BACHELOR THESIS
Sascha Knapp

Browser-based Engine Prototype for a Digital Audio Workstation

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Sascha Knapp

Browserbasierter Engine-Prototyp für eine Digital Audio Workstation

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Informatik Technischer Systeme*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr.-Ing. Lars Hamann

Eingereicht am: 08. Februar 2023

Sascha Knapp

Thema der Arbeit

Browserbasierter Engine-Prototyp für eine Digital Audio Workstation

Stichworte

Web Audio API, AudioWorklet, SharedArrayBuffer, Web Audio Modules, JavaScript, WebAssembly, Digital Audio Workstation, DAW

Kurzzusammenfassung

Diese Abschlussarbeit befasst sich mit der Entwicklung eines browserbasierten Engine-Prototyps für eine Digital Audio Workstation (DAW), die dieselben Qualitätsziele verfolgt wie native DAWs. Die Engine nutzt modernste Webtechnologien, darunter AudioWorklet, SharedArrayBuffer, und Web Worker, um ein effizientes Disk-Streaming zu erreichen. Der Prototyp bietet Plugin-Unterstützung durch die Verwendung vom Web Audio Modules-Standard zusammen mit WebAssembly für die Audioverarbeitung. Ein umfassender Überblick über bestehende browserbasierte DAWs, aktuelle Veröffentlichungen und Evaluationen der Audioverarbeitung im Webbrowser bieten eine Basis für die Entwicklung und die Auswertung des Prototyps. Ein automatisierter Evaluationsprozess mit dem Firefox Profiler wird entwickelt und angewendet, um den entwickelten Disk-Streaming-Ansatz mit einem zweiten Ansatz zu vergleichen, der das AudioBufferSourceNode interface verwendet. Außerdem wird die Performanz der Audioverarbeitung mit WebAssembly und JavaScript verglichen. Die Ergebnisse demonstrieren das Potenzial browserbasierter DAWs, eine brauchbare Alternative zu nativen DAWs zu bieten, da das Web die notwendigen Technologien zur Implementierung ähnlicher Paradigmen zur Audioverarbeitung bereitstellt, die auch schon in nativen DAWs zu finden sind. iv

Sascha Knapp

Title of Thesis

Browser-based Engine Prototype for a Digital Audio Workstation

Keywords

Web Audio API, AudioWorklet, SharedArrayBuffer, Web Audio Modules, JavaScript, WebAssembly, Digital Audio Workstation, DAW

Abstract

This thesis presents the development of a browser-based engine prototype for a digital audio workstation (DAW), aiming for the same quality goals as native DAWs. The engine makes use of cutting-edge web technologies, including AudioWorklet, SharedArrayBuffer, and web workers, to achieve efficient disk streaming. The prototype offers plugin support through the use of Web Audio Modules with WebAssembly for audio processing. A comprehensive review of existing browser-based DAWs, current publications and evaluations of audio processing in the web browser provide a base for the development and the evaluation of the prototype. An automated evaluation process using the Firefox Profiler is developed and applied to compare the developed disk streaming approach with a second approach that uses the AudioBufferSourceNode. Furthermore, the performance of the audio processing with WebAssembly and JavaScript is compared. The results demonstrate the potential of browser-based DAWs to provide a viable alternative to native DAWs, because the web provides the necessary technologies to implement similar audio processing paradigms found in native DAWs.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Problem statement

Applications for professional digital audio production have real-time constraints, mainly running on non-real-time general-purpose operating systems, typically programmed in C or C++. This has excluded the use of web browsers as a platform in the past, since the real-time requirements could not be met with JavaScript and the Web Audio API. JavaScript is a high-level programming language that is commonly used for creating interactive web applications, but it is not well suited for low-level tasks such as audio processing. Additionally, JavaScript is not a compiled language, which means that it may not offer the performance and efficiency needed for a digital audio workstation (DAW). Recent developments like WebAssembly and the AudioWorklet API on the other hand may enable high performance audio processing in web browsers. Another challenge of audio applications in the web browser is the lack of extensibility, as many people use expensive audio plugins in native DAWs that cannot be used in web browsers as of today. There is an attempt to create an open plugin standard for web browsers called Web Audio Modules, which might open the possibility for plugin developers to port their products to the web, too.

When developing a DAW for the web, a crucial consideration is managing memory usage while playing audio [30]. The common method of using AudioBufferSourceNode to load and play entire audio files uses large amounts of memory. Streaming audio by scheduling smaller buffers with AudioBufferSourceNode reduces memory usage but can lead to audio or GUI artifacts, as the AudioBufferSourceNode can schedule audio buffers on the main thread only. An alternative approach is using the fairly new AudioWorkletNode, designed for real-time audio processing. However, there are no examples or comparison studies of using AudioWorkletNode for streaming audio in DAWs for the web. According to the MDN Web Docs, AudioWorkletNode should be used over AudioBufferSourceNode, when audio must be played from disk [9], but the documentation does not provide examples for that. Despite this statement from the MDN Web Docs, DAWs in production use AudioBufferSourceNode, as chapter 3 shows. As far as is known, there is no existing publication that shows a complete end-to-end implementation of streaming audio from disk using AudioWorklet. Additionally, there appears to be a lack of comparison studies between the AudioWorklet and AudioBufferSourceNode in terms of their performance and efficiency for streaming audio in DAWs for the web.

1.2 Goals and Requirements

Creating a web-based DAW offers benefits like reaching more people, using web accessibility features for a better user experience, and removing barriers to entry professional audio software. This thesis specifically aims to reach developers working on creating an engine or DAW for the web platform.

The goal of this work is to evaluate the performance of the web platform for professional audio applications by developing a prototype of the core audio component of a DAW using the Web Audio API and complementary web technologies. The prototype focuses on the DAW use case with the two key aspects of audio playback and the comparison of JavaScript against WebAssembly as audio processing language. To achieve the audio playback, an efficient way for streaming audio from the local hard disk must be implemented.

This work does not aim to make a comparison between browser-based and native DAWs, as this would require a comprehensive evaluation approach beyond the scope of this research. Additionally, this research does not have prior knowledge on how to implement an effective browser-based DAW engine, as this is the primary goal of this work.

1.2.1 Business context

The following figure and table outline an overview over the major context in which the prototype operates, including the key elements such as the developed engine itself, the client that uses the engine, the audio interface, plugins, and audio files, that the engine needs to operate with. These elements will be analyzed and discussed in the further course of this work.

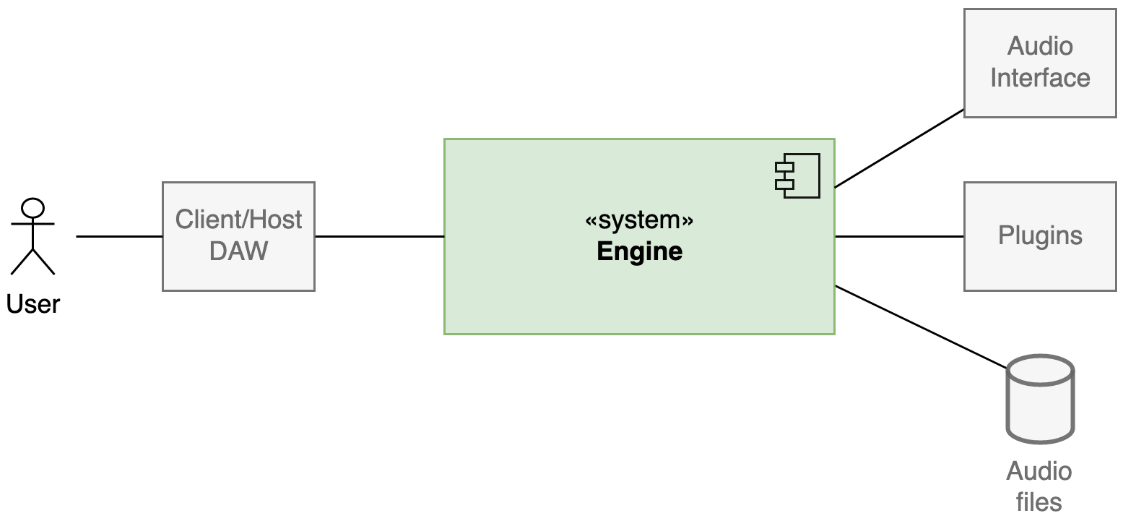


Figure 1.1: Business context

Element	Description
Engine	The engine prototype that is developed in this work.
Client/Host DAW	Client application that uses the engine, usually providing a user interface.
Audio Interface	The device providing audio inputs (optional) and audio outputs to the user. This can be the built-in sound card or a dedicated audio interface for audio production.
Plugins	Audio effects, instruments or visualizers that can be loaded and processed in the engine.
Audio files	Audio files that are stored on hard disk and are read and written by the engine.

Table 1.1: Business Context Description

1.2.2 Requirements

The following list shows the requirements for the DAW engine, that should be implemented for the prototype. The engine provides basic audio functionalities for client DAW applications.

The prototype prioritizes the following functionality and simplifies the model by excluding features for removing tracks, clips, and plugins, but it is designed in a way that allows for easy integration of these functions in the future.

ID	Requirement	Description
REQ1	Create audio tracks	Allows the user to create new tracks for audio recording or importing audio files.
REQ2	Add audio files	Allows the user to import audio files from their local file system and add them to the created tracks.
REQ3	Start playback	Allows the user to play the audio tracks.
REQ4	Seek to playback position	Allows the user to move the playback position to a specific point in the audio track.
REQ5	Stop playback	Allows the user to stop the playback of the audio tracks.
REQ6	Insert audio effects to tracks	This feature allows the user to apply audio effects to the audio tracks such as reverb, delay, and more.

Table 1.2: Requirements list

When developing a DAW, other related topics such as recording, virtual instruments, and MIDI may also be considered. For the purposes of this work, these areas are not extensively covered. Recording functionality is modeled in the developed audio track architecture, but this functionality is not exposed by the engine, to keep the focus on the key aspects of playback and audio processing language comparison. Approaches for recording are similar to those presented for playback and could be implemented similarly, whether through the MediaStream Recording API or the AudioWorklet, but it would need another approach for evaluating the latencies that might be introduced when using the MediaStream API[1]. Virtual instruments are provided using Web Audio Modules. While this work only covers audio effects explicitly, a prototype of a virtual instrument is provided in the repository of this work, to show that this already works out of the box with Web Audio Modules. MIDI is provided by the MIDI API, which is currently only available in the Chromium browser engine and in experimental form in Firefox, with

[1] https://www.w3.org/2021/03/media-production-workshop/talks/ulf-hammarqvist-audio-latency.html

Apple not planning to implement it due to security concerns[2]. This work focuses on using Web APIs that are widely supported across popular browser engines.

1.2.3 Quality goals

Goal	Description
No dropouts	Ensure that audio playback is uninterrupted and without any dropouts, even under heavy load. Dropouts should not occur due to a heavy burden on the main thread from GUI calculation.
Low RAM usage	Minimize memory usage by using audio buffers effectively, avoiding unnecessary storage of large audio files in memory. This helps to ensure that the application runs smoothly even on devices with limited RAM.
Responsive GUI	Ensure that the user interface is responsive and fast, even under heavy audio load.
Low Latency	Achieve low latency both when monitoring audio and during user interface interaction, such as adjusting parameters.
Interoperability/ Extensibility	As the evaluation in chapter 5 compares a AudioBufferSourceNode and a AudioWorkletNode implementation, the streaming should be interchangeable. Additionally, the streaming implementation should be extensible to allow for the exploration of other streaming possibilities in the future, such as WebRTC.

Table 1.3: Quality goals list

[2]https://css-tricks.com/apple-declined-to-implement-16-web-apis-in-safari-due-to-privacy-concerns

1.3 Method

The architecture of the developed system is documented using the arc42 template, a widely-used method for documenting software architectures. The template provides a structure for documenting the key elements of a system and their relationships, and is intended to help developers, architects, and stakeholders communicate and understand the design and organization of a software system [34]. In this work, the arc42 template is used to document the technical context, solution strategy, building block views, and runtime views of the system, as well as the business context and quality goals. As a starting point for the development of the engine, other browser-based DAWs and their use of the Web Audio API is investigated. In order to evaluate the performance of the developed system, relevant evaluations from blogs and papers are also examined to find inspiration for potential experiment setups and to understand current perspectives and trends on the use of WebAssembly for audio processing. To evaluate the developed system, a test client application is implemented that uses the engine. The test client performs automated experiments that log various metrics for comparison. The goal is to compare the performance of the new audio streaming implementation in this work against an existing implementation that only works on the main thread. On the other hand, the implementation of audio processing code in JavaScript and WebAssembly is compared using a Web Audio Module that is specifically implemented for the evaluation. A common metric to evaluate the audio processing performance is the audio callback load, which is relevant for comparing the streaming approaches as well as the audio processing language comparison. The experiments indicate, if the compared solutions create a significant overhead over the other. For the comparison of the streaming implementations, the frame rate on the main thread is also considered, as well as the overall CPU load and memory consumption of the process. As the existing streaming implementation is scheduling audio buffers from the main thread, the frame rate is affected negatively. The experiment is expected to show the advantage of moving the streaming away from the main thread by achieving a stable frame rate. As the developed solution for this work is introducing an additional thread for preloading audio buffers, there is a chance, that the overall CPU load is increased over the other solution.

1.4 Structure

Chapter 2 introduces the reader to the foundational technologies and terminology used in this work, starting the first part of the chapter with explaining DAWs and VST to the reader. This is followed by general concepts and explanations of audio terms. The second part of the chapter shows the audio related technologies that are provided in web browsers, with a focus on the Web Audio API. The chapter closes with the introduction of the Web Audio Modules as a counterpart to VST. Chapter 3 shows the related audio applications and libraries already existing in the web browser as well as research done in that field, which creates the base on which this work builds. In chapter 4, the architecture of the prototype is presented as outlined in the arc42 template. The chapter starts by presenting the constraints that are taken into account during the architecture development, followed by the technical context in which the prototype operates. After that, the solution strategy presents the actions taken to achieve the defined quality goals. The software architecture of the prototype is presented, including the building block view and runtime view. The evaluation in chapter 5 presents the results of the experiments on the performance of the implemented prototype. It includes an overview of the metrics used and the general experiment setup, as well as a comparison of the two audio playback approaches. This is followed by a comparison of using WebAssembly and JavaScript as audio processing languages. The chapter ends with discussing the results of the evaluation. Chapter 6 closes this thesis by summarizing the findings of this work and gives a perspective to potential future work.

2 Background

This chapter introduces the foundation and terminology on which this thesis is build. First, native technologies around Digital Audio Workstations used today are presented following an overview over the possible counterparts which can be found in the web browser.

2.1 Digital Audio Workstation

A digital audio workstation, or DAW, is a software that allows recording, editing, processing and mixing of audio on multiple tracks. The term DAW is also often used to describe a whole system consisting of software and hardware [23]. In this thesis, the term DAW is used, to refer to the software system which runs on a personal computer. Some popular DAWs today are Steinberg Cubase, Avid Pro Tools, Apple Logic Pro, Ableton Live, Bitwig Studio, Presonus Studio One or Cockos Reaper.

Before DAWs were widely used in recording studios, audio was recorded with analog tape machines. A track makes up a small portion located on an analog tape to record individual music instruments independent of each other. Today, the digital counterpart of a track refers to one or multiple audio files that are arranged on a timeline. These audio files are stored on hard disk and are referenced by the tracks. These references are referred to as clips or regions, which can be freely positioned on the timeline of the DAW. In the past days, this was only possible by cutting and splicing analog tape recordings, which is a destructive process [5]. Today, editing of audio clips has no risk as the user can always revert their changes. Multiple tracks together form an arrangement where each track can be recorded, edited and processed individually.

Today, DAWs are not only used in recording studios or at home, but also on stage, where many people listen to the audio output of the DAW, making failures of the audio playback crucial, as it might hurt the eardrums of the listeners as well as the speakers.

2.1.1 Typical DAW Architecture

A DAW consists of two main parts that are the Graphical User Interface (GUI) and the underlying engine. The engine is the core component that provides features such as editing capabilities, audio rendering, recording, and playback. The architecture of a DAW engine varies, but typically it includes a network of processors, known as an audio routing graph [15], which processes blocks of audio. This audio routing graph is represented in a data model, which contains abstractions such as tracks with dedicated inputs and outputs or plugins.

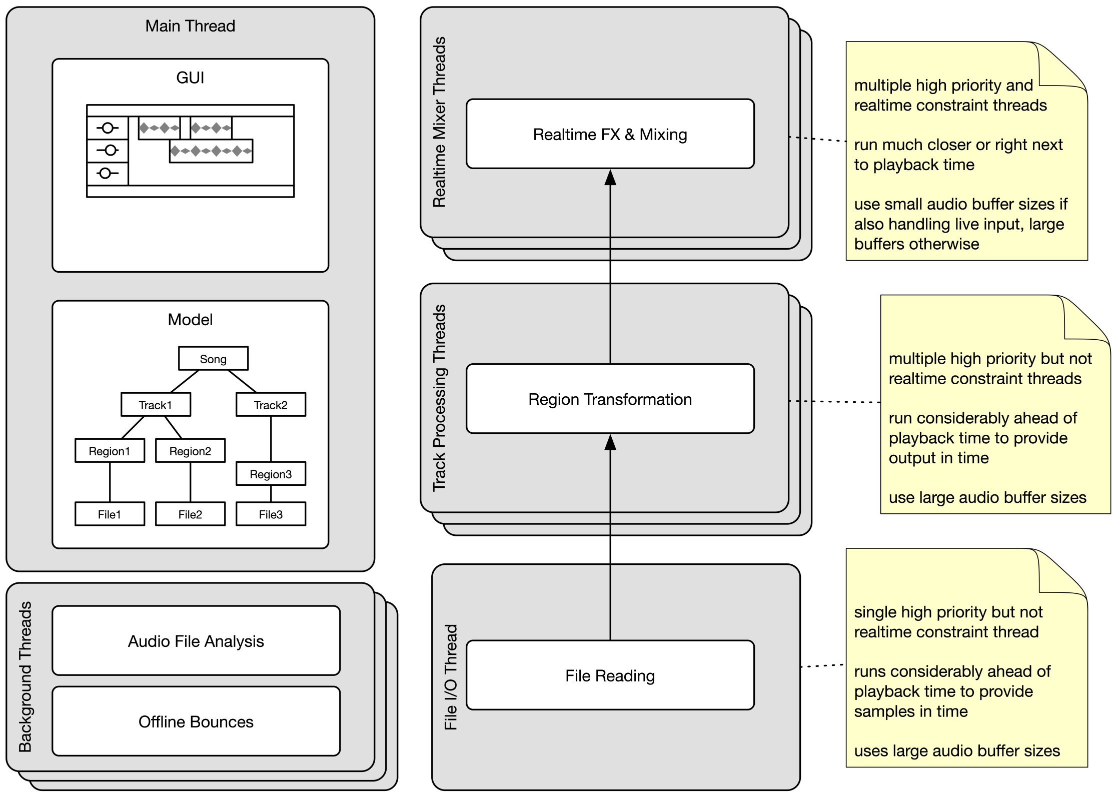

Main Thread
GUI
Model
Song
Track1
Track2
Region1
Region2
Region3
File1
File2
File3
Background Threads
Audio File Analysis
Offline Bounces
Realtime Mixer Threads
Realtime FX & Mixing
Track Processing Threads
Region Transformation
File I/O Thread
File Reading
multiple high priority and realtime constraint threads
run much closer or right next to playback time
use small audio buffer sizes if also handling live input, large buffers otherwise
multiple high priority but not realtime constraint threads
run considerably ahead of playback time to provide output in time
use large audio buffer sizes
single high priority but not realtime constraint thread
runs considerably ahead of playback time to provide samples in time
uses large audio buffer sizes

Figure 2.1: Typical DAW architecture [11], showing which kind of components and their distribution over differently prioritized threads as well as the signal flow from reading audio files to mixing the audio signal

The term *engine* is a vague concept, where no universally accepted definition or reference architecture can be found in the audio context. However, an example for a typical DAW architecture can be found in the ARA SDK documentation as shown in fig. 2.1. The ARA SDK from Celemony is implemented in various DAWs which allows them to provide a good reference, as the documentation is used by developers working on DAWs with ARA support. The architecture also shows the distribution of the components over different threads, as multithreading is an essential part of DAWs. The architecture of a DAW, as described in the ARA SDK, consists of a GUI and data model, which run on the main thread. The audio routing graph is processed in one or multiple real-time threads. A file reading component runs on a separate, non-real-time thread, continuously reading and transferring audio chunks into the audio processing thread for audio playback, as well as receiving audio chunks for audio recording and writing the chunks onto hard disk. This is also referred to as disk streaming [20]. Additionally, there are various non-real-time threads for processing edited audio clips, as well as background threads for audio file analysis and offline bounces as these can be CPU intensive tasks, which should not disturb the GUI and the audio processing. The GUI needs to be highly responsible and in sync with the audio engine state. To keep the GUI responsive, expensive computations are offloaded to other threads. The same applies to the audio processing, which has real-time constraints, so expensive processing is running in separate threads, too.

2.2 Virtual Studio Technology

The virtual studio technology (VST) is a widely used standard that allows the use of virtual instruments, effects and visualizers as audio plugin in a DAW [31]. A plugin is a piece of software that interfaces with another software referred to as host or client, which is the DAW in this case. Before VST, expensive dedicated digital signal processor (DSP) hardware for audio production was used to process the audio signal. Since the VST standard was first released, a countless number of affordable audio plugins were developed and revolutionized the audio production industry. Today audio plugins are an essential part in audio and music production and producers spent a lot of money on audio plugins as well as entire company businesses are based on developing audio plugins.

VST is a representative for one audio plugin specification, next to other audio plugin standards like Audio Unit by Apple or the Avid Audio Extension. Independent of the plugin standard, the fundamental idea of the standards is similar. Standard manufacturers provide specific base classes which C++ objects need to derive from, to encapsulate and create a generic interface to the plugin. The host application only calls functions which are defined by the plugin standard API, which need to be implemented by the plugin.

Audio plugins are delivered in the form of dynamically linked libraries. A dynamically linked library is a compiled version of a software component that can be loaded and executed by another program at runtime. In the context of audio plugins, a dynamically linked library is a compiled version of the plugin that can be loaded by the host. This allows the host to execute the code of the plugin as part of its audio processing chain, while maintaining the modular structure of the plugin and allowing it to be updated or replaced without affecting the rest of the DAW.

To support multiple plugin standards, several plugin frameworks like VST GUI, JUCE or the open source framework iPlug2 exist. They allow compiling the same C/C++ code base to the different plugin standards, making the development process more approachable, as this provides support for multiple plugin specifications and simplifies the development process by allowing plugins to be used in various hosts.

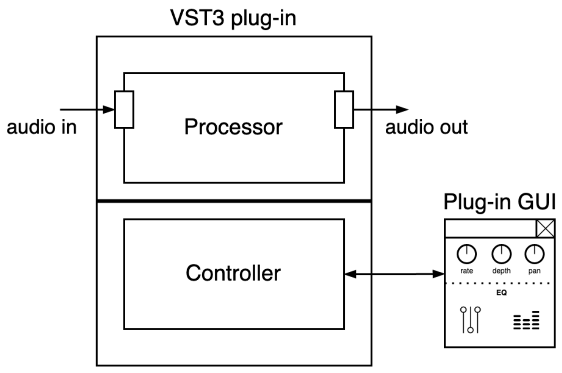


Figure 2.2: VST3 - Dual component architecture, adopted from [31]

The VST standard splits a plugin into two main components, as illustrated in fig. 2.2. The processor refers to the processing of audio signals within the plugin. The VST standard defines a process function that the plugin must implement to perform audio processing. This function is called by the host and is responsible for processing the audio signal and modifying it based on the plugin parameters. The controller refers to the part of the plugin that interacts with the user, typically by providing a GUI. The controller is responsible for displaying the parameters, allowing the user to adjust them, and displaying the results of the processing component. The controller communicates with the host to receive parameter changes and other events. [31]

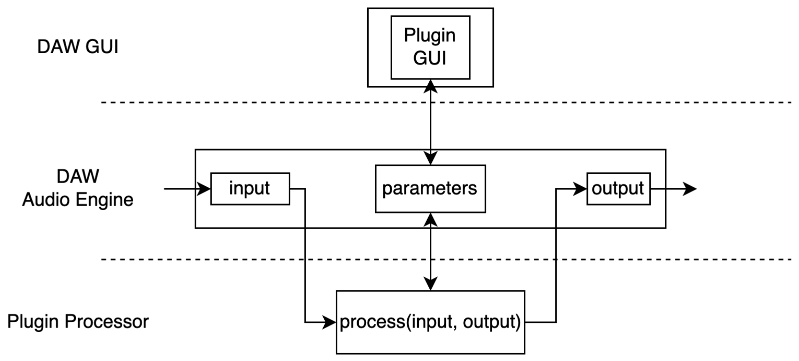


Figure 2.3: Host - plugin communication, adopted from [31]

In fig. 2.3, a general illustration is shown how a host application is connected with a plugin. First, the input audio needs to be transferred to the plugin by calling the process-function of the plugin in the plugin processor. In the process-function, the plugin processor needs to read parameter changes which are transferred from the DAW GUI thread into the plugin processor, because the GUI of the plugin is attached to the DAW GUI. Likewise, the plugin processor can also transfer parameter changes back to the DAW GUI thread. [31]

2.3 Digital Audio

To record an audio input signal in a software, a physical sound wave needs to be converted to an analog electric signal first, for example converting the sound wave of a voice with a microphone. After that, the analog signal needs to be converted into a digital electric signal (see fig. 2.4). Just like that, a digital electric signal needs to be converted to an analog electrical signal and then to a physical sound wave, when audio is played back (see fig. 2.5).

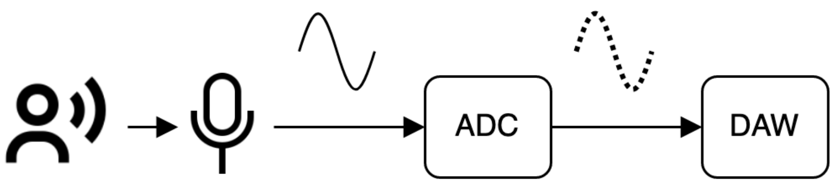


Figure 2.4: Signal flow - recording

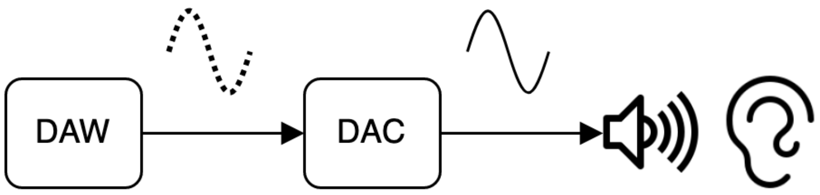


Figure 2.5: Signal flow - playback

2.3.1 Sample Rate

A sound card converts an audio input signal into a digital representation of the audio signal. The input signal is digitized by the analog digital converter in the audio interface. For that, the signal is sampled with a specific sample rate, considering the Nyquist-Shannon sampling theorem. The Nyquist-Shannon sampling theorem states that the sampling rate must be at least twice the frequency of the original audio signal [31].

As an example, audio CDs use a sampling rate of 44,1 kHz. A digital analog converter then must be provided with an output value every 22.7 μs to produce a signal in audio CD quality. If a computer program is not able to provide the value in time, audio dropouts appear. This makes the time to output the samples a real time constraint.

2.3.2 Audio Buffers

To avoid dropouts, audio buffers are used, to provide the audio interface with multiple audio samples at once instead of each sample individually. For example, a buffer of 1024 audio samples at a sampling rate of 44,1 kHz would need to be provided to the audio interface only around every 23,2 ms. Increasing the buffer size decreases the load on the CPU as well as the risk that dropouts happen because of unexpected scheduling. Alongside with this, buffering comes with the downside, that it produces latency. The example audio buffer with a size of 1024 samples and a sampling rate of 44,1 kHz creates a latency of $1024/44,1 = 23,2$ ms. So the latency with a buffer size B and a sampling rate $f_{sampling}$ can be described as:

$$t_{latency} = \frac{B}{f_{sampling}}$$

The audio buffers transferred between audio interface and audio application are also named **audio block** in the context of audio processing.

2.3.3 Latency

Latency in audio applications plays a role in different aspects. The round-trip-latency is important for monitoring, for example when a guitarist uses the DAW as a guitar amp for practicing or recording. The latency needs to be low to not distract the musician. In this case, small audio buffers must be used, to achieve low latencies. Depending on the instrument, smaller or larger latencies are acceptable. The amount of latency acceptable for the human ear varies depending on the musical context, but latency should be smaller than $10ms$ for interactive musical applications [29].

Another aspect appears when an instrument is recorded. The latency between playing the instrument and writing the incoming audio data into memory needs to be known by the application, to correctly align the recorded audio data with the audio that is audible while the musician played. This process is called latency compensation [37]. Latency compensation is not only important when recording but also when real time processing is happening. When single tracks produce latency because of internal buffering in plugins this latency needs to be compensated to align different tracks correctly or synchronize with visualizers while playback [1].

2.4 Concurrency

Multithreading is a crucial component in the architecture of DAWs, as demonstrated in section section 2.1.1. Because of the real time constraints when working with audio, it is important that the code running on the audio thread is efficient. For that, only deterministic and fast code should be executed in the audio thread. Operations like reading from disk need to be done in another thread than the audio thread. This in turn means that audio data has to be transferred to the audio thread, which usually requires the use of locks or mutexes to avoid race conditions. These mechanisms are non-deterministic, so they might lead to the audio thread blocking for an unknown time frame. To solve this problem, lock-free data structures are used for sharing data between the audio thread and other threads. A lock-free single producer, single consumer (SPSC) queue, also known as a ring buffer, is a data structure used for sharing data between the audio thread and other threads. It has two basic operations: push and pop (or enqueue and dequeue). This data structure works well when only one thread calls the push-operation and only one thread calls the pop-operation on the ring buffer. The queue has two pointers, one for the head (read pointer) and one for the tail (write pointer) of the data structure. Both, read and write pointer, need to be accessed by the producer and consumer threads simultaneously. For this, the access to the pointers need to be atomic operations. Atomic operations provide low-level synchronization operations to make sure that there is an enforced ordering when two threads access the same memory location [41]. Atomic operations are also referred to as indivisible, which means that the operation cannot be half-done and cannot be interrupted by other operations, as illustrated in fig. 2.6. The operation cannot be interrupted as the operation acquires a lock, updates the value and releases the lock.

```
void atomic_operation(value) {
  lock_mutex()
  current_value = get_current_value()
  new_value = current_value + value
  set_new_value(new_value)
  unlock_mutex()
}
```

Figure 2.6: Illustration for atomic operations in pseudocode

Instead of using a mutex for ensuring indivisibility like the pseudocode suggests, atomic operations use hardware-level instructions that are executed directly by the processor, without involving the operating system kernel. This results in a significant speed-up compared to using mutexes.

2.5 Audio in Web Browsers

In this section, the web browser technologies relevant for this work are explored, starting by explaining what a web application is and then explaining the Web Audio API, followed by web workers and communication, web storage and WebAssembly.

2.5.1 Web Applications

A web application is a distributed system. Many distributed applications are divided into three layers: The user interface layer, the processing layer, the data layer. A typical web application is a client-server application with a 3-tier-architecture, as shown in fig. 2.7. A web server is passing requests from a client to the application server, which processes the requests and may also need to interact with another database server. According to Van Steen and Tanenbaum, a current trend is to keep the client side thin and handle processing and storage on the server side, while the client side only represents the user interface and at most takes over a minimal part of the processing.

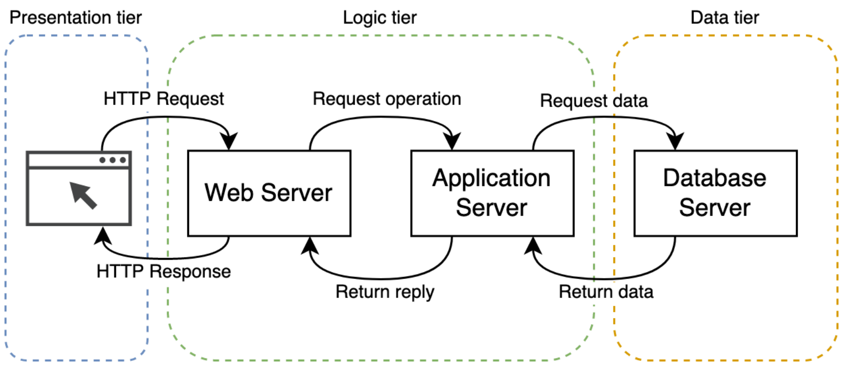


Figure 2.7: Illustration of a typical 3-tier architecture with a web server passing request from a client to the application server and the application server requesting data from a database

Browser-based audio applications on the other hand tend to do the exact opposite by taking over the application layer and large parts of the database layer. The client side needs to keep large parts of audio file data (recorded audio or samples for instruments) on local disk and do the audio processing locally to keep the audio latency low. The server side then is responsible to persist the audio data in a database.

JavaScript

The main programming language used when building applications for the web browser is JavaScript. JavaScript is dynamically typed and allows multiple programming paradigms like object-oriented and functional programming. The biggest impact on audio programming is the memory management of JavaScript. JavaScript uses a garbage collector to free memory automatically, which is very different to systems languages like C/C++ where memory needs to be allocated and freed explicitly. JavaScript had a reputation for being slow in the past due to it being originally purely interpreted [7]. However, JavaScript engines of today use Just-in-Time (JIT) compilers. JIT compilers compile the frequently executed parts of JavaScript code, called hot paths, into machine code at runtime, which results in much faster execution than pure interpretation. Nevertheless, in some cases, the JIT compiler can fail to optimize a specific part of code, and the interpreter will be used instead [13].

Web APIs

JavaScript code can call a set of Web APIs, which form most of the functionality of web browsers. Web APIs are part of the web browser implementation in systems programming languages like C and C++. Web APIs are standardized through collaboration between major web browser vendors which results in cross-browser compatibility, as the APIs are documented in a specification format for implementation by programmers that implement the APIs [21]. The majority of Web APIs used in web browsers are standardized by the World Wide Web Consortium (W3C). Exceptions are draft implementations for APIs. At the time of writing, an example for a draft implementation is the WebUSB API[1], which is only implemented in Chromium based browsers like Google Chrome or Microsoft Edge.

2.5.2 Web Audio API

Web Audio API allows creating an audio routing graph in JavaScript to play and manipulate sounds. The audio routing graph is represented by the **AudioContext**. The graph is build by a set of **AudioNode** objects (see example in fig. 2.8). Each node can have inputs and outputs to connect to other nodes to build the graph. The Web Audio API has a set of built-in AudioNode classes that are ready to use for developers, which can be source nodes to access the audio input device or generate sound with an oscillator node, intermediate effect nodes like a compressor or delay to modify the sound, or output nodes to output the audio with the audio output device [32]. The developers can use these built-in audio nodes as building blocks for audio applications.

[1]https://wicg.github.io/webusb accessed August 5, 2022

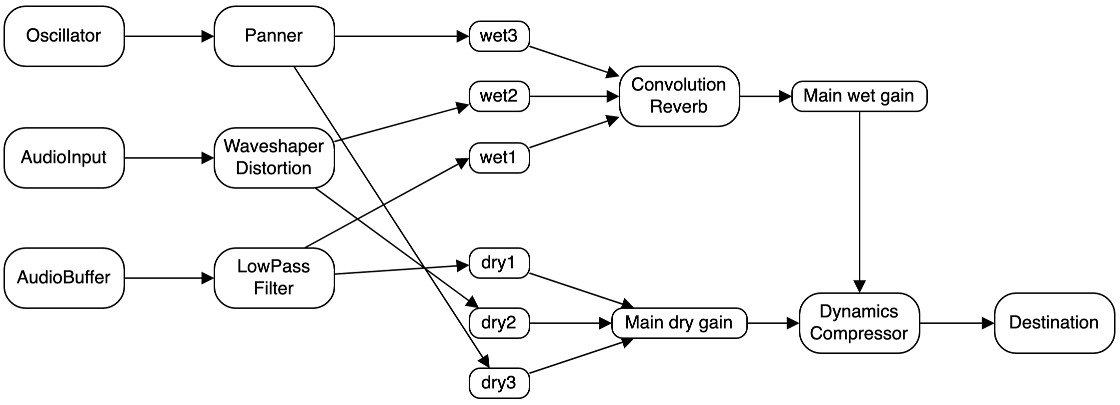


Figure 2.8: Example audio routing graph consisting of built-in AudioNode objects with multiple source nodes, where wet and dry are GainNode objects

The audio context is living on the main thread, while the internal processing of the audio nodes happens on a dedicated real-time thread. At the time of writing, the audio processing of the Web Audio API uses a fixed buffer size of 128 samples per channel. The Web Audio API specification refers to the buffer size as render quantum, as the processing of the audio routing graph is referred to as rendering.

Playback audio

The most conspicuous example for playing back audio on a web page is the HTMLAudioElement. Figure 2.9 shows the GUI of the HTMLAudioElement in Google Chrome. The HTMLAudioElement can stream audio files from disk and the network. It can be combined with the Web Audio APIs audio routing graph by referencing a MediaElementAudioSourceNode instead of an audio file. While the HTMLAudioElement can be controlled by users interacting with the GUI, there is no interface to programmatically synchronize multiple instances of a HTMLAudioElement to use it for playing multiple audio tracks.

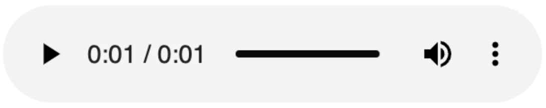


Figure 2.9: HTMLAudioElement in Google Chrome

Precise audio scheduling in the Web Audio API is done by using source nodes that inherit the AudioScheduledSourceNode interface. This can be an OscillatorNode for generating a periodic waveform, a ConstantSourceNode to output a single sample value or the AudioBufferSourceNode to play back audio buffers. The AudioScheduledSourceNode interface allows sample accurate scheduling of audio sources [16].

The AudioBufferSourceNode takes a reference to an audio buffer. The playback of the audio data can be scheduled with the start-method, which then plays the audio once at the given time. If the loop-attribute is not set, the AudioBufferSourceNode instance is destroyed after playing out the audio. Then for every newly played audio data, a new AudioBufferSourceNode instance needs to be created. Concerning the use of the AudioBufferSourceNode for applications like a DAW, the MDN documentation for the AudioBufferSourceNode states the following:

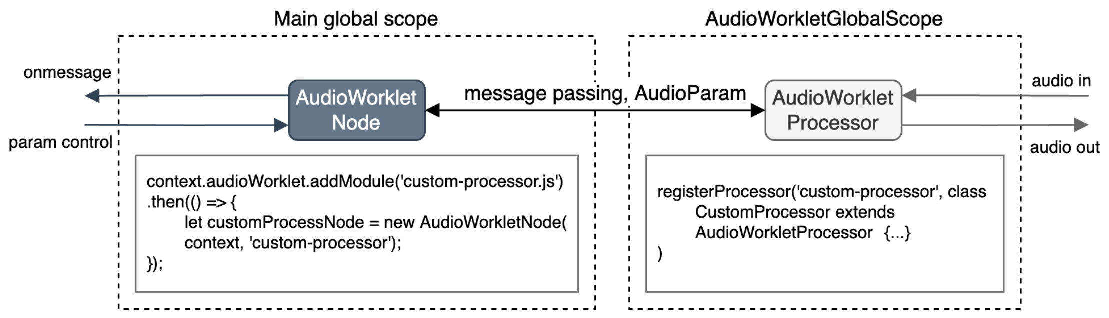
> This interface is especially useful for playing back audio which has particularly stringent timing accuracy requirements, such as for sounds that must match a specific rhythm and can be kept in memory rather than being played from disk or the network. To play sounds which require accurate timing but must be streamed from the network or played from disk, use a AudioWorkletNode to implement its playback.

[14] This means, according to MDN, the AudioBufferSourceNode is explicitly suited to be used for samplers or even a metronome, but not for a DAW, where larger audio files need to be played from disk. Chapter 3 still presents DAW examples which also use AudioBufferSourceNode for playing back audio files.

Instead of the AudioBufferSourceNode, the AudioWorkletNode should be used. The AudioWorklet was first introduced in 2018 as the successor of the ScriptProcessorNode. The ScriptProcessorNode was a way to inject custom audio processing code into the audio graph. For that, an audio callback could be registered to an audio event that is regularly fired, where the event contains buffers for input and output that could be read and written in the callback [17]. The disadvantage of this system is, that the audio data needs to cross the thread barrier from the main thread into the audio thread, for which reason the ScriptProcessorNode is deprecated today. As an alternative, there is the wish to execute audio processing code directly in the audio thread. This is available to the web through worklet interfaces. Worklets provide a way to insert hooks with custom code into the rendering pipeline, which might run in a different thread. Another example for a Worklet than the AudioWorklet is the PaintWorklet that runs paint functions in a dedicated thread [19].

Today, the AudioWorklet interface allows creating custom audio nodes to execute audio processing code in the audio thread. This allows programs for example to manipulate audio input or generate audio directly in the audio thread.

For that, a class deriving the **AudioWorkletProcessor** interface needs to be implemented and registered to the application as a separate module, which is illustrated in fig. 2.10. To instantiate a node with underlying custom processing, an **AudioWorkletNode** needs to be instantiated with the name of the previously registered AudioWorkletProcessor. As any other AudioNode, an AudioWorkletNode can then be inserted into the audio routing graph. The audio processing can be equipped with custom AudioParam parameters as well. The global execution context where AudioWorkletProcessor implementations are defined is called **AudioWorkletGlobalScope**. Custom processor implementations need to be registered in this context.



```
context.audioWorklet.addModule('custom-processor.js')
.then(() => {
        let customProcessNode = new AudioWorkletNode(
        context, 'custom-processor');
});
```

```
registerProcessor('custom-processor', class
        CustomProcessor extends
        AudioWorkletProcessor  {...}
)
```

Figure 2.10: Relationship between the AudioWorkletNode in the main global scope and the AudioProcessorNode in the AudioWorkletGlobalScope, inspired by [32]

Each AudioContext on the other hand has its own AudioWorklet, where the custom processor script needs to be loaded, which is shown in the code listing in fig. 2.10. Similar to web workers that are presented in section 2.5.4, AudioWorkletNode and AudioWorkletProcessor can communicate with each other through asynchronous message passing.

The AudioWorklet enables to implement audio processing code and execute it directly in the audio thread, similarly as the VST plugins paradigm described in section 2.2. This makes the Web Audio API more extensible, as everything that can not be done with the built-in nodes, can potentially be done by using the AudioWorkletNode.

2.5.3 Storage

Compared to native applications, web browsers lack direct access to the file system. There is a W3C draft implemented in Chrome browsers named File System Access API to allow reading, writing or saving files. However, at the time of writing, it is unlikely that this Web API will be available in other web browser engines as well[2]. With the Web Storage API and Indexed DB API, the web browser offers a standardized way to store larger objects on hard disk as key-value pairs, while the size of the storage varies, depending on the hard-drive size of the user as well as the web browser engine implementation. While the Web Storage API allows a synchronized way to store and load hard-disk data, IndexedDB API is designed for large amounts of storage and asynchronous access.

2.5.4 Web Workers

JavaScript itself is a single-threaded language, so it does not provide language-level threading APIs [6]. For that reason, web workers are provided by the Web Worker API, which allows programs to run JavaScript scripts in background threads. A web worker thread does not share states with the main thread or other web workers [36]. Web workers live in their own context called WorkerGlobalScope, which runs its own event loop, while the main thread is sometimes referred to as main global scope [32]. Web workers can only communicate with other workers and the main thread over an asynchronous message system. Unlike concurrency libraries in system programming languages, web workers cannot set the priority of the thread where the JavaScript code is executed.

Message Passing

Web workers can communicate with the main thread and other threads over message passing with methods postMessage and onMessage. A postMessage call copies the message payload and triggers a onMessage event on the receiving end. Objects can be sent using the structured clone algorithm[18] used by postMessage and onMessage internally. However, this only works if the object has no functions. Function objects would otherwise have to be serialized beforehand. The message passing system also distinguishes between transferable and non-transferable object. ArrayBuffer objects are transferable, while

[2] https://mozilla.github.io/standards-positions/#native-file-system

SharedArrayBuffer objects are not. When a transferable object is sent with postMessage, the ownership of the object is transferred over to the receiver of the message.

```
const worker = new Worker('worker.js');

worker.onmessage = (msg) => {
  console.log('message received from worker', msg.data);
};

worker.postMessage('message sent to worker');
```

Figure 2.11: Example usage of a web worker

2.5.5 SharedArrayBuffer

To share memory between the main thread and web workers without message passing, JavaScript provides the SharedArrayBuffer object. Section 3.2 points out, that there is a certain speed-up using SharedArrayBuffer over message passing due to garbage collection, as well as the structured clone algorithm being unnecessary for most audio use-cases. Objects can also be shared over SharedArrayBuffer, by serializing objects into a string, although the message passing system with its structured clone algorithm is going to be faster and safer than serializing objects to strings and writing them to buffers. For that reason, low level data like TypeArrays should be preferred to be shared with SharedArrayBuffer. To synchronize between threads, atomic operations are accessible through the **Atomics** object [24], which guarantees total ordering and enables the implementation of lock free data structures. In combination with the Atomics object, SharedArrayBuffer can be used as backing storage for a ring buffer, like it is introduced in section 2.4, to communicate over thread barriers without blocking the involved threads. The SharedArrayBuffer object itself is not a transferable object as introduced in section 2.5.4. Instead, the object is copied when sending it via postMessage. The copy of the SharedArrayBuffer object then still references the same block of memory as the original SharedArrayBuffer object.

2.5.6 WebAssembly (Wasm)

WebAssembly is a standardized binary code format developed by a W3C community group[3], used to run programs in a stack-based virtual machine [39] that operates similar to the Java virtual machine. It can run in a dedicated runtime or in a web browser's JavaScript engine, and is not restricted to any specific hardware or platform. It is designed to be efficient, fast, and secure. Its small instruction set allows for code optimization for native-like speed, and the binary format is optimized for size and quick loading.

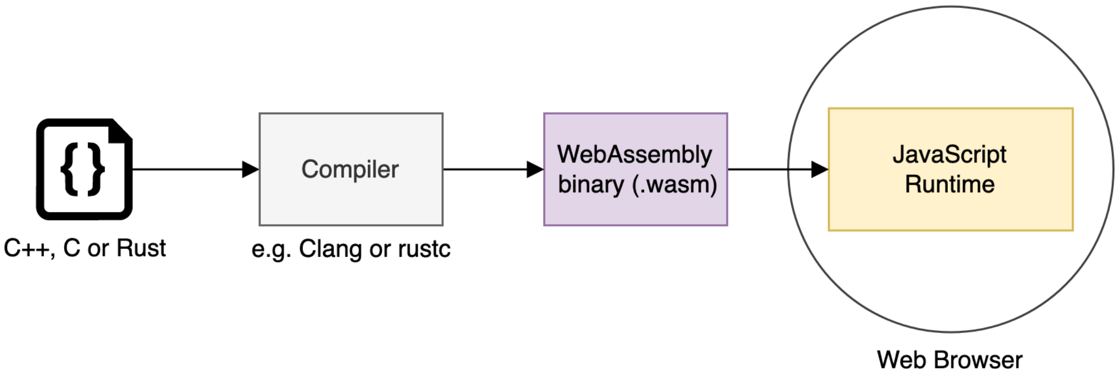


Figure 2.12: Process of compiling code into WebAssembly for the use in the Web Browser

WebAssembly is widely supported across major browsers and can also be used in web servers for its security and portability benefits today, while the initial goal of WebAssembly was, to bring high-performance code into the web browser [40]. WebAssembly has a text representation, known as the WebAssembly Text Format (WAT), with syntax similar to assembly languages. However, WebAssembly is specifically designed to be a compilation target for various programming languages such as C, C++, Rust, or Go and is not intended for being programmed with the text format [22], as illustrated in fig. 2.12.

Rust

Rust is a programming language that aims to achieve memory safety by eliminating undefined behavior while maintaining performance levels similar to C or C++. For

[3]https://www.w3.org/community/webassembly/

that, the Rust compiler checks for forbidden memory accesses such as dangling pointers, multiple frees, and dereferencing of null pointers. Additionally, while the compiler checks array boundaries, Rust also uses run-time checks to prevent buffer overruns. Unlike a C program, a Rust program does not crash if an array is accessed outside its boundaries. Instead, the program exits gracefully with an error message. The same mechanisms that guarantee memory safety also ensure that concurrent Rust code is free of data races. For this, the compiler either knows that the data shared among threads does not change, or it insists on using synchronization primitives such as mutexes, condition variables, and atomics. [9]

Use-cases

Three main use-cases for using WebAssembly in the web browser that are discovered through the course of the work on this thesis that align with the high-level goals defined for WebAssembly [38]:

1. Implement an entire web application in a language other than JavaScript, such as C++ or Rust.
2. Porting an existing code base to the web, which may be implemented in systems languages like C++ or Rust and is then embedded into a JavaScript / HTML application.
3. Use a JavaScript frontend and only build specific parts of the application in WebAssembly to gain performance.

The first and second use-case are similar. The author of a new web application might use Rust to implement the whole application, using crates like web_sys to use Web APIs directly in Rust code [22]. A tool called wasm-bindgen[4] then can generate the binding and JavaScript glue code needed to embed the whole codebase into a JavaScript or HTML file. When an author wants to use an existing Rust codebase on the other hand, porting the code directly may not be possible if the codebase uses libraries that do not target the web. In that case the logic of the codebase needs to be rewritten using the web_sys crate.

[4] https://github.com/rustwasm/wasm-bindgen

Another popular approach for the first and second use-case is to use a compiler toolchain called Emscripten[5]. Emscripten allows compiling an existing C++ code base for the web browser, creating bindings and glue code to integrate the WebAssembly module with JavaScript. There is a high chance that application authors do not need to implement JavaScript code by themselves. They might need to provide a C++ API for a specific JavaScript library that they want to use, which Emscripten needs for the compilation process.

This work concentrates on the third use-case, to speed up the audio processing of a JavaScript application using the tools that the Rust language provides.

Minimal Audio Example: Communication between JavaScript and WebAssembly

The following example, inspired by the code provided in Paul Adenots AudioWorklet workshop at the Web Audio conference 2019[6], illustrates the communication between JavaScript and WebAssembly in a simplified form.

Figure 2.13 shows how a Wasm module is instantiated by using the WebAssembly API. For that, the compiled Wasm module is fetched from JavaScript code.

```
WebAssembly.instantiateStreaming(fetch('example.wasm'))
    .then(wasm => {
        wasm.instance.exports.example_function();
    });
```

Figure 2.13: Instantiation of a Wasm module

For audio, buffers need to be copied into the memory of the Wasm module. A Wasm module has a linear memory in the form of a ArrayBuffer object. For that fig. 2.14 shows how the JavaScript side needs to allocate memory on the Wasm modules' memory in lines 3 and 4. For audio processing algorithms that need an input and generate an output, the JavaScript side needs to allocate memory for an input buffer as well as for an

[5]https://emscripten.org/
[6]https://github.com/padenot/wac-19-audioworklet-workshop

output buffer. For that, the memory needs to be allocated first by calling an allocation function that needs to be implemented and exported by the Wasm module. After that, the JavaScript side can create views on the allocated memory as shown in lines 5 and 10. The result of the process function can then be copied into the buffer of an output audio channel by using the outBuf variable.

```
1  const size = 128;
2
3  let inPtr = wasm.instance.exports.alloc(size);
4  let outPtr = wasm.instance.exports.alloc(size);
5  let inBuf = new Float32Array(
6      wasm.instance.exports.memory.buffer,
7      inPtr,
8      size
9  );
10 let outBuf = new Float32Array(
11 wasm.instance.exports.memory.buffer,
12     outPtr,
13     size
14 );
15
16 wasm.instance.exports.process(inPtr, outPtr, size);
```

Figure 2.14: Allocation of buffers for processing in a Wasm module

2.6 Web Audio Modules

Web Audio Modules (WAMs) is the attempt to have an open web standard for audio plugins that run in the web browser. The idea is to have a similar standard as VST3, which allows compiling for the web in audio plugin frameworks and use plugins in host applications. The latest version Web Audio Modules 2[7] was presented on the Web Audio Conference 2021, which is the version used in this thesis. As fig. 2.15 illustrates, a

[7]https://github.com/webaudiomodules

WAM can have both a GUI and a processor component, similar to the dual component architecture of a VST3 plugin.

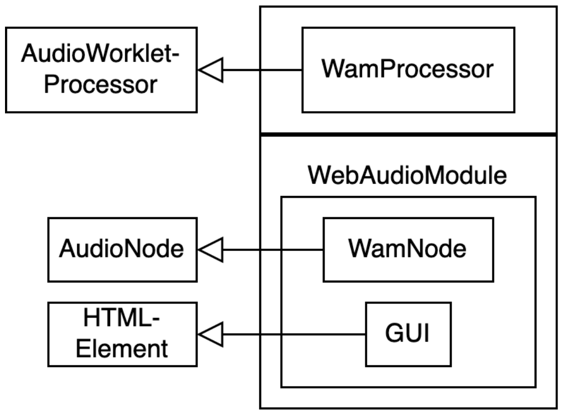


Figure 2.15: WAM - Dual component architecture similar to VST3

When a WAM is instantiated by the host application, the WAM needs to be inserted into the Web Audio APIs audio routing graph. Unlike native audio plugins which always implement a process-function, a WAM can also be a composition of existing AudioNode instances [10], such as the built-in audio nodes described in section 2.5.2. When a processor-component is used, WAMs provide a SDK[8] which is using the AudioWorklet and SharedArrayBuffer to build a similar communication between host application and plugin as described for native VST plugins.

The developers of the WAMs standard had various use-cases in mind. One is prototyping plugins, where developers could rapidly iterate on both the audio processing algorithms and the GUI in the browser, as JavaScript allows for live reloading in the same tab where the plugin is being developed. Another is to enable users to test plugins directly on the vendor's website for marketing purposes. The final use-case is to use WAMs as plugins in online DAWs WAMs are already used in common development tools such as the FAUST Online IDE[9], which lets you make plugins into WAMs, and the C++ framework iPlug2, which supports making WAMs but only version 1, which is not used in this project.

[8]https://github.com/webaudiomodules/sdk
[9]https://faustide.grame.fr/

3 Related work

This chapter starts by introducing browser-based DAWs that successfully use web audio technologies in production. As this work focuses on the performance of the web platform for audio, work from other individuals on performance evaluations is covered. At last this section presents software that was part of the research process and highly influenced the outcome of this work followed by conclusions that are drawn from these.

3.1 Browser-based DAWs

This section introduces the advanced, commercial web DAWs Soundtrap, BandLab and Amped Studio, as well as the open-source research project wam-openstudio. All of the commercial DAWs support conventional audio and MIDI tracks and plugins in some form.

Soundtrap

Soundtrap is the first popular browser-based DAW that was build on top of the Web Audio API in 2013 [2]. Soundtrap was acquired by Spotify in 2017 [25]. The source code of Soundtrap is not open to the public, but the Web Audio panel[1] in Google Chrome allows to inspect the audio routing graph of the current browser tab. As fig. 3.1 shows, Soundtrap seems to use only AudioBufferSourceNodes for playback, when loading an audio file into a track and does not use the AudioWorklet yet. Soundtrap also seems to use the MediaStream Recording API for recording, as a MediaStreamAudioDestinationNode is displayed in the Web Audio panel (not included in the sceenshot).

[1] https://web.dev/profiling-web-audio-apps-in-chrome/

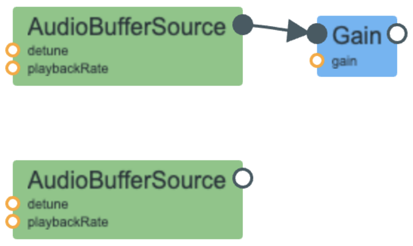


Figure 3.1: Screenshot of Web Audio panel in Google Chrome displaying the audio routing graph for one track playing an audio file in Soundtrap

Soundtrap has built-in instruments and audio plugins, but is not open for third-party plugins. The plugins seem to be implemented as compositions of the built-in Web Audio API nodes.

Soundtrap has some restrictions. Soundtrap explicitly prevents users to create more than 96 tracks. The user can also not activate monitoring on the Windows platform because, according to Soundtrap, the latency is too high on Windows. It is indeed a common perception that Windows systems without running ASIO drivers[2] may have higher latency compared to macOS.

BandLab

The DAW developed by BandLab Technologies uses AudioWorklet for playing back audio files but has strict limitations when it comes to audio file playback on multiple tracks. At the time of writing, BandLab can create a maximum of 16 tracks and can add files with a maximum size of 120 MB. Recording seems to be done without a MediaStreamAudioDestinationNode, so it is probably done with the AudioWorklet.

Amped Studio

Amped Studio uses AudioWorklet for playback as well and has support for internal Web Audio Modules, a lot of them implemented in C++ and compiled for the web [8]. Unlike Soundtrap, Amped Studio can create more than 96 tracks. Amped Studio has a web

[2] https://manual.audacityteam.org/man/asio_audio_interface.html

store where users can purchase plugin licenses and it has support for real VST3 plugins[3]. For that to work, Amped Studio provides a host application, that users must install on their local machine, that is then capable of loading the VST plugins and communicating with Amped Studio.

wam-openstudio

A rather recent research project, wam-openstudio[4], was presented at the Web Audio Conference 2022[5] by Michel Buffa, who supervised the project and is also one author of the Web Audio Modules 2.0 standard introduced in section 2.6. The work shows how WAMs can eliminate the latency when scheduling parameters and MIDI events directly from the audio thread without using AudioParam provided by the Web Audio API. While their project focuses on parameter automation, the authors also use the AudioWorklet for playing back audio files for achieving sample accurate looping [8]. At the time of writing, their system transfers decoded audio files completely into the audio thread using postMessage. This means, their system keeps all audio data in the AudioWorklets memory, which significantly increases the overall memory use of the system. Additionally, the repository description states, that the project uses C++ together with Emscripten for audio processing. The use of C++ primarily involves copying input buffers to output buffers. The purpose of using C++ in their work is not specified, but it may be for performance improvement or simply to demonstrate technical capabilities. However, the fact that the wam-openstudio project exists shows that the Web Audio community is interested in sample level access to audio files.

3.2 Web audio performance evaluations

This section presents papers and blog posts that include various evaluations regarding the Web Audio API and tools to implement these. These studies, which were conducted in recent years, cover a range of topics including audio latency measurements, the use of web-based tools for profiling real-time audio workloads, the implementation of wait-free ring buffers for the web and how this can be tested with a stress testing tool.

[3] https://ampedstudio.com/how-to-use-vst-plugins-inside-amped-studio/
[4] https://github.com/TER-M1/wam-openstudio
[5] https://wac2022.i3s.univ-cotedazur.fr/

The Viability of the Web Browser as a Computer Music Platform [42]

The first evaluation of web audio technologies was created in 2013, shortly after the first Web Audio API version was released. The article focuses on the Web Audio API, but also covers technologies like WebRTC and JackTrip for collaborating over the network via peer-to-peer. The evaluation of the Web Audio API includes an audio latency measurement of Google Chrome compared to a native application, where the native platform is especially superior over the web browser on Windows systems. The article closes with naming two major "pain-points" of using the web platform for music applications. These points are the susceptibility to glitches caused by user interface or garbage collection events, as well as the lack of extensibility. As this work was created long before the AudioWorklet, these points might be no longer valid.

Karpluss-Strong-Stress-Tester

The Karpluss-Strong-Stress-Tester[6] is a web page developed by Jack Schaedler[7], a Software Developer at Ableton. The web page simulates guitar strings by using the Karpluss-Strong-Algorithm[8]. The user can configure how many strings are played simultaneously and can choose if a single AudioWorklet is used for processing a single instance or if one AudioWorklet processes 100 instances of the algorithm. The user can also choose between JavaScript and WebAssembly as processing language. The web page optionally shows a visualization of the simulated guitar strings based on the data provided by the AudioWorklet processing. For that the user can choose between message passing or SharedArrayBuffer to transfer the data between the AudioWorklet and the main thread.

A wait-free single-producer single-consumer ring buffer for the Web

At the time of writing this thesis, Mozilla Developer and Web Audio API specification editor, Paul Adenot, published a blog post about his library rinbuf.js [4], which provides a wait-free single-consumer single-producer (SPSC) ring buffer for web applications using SharedArrayBuffer and the Atomics object, like introduced in chapter 2. The library especially offers higher-level abstractions for audio applications. Besides

[6]https://jackschaedler.github.io/karplus-stress-tester/
[7]https://jackschaedler.github.io/
[8]https://ccrma.stanford.edu/~jos/pasp/Karplus_Strong_Algorithm.html

an introduction of the ringbuf.js library, the published article includes a performance comparison between postMessage and SharedArrayBuffer. For that, the author uses the Karpluss-Strong-Stress-Tester from section 3.2 to generate audio load as well as main thread load for visualizing the guitar strings. The author runs the guitar string simulation on Linux as well as macOS in Firefox and Google Chrome to compare postMessage with SharedArrayBuffer. The author increases the number of simultaneously simulated guitar strings until he observes audible glitches. The findings of this comparison are, that the SharedArrayBuffer increases the load capacity 2.5 times to 6 times over using message passing.

Comparing approaches for new AudioWorklets

This paper by Gerard Roma was presented at the Web Audio Conference 2022 and it is comparing the performance of different approaches for creating AudioWorklets using JavaScript, C++, and AssemblyScript [33]. The author of the paper has implemented a sawtooth oscillator, sine oscillator, and low-pass biquad filter in each of the different languages and has run an experiment to measure the performance by counting the number of worklets that can be created before the audio starts glitching. The author also compares his own implementations with the OscillatorNode and the BiquadFilterNode implemented natively in the Web Audio API. The results show that Google Chrome has the best performance with native AudioNodes, while in Firefox the JavaScript implementations perform better than the native ones. The author points out that the JavaScript implementation holds up against WebAssembly and argues, that the communication cost of WebAssembly needs to be considered and that the WebAssembly implementation is way more complex than the JavaScript or AssemblyScript implementation. However, the author does not further analyze what causes the high communication cost of WebAssembly and does not show a WebAssembly implementation that is faster than the other implementations.

3.3 AudioStore - storage and streaming

AudioStore[9] is a library designed to save large audio files to several blobs into IndexedDB and stream smaller chunks of the audio file, to reduce the memory footprint of audio ap-

[9]https://github.com/kevincennis/AudioStore/tree/master/lib

plications which use the AudioBuffer objects. The library also provides a Streamer class and a StreamCoordinator class. The Streamer class can stream a single audio file using the AudioBufferSourceNode object. The StreamCoordinator handles and synchronizes multiple Streamer objects. The Streamer and StreamCoordinator maintain the current playback position and provide functions for pausing and seeking. The library contains a demo which plays multiple synchronized audio tracks, each playing back a single audio file.

3.4 CPAL (Cross platform audio library)

CPAL[10] is a library that allows developers to play and record audio on various platforms including Linux, Windows, macOS, iOS, Android, and the web. The library is implemented in the Rust programming language and uses the Emscripten or the wasm-bindgen library to generate WebAssembly code. When targeting the web, the library makes use of the web_sys crate, which provides low-level bindings to the Web APIs. Through the use of web_sys, the library interacts with the Web Audio API. The web implementation of CPAL uses the AudioBufferSourceNode instead of the AudioWorklet to play audio buffers. Examining the CPAL documentation does not reveal that audio buffers are scheduled on the main thread when using the web implementation. This means that users of the library need to have a certain knowledge about how audio playback is working on the web platform to fully understand the implications of using the library.

3.5 Conclusion and impact on this work

In summary, there are different approaches for implementing similar web audio applications. It remains unclear which approach is the best to use for playing audio in a DAW.

The main issue with the presented evaluations in *Comparing approaches for new AudioWorklets* and *A wait-free single-producer single-consumer ring buffer for the Web* is, that audible glitches are detected, so it is not possible to automate the presented processes easily. The paper *The Viability of the Web Browser as a Computer Music Platform*

[10] https://github.com/RustAudio/cpal

presents a latency evaluation with a setup that is rarely documented and unlikely to be automatable, so it can also not be used as a base for the evaluation in this work. As a consequence, chapter 5 presents another approach for measuring the audio performance, that is also based on another blog post from Paul Adenot.

The paper *Comparing approaches for new AudioWorklets* also states "WebAssembly is clearly not a magic bullet that makes code automatically faster" without considering what exactly causes that the WebAssembly implementation is not faster than the JavaScript one. Another article *Calls between JavaScript and WebAssembly are finally fast* indeed presents how the communication between WebAssembly and JavaScript used to be way slower than today [12] but also showed in 2018 already, that this is not the case anymore. This uncertainty regarding WebAssembly motivates, that chapter 5 shows another comparison between WebAssembly and JavaScript for audio processing, considering different use-cases, that show under what circumstances WebAssembly is not automatically faster than JavaScript.

The demo project of the AudioStore library already contains functions which this work aims to implement in the prototype. For that reason, this work does not implement the usage of IndexedDB but uses the AudioStore implementation on the one hand. On the other hand, the disk streaming that is implemented in the library using AudioBuffer-SourceNode is used as comparison for evaluating the disk streaming implementation using the AudioWorklet, that is implemented within this work. A fork of this library[11], which is distributed under the MIT License, is used to realize the architecture presented in chapter 4. Furthermore, the wait-free SPSC ringbuffer library ringbuf.js[12] is used to realize the disk streaming of the prototype.

[11]https://github.com/KnappSas/AudioStore/
[12]https://github.com/padenot/ringbuf.js/

4 Architecture

This chapter shows the engine architecture that is designed to achieve the goals presented in section 1.2.2, as well as serving the evaluation in chapter 5. The components designed for the prototype are presented first by showing the interfaces which connect the components and classes, that are first presented as black boxes and then successively zoom into the details.

4.1 Constraints

The following list of constraints must be considered in the development of the prototype:

- The AudioContext cannot be utilized within a web worker context.
- The AudioStore library has dependencies to the AudioContext in the AudioStore class. As this class needs to be used in web workers, the AudioContext needs to be extracted and the functions used from the AudioContext need to be replaced.
- JavaScript cannot share objects between contexts (workers, worklets), and uses a structured clone algorithm to copy objects before transferring them to other threads, which cannot clone functions. This means that only data can be transferred between threads.
- Communication between the audio thread must be performed with SharedArrayBuffer, as postMessage is producing too much garbage, as pointed out in section 3.2. postMessage can be used for initialization only, but not for ongoing communication.
- The built-in function for decoding audio files, BaseAudioContext.decodeAudioData(), is not available in web workers, as the AudioContext is not available. This means that audio files must be decoded on the main thread before they can be read and processed in web workers.

4.2 Technical context

The technical context of the prototype involves various dependencies that are crucial for its functioning. These dependencies are described in the table below, including the Web Audio API that provides the audio routing graph and audio worklet, the Media Capture and Streams API, the Indexed DB API for client-side persistent storage, and the Web Audio Modules that use the Web Audio Module SDK as well as plugins that offer the WebAudioModule interface. These dependencies are central to the architecture of the prototype, and their integration will be further discussed in the following chapters.

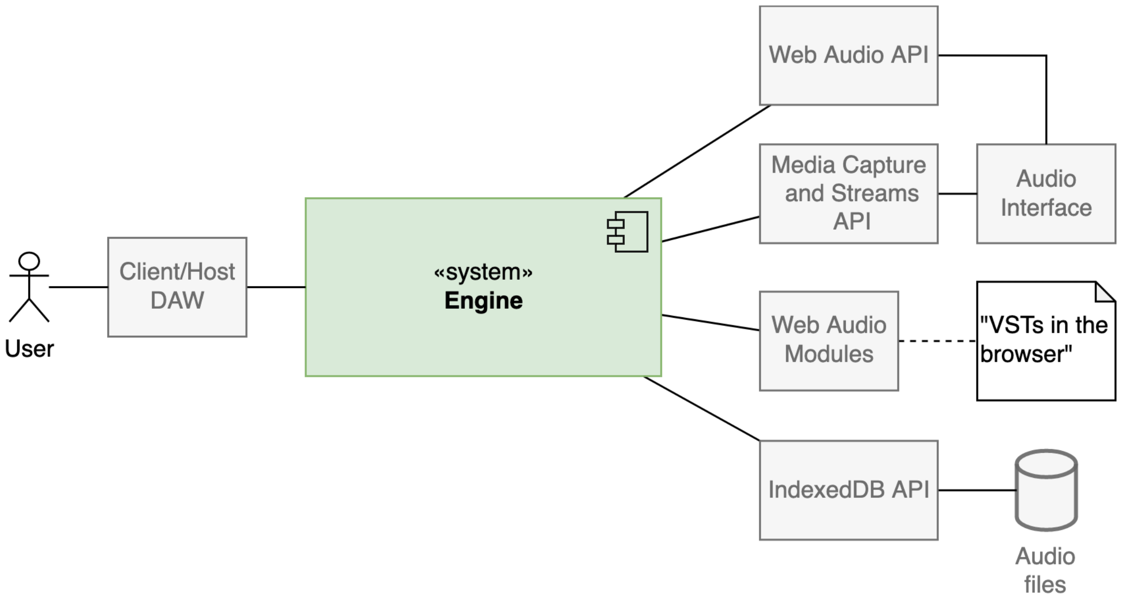


Figure 4.1: Technical context

Dependencies	Description
Web Audio API	The main dependency providing the audio routing graph and the AudioWorklet
Media Capture and Streams API	Provides functions for getting the input audio device to create a MediaStreamAudioSourceNode and for recording audio with the MediaRecorder and the MediaStreamAudioDestinationNode
Indexed DB API	Client-side storage used for storing and reading audio files

Dependencies	Description
Web Audio Modules	Used to implement plugin support similar to VST3 in native DAWs. The system uses the Web Audio Module SDK as well as plugins which implement the WebAudioModule interface

Table 4.1: Technical context Description

4.3 Solution strategy

The Solution strategy section outlines the approaches used to achieve the goals and requirements. It describes how this work addresses the quality goals described in section 1.2.3, as well as how the system needs to be built in such a way that it can be used for the evaluation. Furthermore, this section addresses how the maintainability of the developed system is achieved. Table 4.2 outlines an overview of the actions taken to achieve the quality goals.

Quality Goal	How achieved
No Dropouts	Disk streaming threads use SharedArrayBuffer instead of postMessage() to avoid garbage collection and improve preloading speed. A wait-free SPSC ring buffer library (ringbuf.js) is used to copy audio data from disk to audio processing thread.
Low RAM Usage	Audio data is streamed from disk using small chunks, which are transferred to the AudioWorkletProcessor of a track, rather than scheduling a new AudioBufferSourceNode for each chunk.
Responsive GUI	Audio scheduling is done from a separate disk streaming thread.
Low Latency	-

Quality Goal	How achieved
Interoperability/ Extensibility	Encapsulation - clear boundaries between classes and components, ensure that especially the streaming implementation can be swapped out.

Table 4.2: Solution strategies to achieve the quality goals defined in section 1.2.3

As **dropouts** either happen when disk streaming threads cannot hold up with the audio processing thread or when the load on the audio thread is too high, SharedArrayBuffer is used for streaming instead of postMessage() to avoid garbage collection as well as improving the preloading speed. The disk streaming needs to scale depending on the track count, so for each audio track, a separate SharedArrayBuffer object and an associated AudioWorkletNode is created. A wait-free SPSC ring buffer is used to copy the audio data from the disk streaming into the audio processing thread. For that the library ringbuf.js is used, that is mentionend in section 3.2. The **RAM usage should be minimized** by streaming the audio data from disk by transferring small chunks of audio data over to the AudioWorkletProcessor of a track instead of scheduling a new AudioBufferSourceNode for every chunk of audio data. This should prevent the creation of garbage, as the same objects are re-used for loading the audio data. To **avoid disturbing the GUI** with scheduling audio buffers on the main thread, the audio should be scheduled from another disk streaming thread. **Low latency** can be achieved by keeping the buffer size low. The Web Audio API has a fixed internal buffer size, also referred to as render quantum in the Web Audio API specification. The MediaStream Recorder API might introduce an unknown delay to the overall roundtrip latency and the Web Audio API might also introduce another latency on the audio output path, depending on the Web Audio API implementation. For that reason, the architecture cannot contribute to reducing the latency apart from considering using the AudioWorklet for recording instead of the MediaSteam Recorder API.

In order to ensure the overall quality of the software architecture, the software design principles and patterns found in table 4.3 are applied to the prototype. This should keep the prototype maintainable and also approachable to the reader.

Design Principle	Design Pattern / Techniques
Seperation of concerns	Facade, Handle pattern, factory method

Design Principle	Design Pattern / Techniques
Encapsulation	Handle pattern, proper interfaces, RPC style messaging
Interface segregation	Adapter pattern

Table 4.3: Strategies to achieve software architecture quality

The key consideration is the **separation of concerns** by applying the facade pattern, to keep the complexity of the underlying system away from the client of the engine. To separate the handling of objects within the engine from the clients, the handle pattern is used which is a common design pattern that allows the creation of a reference to an object without exposing the object itself. The handle pattern can be found in code examples in the book *Clean Code* by Robert C. Martin [27], but it is not explicitly discussed or named as such in common design pattern references like the Gang of Four book *Design Patterns. Elements of Reusable Object-Oriented Software*, where the facade pattern can be found also. The same book describes the factory method which is applied to separate the concrete object that is used for streaming the audio from the track. In the case of the developed prototype, the StreamCoordinator class from the AudioStore library serves as the Creator. This makes it also possible that in some future work, another streamer could be implemented inside the AudioStore library, without the need to change the track. Furthermore, the classes of the AudioStore library are treated like a foreign system, even though a fork of the library is used. By applying the adapter pattern, the engine has no dependency to the interface of the StreamCoordinator class.

In addition, proper interfaces are defined between the classes and components to decouple the implementation details from the interface. This also serves the evaluation in chapter 5 as the prototype must be able to switch between the implementation done for this work utilizing the AudioWorklet, as well as the AudioBufferSourceNode implementation which is taken from the AudioStore library introduced in chapter 3. The definition of interfaces also helps to abstract the underlying implementation details of the message passing, that is needed to communicate between the main thread and web workers, which then can then be implemented in a RPC style manner. The remote procedure call (RPC) or RPC pattern [24] is a model to let client-server applications communicate. The goal of this model is to provide transparent distribution [35], allowing local function calls to be indistinguishable from function calls made over a network. This means that the development of a software architecture can be kept similar for both distributed systems and local systems that run in the same process. The same model can be applied when a system is distributed over threads and processes on a local system. Another advantage of the RPC is the improved scalability as the disk streaming might need multiple web workers which can be added as needed.

The principles separation of concern and interface segregation are part of the SOLID principles [28], which are well known principles mostly known from the work of Robert C. Martin (*Clean Code*, *Clean Architecture*)

4.4 Building block view

To differentiate between systems created for this work and third-party systems, fig. 4.3 introduces a legend with colors for the different systems. Third party libraries are marked in blue, systems that are further presented in whitebox views are colored green, Web APIs provided by the web browser are colored in gray.

The system builds on the concept illustrated in fig. 4.2. It shows how the Web Audio API's routing graph is utilized for creating audio tracks. MediaStreamAudioSourceNode is used to get the audio input device, when monitoring is activated. MediaStreamAudioDestinationNode is used to record the input signal of the input node, when recording is activated. The main node of a track is the AudioWorkletNode which is routed into a chain of plugins consisting of WAMNode instances that are exposed by Web Audio Modules. A gain node controls the volume of the track. Another separate gain node is utilized to mute the track if necessary, eliminating the need to model this behavior within the abstracted routing graph. Multiple tracks are mixed with a single gain node before the signal is passed to the destination, which is the audio output device.

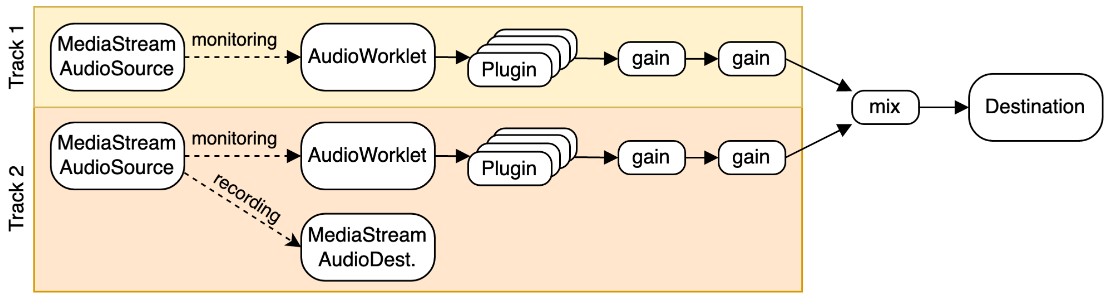


Figure 4.2: Concept for audio routing graph in the engine - dotted lines for recording and monitoring indicate optional nodes

4.4.1 Whitebox overall system

In fig. 4.3 the layered structure of the system is presented. The interface IAudioEngine exposes the functions that serve the requirements presented in section 1.2.2.

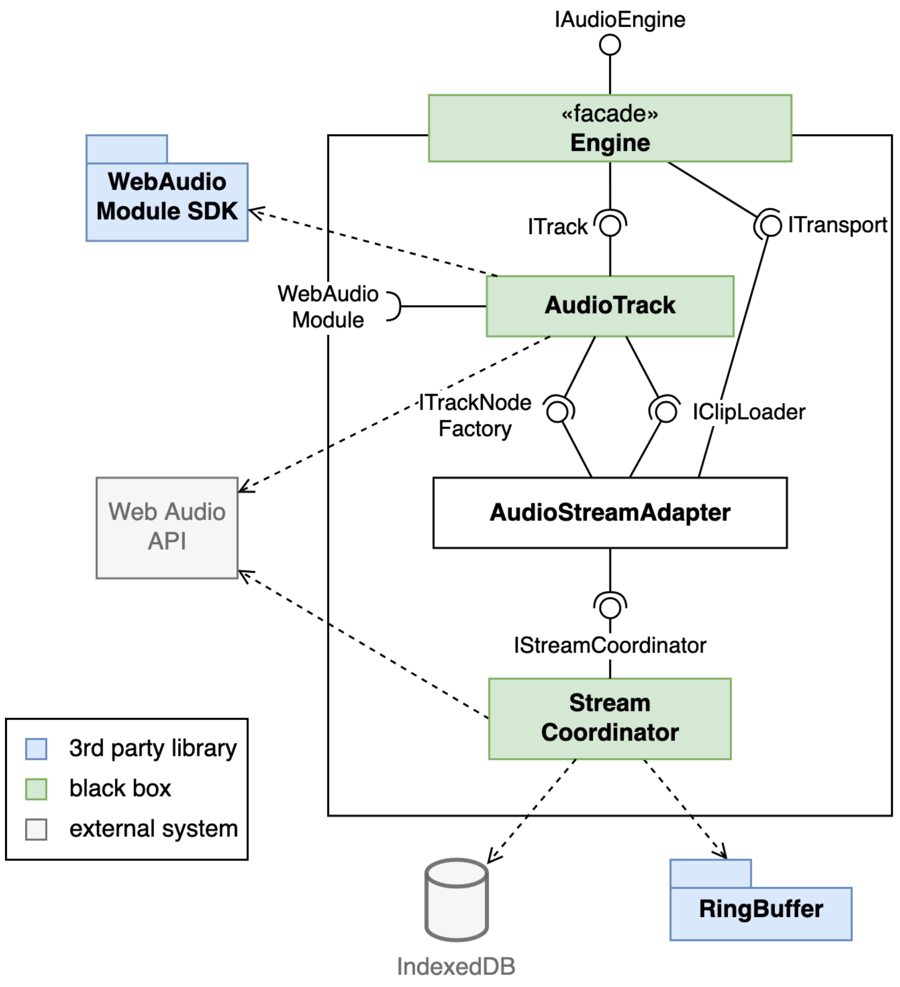


Figure 4.3: Whitebox overall system

Function	Requirement
TrackHandle insertTrack(position)	REQ1
ClipHandle addClipToTrack(filePath, clipPosition, trackHandle)	REQ2
void play	REQ3
void seek(position)	REQ4

Function	Requirement
void pause	REQ5
void insertPluginToTrack(pluginName, trackHandle)	REQ6

Table 4.4: IAudioEngine functions mapped to the requirements

Subsystem	Description
Engine	Facade that provides the interface for the engine. Also responsible for routing the tracks, which would usually be done in a mixer component, which is omitted for this work.
AudioTrack	Main abstraction of Web Audio API nodes for playing back and recording audio and administration of plugins (Web Audio Modules) in a plugin chain
AudioStreamAdapter	Adapt to StreamCoordinator, implements transport interface (ITransport), acts as factory (ITrackNodeFactory), implements interface for loading clips (IClipLoader). All functions are mapped to StreamCoordinator.
StreamCoordinator	Creates and coordinates audio streams, implements transport functions, keeps track of the current playback position, decodes audio files

Table 4.5: Classes and components in the system

4.4.2 Whitebox Track

In fig. 4.4, the whitebox view of the AudioTrack class reveals which parts of the Web Audio API are used, as well as the interfaces that are implemented and used.

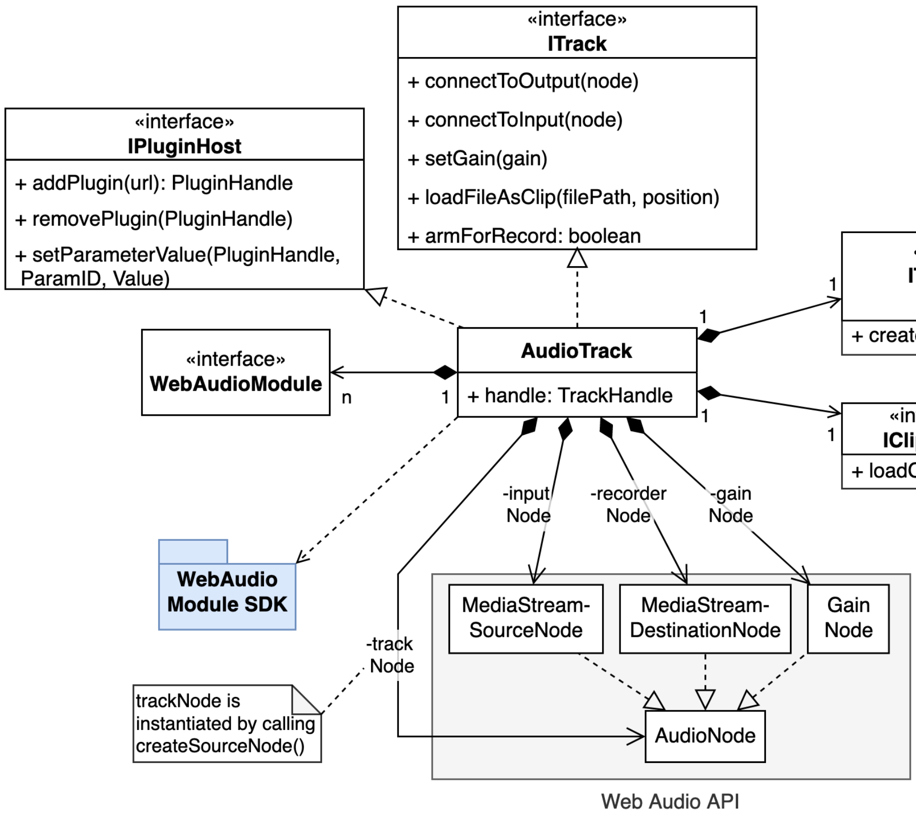


Figure 4.4: Whitebox Track

4.4.3 Whitebox StreamCoordinator

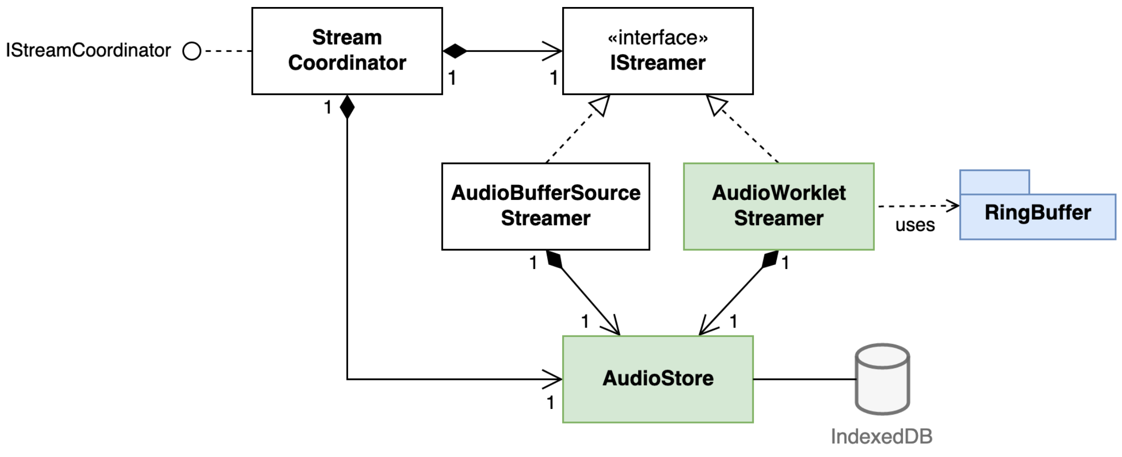


Figure 4.5: Whitebox StreamCoordinator

Subsystem	Description
StreamCoordinator	Class responsible for coordinating streams. Can create streams that implement the IStreamer interface.
AudioWorkletStream	Streams audio from a diskstreaming context into a audioworklet instance assigned to the stream. Audio data is transferred using a ringbuffer.
AudioBufferSourceStream	Schedules AudioBuffer instances using the AudioBufferSourceNode.
Audiostore	IndexedDB creation and entries access

Table 4.6: Classes and components in StreamCoordinator

4.4.4 Whitebox AudioWorkletStreamer

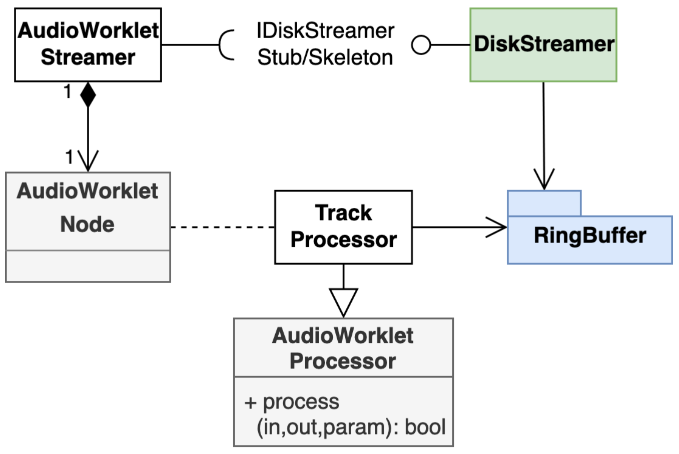


Figure 4.6: Whitebox AudioWorkletStreamer

The AudioWorkletStreamer class uses a stub to transfer calls to the DiskStreamer component, which implements the interface and unmarshalls the received calls. The communication between the two components takes place through message passing and is encapsulated by the stub and skeleton.

Function	Description
StreamCoordinator	Class responsible for coordinating streams. Can create streams that implement the IStreamer interface.
AudioWorkletStream	Streams audio from a diskstreaming context into a audioworklet instance assigned to the stream. Audio data is transferred using a ringbuffer.
AudioBufferSourceStream	Schedules AudioBuffer instances using the AudioBufferSourceNode.
Audiostore	IndexedDB creation and entries access

Table 4.7: Classes and components in AudioWorkletStreamer component

4.4.5 Whitebox DiskStreamer

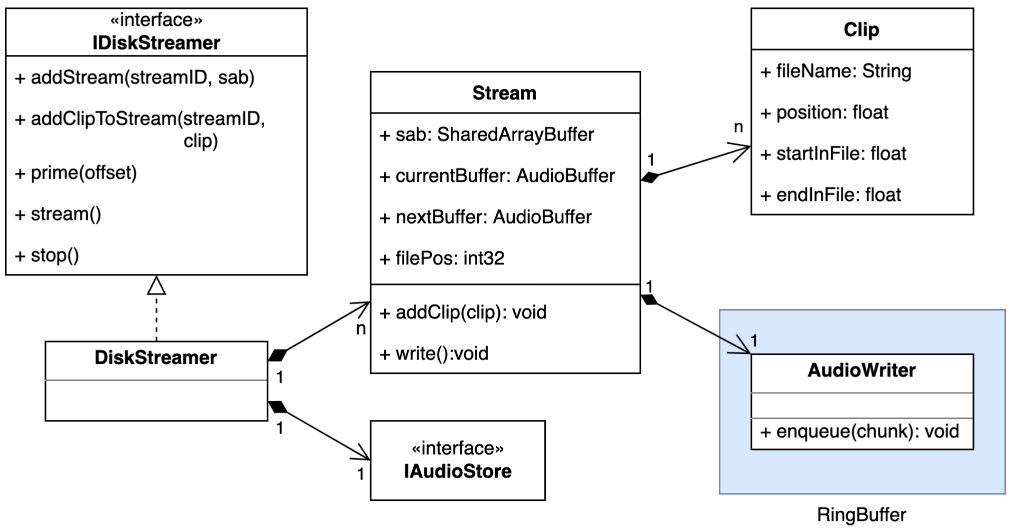


Figure 4.7: Whitebox DiskStreamer

4.4.6 Whitebox AudioStore

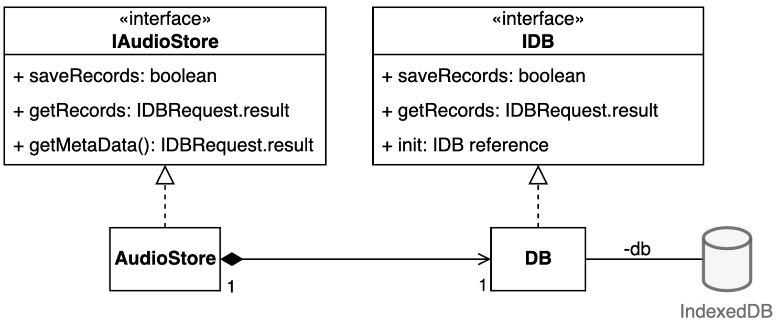


Figure 4.8: Whitebox AudioStore

The AudioStore database has two object stores. The first object store contains metadata in the form that can be found in fig. 4.9. The second object store holds the actual audio chunks in the form shown in fig. 4.10. The channels array contains the audio data blobs for each channel.

```
{
    "name": "audio/sine_-12dB.wav",
    "channels": 2,
    "rate": 44100,
    "duration": 1200,
    "chunks": 240
}
```

Figure 4.9: Meta data object in AudioStore

```
{
    "id": "audio/sine_-12dB.wav-0",
    "name": "audio/sine_-12dB.wav",
    "rate": 44100,
    "seconds": 0,
    "length": 220500,
    "channels": [
        {...},
        {...}
    ]
}
```

Figure 4.10: Chunk object in AudioStore

4.5 Runtime view

The runtime view gives an insight how the presented classes interact. For that sequence diagrams are provided to show the basic functions that are provided by the engine. To illustrate how the disk streaming is working, pseudo code is provided as well as an illustration that shows which routes the streaming is going over the different contexts.

4.5.1 Insert a track

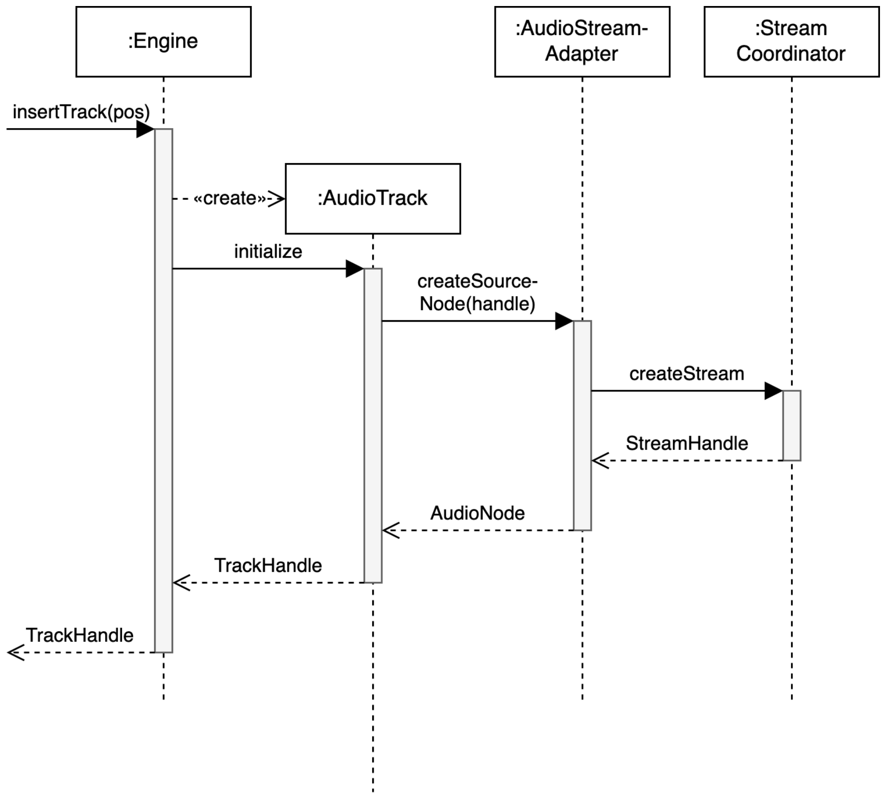


Figure 4.11: Sequence insertTrack

In fig. 4.11, a source node is created for a new audio track. In case of the AudioWorklet, the StreamCoordinator creates the AudioWorkletNode instance and the associated stream, as described in the next sequence. A handle for the stream is stored in the AudioStreamAdapter instance, so that it can map tracks to streams.

4.5.2 Create an audio stream

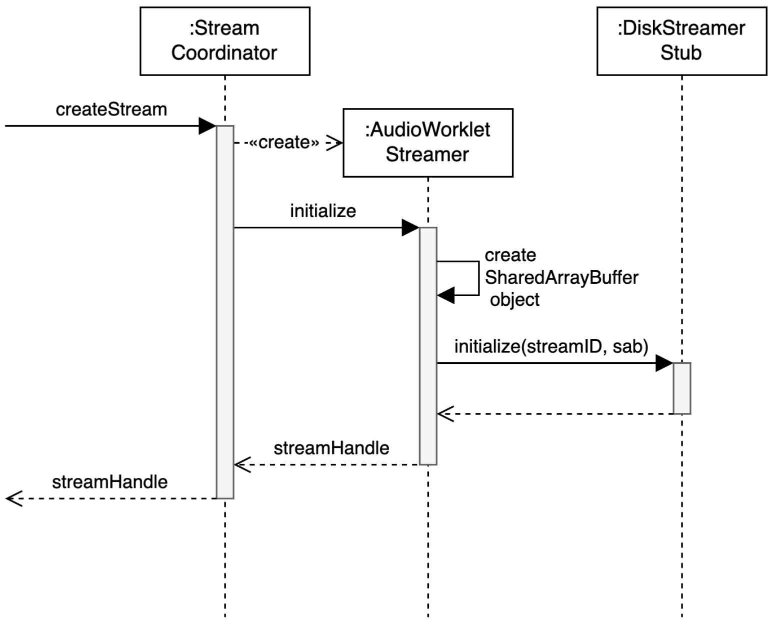


Figure 4.12: Sequence createStream

Figure 4.12 shows, that the StreamCoordinator creates a new AudioWorkletStream, which in turn creates an AudioWorkletNode and a DiskStreamerStub. The DiskStreamerStub allows communication with the DiskStreaming in the web worker. The AudioWorkletStream also creates a SharedArrayBuffer for the DiskStreaming, along with a streamID to identify the SharedArrayBuffer instance with a specific stream.

4.5.3 Load files as clips

When adding a clip to a track, the associated stream is identified by its streamID. Adding the clip model containing metadata and loading the actual audio file is seperated into separate steps. As stated in section 4.1, the file associated with the clip needs to be decoded before it is passed to the DiskStreaming, as the function AudioContext.decodeAudioData() is used, which is not available in the web worker context.

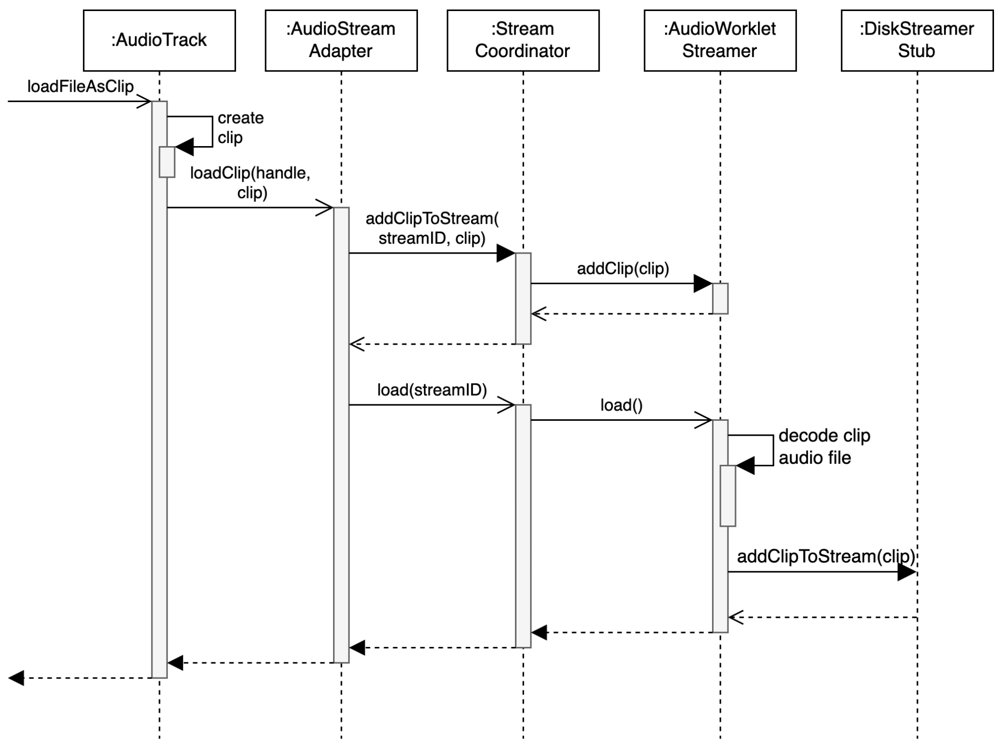


Figure 4.13: Sequence loadFileAsClip

4.5.4 Start audio playback

In fig. 4.14, audio data is preloaded through the call to prime() before streaming begins from the current playback position (elapsed).

4.5.5 Seek

Figure 4.15 illustrates, that when seeking, the playback position is newly set and audio data is preloaded before beginning to stream again.

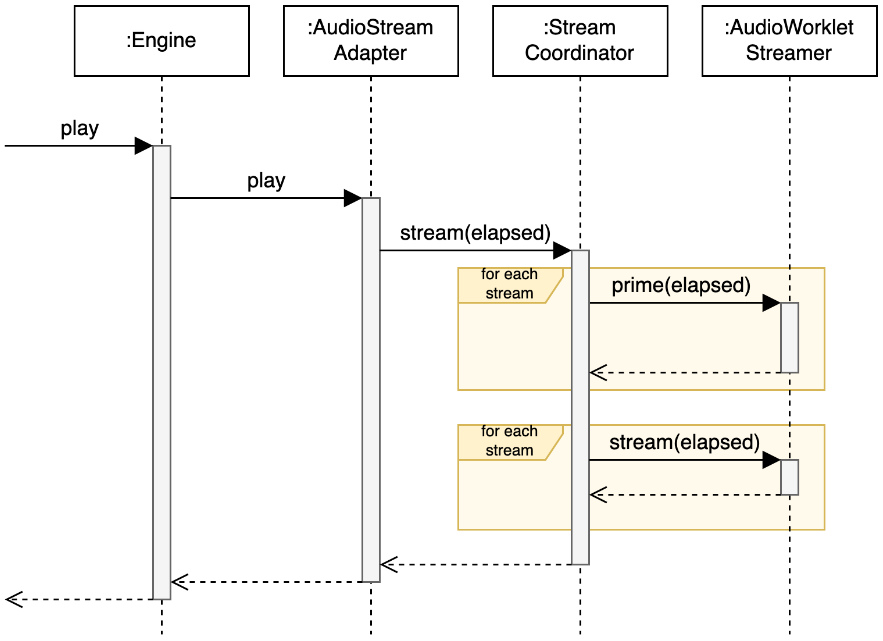


Figure 4.14: Sequence play

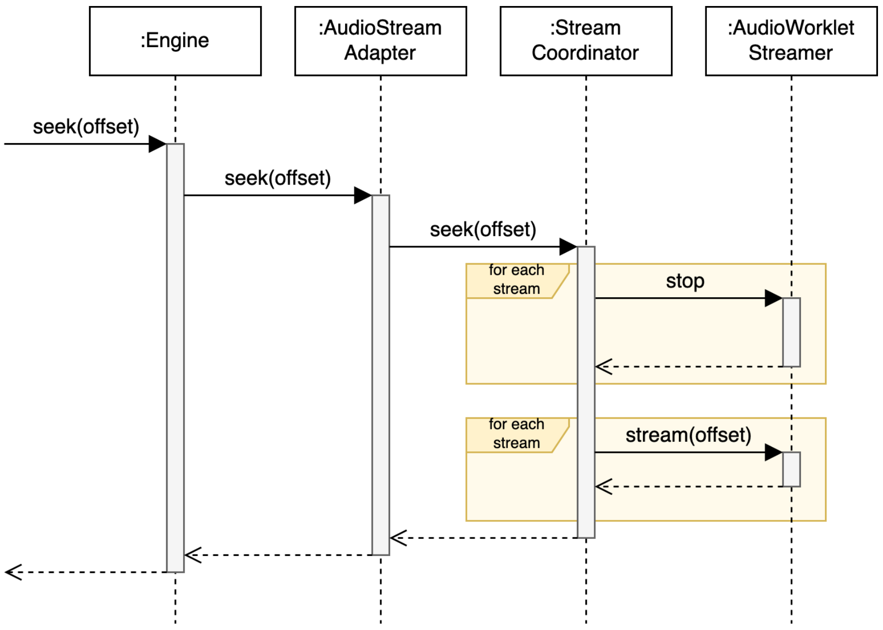


Figure 4.15: Sequence seek

4.5.6 Diskstreamer

The DiskStreamer instance continuously reads audio chunks from the AudioStore and enqueues them into the ringbuffer associated with the stream. A timeout of 10 milliseconds resulted in a satisfactory disk streaming performance.

```
function stream() {
  setRunning(true)
  while (isRunning()) {
    for s in streams {
      while (writeSpaceAvailable(s) and moreDataToWrite(s)){
        chunk = extractChunkFromBuffer(s)
        writeChunkToRingBuffer(s, chunk)
        updateChunkIndex(s) // the current read pos
                            // in the buffer
        if (endOfBufferReached(s)) {
          switchCurrentAndNextBuffer(s) // switch buffers
          getNextAudioBufferFromIndexedDB(s) // async
          resetChunkIndex(s)
          updateFilePosition(s) // pos in the file where
                                //buffers are read from
        }
      }
    }
    waitForTimeout() // wait 10 milliseconds
  }
}
```

Figure 4.16: Disk streaming illustration in pseudo code

4.5.7 Source code locations

The source code for the engine prototype, including the test client, plugins, and the experiment scripts, can be found at the following GitHub repository: https://github.

com/KnappSas/WebAudioEngine. The fork of the AudioStore library can be found at https://github.com/KnappSas/AudioStore.

5 Evaluation

This chapter presents the results of the experiments on the performance of the implemented prototype. The metrics used, and the general experiment setup, are presented first. After that, the two audio playback approaches, AudioWorkletStreamer and AudioBufferSourceStreamer, are evaluated through one experiment. In a separate experiment, the performance of using WebAssembly and JavaScript as audio processing language is compared.

5.1 Metrics

To evaluate the performance of the system, a set of metrics is used that reflect key aspects of audio processing, such as audio callback load, CPU load, RAM usage, and FPS. The individual experiment sections formulate hypotheses about the expected values of these metrics for each approach, based on the understanding and assumptions of their underlying mechanisms.

Audio callback load

The CPU budget that is available to execute audio processing code is the latency between two callbacks from the audio thread, as described in section 2.3.2

$$t_{latency} = \frac{B}{f_{sampling}}$$

The audio callback load then can be defined as

$$load(t) = \frac{audiorendertime(t)}{t_{latency}} * 100$$

where t is the absolute time when an audio callback was called. The load value should be between 0 and 1. If the value is repeatedly greater than 1, the processing on the audio thread takes too long and results in a faulty audio output. This can be influenced by the number of audio channels, the sample rate, and the bit depth of the audio data.

CPU load

The CPU load is a measure of the amount of processing capacity that is being utilized by the system, in this case the test client, which is a child process of the browser. It is generally desirable to minimize CPU load in order to ensure good performance and prevent the system from becoming overloaded.

RAM Usage

RAM usage is an important performance metric to consider when evaluating audio processing systems, as it can impact the overall performance of the system. Higher RAM usage can lead to slower performance, as the operating system may need to swap data in and out of memory to make room for new data. It is generally desirable to minimize RAM usage in order to get better performance.

Frames Per Second (FPS)

The FPS is the number of times that the display of the application is refreshed per second. In the context of audio processing, FPS may not be directly relevant, but it could be indirectly influenced by the CPU load of the audio processing tasks, such as continuously scheduling audio buffers on the main thread.

5.2 Experiment Setup

To run the experiments, a test client is implemented using HTML and JavaScript. The test client allows setting the parameters for the experiments and start audio playback. The Firefox Profiler, which is part of the Firefox Developer Tools, is used to measure the audio callback load in both experiments. The Firefox Profiler was modified by Paul Adenot, a Mozilla Audio Developer and one of the Web Audio API specification editors, to include markers for the execution time of audio callbacks, the current CPU budget, and the time when callbacks happened [3]. These markers can be read by JavaScript code that is executed in the same browser tab as the Firefox Profiler, allowing the computation of the audio processing load. Other browser vendors do not currently have a way to collect and export audio related data, so the experiments are only run using Firefox. In addition to the audio callback load, the overall CPU and RAM consumption of the test client is logged using the Unix tool **ps**, which is called once per second. The frame rate is logged in a JavaScript array and can be exported by the test client itself. Each audio track, that is created with the test client, plays the same audio signal. The volumes of the individual tracks are adjusted to ensure that each track has the same volume and the sum of all track volumes equals the original audio signal's volume. The experiments

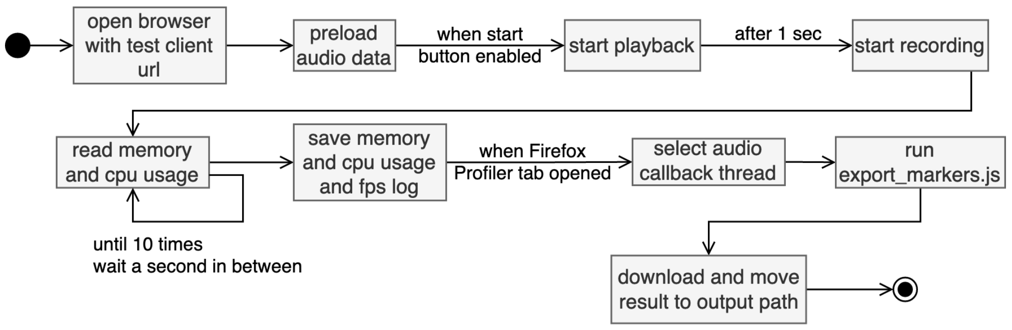

open browser with test client url
preload audio data
when start button enabled
start playback
after 1 sec
start recording
read memory and cpu usage
save memory and cpu usage and fps log
when Firefox Profiler tab opened
select audio callback thread
run export_markers.js
until 10 times wait a second in between
download and move result to output path

Figure 5.1: Activity diagram for a single experiment run

are conducted on a MacBook Pro with a 2.3 GHz 8-Core Intel Core i9 processor and 32 GB of 2667 MHz DDR4 memory, running macOS Big Sur. The onboard audio card is used as an audio output device with a sample rate of 44,1 kHz and two 32-bit float channels. The experiments are automated using Python scripts, utilizing Selenium, a tool for remote controlling web browsers. Selenium is used to start the audio playback in the test client, exporting the fps log, and to select the correct thread in the Firefox Profiler. A single test run is illustrated in fig. 5.1, which always begins with a fresh start of Firefox. The parameters for the individual test runs are not selected with Selenium, as they are included in the URL of the test client, when the browser is opened.

5.3 Streamer comparison

In this section, the two implemented approaches for streaming audio from disk are compared: AudioWorkletStreamer and AudioBufferSourceStreamer. The different streamers were introduced in chapter 4 and the underlying mechanics, AudioWorklet and AudioBufferSourceNode, were introduced in chapter 2. Both approaches stream audio from the local web storage using IndexedDB. The following sections describe the cause variables selected for the experiment and the metrics used to evaluate the performance of the AudioWorklet and AudioBufferSourceNode approaches. They present hypotheses for each of the metrics, based on the expectations of how the different approaches perform.

5.3.1 Experiment design

Table 5.1 and table 5.2 show the cause and effect variables selected for this experiment. The experiment measures the effect variables by running with each type of streamer and different numbers of tracks. The experiment is run 10 times, while every round is running for approximately 10 seconds. The sample size of the audio callback load is given by the duration of the experiment, which makes around 3600 samples. The sample size of the CPU load and Memory are chosen to be smaller to lower the chance that polling the data with **ps** is interfering with the system under test. Similar reasoning applies to the fps, as a significant difference is expected anyway and the experiment should simply show that the expectation is correct. As the sample size is low, the graphics are annotated with a p-value used to determine statistical significance [26]. A p-value higher than 0.05 indicates, that the sample size may not be sufficient and may not accurately represent the truth.

The Firefox Profiler was initially limited to recording approximately 10 seconds, however, a configuration was later discovered that allowed for longer recordings. The results were good enough running the 10-seconds recording, so the experiments are not repeated with longer runtimes. The configuration that allows for longer recordings can be found in the README of the engine repository.

Cause variables
Number of tracks
Type of streamer

Table 5.1: Cause variables for streamer comparison

Effect variables	Sample size
Audio callback Load	∼ 3600
CPU load	100
RAM Usage	100
FPS	∼ 12 to 14

Table 5.2: Effect variables for streamer comparison

The actual input parameters chosen for the experiment are listed in table 5.3. First, the base load of the different streamer types is evaluated by running the experiment with one track. As an example with higher load, 96 tracks are chosen, which is, as explained in chapter 3, the maximum number of tracks that can be created in Soundtrap. First tests also show, that 96 tracks is close to lead to audio tracks getting out of sync with the AudioWorkletStreamer, while 96 tracks is no problem for native DAWs, which easily play back 1000 tracks.

Number of Tracks	Playback node
1	AudioWorkletStreamer
96	AudioBufferSourceStreamer

Table 5.3: Parameters for source node comparison experiment

5.3.2 Hypotheses

One of the expected results is that the AudioWorkletStreamer implementation significantly improves the frame rate compared to the AudioBufferSourceStreamer implementation. This is because the AudioBufferSourceStreamer implementation schedules buffers on the main thread, which can cause significant frame rate drops. Frame rate drops can have negative impact on the user experience and can also indicate overall performance issues of a web application, as many computations happen on the main thread. In terms of audio load, an increase of the load on the audio thread is expected for the AudioWorkletStreamer implementation compared to the AudioBufferSourceStreamer implementation. One reason is that the AudioWorkletStreamer implementation uses a web worker for preloading, which adds an extra layer of processing to the audio processing. This may increase the overall load on the audio thread. Another reason is that the AudioWorklet-Streamer implementation uses a ring buffer to transfer data between the web worker and the AudioWorkletProcessor, which is an extra step for the audio processing. This additional step may also contribute to an increase in audio load. It is also possible that the AudioBufferSourceNode implementation may be more optimized by browser vendors, which could potentially impact the audio load and overall performance of the system. However, it is not possible to directly test this possibility as the experiments are only being run using the Firefox web browser.

5.3.3 Results

The resulting box plots for a single track can be found in the appendix and did not show significant differences in most cases compared to the results with 96 tracks, except for memory consumption. This allows for the focus to remain on significant differences found with the larger number of tracks.

Audio callback load

The results of the experiment show that streaming files with either the AudioWorkletNode or the AudioBufferSourceNode result in no significant difference when run with just one track, as fig. A.1 shows. However, when the experiment runs with 96 tracks, the AudioWorkletNode appears to be more consistent with a lower interquartile range of the box plot, as shown in fig. 5.2, compared to the AudioBufferSourceNode which has a wider range and more outliers. Neither implementation shows exceptional additional overhead compared to the other.

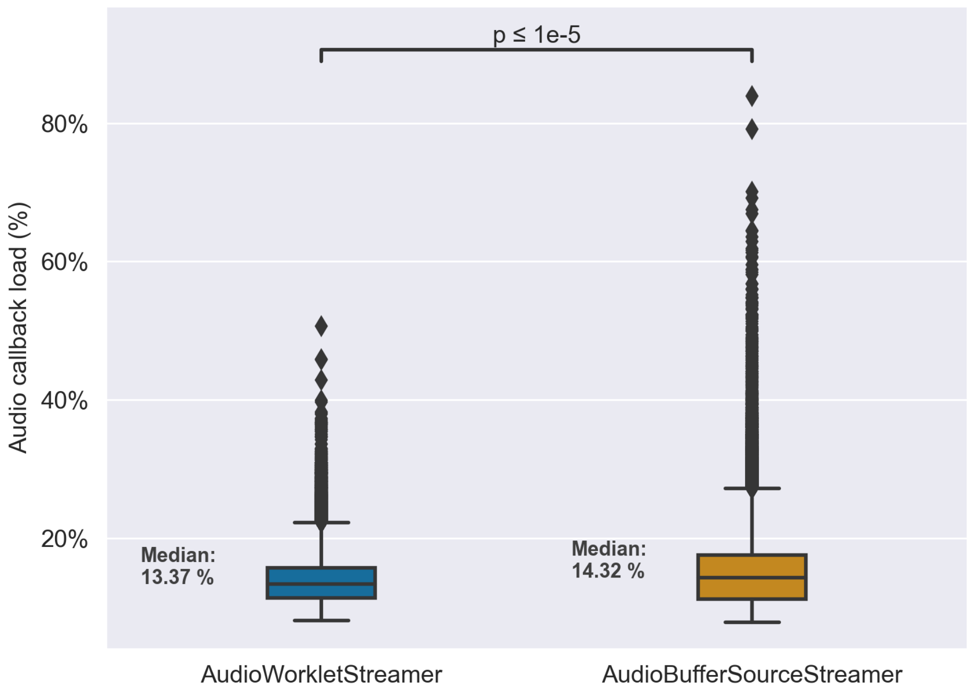


Figure 5.2: Box plot of audio callback load with 96 tracks

FPS

The results of the experiment show that when streaming files with either the AudioWorkletNode or the AudioBufferSourceNode, there is no significant difference in terms of the FPS when run with just one track, as shown in fig. A.2. However, when the experiment is run with 96 tracks, the use of the AudioBufferSourceNode resulted in a significantly lower FPS and even causes significant drops in the frame rate, as shown in fig. 5.3.

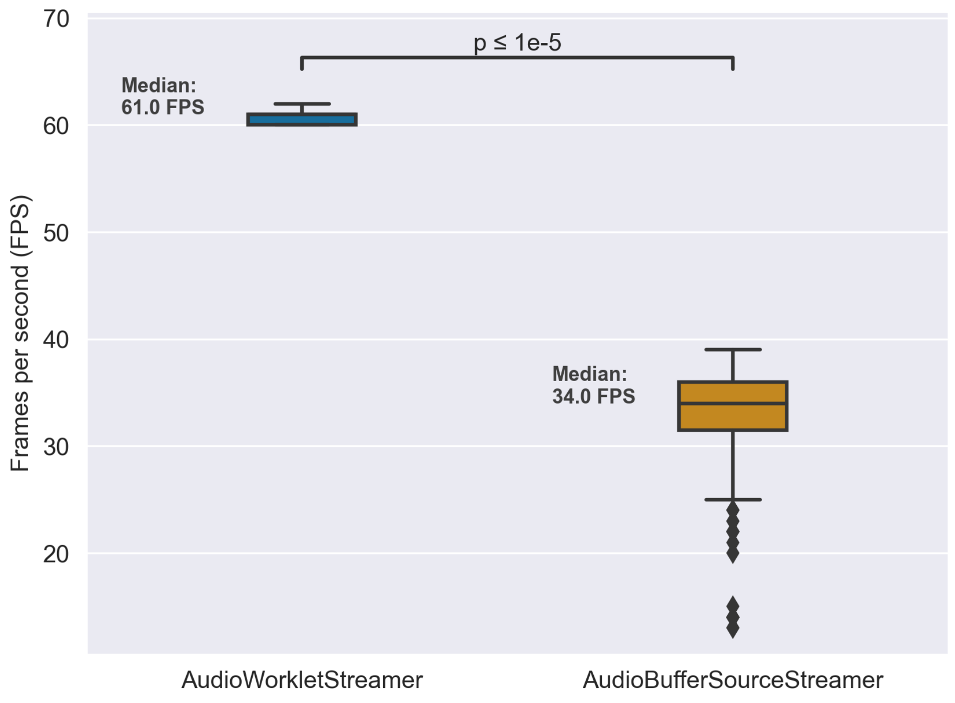


Figure 5.3: Box plot of FPS with 96 tracks

Memory

Comparing the memory consumption of the AudioWorkletStreamer and AudioBufferSourceStreamer, the experiment shows that the AudioWorklet solution has a lower average memory consumption than the AudioBufferSourceNode solution. When running with 1 track, the AudioWorkletStreamer consumes an average of ∼127 MB of memory, as shown in fig. 5.4. The interquartile range is ∼ 24 MB, with ∼118 MB being the first quartile and ∼ 142 MB being the third quartile, while the AudioBufferSourceNode solution consumed an average of ∼120 MB and has the interquartile range of ∼10 MB, the first quartile being 115 $\sim MB$ and the third quartile being ∼125 MB.

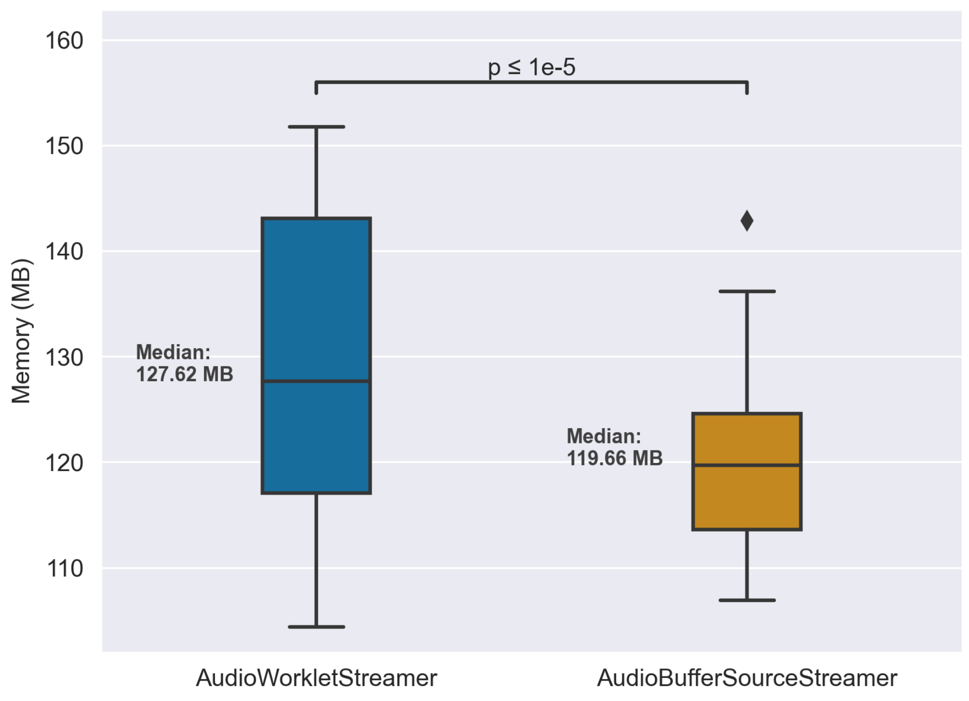


Figure 5.4: Box plot of memory consumption with 1 track

However, when running with 96 tracks, the results show that the AudioBufferSourceNode solution has a higher and widely distributed memory consumption, as shown in fig. 5.5, with an average of 1172.73 MB, while the memory consumption goes up to ∼2500 MB. The AudioWorkletStreamer solution has a stable memory consumption, with a median of 565.56 MB and an interquartile range of ∼100 MB.

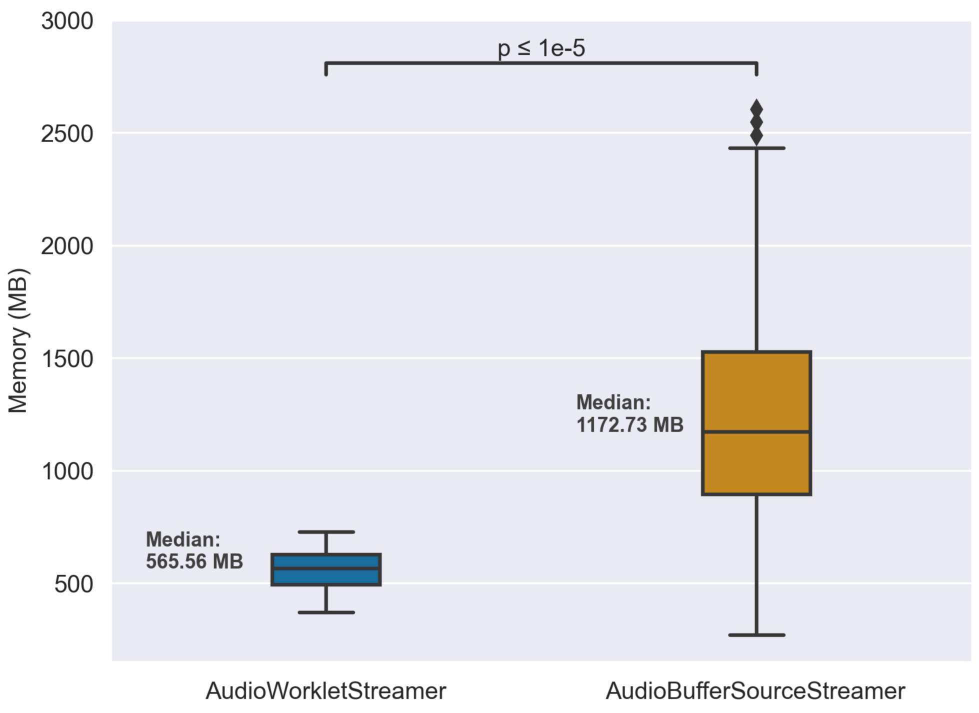


Figure 5.5: Box plot of memory consumption with 96 tracks

The box plot of the AudioBufferSourceStreamer gives no insight, why the memory consumption increases. The line chart in fig. 5.6 shows that the memory of a single test run with the AudioBufferSourceStreamer accumulates over time and is only partially freed during the test run, resulting in higher memory consumption afterwards.

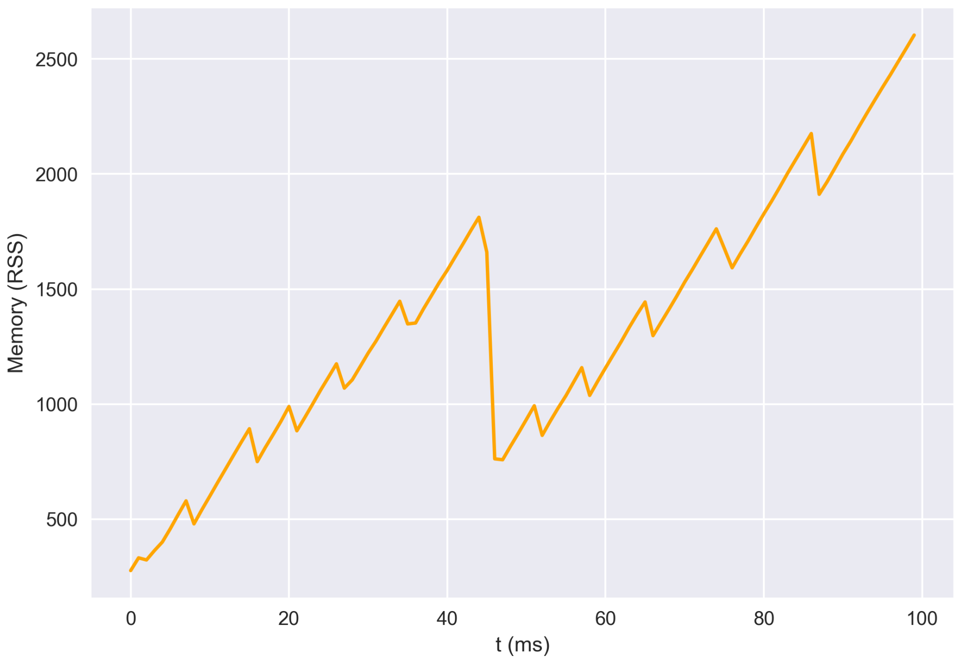


Figure 5.6: Line chart of a single test run showing the memory consumption with AudioBufferSourceStreamer

CPU

The results from the box plot for a single track, as shown in A.3, do not show a substantial difference in CPU load between the two implementations. The box plot for 96 tracks, illustrated in 5.7, also displays no fundamental difference between the two implementations. The p-value of 0.25 also suggests that the sample size may be insufficient for drawing strong conclusions. However, the median of the AudioWorkletStreamer (142%) and AudioBufferSourceStreamer (148%) is relatively similar. The results show that AudioBufferSourceStreamer seems to generate a higher CPU load, as indicated by the upper whisker reaching close to 225%, compared to the upper whisker of AudioWorkletStreamer which is at 175%.

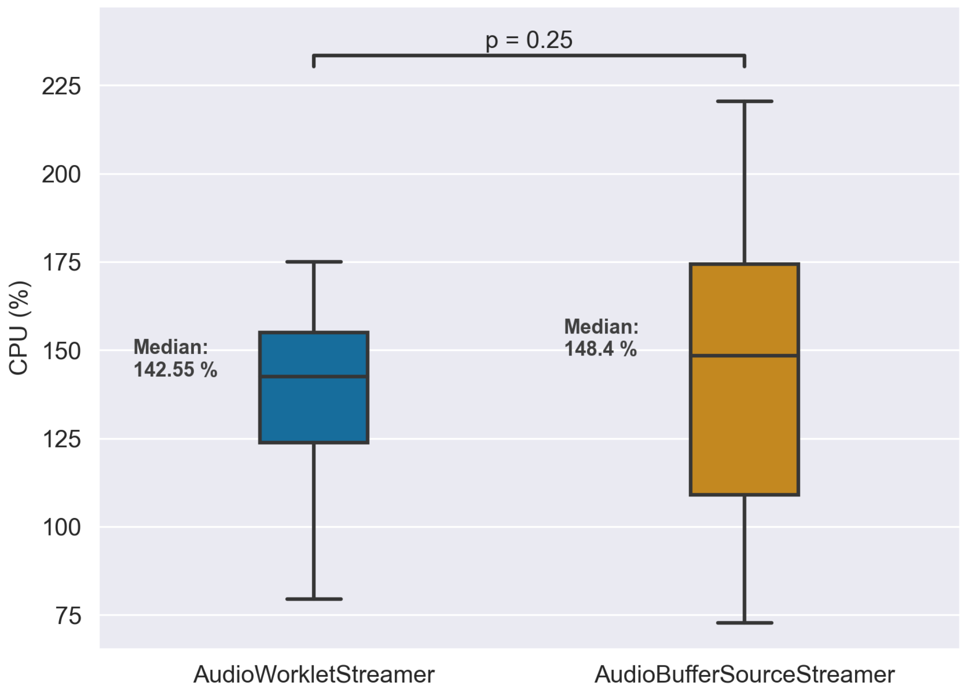


Figure 5.7: Box plot of process CPU load with 96 tracks

5.4 Audio processing language comparison

WebAssembly is supposed to achieve native performance, but the communication between JavaScript code and WebAssembly modules used to be slow [12] especially when calling JavaScript functions from inside a WebAssembly module. The communication performance improved a lot, but still there are contradictory statements on the performance of WebAssembly today, as pointed out in chapter 3. This section evaluates the communication cost as well as the processing speed-up of WebAssembly compared to JavaScript when implementing audio processing algorithms.

For this experiment, three scenarios are identified that are likely to affect, how WebAssembly performs for audio processing:

1. Audio buffers are processed completely in WebAssembly. This is expected to be faster than pure JavaScript.
2. The WebAssembly module calls a JavaScript function once before processing an audio buffer. This is expected to be faster than pure JavaScript.
3. The WebAssembly module calls a JavaScript function for each sample in the audio buffer. This is expected to be slower than pure JavaScript.

The last scenario in particular might seem exaggerated or even specifically designed for this evaluation, but examining some WAM2 examples[1] compiled with Emscripten shows that common JavaScript functions are called by the WASM modules. As an example, the WASM module of the Big Muff WAM can be disassembled to WAT code, which then reveals that the module calls the imported JavaScript functions _powf and _tanhf in a loop. These functions are mapped to Math.pow() and Math.tanh() in the JavaScript glue code.

5.4.1 Load generator plugin

To compare the performance of WebAssembly and JavaScript for audio processing, a load generator plugin is implemented as a Web Audio Module (WAM). The load generator creates a generic load on the audio thread with a load factor parameter for controlling the amount of processing required. The plugin can process audio buffers entirely in either WebAssembly or pure JavaScript, or it can include an additional step where the WebAssembly module either calls a built-in JavaScript function before processing an audio buffer or for every sample in the audio buffer. In the JavaScript implementation, the additional step is done entirely using JavaScript. These additional steps have been implemented to evaluate the scenarios that were introduced above.

The load generator utilizes the WamParameterInfo in the WamProcessor and defines the parameters listed in table 5.4. The table explains the load generation in the different scenarios.

The WebAssembly module is implemented using Rust and compiled with the optimization level 3, which is the default.

[1]https://github.com/webaudiomodules/wam-examples | accessed: September 21, 2022

Parameter	Description
load	A floating point value between 0 and 1 that controls how many time each sample is applied with a gain compensation factor in one audio callback call. When the load is 0, the factor is only applied once per sample, when the load is 1, the factor is applied 1000 times.
use_wasm	A boolean value that holds whether the audio processing is done in JavaScript (false) or in WebAssembly (true).
sqrt_block	A boolean value that holds whether the processing applies Math.sqrt to the gain compensation factor that is multiplied with each sample, before the audio block is processed.
sqrt_samples	A boolean value that holds whether the processing applies Math.sqrt to each sample of the audio block additionally to the gain compensation factor.

Table 5.4: Parameter defined in load generator WamProcessor

5.4.2 Experiment design

For executing the experiment, the cause and effect variables in table 5.5 are selected.

Cause variables	Effect variables
Processing Language	Audio callback Load
Number of Tracks	
Load factor	
Scenario	

Table 5.5: Cause and effect variables for audio processing language comparison

As table 5.6 shows, the experiment runs once with each WebAssembly and JavaScript. The number of instances of the load generator plugin is controlled by selecting the number of tracks that a test is run with. Each track then has one plugin instance. Additionally, the load is changed between test runs and for each of the scenarios laid down, a different algorithm of the load generator plugin is selected, which is implementing the individual scenarios.

Language	Load factor	Number of Tracks	Scenario
JavaScript	0	0	1
WebAssembly	50	96	2
			3

Table 5.6: Parameters for language comparison experiment

5.4.3 Results

The box plot in fig. 5.8 displays the load created by all the three scenarios when run with 96 tracks and without additional load.

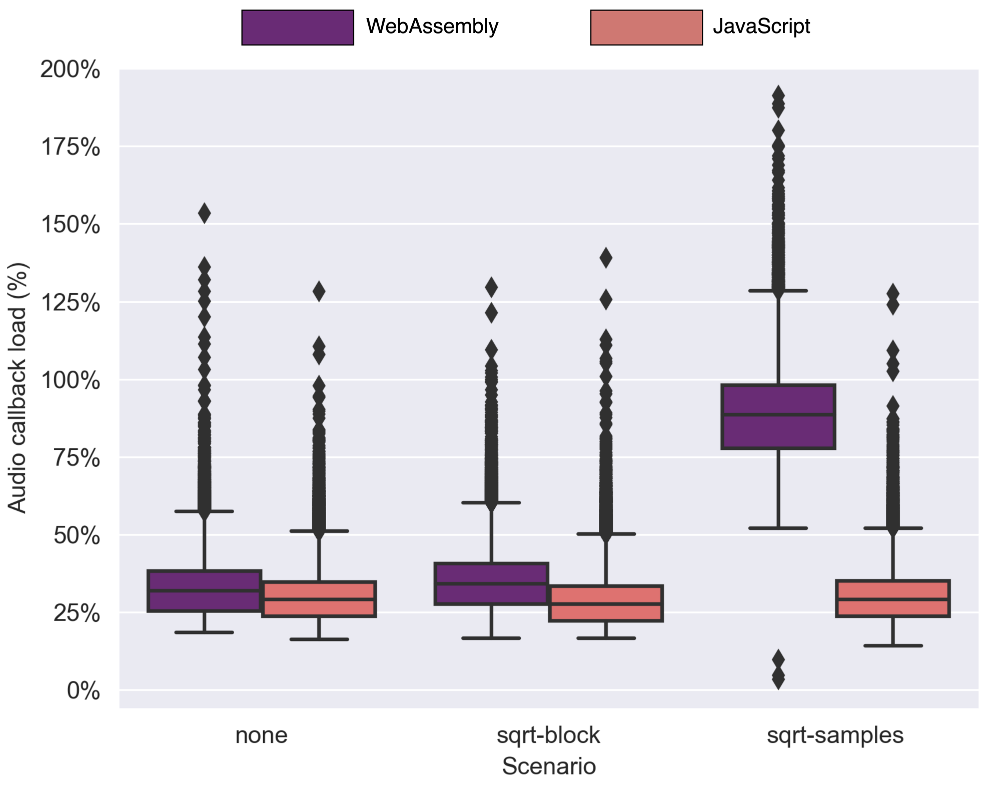


Figure 5.8: Box plot of audio callback load with 96 tracks in different scenarios

The results for a single track and 12 tracks are shown in fig. A.4 and fig. A.5 and display similar results. However, when playing 96 tracks, the box plots clearly show the effect of the "sqrt-samples" scenario, as the upper whisker of "sqrt-samples" WebAssembly box exceeds 125 % load and the lower whisker is on the same level as the upper whisker from the JavaScript version. The scenario "none" and the scenario "sqrt-block" show that WebAssembly is slower in both cases. Without additional load, JavaScript is consistently faster across all scenarios and track counts.

When additional load is introduced, WebAssembly shows better performance over JavaScript, except for the "sqrt-samples" scenario.

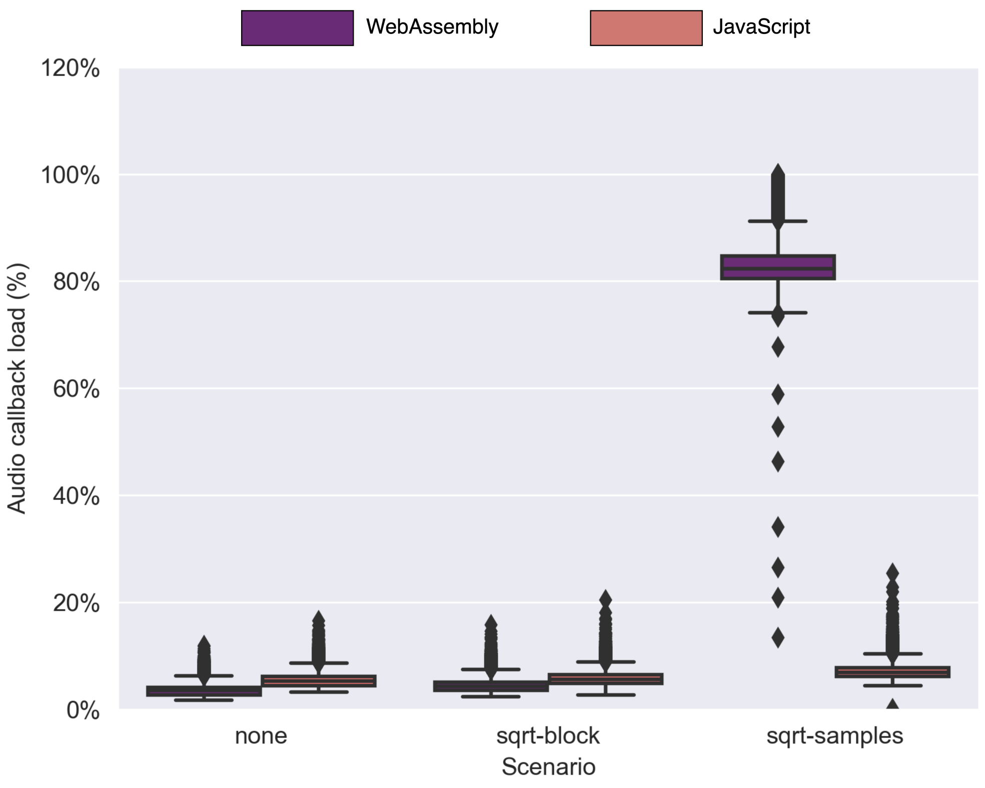


Figure 5.9: Box plot of audio callback load with 1 instance and additional load in different scenarios

The box plot in fig. 5.9 illustrates that the WebAssembly version of the "sqrt-samples" scenario generates a substantial amount of load when operating with just a single track and additional load. Although the results of the other two scenarios are difficult to recognize, it is apparent that WebAssembly outperforms the other implementations in those cases. The audio load generated by WebAssembly increases dramatically with additional number of tracks, leading to the exclusion of these results from fig. 5.10 to maintain the clarity of the comparison with the other two scenarios which results show that WebAssembly creates half the audio load compared to JavaScript in the "sqrt-block" scenario and even less in the "none" scenario.

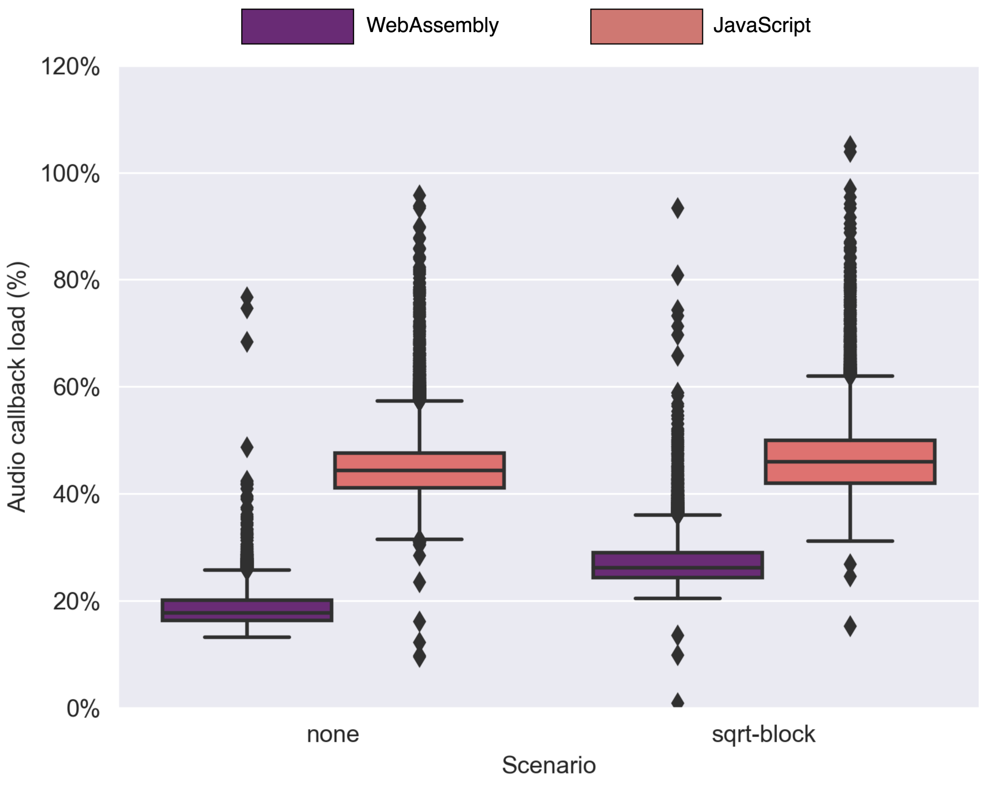


Figure 5.10: Box plot of audio callback load with 12 instances and additional load in different scenarios

With these observations, the following classification into best-case, average-case and worst-case scenario can be done:

Best-case scenario "none"

The largest performance speed-up can be achieved by implementing expensive algorithms exclusively in WebAssembly without invoking JavaScript functions. If the goal is to achieve the best possible performance with WebAssembly, using tools like Emscripten with an existing C++ code base has to be done carefully, as the compiled WebAssembly module might call JavaScript functions.

Average-case scenario "sqrt-block"

The average-case scenario shows that WebAssembly is likely to be faster in most situations, and the occasional call of JavaScript functions does not hinder performance. It's possible that numerous instances with low load could slow down WebAssembly at some point, or if the computation is too simple, it may run faster in JavaScript.

Worst-case scenario "sqrt-samples"

In the most unfavorable scenario, implementing the same algorithm directly in JavaScript results in consistently faster performance compared to WebAssembly due to a significant accumulation of communication overhead.

5.5 Discussion

The evaluation shows the success of using AudioWorklet and web workers with SharedArrayBuffer for streaming in terms of meeting the quality goals. It reveals that there is a problem with AudioBufferSourceNode causing unstable FPS. The developed solution does not have any significant drawbacks compared to AudioBufferSourceNode in terms of audio callback load, memory usage, and CPU. The evaluation has shown that streaming with AudioBufferSourceNode requires more memory than the implementation with AudioWorkletNode, which has a very stable memory footprint. The evaluation also revealed that the increased memory usage in AudioBufferSourceStreamer compared to AudioWorkletStreamer is due to a gradual build up over time that is only partially freed. This indicates that the AudioBufferSourceStreamer implementation may have memory allocation problems, or that the garbage collector will not allow such an implementation to scale as much.

The comparison of audio processing languages provides a clearer understanding of the performance of WebAssembly compared to JavaScript. By examining the cases from chapter 3, it can be concluded that the implementation in C++ with wam-openstudio (section 3.1) matches the average case and does not harm performance. No expensive

JavaScript functions are being called, only the buffers are being copied to and from WebAssembly, which results in an overhead, but it is not significant. The paper introduced in section 3.2, on the other hand, can be considered as a worst-case scenario as the differences between WebAssembly and JavaScript were found to be significant, and it seemed that WebAssembly modules are only calling JavaScript functions or very cheap operations. Overall, WebAssembly is indeed faster than JavaScript when implementing algorithms that are computationally expensive.

A more realistic comparison than the one made, would be to compare algorithms like Fast Fourier Transform (FFT) or filters like IIR filters over this generic implementation. The generic implementation on the other hand simplifies the comparison of the programming languages, as it separates the performance of the different implementations from the specific characteristics of the audio processing algorithms being used. The load generator plugin allows easy adjustment of the load factor as needed to explore the performance of the different implementations under different conditions, which would be harder to achieve with specific algorithms.

Another limitation of the evaluation is that it only partially covers the quality goal of preventing dropouts. It focuses on audio callback load, but misses to address dropouts caused by the disk streaming not providing audio fast enough. As the experiment setup already mentions, playing 96 tracks is close to the limit for the AudioWorklet solution before tracks become out of sync. Adding more web workers to help with the disk streaming would consume more memory, as web workers cannot share states. It would be of value to find out the maximum number of tracks that can be played without dropouts, and if adding more workers would have a negative impact on memory consumption.

6 Conclusion

This work provides insight into the potential utilization of the Web Audio API for implementing the engine of a Digital Audio Workstation (DAW). It demonstrates the potential offered by the combination of AudioWorklet, Web Workers and SharedArray-Buffer. These technologies enable the development of complex solutions that expand the built-in functionality of the Web Audio API. The solution presented in this work demonstrates the possibility of overcoming the limitations given by scheduling audio on the main thread and potentially disrupting the GUI or other part of the application, by applying paradigms that are known from developing native DAWs. The evaluation shows that WebAssembly can indeed enhance the speed of audio processing tasks. However, it is important to note that WebAssembly is a tool that requires proper usage in order to achieve optimal performance. Incorrect usage may result in slower performance compared to JavaScript. The developed architecture gives insight into one approach to implement a browser-based DAW and how common software engineering practices can be applied. Furthermore, the engine has been designed to load Web Audio Modules (WAMs) and the architecture of these modules is also examined in this work. The implementation of the load generator plugin is an important part of this work, as it effectively enables the comparison of the audio processing languages. This gives further insight into the potential for modularity and extensibility in audio processing using the Web Audio API. The experimental setup provides a foundation for web developers who are implementing audio applications and require an automatable process for gaining insight into the performance of their audio processing.

6.1 Outlook

The Web Audio API specification group and browser implementers aim to expose the AudioContext to web workers[1]. This would allow audio buffer scheduling on a dedicated

[1] https://github.com/WebAudio/web-audio-api/issues/2423

thread without affecting the main thread. This approach has the potential to make it easier to achieve the quality goals set out in this work. It would be interesting to evaluate the performance of the AudioBufferSourceNode on a dedicated thread, as well as to explore any potential drawbacks that might come with this approach. As web workers have limited access to Web APIs, this could open up other challenges, like synchronization issues with the GUI.

In future work, it would be valuable to further investigate the latency introduced by the Web Audio API, as this was not explored in depth in this study. A plugin for measuring the round-trip latency is available in the repository of the engine and could serve as a starting point for this investigation. By measuring the latency of a test signal using a plugin, insights can be gained into the performance of the audio processing pipeline. Additionally, exploring the recording latency of the audio system is another area of interest. This could be achieved by recording a test signal and analyzing the recorded data to determine the amount of latency introduced by the MediaStream Recording API or other components of the audio processing pipeline. It would be important to perform latency measurements on multiple platforms, including macOS, Windows, Linux, Android, and iOS, to have a better understanding of the latencies introduced by the Web Audio API. This is because the different platforms can have different implementations and might use different APIs, resulting in varying latencies. As mentioned in section 3.1, Soundtrap disables monitoring on Windows due to high latencies. Additionally, testing on different browsers could also provide insight into how different implementations may impact the latency of the audio processing pipeline. Another aspect to consider is the use of ASIO drivers on Windows, which have been known to allow for low latencies and are not currently supported by browsers.

Multithreading with web workers is another area that deserves further exploration in the context of audio processing. As mentioned in section 5.5, this work explored the use of web workers for disk streaming, focusing on a single web worker. Future work could investigate the potential for using multiple web workers for audio processing, including multithreaded audio processing algorithms, to potentially improve performance. Although there is at least one article describing a pattern of using web workers for audio processing[2], this pattern is primarily designed as a workaround for existing code bases being ported to the web using Emscripten, and may not be the best solution for all audio processing needs. Therefore, the potential benefits and drawbacks of using web workers for multithreading audio processing could be further evaluated in future work.

[2]https://developer.chrome.com/blog/audio-worklet-design-pattern

The application of Single Instruction, Multiple Data (SIMD) in WebAssembly has not been explored in this work, but it could be a promising area for future research. This optimization technique has already shown to improve the performance of audio processing algorithms, such as the fast Fourier transform (FFT), as demonstrated by the library pffft.wasm[3].

Apart from the FPS evaluation, the user interface was not addressed in this work, but is an important aspect for performance. Traditional DAWs have intensive animations and visualizations that must be synchronized with the audio playback. It would be interesting to adapt and evaluate this in a web browser. This topic was also touched in the Karpluss-Strong-Stress-Tester, as seen in section 3.2, which included an optional GUI for stress testing and could be a starting point for further investigation.

The possibilities offered by the web browser seem to be truly endless, and there is so much more to explore. For instance, it would be interesting to incorporate WebRTC with the StreamCoordinator by adding a new WebRTCStreamer for web-based streaming. This could be compared with established collaboration technologies like JackTrip or Steinberg's VST Connect for remote recording. Another completely different investigation could be the approach by AmpedStudio, mentioned in section 3.1, using VST3 plugins within the browser by communicating between the plugin's remote host and the browser. The future of browser-based audio processing is exciting and offers a lot of potential.

[3]https://github.com/JorenSix/pffft.wasm

Bibliography

[1] *Delay Compensation FAQ.* https://help.ableton.com/hc/en-us/articles/209072409-Delay-Compensation-FAQ. – [Online; accessed 06-February-2023]

[2] *About Soundtrap.* https://www.soundtrap.com/about. – [Online; accessed 04-September-2022]

[3] Adenot, Paul: *Profiling real-time audio workloads in Firefox.* https://blog.paul.cx/post/profiling-firefox-real-time-media-workloads. – [Online; accessed 22-September-2022]

[4] Adenot, Paul: *A wait-free single-producer single-consumer ring buffer for the Web.* https://blog.paul.cx/post/a-wait-free-spsc-ringbuffer-for-the-web. – [Online; accessed 22-September-2022]

[5] *Analog Tape Recording Basics.* https://www.uaudio.com/blog/analog-tape-recording-basics/. – [Online; accessed 06-February-2023]

[6] Atencio, Luis: *The Joy of JavaScript.* Manning Publications, 2021. – ISBN 978-1-617-29586-7

[7] (Author), Chris M. ; Holland, Eva: *Coding with JavaScript For Dummies.* O'Reilly Media, 2015. – ISBN 978-1-119-05607-2

[8] Batisse, Dylann ; Cousson, Antoine ; Mazuy, Antoine V. ; Carlens, Jean-Philippe ; Buffa, Michel: *Host and Plugins : Parameter Automation Without Crossing the Audio Thread Barrier.* https://zenodo.org/record/6769641. – [Online; accessed 02-August-2022]

[9] Blandy, Jim ; Orendorff, Jason ; Tindall, Leonora F . S.: *Programming Rust, 2nd Edition.* O'Reilly Media, 2021. – ISBN 978-1-492-05254-8

[10] Buffa, Michel ; Ren, Shihong ; Campbell, Owen ; Burns, Tom ; Yi, Steven ; Kleimola, Jari: *Web Audio Modules 2.0: an Open Web Audio Plugin Standard*. https://dl.acm.org/doi/pdf/10.1145/3487553.3524225. – [Online; accessed 05-February-2023]

[11] Celemony: *ARA SDK 2.1.0 Documentation*. https://github.com/Celemony/ARA_Library. – [Online; accessed 11-August-2022]

[12] Clark, Lin: *Calls between JavaScript and WebAssembly are finally fast*. https://hacks.mozilla.org/2018/10/calls-between-javascript-and-webassembly-are-finally-fast. – [Online; accessed 22-September-2022]

[13] Clark, Lin: *A crash course in just-in-time (JIT) compilers*. https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/. – [Online; accessed 07-February-2023]

[14] contributors, MDN: *AudioBufferSourceNode*. https://developer.mozilla.org/en-US/docs/Web/API/AudioBufferSourceNode. – [Online; accessed 8-August-2022]

[15] contributors, MDN: *AudioScheduledSourceNode*. https://developer.mozilla.org/en-US/docs/Web/API/AudioNode. – [Online; accessed 31-January-2023]

[16] contributors, MDN: *AudioScheduledSourceNode*. https://developer.mozilla.org/en-US/docs/Web/API/AudioScheduledSourceNode. – [Online; accessed 8-August-2022]

[17] contributors, MDN: *ScriptProcessorNode*. https://developer.mozilla.org/en-US/docs/Web/API/ScriptProcessorNode. – [Online; accessed 8-August-2022]

[18] contributors, MDN: *The structured clone algorithm*. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm. – [Online; accessed 8-Dezember-2022]

[19] contributors, MDN: *Worklet*. https://developer.mozilla.org/en-US/docs/Web/API/Worklet. – [Online; accessed 8-August-2022]

[20] CORP., NATIONAL I.: *Saving Memory using Disk Streaming.* https://www.ni.com/docs/en-US/bundle/labview/page/lvconcepts/basics_disk_streaming.html. – [Online; accessed 28-January-2023]

[21] FLANAGAN, David: *JavaScript: The Definitive Guide, 7th Edition.* O'Reilly and Associates, 2021. – ISBN 978-0-596-80552-4

[22] HOFFMAN, Kevin: *Programming WebAssembly with Rust: Unified Development for Web, Mobile, and Embedded Applications.* O'Reilly Media, 2019. – ISBN 978-1-680-50636-5

[23] HOSKEN, Dan: *An Introduction to Music Technology.* Routledge, 2011. – ISBN 978–0–415–87827–2

[24] HUNTER, Thomas ; ENGLISH, Bryan: *Multithreaded JavaScript: Concurrency Beyond the Event Loop.* O'Reilly Media, 2021. – ISBN 978-1-098-10443-6

[25] KOLADA, Brian: *Spotify acquires cloud-based recording software Soundtrap.* https://ra.co/news/40457. 2017

[26] LOPP, Sean: *Boxplots and p-values.* https://loppsided.blog/posts/2021-06-18-boxplots-and-p-values/. 2021. – [Online; accessed 28-January-2023]

[27] MARTIN, Robert C.: *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall, 2008. – ISBN 978-0-132-35088-4

[28] MARTIN, Robert C.: *Clean Architecture: A Craftsman's Guide to Software Structure and Design: A Craftsman's Guide to Software Structure and Design.* Addison-Wesley, 2017. – ISBN 978-0-134-49416-6

[29] MCPHERSON, Andrew P. ; JACK, Robert H. ; MORO, Giulio: *Action-Sound Latency: Are Our Tools Fast Enough?* http://eecs.qmul.ac.uk/~andrewm/mcpherson_nime2016.pdf. 2016. – [Online; accessed 11-July-2022]

[30] PENDHARKAR, Chinmay ; BÄCK, Peter ; WYSE, Lonce: *Adventures in scheduling, buffers and parameters : Porting a dynamic audio engine to Web Audio.* https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.683.7838&rep=rep1&type=pdf. – [Online; accessed 02-February-2023]

[31] PIRKLE, Will C.: *Designing audio effect plugins in C++ : for AAX, AU, and VST3 with DSP theory.* New York London: Routledge, Taylor & Francis Group, 2019. – ISBN 978-1-138-59189-9

[32] Reiss, Joshua: *Working with the Web Audio API*. Focal Press, 2022. – ISBN 978-1-032-11867-3

[33] Roma, Gerard: *Comparing approaches for new AudioWorklets*. https://zenodo.org/record/6767468. – [Online; accessed 22-September-2022]

[34] Starke, Gernot ; Hruschka, Peter: *arc42 in Aktion*. Carl Hanser Verlag GmbH & Co. KG, 2016. – ISBN 978-3446448018

[35] Steen, Maarten van ; Tanenbaum, Andrew S.: *Distributed Systems*. CreateSpace Independent Publishing Platform, 2017. – ISBN 978-1-543-05738-6

[36] Surma: *The State Of Web Workers In 2021*. https://www.smashingmagazine.com/2021/06/web-workers-2021/. – [Online; accessed 28-January-2023]

[37] *The Ardour Manual*. https://manual.ardour.org/synchronization/latency-and-latency-compensation/. – [Online; accessed 06-February-2023]

[38] *Use Cases*. https://webassembly.org/docs/use-cases/. – [Online; accessed 06-February-2023]

[39] *WebAssembly*. https://webassembly.org/. – [Online; accessed 06-February-2023]

[40] *WebAssembly Introduction*. https://webassembly.github.io/spec/core/intro/introduction.html. – [Online; accessed 06-February-2023]

[41] Williams, Anthony: *C++ Concurrency in Action*. Manning, 2011. – ISBN 978-1-617-29469-3

[42] Wyse, Lonce ; Subramanian, Srikumar: *The Viability of the Web Browser as a Computer Music Platform*. https://www.researchgate.net/publication/262216013_The_Viability_of_the_Web_Browser_as_a_Computer_Music_Platform. – [Online; accessed 02-August-2022]

A Appendix

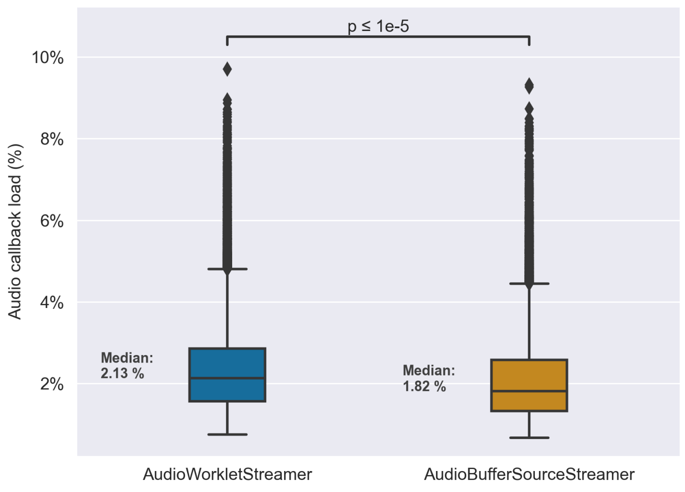


Figure A.1: Box plot of audio callback load with 1 track

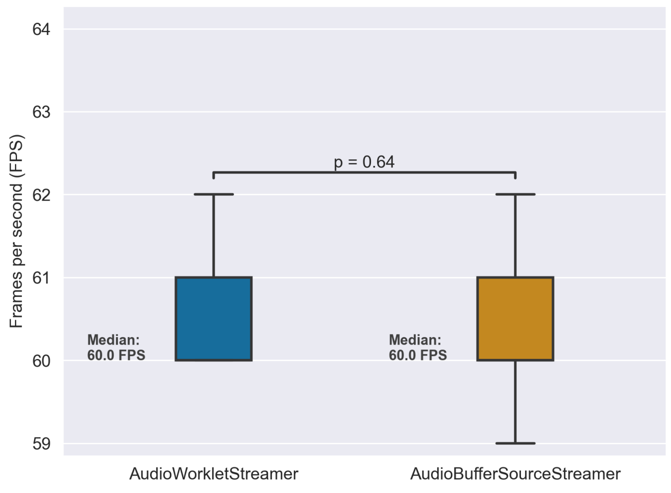


Figure A.2: Box plot of FPS with 1 track

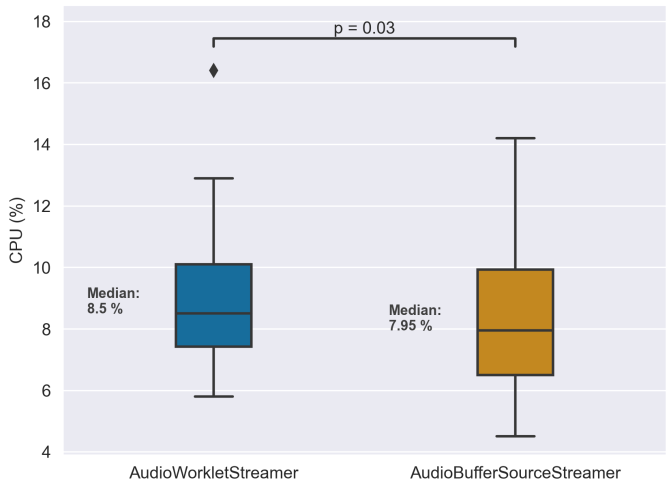


Figure A.3: Box plot of process CPU with 1 track

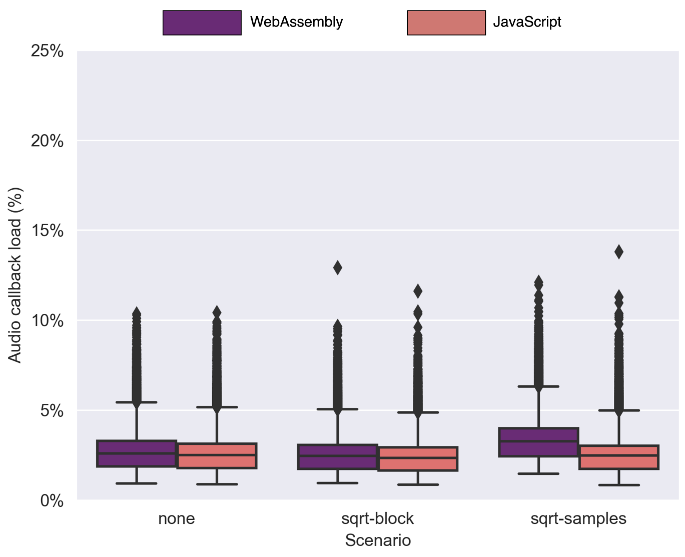


Figure A.4: Box plot of audio callback load with 1 track in different scenarios

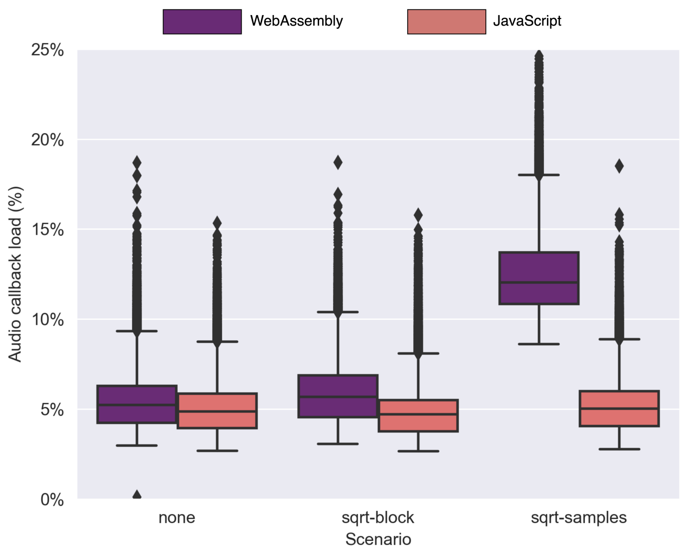


Figure A.5: Box plot of audio callback load with 12 tracks in different scenarios

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original