. 93
    5.2.3 Drag-and-Drop ........................................... 93
    5.2.4 Tags UI .................................................. 94
    5.2.5 MIME Support ............................................ 94
    5.2.6 Integrating with StarLogo Nova ......................... 95
    5.2.7 TSL Feedback ............................................ 96
List of Figures

3-1 AuO v0.1 controls UI, in the “idle state.” ............................... 33

3-2 A two-channel waveform of Rick Astley’s 1987 song, Never Gonna Give You Up [2]. AuO supports multiple channels as of v0.3, and file loading as of v1.4. ......................................................... 34

3-3 Zoom UI, for desktop (left) and mobile (right). ...................... 35

3-4 The ticker on the audio track shown in Figure 3-2 when representing a time in the first half of the audio track. .......................... 36

3-5 The ticker on the audio track shown in Figure 3-2 when representing a time in the second half of the audio track. ......................... 36

3-6 Start and end trimmers used on the audio track shown in Figure 3-2. 37

3-7 Save UI in AuO version 1.0. ....................................................... 41

3-8 Resources indicator, in online mode (left) and local mode (right). . 45

3-9 Load UI, in online mode (left) and local mode (right). ............... 45

3-10 Save UI, in online mode (left) and local mode (right). .............. 45

3-11 Tags UI, with three tags. ................................................. 46

3-12 Modals for the default save callback (top) and tag labeling (bottom). 47

3-13 State diagram showing the summary of the UI design. Users can set tags and close out (suspend) at any time. The red line connects the “Idle state” nodes that represent the same state, duplicated to make the diagram easier to read. ........................................ 50

3-14 State diagram showing the summary of the API design. .......... 50

4-1 Demonstrating AuO’s stance on where curly braces belong. ........ 57
4-2 Example code for building a web UI using the current HTML DOM API. ................................................. 59
4-3 Example pseudocode for building a web UI using the ideal HTML DOM API. ............................................. 60
4-4 Ticker initially positioned at 3.142s (left), moved along with trimmer to 6.283s (middle), and reduced-trimming to 4.712s (right). ............. 66
4-5 The modal for customizing the tag’s label during creation. ............. 77
5-1 JUnit report file naming scheme. .................................................. 91
5-2 An example JUnit report’s filename. ............................................. 91
5-3 Diagram of AuO’s testing framework building on Eclipse Jetty, JUnit, Selenium WebDriver, Google Chrome, and Xvfb. ....................... 92
List of Tables

4.1 A short summary of functionalities, major associated FunctionalElement objects, and major associated functions. . . . . . . . . . . . . . 81

5.1 A tabulation of key client and user testing feedback for AuO. . . . . . 99

6.1 Tabulated summary of key potential future work on AuO, sorted by priority with high priority at the top of the table. . . . . . . . . . . . . . 107
List of Terminologies

The amplitude of a sound wave refers to the maximum absolute sound pressure value of that sound wave.

Audioception refers to the sense of auditory perception, or hearing.

AuO (IPA: /ao/) is the audio recorder and editor produced as part of this thesis. Its name comes from a portmanteau of the words “audio” and “online,” and coincidentally looks like the subscript-less chemical formula for gold oxide, a compound with applications in electronics and optics due to its thermal properties.

Browser-native applications rely only on technologies bundled with web browsers by default. Reliance on external plugins (e.g. Flash or Java applications) or extensions does not satisfy browser nativity.

Channels refer to a single audio wave. The number of channels in a single audio track can vary from one to many more, depending on the input device. Examples of channels include separate recordings for each instrument in an orchestral piece, each voice in a band, or the sole channel in a mono audio recording.

Client refers to another party, such as another application’s programmers, who uses AuO in their application.

Developer refers to someone working specifically on AuO, except in the context of a “web developer.”
Idle state refers to the state where AuO has some audio in memory, while not performing a recording, a playback, a file load, or a file save.

Launch state refers to the state where the client has just launched the AuO instance. Consequently, AuO has no audio in memory, resulting in disabled playback and save buttons.

Opthalmoception refers to the sense of visual perception, or sight.

Oscillograms visualize auditory data by plotting the sound pressure on the vertical axis against time on the horizontal axis.

Tracks contain one or more channels to form a complete audio file. Mono tracks have one channel, stereo tracks have two channels, etc. Examples of tracks include a concert recording of an orchestra, a studio recording for an album, and the audio component of a sound-enabled video.

Trimming in the context of audio data refers to the act of excising some data such that the audio plays seamlessly around that data block, as if that data block did not exist in the original audio.

User refers to another party, such as a teacher, who uses an application built by a client.

Web developers encompass the general category of programmers who build web applications. In AuO’s case, this includes clients and AuO’s developers, but not necessarily users.
Chapter 1

Introduction

Of the five traditional senses, we have the capacity to experience only two through a web browser: opthalmoception (the visual sense) and audioception (the auditory sense). Between these two senses, however, opthalmoception vastly dominated audioception in the history of the Internet. While HTML has incorporated images ever since the 1995 release of HTML 2.0 [5], audio support did not officially exist until HTML5 [10] in 2014, even though support for other multimedia, such as Java Applets, existed in both the 1997 HTML 3.2 [18] specification and the 1999 HTML 4.01 [19] specification. Until HTML5, only proxies, such as the `<object>` element and ActionScript (Flash), allowed audio playback on the web.

Meanwhile, on the desktop, audio editing applications, such as Audacity®, have existed natively for decades. Although several online applications for audio editing already exist, a modular, well-written one built natively for the web has yet to exist. Furthermore, unlike desktop applications, web applications have the capacity to integrate modularly with other applications through the use of APIs. With the recent advancements in standardized web audio APIs and surge of immersive web applications that appeal to both opthalmoception and audioception, the current time presents the perfect opportunity to fill the niche for a modular, efficient, and powerful
native web application for audio recording and editing.

1.1 Motivation

StarLogo Nova, an online ActionScript-based visual programming application, has historically allowed users to incorporate, into their own projects, sounds recorded from their local microphones and edited in the browser. Since the first release of StarLogo Nova in 2011, however, newer web technologies like HTML5 have become the preferred platform for web applications. As part of the transition to HTML5, StarLogo Nova will need to also incorporate the same kind of audio recording features as its ActionScript implementation. By using AuO, StarLogo Nova can easily incorporate the module as the backbone for its audio features in the native web environment.

Projects other than StarLogo Nova can also benefit from AuO. As web applications begin to increase user control and input on aspects like audio and video, there exists an increasing need for a modular online audio recorder and editor using only techniques native to the browser, optimized for the browser. Other similar projects may find themselves needing to incorporate audio as well, such as the browser-based GameBlox, a game editor that also uses a visual programming language, and TaleBlazer, a platform for augmented reality applications. Although each project can build a custom audio recorder and editor, the easier approach results in having all three projects simply integrate with another application to provide audio recording and editing features.

1.2 Constraints Overview

This project inherits many of its physical constraints from its primary integration target, StarLogo Nova. A visual programming platform designed for educational purposes, StarLogo Nova’s current user population primarily comprises students and teachers in secondary schools, whose experiences with audio-related software may
vary from no prior experience to immersively fluent, which StarLogo Nova and anything it uses must accommodate. For the most part, these users access StarLogo Nova on school laptop and desktop machines, primarily using the Google Chrome browser, version 49 and higher. Due to restrictions placed by schools and school districts, these machines typically have restricted access to browsers plug-ins like ActionScript (Flash) and Java, which undermines the availability of applications using these technologies. Furthermore, some machines run on older hardware and bandwidth-limited Internet connections, which restricts the CPU and network resources available to applications.

Other than the constraints inherited from StarLogo Nova’s user population, this project also takes into account the developer body of StarLogo Nova. Built at MIT’s Media Lab in the Scheller Teacher Education Program, StarLogo Nova’s developers predominantly comprise undergraduates with widely-varying programming backgrounds, who work on StarLogo Nova for durations of anywhere from one to three semesters or more. As a result, this project must produce an API that StarLogo Nova’s developers can easily learn, use, and integrate into StarLogo Nova. Furthermore, undergraduate students typically do not take code quality into consideration in their projects, which usually results in brittle code with short lifespans, a problem that this project must counteract in order to produce an application that encompasses both maintainability and longevity.

### 1.2.1 Project Metrics

For the purpose of determining whether the final product of this project satisfies these constraints, I enumerate 38 concrete metrics. Meeting all of these metrics would indicate that the project has met complete success. To help connect the metrics with the broader constraints, each constraint represents a category of metrics.

**Accommodating New Users**

1. The UI self-documents a way to start a new recording.
2. The UI self-documents a way to stop an ongoing recording.
3. The UI self-documents a way to start a new playback.
4. The UI self-documents a way to stop an ongoing playback.
5. The UI self-documents a way to trim an audio track.
6. The UI self-documents a way to save the trimmed audio track.
7. The UI visually notifies users when it has an ongoing recording.
8. The UI visually notifies users when it has an ongoing playback.

**Accommodating Fluent Users**

9. The UI offers a pathway for quickly starting a new recording.
10. The UI offers a pathway for quickly stopping an ongoing recording.
11. The UI offers a pathway for quickly starting a new playback.
12. The UI offers a pathway for quickly stopping an ongoing playback.
13. The UI offers a pathway for quickly saving an audio track.
14. The UI offers pathways for quickly performing other UI actions.
15. The UI offers a pathway for higher-precision trimming.

**Browser Support**

16. Users can record audio on Chrome 49 and higher.
17. Users can play back audio on Chrome 49 and higher.
18. Users can save audio on Chrome 49 and higher.

**Independence from Browser Plug-ins**

19. Users can record audio without the aid of browser plug-ins.
20. Users can play back audio without the aid of browser plug-ins.
21. Users can save audio without the aid of browser plug-ins.

**Preserving CPU Responsiveness**

22. Users perceive a frame rate of 60 frames per second or higher.
23. CPU-intensive subroutines do not cause the page to lose responsiveness.
Conserving Network Resources

24. Aside from loading the library, users can record audio without Internet access.
25. Aside from loading the library, users can play back audio without Internet access.
26. Aside from loading the library, users can save audio without Internet access.
27. Aside from loading the library, other interactions have pathways that do not require Internet access.

Easy for Clients to Integrate

28. The library contains as few files for clients to include as possible.
29. The API does not require clients to set configurations.
30. The API gives clients the option to set configurations.
31. The API gives clients the ability to display the UI.
32. The API gives clients the ability to hide the UI.
33. Clients can show the UI at any time.
34. Clients can hide the UI at any time.

Library Maintainability and Longevity

35. The project adheres to a fixed code style.
36. The project’s code has a deliberate and documented organization pattern.
37. The project’s code has consistent hygiene, style, and organization.
38. The project enforces its code hygiene, style, and organization rules.

1.3 Contributions Overview

In this thesis, I present AuO, an online audio recorder and editor that I developed with the goal of filling the niche for a modular, efficient, and powerful native web audio recording and editing application. Built expressly with the intention of providing backbone support for audio recording and editing in other web applications, AuO
leverages modern browser APIs for capturing and manipulating audio in the web, enabling users to capture and trim audio in web applications without resorting to ActionScript (Flash), Java, or other external-to-browser technologies.

After my completion of this academic program, AuO will remain under development and maintenance at https://github.com/wqian94/AuO. The testing suite will also remain under development and maintenance on GitHub as a public repository at https://github.com/wqian94/AuO-tests.

1.4 Thesis Organization

Chapter 2 describes the background for this project, as well as some previous work in the area of audio recording and editing web applications. Then, Chapter 3 delimits the design considerations and constraints for this project, followed by Chapter 4, which documents the implementation and development process of AuO from the ground up, and Chapter 5 delivers the testing methodology for AuO’s test suite. Afterward, Chapter 6 discusses potential future work on AuO, and finally Chapter 7 details the full contributions of this project and concludes this thesis.
Chapter 2

Background

When video games first incorporated 8-bit music in the 1980s, consumers readily applauded the hit sensation. Ever since, audio has remained an integral part of the gaming experience. Of course, audio has gained importance in other technologies as well. For example, in recent years, audio alerts have become more common in instant messaging services, such as Google Hangouts, Facebook Messenger, Skype, Slack, and many others. Undeniably, we have woven audio intricately with the modern technology experience.

At the same time, all of these applications have one major common point: the server predetermines the audio available to clients. Users download video games prepackaged with all the musics and sounds that the game will ever use, perhaps updated through patches and extra content bundles. Each instant message client provides users with its own list of possible alert sounds from which to choose. At the moment, rarely do we encounter an application that not only allows the user to select an audio file, but also allows the user to generate and edit audio files to integrate into the main application itself.

As a result, while we have seen a steady increase of auditory immersion in web applications, we have yet to see user-provided audio in many of these applications.
In the current day and age, when even personal computers have high-quality built-in audio input devices, web programmers can begin to incorporate user audio input into their applications. With the progressing work on W3C’s Web Audio [1] and MediaStream [6] APIs, the time has come for web applications to leverage the full power of HTML5, and start incorporating user-generated and user-edited audio input through only native-to-browser techniques. This is the vision for AuO.

2.1 Previous Work

2.1.1 Online Audio Recording and Editing Applications

A simple search shows that several applications for online audio recording and editing already exist at this time, many already full-fledged as standalone applications; however, at the moment, each of these applications has some flaw that precludes its usage in applications like StarLogo Nova, as described below.

Many applications for online audio recording and editing, especially ones written several years ago, rely on third-party plug-ins, such as Adobe Flash. Due to the reliance on such third-party software, I do not consider these applications to run natively in the browser. One reason for this relates to mobile devices. With the recent surge in smartphones, mobile browsers have represented an ever-increasing market for online applications, and with their limited hardware resources, mobile browsers oftentimes do not support third-party plug-ins. Similarly, another reason relates to schools. Since projects like StarLogo Nova, GameBlox, and TaleBlazer have students as their target audiences, it is imperative that school computers can run these applications, and since most schools disable third-party plug-ins, these projects and anything they integrate with, such as AuO, must run independently of these plugins. As a result, applications relying on third-party plugins do not satisfy our needs because they do not run natively in web browsers.

Rather, projects like StarLogo Nova need an application that supports both audio
recording and audio editing without resorting to third-party plug-ins, and can easily integrate into these projects. In order to achieve this, such an application must use only HTML5, CSS3, and JavaScript, provide a clean and usable client API for integration, and support user interfaces for both audio recording and audio editing.

At the moment, the application that matches the most of the above-stated criteria among those that I could find is Hya-Wave (wav.hya.io), which provides both functions in a well-designed and easy-to-use interface. Unfortunately, Hya-Wave exists as part of the hya-io suite, and at the moment lacks the ability to integrate with projects like StarLogo Nova.

2.1.2 Native Browser APIs

Browsers already provide tools for audio stream capture through the Web Audio API. In addition, to demonstrate the features of the API, the Mozilla Developer Network has produced a lightweight oscilloscope application for live audio capture through the user’s microphone and visualizing this as an oscillogram [17].

Unfortunately, the oscilloscope demo only works with the live data stream; the demo facilitates neither storage nor editing of the recorded audio. Furthermore, the Web Audio API seems to focus more on audio streams rather than audio storage and playback. The API has a lot of tools for transforming and displaying audio stream data, while providing few tools for editing audio samples, such as trimming the start from audio files.

AuO, however, does provide users with the ability to record and edit audio files while relying on only the Web Audio and MediaStream APIs. AuO also allows users to load audio from and save audio to online servers, backed by the XMLHttpRequest API [14].
Chapter 3

Design

AuO aims to provide an easy, intuitive, and convenient audio recording and editing interface for other applications, such as StarLogo Nova, to use as the backbone for their audio recording and editing features. To this end, AuO’s design process involved the consideration of many factors, which forms the core of AuO’s engineering ideologies in response to the physical constraints for AuO. Additionally, since it drives the primary motivation for AuO, StarLogo Nova also influences the design decisions on what to incorporate now versus later, though AuO also received feature requests from GameBlox.

In terms of platforms, AuO currently primarily supports Chrome 49+ on the desktop. Although AuO also runs in Firefox 45+, Chrome 49+ remains the primary development target. On the other hand, Safari, Opera, Edge, and Internet Explorer lack support for ES6 [24], as well as either the NavigatorUserMedia or MediaDevices [6] APIs at this time, so AuO does not support these browsers. For browsers that do not support ES6, AuO fails ungracefully on load due to AuO’s reliance on ES6 constructs. For browsers that do support ES6 but not the NavigatorUserMedia or MediaDevices APIs, AuO fails only on recording, though future work may provide more graceful failing in this case.
Furthermore, though AuO has some compatibility with mobile browsers, many of the features do not have equivalent mobile support, so AuO does not have good support for mobile browsers. As with desktop browsers, AuO will ungracefully fail on load in mobile browsers that do not support ES6, and will only fail on recording in mobile browsers that do support ES6 and not the NavigatorUserMedia or MediaDevices APIs. For mobile browsers that support both ES6 and either one of the NavigatorUserMedia or MediaDevices APIs, AuO should not produce any errors, as the unavailable features simply do not activate.

The remainder of this chapter focuses on the design process for AuO, touching upon the major design goals and decisions. The first section describes how AuO’s design answers the constraints outlined in Section 1.2 with its own design goals. The second section highlights the high-level design goals for AuO, which gives context to some of AuO’s restrictions and trade-offs. Then, the next sections describe the design improvements in the initial design (v0.1), the minimum viable product (v0.2), the stable release (v1.0), and the thesis release (v1.6). Finally, this chapter closes with remarks about some design considerations that take future work into account.

3.1 High-Level Design Goals

AuO’s high-level design focuses on delivering three key components: an easy-to-use UI for users, an easy-to-use API for clients, and an easy-to-integrate application to pair with other web applications. Each of these has a set of holistic requirements to satisfy.

3.1.1 User Interface

As its name suggests, the UI directly faces the user, and as discussed in Section 1.2, the user may have any amount of technical knowledge with any amount of familiarity with similar tools and applications. Thus, to answer that constraint, AuO’s UI must give new users an intuitive and self-documenting interface, while providing familiar
users with a fast-paced workflow. Such a design ensures that new users can quickly and easily use AuO without having to read documentation or manuals, and at the same time, experienced users can use AuO to accomplish their tasks efficiently and without frustration.

3.1.2 Client API

Since AuO aims to provide a convenient way for other applications to embed user-produced audio, AuO must provide an easy-to-use API for clients, such as StarLogo Nova developers. Since AuO’s clients can vary in programming background and experience, AuO answers the constraint of catering to a large variety of clients by ensuring that the API enables clients to perform the actions they need without making the API overwhelming to use. This means that only the minimal, necessary API functions should exist, while still giving clients customizability where they expect it. Furthermore, the minimum necessary code needed for clients to run AuO should remain as simple as possible. This encourages clients to try out AuO without much hassle, while still allowing clients to customize AuO for their needs after they commit to using AuO.

Until the introduction of the tags UI, described in Subsection 3.5.4, AuO had only three API functions: the constructor, the launch() function, and the suspend() function. The constructor also allows clients to configure an AuO instance by passing in constructor arguments, satisfying metric 30 from Subsection 1.2.1, though without any arguments, the constructor will use default settings, satisfying metric 29. The launch() function displays the AuO instance at any time, satisfying metrics 31 and 33, while suspend() allows clients to, at any time, hide the instance and frees up most of the resources the instance used while running, satisfying metrics 32 and 34.

3.1.3 Integration

As an application aiming to provide the audio recording and editing backbone for other applications, AuO must offer clients, such as StarLogo Nova’s developers, a
hassle-free integration process. For the easiest convenience of incorporating AuO into any project, the entire AuO library should fit into one file for clients to source. This means that AuO cannot depend on third-party plug-ins, from ActionScript and Java, or other external libraries, like jQuery and Bootstrap. Such requirements help minimize the amount of code that clients need to write in order to integrate AuO into their applications.

3.2 Initial Design (v0.1)

AuO version 0.1 comprises the bare minimum code necessary to demonstrate the viability of the application, using the Web Audio [1], NavigatorUserMedia, and MediaRecorder [8] APIs. Thus, although this version of AuO includes interactions like recording and saving online, as well as a basic UI featuring zooming and a waveform display, it does not include audio editing features, which make it insufficient as a minimal viable product. Overall, AuO version 0.1 shows the feasibility of an online audio recorder that does not use ActionScript or other third-party plug-ins, while also prototyping AuO’s first UI and API experiences.

3.2.1 Recording

The recording interaction begins with a user-initiated click on the Record button. This signals for AuO to use the NavigatorUserMedia API to begin capturing the recorded audio and use the Web Audio API to store the audio data into an internal buffer, satisfying metrics 16, 19, and 24. As the recording progresses, users can visually see the captured audio as a waveform in the UI. The user can then stop recording at any time by clicking the Stop button. Currently, if the user initiates a new recording interaction, AuO will discard any previous audio data, e.g. data the user had just recorded, thus overwriting the existing audio data with the new recording and deleting any unsaved data.

The buttons themselves also have deliberate rationales. By clearly labeling the pur-
poses of the buttons on the buttons themselves, the $[\text{Record}]$ button satisfies metric 1 in Subsection 1.2.1, and the $[\text{Stop}]$ button satisfies metric 2.

In AuO v0.1, no post-processing occurs after recording ends. This resulted in a pre-processing step to convert the data into an AudioBuffer that ran before each and every playback and save, increasing the runtime cost of both of these interactions and reducing users’ workflow efficiencies in these steps. In later versions of AuO, once recording ends, AuO post-processes the recorded audio data into a single AudioBuffer, which playback and saving can then use. This means that the runtime cost of converting to an AudioBuffer only occurs once per recording, as opposed to once per playback or save.

Interestingly, due to additional restrictions on the NavigatorUserMedia API in some browsers, recording must occur only on SSL-secured or localhost URLs, so only HTTPS and localhost servers can properly serve AuO-enabled applications. Since most applications that may have uses for AuO will likely have an authentication system, and as a result, will likely all use HTTPS URLs as well, this does produce conundrums later in the audio loading interaction, as described in Subsection 3.5.2.

### 3.2.2 Playback

AuO’s audio playback feature leverages the Web Audio API, which satisfies metrics 17 and 25. To initiate a playback, users can press the $[\text{Play}]$ button. To stop an ongoing playback, users can press the $[\text{Stop}]$ button. As with the $[\text{Record}]$ and $[\text{Stop}]$ buttons for recording, these buttons satisfy metrics 3 and 4, respectively.

The choice of the Web Audio API over the conventional HTML5 audio tag precipitated from inconveniences associated with the latter. In order to produce the data to pass into the audio tag, AuO had to convert the internally-stored audio data into a blob and generate a URL for that blob – a process that also used the Web Audio API to pass the data into a MediaRecorder.
Instead of going all the way to using a MediaRecorder and conjuring the blob, AuO only goes halfway on this process, and directly uses the Web Audio API to play the buffered audio data, satisfying metric 20. By converting an AudioBuffer containing the audio data to an AudioNode and connecting it to the AudioContext’s default destination of the user’s speakers, AuO can easily play a trimmed audio recording by specifying a start time and duration when requesting the AudioNode to play.

Later, when the issue of supporting Chrome 49 arose, removing the playback interaction’s reliance on the MediaRecorder API proved fortuitous, as Chrome 49 does not support the MediaRecorder API. This allowed recording and playback to remain functional, while saving dysfunctioned, leading to additional work on the saving interaction, as described below.

3.2.3 Saving

In version 0.1, users can click the **Save** button to initiate an online save in the WebM format. This uses the MediaRecorder API to play the trimmed audio track into a WebM blob, satisfying metric 21, which then passes through the XMLHttp request API [14] to send the raw audio data to an external server, dictated by a constructor parameter that the client can configure. Unfortunately, the use of the XMLHttpRequest does not satisfy metric 26.

Unfortunately, as alluded to in the previous subsection, this reliance on the MediaRecorder API produced compatibility problems with Chrome 49. Subsection 3.4.2 details how AuO addresses this incompatibility with Chrome 49, and added the ability to save for users in Chrome 49.

As for other encodings, due to restrictions on audio codecs, many common audio formats, such as MP3, require royalties or paid licenses in order to write an encoder for those formats. Since AuO aims to integrate with all kinds of applications including open-source ones, AuO cannot rely on royalty and paid-license technologies, thus precluding AuO’s ability to encode into these formats.
Also, for additional clarity, the [Save] button has a self-documenting label. This means that it satisfies metric 6.

### 3.2.4 Controls UI

![Figure 3-1: AuO v0.1 controls UI, in the “idle state.”](image)

In the early versions of AuO, the controls UI contained the buttons for interacting with AuO’s internally-store audio data. The [Record] button clears out any preexisting buffered data and initiates a record, the [Play] button initiates a playback assuming the preexistence of buffered data, and the [Stop] button ends the active recording or playback. The [Save] button, which initially composed the controls UI alongside the [Record], [Play], and [Stop] buttons, moved into its own save UI starting in version 1.0 and underwent a name change in version 1.6 to distinguish between its online and local functionalities.

The “idle state” occurs when AuO has data in the internal audio buffer but no active interaction, and in this state, users can start any new interaction, including editing. Otherwise, during recording, playback, saving, or loading, users should not have the ability to start a new recording, playback, save, edit, load, or other interactions that involve directly reading from or writing to the internally stored audio data; this prevents race conditions with other writes, and using stale information for the reads.

### 3.2.5 Overlaying

When launched, AuO assumes that users want to see AuO first and foremost. Therefore, AuO creates an overlay that displays over everything else on the page and contains the entire AuO UI. Later, as described in Subsection 3.5.6, AuO’s custom modals use this very same overlaying concept to produce a proper modal effect, above the rest of the AuO UI.

This contributes to the satisfaction of metric 33, since the overlay’s behavior ensures
that, when AuO launches, it will certainly appear to users.

3.2.6 Waveform Displaying

Figure 3-2: A two-channel waveform of Rick Astley’s 1987 song, *Never Gonna Give You Up* [2]. AuO supports multiple channels as of v0.3, and file loading as of v1.4.

The waveform display appears in the middle of the AuO UI, and serves as the primary way for users to visually interpret the audio track. In AuO version 0.1, the waveform display composes the entire audio UI, and contains only the waveform and timestamps. Later versions of AuO incorporate the audio ticker and trimmers as well.

While many audio recording and editing applications use a variable width and fixed time-to-pixel ratio, AuO uses a fixed width and variable time-to-pixel ratio. This means that, as recording progresses and AuO collects audio data, the waveform compresses horizontally to fit more samples in the same amount of pixels. Although I made this design choice with the aim of providing a better user experience by allowing users to easily see the general shape of their audio tracks’ waveforms before zooming in, further user testing will determine users’ opinions on which display style they prefer.

Also, to satisfy metric 7, during a recording, the waveform in the waveform display will constantly update itself with the latest buffered waveform. Thus, when users see the waveform changing, they will know that AuO has an ongoing recording.
3.2.7 Zooming

Zooming allows users to increase and decrease the time-to-pixel ratio through the \textit{Zoom in}, \textit{Zoom out}, and \textit{Zoom reset} buttons. The area next to the zoom buttons displays the current zoom value as a percentage of the base zoom value of 100\%, defined below. Since this works regardless of Internet connectivity, it satisfies metric 27.

Zooming in results in a lower time-to-pixel ratio, producing a magnified view of a smaller section of the audio track. Conversely, zooming out results in a higher time-to-pixel ratio, producing a reduced view of a larger section of the audio track. Resetting the zoom returns the zoom value to the base zoom of 100\%, which corresponds to the most zoomed-out view, where the entire audio track fits exactly in the width of the UI.

As the user increases the zoom value, the display’s width increases and a horizontal scrollbar appears below the display. The user can then scroll left and right to adjust the visible section of the waveform.

3.2.8 Responsiveness

Although not designed for optimal performance in mobile browser, AuO still has a flexible and responsive UI that allows it to display properly in any reasonable browser size. This property allows for future work in adding better mobile compatibility to AuO, and also has applications outside of the mobile world, such as laptops with lower screen resolutions.
3.3 Minimum Viable Product (v0.2)

AuO version 0.2 bundles together the minimum viable product, which has all the features that StarLogo Nova needs, sans Chrome 49 support. In this version, AuO has the ability to trim audio as its sole editing feature, a ticker to indicate the current location of the playback in the audio track, as well as drag-and-drop interactions for repositioning the ticker and trimmers.

3.3.1 Ticker

The ticker, or red bar on the waveform display, shows users the present location of the recording or playback in the context of the entire audio track. In the case of a recording, the ticker displays far to the right, since recording currently only appends audio data to the then-current audio data. During a playback, the ticker moves forward to the right along with the playback through the audio track, which satisfies metric 8.

Figure 3-4: The ticker on the audio track shown in Figure 3-2 when representing a time in the first half of the audio track.

Figure 3-5: The ticker on the audio track shown in Figure 3-2 when representing a time in the second half of the audio track.

As demonstrated in Figure 3-5, in the idle state, users can reposition the ticker
anywhere along between the blue trimming boxes, in the untrimmed part of the audio track. Hovering over the ticker displays the time that the ticker represents, with the label displaying on the left or right of the ticker, depending on whether the ticker represents a time in the first half or the second half of the audio track. Additionally, users can only reposition the ticker in the idle state, and not during a recording or playback, since the former has no meaning and the latter can cause jittery playback behavior.

3.3.2 Trimmed

Version 0.2 introduced the first, and currently only, editing feature in AuO: audio trimming. Trimming allows users to clip either end of their audio tracks, usually to remove leading and trailing noise in recordings.

![Figure 3-6: Start and end trimmers used on the audio track shown in Figure 3-2.](image)

As Figure 3-6 shows, users can drag the start- and end-trimmers to indicate the section of the audio to trim. This results in a semi-transparent blue highlight over the trimmed sections. Neither trimmer can move past the other, and if trimming results in invalidating the ticker’s location, the ticker moves with the trimmer as the trimmer passes through the ticker’s location.

While users can only move the trimmers during the idle state, the trimmers do affect playback and saving, as those interactions include only the untrimmed section. Additionally, users can see the trimming amounts by hovering over the respective trimmer. The start-trimmer displays the trimming amount as an amount of time from the start, while the end-trimmer displays the trimming amount as an amount of time from the
end. Initially, the end-trimmer displayed the offset as a positive value, but after some user testing, users reported that a negative sign adds to the information process, so the end-trimmer’s label adds a negative sign to prefix the trimming amount as of AuO version 0.4.

In terms of metrics, the trimming boxes do not fully satisfy metric 5. Without text, the trimming boxes do not self-document their purpose, though version 1.5 partly alleviates this by giving the trimming boxes hover text that alludes to their purposes.

### 3.3.3 Drag-and-Drop

Users can interact with the ticker and trimmers through HTML5’s drag-and-drop API [9] by dragging the ticker or trimmers and dropping once they have reached the desired positions. Since the time labels appear during a mouse hover, they also appear during the drag-and-drop interaction. In addition, users can also pan left and right by dragging the untrimmed part of the waveform display directly. This mimics touchscreen behavior on phones.

Unfortunately, the drag-and-drop behavior has its own quirks. In AuO version 0.2, the interaction produced poor effects and low fidelity in the accuracy of the drop location. Release version 1.0 improved the fidelity a bit by adding more drop handlers, and release version 1.5 finally closed the fidelity problem by binding drop handlers to almost every single element on the UI.

### 3.4 Stable Release (v1.0)

AuO release 1.0 makes several performance improvements over the minimal viable product, along with adding new features and browser compatibility components. Between versions 0.2 and 1.0, AuO underwent several minor redesigns to reduce runtime performance impact. Additionally, since StarLogo Nova supports Chrome 49+, AuO version 1.0 also includes compatibility features to counteract some missing features in Chrome 49.
3.4.1 Runtime Performance

In the beta pre-releases, users reported two major performance hindrances: recording for longer than 45 seconds, and playback and saving a track longer than 45 seconds. In both these cases, AuO slowly down noticeably and considerably, resulting in these poor user experiences for audio tracks that do not even reach 1 minute in length. Version 1.0 corrected both of these complaints.

Improving Rendering Performance

The first complaint, though it occurs during the recording step, actually had very little to do with the actual recording process. Rather, the waveform display up to that time had rendered each frame by stepping through each audio sample and plotting that information on the HTML5 Canvas [7] display. Unfortunately, at a sampling frequency of 44.1 kHz, this means that at 45 seconds, each frame required stepping through about 2 million samples and rendering each separately, causing noticeable lag.

To address this complaint, I redesigned the waveform display to step not by samples, but by pixels. This meant that the waveform only accessed as many samples as total pixels drawn. Choosing pixels as the granularity allowed for a display that rendered the necessary visual information very quickly. Unfortunately, this did mean that the waveform display lost some accuracy because AuO no longer renders all the samples, but the waveform still produces the general shape of the wave, and users can see the finer features by zooming in on the desired section of the audio.

Improving Buffering Performance

The second complaint resulted from having to convert from the internally-stored array of audio segments format to a single, continuous AudioBuffer. Making this conversion required duplicating the data, and when done once for each playback and save, the few seconds that this process requires adds increasingly noticeable lag to the process.
So, instead of making the conversion for each playback and save, AuO now makes this conversion at the end of each recording. This means that, rather than have a pre-processing step before each playback and save, each recording has a post-processing step. Although this does not eliminate the conversion cost, it does reduce its frequency: instead of incurring the cost on each playback and save, which the user may do many times for a single recording, each recording incurs the cost once, contributing to AuO satisfying metric 23.

3.4.2 WAV Support and Chrome 49 Compatibility

In the early versions of AuO, the MediaRecorder API backed the saving interaction by reading the AudioBuffer into a WebM blob. Unfortunately, this procedure does not work on Chrome 49. Since StarLogo Nova allows clients to use Chrome 49, AuO must also support clients using Chrome 49, which meant that a different flow through the saving mechanism that does not use the MediaRecorder API. Since the AudioBuffer data already exists in IEEE 32-bit floating-point format, this second flow for saving manually writes a WAV audio file according to the IEEE floating-point format [13].

Leaving the details of implementing the WAV audio encoder to Chapter 4, here I discuss the design considerations involved in this change. Externally, this required AuO to give users the option to select the format to encode the audio into for saving, with the WebM option disabled for Chrome 49 users. Internally, AuO now needs to handle multiple pathways through the encoding part of the save procedure. By leaving the other AuO components intact, the addition of the WAV save option affected no other aspects of AuO.

Since inception, AuO has already satisfied metrics 16 and 17, but with the addition of WAV save option, AuO finally satisfies metric 18. Additionally, the WAV save option also satisfies the requirements of metric 21, such that audio saving still preserves its independence from browser plug-ins.
3.4.3 Save UI

Now that users have the ability to choose between multiple MIME types in which to save, the Save button and the type selector no longer belonged in the controls UI. As a result, the Save button moved into its own save UI, adding alongside it the ability for users to select the encoding MIME type. Fortunately, this the FunctionalElement class made this modification rather painless. Figure 3-7 shows the save UI in version 1.0.

![Save file format: WAV](image)

Figure 3-7: Save UI in AuO version 1.0.

Additionally, now that multiple encoding pathways can exist, AuO internally switches on the selected encoding MIME type, creating a different subroutine for each kind of encoding pathway, allowing for the possibility of other pathways to exist as well. Currently, however, still only two save options exist: WAV and WebM.

Furthermore, the save UI now allows clients to specify a callback function to use for when the server has responded with a success message, taking the XMLHttpRequest object used to upload the audio track as the sole parameter. If not specified by the client as the second constructor argument, the default behavior displays the server’s response to the user in a prompt dialog, later replaced by AuO’s custom modals, as described in Subsection 3.5.6.

3.4.4 Multiple Channels

AuO version 1.0 also added support for multi-channel audio tracks. Previously, AuO could only properly handle mono-channel audio tracks, since it discarded channels other than the base channel. In version 1.0, AuO now properly loads and displays multiple channels into the waveform display, as shown in Figure 3-2. Furthermore, the MediaRecorder API already supports encoding into multi-channel WebM files, and the WAV encoder only needs to interleave the channels by sample-frame to support
saving multi-channel audio. With multi-channel audio support in AuO, clients can now record and edit even surround-sound files.

3.4.5 Concurrency Considerations

As AuO’s runtime performance efficiency increased, numerous potential race conditions arose, due to the asynchronous nature of the Web Audio, NavigatorUserMedia, and XMLHttpRequest APIs, despite the fact that JavaScript’s execution engine runs entirely on a single thread. In particular, tests detected two major hotspots for race conditions: the recording subprocess for storing the recorded audio data, and the waveform display update loop.

Unfortunately, JavaScript does not provide strong mutual exclusion objects, and writing such objects could severly undermine many of the runtime performance gains. Instead, AuO relies on weaker mutual exclusion guarantees using flags. This dramatically reduces the chances of malignant race conditions in both cases, while still incurring very little runtime performance impact.

While this technique does still allow for the potential of race conditions, I made note of two major ideas in deciding on this implementation. First, users cannot perceive if a single display frame renders improperly, since the next rendering would overwrite the bad frame at a rate faster than the eye can perceive, which means that a bad frame every once in a while presents no problem, as long as no chains of bad frames occur. Second, in a similar vein, users cannot perceive if a few samples of audio data get mangled, either from a later sample inserted before an earlier one, or even losing samples entirely. Taking advantage of this, as long as AuO does not mangle a significant amount of samples, the inaccuracies resulting from the few race conditions should not result in a noticeable problem.
3.5 Thesis Release (v1.6)

AuO version 1.6 contains the latest release at the time of this thesis. Between version 1.0 and version 1.6, six new features have made their ways into AuO itself, along with tune-ups for existing AuO components.

3.5.1 Local Saving

In addition to uploading audio tracks online to a server, users can also download them to the local filesystem in AuO version 1.6. As with uploading, clients can overwrite the download behavior by passing in a third constructor argument that handles a blob containing the audio data as its parameter; by default, the local saving interaction triggers the browser’s native file download prompt, giving users a familiar and convenient way to download their recorded and edited audio tracks. This feature allows AuO to finally satisfy metric 26.

Additionally, after some feedback from users, the Save button has morphed into the Upload / Download buttons, which appear depending on whether the user has activated the online or local saving interactions, respectively. Since these labels still clearly describe the purpose of the button, they still satisfy metric 6.

3.5.2 Loading

Not only can users save audio tracks in AuO version 1.6, they can also load audio tracks into AuO, from either an online URL or the local filesystem, which satisfies metric 27. As with recording, loading an audio track into AuO will forcibly discard AuO’s version of that audio track, losing any unsaved data.

Subsection 3.2.3 has already pronounced the restrictions on AuO’s ability, or rather inability, to encode into several audio formats, and a similar restriction applies to loading audio data. Fortunately, many containers allow royalty-free decoding, which means that AuO can decode and load many more audio containers than it can encode.
Notably, AuO does have the capability to decode audio data from some video formats, so users can also extract the audio component from multimedia sources.

Users and clients may also encounter a variety of other issues when loading online. AuO fetches audio files to load from the Internet by using the XMLHttpRequest API. Since AuO requires users to access integrated applications from an HTTPS URL, as mentioned in Subsection 3.2.1, the XMLHttpRequest objects must also point to HTTPS links, else they risk encountering browser errors that prevent “mixed-content” due to the insecurity of accessing an HTTP URL from a SSL-secured page. This, however, introduces yet another inconvenience, since most servers ban “cross-origin requests” through HTTPS URLs, which means that a vast majority of servers on a hostname different from the one launching AuO will refuse to serve the HTTPS URL. As a result, online loading remains predominantly restricted to cases where the user can use an HTTPS URL from the same hostname, until when browsers follow Firefox’s example and enable the NavigatorUserMedia or MediaDevices API on non-HTTPS URLs.

Barring the complications associated with online loading, the load UI gives users the ability to also edit tracks from most digital audio containers, as well as recordings that users have previously saved. This allows users to not only live-generate any audio that they may want to record and edit, but also manipulate legally-obtained digital copies of soundtracks from their favorite artists, movies, games, and other multimedia.

Additionally, also following user feedback and advice, the original \texttt{Load} button in the first iteration of the load UI has since transformed into the \texttt{Fetch}/\texttt{Load} buttons, with \texttt{Fetch} referring to loading from a URL and \texttt{Load} referring solely to loading from the local filesystem.
3.5.3 Resources

Added after the development of the load and save UIs, the resources indicator gives users a more obvious and intuitive way to toggle between loading and saving online versus locally. Since the indicator itself does not require Internet access, it satisfies metric 27. The design of the resources indicator gives users both a clear notice of the active mode, as well as the ability to change the mode between interactions, so users can load from the local filesystem, edit, and then upload online, or fetch from a URL, edit, and then download to the local filesystem.

![Figure 3-8: Resources indicator, in online mode (left) and local mode (right).](image)

Figure 3-8: Resources indicator, in online mode (left) and local mode (right).

![Figure 3-9: Load UI, in online mode (left) and local mode (right).](image)

Figure 3-9: Load UI, in online mode (left) and local mode (right).

![Figure 3-10: Save UI, in online mode (left) and local mode (right).](image)

Figure 3-10: Save UI, in online mode (left) and local mode (right).

As shown in Figure 3-8, the indicator highlights the active mode in a green background, while graying out the inactive mode. By clicking on the desired half of the indicator, users can switch between the modes. As the modes change, so do the labels on the load and save UIs’ buttons, as alluded to earlier and seen in Figure 3-9 and Figure 3-10.

3.5.4 Tags

At the request of GameBlox, another client in consideration, AuO also added the tags UI, allowing users to mark and label times by clicking the button. This creates a tag for the time that the ticker represented at the time of the click, producing a modal for users to enter a string label for the tag, with the default string label set as the time of the tag, in seconds relative to the start of the untrimmed
audio track. Since the tags UI does not require Internet access for itself, it satisfies metric 27.

Figure 3-11: Tags UI, with three tags.

As Figure 3-11 shows, AuO automatically sorts the tags by time relative to the untrimmed audio track. Clicking on a tag in the idle state moves the ticker to the time that the tag represents. Holding [Shift] while clicking the tag deletes the tag.

Tagging also introduces a new API function for clients, the getTags() function, to allow clients to retrieve users’ tags on the audio track. This returns a JavaScript associative array (also known as a “map” or “dictionary” in other languages) with the times as keys and the labels as the values. Notably, the times in the associative array reflect the tags’ times relative to the start of the trimmed audio, in integer milliseconds. Any tags that land in a trimmed part of the audio track do not appear in the associative array. Furthermore, AuO makes no sorting or ordering guarantee regarding the returned associative array.

### 3.5.5 Hotkeys

In addition to buttons and drag-and-drop interactions, AuO also provides experienced users with hotkeys to further boost the speed of their workflows and satisfy metrics 9 through 15. The AuO documentation contains the full list of key bindings, also reproduced in Appendix A. Using a white-listing system of capturing only the keybindings in which AuO shows interest, the hotkeys system allows for faster interactions and finer editing without also disabling critical browser shortcuts, like [Ctrl] + [R] for refreshing a page.
3.5.6 Modals

As alluded to in Subsection 3.2.5, since version 1.5, AuO has replaced all system alert, prompt, and confirm dialogs with custom, AuO-styled modal dialogs. One incentive for this change results from browser behavior when showing alert, prompt, and confirm dialogs. In some browsers, the JavaScript execution engine pauses while displaying these dialogs, which can lead to delayed event firing and other problematic JavaScript runtime problems.

Another incentive for this relates to user experience. By using modal dialogs custom to AuO, the user experiences a more consistent UI, rather than a combination of the AuO UI and the potentially-contrasting browser system UI.

The local load and save dialogs remain the browser’s, however, as such dialogs neither pause JavaScript execution, nor do they provide an inconsistent feel to the load and save process. In fact, when asked if an AuO-themed load and save dialog would improve their experiences, most users responded with a preference for the browser’s dialog, citing many factors, including familiarity, convenience due to pre-customized

Figure 3-12: Modals for the default save callback (top) and tag labeling (bottom).
shortcuts, and unfavorable prior experiences with other applications’ dialogs.

3.6 Designing for Future Work

Although AuO has already reached a point where clients can incorporate AuO version 1.6 as their audio recording and editing backbones, development on AuO does not stop here. In the process of building AuO, I also made several considerations for the future growth of this application.

3.6.1 Multiple MIMEs

Originally written to accommodate both the WebM and WAV audio formats, AuO’s MIME handling system for both loading and saving can easily incorporate other MIMEs as well. The load UI has helper functions that dictate what MIME types and file extensions the online loader and the browser’s file selector should accept, while the save UI can add options by simply changing the DOM tree and adding pathways in the saving procedure’s processing subroutine. Future work in allowing AuO to load from and save to more formats can simply make modifications to these areas of the code, resulting in simpler and easier updates.

3.6.2 Default Arguments in Functions

Although standardized in ES6, the ability to assign default values to function arguments, which AuO uses heavily, remains elusive in browsers like Safari, Opera, and Edge. Currently, AuO has no compatibility with these browsers because they do not have support for either the NavigatorUserMedia API or the MediaDevices API. If these browsers do eventually provide support for either API before allowing default arguments in functions, AuO’s built-in workaround for assigning default arguments will allow AuO to run even without support for default arguments.
3.6.3 Multiple Tracks

At the moment, AuO only allows users to have one audio track loaded in memory at a time. Of course, this translates to loading only one file at a time, but loading procedure intentionally assumes that it receives an array of blobs to load and the design makes it easy to translate into a multiple-track loader, as opposed to a single track loader. Once AuO has a defined behavior for handling multiple tracks, this design will allow for easier conversion to handle loading multiple tracks.

3.7 Design Summary

AuO inherits its predominant user pool from StarLogo Nova’s end users, which comprise students and teachers with varying levels of experience using audio recording and editing software. To address this wide range of users, AuO presents an intuitive UI that unfamiliar users can easily learn, while also providing hotkeys to allow seasoned users to quickly accomplish their tasks. Figure 3-13 shows a state diagram of users’ interactions with the UI, except for setting tags and closing the UI, since users can access both of these interactions at any time.

Similarly, AuO’s API must cater to the widely-varying programming backgrounds of StarLogo Nova developers. To accomplish this, AuO presents clients with a limited API, as shown in Figure 3-14. Future work may expand the API functions available.
Figure 3-13: State diagram showing the summary of the UI design. Users can set tags and close out (suspend) at any time. The red line connects the “Idle state” nodes that represent the same state, duplicated to make the diagram easier to read.

Figure 3-14: State diagram showing the summary of the API design.
Chapter 4

Implementation

From the start, AuO’s implementation has taken both the present and the future into consideration. Aside from designing for good user and client experiences, I have also aimed to make AuO’s code easily maintainable and extensible.

This chapter presents an exposition of how I built AuO, keeping in mind the quality of user, client, and developer experiences, as well as the physical constraints as described in Section 1.2. I first describe some macro-level implementation characteristics before I discuss the micro-level implementation details of the AuO library itself. In particular, the micro-level discussion will touch upon AuO’s code quality, FunctionalElement class, displaying feature, drag-and-drop feature, recording feature, playback feature, saving feature, loading feature, tagging feature, hotkeys system, and modals system.

4.1 Implementation Goals

Persistent and ongoing, AuO’s development cycle will eventually include contributions from developers in StarLogo Nova and other projects. Such an ongoing development cycle remains critical for ensuring that AuO endures the test of time. To facilitate only the best developer experiences, though, I have made several conscious implementation decisions, aimed at reducing the chance that AuO will cease to have active developers.
Most notably, I address the issue of code organization, code style, and programming paradigms. Since AuO may have any variety of developers in the future, such as StarLogo Nova’s predominantly-undergraduate developers, AuO must start off early with good programming practices that make the development process smoother for everyone. Such consistent and well-formed programming practices will certainly encourage developers to maintain and update AuO, rather than scrap the application and start from scratch, as often happens in academia and poorly-maintained open-source software.

By laying down the rules for developing on AuO early in the development process, I hope to ensure a long lifespan for AuO, regardless of the prior experiences of AuO’s future developers.

4.2 The AuO Library

In Subsection 3.1.3, I discussed the importance of minimizing the amount of code that clients need to write before they can begin using AuO. Along this vein, I imposed an important restriction on the AuO library itself: the distributed library must contain exactly one file for clients to include in their projects. This guarantees satisfaction of metric 28.

Due to this restriction, the AuO library exists entirely in JavaScript, the only choice out of HTML, CSS, and JavaScript that allows clients to include all three in one language. Using ES6 [24], which includes the Web Audio [1] and NavigatorUserMedia [6] APIs, as its JavaScript standard, AuO builds its HTML5 [10] DOM tree using the DOM API [15] for JavaScript, and produces its CSS using the DOM Level 2 CSSStyleSheet API [23].

4.2.1 JavaScript Classes

Although ES6 [24] provides syntactic sugar for classes, constructors, static functions, and instance functions, AuO currently uses JavaScript’s more traditional function-
style syntax. Originally, this choice resulted from a desire to allow AuO to run on browsers that do support the audio recording and editing APIs that AuO uses, but not the class-style syntax. As no such browsers satisfy this description, however, future work may see AuO move into class-style syntax.

4.3 Standalone AuO

AuO not only supports integration with other applications, it can also act as a standalone application by “integrating” with a plain web page that has no other functionality other than to include and use AuO. The AuO repository provides a simple example of a “standalone” version of AuO, which includes the AuO library and launches an instance of AuO over a button that users can click to re-launch the instance if they ever close out of it. This “standalone” version also produces an example of how clients can use the AuO library.

4.4 Example Server

AuO’s repository also provides a sample PHP script that clients can run on their servers to handle online save interactions in AuO and meet the specifications for the server. This PHP script receives the audio file as binary data, dumps it into a unique file, and returns a link to that file as the server’s response, which satisfies AuO’s requirements for the default server response.

4.5 Code Quality

Based on my prior experience in the software engineering industry, I aimed to ensure some coding conventions in developing AuO and writing the code. These conventions generally fall among three categories: code hygiene, code organization, and code style.
4.5.1 Code Hygiene

AuO’s code hygiene focuses on ensuring that AuO’s library code remains “elegant” and pleasing to developers’ eyes. This allows developers to focus on solving problems and writing substantial code for AuO without getting mired in unreadable code. Additionally, by establishing and following these rules, AuO can satisfy metric 37.

Thus, the first rule in AuO’s code hygiene rulebook restricts all JavaScript and CSS code to lines of 100 characters or fewer, except in situations where line breaks would result in broken code. This restriction allows developers to see the code without having to think about line wrapping and worry about complex statements embedded all into one line. As Subsection 4.5.3 indicates, AuO’s JavaScript variables make a compromise between brevity and self-documentation, resulting in the magic value of 100, reflecting the compromise between the conventional 80-character line for most languages with Java’s 120-character line to allow for self-documenting variable names.

Additionally, statements should use indentation to indicate the statements’ nesting depths. Each indentation should consist of exactly four space characters, for uniformity in displaying the code in text editors. In situations where a non-control (i.e. not an if-, for-, while-, or similar statement) statement requires line continuation, the continued parts of the statement should also experience an increased indent. In cases where a control statement requires line continuation, the continued part of the statement should indent twice, to offset it from the indented code that appears within the block.

AuO’s third code hygiene rule appears commonly and widely in modern software suites: documentation. AuO stipulates that every function must have associated documentation, placed in a block comment right before the function’s definition. In some cases, the function may require very little documentation, and a few-worded sentence may suffice, but in other cases, only an elaborate and detailed documentation serves the purpose. AuO makes no rule about how much documentation each function
must have, and leaves this decision up for developers to apply their own discernments.

### 4.5.2 Code Organization

Along with good code hygiene, AuO also stresses the necessity for well-organized code. Like code hygiene, proper code organization helps developers easily index and find any code of interest, and also satisfies metric 36.

Foremost, AuO’s library consists of two components: the AuO class, a JavaScript class that contains all of the per-instance AuO code, and a global-scope code block that executes once per page load, which contains the CSS declarations and other code that need to execute only once.

Within the AuO class, partitions separate distinct groups of functions, with the partitions containing brief descriptions of the code group. This separates the logic within the thousands of lines of code that compose the AuO class into smaller collations of common-functionality code, allowing developers to easily locate, insert, and edit code. As of AuO version 1.6, the partitions within the AuO class definition follow this order: AuO API functions, runtime code, setup for the Web Audio and MediaStream APIs, partitions for the callbacks of each interaction, constants and helper functions, the FunctionalElement class definition, creating the DOM elements, click handlers, drag-and-drop handlers, other handler, building the DOM tree, and configuring the DOM elements.

### 4.5.3 Code Style

For AuO, as with any other software, good, consistent code style breeds good, consistent code quality. By requiring developers to write high-quality code, AuO motivates future developers to maintain the library, increasing the chance that AuO will experience a prolonged lifespan as it evolves alongside modern technology. Thus, these rules allow AuO to not only satisfy metric 35, but also ensure the library’s longevity.

Chiefly, AuO invests heavily in overall code flexibility and agility. As much as possible,
AuO avoids cumbersome and brittle code, encouraging developers to segment the code into functions that call one another, rather than lumping giant blocks of code together. In particular, as Section 4.6 discusses, AuO makes heavy use of the HTML DOM API for JavaScript. Due to the clumsiness of this API, however, AuO has taken additional measures to elide many of the brittleness relating to HTML DOM, producing more elegant and readable code.

Moreover, as alluded to earlier, AuO’s JavaScript code uses Java-like naming conventions for variables and functions. Constant values have all-uppercase names, with underscores to separate the words, while all other variables use camelCase. In both cases the names add some amount of self-documentation, producing meaningful names, while retaining some brevity in these names, unlike Java’s exceedingly-verbose names.

On the other hand, AuO’s CSS uses hyphenation to delimit words, and with the exception of the base class, all CSS selectors must use only lowercase letters, refraining from using uppercase letters. This restriction mainly arises from competing browser interpretations of CSS: some browsers assume case-sensitive CSS selectors, while others assume case-insensitive selectors. To prevent confusion, then, all selectors use only one case, except the noted exception of the .AuO base class, which has capitalization.

Additionally, AuO encourages developers to use the const keyword to define variables, rather than the var keyword. This prevents accidental reassignment of variables that other areas of the code expect to contain fixed references. Loop variables, should generally also follow this rule, but since some browsers, such as Firefox, do not allow the for (const ...) syntax, thus requiring the for (var ...) syntax instead, followed by a reassignment of the iterating variable to a constant reference for use in the remainder of the loops’ bodies.

The fourth rule stipulates that operators should have one-space paddings on either side. Operators that occur in “open” and “close” pairs, such as brackets, parentheses, and braces, should only have a padding on the “outside” of the pair, i.e. on the left of
the “open” operator and on the right of the “close” operator. Commas and semicolons should have a space only to the right of the operator. Colons should, too, when used in non-ternary situations. Periods should not have padding on either side.

Also, since AuO uses JavaScript as its main language of choice, AuO has an explicit policy regarding semicolon insertions. To make code easier to read and statements easier to distinguish from one another, AuO strongly discourages reliance on semicolon insertion, and requires that every statement end with a deliberate semicolon, indicating the end of that statement. While this introduces more characters into the code and increases the AuO library’s file size, the boost in readability make this requirement well worth the cost.

AuO’s code style also decides on the contentious issue of curly braces. The general rule indicates that, when used for indicating code blocks in a function, loop, or similar situations, the open curly brace belongs on the same line as the start of the statement, while the close curly brace appears on a line separate from the body of the block, and unindented once compared to the body. For if-else statements, the else belongs on the same line as the previous if’s close curly brace. For associative arrays, the curly braces act like square brackets for in-line definitions, or like block statements for multi-line definitions. Figure 4-1 demonstrates the curly brace behavior in AuO.

```
function () {
  if (...) {
    const assocArray1 = {first: 1, second: 2};
  } else {
    const assocArray2 = {
      first: 1,
      second: 2
    };
  }
}
```

Figure 4-1: Demonstrating AuO’s stance on where curly braces belong.
4.5.4 Linter

Although currently nonexistent, an AuO linter will soon make its debut, once the real need arises. Until then, metric 38 remains unsatisfied, and AuO relies on its developers to maintain, in good faith, AuO’s code quality. Developers should encourage discussions and arguments on updating the progressing documentation on AuO’s code quality, but still follow the conventions at the time of development, even if some developers have strong opinions about some commonly-contentious topics, such as where the curly braces belong and tabs versus spaces.

4.6 FunctionalElement Class

As an application with a strong focus on its UI component, AuO’s entire code library makes heavy use of DOM interactions. Unfortunately, using the HTML DOM API for JavaScript entails incredibly cumbersome and brittle code, due to the large volume of function calls to a procedural-style API. This unveiled the need for a piping system, inspired by such a feature in functional programming paradigms. The remainder of this section focuses on deepening the reader’s understanding of the motivation, implementation, and effects of the FunctionalElement class.

4.6.1 Motivation

Similar to many other applications, AuO builds its own UI in JavaScript, using the HTML DOM API, as mentioned in Section 4.2. Unfortunately, the HTML DOM API entails a lot of repetitive code, resulting in cumbersome, verbose, and brittle JavaScript code, as referenced in the introduction to this section. Figure 4-2 presents an example of building a UI directly using the HTML DOM API.

Although many problems with this syntax exist, the most blatant problem highlights the brittleness of using the HTML DOM API: repetitiveness of the DOM element identifier. In this case, the identifier `element` repeats over and over again, and if one wishes to copy these configurations for another DOM element, one must find
const element = document.createElement("div");

// Configuring the element.
element.style.someProperty = somePropertyValue;
... 
element.classList.toggle(someClass, true);
element.innerHTML = someHTML;
element.setAttribute(someAttribute, someAttributeValue);
...

// Building the DOM tree.
parentElement.appendChild(element);
element.appendChild(someChild);
...

Figure 4-2: Example code for building a web UI using the current HTML DOM API.

and replace each instance of the element identifier. If the text of the identifier also appears in part of the configuration, then one would have to manually fix these overcorrections, lengthening the process even more.

This initiated a desire for an HTML DOM API that avoids the repetitiveness of the identifiers. Such an API would allow developers to chain together the configurations, so that even when the identifier changes, a single change can propagate along to the entire chain of configurations. Figure 4-3 presents the code snippet from Figure 4-2, rewritten using the “ideal” HTML DOM API.

This pseudocode laid the foundation for what eventually evolved into the FunctionalElement class.

4.6.2 API Overview

Fundamentally, the FunctionalElement class allows developers to configure and build the DOM tree while using each element’s identifier only once. FunctionalElement achieves this by returning a self-reference at the end of each mutator function. FunctionalElement initially divided the mutator functions into five main categories: inline CSS mutations, CSS class mutations, attribute mutations, event handler mutations,
and DOM tree mutations. Later, after FunctionalElement also allowed multiplicity (Subsection 4.6.8), FunctionalElement also incorporated the sixth type of mutation: multiplicity mutations.

Aside from mutations, FunctionalElement also allows developers to directly access the underlying HTML DOM elements using accessor methods. This allows developers to make use of the HTML DOM API when the FunctionalElement API does not suffice, as in instances where the DOM element has type-specific functionality, e.g. the audio tag with its controls API. The multiplicity-supporting FunctionalElement also allows developers to access the multiplicity count and specific DOM elements.

Finally, the FunctionalElement also includes a simple constructor, which creates the underlying DOM elements.

### 4.6.3 Inline CSS

The FunctionalElement class wraps the `element.style.setProperty` function with the instance function `style(property, value)`, which takes a CSS property name and a value for the property as arguments. Unlike the HTML DOM API syntax
of \texttt{element.style.property = value}, the FunctionalElement’s \texttt{style} function uses CSS-style properties, rather than converting the hyphenated property names into camelCase. This adds the additional perk of developers needing to rely only on one naming scheme for all of AuO’s CSS.

### 4.6.4 CSS Classes

The FunctionalElement class wraps the \texttt{element.classList.toggle} function with the instance function \texttt{class(classname, add)}, which takes a CSS class name and a boolean, indicating whether to apply the class to the element, as arguments. Unlike the all-too-common syntax of parsing and rewriting the \texttt{element.className} string, this function requires neither parsing nor rewriting of the \texttt{className} string. This reduces the risk of buggy behavior due to string parsing errors, and also provides an idempotent function for applying classes to and removing CSS classes from elements, eliding any worries regarding behavior when the same CSS class gets added or deleted multiple times.

### 4.6.5 Attributes

The FunctionalElement class wraps the \texttt{element.setAttribute} function and the \texttt{element.removeAttribute} function with the function \texttt{set(attribute, value)}, an instance function that takes a DOM attribute name and a value for the attribute as arguments. Notably, this function combines two DOM element functions into one, by special-casing the \texttt{null} value as an indication to remove the attribute.

Furthermore, \texttt{set} also can also set the innerHTML of elements, even though the HTML DOM API does not consider innerHTML as a DOM element attribute for purposes of \texttt{setAttribute} and \texttt{removeAttribute}. As a result, the special value \texttt{null} results in an empty string for the element’s innerHTML, rather than deleting the element’s innerHTML, which JavaScript does not support.
4.6.6 Event Handling

The FunctionalElement class wraps the `element.addEventListener` function with the instance function `listen(event, callback)`, which takes an event’s name and the callback function for when that event fires as arguments. Other than shortening the name of the function, providing multiplicity support along with FunctionalElement, and returning a self-reference, this wrapper function does nothing more than `element.addEventListener`. AuO does, however, make heavy use of this function for its many event bindings.

At the moment, FunctionalElement does not have a corresponding instance function for wrapping the `element.removeEventListener` function. Future development work may necessitate such a function, however, and FunctionalElement will then include the wrapper function at such time. Until then, FunctionalElement has no reason to provide such a wrapper function.

4.6.7 DOM Tree

The FunctionalElement class provides four wrappers for building DOM trees. These four functions represent the Cartesian product of the interaction set (add node and remove node) with the relation set (parent to child and child to parent), providing a lot of syntactic sugar flexibility in building and rearranging the DOM tree.

The `append` wrapper function takes a child FunctionalElement object and appends that child to the instance. On the other hand, the `attach` wrapper function takes a parent FunctionalElement object and appends the instance to that parent.

Similarly, `remove` wraps the act of removing a child from the instance, while `detach` wraps the act of removing the instance from its parent.

Simply by converting from using the HTML DOM API to the FunctionalElement class, the code for building AuO’s UI transformed from incredibly cluttered and brittle code to elegant and terser code.
4.6.8 Multiplicity

The FunctionalElement incorporated the concept of multiplicity as a means of working with multiple DOM elements at once. Fueled by an original ambition to abstract out the increasing and decreasing number of HTML5 Canvases [7] in the audio UI display with the number of channels in the audio track, FunctionalElement’s multiplicity abstraction allows for easy shallow cloning of DOM elements and bundling DOM elements together for less repetitive code by using loops inside all of its mutators and accessors, to give developers both the option to work with a specific DOM Element and the option to work with a group of elements while treating them as a single element.

As a direct result of this, FunctionalElement’s constructor takes in a size argument, and its accessors DOM tree mutators all take an index as an argument. The size argument for the constructor indicates how many copies the FunctionalElement should represent at construction. The index arguments indicate which element to target, for situations that require a targeted element, and defaults to the first created element in the bundle of elements. Accessors use the indices to indicate which element to access, while the DOM tree mutators, specifically `append` and `attach`, use the indices to indicate which parent element to choose for appending the children.

Additionally, if resizing produces new elements, these elements get appended to the parent of the first element in the bundle, if the parents happen to differ as a result of manual access and separately appending each element. Furthermore, although developers can technically set the multiplicity to a value less than 1, doing so breaks a FunctionalElement invariant, resulting in unpredictable behavior, such as infinite loops and index out of bounds exceptions. A future improvement may either throw more meaningful errors or forcibly readjust size parameters to fall within the valid space.
4.7 Displaying

AuO’s audio UI updates on an interval, similar to many other JavaScript-based animation loops. For each update call, AuO makes three subroutine calls: one for the waveform display, one for the ticker, and one for the trimmers. Together, these updates form the current audio UI animation trifecta.

4.7.1 Updating the Waveform Display

The main attraction of the audio UI, AuO’s waveform display receives updates by clearing the Canvas and redrawing the waveform for each update frame. To minimize errors resulting from asynchronization, the subroutine begins by taking a snapshot of the current data characteristics, such as number of sample-frames. This information then provides a fixed boundary for how much data the subroutine should consume, to avoid race conditions and stale data errors. Subsection 3.4.5 describe other concurrency considerations that evolved along with AuO.

Plotting the Waveform

AuO plots the waveform directly onto an HTML5 Canvas element, which has a width determined solely by the window width and the zoom ratio. Thus, even as the number of sample-frames changes in the data, the width of the Canvas remains the same, except for when the window size changes or the zoom ratio changes.

In the initial versions of AuO, the subroutine plotted the data on the display by computing from each data point in each channel an \((x, y)\) coordinate pair. The \(x\) value corresponds to the frame number, scaled with respect to the Canvas width over the number of sample-frames. The \(y\) value corresponds to the data values of the audio samples. These values, given as wave pressures, range from \(-1\) to \(1\), so the \(y\) values \((0, \text{height})\) correspond to the wave pressures of \((1, -1)\). This means that a wave pressure of \(1\) appears at the top of the Canvas, while a wave pressure of \(-1\) appears at the bottom of the Canvas, since the Canvases’ origins of \((0, 0)\) correspond
to the top-left corners.

4.7.2 Updating the Ticker

The audio ticker, added as an interactive visual indicator of playback progress through an audio track, also updates through the same loop as the waveform. Each update synchronizes the ticker by moving the ticker to its appropriate location in the audio UI, after computing it in a conversion from time to position by making calls to the `state.audioPlaybackCurrentTime()` function. The rest of AuO’s code makes use of this dependency, and redefines the function to indirectly reposition the ticker.

Additionally, as specified in the design, the ticker also has a label that appears when the user hovers over the ticker. This label appears to the right of the ticker if it represents a time in the first half of the track, and on the left if representing a time in the second half of the track. In other words, the ticker’s label always appears on the “inner-most” side of the ticker. This choice allows users to see the ticker’s label at the ends of the audio track without the trimmers obscuring the user’s view of the label. As with the position of the ticker, the label’s value and position also depend on the return value of the `state.audioPlaybackCurrentTime` function.

During updates, the ticker experiences soft boundaries placed on its position in the horizontal direction. These boundaries, determined by the start- and end-trimming amounts, do not alter the `state.audioPlaybackCurrentTime` function, and as a result, when users trim the audio to include the ticker’s current position in the trimming, the trimmer visually moves to remain within the untrimmed part of the audio, but will move back toward its original location when the user reduces the trimming, provided that no recording, playback, or loading has occurred in the meantime. Figure 4-4 demonstrates this soft boundary using the start-trimmer and ticker. Initially, the ticker rests at 3.142 seconds in the left screenshot. The start-trimmer then moves to 6.283 seconds in the middle screenshot, moving the ticker to 6.283 seconds as well. In the right screenshot, the start-trimmer has retracted to 4.712 seconds, and since
no recording, playback, or loading has occurred, the ticker moves as close as it can to 3.142 seconds, stopping at 4.712 seconds due to the start-trimmer.

Figure 4-4: Ticker initially positioned at 3.142s (left), moved along with trimmer to 6.283s (middle), and reduced-trimming to 4.712s (right).

**Visual Aesthetics**

Users may notice that, when hovering over the ticker, the ticker gains a thin, blue border to indicate that users have hovered over the ticker. This visual feedback, achieved using CSS, actually decreases the height of the ticker to allow room for the top and bottom borders. As a result, the ticker label’s text begins a few pixels offset from the top of the display. While all of this coincidentally provides aesthetically-pleasing effects, the real driving factor underlying these effects has to do with making sure that the ticker’s borders do not alter the height of the ticker’s bounding box.

Additionally, in order to make the ticker’s label display properly when on the left of the ticker, the JavaScript alters the `direction` CSS property for both the ticker and the label. When the label displays on the left, the `direction` property has value `rtl`, and when it displays on the right, the property has value `ltr`. 
4.7.3 Updating the Trimmers

Unlike the ticker, which has many curious behaviors, the start- and end-trimmers each have a value to indicate the trimming amount, as opposed to a function akin to the ticker’s `state.audioPlaybackCurrentTime` function. There do exist very reasonable boundaries for the trimmers, however; the start-trimmer can only trim between time 0 and the end trimmer, and similarly, the end-trimmer can only trim between the start-trimmer and the end of the audio track.

Internally, AuO stores the trimmers’ trimming amounts as non-negative floating-point values. The start-trimming amount represents the offset from the start, and similarly, the end-trimming amount represents the offset from the end. Thus, as the trimming amounts increase, the trimmers move toward each other; as the trimming amounts decrease, the trimmers move away from each other.

4.7.4 Optimizations

As the single most CPU-intensive component of AuO, the waveform display emerged as a hotbed for runtime performance improvements. During the version upgrade from version 0.3 to version 0.4, the waveform display experienced two main optimizations, one minor and one major.

**Minor Optimization: JavaScript Closure**

The first main optimization, though minor, produced noticeable effects in improving AuO’s runtime performance. Inspired by a mention in [3], this optimization cut out many JavaScript closure evaluations in the display update subroutine, resulting in a noticeable 2\% performance boost after around 3.5 seconds of recording, at which point the display subroutine had originally lagged to about 100 milliseconds per iteration. Although a promising result, this remained insufficient for fixing frame skipping after only a few seconds of recording.
Major Optimization: Stepping by Intervals

The major optimization breakthrough resulted in a major change in the rendering of the waveform display. Originally, the waveform display stepped through each sample-frame and drew them on the Canvas, resulting in the display having a resolution much higher than the pixels-per-inch on most modern computer screens. As a result, even though the JavaScript execution engine does all the work of rendering every sample-frame, only a few of them actually display properly on the Canvas.

Thus, to drastically reduce the toll on the JavaScript execution engine, the Canvas rendering process now steps on a pixel-by-pixel basis, drawing only the sample-frame closest to each pixel on the display, and skipping all the other sample-frames. After this optimization, the rendering time for each frame dropped to only 4.9 milliseconds on a 1376-pixel display, which more than satisfies metric 22. Note that the new runtime has units of time over pixels, because the time now depends on the width of the Canvas, not the length of the recording.

4.8 Drag Events

AuO’s audio UI has four draggable components: the ticker, the start trimmer, the end trimmer, and the graph itself. Implemented using the HTML5 Drag-and-Drop API [9], AuO once again makes maverick use of the state variable to instigate drag-and-drop event handling based on the dragged element, as opposed to the drop target.

When a dragstart event fires on a draggable element, the callback function does three important things:

1. It locally caches the current position or offset in a constant,
2. Redefines the state.audioOnDrag function, which fires when dragging over the audio display, and
3. Redefines the state.audioOnDrop function, which fires when the user has released the dragged element.
The elements that allow dragover then fire the `state.audioOnDrag` function, passing in as the argument the position of the dragged element at the time that the event fired. The dragged elements’ `state.audioOnDrag` functions then compute the amount of offset, and reposition themselves accordingly. In order to make the offsets mathematically correct, however, all of the draggable elements and the dragover-enabled elements must use the same reference for computing the position. By restricting the dragover-enabled elements to only the elements that reside in the audio UI, the positions can use the audio UI’s bounding box as reference, leading to normalized position values.

Then, when users have finished dragging an element, the drop event fires on the dropped element, which calls the `state.audioOnDrop()` function. Unusually, in order to ensure that the drop event fires, the drop target must also have a dragover handler to set up some configurations, so all drop targets also have dragover events in order to ensure that the drop events fire.

Initially, only elements in the audio UI had drop event handlers, making them the only drop targets. As a result, the `state.audioOnDrop` function would not fire if users over-zealously dropped an element outside the audio UI. To rectify this in version 1.5, almost all elements in the AuO DOM tree have drop event handlers, making everything in the UI a drop target; only the Canvas elements do not have drop event handlers, but they do have an overlay above them to capture the drop events that would otherwise fall onto the Canvas elements. The subroutine responsible for attaching all the drop handlers do a crawl through AuO’s DOM tree; as a result, the subroutine call must occur after the DOM tree already exists, but still needs to occur at construction time, so the `runtimeAtConstruction` function calls this subroutine to attach all the drop handlers.
4.9 Recording

At the time of this thesis, only the NavigatorUserMedia API enables users to capture audio from their microphones without relying on ActionScript, Java, or other third-party applications. In brief, for the audio capture part of the process, AuO passes in a handler for working with the recorder’s MediaStream using the `getUserMedia` function. This handler then connects the stream to a ScriptProcessorNode and a no-op destination node. The ScriptProcessorNode then captures through the AudioBuffer that it receives, and saves the recorded audio data into the `state.data` array, but only when the `state.recording` flag has toggled on. The `state.data` array then contains bundles of audio data as its elements, with each bundle represented as an array of channels of non-interleaved data. Note that, since the ScriptProcessorNode retrieves the data from an AudioBuffer, it must make a clone of the data from each channel, otherwise it risks making shallow copies of only the references, causing the next iteration through the ScriptProcessorNode to overwrite the previous iteration’s data.

Additionally, starting from AuO version 0.2, after the user has stopped recording, the capture audio data undergoes post-processing to reorganize itself into a a single bundle of channels of data, represented by an AudioBuffer stored as `state.audioBuffer`. This change occurred as a result of noticing that a period of lag accompanied every playback and save, resulting in deteriorated user experience, especially since this lag occurred with each playback. Analysis linked the lag to a part of the code that originally converted the `state.data` array into an AudioBuffer each time a playback or save occurred. By moving this conversion into the post-processing step at the end of recording, the performance hit occurs only once per recording, as opposed to once per playback and save.
4.9.1 NavigatorUserMedia API

The `getUserMedia` function initiates media capture through the user’s microphone. Although the web standard has already defined and begun to recommend developers to use `navigator.MediaDevices.getUserMedia` variant of the `getUserMedia` function, browsers lag behind, with the `navigator.getUserMedia` variant as the only commercially-available form of `getUserMedia`. As such, AuO version 1.6 still uses `navigator.getUserMedia`.

Up through version 1.5, rather than initiating a new stream for each recording interaction, AuO opens a stream at construction time, and keeps the stream open for the duration of the object. This means that the ScriptProcessorNode continually receives data, even when the client has suspended AuO. As such, the ScriptProcessorNode must take up only a minimal amount of resources when the AuO instance has no active recording interaction. To accomplish this, AuO hides all substantial code behind the recording-related flags, so that in the absence of an active recording, the ScriptProcessorNode simply discards the audio stream capture.

Since version 1.6, AuO has moved to initiating a new `getUserMedia` each time users hit the [Record] button, and closing the stream after each recording ends, after user and client feedback. This introduced a few milliseconds of silence at the beginning of each recording, along with a delay at the end of recording, attributed to runtime cycles necessary for properly stopping the recording stream. Although AuO does not address the first problem, since I have deemed the silence to only negligibly impact user experience, AuO does address this second issue by spinning the stopping part off to run asynchronously, so that by the time that the post-processing for the recording has completed, the recording stream has also stopped.

4.9.2 Post-processing into an AudioBuffer

When the user chooses to end a recording, AuO stops storing capture audio data, and begins to rearrange the data from its `state.data` form into an AudioBuffer. First,
AuO computes the total number of sample-frames in the data set and retrieves the audio data properties and hardware specifications for the user’s microphone, such as the length of the recording and the recording frequency, respectively. Finally, AuO caches some of this metadata, passes it into the AudioBuffer’s constructor, and populates the AudioBuffer with the data.

4.10 Playback

The Web Audio API provides the full API support for AuO’s audio playback interaction. By creating a AudioBufferSourceNode using the `createBufferSource` function in the Web Audio API, AuO populates the AudioBuffer that backs the source node with the complete audio track data, stored internally in the `state.audioBuffer` variable. AuO then connects this node to the Web Audio API’s default destination node, generally the speakers, and starts playback on the AudioBufferSourceNode.

Trimming affects playback by altering the start and duration parameters of the function call that starts playback on the AudioBufferSourceNode. The subroutine also redefines the `state.audioPlaybackCurrentTime` function to return the time since the playback began. Due to the asynchronous nature of initiating the playback, however, the time will not match perfectly with the audio, but should sufficiently approximate the time in a visual manner for users.

4.11 Saving

AuO’s saving interaction also leverages both the Web Audio and the MediaRecorder APIs. Currently, AuO supports saving into two different audio formats, WebM and WAV, as well as two destinations, online and the local filesystem.

4.11.1 WebM Saving

Ever since version 0.1, AuO has supported saving the audio track online in the WebM format. To create the WebM audio file, AuO uses the Web Audio API to build
yet another MediaStream circuit, connecting the AudioBufferSourceNode to a MediaRecorder. By playing just the untrimmed part of the audio track and asking the MediaRecorder to produce a WebM file, AuO can retrieve a WebM-encoded audio blob from the MediaRecorder at the end of the encoding subroutine.

4.11.2 WAV Saving

In AuO version 0.5, AuO gained the ability to also encode audio tracks into the WAV format. Although AuO produces the WebM encoding through MediaRecorder, AuO produces the WAV format by manually encoding the trimmed audio track using the 32-bit IEEE-float format. This allowed users on Chrome 49, the minimum Chrome version that AuO supports, to still save their trimmed audio tracks, even though encoding through the MediaRecorder API does not work on Chrome 49.

4.11.3 Online Saving

The online saving option allows users to upload the trimmed audio track to a remote server, whose location the client can specify as a constructor parameter. If the client provides no location, then AuO disables online saving, preventing users from uploading their trimmed audio tracks for saving.

AuO uploads the audio track by sending the encoded blob through an XMLHttpRequest object using the POST method. Upon receiving the server’s response, AuO runs the client’s online save callback function, with the XMLHttpRequest object as the parameter. If the client did not provide a callback function, AuO defaults to displaying a modal that contains the server’s response in a highlighted text field.

Since some browsers, such as Chrome, require users to run `getUserMedia` from an SSL-secured domain or localhost, clients generally must run AuO from an SSL-secured domain, and thus can only upload to an SSL-secured domain, such as an HTTPS URL. Otherwise, the browser will likely throw an error regarding mixed content. At the same time, however, most servers prevent cross-origin requests on their SSL-secured
domains, restricting saves to mostly just the server hosting the AuO instance.

4.11.4 Local Saving

In contrast to the conditionally-available online saving interaction, the local saving interaction will always remain available to users, as it does not inherently depend on the user having an active Internet connection. When the user chooses to save locally, AuO will send the encoded audio blob to the client-specified local save callback function. If the client did not specify one during construction, AuO will default to the default behavior of triggering the browser’s default file download prompt by creating a phantom anchor tag and forcibly firing a click even on the anchor tag.

4.12 Loading

The loading interaction allows users to load an audio track from a file – either locally or through a web URL. Since AuO currently has no concept of multiple tracks, users can only load in one file at a time, with each load overwriting the previously-buffered data. The interaction itself has two distinct components: file retrieval and data retrieval. File retrieval refers to the first process of obtaining the file containing the audio data, and data retrieval refers to actually converting the file’s bytes into the internally-stored AudioBuffer and channel array.

4.12.1 File Retrieval

As with saving, loading also provides users with ability to load from either the local filesystem or a web URL. Based on the active resource selected in the resource UI, AuO’s load UI will choose different methods to prompt the user for the file to load.

For online loading, users enter the URL to the file for AuO to load in the provided text field in the load UI. AuO then uses the XMLHttpRequest API to retrieve the file from the user-provided URL. As with saving, AuO also uses POST requests to load, requiring an HTTPS URL, and as such, remains subject to the same constraints as
described in Subsection 4.11.3

For local loading, users click an active area to open up the browser’s native file selector, implemented by emulating a click on a file-type input element. This does not require data transmission over the Internet, so POST requests and the associated limitations do not apply to local loading.

4.12.2 Data Retrieval

After retrieving the file containing the audio data to load, regardless of the source, AuO then uses the FileReader interface from the File API [20] to retrieve the byte data from the file. AuO then uses the decodeAudioData function to read the byte data into an AudioBuffer, which then gets post-processed to populate the state.data array. This concludes the file loading process, which leaves AuO in the idle state.

4.13 Tagging

The tags UI, developed as a feature request from GameBlox, allows users to cache different times along the current audio track. Users can optionally give the tagged times string names for additional description detail, and can also click on the tags themselves in the tags UI to move the ticker to the time that the tag represents, subject to the boundaries imposed by the trimmers. Clients can also retrieve the most up-to-date tags list by calling the getTags API function, which produces a JavaScript associative array that uses the trimmed times in integer milliseconds as keys and the tag labels as values.

4.13.1 Updating the Tags UI

As with the elements in the audio UI, the labels in the tags UI get updated in an update loop. Called on a fixed interval, this update loop first checks to see if the internal tags list, stored as state.tags, has mutated since the previous update. If so, then it makes the actual animation request for updating the tags UI.
During the tags UI update subroutine, AuO copies the data in `state.tags` to obtain a snapshot of the current tags, then sorts them by time, the same as the sort order for the visible tags in the tags UI. The subroutine then compares the sorted list of current tags with the sorted list of a snapshot of the visible tags, and adds, relabels, and deletes tags as it makes the passes through the two lists. The snapshot of the visible tags has a crucial role in this updating process: it prevents the need to deal with elements re-indexing as a result of elements getting removed from the DOM tree, allowing for more readable code.

**Comparing the Tags**

AuO determines which tags to add, relabel, and delete by using an algorithm akin to a binary merge algorithm [12], as described in 4.13.1.

---

**Algorithm 4.13.1: UPDATE_TAGS()**

```plaintext
visible ← reference to first visible tag
current ← reference to first current tag from state.tags

while neither visible nor current points to the last element
  do
    if current comes before visible
      then
        insert current before visible in DOM
        iterate current
    else if current comes after visible
      then iterate visible
    else
      update visible’s label to current’s label
      iterate current and visible
  if visible has more elements
    then delete remaining elements of visible from the tags UI
  if current has more elements
    then append remaining elements of current to the tags UI
```

Using this algorithm, the final view of the tags UI will exclude any tags that the user
has deleted, update any changed labels, and include any new tags that the user has added since the previous UI update.

**Maintaining State**

At the moment, tags have the time values that they represent stored as the value attribute on the DOM element itself. While this suffices for current needs, it does expose the live state of the tags to the users and clients, who can then manipulate the values in the attribute fields of the tags, potentially breaking sorting invariants in the tags UI.

### 4.13.2 User Interactions

Users can interact with the tags UI in three ways: create a tag, select a tag, and delete a tag.

Creation causes the ticker’s current location to get tagged, with a modal prompting the user for a custom string label for the tag, with the time as the default label string; if the tag already exists, then the string label will update, though the default label string will still reflect the time, as opposed to the old string label. Figure 4-5 shows an example of the modal that appears during tag creation.

![Figure 4-5: The modal for customizing the tag’s label during creation.](image)

Selection requires the user to click on a tag, and will move the ticker to the time represented by that tag, subject to the constraints of the trimmers. This allows users to use the tags as markers for returning to certain points in the audio tracks.

Deletion occurs when the user holds the Shift key while clicking on the tag. Deleting
the tag will cause it to disappear from the UI without moving the ticker, and also
removes it from the associated array of tags that clients receive when calling getTags.

4.13.3 Client Interactions and the API

When the client calls getTags, AuO iterates through every tag in state.tags, skipping any
tags that lie on a trimmer part of the audio track. While iterating through the
tags, AuO copies the tags in the untrimmed part of the audio track into another
associative array. Unlike in state.tags, where the time keys occur as floating-point
seconds relative to the start of the original audio track, the new time keys occur
as integer-value milliseconds relative to the trimmed start of the audio track. This
means that, after a user has saved an audio track, the tags returned by getTags will
have keys that correspond to the times in the saved track, in milliseconds.

Clients will also note that, unlike many of the other state variables, the tags associa-
tive array does not get wiped during a suspension. This means that, even after users
have suspended an AuO instance, clients can still retrieve the tags from the instance,
until the instance re-launches.

4.14 Hotkeys

Added as a feature to improve user productivity while using AuO, the hotkeys system
allows for rapid and fast-paced interactions with AuO’s main controls. Readers can
see the full hotkey listing in AuO’s documentation, also reproduced in Appendix A
on the eighth page.

The hotkeys feature relies on key event bindings to the overlay that acts as the
outermost container for AuO. Notably, this layer also has the base .AuO CSS class.
By attaching the hotkey handlers here, AuO can ensure that the handlers will capture
all of the key events.

Additionally, all the hotkey handlers use the same function. This function, which
takes a KeyboardEvent object, uses a very long switch-case statement to determine what action to take, given the keystrokes recorded in the KeyboardEvent object. Notably, instead of stopping all KeyboardEvent objects from propagating, AuO only prevents the AuO hotkeys from propagating, resembling a whitelist for the hotkeys, rather than resorting to a blacklist technique to avoid capturing key combinations that AuO should not capture, such as \texttt{Ctrl} + \texttt{R}. AuO also prevents key capture when the browser has focused on an input element, to avoid capturing key strokes when typing into input fields.

### 4.15 Modals

Designed to replace browsers’ native alert, prompt, and confirm dialog boxes, AuO’s custom modals allow message reporting to users without pausing JavaScript execution, which the native alert, prompt, and confirm dialog boxes may do, depending on the browser. By recycling the styles for the containers around the main UI, AuO creates AuO-themed modals by creating a DOM tree that has the same structure as the main AuO application for the first few DOM elements. The subroutine that handles creating the modals then injects an inner DOM tree, passed in as a parameter, to complete the modal.

Since the modal creation subroutine currently lives only in the scope of AuO’s instance code, it currently expects developers to provide a means within the inner DOM tree for closing the modal – the modal creation function does not directly insert a way for users to close the modal. If this modal system should also allow clients to access it, then this behavior should change to always give users a way of closing the modal, to avoid accidentally leaving users with a modal dialog that they cannot close.

Additionally, as a special property of the modal dialog, when the modal loses focus, such as when an ongoing playback ends and focus returns to the underlying AuO instance while interacting with the tag modal, the modal automatically recovers focus and places it on the first input or button element in the modal’s inner DOM tree. This
prevents users from accidentally activating hotkeys while interacting with a modal.

4.16 Implementation Summary

Code quality, a cornerstone of AuO’s implementation ideologies, has three core components: code hygiene, code organization, and code style. Together, these three ideas build the foundation for ensuring AuO’s lifespan and future maintenance. I leave AuO’s code hygiene and code style at the mercy of AuO’s future developers along with suggestions, but as for code organization, Table 4.1 summarizes the key players in making AuO work internally. Organized by functionality, this table presents a quick guide to help developers understand where changes could happen, and where their ramifications might manifest.
<table>
<thead>
<tr>
<th>Functionality</th>
<th>Associated Elements</th>
<th>Associated Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hotkeys</td>
<td>container</td>
<td>hotkeyHandler</td>
</tr>
<tr>
<td>Loading</td>
<td>buttonLoad, loadFile, loadFileLabel, loadUI</td>
<td>beginAudioLoad, loadArrayOfBlobs, loadFileHandler</td>
</tr>
<tr>
<td>Playback</td>
<td>buttonPlay, buttonStop, state</td>
<td>beginAudioPlayback</td>
</tr>
<tr>
<td>Recording</td>
<td>buttonRecord, buttonStop, state</td>
<td>beginAudioRecording, endAudioRecording, processAudioRecording</td>
</tr>
<tr>
<td>Saving</td>
<td>buttonSave, saveOptions, saveOptionsLabel, saveUI</td>
<td>beginAudioSave, LOCAL_SAVE_CALLBACK, SAVE_CALLBACK, SAVE_URL</td>
</tr>
<tr>
<td>Tagging</td>
<td>buttonTag, tagList, tagUI</td>
<td>beginTagUpdateLoop, tagClickHandler, tagCreateLabel, tagLabelComparator, tagsUpdateUI</td>
</tr>
<tr>
<td>Ticker</td>
<td>audioTicker, audioTickerLabel</td>
<td>addAudioDropHandlers, animateAudioTicker, beginAudioDisplayLoop</td>
</tr>
<tr>
<td>Waveform Display</td>
<td>audioDisplay, audioDisplayContainer, audioUI, audioVisualizer</td>
<td>beginAudioDisplayLoop, animateAudioDisplay, animateAudioDisplayByForce</td>
</tr>
<tr>
<td>Zooming</td>
<td>buttonZoomIn, buttonZoomOut, buttonZoomReset, zoomDisplay, zoomUI</td>
<td>zoomFactor, zoomUpdate</td>
</tr>
</tbody>
</table>

Table 4.1: A short summary of functionalities, major associated FunctionalElement objects, and major associated functions.
Chapter 5

Testing

As AuO grew, two kinds of testing aided in directing the development process: automated testing, and client and user testing. Both kinds of testing help ensure that development changes to AuO do not break the existing build, as commonly happens when multiple people work on the same code repository. Especially since young programmers, who have only just started programming, have limited experience with thinking broadly about the effects of their code, automated tests help capture the obvious undesired side effects, while client and user testing help direct the direction of AuO’s development. Even for seasoned engineers, both forms of testing help ensure that AuO’s released code remains healthy and AuO’s UI and API gravitate toward providing better client and user experiences.

Broadly speaking, AuO’s automated tests ensure that, as AuO evolves, continued development does not accidentally break pre-existing work; this ensures that users and clients will not see unintended changes in the UI and API, respectively. More globally, AuO’s client and user tests provide bug reports, as well as feedback for pinpointing where clients and users would like to see improvement or changes; this helps direct and prioritize what to implement and change in AuO. The remainder of this chapter will describe the automated testing process and elaborate on some key
feedback from client and user testing.

5.1 Automated Testing

Bundled in its own, separate repository, AuO’s automated test suites, written in Java 8, use Eclipse Jetty 9.3 [16] and Selenium WebDriver [11] with JUnit 4.12 [21], with an optional external dependency on the Xvfb [22] binary, for running browsers in a headless virtual screen to avoid displaying browser windows on tests’ actual screens. Additionally, the automated test suites require access to a binary of each browser type used in the test suites, thus imposing a further dependency on the Google Chrome binary, due to AuO inheriting Google Chrome 49+ as the main target platform from StarLogo Nova.

5.1.1 Environment

I developed the automated tests suites almost entirely using the Eclipse IDE, with some assistance from Maven for managing dependencies. The automated testing repository contains pom.xml file, used by Maven for downloading and building, so that other testers can easily set up and run the automated test suites. The choice of Java 8 for a language comes from its lambda function syntax, which makes some testing code easier to read and develop.

5.1.2 Code Style

As the code for the automated test suites use Java 8, AuO’s default code style adheres to general Java code style. The Eclipse code style settings files exist in the .settings subdirectory of the repository.

5.1.3 Code Organization

In the context of the AuO repository, the automated test suites exist as a git submodule. This gives AuO developers easy access to the automated test suites. Additionally,
this provided an expected directory structure, such that, relative to the root of the automated testing suite, the AuO library exists at ../lib/AuO.js.

The automated testing suite itself has a three-package framework: one part tests and two parts setup. One setup package provides a Java implementation of a Jetty web server wrapper for serving AuO and its save functionalities, and the other setup package provides utility functions to simplify and streamline the process of setting up a browser window for Selenium WebDriver to use. The tests package, then, provides an abstract JUnit class, which the various concrete test classes can extend, to work with some basic configuration protocols in a non-repetitive manner. Additionally, the tests package also contains all the concrete tests.

5.1.4 Server Package

Effective testing relies on starting from a standardized baseline and checking for expected observational behavior between actions. The automated test suites defines the standardized baseline through the server package. Using a virtual Java Jetty server to handle connections and serve web pages, the server package's AuoServer class acts as any other web server, with the exception of four special links, three of which serve AuO-specific purposes.

The first AuO-specific link, accessed at /TEST, delivers a web page that loads the AuO library and initiates a new AuO instance. This provides the standardized baseline web page with which tests can expect to start. Notably, the server generates the HTML for this page through a hard-coded string.

The second AuO-specific link, accessed case-insensitively at /AuO.js, delivers the AuO library. Although the testing suite has an expected relative path to the AuO library, the server itself actually does not hard-code this path into the server itself. Instead, the server takes the path prefix as a constructor argument, which the abstract JUnit class provides as the hard-coded relative path.
The third AuO-specific link, accessed at /TEST-SAVE, emulates a server for handling an upload request. Unlike deployed instances of AuO, however, the automated testing suite uses localhost to serve AuO, which eliminates the need for generating SSL certificates for the server, since getUserMedia will run on a non-SSL localhost server. While this means that the automated tests will not capture any errors relating to deploying over SSL-secured servers, developers can easily manually test for behavior on an SSL-secured server, to ensure that the application will launch properly.

One final special link, accessed at /stop, will terminate the Java server. Although the automated testing suite does not make use of this link, it exists primarily to conform to Jetty’s default behavior using that link and shut down additional resources used for the server.

Manual Testing

Since AuOServer runs as a general-purpose server with modifications to accommodate AuO’s needs, it can also serve the AuO library for the purpose of manual testing. The AuOServer class contains a main entry point for this very purpose, and the special URLs will work per usual. AuOServer can also serve other files, relative to the ../lib/ directory, though using AuOServer to serve a real web site may introduce security issues.

Ports

AuOServer works by using Jetty to connect to a port on localhost, and then serving as the gateway for traffic on that port. AuOServer’s constructor takes the relative path to the server’s root directory and a port as parameters, with the constructor throwing a RuntimeException if another application has already bound the port. The magic port value of 0 indicating that Jetty should randomly select an unbound port during construction.

As a note about selecting ports, most operating systems have a block of “reserved” ports, such as 443 for SSL-secured (HTTPS) traffic. Intentionally trying to bind to
one of these ports will likely result in an error, though Jetty will never randomly select one of these ports. Thus, unless there exists a real rationale for selecting a specific port, I would generally recommend letting Jetty randomly select an unbound port, rather than specifying a port.

Caching

As many already know, file IO operations can easily bottleneck otherwise-fast routines. AuoServer already minimizes the file IO necessary to just the AuO.js library file by electing to hard-code the /TEST and /TEST-SAVE pages; however, this still meant that AuO does a fresh file read operation for each /AuO.js request. Unlike the other two special links, AuoServer cannot simply hard-code the library JavaScript file, as such a hard-coding would result in inefficient updates to the testing framework, fraught with opportunities for error.

Instead, AuoServer employs a simple web cache solely for the purpose of serving the AuO.js library file. On the first request, AuoServer will cold-fetch the file as a read from memory, cache it in memory as a string, and keep track of the file’s last-modified time. On subsequent requests for the file, AuoServer will check the last-modified time of the file in the filesystem. If the file has not experienced any modifications since the caching, then AuoServer will directly use the cached copy of the file, thus avoiding a file read operation. Otherwise, AuoServer will discard the existing cache as dirty, and reread the file from the filesystem, once again caching it in memory and rerecording the last-modified time. In general, assuming that the AuO.js library file does not change during one run of the tests, the server will need to read only once from the filesystem, greatly reducing the time needed to run all the tests in the automated test suites.

Logging

In keeping with the tradition of web servers printing informative output to the standard output and errors to the standard error streams, the server package also provides
the Log class, which the automated testing suite uses for general logging purposes, from standard output to errors. In particular, the Log class provides static functions that add additional tagging and timestamping to log messages using formatted strings. As a result, the Log class has versatile and easily-extensible behavior, making it extremely well-suited for an evolving software package such as AuO’s automated testing suite.

5.1.5 Client Package

The client package provides the Web class, which handles all of the client-side setup necessary to start a virtual screen environment using Xvfb, emulate the web browser, select elements from the page, and test for conditions, as well as perform resource clean-up to deallocate the virtual screen and emulated browser windows.

Notably, the Web class abstracts out the process of spawning a new browser window, which differs from browser to browser through the getDriver function, which takes an enumerated Browser object as its sole parameter. Currently, the Web class only handles Google Chrome, with the selection of browsers limited by the available enumerated values of Browser. The addition of Firefox and other browsers, however, should not present any problem from an API perspective, as the Web class only uses the WebDriver class’s API functions, except for when initializing the WebDriver instance itself.

Regarding its usage in the automated tests, the Web class has four main API functions, which act as macros for some common use cases of the WebDriver API. Two of these functions, cssSelect and cssSelects, simplify the WebDriver API calls for selecting elements using CSS selectors. Since the AuO UI avoids identifier confusion resulting from multiple instances of AuO by using CSS classes to distinguish between elements instead of the id attribute, these macros considerably simplify the code.

The other two functions, called test and wait, set up a timeout for a chunk of code to run, and in test’s case, return true. If the timeout occurs before the internal code
has run to completion or the test returns false, then the macro fails the assertion. Otherwise, the macro simply finishes executing without raising any exceptions, and the caller’s code continues. These pair of macros hide away a lot of the framework setup necessary, and simplify the API call to just a driver and a lambda function.

5.1.6 Tests Package

Containing all the JUnit tests, the tests package itself has yet one more core class to further simplify the test-writing process. The AbstractAuOTest class serves as the superclass for all of AuO’s automated test suites and unifies together the client and server packages for use in the test suites. Notably, AbstractAuOTest defines the setUpClass, tearDownClass, and tearDown functions.

In setUpClass, which runs at the beginning of each test suite, AbstractAuOTest initializes the instance of AuoServer that will serve pages for the test suites and calls the Web class’s initiate. By initializing the AuoServer, AbstractAuOTest centrally defines the relative path to AuO.js and the port to use. By initiating the Web class, AbstractAuOTest centrally defines the browser that the test suites will use and allocates the virtual screen in which the browsers will run. Centralizing all of these variables in the test suites enable testers to easily change the testing environment without tediously changing the setup definitions for each test suite separately. As the test suites grow, this feature will greatly enhance the speed of the testing cycle.

In tearDownClass, which runs at the end of each test suite, AbstractAuOTest terminates the AuoServer instance and the Web class. This unbinds the AuoServer’s port, ends the virtual screen session, and deallocates the memory and resources used by the test suite. After this teardown process, another test suite can start again without any problems, as currently the Web class has hard-coded in a display port for the virtual display, preventing multiple test suites from running at the same time without changing the hard-coded display port for each additional concurrent test suite.

In tearDown, which runs at the end of each test case, AbstractAuOServer terminates
all browser windows used in the test case, leaving the next test case with a clutter-free environment in which to run. This allows test cases to run mostly independently of each other, without major interference, sans the caching behavior described in Subsection 5.1.4.

Test Suites

The automated test suites divide the test cases by sub-UIs. Each sub-UI, such as the controls and audio UIs, owns its own main functionality, and all tests focusing on components of that sub-UI fall within its specified test suite. The main UI, which contains all the sub-UIs, exists as a special exception, since it contains any tests that do not fit under any sub-UI, such as the sub-UIs' layout pattern.

As with many other applications that exist as syntheses of several modules, AuO has both unit and integration tests. For AuO, both kinds of tests can exist in a single sub-UI's test suite. Unit tests generally fall easily into a sub-UI suite, while integration tests often end up in the sub-UI suite for the sub-UI that the integration test correlates to the most. Since integration tests currently live amongst unit tests, they can oftentimes feel out of place, and perhaps future work on the test suites will further differentiate the unit and integration tests.

5.1.7 Limitations

While AuO sports an extremely robust and flexible test framework, there do exist limitations on the test content. Notably, although Selenium WebDriver’s HtmlDriver variant of WebDriver supports drag-and-drop behavior, ChromeDriver, the Google Chrome variant of WebDriver, does not. As a result, drag-and-drop tests consistently fail for Google Chrome. Thus, though tests for drag-and-drop behavior do exist in the test suites, they largely have \@Ignore decorators, which means that they do not actually run during actual testing.

Another limitation prevents tests aimed at mobile operating systems. At this time,
Selenium WebDriver provides no support for emulating mobile browsers, and has made no indication that there exists any development in that direction. This means that, if and when AuO begins to support mobile browsers, developers will likely need to use a different framework for testing on mobile platforms.

5.1.8 Validation Procedure

Each major release and some minor AuO releases also has accompanying releases for the test suites, along with an Eclipse-generated JUnit report. The reports’ naming convention follows the scheme in Figure 5-1.

(suites)-(browser)-AuO(AuO version)-tests(test suites version).xml

Figure 5-1: JUnit report file naming scheme.

All fields use lowercase lettering, except for AuO’s name. The suites field contains a comma-separated list of the test suites included, with any spaces replaced by underscores. If the report includes all test suites, then the field contains the special value all_suites. The browser field contains the name of the browser, followed by the major version number. If the browser’s minor versions delineate significant differences between the major versions, then the field can include the minor version number as well. The AuO version and test suites version fields both follow the major-minor format, with a prefixed ‘v’ before the major version number.

Figure 5-2 provides an example of a report’s filename, following the naming scheme. This report ran through all the test suites, on Chrome 51, for AuO version 1.0 and test suites version 1.0.

all_suites-chrome51-AuOv1.0-tests1v1.0.xml

Figure 5-2: An example JUnit report’s filename.

5.1.9 Automated Testing Summary

AuO’s automated tests help catch systematic bugs, such as changes to the UI, behavior, or similar procedural aspects. Each sub-UI contains the tests associated with
it, and the main UI contains the broader UI tests that do not fit into any sub-UI. Figure 5-3 depicts the hierarchy of different technologies on which AuO’s automated testing suites build.

![Diagram of AuO's testing framework](image)

**Figure 5-3: Diagram of AuO’s testing framework building on Eclipse Jetty, JUnit, Selenium WebDriver, Google Chrome, and Xvfb.**

### 5.2 Client and User Testing

Although automated testing suffices for basic validation purposes, AuO relies on client and user testing to cover for cases that automated testing does not. User experience, for example, tests best as user tests, since automated tests would have a much harder time ensuring that a UI provides good user experience. Additionally, as mentioned in the previous section, Selenium currently cannot emulate drag-and-drop events in ChromeDriver, so user and manual testing provide the only feedback regarding drag-and-drop events.

As another crucial function of user testing, AuO’s development priorities hinged on what features clients and users desired the most. Given AuO’s nascence, the direction
that clients and users provided helped to ensure that AuO’s earliest features have relevance and use in AuO’s first integrations.

5.2.1 Key Contributors

StarLogo Nova, GameBlox, and MIT’s Teaching Systems Lab (TSL) provided the bulk of the feedback as AuO’s earliest clients and users. StarLogo Nova, as AuO’s primary client, gave the most feedback with regard to AuO’s direction and earliest features, and also provided the fullest feedback with regard to the process of integrating AuO into StarLogo Nova.

5.2.2 WAV Support

In AuO version 0.4, the TSL noticed one major problem with AuO: the save feature does not work on Chrome 49. At this point, AuO only supported encoding into the WebM format by using the MediaRecorder API [8]. After some investigation, the problem of Chrome 49 not supporting the MediaRecorder API arose, and the issue did not appear earlier because AuO’s early development used only Chrome 50 and Chrome 51. With input from StarLogo Nova, AuO resolved this problem by building in a save path that does not use the MediaRecorder API: encoding as WAVs. By manually encoding the captured audio data into IEEE 32-bit floating point WAV format, AuO version 0.5 added a new pathway for saving, thereby allowing users on Chrome 49 to use AuO.

5.2.3 Drag-and-Drop

Ever since AuO first introduced drag-and-drop behavior into its UI in version 0.2, GameBlox had noticed that the behavior had various visual bugs associated with it. Over the course of the next eight minor version updates, AuO slowly addressed the visual errors until version 1.5 finally produced smooth and consistent drag-and-drop behaviors for all the UI components that relied on drag-and-drop, such as the audio ticker and trimmers.
5.2.4 Tags UI

AuO version 1.5 introduced the tags UI, a new feature that allows users to mark certain times in the audio track with string labels and clients to retrieve these times and labels for annotation purposes. Suggested by GameBlox as a feature request, the tags UI underwent several revisions before its final release in AuO version 1.5, with each revision motivated by feedback from the GameBlox team. The highlights of these changes include:

- Changing tag labels from the time to a user-defined string,
- Sorting the tags by time instead of by label,
- Adding an API function to retrieve the time-to-label associations, and
- Re-normalizing the time values in the retrieved associations to reflect the times in the trimmed audio track, as opposed to the original audio track.

Most notably, for the last item above, the initial design for the tags UI gave clients times relative to the original audio track, as if no start- or end-trimming existed. GameBlox noted that this made the times absolutely unusable, as clients have no way of determining the trimming amounts for the saved audio tracks, and thus cannot properly associate the labels with the correct times. GameBlox then gave the suggestion of producing times that take the trimming amounts into account. This way, clients can retrieve the times and labels at the same time that the audio track gets saved to obtain accurate times that do not require any additional offsetting on the client’s part.

5.2.5 MIME Support

At the same time that GameBlox made the feature request for the tags UI, GameBlox also suggested adding support for at least loading M4A-format audio files, since GameBlox’s audio tracks existed as M4A files. This led to the expansion of loadable MIMEs in AuO version 1.5, to include M4A files as well as many other formats. Although the list of loadable MIME types expanded, the list of saveable MIME types
did not, as many of the presently-unsupported encodings have royalty restrictions, while the others would require manual encoder implementations, similar to how AuO supports saving as WAV files. Due to the undesirability of royalties in the first case and the complexity of implementing the encoders in the second case, AuO did not expand its list of encodable MIME types in version 1.5.

5.2.6 Integrating with StarLogo Nova

As AuO’s first major test of integrability, StarLogo Nova’s process of integrating with AuO provided the first feedback on the matter. For the most part, StarLogo Nova reported a smooth integration process, with no comments on the library inclusion and API usage procedures. StarLogo Nova did, however, report that launching and suspending did a bit more than what StarLogo Nova expected. Instead of deallocating resources, StarLogo Nova wanted to have the ability to only hide and show the UI, which StarLogo Nova has currently implemented by directly modifying the display style of AuO’s container element. Additionally, while StarLogo Nova does indeed upload to a server, it would like to do so on its own terms, and thus does not make use of the online saving feature, preferring to obtain the blob directly from the local saving callback instead. These comments have resulted in planned future work in analyzing and potentially implementing changes to reflect this feedback.

Activating NavigatorUserMedia

StarLogo Nova noted that, as soon as AuO loads, the microphone stream opens. This results in browsers showing users that the site has an active recording channel open, even in the absence of an ongoing audio capture. Deemed too “creepy,” StarLogo Nova asked for the microphone recording indicator to appear only when the user actively records. Implemented in AuO version 1.6, this moved the process of activating and deactivating the NavigatorUserMedia API to activating when recording begins, and deactivating when recording ends.
5.2.7 TSL Feedback

A second test of integrability, MIT’s TSL also provided feedback regarding AuO as an application. Furthermore, due to the fact that the TSL has already deployed user tests that involve using AuO, the TSL also provided user feedback on top of client feedback. Some of the noticeable concerns reflect the comments from StarLogo Nova, while others come afresh from applications that did not have a direct influence on AuO’s development.

Client Feedback

One major piece of feedback from the TSL addresses AuO’s entire workflow. Together, AuO flows quite well, but the TSL noticed that AuO can also “unbundle” into the recording, playback, etc. components, such that clients can selectively use only parts of AuO. The appeal of such a system comes from the greater ability for clients to simply embed bits and pieces of AuO into the larger product as a whole. Interestingly, this idea of stitching together multiple pieces to work in collaboration also permeates throughout AuO’s implementation itself, though AuO’s components remain cohesive to clients externally. This feedback from the TSL suggests that perhaps AuO really should open up the stitching to the clients, as opposed to hiding it all away under AuO’s implementation details.

Also, the TSL has indicated that, rather than relying on AuO’s UI directly, the TSL would like to use its own UI to call AuO’s different functions. Currently, the TSL has emulated this by hiding AuO’s HTML and simulating button clicks, though the TSL has also stated that an API that allows clients to make these calls directly without needing the show an API would prove beneficial for use cases similar to the TSL’s.

Additionally, the TSL commented that, while the bundled server-side PHP script seems helpful, few clients would choose to use it. As a result, it seems extraneous and somewhat off-putting due to the very fact that it uses PHP. An idea for an alternative would instead have several variations of the server exist, as opposed to
just one language’s server-side script.

Furthermore, the TSL pointed out that the CSS injection that AuO makes at the beginning could result in confusion for clients. After all, simply by loading the AuO library, the CSS sheet gets added, which clients might not expect. A suggested resolution involves re-implementing the CSS loading such that the CSS loaded the first time that an AuO instance gets created.

**Automated Tests**

The TSL also pointed out that clients and future developers alike would find a Selenium automated testing framework both cumbersome and unappealing, preferring technologies like Jasmine, Mocha, and Unit.js instead. Notably, these suggestions run natively in the browser, like AuO itself, as opposed to AuO’s current automated testing setup. After all, unless AuO’s tests take advantage of something in Selenium WebDriver that the JavaScript testing frameworks do not offer, AuO has no reason to choose Selenium WebDriver over the JavaScript frameworks for its tests.

**User Feedback**

The TSL’s user test, conducted on a group of local schoolteachers, resulted in an important comment from the TSL regarding the user experience, noting that the UI, while useful for the composition of AuO’s functions, can seem overwhelming for users who only need to perform a fixed workflow, such as record, play, and save. Thus, for first-time users, the UI does not do a great job of helping users find the functions they want to use.

This indicates that the UI needs additional visual hints for users, to help them easily find the functions that they want. Potential methods include using colors, using symbols, and displaying an informational side panel. Further user testing will show what combination of methods do or do not work.
5.2.8 Client and User Testing Summary

Client and user testing has helped direct where best to focus AuO’s development efforts. This has led to several interesting feature additions for AuO, as well as the awareness of several of AuO’s limitations at this time. Table 5.1 tabulates the key feedback from client and user testing, as well as how AuO responded or will respond to the feedback.
<table>
<thead>
<tr>
<th>Client/User Party</th>
<th>Feedback</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIT TSL</td>
<td>AuO version 0.4 does not save on Chrome 49.</td>
<td>Hard-coded WAV saving path to avoid the MediaRecorder API.</td>
</tr>
<tr>
<td>GameBlox</td>
<td>Drag-and-drop behavior produces jittery behavior.</td>
<td>Make everything in the UI a drop target.</td>
</tr>
<tr>
<td>GameBlox</td>
<td>Feature request for a way to associate times with user-generated labels.</td>
<td>The tags UI and the tagging interactions.</td>
</tr>
<tr>
<td>GameBlox</td>
<td>Feature request for uploading M4A audio into AuO.</td>
<td>Expanded list of MIMEs that AuO can load.</td>
</tr>
<tr>
<td>StarLogo Nova</td>
<td>Integration nominally works fine. Online saving feature not used due to preference for directly handling the audio blobs.</td>
<td>Future work slated for expanding the API to allow access to AuO’s DOM.</td>
</tr>
<tr>
<td>StarLogo Nova</td>
<td>Would like the ability to hide the UI while still accessing AuO’s core features.</td>
<td>Reconsidering the purpose of the save UI.</td>
</tr>
<tr>
<td>StarLogo Nova</td>
<td>Microphone notification in browser on load seems “creepy.”</td>
<td>Recording notification only occurs during recording in AuO version 1.6.</td>
</tr>
<tr>
<td>MIT TSL</td>
<td>Echoing StarLogo Nova’s request for access to API without the UI.</td>
<td>Categorized as future work.</td>
</tr>
<tr>
<td>MIT TSL</td>
<td>Consider changing AuO to allow clients to construct their own flows through AuO.</td>
<td>Assessment and implementation categorized as future work.</td>
</tr>
<tr>
<td>MIT TSL</td>
<td>CSS injection makes clients somewhat nervous.</td>
<td>Future work slated for reassessing the CSS loading procedure.</td>
</tr>
<tr>
<td>MIT TSL</td>
<td>Suggestion to use JS-based testing framework instead of Selenium WebDriver and all the binary dependencies.</td>
<td>Future work slated for investigating and potentially moving testing framework to a JS-based one.</td>
</tr>
<tr>
<td>MIT TSL</td>
<td>First-time user experience a bit overwhelming due the wide variety of available features.</td>
<td>Future work slated for investigating how to improve first-time user experience.</td>
</tr>
</tbody>
</table>

Table 5.1: A tabulation of key client and user testing feedback for AuO.
Chapter 6

Future Work

Much of AuO’s development priorities centered on providing StarLogo Nova with the features it needs for its migration away from ActionScript in its August 2016 relaunch. As a result, there exists many low-hanging fruit in terms for future work on AuO. In this chapter, I address some potential next steps for future work on AuO.

Each action item falls into one of three categories: maintenance, tune-ups, and expansion. Maintenance projects focus on the most basic and necessary updates for AuO to remain current with the latest web technologies, such as switching from an obsolete API to a newer, standardized one. Tune-up projects enhance the existing AuO application by improving browser compatibility, runtime and memory performance, and client and user experiences. Example tune-up projects could reduce AuO’s memory footprint, add important API functions, or make the UI more intuitive. Finally, expansion projects aim to add entirely new feature branches to AuO, beyond AuO’s current capabilities, such as defining and adding support for multiple tracks.

Additionally, each project also has a priority from among the three levels: high, medium, and low. High-priority projects generally have time constraints, usually due to either deadlines or the logistical difficulty of implementing the project after AuO has grown significantly more. Medium-priority projects, representing the bulk of
future work, tend to improve the experience of using or developing AuO, but have no real time constraint, and thus can wait until the high-priority projects have finished. Low-priority projects often have minor impact on users, clients, and developers; thus, these projects can usually lie dormant for long durations without crippling any aspect of AuO, though they could eventually escalate in priority.

6.1 Tags Security

Currently, the tags system insecurely stores the time value for each tag as a DOM attribute, leaving the system open for clients and users to exploit; this presents the opportunity for a high-priority maintenance project. One potential direction for this project involves exploring the feasibility of using JavaScript closure to reduce the scope of the tags’ time values, such that clicking a tag retains its current behavior of moving the ticker, while eliminating client and user access to the time values. If feasible, such a project can then proceed to implement the change, thus resolving this security flaw. Otherwise, the project can continue to explore other possible options for withdrawing direct client and user to the tags’ time values.

6.2 Multiple Tracks

A high-priority expansion project, defining and implementing AuO’s model for multiple tracks can introduce a major feature release that greatly boosts AuO’s value in the web audio recording and editing world. Even though AuO currently lacks both the model and concept for multiple tracks, some of the code already have the latent ability to transform easily to support multiple tracks. Thus, the first major component of this project can focus on defining AuO’s treatment of multiple tracks, and the second major component can focus on implementing the behaviors necessary for AuO to live up to the defined treatment of multiple tracks. For this latter component especially, some of the work can introduce novel code into the AuO library, while other work can focus on integrating the existing code to work under the newly-defined behavior for multiple tracks.
6.3 Client-Side DOM Tree Access

During the integration of AuO into StarLogo Nova, the StarLogo Nova team ran into the issue of needing to simply hide the AuO UI without formally suspending it. Since AuO does not have an API function for this yet, StarLogo Nova current resorts to toggling the display style of the AuO’s container element between “none” and “” (the empty string). Thus, a high-priority expansion project relating to this can analyze what kinds of DOM interactions clients may want to make with AuO’s UI and decide what additional API functions to add as a result of the analysis.

6.4 Automated Testing

Currently, AuO’s automated testing suites run in Java, using Selenium WebDriver and JUnit. Although this choice aimed to widen the field of testing possibilities, as the TSL mentioned in Subsection 5.2.7, the cumbersome setup and narrow scope of the tests have made the current choice of testing framework rather unsatisfying. A high-priority maintenance project would look into the necessity of continuing with Selenium WebDriver and JUnit, as opposed to switching to a browser-based dedicated JavaScript testing framework, such as Jasmine, Mocha, or Unit.js. If deemed necessary, this project would also implement the transition to the new framework of choice.

6.5 UI Improvements

As mentioned in Subsection 5.2.7, AuO’s current UI does not cater the best to new users. A high-priority maintenance project can aim to improve the UI for a better new user experience, while also giving experienced users a smooth and fast-paced workflow, similar to what AuO already offers. Unlike many of the other future work projects listed here, this project could continue through several iterations, even after having reached a satisfactory UI.
6.6 Performance Benchmarking

So far, AuO’s performance testing has relied solely on empirical qualitative performance metrics. This limits the “data” on AuO’s current performance to values like “lags,” “doesn’t lag,” and so forth. A medium-priority tune-up project can more concretely define quantitative performance metrics and benchmarks for AuO, and depending on the granularity of these metrics and benchmarks, help pinpoint where AuO may require some performance engineering. Online resources, such as [4], can potentially inspire contributions to such a project.

6.7 The MediaDevices API

Although the web standards have already deprecated NavigatorUserMedia API [6] and introduced the MediaDevices API, which succeeds the NavigatorUserMedia API, at this time, not all AuO-supported browsers on all operating systems provide adequate support for the MediaDevices API, thus relegating AuO to use the deprecated API instead at the present. Thus, a medium-priority maintenance project for the future can add the use of the promise-based MediaDevices API, once it has garnered sufficient browser support. Thereafter, the project can also remove the old path through the NavigatorUserMedia API once all supported browsers have fully implemented the MediaDevices API.

6.8 The AudioWorkerNode

In another scenario of a deprecated API’s successor having yet to gain ground with major web browsers, the Web Audio [1] API’s AudioWorkerNode, though standardized to replace the deprecated ScriptProcessorNode, currently has no support in major browsers. As a result, AuO presently uses ScriptProcessorNodes, rather than AudioWorkerNodes. Once major supported browsers have implemented AudioWorkerNodes, a medium-priority maintenance project can add a pathway for those browsers to use AudioWorkerNodes instead of ScriptProcessorNodes. Then, once all of AuO’s sup-
ported browsers have implementations of AudioWorkerNodes, the old path through ScriptProcessorNodes can disappear along with the ScriptProcessorNode.

6.9 Mobile Development

At this time, AuO only has partial compatibility with mobile platforms. Notably, even though the AuO UI displays properly on NavigatorUserMedia-enabled browsers, drag-and-drop events do not exist on mobile browsers in the same way that they exist for desktop browsers. As a result, AuO’s drag-and-drop behaviors do not work on most mobile browsers. A medium-priority expansion project can rectify this disparity by recomposing the audio UI’s interactions so that they work for mobile browsers, perhaps by specifically targeting mobile browsers, on top of some additional visual updates to the audio UI’s waveform display. Thereafter, such a project would also bear the responsibility of vetting new features and adding mobile browser ports if the features do not work on mobile browsers by default.

6.10 JavaScript Classes

JavaScript, unfortunately, provides both fast and slow methods of doing the same thing for many functionalities. As [3] suggests, AuO’s current implementation method could benefit from switching to an implementation that defines class methods using the class prototype. A low-priority tune-up project can explore the benefits of massively changing AuO’s current implementation style, and potentially make such a change if deemed necessary.

6.11 Audio Splicing

As a result of StarLogo Nova only needing the trimming feature, AuO’s editing toolbox has only one option: trimming. A low-priority expansion project can take trimming one step further by implementing audio splicing, which selects parts of an audio track to play. This would also encompass the existing trimming feature, as audio
trimming can occur as a special case of audio splicing.

6.12 Audio Amplitude Modulation

Another opportunity for adding new editing features in AuO comes from audio amplitude modulation. A common feature used to change the volume of audio tracks, this low-priority expansion project can give users the ability to not only select which parts of the audio track to use, but also give them the ability to change the volumes of these tracks. Once AuO supports multiple tracks, this feature can, along with audio splicing, allow users to mix audio tracks in AuO.

6.13 Future Work Summary

Table 6.1 contains the short, tabulated summary of these twelve potential future works on AuO. Although future work should aim to work on the high-priority projects first, the priorities assigned to these projects can also change with time, depending on how web technologies evolve.
<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th><strong>Category / Priority</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tags Security</td>
<td>maintenance / high</td>
<td>Remove potential client and user exploit for tags due to the current system storing the time value directly in the DOM element as an attribute.</td>
</tr>
<tr>
<td>Multiple Tracks</td>
<td>expansion / high</td>
<td>Define and implement support and behavior for multiple tracks in AuO.</td>
</tr>
<tr>
<td>Client-Side DOM Access</td>
<td>expansion / high</td>
<td>Add more API functions for clients to work with AuO’s DOM.</td>
</tr>
<tr>
<td>Automated Testing</td>
<td>maintenance / high</td>
<td>Evaluate whether Selenium WebDriver provides needed features unavailable in JavaScript testing frameworks. If not, then implement a migration to a JavaScript testing framework.</td>
</tr>
<tr>
<td>New-User UI Improvements</td>
<td>maintenance / high</td>
<td>Improve the new-user experience for the UI, to feel more manageable.</td>
</tr>
<tr>
<td>Performance Benchmarking</td>
<td>tune-up / medium</td>
<td>Develop a performance benchmarking suite for AuO.</td>
</tr>
<tr>
<td>The MediaDevices API</td>
<td>maintenance / medium</td>
<td>Switch from the NavigatorUserMedia API to the MediaDevices API once available.</td>
</tr>
<tr>
<td>AudioWorkerNode</td>
<td>maintenance / medium</td>
<td>Switch from ScriptProcessorNode to AudioWorkerNode once available.</td>
</tr>
<tr>
<td>Mobile Support</td>
<td>expansion / medium</td>
<td>Implement full mobile browser support for AuO.</td>
</tr>
<tr>
<td>JavaScript Classes</td>
<td>tune-up / low</td>
<td>Switch from current implementation to using JavaScript class prototypes improved runtime performance once necessary.</td>
</tr>
<tr>
<td>Audio Splicing</td>
<td>expansion / low</td>
<td>Instead of just trimming, enable editing by splicing audio.</td>
</tr>
<tr>
<td>Audio Amplitude Modulation</td>
<td>expansion / low</td>
<td>Add the ability to tweak the volume of the audio track.</td>
</tr>
</tbody>
</table>

Table 6.1: Tabulated summary of key potential future work on AuO, sorted by priority with high priority at the top of the table.
Chapter 7

Conclusion

This thesis presents a first-hand account of the development process for AuO, a native-to-browser web application that uses the Web Audio [1], NavigatorUserMedia [6], and MediaRecorder APIs to enable audio recording, playback, editing, loading, and saving on the web. Using these web APIs in tandem with core JavaScript libraries, such as the JavaScript XMLHttpRequest [14] and File [20] APIs, alongside HTML5 [10], CSS3, and JavaScript subject to the standards of ES6 [24], I have built AuO into a single-file JavaScript library that which allows clients, such as StarLogo Nova, GameBlox, and the MIT Teaching and Systems Lab (TSL), to integrate AuO into their projects as their core user-interactive audio recording and editing components, providing a UI that caters to users with varying experience levels, an API that takes undergraduates into account as potential clients, and a codebase that encourages good coding paradigms from the get-go.

First, this thesis enumerates the concrete metrics used to measure this project’s success, of which only two metrics, numbers 5 and 38, remain unsatisfied. Then, the thesis provides the background and motivation that led to the inception of AuO. Initiated by StarLogo Nova’s need for an audio recording and editing component that aligns with StarLogo Nova’s move from ActionScript to purely HTML, CSS, and
JavaScript, AuO’s inception resulted from a desire for an audio recorder and editor that scores highly in the three key properties of integrability, usability, and browser nativity.

Then, this thesis elaborates on AuO’s design, which aims to satisfy the targets set forth in the motivation. In response to these targets, AuO’s high-level design goals outline the key constraints in AuO’s design (satisfying metrics 29, 30, 31, 32, 33, and 34). First, the initial design in AuO version 0.1 describes the design considerations for recording (satisfying metrics 1, 2, 16, 19, and 24), playback (satisfying metrics 3, 4, 17, 20, and 25), saving (satisfying metrics 6, 21, and 26), and characteristics of AuO’s UI (satisfying metric 7 and contributing to satisfying 33). Then, the design of the minimal viable product details the design considerations for AuO’s ticker (satisfying metric 8), trimmers (partly satisfying metric 5), and drag-and-drop features. Next, for AuO version 1.0, the stable release, the design considerations affected AuO’s runtime performance (satisfying metric 23), the added support for WAV files (satisfying metrics 18 and 21), the newly-founded save UI, the handling of multiple channels, and the concurrency concerns. Thereafter, for AuO version 1.6, the thesis release, the design considerations include loading (satisfying metric 27), online and local resources (satisfying metric 26), the tags UI (satisfying metric 27), the hotkeys system (satisfying metrics 9, 10, 11, 12, 13, 14, and 15), and the modals created by mirroring AuO’s own DOM tree. Finally, the chapter wraps up by discussing aspects of the design that leave open projects for post-thesis development, such as handling multiple MIMEs and working with multiple tracks in AuO.

After the design chapter, this thesis documents the implementation details of AuO’s development. This chapter starts off by describing the general implementation guidelines involved in ensuring that AuO emulates industry-level code quality and organization (satisfying metrics 35, 36, and 37, but not 38). Then, this thesis adds details regarding the process of implementing the library (satisfying metric 28) and each of AuO’s components, both seen and unseen, as well as some optimizations (satisfying metric 22). From the library’s FunctionalElement class to the UI itself, this chapter
provides a detailed treatment of the nuances involved in implementing AuO from the
ground up.

Then, this thesis articulates the automated test suites, as well as the results of the
client and user tests. First, the thesis documents the automated test suites, the
framework, and the limitations of the automated tests. Then, the thesis follows up
with a discussion on the results of the client and user testing, and how they affected
AuO’s development process.

Finally, this thesis wraps up with describing twelve potential projects for future work
on AuO. Each of these projects has a priority level of high, medium, or low, as well
as falling into a category of maintenance, tune-up, or expansion projects. Aimed at
providing long-term support for ensure AuO’s survival as a web application, these
potential future works range from updating the UI to adding anticipated features,
with each project targeted at solving a specific problem that exists in the current
AuO implementation.

In the few months since its inception, AuO has already begun to carry responsibilities
and fill a previously-missing niche in web applications. At the same time, however,
AuO has a long road stretched ahead, filled with the promises of delivering fast
and reliable online audio recording and editing, free from third-party plug-ins, and
versatile for many use cases. As the first half of its namesake suggests, AuO’s future
will have golden opportunities, and I look forward to seeing each and every one.
Appendix A

README

AuO’s main documentation takes the form of a README Markdown file, written in GitHub-flavored Markdown. Below, I have reproduced the release version 1.6 snapshot of the README.md. To access the latest version’s README.md documentation, visit https://github.com/wqian94/AuO/blob/gh-pages/README.md.
AuO version 1.6 (stable)

AuO (IPA: /əu/), a browser-based audio recording and editing application. Uses browser-native technologies to avoid third-party dependencies.

Licensing

AuO is released under the MIT License. See the LICENSE file in the repository for full license details.

Using AuO

Include AuO in any application by including AuO.js. Then, create an instance with

```javascript
const auo = new AuO(link_to_server_url, online_save_callback, local_save_callback);
```

where `link_to_server` is the URL to upload audio clips and `online_save_callback` is the callback function used to process the server's response after uploading the saved audio file. Similarly, the `local_save_callback` is invoked when the user chooses to save offline, and receives a blob as its sole parameter. This triggers the download of the audio recording with the name `recording.ext` where `ext` is the appropriate extension for the save format chosen from the UI.

To use the default callbacks (a modal dialog that displays the server's response for online and a save prompt for local), either set the callbacks to `null` during construction, or do not set them at all (they will default to `null`, then be replaced with their defaults).

To launch AuO, simply call

```javascript
auo.launch();
```

and the interface should appear. To suspend AuO, click Close or call

```javascript
auo.suspend();
```

to make the interface disappear and prevent resource use. When you are ready to use AuO once more, call

```javascript
auo.launch();
```
auo.launch();

again to relaunch the interface using the same instance.

**Supported Browsers and Operating Systems**

<table>
<thead>
<tr>
<th></th>
<th>Linux (Debian/Ubuntu)</th>
<th>Windows</th>
<th>Max OS X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chrome</td>
<td>49.0+</td>
<td>49.0+</td>
<td>49.0+</td>
</tr>
<tr>
<td>Firefox</td>
<td>45.2+</td>
<td>45.2+</td>
<td>45.2+</td>
</tr>
<tr>
<td>Edge</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Internet Explorer</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Opera</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Safari</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

**Notes:**

1. Chrome 49 does not support saving as WebM, so that save option is not shown in Chrome 49.
2. Firefox does not support loading some file formats, claiming that those formats are malformed.
3. At the moment, Safari, Opera, and Edge do not support the MediaRecorder API, which is required for capturing audio streams in JavaScript.

**The AuO User Interface**

AuO currently supports audio recording, playback, editing, and saving to a remote server. Currently, there is no support for saving to any format other than WebM and WAV, though many formats are supported for loading. In addition, the user can also zoom in and out to focus on parts of the audio clip. At the moment, AuO supports up to 16 levels of zoom, for a total magnification of approximately 18.5 times the original view of the waveform.

**Initial Launch**

When AuO first launches, the user will see only the options to record and zoom, since AuO has not captured any audio data yet. The first thing the user should do at this stage is click the **Record** button.
Audio Recording

By clicking the **Record** button, the user can begin to gather audio feed live from the computer’s microphone, or potentially other connected audio recording devices as well. For first-time users, a dialog box should appear in the browser to request permission to access the audio recording device, and choose a device if the computer has multiple available devices. If this dialog box does not appear, make sure that the browser’s URL points to an `https://` URL (or `http://` if running on localhost) and not a `file://` URL.

During recording, the user can zoom in and out, as well as reset the zoom to 100% if the current zoom does not equal 100%.

To stop recording, the user can hit the **Stop** button. AuO will briefly pause to preprocess the data for playbacks and saves, before entering the idle state. At this point, the browser's status bar should also indicate that AuO is no longer using the microphone.

Zooming

At any point during recording and playback, or when AuO is idling, the user can click the **Zoom in** and **Zoom out** buttons to zoom in and out of the visualizer graph, respectively. The graph will automatically resize and relabel the times accordingly. The user can also reset the zoom to the default value of 100% by clicking the **Zoom reset** button.

Increasing the zoom allows the user to set trimmings and playback points more accurately; decreasing the zoom allows the user to better navigate through the entire audio track.

Panning

At any point, the user can pan by dragging the graphs where the ticker and trimming boxes do not cover the graphs. Dragging from right to left will pan right, and dragging from left to right will pan left.

Idling

AuO’s idle state occurs when AuO has a recording stored, but no active actions on that recording. The user can identify this by seeing if a graph exists, and whether the **Record**, **Play**, and **Upload** (or **Download**) buttons have been disabled. If the graph does not exist, then AuO has not recorded anything yet and is not in the idle state; similarly, if any of the three mentioned buttons has been disabled, then AuO is currently performing that action and is not in the idle state.
In the idle state, the user can trim the audio clip from either end, as well as reposition the ticker to begin playback at different parts of the audio recording.

**Ticker**

The ticker is the red bar on the graph. This represents where AuO will begin playing the audio recording when the user clicks Play. By moving this bar around, the user can allow AuO to begin playing at different locations in the recording.

When hovering over the ticker, a label will appear next to the ticker to indicate the time that the ticker is at.

**Tagging**

At any point, users can tag the current location of the ticker by clicking the Tag... button. This will pop up a modal that prompts the user for a string label. If an empty string is given, the label will default to the time for that tag. Users can then return to that time in the idle state by clicking the tag. Shift-clicking a tag will cause it to disappear, deleting it from the list of current tags.
Please enter a label for the tag at 0.00s. Leave the field blank to use the time as the label.

```
0.00s
OK
```

Additionally, tagging an already-tagged time will overwrite the previous label and can be used to update labels. Note that the default value for the label will always be the time; re-tagging and then closing the modal will automatically relabel the tag with the time.

Furthermore, the call

```
auo.getTags();
```

returns a map from the times (in integer milliseconds) to their string labels. Note that these times are relative to the trimmed start at the time that the API call is made, and tags that fall outside the playable range are not included in the map.

### Trimming Boxes

At either end of the graph, there is a blue box with an arrow pointing toward the graph, as shown in the screenshots above. These are the trimming boxes. The one on the left is the start-trimming box, and the one on the right is the end-trimming box. These boxes represent the start and end of the trimmed audio recording, respectively, and if one were to save the recording, the produced audio clip will contain only the audio between the two boxes.

Similar to the ticker, when hovering over the trimming boxes, a label will appear next to the inner edge of the box, indicating the trimming. Note that the start-trimming box displays positive time to indicate offset from the start, and the end-trimming box displays negative time to indicate offset from the end.

### Audio Playback

By clicking the **Play** button, the user can play back the recorded audio file, starting at the red ticker and ending at the end-trimming box. Once it reaches the end, AuO will automatically stop playback and return to the idle state. If the user clicks play when the ticker is already at the end-trimming box, the ticker will instead loop to the start-trimming box and play the entirety of the trimmed audio.

### Audio Editing

Currently, AuO only supports audio editing by trimming.

### Trimming

To trim an audio recording, the user simply drags either the start-trimming box or the end-trimming box to reposition the start and end of the recording, respectively. If this leaves the ticker in a location that would not be played as a result of the trimming, the ticker will automatically move with the trimming boxes. If the trimming was made in error, repositioning the trimming box without playing the trimmed audio will allow the ticker to reposition itself as close to its original position as it can.

### Loading and Saving

AuO has built-in functionality to allow users to load audio from and save audio to both the local filesystem as well as external servers. This functionality is toggled through the resource marker, which is between the Load and Save UIs. There are two options, **Online** and **Offline**, which affects both loading and saving at the same time. The active mode is highlighted in green, while the inactive mode is grayed out. By clicking on the two toggle options, the user can switch between the two modes.
Audio Loading

Users can select an audio file to upload into the editor by clicking the area next to the Load button. This will produce a popup window for selecting a file from the local filesystem to load into the audio editor. After selecting a file, the name of the file to load will show up next to the Load button. Clicking the now-enabled Load button will load the file into the editor. At this point, the user can continue to use AuO as if the audio track had just been recorded.

The audio formats supported for loading are listed below:

- Advanced Audio Coding (AAC; .aac)
- Moving Pictures Experts Group (MPEG, MP3, MP4; .m4a, .m4p, .mp3, .mp4, .mpa, .mpe, .mpg, .mpeg)
- Ogg (OGG; .ogg, .ogv)
- Waveform Audio File Format (WAVE; .wav, .wave)
- WebM (WEBM; .webm)

Audio Saving

When in online mode, the user can click the Upload button to save and upload the audio recording to the server, which should reply with a link to the saved audio clip. The user can then retrieve this link from the modal dialog that pops up when the saving has succeeded. Both the location of the server to upload to and the callback handler upon a successful save can be changed by passing in the appropriate parameters to AuO’s constructor. If using a custom handler on successfully save, the handler should take an XMLHttpRequest object as its parameter, and the link should reside in the response property of that object.

When in offline mode, the user can click the Download button to initiate a local file download using the browser’s file download prompt. This can be customized with a handler as well; this handler should accept a JavaScript blob containing the audio data to save.

If the server URL has been omitted or set to null, then online saving is disabled, and the Upload button will not be enabled.

Users can choose the format in which AuO will save the audio recording by selecting the format from the dropdown menu next to the Upload or Download button.

Currently, for Chrome 49, only WAV is available, while for Chrome 50+, both WAV and WebM are available.

Callback Function

The callback function is the second parameter in the constructor for a new AuO instance, and is optional. This callback function is called once the server has responded with an HTTP 2xx in response to the user’s save request. If this parameter is omitted during construction, AuO will use the default callback function, which produces a modal dialog box with the server’s response, as shown in the image below:

If the parameter was provided during construction, AuO will call that function and pass in one parameter: the XMLHttpRequest object, whose response field contains the server’s response.
## Hotkeys

For convenience, AuO has hotkeys mapped to several of the main functions of AuO. They are mapped below:

<table>
<thead>
<tr>
<th>Key combinations</th>
<th>Key description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>◄</td>
<td>Left arrow key</td>
<td>Move ticker left by 1 pixel, to an earlier time.</td>
</tr>
<tr>
<td>►</td>
<td>Right arrow key</td>
<td>Move ticker right by 1 pixel, to a later time.</td>
</tr>
<tr>
<td>shift + ◄</td>
<td>Shift + left arrow key</td>
<td>Move ticker left by 100 pixels, to an earlier time.</td>
</tr>
<tr>
<td>shift + ►</td>
<td>Shift + right arrow key</td>
<td>Move ticker right by 100 pixels, to a later time.</td>
</tr>
<tr>
<td>ctrl + ◄</td>
<td>Ctrl + left arrow key</td>
<td>Scroll waveform display left by 10 pixels, to an earlier time.</td>
</tr>
<tr>
<td>ctrl + ►</td>
<td>Ctrl + right arrow key</td>
<td>Scroll waveform display right by 10 pixels, to a later time.</td>
</tr>
<tr>
<td>[</td>
<td>Left square bracket key</td>
<td>Increase size of the start trimming box.</td>
</tr>
<tr>
<td>]</td>
<td>Right square bracket key</td>
<td>Increase size of the end trimming box.</td>
</tr>
<tr>
<td>shift + [ (a.k.a. {)</td>
<td>Shift + left square bracket key (a.k.a. left curly brace)</td>
<td>Decrease size of the start trimming box.</td>
</tr>
<tr>
<td>shift + ] (a.k.a. })</td>
<td>Shift + right square bracket key (a.k.a. right curly brace)</td>
<td>Decrease size of the end trimming box.</td>
</tr>
<tr>
<td>+</td>
<td>Plus key</td>
<td>Zoom in.</td>
</tr>
<tr>
<td>-</td>
<td>Minus key</td>
<td>Zoom out.</td>
</tr>
<tr>
<td>0</td>
<td>Zero key</td>
<td>Zoom reset.</td>
</tr>
<tr>
<td>f</td>
<td>F key</td>
<td>Focus on the ticker, centering the ticker in the display as much as possible without having the display go out of bounds.</td>
</tr>
<tr>
<td>l</td>
<td>L key</td>
<td>Focus on the audio file loader. If in online mode, will focus on the URL input box. If in offline mode, will pop up file selector dialog.</td>
</tr>
<tr>
<td>o</td>
<td>O key</td>
<td>Toggle between online/offline modes.</td>
</tr>
<tr>
<td>r</td>
<td>R key</td>
<td>Begin recording an audio clip.</td>
</tr>
<tr>
<td>s</td>
<td>S key</td>
<td>Save the trimmed audio clip in the mode (online/offline) that is currently active.</td>
</tr>
<tr>
<td>t</td>
<td>T key</td>
<td>Tag the current ticker time.</td>
</tr>
<tr>
<td>Space</td>
<td>Spacebar</td>
<td>Emulates clicking either the Stop or Play button, whichever one is enabled.</td>
</tr>
</tbody>
</table>

## Providing a backend for AuO

The code in receive.php provides a simple PHP server-side script for supporting online file uploads from AuO. This code is reproduced below:
<?php
/**
 * receive.php
 *
 * A simple server-side script to interact with AuO for saving audio clips to a server. Update
 * $HOSTNAME to match the server's hostname and this file is all good to go. Just ensure that the
 * script has the requisite permissions to write to the server.
 */

$HOSTNAME = "https://localhost"; // Set this differently if you have a different host server.

// Make sure that the script has write permissions here!
if (!is_dir("sounds")) {
    mkdir sounds;
}

// Necessary for cross-origin requests, if AuO is hosted under a different host name than this
// script.
header("Access-Control-Allow-Headers: content-type");
header("Access-Control-Allow-Origin: *");

// Content type information.
$content_type = $_SERVER["CONTENT_TYPE"];
preg_match("/audio/([^; \]+); codecs=(([^; ]+))/", $content_type, $content_type_matches);
$save_format = $content_type_matches[1];

$raw = file_get_contents("php://input");

// Create a unique filename by md5 hashing.
$counter = 0;
while (file_exists("sounds/".$filename)) {
    $filename = md5(date('U') . $raw . (++$counter) . "$save_format");
} file_put_contents("sounds/$filename", $raw);

// Will generate the correct link to the saved audio file.
$link = substr($_SERVER["PHP_SELF"], 0, strpos($_SERVER["PHP_SELF"], "/"));
$link = "$HOSTNAME/link/sounds/$filename";

echo $link;
?>

Development

AuO is currently developed and maintained by William Qian as part of his Master of Engineering thesis project with Daniel
Wendel and Eric Klopfer in the MIT Scheller Teacher Education Program.

Bugs and issues should be reported in the GitHub issues page for AuO at https://github.com/wqian94/AuO/issues along with
any helpful information that you can provide, such as operating system, browser (name and version), and any screenshots or
videos that can help recreate the bug.
Appendix B

Summaries

This appendix contains all the summary figures and tables produced in this thesis, as a quick-guide reference.
Figure 3-13: State diagram showing the summary of the UI design. Users can set
tags and close out (suspend) at any time. The red line connects the “Idle state”
nodes that represent the same state, duplicated to make the diagram easier to read.

Figure 3-14: State diagram showing the summary of the API design.
<table>
<thead>
<tr>
<th>Functionality</th>
<th>Associated Elements</th>
<th>Associated Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hotkeys</td>
<td>container</td>
<td>hotkeyHandler</td>
</tr>
<tr>
<td>Loading</td>
<td>buttonLoad, loadFile,</td>
<td>beginAudioLoad,</td>
</tr>
<tr>
<td></td>
<td>loadFileLabel, loadUI</td>
<td>loadArrayOfBlobs,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>loadFileHandler</td>
</tr>
<tr>
<td>Playback</td>
<td>buttonPlay, buttonStop,</td>
<td>beginAudioPlayback</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td></td>
</tr>
<tr>
<td>Recording</td>
<td>buttonRecord, buttonStop,</td>
<td>beginAudioRecording,</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>endAudioRecording,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>processAudioRecording</td>
</tr>
<tr>
<td>Saving</td>
<td>buttonSave, saveOptions,</td>
<td>beginAudioSave,</td>
</tr>
<tr>
<td></td>
<td>saveOptionsLabel, saveUI</td>
<td>LOCAL_SAVE_CALLBACK,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAVE_CALLBACK,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAVE_URL</td>
</tr>
<tr>
<td>Tagging</td>
<td>buttonTag, tagList, tagUI</td>
<td>beginTagUpdateLoop,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tagClickHandler,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tagCreateLabel,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tagLabelComparator,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tagsUpdateUI</td>
</tr>
<tr>
<td>Ticker</td>
<td>audioTicker, audioTickerLabel</td>
<td>addAudioDropHandlers,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>animateAudioTicker,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>beginAudioDisplayLoop</td>
</tr>
<tr>
<td>Trimming</td>
<td>audioEndTrimmer,</td>
<td>addAudioDropHandlers,</td>
</tr>
<tr>
<td></td>
<td>audioEndTrimmerLabel,</td>
<td>animateAudioTrimmers,</td>
</tr>
<tr>
<td></td>
<td>audioEndTrimmerVisual,</td>
<td>beginAudioDisplayLoop</td>
</tr>
<tr>
<td></td>
<td>audioStartTrimmer,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>audioStartTrimmerLabel,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>audioStartTrimmerVisual</td>
<td></td>
</tr>
<tr>
<td>Waveform Display</td>
<td>audioDisplay, audioDisplay</td>
<td>beginAudioDisplayLoop,</td>
</tr>
<tr>
<td></td>
<td>Container, audioUI, audioVisualizer</td>
<td>animateAudioDisplay,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>animateAudioDisplayByForce</td>
</tr>
<tr>
<td>Zooming</td>
<td>buttonZoomIn, buttonZoomOut,</td>
<td>zoomFactor, zoomUpdate</td>
</tr>
<tr>
<td></td>
<td>buttonZoomReset, zoomDisplay,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>zoomUI</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: A short summary of functionalities, major associated FunctionalElement objects, and major associated functions.
AuO's concrete tests

AuO's abstract test class

AuoServer class

JUnit

Web class

Eclipse Jetty

Selenium WebDriver

Google Chrome

Xvfb

Figure 5-3: Diagram of AuO's testing framework building on Eclipse Jetty, JUnit, Selenium WebDriver, Google Chrome, and Xvfb.
<table>
<thead>
<tr>
<th>Client/User Party</th>
<th>Feedback</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIT TSL</td>
<td>AuO version 0.4 does not save on Chrome 49.</td>
<td>Hard-coded WAV saving path to avoid the MediaRecorder API.</td>
</tr>
<tr>
<td>GameBlox</td>
<td>Drag-and-drop behavior produces jittery behavior.</td>
<td>Make everything in the UI a drop target.</td>
</tr>
<tr>
<td>GameBlox</td>
<td>Feature request for a way to associate times with user-generated labels.</td>
<td>The tags UI and the tagging interactions.</td>
</tr>
<tr>
<td>GameBlox</td>
<td>Feature request for uploading M4A audio into AuO.</td>
<td>Expanded list of MIMEs that AuO can load.</td>
</tr>
<tr>
<td>StarLogo Nova</td>
<td>Integration nominally works fine. Online saving feature not used due to preference for directly handling the audio blobs.</td>
<td>Future work slated for expanding the API to allow access to AuO’s DOM.</td>
</tr>
<tr>
<td>StarLogo Nova</td>
<td>Would like the ability to hide the UI while still accessing AuO’s core features.</td>
<td>Reconsidering the purpose of the save UI.</td>
</tr>
<tr>
<td>StarLogo Nova</td>
<td>Microphone notification in browser on load seems “creepy.”</td>
<td>Recording notification only occurs during recording in AuO version 1.6.</td>
</tr>
<tr>
<td>MIT TSL</td>
<td>Echoing StarLogo Nova’s request for access to API without the UI.</td>
<td>Categorized as future work.</td>
</tr>
<tr>
<td>MIT TSL</td>
<td>Consider changing AuO to allow clients to construct their own flows through AuO.</td>
<td>Assessment and implementation categorized as future work.</td>
</tr>
<tr>
<td>MIT TSL</td>
<td>CSS injection makes clients somewhat nervous.</td>
<td>Future work slated for reassessing the CSS loading procedure.</td>
</tr>
<tr>
<td>MIT TSL</td>
<td>Suggestion to use JS-based testing framework instead of Selenium WebDriver and all the binary dependencies.</td>
<td>Future work slated for investigating and potentially moving testing framework to a JS-based one.</td>
</tr>
<tr>
<td>MIT TSL</td>
<td>First-time user experience a bit overwhelming due the wide variety of available features.</td>
<td>Future work slated for investigating how to improve first-time user experience.</td>
</tr>
</tbody>
</table>

Table 5.1: A tabulation of key client and user testing feedback for AuO.
<table>
<thead>
<tr>
<th>Title</th>
<th>Category / Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tags Security</td>
<td>maintenance / high</td>
<td>Remove potential client and user exploit for tags due to the current system storing the time value directly in the DOM element as an attribute.</td>
</tr>
<tr>
<td>Multiple Tracks</td>
<td>expansion / high</td>
<td>Define and implement support and behavior for multiple tracks in AuO.</td>
</tr>
<tr>
<td>Client-Side DOM Access</td>
<td>expansion / high</td>
<td>Add more API functions for clients to work with AuO’s DOM.</td>
</tr>
<tr>
<td>Automated Testing</td>
<td>maintenance / high</td>
<td>Evaluate whether Selenium WebDriver provides needed features unavailable in JavaScript testing frameworks. If not, then implement a migration to a JavaScript testing framework.</td>
</tr>
<tr>
<td>New-User UI Improvements</td>
<td>maintenance / high</td>
<td>Improve the new-user experience for the UI, to feel more manageable.</td>
</tr>
<tr>
<td>Performance Benchmarking</td>
<td>tune-up / medium</td>
<td>Develop a performance benchmarking suite for AuO.</td>
</tr>
<tr>
<td>The MediaDevices API</td>
<td>maintenance / medium</td>
<td>Switch from the NavigatorUserMedia API to the MediaDevices API once available.</td>
</tr>
<tr>
<td>AudioWorkerNode</td>
<td>maintenance / medium</td>
<td>Switch from ScriptProcessorNode to AudioWorkerNode once available.</td>
</tr>
<tr>
<td>Mobile Support</td>
<td>expansion / medium</td>
<td>Implement full mobile browser support for AuO.</td>
</tr>
<tr>
<td>JavaScript Classes</td>
<td>tune-up / low</td>
<td>Switch from current implementation to using JavaScript class prototypes improved runtime performance once necessary.</td>
</tr>
<tr>
<td>Audio Splicing</td>
<td>expansion / low</td>
<td>Instead of just trimming, enable editing by splicing audio.</td>
</tr>
<tr>
<td>Audio Amplitude Modulation</td>
<td>expansion / low</td>
<td>Add the ability to tweak the volume of the audio track.</td>
</tr>
</tbody>
</table>

Table 6.1: Tabulated summary of key potential future work on AuO, sorted by priority with high priority at the top of the table.
Bibliography


