
Advanced Sound for Games and Interactive Apps
Web Audio API
O'REILLY®
Boris Smus

Оглавление

Предисловие

Спасибо, что читаете первую книгу, посвящённую Web Audio API. Когда я впервые узнал о Web Audio API, я был новичком в цифровом аудио и только начинал путь к изучению и пониманию этого API и базовых концепций работы с аудио. Этой книги мне очень не хватало, когда я начал экспериментировать с API в 2011 году. Она призвана стать отправной точкой для веб-разработчиков вроде меня в прошлом, которые мало знакомы или совсем не знакомы с цифровым аудио. В ней собрано то, что я узнал за год изучения области цифровой обработки аудио, общения с экспертами и собственных экспериментов с API.

Я буду раскрывать теоретическую часть в специальных вставках с объяснением основных концепций. Если вы гуру цифрового аудио, можете смело их пропускать. Практическая часть будет сопровождаться примерами кода, чтобы было проще понимать, как работает API. Многие примеры также содержат ссылки на рабочие демо, который доступны на сайте Web Audio API.

Структура книги

Книга призвана дать общее представление о ключевых возможностях Web Audio API, но не является исчерпывающим исследованием по каждой из них. Она задумана не как полный справочник, а как удобная отправная точка. Большинство разделов книги начинаются с описания применения API: формулируется задача и её решение, а затем приводится соответствующий пример кода на JavaScript с использованием Web Audio API. Вставки с теорией по ходу текста объясняют основные аудиоконцепции в более общем виде. Книга структурирована следующим образом:

1. Основы посвящены базовым принципам работы аудиографов, их стандартным конфигурациям, аудиоузлам, из которых состоят аудиографы, а также загрузке и воспроизведению звуковых файлов.

2. Идеальный тайминг и задержка рассказывают о точном планировании проигрывания звука в будущем, одновременном воспроизведении нескольких звуков, изменении параметров звука напрямую или во времени и использовании кроссфейда.

3. Громкость и воспринимаемая звучность объясняют понятия усиления, громкости и воспринимаемой звучности, а также способы предотвращения клиппинга с помощью измерителей и динамической компрессии.

4. Высота звука и частотный спектр — это раздел о частоте звука, ключевом свойстве периодических колебаний. Также мы разбираем работу осцилляторов и анализ звука в частотной области.

5. Анализ и визуализация отходят от тем синтеза и обработки звука и углубляются в такие темы, как анализ и визуализация.

6. Продвинутые темы продолжают материал предыдущих разделов и углубляются в более сложные темы, такие как биквадратные фильтры, симуляция акустики помещений и пространственная обработка звука.

7. Интеграция с другими технологиями описывает способы взаимодействия Web Audio API с такими веб-API, как WebRTC и тег `<audio>`.

Исходный код этой книги опубликован под лицензией Creative Commons и доступен на GitHub.

Условные обозначения

В книге используются следующие правила оформления текста:

Курсив

Используется для обозначения новых терминов, URL-адресов, адресов электронной почты, названий и расширений файлов.

`Моноширинный шрифт`

Применяется для листингов программ, а также внутри текста для упоминания элементов программ, таких как имена переменных или функций, базы данных, типы данных, переменные окружения, операторы и ключевые слова.

`Моноширинный жирный`

Показывает команды или другой текст, который должен быть введен пользователем буквально.

`Моноширинный курсив`

Обозначает текст, который нужно заменить значениями, вводимыми пользователем, или значениями, определяемыми контекстом.

СОВЕТ

Эта иконка обозначает совет, рекомендацию или общее примечание.

ПРЕДУПРЕЖДЕНИЕ

Эта иконка указывает на предупреждение или предостережение.

Использование примеров кода

Эта книга создана, чтобы помочь вам в работе. В общем случае, если в книге приведены примеры кода, вы можете использовать их в своих программах и документации. Вам не нужно получать наше разрешение, если только вы не воспроизводите значительную часть кода. Например, написание программы, в которой используется несколько фрагментов кода из этой книги, не требует разрешения. Продажа или распространение CD-ROM с примерами из книг O'Reilly уже требует разрешения. Отвечая на вопрос, вы можете сослаться на эту книгу и привести пример кода без запроса разрешения. Однако включение значительного количества примеров кода из этой книги в документацию к вашему продукту требует получения разрешения.

Мы будем признательны за упоминание источника, но это не является обязательным. Обычно атрибуция включает название книги, автора, издателя и ISBN. Например: "*Web Audio API* by Boris Smus (O'Reilly). Copyright 2013 Boris Smus, 978-1-449-33268-6."

Если вы считаете, что использование вами примеров кода выходит за рамки добросовестного использования или вышеупомянутого разрешения, свяжитесь с нами по адресу permissions@oreilly.com.

Safari® Books Online

ПРИМЕЧАНИЕ

Safari Books Online (www.safaribooksonline.com) — это цифровая библиотека по запросу, предоставляющая экспертный контент в формате книг и видео от ведущих авторов в области технологий и бизнеса.

Профессионалы в сфере технологий, разработчики программного обеспечения, веб-дизайнеры, а также специалисты в области бизнеса и креативных индустрий используют Safari Books Online как основной ресурс для исследований, решения задач, обучения и подготовки к сертификации.

Safari Books Online предлагает различные комбинации продуктов и тарифные программы для организаций, государственных учреждений и частных лиц. Подписчики получают доступ к тысячам книг, обучающих видео и рукописей до их официальной публикации — всё это в одной полнотекстово индексируемой базе данных от таких издателей, как O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology и многих других. Для получения дополнительной информации о Safari Books Online, пожалуйста, посетите наш сайт.

Как связаться с нами

Все комментарии и вопросы по этой книге просим адресовать издателю:

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (in the United States or Canada)
- 707-829-0515 (international or local)
- 707-829-0104 (fax)

У этой книги есть веб-страница, на которой мы публикуем исправления, примеры и дополнительную информацию. Страница доступна по адресу: *http://oreil.ly/web-audio-api*.

Для отправки комментариев или запросов по техническим вопросам по этой книге, отправьте электронное письмо на *bookquestions@oreilly.com*.

Для получения дополнительной информации о наших книгах, курсах, конференциях и новостях, посетите наш сайт по адресу: *http://www.oreilly.com*.

Ищите нас в Facebook: *http://facebook.com/oreilly*.

Следите за нами в Twitter: *http://twitter.com/oreillymedia*.

Смотрите нас на YouTube: *http://www.youtube.com/oreillymedia*.

Благодарности

Я не эксперт в области цифровой обработки аудио, мастеринге или сведении звука. Я — инженер-программист и музыкант-любитель, которому хватило интереса вникнуть в цифровое аудио и уделить время изучению Web Audio API, а также разобраться в некоторых его ключевых концепциях. Чтобы написать эту книгу, мне приходилось постоянно беспокоить других людей, у которых было намного больше опыта в цифровом аудио, чем у меня. Я хочу поблагодарить их за то, что они отвечали на мои вопросы, проводили технические обзоры этой книги и поддерживали меня на всём пути.

В частности, этой книги не было бы без щедрой поддержки и наставничества Криса Роджерса — основного автора спецификации Web Audio и главного разработчика её реализации в WebKit/Chrome. Огромное спасибо Крису Уилсону, который сделал невероятно тщательный технический обзор содержания этой книги, и Марку Голдстину, который провёл немало ночей, вычитывая текст. Я благодарен Кевину Эннису за то, что он предоставил сайт webaudioapi.com для размещения примеров кода из книги. И последнее, но не менее важное: я бы никогда не написал эту книгу без поддержки и интереса энергичного интернет-сообщества Web Audio API.

Ну что ж, без лишних слов — приступим!

Основы

В этой главе мы рассмотрим, как начать работу с Web Audio API: какие браузеры его поддерживают, как определить доступность API, что такое аудиограф, что такое аудиоузлы, как соединять узлы между собой, познакомимся с основными типами узлов, а также узнаем, как загружать и воспроизводить звуковые файлы.

Краткая история аудио в вебе

Первым способом воспроизведения звуков в вебе был тег `<bgsound>`, который позволял авторам сайтов автоматически запускать фоновую музыку при открытии страницы пользователем. Эта функция была доступна только в Internet Explorer, не была стандартизирована и не получила поддержки в других браузерах. Netscape реализовал похожую функцию с помощью тега `<embed>`, обеспечивая по сути тот же самый функционал.

Первым кроссбраузерным способом воспроизведения звука в интернете стал Flash, однако у него был серьёзный недостаток — для работы требовался плагин. Позже производители браузеров объединились вокруг элемента HTML5 `<audio>`, который теперь нативно поддерживает воспроизведение аудио во всех современных браузерах.

Хотя воспроизведение аудио на веб-страницах больше не требует плагинов, тег `<audio>` имеет существенные ограничения при создании сложных игр и интерактивных приложений. Вот лишь некоторые из этих ограничений:

- отсутствие возможности точного контроля времени;
- очень низкий предел на количество одновременно проигрываемых звуков;
- нет надёжного способа предварительной буферизации звука;
- нет возможности применять эффекты в реальном времени;
- отсутствие инструментов для анализа звука.

Предпринималось несколько попыток создать мощный аудио API для веба, чтобы устранить некоторые из упомянутых ранее ограничений. Один из заметных примеров — это Audio Data API, разработанный в Mozilla Firefox. Подход Mozilla

предусматривал расширение возможностей JavaScript API элемента `<audio>`. Этот API предоставлял ограниченный аудиограф (подробнее об этом позже в разделе Аудиоконтекст) и не вышел за рамки первой реализации. Сейчас он считается устаревшим в Firefox и заменён на Web Audio API.

В сравнении с Audio Data API, Web Audio API предложил совершенно новую систему обработки и синтеза звука, никак не завязанную на элемент `<audio>`, однако предоставил возможности для интеграции с другими Web API (подробнее в разделе Интеграция с другими технологиями). Web Audio API — это высокоуровневый JavaScript API, использующийся для обработки и синтеза аудио в веб-приложениях. Цель этого API — предоставить возможности, которые есть у игровых движков, и у современных десктопных приложений для работы со звуком и решать задачи, связанные с микшированием, обработкой аудио и применением аудиофильтров. В результате получился универсальный API, который может использоваться во множестве задач, связанных с аудио: от использования в играх и интерактивных приложениях, до создания продвинутых приложений для синтеза звука и визуализаций.

Игры и интерактивность

Звук — это огромная часть того, что делает интерактивные впечатления настолько увлекательными. Если вы не верите — попробуйте посмотреть фильм с выключенным звуком.

Игры тоже не исключение! Мои самые тёплые воспоминания о видеоиграх связаны именно с музыкой и звуковыми эффектами. Спустя почти два десятилетия после выхода некоторых моих любимых игр я до сих пор не могу выбросить из головы саундтреки Кодзи Кондо к *Zelda* и Мэтта Юльмена к *Diablo*. Даже звуковые эффекты этих мастерски созданных игр мгновенно узнаваемы — от звуковых откликов юнитов при клике в сериях *Warcraft* и *Starcraft* от Blizzard до семплов из классики Nintendo.

Звуковые эффекты играют огромную роль не только в играх. Они появились в пользовательских интерфейсах (UI) ещё во времена командной строки, когда некоторые ошибки сопровождались звуковым сигналом. Эта идея продолжает жить и в современных интерфейсах, где грамотно подобранные звуки критически важны для уведомлений, сигналов, а также для приложений аудио- и видеосвязи,

таких как Skype. Ассистенты вроде Google Now и Siri обеспечивают богатую звуковую обратную связь. По мере того как мы всё глубже погружаемся в мир повсеместных вычислений, интерфейсы на основе речи и жестов, которые не требуют экрана, всё сильнее зависят от звукового наполнения. Наконец, для пользователей с нарушением зрения звуковые подсказки, синтез и распознавание речи критически важны для создания удобного пользовательского опыта.

Интерактивный звук ставит перед разработчиками ряд интересных задач. Чтобы создать убедительное музыкальное сопровождение внутри игр, дизайнерам приходится учитывать все потенциально непредсказуемые игровые состояния, в которых может оказаться игрок. На практике это означает, что отдельные участки игры могут длиться неопределённое время, а звуки — взаимодействовать с окружением и смешиваться между собой сложным образом, требуя эффектов, которые зависят от окружающей среды, и относительного позиционирования источников звука. Наконец, в игре может одновременно воспроизводиться очень много звуков, и все они должны хорошо сочетаться друг с другом и звучать без потери качества и производительности.

Аудиоконтекст

Web Audio API выстроен вокруг концепции аудиоконтекста. Аудиоконтекст — это ориентированный граф аудиоузлов, который определяет, как аудиопоток проходит от источника (чаще всего это аудиофайл) до финального аудиовыхода (например, к вашим колонкам). По мере прохождения аудио через каждый узел, его свойства могут меняться или подвергаться анализу. Самый простой аудиоконтекст — это прямая связь от узла источника к аудиовыходу (см. Рисунок 1-1).

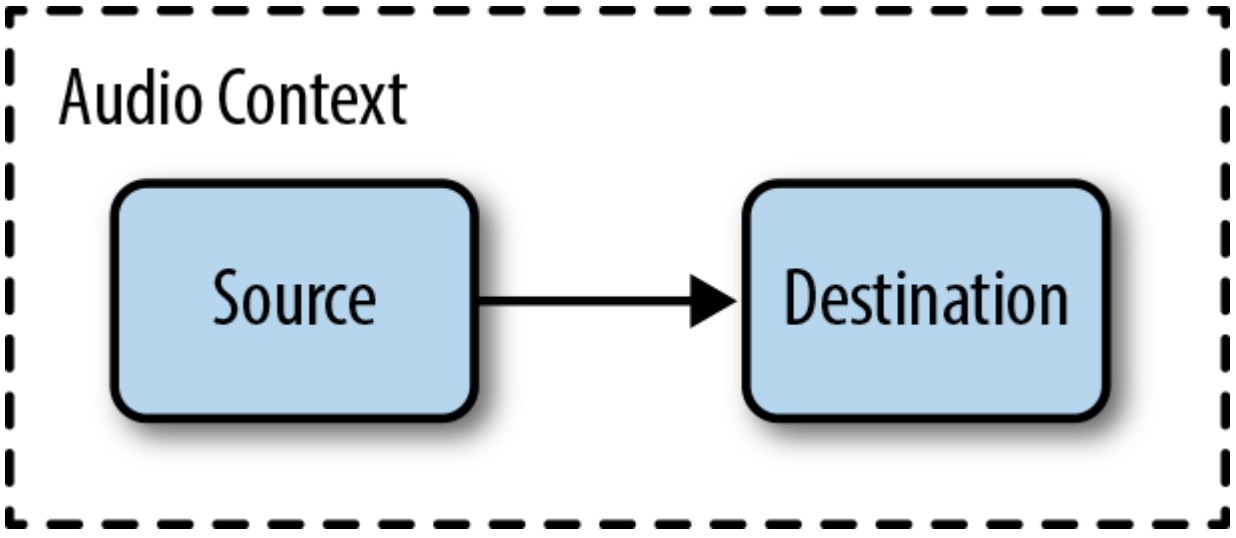

Рисунок 1-1. Простейший аудиоконтекст

Аудиоконтекст может быть сложным, состоящим из множества узлов, работающих между источником и выходом звука (см. Рисунок 1-2), выполняющих продвинутый синтез или анализ аудио.

На Рисунке 1-1 и Рисунке 1-2 показаны аудиоузлы в виде блоков. Стрелки представляют собой связи между узлами. Узлы часто могут иметь несколько входящих и исходящих связей. По умолчанию, если в узел приходит несколько входящих связей, Web Audio API смешивает входящие звуковые сигналы в один общий.

Концепт аудиографа не был изобретён в Web Audio API — до этого он уже был реализован в таких популярных аудиофреймворках, как CoreAudio от Apple, который имеет свой собственный Audio Processing Graph API. Сама идея аудиографа зародилась ещё раньше и берёт своё начало из таких аудиосред как модульный синтезатор Moog, разработанный в 1960-ых годах.

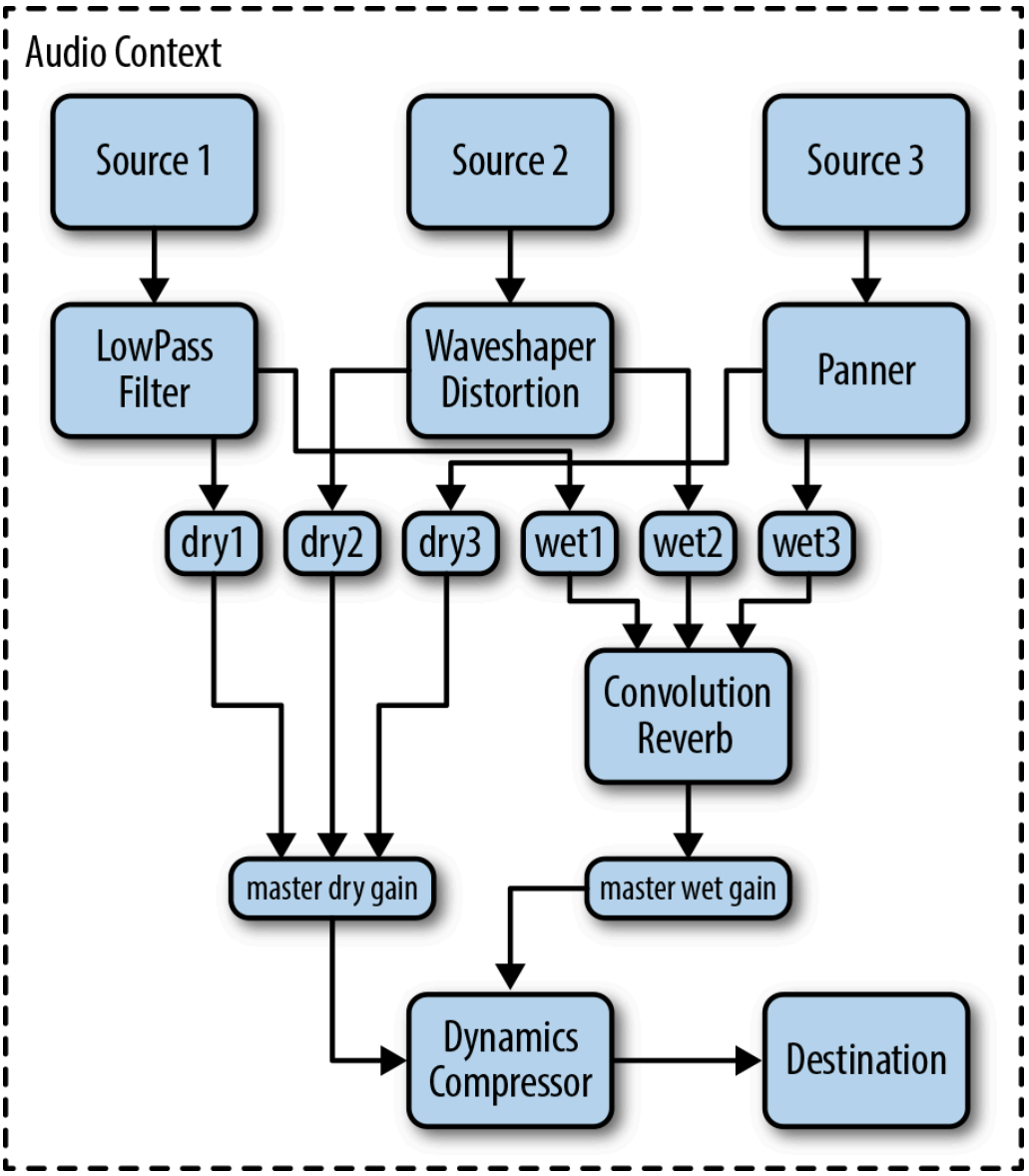

Рисунок 1-2. Более сложный аудиоконтекст

Инициализация аудиоконтекста

В настоящее время Web Audio API реализован в браузерах Chrome и Safari (включая MobileSafari, начиная с iOS 6) и доступен веб-разработчикам через JavaScript. В этих браузерах конструктор аудиоконтекста использует префикс `webkit`, то есть вместо создания `new AudioContext` необходимо использовать `new webkitAudioContext`. Наверняка это изменится в будущем, по мере стабилизации API и появления реализации без префиксов, а также по мере появления реализации у других производителей браузеров. Mozilla публично заявила о намерении реализовать Web Audio API в Firefox, а Opera уже начала участвовать в рабочей группе.

С учётом этого, ниже представлен самый универсальный способ инициализации аудиоконтекста, который учитывает возможные реализации в разных браузерах (в том числе и будущие):

```
var contextClass = (window.AudioContext ||
  window.webkitAudioContext ||
  window.mozAudioContext ||
  window.oAudioContext ||
  window.msAudioContext
);

if (contextClass) {
  // Web Audio API доступен
  var context = new contextClass();
} else {
  // Web Audio API недоступен
  // Попросите пользователя использовать современные браузеры
}
```

Один аудиоконтекст поддерживает несколько источников аудио и сложные аудиографы, так что, как правило, в рамках одного приложения достаточно одного экземпляра аудиоконтекста. Экземпляр аудиоконтекста содержит множество методов для создания аудиоузлов и управления глобальными настройками аудио. К счастью, эти методы работают относительно стабильно и не требуют использования префикса `webkit`. Однако стоит помнить, что API всё ещё развивается, поэтому могут появляться изменения, которые не будут иметь обратной совместимости (см. Устаревшие возможности API).

Типы аудиоузлов

Аудиоузлы — это основные компоненты Web Audio API. Они представляют собой объекты, которые могут принимать и генерировать звуковые сигналы.

Узлы источников звука

Источниками звука могут быть аудиобуферы, живые аудиовходы, элементы `<audio>`, осцилляторы и JS-обработчики.

Модифицирующие узлы

Фильтры, узлы реверберации, узлы панорамирования и т.д.

Узлы для анализа звука

Анализаторы и JS-обработчики.

Узлы аудиовыходов

Аудиовыходы и оффлайн-буферы для отложенной обработки звука.

Источниками звука могут быть не только аудиофайлы, но также аудиопоток в реальном времени от музыкального инструмента или микрофона, перенаправление аудиовыхода из HTML-элемента `<audio>` (см. Настраиваем фоновую музыку с помощью HTML-элемента `<audio>`) или вообще полностью синтезированные звуки (см. Обработка аудио с помощью JavaScript). Обычно в качестве аудиовыхода используются колонки, но вы можете обрабатывать звук без его воспроизведения (например, для чистой визуализации) или выполнять его обработку в оффлайн-режиме, при которой аудиопоток записывается в буфер назначения для отложенного использования.

Подключение узлов в аудиографе

Любой аудиоузел может быть подключен к другому аудиоузлу через метод `connect()`. Ниже мы подключим выход узла аудиоисточника к узлу усиления, а узел усиления к глобальному аудиовыходу:

```
// Создадим источник
var source = context.createBufferSource();

// Создаём узел усиления звука
var gain = context.createGain();

// Подключаем источник к узлу усиления, а фильтр к выходу
var destination = context.destination;
source.connect(gain);
gain.connect(destination);
```

Обратите внимание, что `context.destination` — это специальный узел, который связан с дефолтным аудиовыходом вашей системы. Конечный аудиограф будет выглядеть как на Рисунке 1-3.

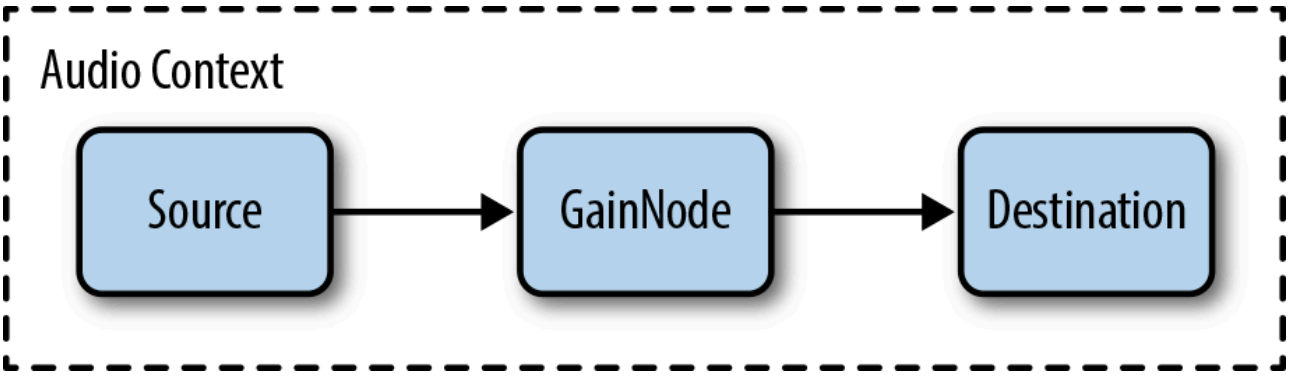

Рисунок 1-3. Наш первый аудиограф

Теперь мы можем динамически изменять этот граф. Мы можем отключать аудиоузлы, вызывая `node.disconnect(outputNumber)`. Для примера, направим подключение от источника к аудиовыходу, минуя промежуточный узел:

```
source.disconnect(0);
gain.disconnect(0);
source.connect(context.destination);
```

Сила модульной маршрутизации

Во многих играх различные источники звука могут быть объединены в один для создания финального микса. Такими источниками могут быть: фоновая музыка, звуковые эффекты, звуки интерфейса, и даже голосовой чат в мультиплеер-играх. Web Audio API позволяет разделить все эти каналы и даёт полный контроль над каждым каналом отдельно и над всеми вместе. Граф обработки аудио для такой системы может выглядеть как на Рисунке 1-4.

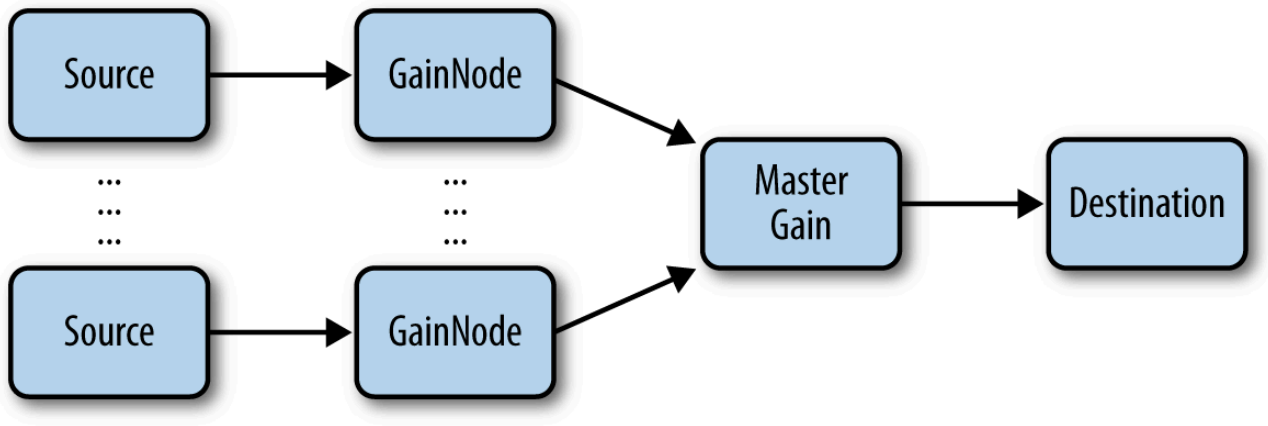

Рисунок 1-4. Несколько источников с индивидуальным управлением усилением и мастер-регулятором усиления

Здесь каждый узел усиления связан со своим отдельным источником звука, а контроль над усилением звука со всех каналов одновременно осуществляется через отдельный *мастер*-узел. Такая система позволяет легко контролировать уровень каждого канала отдельно.

Что такое звук?

С точки зрения физики, звук — это продольная волна (иногда называемая волной давления), которая распространяется в воздухе или другой среде. Источник звука заставляет молекулы воздуха вибрировать и сталкиваться друг с другом. В результате возникают области повышенного и пониженного давления, которые чередуются, сходятся и расходятся в виде полос. Если бы можно было изобразить узор звуковой волны, остановленной во времени, он мог бы выглядеть как на Рисунке 1-5.

Рисунок 1-5. Звуковая волна, распространяющаяся через воздушные частицы

Математически, звук может быть представлен функцией, отображающей изменение давления в зависимости от времени. На Рисунке 1-6 показан график такой функции. Области с большими значениями на графике соответствуют областям с частицами, расположенными более плотно на Рисунке 1-5 (области высокого давления), а с меньшими значениями — областям с редко расположенными частицами (области низкого давления).

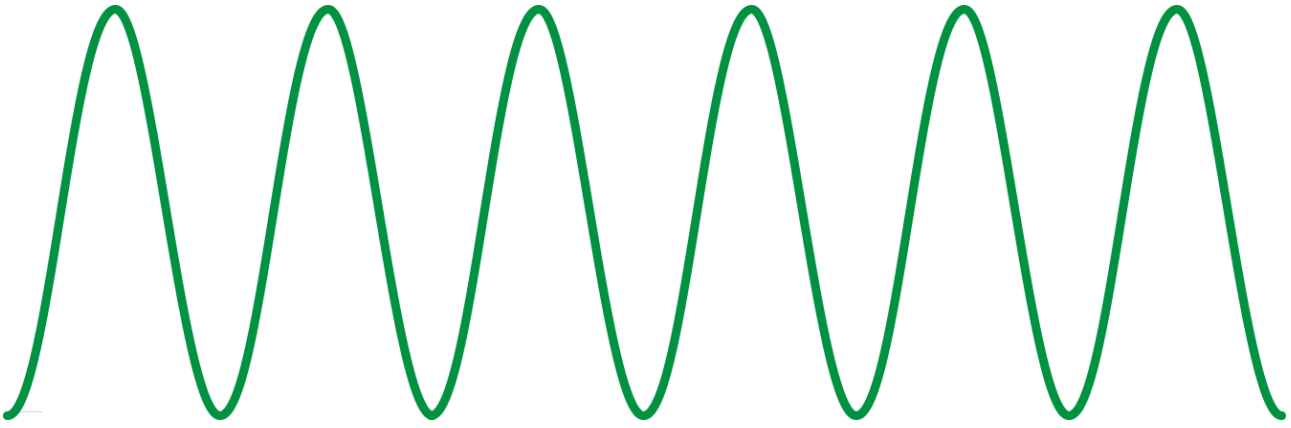

Рисунок 1-6. Математическое представление звуковой волны из Рисунка 1-5

Ещё с начала двадцатого века электроника позволяла воспринимать и воссоздавать звуки. Микрофоны могут воспринять волну давления звука и превратить её в электрический сигнал, в котором (для примера) `+5` вольт будут соответствовать самому высокому уровню давления, а `-5` — самому низкому. И наоборот, аудиоколонка может воспринимать эти значения напряжения сигнала в вольтах и превращать их в волну давления, которую мы можем слышать.

Неважно, анализируем мы звук или синтезируем его, всё самое интересное для программистов аудио происходит внутри «чёрного ящика» (см. Рисунок 1-7), отвечающего за обработку аудиосигнала. Раньше этот процесс осуществлялся с помощью аналоговых фильтров и другого оборудования, которое по современным меркам выглядит устаревшим. Сегодня существуют цифровые реализации большинства этих старых аналоговых устройств. Но прежде чем мы сможем приступить к программной реализации этих интересных задач, нам нужно представить звук в той форме, с которой могут работать компьютеры.

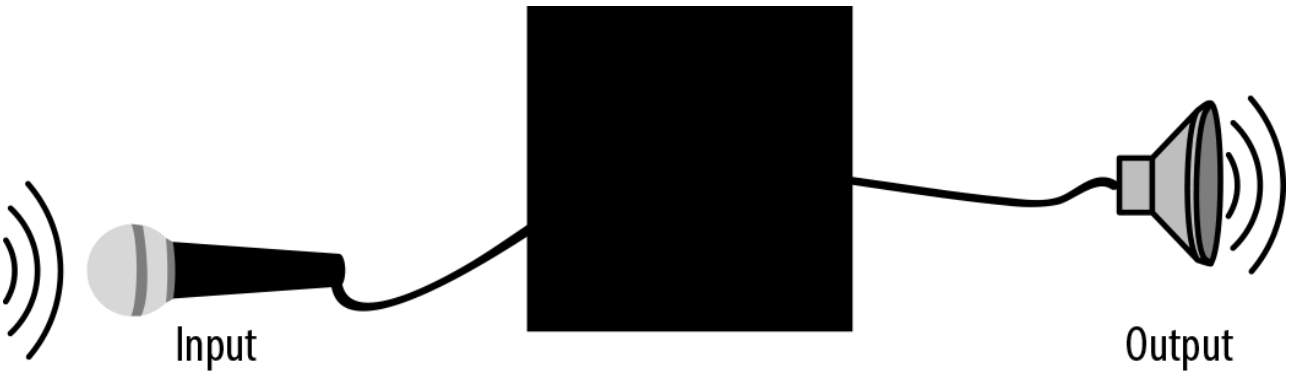

Рисунок 1-7. Запись и воспроизведение звука

Что такое цифровой звук?

Мы можем перевести аналоговый звук в цифровой посредством его семплирования — то есть измерения частоты аналогового сигнала через небольшие отрезки времени. Далее мы можем закодировать сигнал в каждом семпле в виде конкретного числа. Частота таких измерений аналогового сигнала называется *частотой дискретизации*. Самая распространенная частота дискретизации — 44.1 kHz. Это означает, что для каждой секунды звука записывается 44 100 чисел. Сами числа находятся в определённом диапазоне значений. Каждому значению обычно выделяется определённое количество битов

— это называется *глубиной битности*. В большинстве случаев для цифровой аудиозаписи, включая CD-диски, используется глубина 16 бит, которой обычно достаточно для большинства слушателей. Аудиофилы могут предпочитать 24-битную глубину, которая обеспечивает такую точность, выше которой человеческое ухо уже перестаёт улавливать какую-либо разницу.

Процесс конвертации аналогового сигнала называется *квантованием* (или *дискретизацией*) (см. Рисунок 1-8).

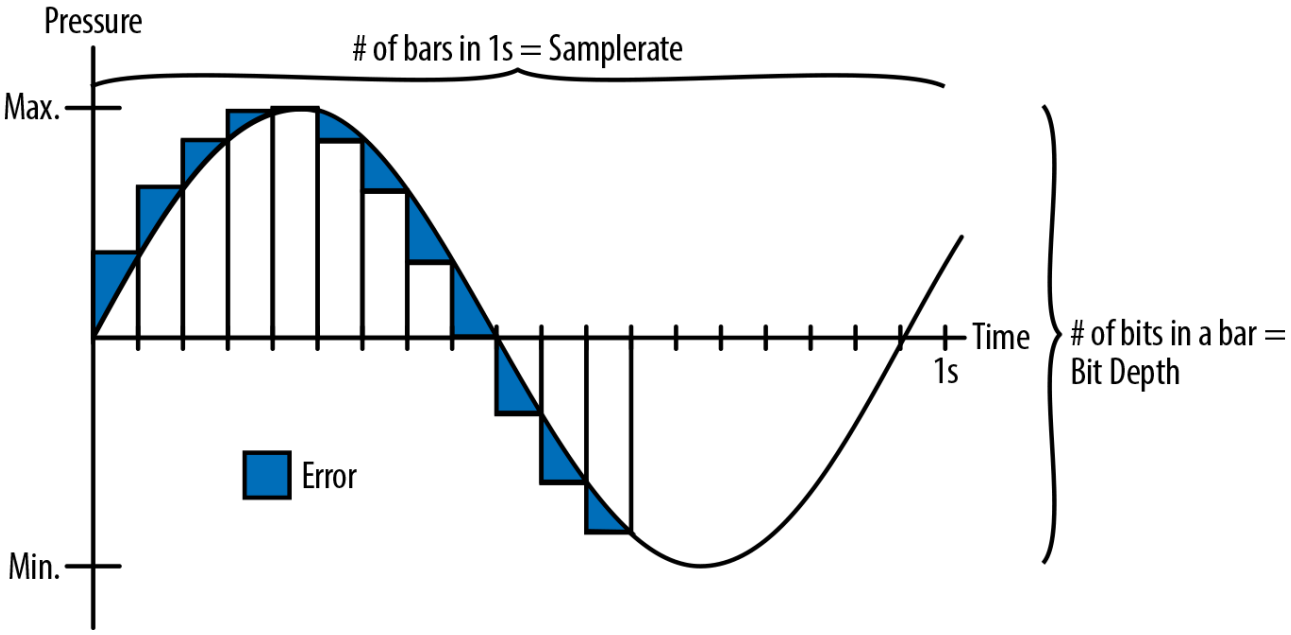

Рисунок 1-8. Аналоговый звук, преобразуемый в цифровой

Как видно из Рисунка 1-8, квантованая версия сигнала отличается от исходного аналогового. Эта разница (на графике выделена синим) уменьшается при увеличении частоты дискретизации и битовой глубины. Однако, уменьшение разницы может увеличить размер файла для хранения звука. Поэтому, к примеру, телефонные системы часто использовали частоту дискретизации всего 8 kHz, так как диапазон частот для передачи человеческого голоса намного меньше полного диапазона слышимых человеческим ухом частот (обычно от 20 Hz до 20 kHz).

Для оцифровки звука компьютеры могут оперировать длинными массивами чисел. Этот способ кодирования звука называется *импульсно-кодовой модуляцией* (pulse-code modulation, PCM). В Web Audio API такие массивы с числами используются в абстракции `AudioBuffer`. Аудиобуферы могут хранить множество аудиоканалов (чаще всего в стерео — правый и левый каналы), представленных в виде массива

чисел с плавающей точкой в диапазоне от –1 до 1. Тот же сигнал может быть представлен в виде массива целых чисел: если глубина битности 16 бит, то они окажутся в диапазоне от -2^{15} до $2^{15} - 1$.

Форматы кодирования аудио

Необработанное аудио в формате PCM занимает слишком много места, поэтому чаще всего аудио хранится в сжатых форматах. Существует два типа сжатия: *lossy* (с потерями) and *lossless* (без потерь). Сжатие без потерь (например, FLAC) гарантирует идентичность бит у сжатого и восстановленного звука. Сжатие с потерями (например, MP3) использует особенности человеческого слуха, чтобы сохранить дополнительное место, вырезав биты сигнала, которые скорее всего не будут услышаны. Форматы с потерями в целом подходят для большинства слушателей, кроме некоторых аудиофилов.

Метрика степени сжатия звука называется *битрейт* (bit rate) и обозначает количество бит аудио, необходимых для передачи звука, длительностью в секунду. Чем выше битрейт, тем больше данных может быть выделено на единицу времени, тем меньше степень сжатия и выше качество звука. Такие форматы как MP3 часто описываются используемым битрейтом (128 или 192 Kb/s). Форматы с потерями позволяют кодировать сигнал с произвольным битрейтом. Например, для передачи человеческого голоса по телефону часто используется битрейт 8 Kb/s. Такие форматы как OGG поддерживают вариативный битрейт, который может меняться со временем. Не путайте этот параметр с частотой дискретизации или разрядностью (см. Что такое звук?)!

Поддержка различных аудиоформатов в браузерах значительно различается. Как правило, если Web Audio API реализован в браузере, он использует тот же механизм загрузки, что и тег `<audio>`, поэтому матрица поддержки форматов у `<audio>` и Web Audio API совпадает. Обычно WAV (простой, без потерь и, как правило, несжатый формат) поддерживается во всех браузерах. Формат MP3 по-прежнему обременён патентами, поэтому он недоступен в некоторых полностью открытых браузерах (например, Firefox и Chromium). К сожалению, менее популярный, но свободный от патентов формат OGG до сих пор не поддерживается в Safari на момент написания этого текста.

С более актуальным списком поддерживаемых аудиоформатов можно ознакомиться здесь: https://developer.mozilla.org/en-US/docs/Web/Media/Guides/Formats.

Загрузка и воспроизведение звуков

Web Audio API чётко разделяет аудиобуферы и узлы источников. Идея такой архитектуры в том, чтобы разделить аудиоресурс от состояния воспроизведения. Если рассматривать это на примере винилового проигрывателя, буферы можно сравнить с пластинками, а источник звука — со считывающими головками проигрывателя. Разница в том, что в мире Web Audio API вы можете проигрывать одну и ту же пластинку на каком угодно количестве считывающих головок одновременно! Поскольку во многих приложениях требуется одновременное воспроизведение нескольких версий одного и того же буфера, этот подход является ключевым. Например, если вы хотите, чтобы звуки подпрыгивающего мяча срабатывали в быстрой последовательности, достаточно загрузить звуковой буфер один раз и запланировать воспроизведение нескольких источников на его основе (см. Множество одновременных звуков с вариациями).

Для загрузки аудиосемпла в Web Audio API мы можем использовать `XMLHttpRequest` и обработать результаты с помощью `context.decodeAudioData`. Всё это будет происходить асинхронно и не будет блокировать основой поток:

```
var request = new XMLHttpRequest();
request.open('GET', url, true);
request.responseType = 'arraybuffer';

// Декодируем асинхронно
request.onload = function() {
  context.decodeAudioData(request.response, function(theBuffer) {
    buffer = theBuffer;
  }, onError);
}

request.send();
```

Аудиобуферы — это лишь один из возможных источников воспроизведения. Другие источники включают прямой вход с микрофона или линейного входа, а также тег `<audio>` и другие (см. Интеграция с другими технологиями).

После загрузки буфера вы можете создать для него соответствующий аудиоузел `AudioBufferSourceNode`, подключить к вашему аудиографу и вызвать `start(0)` у узла аудиоисточника. Для остановки проигрывания вызовите `stop(0)`. Оба этих метода принимают параметр, обозначающий время старта/остановки проигрывания в секундах относительно системы координат текущего аудиоконтекста (см. Идеальный тайминг и задержка):

```
function playSound(buffer) {
  var source = context.createBufferSource();

  source.buffer = buffer;
  source.connect(context.destination);
  source.start(0);
}
```

В играх часто используется фоновая музыка, играющая по циклу. Однако стоит быть осторожнее с чрезмерной повторяемостью: если игрок застрянет в каком-то уровне или области, и один и тот же трек будет непрерывно звучать на фоне, это может начать вызывать раздражение. В таких случаях имеет смысл постепенно приглушать трек, чтобы избежать переутомления слушателя. Другой стратегией может быть использование различных вариантов трека с разной интенсивностью и плавных переходов между ними в зависимости от ситуации в игре (см. Постепенно меняющиеся параметры аудио).

Собираем всё вместе

Как видно из предыдущих примеров кода, для воспроизведения звуков с помощью Web Audio API требуется произвести некоторую настройку. В реальной игре имеет смысл реализовать абстракцию на JavaScript, которая упростит дальнейшую работу с этим API. Один из примеров такой абстракции — класс `BufferLoader`. Он объединяет все этапы в простой загрузчик, который принимает набор путей к аудиофайлам и возвращает соответствующие аудиобуферы. Вот пример того, как можно использовать такой класс:

```
window.onload = init;

var context;
var bufferLoader;

function init() {
  context = new webkitAudioContext();

  bufferLoader = new BufferLoader(
    context,
    [
      '../sounds/hyper-reality/br-jam-loop.wav',
      '../sounds/hyper-reality/laughter.wav',
    ],
    finishedLoading
  );

  bufferLoader.load();
}

function finishedLoading(bufferList) {
  // Создадим два источника аудио и воспроизведём их одновременно
  var source1 = context.createBufferSource();
  var source2 = context.createBufferSource();

  source1.buffer = bufferList[0];
  source2.buffer = bufferList[1];

  source1.connect(context.destination);
  source2.connect(context.destination);

  source1.start(0);
  source2.start(0);
}
```

Простейший пример реализации `BufferLoader` можно посмотреть тут: http://webaudioapi.com/samples/shared.js.

Идеальный тайминг и задержка

В отличие от тега `<audio>`, Web Audio API предлагает модель с низкой задержкой и высокой точностью управления временем.

Низкая задержка очень важна для игр и других интерактивных приложений, так как в них часто требуется максимально быстро озвучивать действия пользователя. Если обратная связь не произойдёт немедленно, пользователь ощутит задержку, что может привести к разочарованию и фрустрации. На практике, по причине несовершенства человеческого слухового аппарата, допустима задержка примерно до 20 мс, но этот предел зависит от многих факторов.

Точное управление временем позволяет планировать события в конкретном времени в будущем. Это очень важно для заранее подготовленных сцен и в музыкальных приложениях.

Модель измерения времени

Одним из ключевых элементов, которые предоставляет аудиоконтекст, является согласованная модель измерения времени и единая система отсчёта. Важно, что эта модель отличается от привычных в JavaScript таймеров, таких как `setTimeout`, `setInterval` и `new Date()`. Она также отличается от счётчика времени, предоставляемого `window.performance.now()`.

Любое абсолютное время, с которым вы будете работать внутри Web Audio API, измеряется в секундах и находится внутри координатной системы конкретного аудиоконтекста. Текущее время может быть извлечено из аудиоконтекста с помощью свойства `currentTime`. Хоть время измеряется в секундах, оно хранится в виде чисел с плавающей точкой с большой степенью точности.

Точное воспроизведение и возобновление проигрывания

Функция `start()` позволяет легко запланировать проигрывание звука в точное время. Для начала необходимо проверить, что ваши звуковые буферы предзагружены (см. Загрузка и воспроизведение звуков). Без этого вам придётся

ждать неизвестное количество времени, пока браузер загрузит звуковой файл и пока Web Audio API декодирует его. Если вы хотите, чтобы звук проигрался в конкретное время, это может не сработать, если аудиобуфер всё ещё не готов.

Звуки могут быть запланированы к проигрыванию в конкретное время с помощью первого параметра `when` функции `start()`. Значение этого параметра указывается относительно свойства `currentTime` текущего `AudioContext`. Если указанное время меньше `currentTime`, воспроизведение начнётся немедленно. Таким образом, `start(0)` всегда проиграет звук сразу после вызова. Для того, чтобы запланировать проигрывание звука через 5 секунд — вызовите `start(context.currentTime + 5)`.

Аудиобуферы могут быть воспроизведены начиная с заданного смещения, если функция `start()` вызвана с вторым параметром `offset`, а если передан ещё и третий параметр `duration`, звук будет воспроизводиться только заданное количество секунд. Например, если мы хотим приостановить звук и затем воспроизвести его с того же места, где остановились, мы можем реализовать паузу, отслеживая, сколько времени звук воспроизводился в текущей сессии, а также запоминая последний смещённый момент времени для последующего возобновления проигрывания:

```
// Предположим, что буфер предзагружен внутри заранее созданного
контекста
var startOffset = 0;
var startTime = 0;

function pause() {
  source.stop();

  // Измеряем, сколько времени прошло с последней паузы
  startOffset += context.currentTime - startTime;
}
```

Когда аудио закончит проигрываться, его уже нельзя будет воспроизвести повторно. Для того, чтобы воспроизвести аудиобуфер ещё раз, необходимо создать новый узел аудиоисточника (`AudioBufferSourceNode`) и вызвать `start()`:

```
function play() {
  startTime = context.currentTime;
  var source = context.createBufferSource();

  // Подключаем граф
  source.buffer = this.buffer;
  source.loop = true;
  source.connect(context.destination);

  // Начинаем воспроизведение, но только убедившись,
  // что мы остаёся в пределах исходного аудиобуфера
  source.start(0, startOffset % buffer.duration);
}
```

Хотя пересоздание узла аудиоисточника может показаться неэффективным, важно помнить, что эти узлы отлично оптимизированы для работы в подобном стиле. Запомните, что если вы продолжаете работать с тем же аудиобуфером, вам не нужно обращаться за новым ресурсом, чтобы снова воспроизвести тот же самый звук. Таким образом, вы получаете чёткое разделение между буфером и проигрывателем и можете спокойно воспроизводить разные версии одного и того же буфера. Если вы заметите, что постоянно повторяете этот паттерн, вы можете реализовать простую функцию `playSound(buffer)` с ранее продемонстрированными примерами кода.

Планирование ритмически организованного проигрывания

Web Audio API позволяет разработчикам точно планировать воспроизведение в будущем. Давайте настроим простой ритмический трек, чтобы это продемонстрировать. Возможно простейший и самый известный ритмический паттерн (см. Рисунок 2-1) — это хай-хет, который играется каждую восьмую ноту и кик со снэйром, которые играются чередующимися четверными нотами в тактовом размере 4/4.

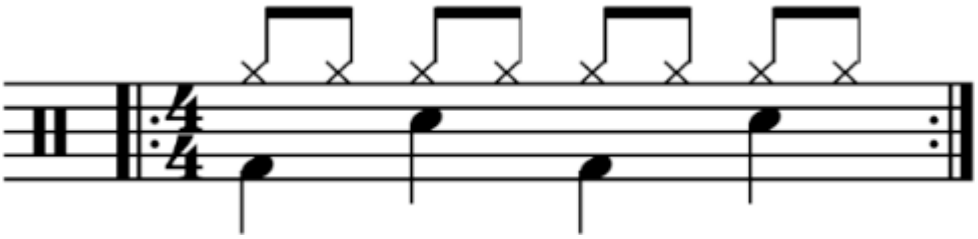

Рисунок 2-1. Музыкальная нотация для одного из самых простых ритмических паттернов

Предположим, мы уже предзагрузили звуки кика, снэйра и хай-хета в буфер и теперь напишем код, который реализует этот ритмический рисунок:

```
for (var bar = 0; bar < 2; bar++) {
  var time = startTime + bar * 8 * eighthNoteTime;

  // Проиграем басовый барабан (кик) на счёт 1 и 5
  playSound(kick, time);
  playSound(kick, time + 4 * eighthNoteTime);

  // Проиграем малый барабан (снэйр) на счёт 3 и 7
  playSound(snare, time + 2 * eighthNoteTime);
  playSound(snare, time + 6 * eighthNoteTime);

  // Проиграем хай-хет каждую восьмую ноту
  for (var i = 0; i < 8; ++i) {
    playSound(hihat, time + i * eighthNoteTime);
  }
}
```

После того как вы запланировали воспроизведение звука, отменить это событие уже нельзя. Поэтому, если вы работаете с приложением, в котором всё быстро меняется, не рекомендуется планировать звуки слишком надолго вперёд. Хорошее решение этой проблемы — реализовать собственный планировщик с помощью таймеров JavaScript и очереди событий. Этот подход описан в Сказке о двух часах.

Изменение параметров аудио

У разных аудиоузлов есть возможность менять параметры. Например, у узла `GainNode` можно менять параметр усиления, задающий коэффициент громкости для всех звуков, которые проходят через этот узел. Например, значение усиления

равное `1` никак не повлияет на амплитуду звука, значение `0.5` уменьшит её вдвое, а значение `2` увеличит её в два раза (см. Громкость, усиление и воспринимаемая звучность). Для демонстрации этого настроим аудиограф:

```
// Создадим узел усиления
var gainNode = context.createGain();

// Подключим аудиоисточник к узлу усиления
source.connect(gainNode);

// Подключим узел усиления к аудиовыходу
gainNode.connect(context.destination);
```

В Web Audio API параметры аудио представлены в виде экземпляров интерфейса `AudioParam`. Значения параметров можно менять напрямую через свойство `value` конкретного параметра:

```
// Уменьшим громкость в два раза
gainNode.gain.value = 0.5;
```

Значения также могут быть изменены отложенно. Мы можем использовать `setTimeout` для отложенного планирования изменения значений параметров, но это не даст необходимой точности по следующим причинам:

1. Тайминг с точностью до миллисекунд может оказаться недостаточным.
2. Основной поток выполнения JS может быть занят высокоприоритетными задачами, такими как: отрисовка страницы, сбор мусора и вызов колбеков из других API, что может замедлить работу таймера.
3. JS-таймеры зависят от состояния вкладки браузера. В фоновых вкладках таймеры замедляются.

Вместо того, чтобы устанавливать значения напрямую, мы можем использовать метод `setValueAtTime()`, который принимает в аргументах нужное для установки значение и стартовое время. К примеру, следующий код установит заданное значение усиления через одну секунду:

```
gainNode.gain.setValueAtTime(0.5, context.currentTime + 1);
```

Постепенно меняющиеся параметры аудио

Очень часто нужно менять параметры аудио не резко, а постепенно. К примеру, когда вы строите приложение, которое проигрывает музыку, вы можете захотеть добавить эффект затухания в конце проигрывания старого трека и эффект набирания громкости в начале проигрывания нового, чтобы избежать резкого перехода между ними. Конечно, можно добиться этого множественными вызовами метода `setValueAtTime()`, однако это неудобно.

Web Audio API предоставляет набор методов `RampToValue`, позволяющих менять любые параметры постепенно. Это методы `linearRampToValueAtTime()` и `exponentialRampToValueAtTime()`. Различие между ними заключается в характере перехода: линейном или экспоненциальном. В некоторых случаях экспоненциальный переход оказывается более уместным, поскольку многие свойства звука мы воспринимаем именно экспоненциально.

Давайте возьмём пример реализации кроссфейда (плавного перехода между аудиотреками с постепенным изменением громкости звука) и планирования его в будущем. При наличии плейлиста мы можем переходить между треками, запланировав затухание громкости для текущего трека и нарастание громкости для следующего, выполняя обе операции чуть раньше завершения текущего трека:

```
function createSource(buffer) {
  var source = context.createBufferSource();
  var gainNode = context.createGainNode();

  source.buffer = buffer;

  // Подключим источник к узлу усиления
  source.connect(gainNode);

  // Подключим узел усиления к аудиовыходу
  gainNode.connect(context.destination);

  return {
    source: source,
    gainNode: gainNode
  };
}

function playHelper(buffers, iterations, fadeTime) {
  var currTime = context.currentTime;

  for (var i = 0; i < iterations; i++) {
    // Для каждого буфера запланируем его воспроизведение в будущем
    for (var j = 0; j < buffers.length; j++) {
      var buffer = buffers[j];
      var duration = buffer.duration;
      var info = createSource(buffer);
      var source = info.source;
      var gainNode = info.gainNode;

      // Запланируем постепенное нарастание громкости
      gainNode.gain.linearRampToValueAtTime(0, currTime);
      gainNode.gain.linearRampToValueAtTime(1, currTime + fadeTime);

      // Запланируем постепенное затухание громкости
      gainNode.gain.linearRampToValueAtTime(1, currTime + duration-
fadeTime);
      gainNode.gain.linearRampToValueAtTime(0, currTime + duration);

      // Запускаем проигрывание трека
      source.noteOn(currTime);

      // Увеличиваем время для следующей итерации
      currTime += duration - fadeTime;
```

```
    }
  }
}
```

Произвольные кривые времени

Если вам не подходят методы для линейного или экспоненциального переходов, вы можете задать свою кривую изменения через задание массива чисел с помощью метода `setValueCurveAtTime()`. Этот метод позволит объединить множественные вызовы метода `setValueAtTime()`. Для примера, мы можем создать эффект тремоло, применив кривую осцилляции к параметру усиления звука через узел `GainNode` (см. Рисунок 2-2).

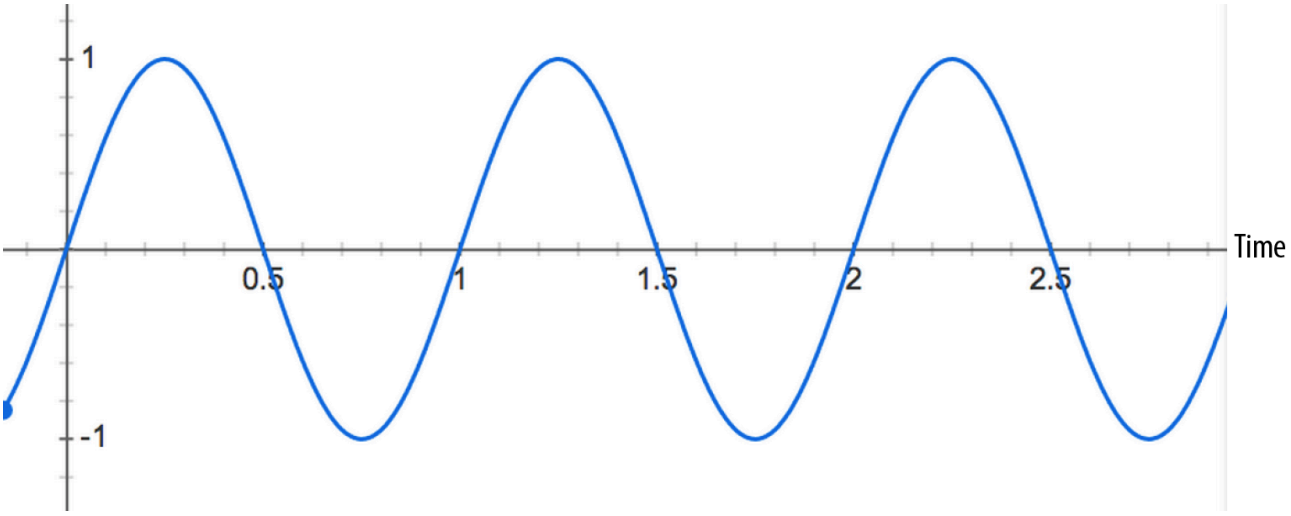

Рисунок 2-2. Кривая изменения параметра усиления звука

Давайте реализуем этот эффект в коде:

```
var DURATION = 2;
var FREQUENCY = 1;
var SCALE = 0.4;

// Разделим время на дискретные шаги в размере valueCount
var valueCount = 4096;

// Создадим кривую в форме синусоиды
var values = new Float32Array(valueCount);

for (var i = 0; i < valueCount; i++) {
  var percent = (i / valueCount) * DURATION*FREQUENCY;

  values[i] = 1 + (Math.sin(percent * 2*Math.PI) * SCALE);

  // Установим последнее значение в 1, чтобы восстановить
  // значение playbackRate к нормальному в конце
  if (i == valueCount - 1) {
    values[i] = 1;
  }
}

// Применим это к узлу усиления незамедлительно,
// и укажем время работы функции в 2 секунды
this.gainNode.gain.setValueCurveAtTime(
  values,
  context.currentTime,
  DURATION
);
```

В данном примере кода мы вручную задали синусоиду и применили её к параметру усиления, чтобы создать эффект тремоло. Это потребовало некоторых математических знаний.

Это приводит нас к довольно изящной возможности Web Audio API, которая позволяет реализовать этот эффект намного проще. Мы можем взять любой аудиопоток, который подключён к другому `AudioNode` и вместо этого подключить его к любому `AudioParam`. Эта важная возможность позволяет создавать множество интересных эффектов. Предыдущий код — это пример так называемого низкочастотного осциллятора (LFO), который применён к узлу усиления. LFO используется в создании таких эффектов как вибрато, смещение

фазы (фейзинг) и тремоло. Используя встроенный узел осциллятора (см. Прямой синтез звука на основе осциллятора), мы можем легко перестроить предыдущий пример с тремоло:

```
// Создадим осциллятор
var osc = context.createOscillator();
osc.frequency.value = FREQUENCY;

var gain = context.createGain();
gain.gain.value = SCALE;

osc.connect(gain);
gain.connect(this.gainNode.gain);

// Запускаем эффект сразу и останавливаем через 2 секунды
osc.start(0);
osc.stop(context.currentTime + DURATION);
```

Этот подход более эффективен, чем создание произвольных кривых и позволяет избежать ручного вычисления синусоид только для того, чтобы просто зациклить эффект.

Громкость и воспринимаемая звучность

После того как звук готов к воспроизведению — из `AudioBuffer` или любого другого источника — один из основных параметров, который мы можем регулировать, это субъективно воспринимаемая звучность звука.

Основной способ повлиять на звучность — использовать узлы `GainNode`. Как ранее упоминалось, эти узлы имеют параметр `gain`, который является множителем амплитуды приходящего звука. Значение `1` — начальное, оно никак не влияет на амплитуду. Значения между `0` и `1` уменьшают звучность, а значения больше `1` — усиливают. Отрицательные значения `gain` инвертируют форму звуковой волны (т.е. амплитуда меняется на противоположную).

Громкость, усиление и воспринимаемая звучность

Давайте начнём с определений. *Воспринимаемая звучность* (loudness) — это субъективная оценка того, насколько интенсивно наши уши ощущают звук. *Громкость* (volume) — это измерение физической амплитуды звуковой волны. *Усиление* (gain) — это множитель, который влияет на амплитуду звука во время его обработки.

Другими словами, при усилении амплитуда звуковой волны масштабируется, а значение усиления используется в качестве множителя. Посмотрим на пример изменения формы звуковой волны при усилении амплитуды в два раза на Рисунке 3-1.

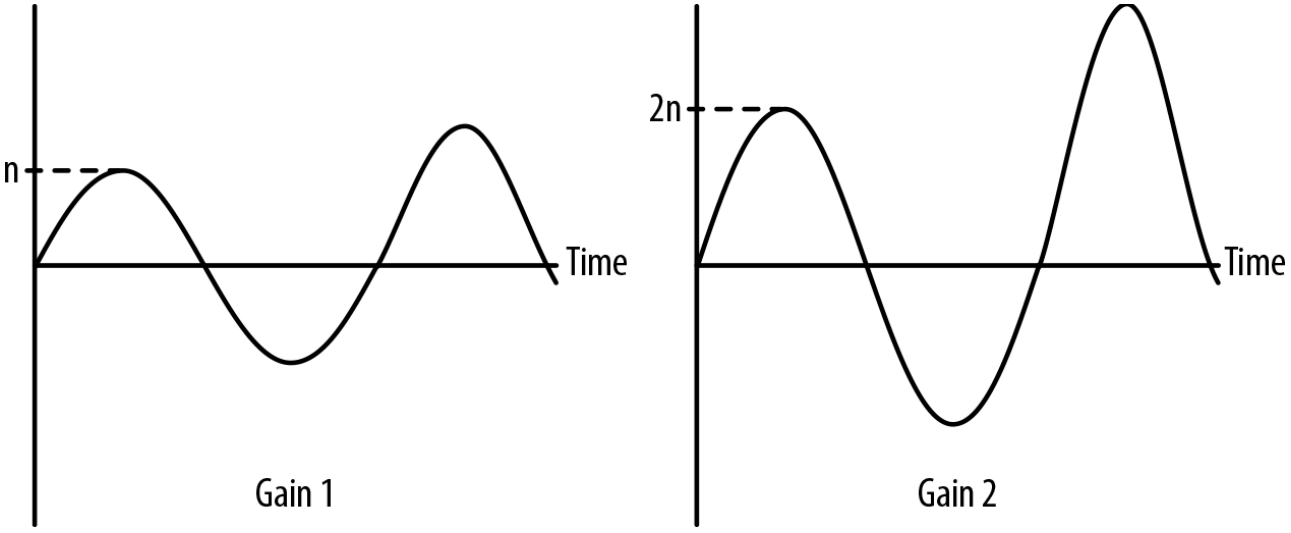

Рисунок 3-1. Слева: оригинальная форма звуковой волны, справа: усиленная в два раза

Мощность волны измеряется в децибелах (сокращенно `dB`) или в одной десятой бела — единице измерения, названной в честь Александра Грэхема Белла. Децибелы — относительная логарифмическая единица измерения, которая сравнивает уровень мощности волны с некоторой опорной точкой. Существует множество различных опорных точек для измерения `dB`, и тип опорной точки обозначается в виде дополнительного суффикса вместе с единицей измерения. Значение в `dB` бессмысленно без знания этой референсной точки. К примеру, `dBV`, `dBu` и `dBm` полезны для измерения электрических сигналов. Но так как мы имеем дело с цифровым аудио, обычно мы будем использовать два типа измерений: `dBFS` и `dBSPL`.

Первый тип — это `dBFS` (*decibels full scale, децибелы относительно полной шкалы*). Максимально возможный уровень звука, который может воспроизвести оборудование, равен `0 dBFS`. Все остальные уровни выражаются отрицательными числами.

Математически `dBFS` вычисляются как:

```
dBFS = 20 * log( [sample level] / [max level] )
```

Максимальное значение `dBFS` в 16-битной аудиосистеме равно:

```
max = 20 * log(1111 1111 1111 1111 / 1111 1111 1111 1111) = log(1) = 0
dBFS
```

Обратите внимание, что максимальное значение `dBFS` равно `0` по определению, так как `log(1) = 0`. Минимальное значение `dBFS` в той же аудиосистеме равно:

```
min = 20 * log(0000 0000 0000 0001 / 1111 1111 1111 1111) = -96 dBFS
```

`dBFS` — величина усиления, а не громкости. Вы можете воспроизвести сигнал мощностью `0 dBFS` через ваше стереоустройство с очень низким уровнем усиления и почти ничего не услышите. И наоборот, можно проиграть сигнал на уровне `-30 dBFS` с максимальным усилением и оглушить себя.

Тем не менее, вы, вероятно, слышали, как кто-то описывал громкость звука в децибелах. Технически речь шла о `dBSPL` — типе измерения мощности звука в децибелах относительно уровня звукового давления. Здесь опорой является точка в `0.000002 ньютона на кв. м.` (что примерно соответствует звуку летящего комара на расстоянии 3 метров). У `dBSPL` нет верхнего предела, но на практике важно оставаться ниже уровня, при котором может повредиться слух (около `120 dBSPL`) и значительно ниже уровня болевого порога (около `150 dBSPL`). Web Audio API не использует `dBSPL`, поскольку итоговая громкость звука зависит от усиления на уровне ОС и самих колонок; API работает только с `dBFS`.

Логарифмическое определение децибелов в какой-то мере соответствует тому, как наши уши воспринимают звучность, но сама воспринимаемая звучность остаётся весьма субъективным понятием. Если сравнить значения в `dB` для звука и для него же с усилением в 2 раза, мы увидим прибавку примерно на `6 dB`:

```
diff = 20 * log(2 / 2^16) - 20 * log(1 / 2^16) = 6.02 dB
```

Каждый раз, когда мы добавляем примерно по `6 dB`, мы по сути удваиваем амплитуду сигнала. Сравнивая мощность звука на рок-концерте (`~110 dBSPL`) с мощностью звука будильника (`~80 dBSPL`), разница между ними будет в `(110 - 80) / 6 dB`, или примерно в пять раз — в этом случае множитель усиления будет равен `2^5 = 32x`. Ручка громкости на стереоколонках также калибрует амплитуды экспоненциально. Другими словами, поворачивая ручку громкости на 3 пункта выше, вы умножаете амплитуду сигнала примерно в 8 раз (`2^3`). На практике, описанная здесь экспоненциальная модель является всего лишь приближением к тому, как наши уши воспринимают звучность, и производители аудиооборудования часто используют собственные кривые усиления, которые не являются ни линейными, ни экспоненциальными и пересекаются на более высоком уровне амплитуды.

Кроссфейд с равной мощностью

Часто в игре может возникнуть ситуация, когда необходимо осуществить плавный переход между двумя средами, с которыми связано разное звуковое сопровождение. Однако заранее неизвестно, когда и насколько нужно делать

кроссфейд — возможно, это будет зависеть от положения игрового персонажа, которым управляет игрок. В таком случае мы не сможем реализовать автоматический переход.

Применение прямого линейного затухания приведёт к графику на Рисунке 3-2, в котором звук может быть несбалансированным из-за провала громкости между двумя семплами.

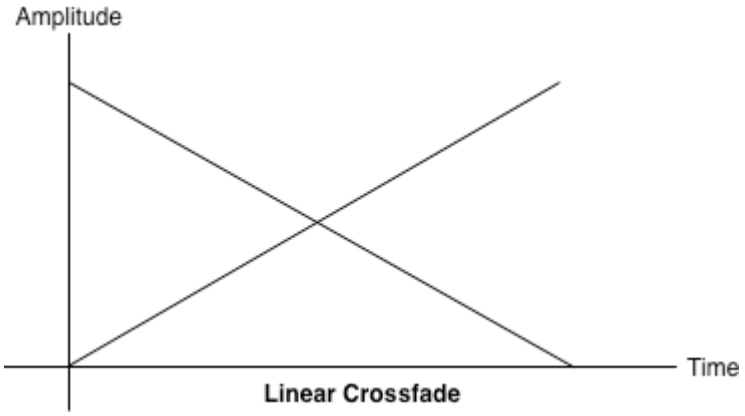

Рисунок 3-2. Линейный кроссфейд между двумя треками

Чтобы решить эту проблему, мы можем использовать кривую равной мощности, в которой кривые усиления не являются ни линейными, ни экспоненциальными, а пересекаются при более высокой амплитуде. Это поможет избежать провала громкости в средней части кроссфейда, когда оба звука смешиваются в равной степени, как на Рисунке 3-3.

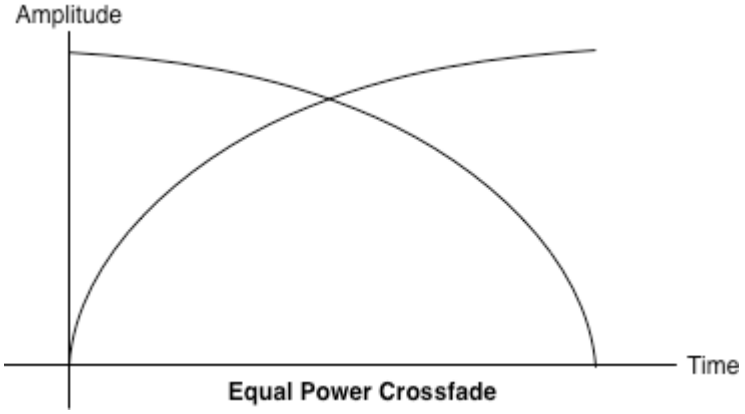

Рисунок 3-3. Кроссфейд с равной мощностью звучит намного лучше

График на Рисунке 3-3 может быть сгенерирован в коде с помощью не очень сложных вычислений:

```
function equalPowerCrossfade(percent) {
  // Используем кривую кроссфейда равной мощности
  var gain1 = Math.cos(percent * 0.5 * Math.PI);
  var gain2 = Math.cos((1.0 - percent) * 0.5 * Math.PI);

  this.ctl1.gainNode.gain.value = gain1;
  this.ctl2.gainNode.gain.value = gain2;
}
```

Клиппинг и измерение уровня сигнала

Подобно изображениям, выходящим за границы холста, звуки тоже могут быть обрезаны (клиппированы), если их волна превышает максимальный допустимый уровень. Возникающее при этом характерное искажение считается явно нежелательным. Поэтому аудиооборудование часто снабжено индикаторами, которые показывают величину уровней сигнала и помогают инженерам и слушателям избежать клиппинга. Эти индикаторы называются измерителями (см. Рисунок 3-4) и обычно имеют зелёную зону (норма, без клиппинга), жёлтую зону (близко к клиппингу) и красную зону (клиппинг).

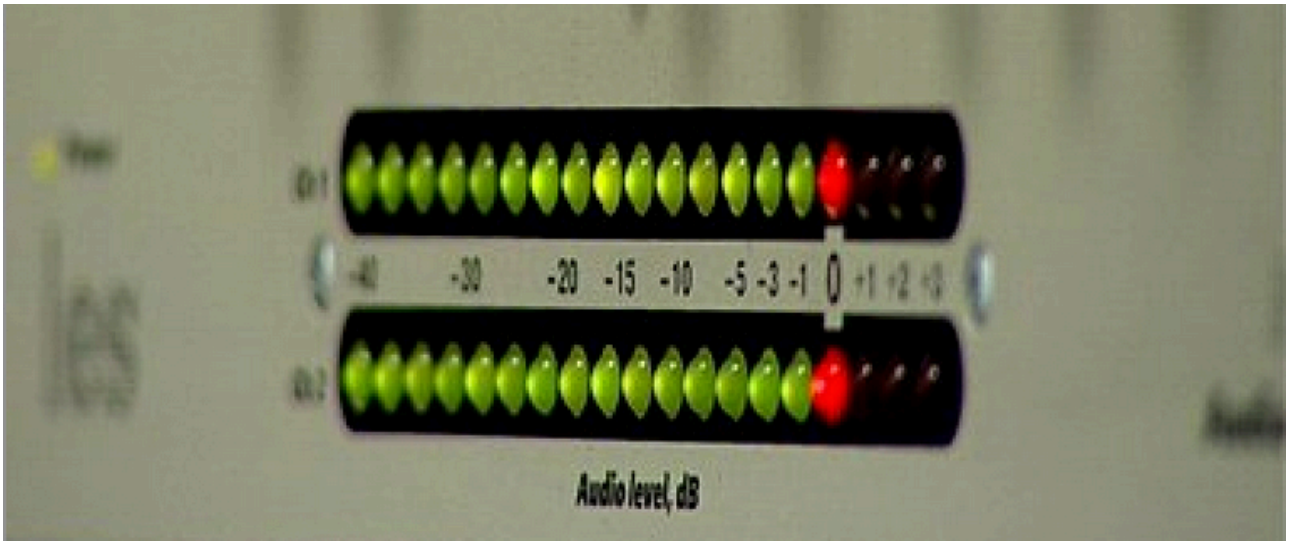

Рисунок 3-4. Измеритель в типовом радиоприёмнике

Обрезанные звуки выглядят плохо на мониторе и так же плохо звучат. Очень важно обращать внимание на резкие искажения, или, наоборот, на слишком приглушённые миксы, которые заставляют слушателя повышать громкость.

Использование измерителей для обнаружения и предотвращения клиппинга

Из-за того, что множество звуков, проигрываемых одновременно, суммируются без снижения уровня, вы можете оказаться в ситуации, когда превысите порог возможностей вашего динамика. Максимальный уровень звука равен `0 dBFS` или `2^16` для 16-битной аудиосистемы. В версии сигнала с плавающей точкой эти значения битов находятся в диапазоне `[-1, 1]`. Волновая форма звука, подвергшегося клиппингу, выглядит примерно так, как показано на Рисунке 3-5. В контексте Web Audio API, клиппинг происходит тогда, когда значения, отправленные на узел аудиовыхода, выходят за пределы допустимого диапазона. Хорошей практикой считается оставлять небольшой запас (так называемый *headroom*) в финальном миксе, чтобы не приближаться слишком близко к порогу клиппинга.

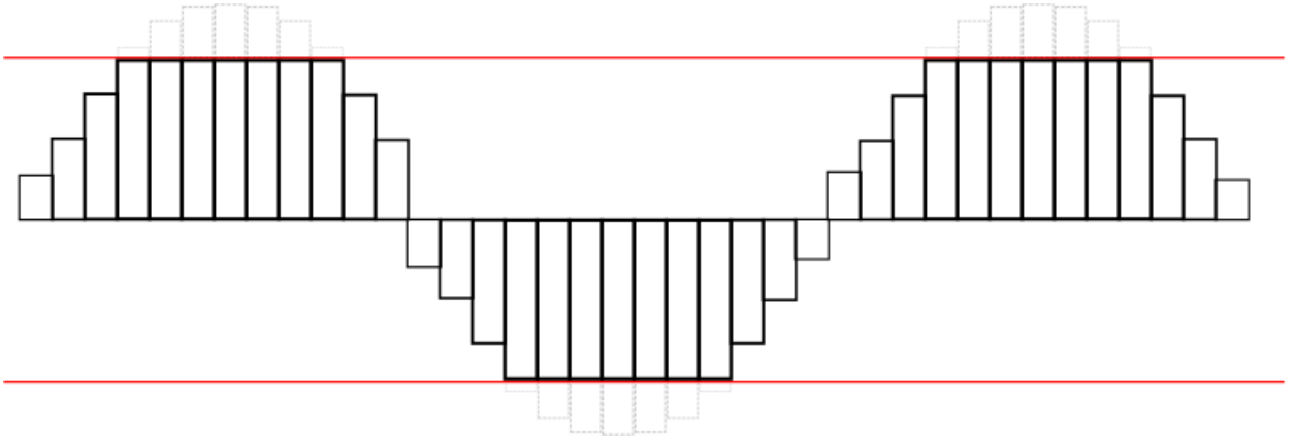

Рисунок 3-5. Волновая форма звука, подвергшегося клиппингу

Помимо непосредственного прослушивания, вы можете определить наличие клиппинга и программно — добавив в свой аудиограф узел `ScriptProcessorNode` *(прим. переводчика: этот узел устарел и заменён на* `AudioWorklet` *в новых версиях Web Audio API)*. Клиппинг может возникнуть, если какие-либо значения PCM-буфера выходят за пределы допустимого диапазона. В этом примере мы проверяем левый и правый каналы на наличие клиппинга и, если он обнаружен, сохраняем время обнаружения последнего клиппинга:

```
function onProcess(e) {
  var leftBuffer = e.inputBuffer.getChannelData(0);
  var rightBuffer = e.inputBuffer.getChannelData(1);

  checkClipping(leftBuffer);
  checkClipping(rightBuffer);
}

function checkClipping(buffer) {
  var isClipping = false;

  // Проходим по аудиобуферу в цикле, чтобы проверить,
  // вышло ли какое-либо значение из допустимого диапазона
  for (var i = 0; i < buffer.length; i++) {
    var absValue = Math.abs(buffer[i]);

    if (absValue >= 1.0) {
      isClipping = true;
      break;
    }
  }

  this.isClipping = isClipping;

  if (isClipping) {
    lastClipTime = new Date();
  }
}
```

Альтернативная реализация измерения уровня сигнала может опрашивать в реальном времени анализатор в аудиографе, используя метод `getFloatFrequencyData` во время рендера, синхронизированного с `requestAnimationFrame` (см. Анализ и визуализация). Этот подход более эффективен, но может привести к потере большой части сигнала (включая места, где он потенциально обрезается). Это может происходить тогда, когда аудиосигнал меняется намного быстрее, чем вызывается рендеринг (что происходит примерно 60 раз в секунду).

Рабочий способ для предотвращения клиппинга — это уменьшение общего уровня сигнала. Для борьбы с клиппингом нужно применить некоторое усиление на основном (мастер) узле усиления `GainNode`, чтобы снизить уровень микса до

уровня, который уберёт клиппинг. Короче говоря, вы должны подобрать нужный коэффициент, чтобы предвидеть наихудший сценарий, однако сделать это правильно — это скорее искусство, чем наука. На практике, поскольку звуки, которые воспроизводятся в игре или интерактивном приложении, могут зависеть от множества факторов, которые определяются во время выполнения, может быть сложно выбрать значение мастер-усиления, способное предотвратить клиппинг во всех случаях. Для таких непредсказуемых ситуаций стоит использовать динамическую компрессию (см. Динамическая компрессия).

Понятие динамического диапазона

В мире аудио динамический диапазон означает разницу между самыми громкими и самыми тихими участками звука. Объём динамического диапазона в музыкальных произведениях сильно варьируется в зависимости от жанра. Классическая музыка имеет большой динамический диапазон и часто содержит очень тихие фрагменты, за которыми следуют относительно громкие. Во множестве популярных жанров типа рока или электроники обычно задействован небольшой динамический диапазон. Музыка этих жанров обычно довольно громкая из-за конкуренции за слушателя (так называемая «Война Звучности») и для удовлетворения запросов потребителей. Такая равномерная звучность обычно достигается за счёт сжатия динамического диапазона.

Тем не менее, существует множество оправданных применений компрессии. Иногда записанная музыка имеет настолько широкий динамический диапазон, что некоторые фрагменты звучат так тихо или громко, что слушателю постоянно приходится держать палец на ручке громкости. Компрессия может сделать тише громкие участки и поднять громкость тихих участков так, чтобы они стали слышимы. На Рисунке 3-6 изображена исходная волна (сверху) и та же самая волна с применённой компрессией (внизу). Можно увидеть, что звук стал в целом громче, а в амплитуде стало меньше отличающихся участков.

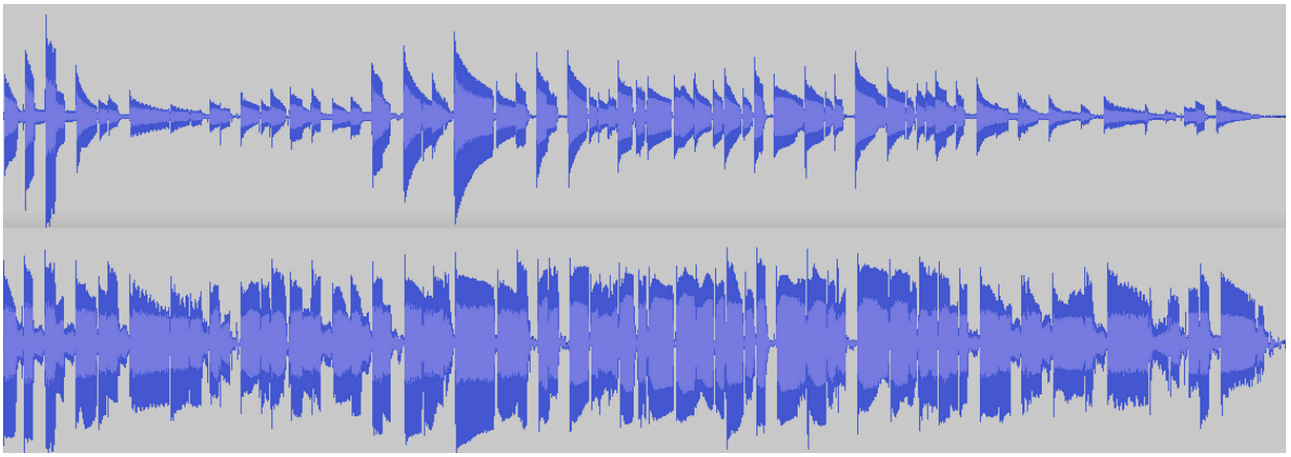

Рисунок 3-6. Эффект от динамической компрессии

В играх и интерактивных приложениях вы можете не знать заранее, как будет выглядеть исходный звук. Из-за динамической природы игр, у вас могут быть очень тихие участки (например, в режиме стелса), за которыми сразу следуют очень громкие (например, в режиме боя). Узел компрессора может быть полезен в таких неожиданных ситуациях, для того чтобы снизить вероятность клиппинга (см. Клиппинг и измерения уровня сигнала).

Компрессоры могут быть смоделированы с помощью кривой компрессии с несколькими параметрами, каждый из которых можно настраивать в Web Audio API. Два основных параметра компрессии — это порог и коэффициент сжатия. Порог отвечает за значение самой низкой громкости, с которого компрессор начнёт уменьшать динамический диапазон. Коэффициент сжатия определяет степень снижения усиления, применяемую компрессором. На Рисунке 3-7 показано влияние порога и различных коэффициентов сжатия на кривую компрессии.

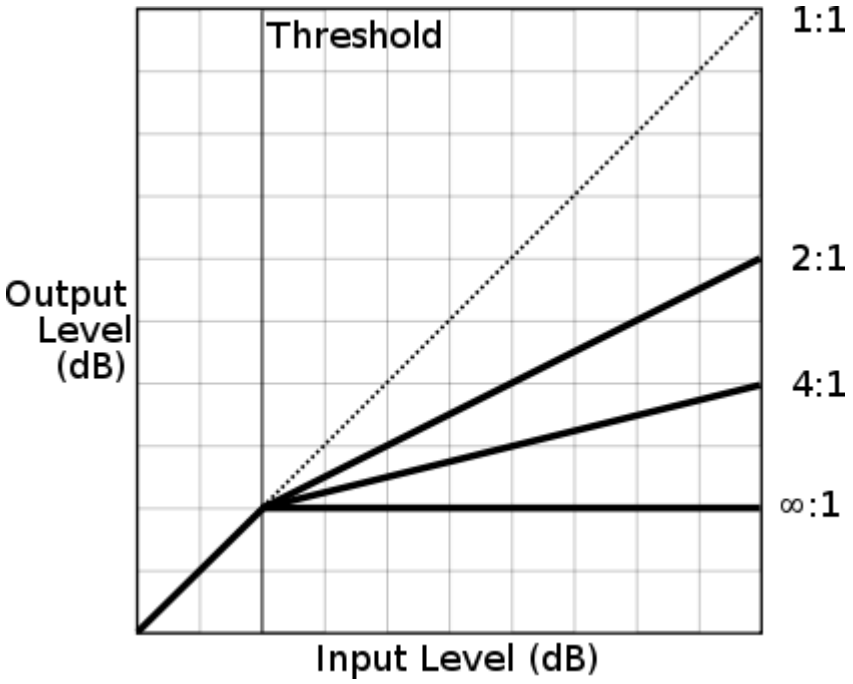

Рисунок 3-7. Пример кривой компрессии с базовыми параметрами

Динамическая компрессия

Компрессоры доступны в Web Audio API в виде узлов `DynamicsCompressorNode`. Использование умеренного количества динамической компрессии в вашем миксе является хорошей идеей, особенно в играх, в которых, как ранее упоминалось, вы не знаете заранее какие звуки будут проиграны и когда. Случай, в котором компрессия нежелательна, — это если вы имеете дело с кропотливо отмастеренными треками, которые уже настроены на правильное звучание и не микшируются с другими треками.

Реализация динамической компрессии в Web Audio API доступна через подключение узла динамической компрессии к вашему аудиографу, обычно в виде последнего узла перед аудиовыходом:

```
var compressor = context.createDynamicsCompressor();

mix.connect(compressor);
compressor.connect(context.destination);
```

Узел компрессора может быть инициализирован с дополнительными параметрами, однако параметры по умолчанию уже достаточно хорошо подходят в большинстве случаев. Дополнительную информацию о конфигурации кривой компрессии можно

посмотреть в спецификации Web Audio API.

Высота звука и частотный спектр

К этому моменту мы уже изучили особенности некоторых свойств звука, а именно тайминг и громкость. Чтобы делать более сложные вещи со звуком, такие как эквализация (например, увеличение низких, басовых частот и уменьшение высоких), нам нужны более сложные инструменты. В этом разделе мы рассмотрим некоторые инструменты, которые позволяют производить более интересные преобразования звука, включая возможность имитировать различные виды сред и манипулировать звуками напрямую с помощью JavaScript.

Основы высоты звука в музыке

Музыка состоит из множества звуков, воспроизводимых в разное время и одновременно. Звуки, которые извлекаются из музыкальных инструментов, могут быть очень сложными, так как звук отражается от разных частей инструмента и его форма становится уникальной. Однако эти музыкальные тоны имеют одну общую черту: физически они представляют собой периодические волновые формы. Эта периодичность воспринимается нашими ушами как высота звука. Высота измеряется частотой колебаний волны или числом повторений волнового рисунка в секунду, которое указывается в герцах. Частота — это время (в секундах) между гребнями волны. На Рисунке 4-1 видно, что, если мы разделим волну пополам по шкале времени, то получим удвоенную частоту, которая будет звучать как исходный тон, но на одну октаву выше. И наоборот, если уменьшить частоту волны в два раза, тон понизится на октаву. Таким образом, высота звука (как и громкость) воспринимается ушами экспоненциально: на каждой октаве частота удваивается.

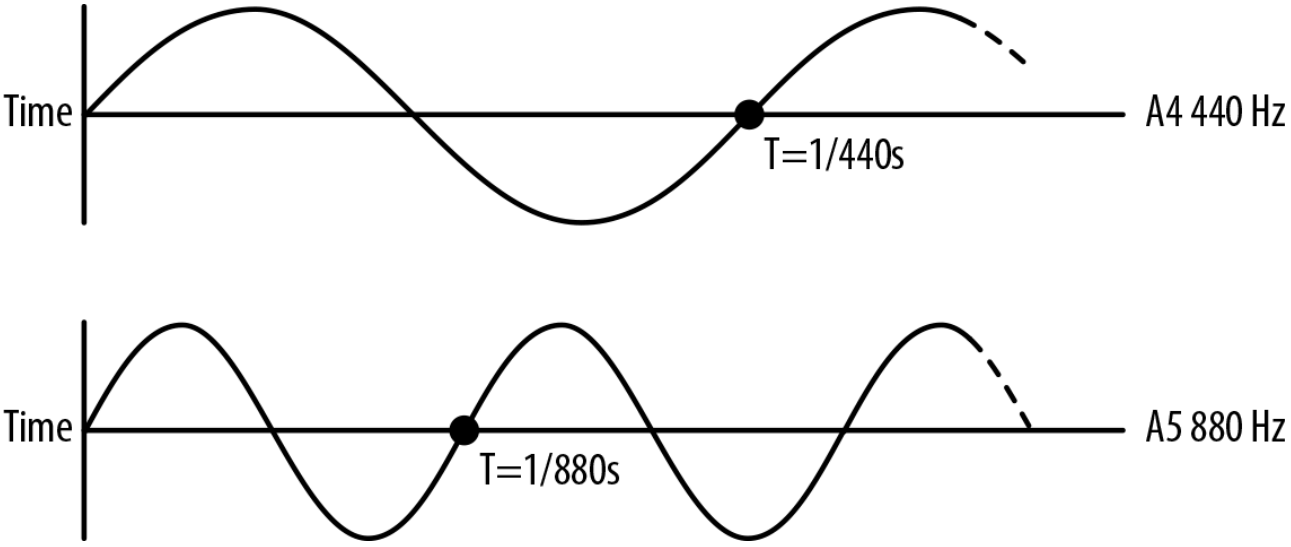

Рисунок 4-1. Графики идеальных нот A4 и A5

Октавы разделены на 12 полутонов. Каждая соседняя пара полутонов имеет одинаковое соотношение частот (по крайней мере, в равномерно темперированном строе). Другими словами, соотношение частот `A4` к `A#4` идентично соотношению частот `A#4` к `B4`.

Рисунок 4-1 показывает то, как можно вывести соотношение между каждым последующим полутоном, учитывая, что:

1. Для транспонирования ноты на октаву выше, мы удваиваем частоту ноты.
2. Каждая октава делится на 12 полутонов, которые при равномерно темперированном строе имеют одинаковые соотношения частот.

Пусть `f_0` — некоторая частота, а `f_1` — эта же частота, но на одну октаву выше. Мы уже знаем отношение между ними:

```
f_1 = 2 * f_0
```

Далее, пусть `k` будет фиксированным множителем между двумя любыми соседними полутонами. Так как всего в октаве 12 полутонов, мы можем вывести следующее:

```
f_1 = f_0 * k * k * k * ... * k (12x) = f_0 * k^12
```

Если решить систему этих двух уравнений, получим:

```
2 * f_0 = f_0 * k^12
```

Найдём `k`:

```
k = 2^(1/12) ~= 1.0595...
```

К счастью, все эти вычисления смещений, связанных с полутонами, обычно не нужно делать вручную, так как многие аудиосреды (включая Web Audio API) включают в себя понятие *расстройки* (detune), которое переводит шкалу частот в линейный вид. Расстройка измеряется в центах: каждая октава содержит 1200 центов, а каждый полутон — 100 центов. Указав расстройку в 1200 центов, вы поднимаетесь на октаву выше. Соответственно, расстройка в -1200 центов понижает частоту на октаву.

Высота звука и `playbackRate`

Web Audio API предоставляет параметр `playbackRate` для каждого узла `AudioSourceNode`. Это значение можно задать для изменения высоты звука из любого аудиобуфера. Обратите внимание, что в этом случае будут изменены как высота звука, так и длительность семпла. Существуют специальные методы, которые пытаются поменять частоту, не затрагивая длительность семпла, однако довольно сложно сделать это универсальным способом, не внося в микс помех, скретчей и других нежелательных артефактов.

Как мы ранее обсуждали в разделе Основы высоты звука в музыке, для того, чтобы вычислить частоты последовательных полутонов, нужно умножить частоту на соотношение `2^(1/12)`. Это очень полезно, если вы разрабатываете музыкальный инструмент или используете изменение высоты звука для рандомизации в игровой среде. Следующий код воспроизводит тон с заданным смещением частоты в полутонах:

```
function playNote(semitones) {
  // Предположим, что новый источник звука был создан из буфера
  var semitoneRatio = Math.pow(2, 1/12);

  source.playbackRate.value = Math.pow(semitoneRatio, semitones);
  source.start(0);
}
```

Ранее мы обсуждали, что наши уши воспринимают высоту звука экспоненциально. Обрабатывать высоту звука как экспоненциальную величину может быть не очень удобно, поскольку в вычислениях мы постоянно должны использовать такие неудобные значения, как корень двенадцатой степени из двух. Вместо этого мы можем использовать параметр `detune`, чтобы задать смещение частот в центах. Таким образом можно переписать этот код, используя параметр `detune`:

```
function playNote(semitones) {
  // Предположим, что новый источник звука был создан из буфера
  source.detune.value = semitones * 100;
  source.start(0);
}
```

Если сместить высоту звука на слишком большое количество полутонов (например, вызвав `playNote(24)`), вы начнёте слышать искажения. Поэтому цифровые пианино включают в себя несколько семплов для каждого инструмента. Хорошие цифровые пианино избегают изменения высоты звука совсем, и включают в себя отдельные семплы для каждой клавиши. Отличные цифровые пианино включают в себя несколько семплов для каждой клавиши, которые воспроизводятся в зависимости от силы нажатия на клавишу.

Множество одновременных звуков с вариациями

Ключевая особенность звуковых эффектов в играх — это то, что их может быть сразу много одновременно. Представьте, что вы в центре перестрелки между множеством игроков, стреляющих друг в друга из пулемётов. Каждый пулемёт стреляет много раз в секунду, вызывая десятки звуковых эффектов, которые

должны быть проиграны в один момент. Воспроизведение звуков из множества источников в конкретное и одно и то же время — это то, где Web Audio API по-настоящему отлично себя показывает.

Итак, если все пулемёты в вашей игре звучат абсолютно одинаково, это будет довольно скучно. Конечно же звуки могут варьироваться относительно дистанции до цели и позиции слушателя к другим игрокам (см. Пространственный звук), но даже этого может быть недостаточно. К счастью, Web Audio API предоставляет способ легко модифицировать данный пример как минимум двумя простыми способами:

1. Добавив небольшой сдвиг по времени между выстрелами.
2. Изменив высоту звука для лучшей имитации случайности реального мира.

Используя наши знания о тайминге и высоте звука, мы можем легко реализовать эти два эффекта:

```
function shootRound(numberOfRounds, timeBetweenRounds) {
  var time = context.currentTime;

  // Создаём несколько источников аудио из одного и того же буфера
  // и проигрываем в быстрой последовательности
  for (var i = 0; i < numberOfRounds; i++) {
    var source = this.makeSource(bulletBuffer);

    source.playbackRate.value = 1 + Math.random() * RANDOM_PLAYBACK;
    source.start(time + i * timeBetweenRounds + Math.random() *
RANDOM_VOLUME);
  }
}
```

Web Audio API автоматически объединяет несколько звуков, воспроизводимых одновременно, по сути просто складывая формы волн. Это может привести к таким проблемам, как клиппинг, о котором мы говорили ранее в разделе Клиппинг и измерения уровня сигнала.

Этот пример добавляет некоторую вариативность к узлам `AudioBuffer`, которые мы загрузили из звуковых файлов. В некоторых случаях желательно использовать полностью синтезированные звуки и не пользоваться буферами в

принципе (см. Процедурно сгенерированный звук).

Понятие частотного спектра

До этого момента в наших теоретических упражнениях мы рассматривали звук как функцию давления, меняющегося со временем. Однако есть и другой полезный способ анализа звука — отобразить амплитуду и посмотреть, как она меняется в зависимости от частоты. В итоге мы получим графики, в которых область определения функции (ось X) выражается в единицах частоты (Hz). Такие графики описывают звук в *частотном спектре*.

Связь между графиками во временном и частотном спектрах основана на идее *разложения Фурье*. Как мы уже видели, звуковые волны часто имеют циклическую природу. Математически периодические звуковые волны можно рассматривать как сумму нескольких простых синусоид разных частот и амплитуд. Чем больше таких синусоид мы складываем, тем точнее получается приближение исходной функции. Мы можем взять сигнал и выделить его составляющие синусоиды, применив преобразование Фурье. Для такого разложения существует множество алгоритмов, самый известный из них – *быстрое преобразование Фурье* (FFT). К счастью, в Web Audio API уже встроена реализация этого алгоритма. Далее мы рассмотрим, как это устроено (см. Частотный анализ).

В общем случае мы можем взять звуковую волну, вычислить составляющую её синусоиду и нанести в виде точек (частоту, амплитуду) на график, чтобы получить график частотного спектра. На Рисунке 4-2 показана чистая нота ля (`A`) на частоте `440 Hz` (`A4`).

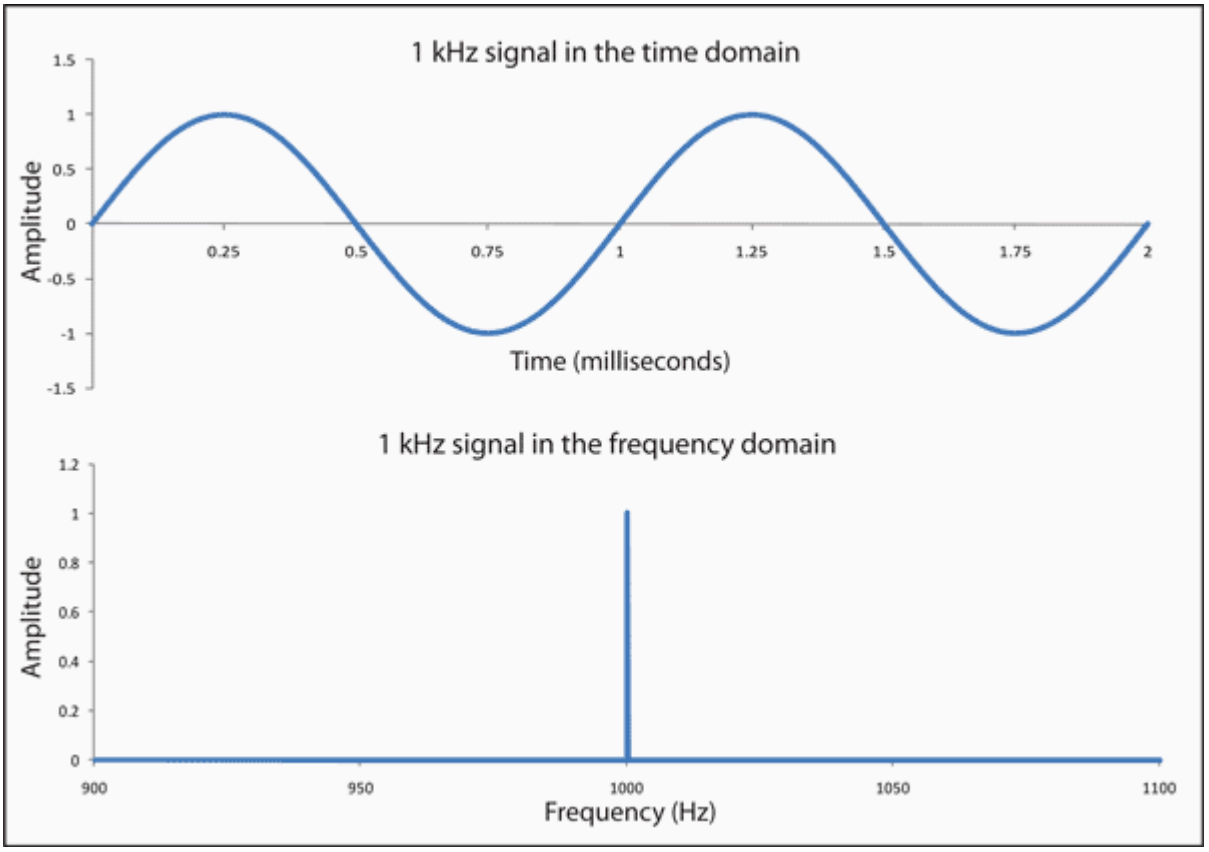

Рисунок 4-2. Идеальная синусоидальная волна 1 кГц, представленная в частотном и временном спектрах

Анализ графика частотного спектра может дать лучшее представление о качествах звука, таких как высота тона, содержание шума в спектре и многих других. На основе частотного спектра можно построить продвинутые алгоритмы, такие как определение высоты тона. У звука, который произведён реальными музыкальными инструментами есть обертоны, так что нота `A4`, сыгранная на пианино, будет иметь график частотного спектра, который сильно отличается от графика той же ноты `A4`, но сыгранной на трубе. Независимо от сложности звуков, здесь будут также применимы идеи разложения Фурье. На Рисунке 4-3 показан более сложный фрагмент звука как во временном, так и в частотном спектре.

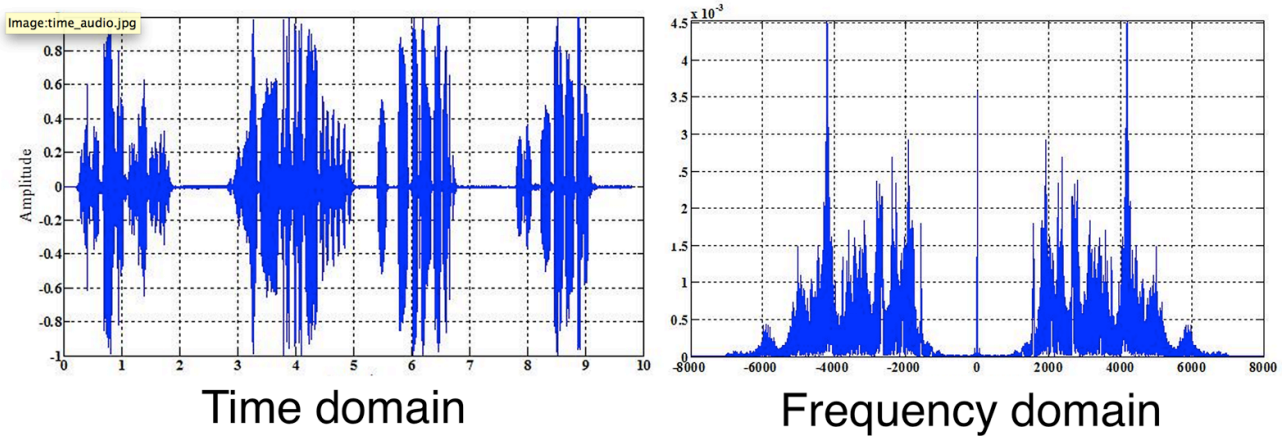

Рисунок 4-3. Сложный звуковой сигнал, представленный в частотном и временном спектрах

Эти графики ведут себя довольно по-разному с течением времени. Если бы вы очень медленно проигрывали звук, который изображён на Рисунке 4-3 и наблюдали бы изменение обоих графиков, вы бы заметили, что график временного спектра (слева) продвигается слева направо. График частотного спектра (справа) — это результат анализа звуковой волны в определённый момент времени, поэтому он мог бы меняться намного быстрее и куда менее предсказуемо.

Важно отметить, что частотный анализ по-прежнему полезен, даже когда исследуемый звук не воспринимается как имеющий определённую высоту. Звуки ветра, ударов и выстрелов имеют характерный вид на графике частотного спектра. Например, белый шум имеет плоский спектр в частотной области, так как все частоты в нём представлены одинаково.

Прямой синтез звука на основе осциллятора

Ранее мы обсуждали, что цифровой звук в Web Audio API представлен в виде массива чисел в узлах `AudioBuffer`. В большинстве случаев, буфер создаётся при загрузке звукового файла или на лету из какого-то звукового потока. Но иногда нам может понадобиться синтезировать собственные звуки. Мы можем сделать это, программно создав аудиобуферы через JavaScript, просто вычисляя математическую функцию через равные промежутки времени и записывая значения в массив. Такой подход позволяет вручную изменять амплитуду и частоту синусоиды или даже объединять несколько синусоид, чтобы создавать произвольные звуки (вспоминаем преобразования Фурье из раздела Понятие частотного спектра).

Хоть это и реализуемо, но разработка такого функционала на JavaScript будет сложной и неэффективной. Вместо этого Web Audio API предоставляет примитивы, которые позволяют делать то же самое при помощи узлов осцилляторов `OscillatorNode`. Эти узлы имеют настраиваемые параметры частоты и расстройки (см. Основы высоты звука в музыке). Также можно задать тип генерируемой волны. Встроенные типы включают синусоидальную, треугольную, пилообразную и прямоугольную волны, как показано на Рисунке 4-4.

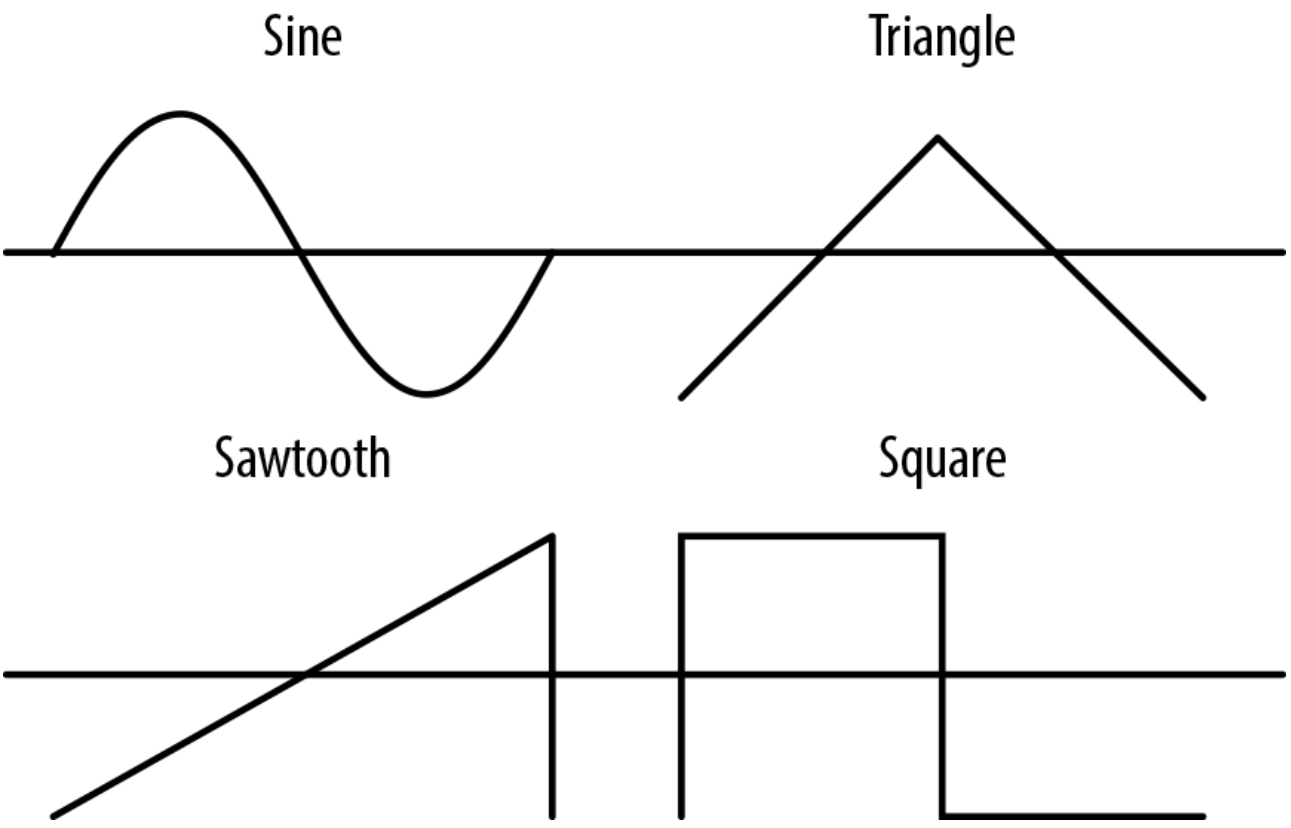

Рисунок 4-4. Базовые типы волн, которые может генерировать осциллятор

Осцилляторы могут быть легко использованы в аудиографах вместо узлов `AudioBufferSourceNode`. Посмотрим на пример ниже:

```
function play(semitone) {
  // Создадим необходимые узлы
  var oscillator = context.createOscillator();
  oscillator.connect(context.destination);

  // Проиграем синусоидальную волну на частоте в 440 Hz (A4)
  oscillator.frequency.value = 440;
  oscillator.detune.value = semitone * 100;

  // Обратите внимание, что эта константа будет заменена на "sine"
  oscillator.type = oscillator.SINE;
  oscillator.start(0);
}
```

Помимо этих базовых типов волн, вы можете создать собственную волну для осциллятора, используя таблицы гармоник. Это позволит эффективно формировать волновые формы, значительно более сложные, чем рассмотренные ранее. Эта тема имеет большое значение для задач музыкального синтеза, однако выходит за рамки данной книги.

Анализ и визуализация

До этого мы говорили только об обработке и синтезе аудио, но это только половина всех возможностей, которые предоставляет Web Audio API. Другая половина — анализ аудио — позволяет понять, какими свойствами обладает проигрываемый звук. Классический пример использования такого функционала — это визуализация звука, но существует также очень много применений анализа, которые выходят за границы этой книги, включая в себя определение высоты звука, темпа и распознавание речи.

Это очень важная тема для разработчиков игр и интерактивных приложений по двум причинам. Во-первых, хороший визуальный анализатор может использоваться как отладочный инструмент (в дополнение к нашим ушам и хорошо настроенной измерительной аппаратуре) для тонкой настройки звука. Во-вторых, визуализация критична для игр и музыкальных приложений, таких как *Guitar Hero* или музыкального ПО типа *GarageBand*.

Частотный анализ

Основное средство для анализа звука в Web Audio API — это узлы `AnalyserNode`. Эти узлы не меняют сам звук и могут быть добавлены в любом месте аудиографа. Будучи подключёнными к аудиографу, они могут использоваться для анализа звуковой волны как во временном, так и в частотном спектре.

Полученные результаты основаны на *FFT*-анализе буфера определённого размера. У нас есть несколько способов повлиять на выходные данные узла `AnalyserNode`:

`fftSize`

Этот параметр задаёт размер буфера, который используется для анализа. Он должен быть равен степени двойки. Более высокие значения этого параметра позволят проводить более детальный анализ, но могут привести к некоторому снижению производительности.

`frequencyBinCount`

Это значение только для чтения, задаётся автоматически как `fftSize/2`.

`smoothingTimeConstant`

> *Это значение должно находиться в диапазоне от `0` до `1`. При значении `1` используется большое окно скользящего среднего, и результаты получаются максимально сглаженными. При значении `0` скользящее среднее не применяется, и результаты изменяются быстро и резко.*

Вот как выглядит базовая настройка узла анализатора, который можно подключить к интересующей нас части аудиографа:

```
// Предположим, что узел A соединён с узлом B
var analyser = context.createAnalyser();

A.connect(analyser);
analyser.connect(B);
```

Далее мы можем получить массив значений частотного или временного спектра:

```
var freqDomain = new Float32Array(analyser.frequencyBinCount);

analyser.getFloatFrequencyData(freqDomain);
```

`freqDomain` в этом примере — это массив 32-битных чисел с плавающей точкой, относящихся к частотному спектру. Эти числа нормализованы и находятся в диапазоне от нуля до единицы. Индексы выходных данных могут быть линейны сопоставлены между нулём и *частотой Найквиста*, которая определяется как половина частоты дискретизации (значение доступно в Web Audio API через `context.sampleRate`). Следующий фрагмент кода сопоставляет частоту с подходящим сегментом в массиве частот:

```
function getFrequencyValue(frequency) {
  var nyquist = context.sampleRate / 2;
  var index = Math.round(frequency / nyquist * freqDomain.length);

  return freqDomain[index];
}
```

К примеру, если мы попытаемся проанализировать синусоидальную волну частотой `1000 Hz`, мы можем ожидать, что вызов `getFrequencyValue(1000)` вернёт пиковое значение на графике, как показано на Рисунке 5-1.

Частотный спектр также доступен в виде 8-битных беззнаковых целых чисел через вызов `getByteFrequencyData()`. Значения этих чисел масштабируются таким образом, чтобы соответствовать диапазону между значениями `minDecibels` и `maxDecibels` (в `dBFS`) на узле анализатора, поэтому эти параметры можно настраивать для масштабирования выходных данных по желанию.

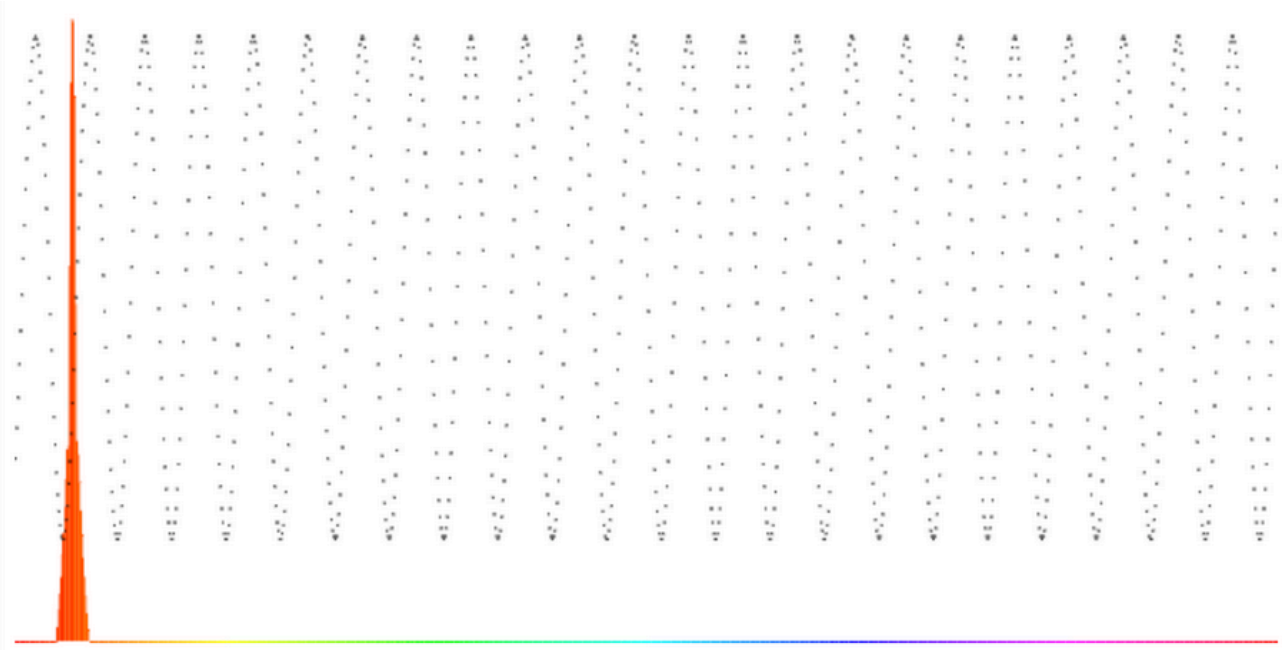

Рисунок 5-1. Визуализация синусоидальной волны частотой 1000 Hz (полный диапазон от 0 до 22,050 Hz)

Анимация с `requestAnimationFrame`

Если мы хотим построить визуализацию формы звуковой волны, мы должны периодически опрашивать анализатор, обрабатывать результаты и отображать их на странице. Мы можем сделать это, установив JavaScript-таймеры через `setInterval` или `setTimeout`, но есть более подходящий способ: использовать `requestAnimationFrame`. Он позволяет браузеру встроить вашу кастомную функцию отрисовки внутрь цикла отрисовки браузера, который отлично оптимизирован для подобного использования. Вместо того, чтобы заставлять эту функцию отрисовки запускаться в конкретные промежутки

времени и конкурировать с другими браузерными событиями, мы можем просто запланировать события отрисовки в очереди событий и браузер выполнит их автоматически, как только сможет.

Так как `requestAnimationFrame` всё ещё экспериментальный API, мы должны использовать версии с префиксами для каждого браузера, и предоставить фолбэк для браузеров, которые его не поддерживают в виде `setTimeout`:

```
window.requestAnimationFrame = (function() {
  return window.requestAnimationFrame  ||
    window.webkitRequestAnimationFrame ||
    window.mozRequestAnimationFrame    ||
    window.oRequestAnimationFrame      ||
    window.msRequestAnimationFrame     ||
    function(callback){
      window.setTimeout(callback, 1000 / 60);
    };
  })();
```

Теперь мы можем использовать `requestAnimationFrame` для того, чтобы опрашивать анализатор для получения детальной информации о состоянии аудиопотока.

Визуализация звука

Объединив всё вместе, мы можем настроить цикл рендера, который, как и раньше, будет опрашивать анализатор и отображать его текущий частотный анализ в массив `freqDomain`:

```
var freqDomain = new Uint8Array(analyser.frequencyBinCount);
analyser.getByteFrequencyData(freqDomain);

for (var i = 0; i < analyser.frequencyBinCount; i++) {
  var value = freqDomain[i];
  var percent = value / 256;
  var height = HEIGHT * percent;
  var offset = HEIGHT - height - 1;
  var barWidth = WIDTH/analyser.frequencyBinCount;
  var hue = i/analyser.frequencyBinCount * 360;

  drawContext.fillStyle = 'hsl(' + hue + ', 100%, 50%)';
  drawContext.fillRect(i * barWidth, offset, barWidth, height);
}
```

Мы можем сделать похожую вещь и для временного спектра:

```
var timeDomain = new Uint8Array(analyser.frequencyBinCount);
analyser.getByteTimeDomainData(timeDomain);

for (var i = 0; i < analyser.frequencyBinCount; i++) {
  var value = timeDomain[i];
  var percent = value / 256;
  var height = HEIGHT * percent;
  var offset = HEIGHT - height - 1;
  var barWidth = WIDTH/analyser.frequencyBinCount;

  drawContext.fillStyle = 'black';
  drawContext.fillRect(i * barWidth, offset, 1, 1);
}
```

Этот код нарисует график временного спектра поверх цветного графика частотного спектра, используя HTML5 Canvas API. Результат отрисовки канваса изображен на Рисунке 5-2 и будет меняться со временем.

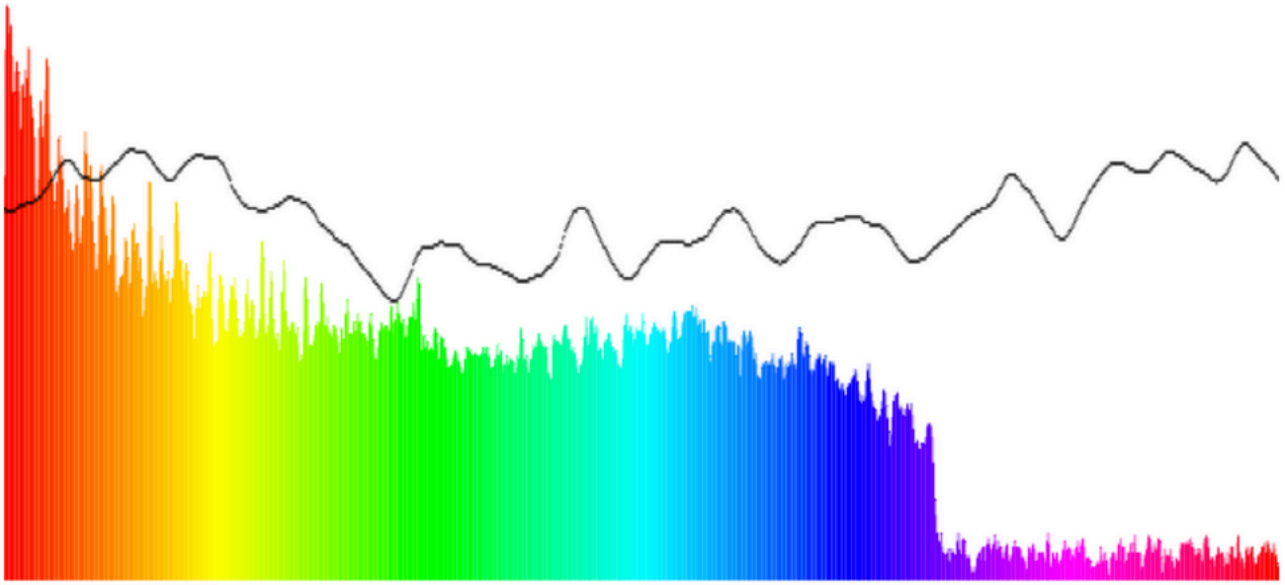

Рисунок 5-2. Изображение визуализатора в действии

Наш подход к визуализации упускает множество важных данных. Для визуализации музыки этого достаточно, однако, если мы хотим провести комплексный анализ всего аудиобуфера, следует обратиться к другим методам.

Продвинутые темы

Этот раздел покрывает очень важные темы, однако они чуть более сложные, чем в остальной книге. Мы погрузимся в вопросы добавления эффектов к звукам, генерации синтезированных звуковых эффектов без использования аудиобуферов, симуляции эффектов различных акустических сред, а также пространственного звука в 3D-пространстве.

Биквадратные фильтры

Фильтр может усиливать или ослаблять определённые части частотного спектра звука. Визуально это можно показать в виде графика в частотной области, который называется *графиком частотной характеристики* (см. Рисунок 6-1). Для каждой частоты действует правило: чем выше значение графика, тем сильнее подчёркивается эта часть диапазона. Если график идёт на спад, это значит, что больший акцент делается на низких частотах и меньший — на высоких.

Фильтры Web Audio можно настроить с помощью трёх параметров: *усиления* (gain), *частоты* (frequency) и *коэффициента качества* (quality factor, также известный как *Q*). Каждый из этих параметров по-своему влияет на график частотной характеристики.

Существует множество фильтров, которые используются для создания разных эффектов:

Низкочастотный фильтр (Low-pass filter)
: *Делает звук более приглушённым.*

Высокочастотный фильтр (High-pass filter)
: *Делает звук более «тонким», с акцентом на высоких частотах.*

Полосовой фильтр (Band-pass filter)
: *Отсекает низкие и высокие частоты (например, «телефонный» эффект).*

Полочный фильтр нижних частот (Low-shelf filter)
: *Управляет количеством баса в звуке (как регулятор Bass на стереосистеме).*

Полочный фильтр верхних частот (High-shelf filter)

Управляет количеством высоких частот (как регулятор Treble на стереосистеме).

Пиковый фильтр (Peaking filter)

Управляет количеством средних частот (как регулятор Mid на стереосистеме).

Режекторный фильтр (Notch filter)

Удаляет нежелательные звуки в узком диапазоне частот.

Всепропускающий фильтр (All-pass filter)

Создаёт эффекты типа phaser.

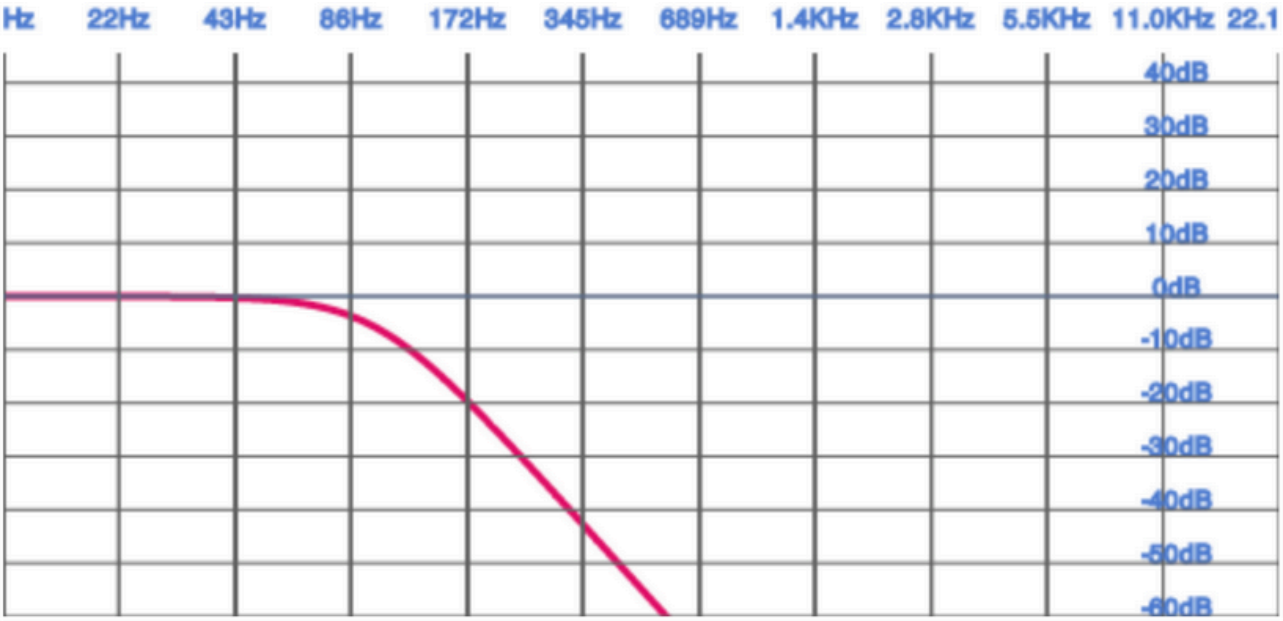

Рисунок 6-1. График частотной характеристики низкочастотного фильтра

Все эти биквадратные фильтры основаны на общей математической модели, и их графики можно построить так же, как у низкочастотного фильтра на Рисунке 6-1. Более подробную информацию о таких фильтрах можно найти в книгах с более серьёзной математической базой, например, *Real Sound Synthesis for Interactive Applications* Перри Р. Кука (A K Peters, 2002). Я настоятельно рекомендую её прочесть, если вы интересуетесь фундаментальными основами аудио.

Добавление эффектов с помощью фильтров

Используя Web Audio API, мы можем применять фильтры с помощью узлов `BiquadFilterNode`. Этот тип аудиоузла очень часто используется для построения эквалайзеров и различных звуковых эффектов. Давайте настроим

простой фильтр нижних частот, чтобы убрать низкочастотный шум из звукового семпла:

```
// Создаём фильтр
var filter = context.createBiquadFilter();

// Примечание: спецификация Web Audio переходит от констант к строкам.
// filter.type = 'lowpass';
filter.type = filter.LOWPASS;
filter.frequency.value = 100;

// Подключаем источник к фильтру, и фильтр к аудиовыходу.
source.connect(filter);
filter.connect(destination);
```

Узлы `BiquadFilterNode` поддерживают все часто используемые фильтры второго порядка. Мы можем настроить эти узлы с помощью параметров, о которых мы говорили ранее, а также визуализировать их частотные характеристики, используя метод `getFrequencyResponse()`. Метод принимает массив частот и возвращает массив значений амплитудных откликов, соответствующих каждой переданной частоте.

Крис Уилсон и Крис Роджерс подготовили отличный пример визуализатора (см. Рисунок 6-2), который показывает частотные характеристики всех типов фильтров, доступных в Web Audio API.

Рисунок 6-2. График частотной характеристики низкочастотного фильтра с параметрами

Процедурно сгенерированный звук

До этого мы исходили из того, что источники звука (например, в игре) статичны. Дизайнер аудио создаёт множество аудиоресурсов и передаёт их далее разработчикам. Разработчики, в свою очередь, проигрывают их с использованием некоторых настроек, в зависимости от условий локации (например, настройки атмосферы помещения и относительного расположения источников звука и слушателей). Этот подход имеет несколько недостатков:

1. Звуковые файлы будут слишком большими. Это особенно плохо для веба, в котором звуки загружаются не с жёсткого диска, а из интернета (по крайней мере, в первый раз), что примерно на порядок медленнее.
2. Даже при наличии множества разных ресурсов и изменений в них, их вариативность ограничена.
3. Вам необходимо найти ресурсы, просматривая библиотеки звуковых эффектов, а затем, возможно, беспокоиться о роялти их авторам. К тому же, скорее всего, тот или иной звуковой эффект будет уже использоваться в других приложениях, и у пользователей могут возникнуть непредвиденные ассоциации.

Мы можем использовать Web Audio API для того, чтобы полностью сгенерировать звуки процедурно. Для примера, давайте сэмулируем звук стрельбы. Мы начнём с буфера, содержащего белый шум, для этого сгенерируем шум с помощью узла `ScriptProcessorNode` *(прим. переводчика: этот узел устарел и заменён на* `AudioWorklet` *в новых версиях Web Audio API)*:

```
function WhiteNoiseScript() {
  this.node = context.createScriptProcessor(1024, 1, 2);
  this.node.onaudioprocess = this.process;
}

WhiteNoiseScript.prototype.process = function(e) {
  var L = e.outputBuffer.getChannelData(0);
  var R = e.outputBuffer.getChannelData(1);

  for (var i = 0; i < L.length; i++) {
    L[i] = ((Math.random() * 2) - 1);
    R[i] = L[i];
  }
};
```

Для получения дополнительной информации о `ScriptProcessorNode` смотрите раздел Обработка аудио с помощью JavaScript.

Этот код не слишком производителен, потому что JavaScript вынужден динамически создавать поток белого шума. Чтобы увеличить производительность, мы можем программно сгенерировать моноканальный аудиобуфер с белым шумом, как в следующем примере кода:

```
function WhiteNoiseGenerated(callback) {
  // Генерируем 5-секундный буфер белого шума
  var lengthInSamples = 5 * context.sampleRate;
  var buffer = context.createBuffer(1, lengthInSamples,
context.sampleRate);
  var data = buffer.getChannelData(0);

  for (var i = 0; i < lengthInSamples; i++) {
    data[i] = ((Math.random() * 2) - 1);
  }

  // Создаём источник из буфера
  this.node = context.createBufferSource();
  this.node.buffer = buffer;
  this.node.loop = true;
  this.node.start(0);
}
```

Далее мы можем смоделировать различные фазы выстрела с помощью *огибающей* (envelope): *атаку* (attack), *спад* (decay) и *затухание* (release):

```
function Envelope() {
  this.node = context.createGain()
  this.node.gain.value = 0;
}

Envelope.prototype.addEventToQueue = function() {
  this.node.gain.linearRampToValueAtTime(0, context.currentTime);
  this.node.gain.linearRampToValueAtTime(1, context.currentTime +
0.001);
  this.node.gain.linearRampToValueAtTime(0.3, context.currentTime +
0.101);
  this.node.gain.linearRampToValueAtTime(0, context.currentTime +
0.500);
};
```

Наконец, мы можем подключить выходы голоса к фильтру, чтобы имитировать эффект расстояния:

```
this.voices = [];
this.voiceIndex = 0;

var noise = new WhiteNoise();

var filter = context.createBiquadFilter();
filter.type = 0;
filter.Q.value = 1;
filter.frequency.value = 800;

// Инициализация нескольких голосов
for (var i = 0; i < VOICE_COUNT; i++) {
  var voice = new Envelope();

  noise.connect(voice.node);
  voice.connect(filter);

  this.voices.push(voice);
}

var gainMaster = context.createGainNode();
gainMaster.gain.value = 5;
filter.connect(gainMaster);

gainMaster.connect(context.destination);
```

Этот пример заимствован со страницы BBC, посвящённой звуковым эффектам выстрелов, с небольшими изменениями, включая портирование на JavaScript.

Как видите, этот подход очень мощный, но довольно быстро становится сложным и выходит за рамки этой книги. Для более подробной информации о процедурной генерации звука рекомендую ознакомиться с учебными материалами и книгой Энди Фарнелла Practical Synthetic Sound Design.

Эффекты помещения

Прежде чем звук достигнет наших ушей, он отразится от стен, зданий, мебели, ковров и других объектов. Каждое такое столкновение меняет свойства звука. Например, хлопок в ладоши на улице звучит совсем не так, как хлопок внутри большого собора, где реверберация может быть слышна несколько секунд. Игры с

высоким уровнем проработки стремятся имитировать такие эффекты. Создание отдельного набора семплов для каждой акустической среды часто оказывается чрезмерно затратным, так как требует больших усилий от звукового дизайнера, множества файлов ресурсов и, как следствие, значительно увеличивает объём игровых данных.

В Web Audio API есть инструмент для симуляции различных акустических сред — `ConvolverNode`. С его помощью можно создавать такие эффекты, как *хорус* (chorus), *реверберацию* и звук, напоминающий телефонную речь.

Идея создания эффектов помещения заключается в том, чтобы воспроизвести опорный звук в комнате, записать его, а затем (образно говоря) вычесть из записанного оригинала. Результатом такого процесса является импульсная характеристика, которая фиксирует влияние помещения на звук. Эти импульсные характеристики тщательно записываются в специальных студийных условиях, и для того, чтобы сделать это самостоятельно, требуется серьёзная подготовка. К счастью, существуют сайты, где можно найти множество уже готовых файлов импульсных характеристик (хранящихся в виде аудиофайлов) для удобного использования.

Web Audio API предоставляет простой способ применить такие импульсные характеристики к вашим звукам с помощью `ConvolverNode`. Этот узел принимает буфер импульсной характеристики, представляющий собой обычный `AudioBuffer`, в который загружен соответствующий файл. По сути, конволвер — это очень сложный фильтр (подобно `BiquadFilterNode`), но вместо выбора из набора готовых типов эффектов его можно настроить на произвольную частотную характеристику:

```
var impulseResponseBuffer = null;

function loadImpulseResponse() {
  loadBuffer('impulse.wav', function(buffer) {
    impulseResponseBuffer = buffer;
  });
}

function play() {
  // Создаём узел-источник для семпла
  var source = context.createBufferSource();
  source.buffer = this.buffer;

  // Создаём узел Convolver для импульсной характеристики
  var convolver = context.createConvolver();

  // Устанавливаем буфер импульсной характеристики
  convolver.buffer = impulseResponseBuffer;

  // Соединяем граф
  source.connect(convolver);
  convolver.connect(context.destination);
}
```

Узел `ConvolverNode` «смешивает» входной звук с его импульсной характеристикой, выполняя *свёртку* — вычислительно сложную математическую операцию. В результате получается звук, который звучит так, словно он был записан в том помещении, где была сделана импульсная характеристика. На практике часто имеет смысл смешивать исходный сигнал (называемый *dry mix*) с обработанным сигналом (*wet mix*), используя *кроссфейд с равной мощностью* (equal-power crossfade), чтобы контролировать, какую долю эффекта вы хотите применить.

Также можно синтезировать такие импульсные характеристики, но эта тема выходит за рамки данной книги.

Пространственный звук

Игры часто существуют в мире, где объекты имеют позиции в пространстве — будь то в 2D или в 3D. В таких случаях пространственный звук может значительно повысить эффект погружения. К счастью, в Web Audio API есть встроенные возможности позиционирования звука (пока только в стерео), и использовать их достаточно просто.

Экспериментируя с пространственным звуком, убедитесь, что вы слушаете его через стереоколонки (лучше — в наушниках). Так вы получите более точное представление о том, как левый и правый каналы изменяются в зависимости от выбранного вами метода пространственной обработки.

Модель Web Audio API включает три уровня сложности (во многом заимствованные из OpenAL):

1. Позиция и ориентация источников и слушателей.
2. Параметры, связанные с направленными аудиоконусами источника.
3. Относительные скорости источников и слушателей.

В Web Audio API есть единственный слушатель (узел `AudioListener`), прикреплённый к контексту, который можно настраивать в пространстве, задавая его позицию и ориентацию. Каждый источник можно пропустить через панорамирующий узел `AudioPannerNode`, который выполняет пространственную обработку входного аудио. На основе относительного положения источников и слушателя Web Audio API рассчитывает необходимые изменения усиления.

Есть несколько вещей, которые важно знать об используемых в API допущениях. Во-первых, по умолчанию слушатель находится в начале координат (0, 0, 0). Координаты позиционирования в API не имеют единиц измерения, поэтому на практике нужно вручную подбирать множители так, чтобы звук воспринимался, как вы хотите. Во-вторых, ориентация задаётся векторами направления (длиной 1). И, наконец, в этой системе координат положительная ось Y направлена вверх, что противоположно большинству систем компьютерной графики.

Учитывая это, вот пример того, как можно изменить позицию узла-источника, который пространственно обрабатывается в 2D с помощью панорамирующего узла `PannerNode`:

```
// Размещаем слушателя в начале координат (по умолчанию, просто для
ясности)
context.listener.setPosition(0, 0, 0);

// Размещаем узел панорамирования
// Предполагаем, что X и Y заданы в координатах экрана,
// а слушатель находится в центре экрана
var panner = context.createPanner();
var centerX = WIDTH/2;
var centerY = HEIGHT/2;

var x = (X - centerX)  / WIDTH;
// Координата Y инвертируется, чтобы совпадать с системой координат
canvas
var y = (Y - centerY) / HEIGHT;
// Координату Z размещаем немного позади слушателя
var z = -0.5;

// Подбираем множитель по необходимости
var scaleFactor = 2;
panner.setPosition(x * scaleFactor, y * scaleFactor, z);
// Преобразуем угол в единичный вектор
panner.setOrientation(Math.cos(angle), -Math.sin(angle), 1);

// Подключаем узел, который нужно пространственно
// обрабатывать, к панорамирующему узлу
source.connect(panner);
```

Помимо учёта относительных позиций и ориентаций, у каждого источника есть настраиваемый аудиоконус, как показано на Рисунке 6-3.

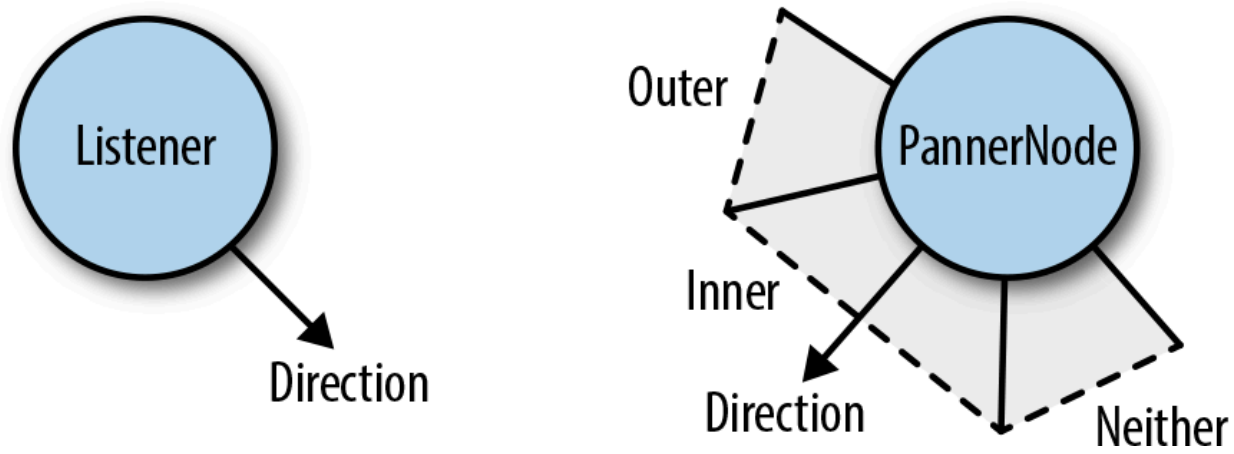

Рисунок 6-3. Диаграмма панорамирующих узлов и слушателя в 2D пространстве

После того как вы задали внутренний и внешний конус, пространство делится на три зоны, как на Рисунке 6-3:

1. Внутренний конус.
2. Внешний конус.
3. Зона вне обоих конусов.

Для каждой из этих зон можно задать коэффициент усиления, служащий дополнительным параметром для настройки позиционной модели. Например, чтобы имитировать направленный звук, можно использовать следующую конфигурацию:

```
panner.coneInnerAngle = 5;
panner.coneOuterAngle = 10;
panner.coneGain = 0.5;
panner.coneOuterGain = 0.2;
```

Рассеянный звук может иметь совершенно иной набор параметров. У всенаправленного источника внутренний конус составляет 360 градусов, и его ориентация не оказывает влияния на пространственную обработку:

```
panner.coneInnerAngle = 180;
panner.coneGain = 0.5;
```

Помимо позиции, ориентации и аудиоконусов, для источников и слушателя можно задать скорость. Этот параметр важен для имитации изменения высоты тона в результате доплеровского эффекта.

Обработка аудио с помощью JavaScript

Web Audio API стремится предоставить достаточный набор примитивов (в основном через аудиоузлы), чтобы можно было выполнять большинство стандартных задач работы со звуком. Эти модули написаны на C++ и работают значительно быстрее, чем аналогичный код на JavaScript.

Однако Web Audio API также предоставляет `ScriptProcessorNode` *(прим. переводчика: этот узел устарел и заменён на* `AudioWorklet` *в новых версиях Web Audio API)*, позволяющий веб-разработчикам напрямую синтезировать и обрабатывать звук на JavaScript. Например, с его помощью можно прототипировать собственные DSP-эффекты или наглядно демонстрировать концепции в образовательных приложениях.

Для начала создайте `ScriptProcessorNode`. Этот узел обрабатывает звук кусками, размер которых задаётся параметром `bufferSize`. Размер буфера должен быть равен степени двойки. Лучше использовать буфер побольше, так как это даст больше запаса прочности для предотвращения сбоев, если основной поток занят другими задачами — такими, как переразметка страницы, сборка мусора или вызовы функций JavaScript:

```
// Создаём ScriptProcessorNode
var processor = context.createScriptProcessor(2048);

// Назначаем функцию onProcess для вызова при обработке каждого буфера
processor.onaudioprocess = onProcess;

// Подключаем существующий источник к ScriptProcessorNode
source.connect(processor);
```

Когда аудиоданные начнут поступать в функцию процессора, вы можете начать анализировать поток, исследуя входной буфер, или напрямую изменять выход, модифицируя выходной буфер. Например, мы можем легко поменять местами

левый и правый каналы, реализовав следующий скриптовый процессор:

```
function onProcess(e) {
  var leftIn = e.inputBuffer.getChannelData(0);
  var rightIn = e.inputBuffer.getChannelData(1);
  var leftOut = e.outputBuffer.getChannelData(0);
  var rightOut = e.outputBuffer.getChannelData(1);

  for (var i = 0; i < leftIn.length; i++) {
    // Меняем местами левый и правый каналы
    leftOut[i] = rightIn[i];
    rightOut[i] = leftIn[i];
  }
}
```

Обратите внимание, что в продакшене так менять каналы не стоит, так как использование узлов `ChannelSplitterNode` вместе с `ChannelMergerNode` гораздо эффективнее. Ещё один пример: мы можем добавить к сигналу случайный шум. Для этого достаточно прибавить к нему случайное смещение. Если сделать сигнал полностью случайным, мы получим белый шум, который может быть очень полезен в разных ситуациях (см. Процедурно сгенерированный звук):

```
function onProcess(e) {
  var leftOut = e.outputBuffer.getChannelData(0);
  var rightOut = e.outputBuffer.getChannelData(1);

  for (var i = 0; i < leftOut.length; i++) {
    // Добавляем немного шума
    leftOut[i] += (Math.random() - 0.5) * NOISE_FACTOR;
    rightOut[i] += (Math.random() - 0.5) * NOISE_FACTOR;
  }
}
```

Основная проблема использования узлов скриптовой обработки — это производительность. Реализация таких вычислительно сложных алгоритмов на JavaScript работает значительно медленнее, чем их выполнение напрямую в нативном коде браузера.

Интеграция с другими технологиями

Web Audio API делает обработку и анализ звука фундаментальной частью веб-платформы. Будучи ключевым строительным блоком для веб-разработчиков, он спроектирован так, чтобы легко интегрироваться и работать совместно с другими технологиями.

Настраиваем фоновую музыку с помощью HTML-элемента `<audio>`

Как я упоминал в самом начале книги, у тега `<audio>` есть множество ограничений, из-за которых он плохо подходит для игр и интерактивных приложений. Однако у него есть одно важное преимущество — встроенная поддержка буферизации и стриминга, что делает его идеальной технологией для воспроизведения длительных аудиотреков. Загрузка большого буфера медленна с точки зрения сети и затратна с точки зрения управления памятью. Поэтому использование `<audio>` идеально подходит для воспроизведения музыки или саундтрека в игре.

Вместо того, чтобы идти обычным путём — загружать звук напрямую через `XMLHttpRequest` и затем декодировать буфер — вы можете использовать узел источника медиапотока `MediaElementAudioSourceNode`. Он создаёт узлы, которые ведут себя почти так же, как узлы источников звука `AudioSourceNode`, но работают поверх существующего тега `<audio>`. Как только мы подключим этот узел к нашему аудиографу, можно применить все полученные ранее знания о Web Audio API для создания интересных эффектов. В этом небольшом примере к тегу `<audio>` применяется низкочастотный фильтр:

```
window.addEventListener('load', onLoad, false);

function onLoad() {
  var audio = new Audio();
  source = context.createMediaElementSource(audio);

  var filter = context.createBiquadFilter();
  filter.type = filter.LOWPASS;
  filter.frequency.value = 440;

  source.connect(this.filter);
  filter.connect(context.destination);

  audio.src = 'http://example.com/the.mp3';
  audio.play();
}
```

Захват звука в реальном времени

Одной из наиболее востребованных функций Web Audio API является интеграция с `getUserMedia`, которая даёт браузерам доступ к аудио- и видеопотокам подключённых микрофонов и камер. На момент написания этой книги эта функция была доступна в Chrome только при включении специального флага. Чтобы её активировать, нужно было открыть `about:flags` и включить эксперимент `Web Audio Input`, как на Рисунке 7-1.

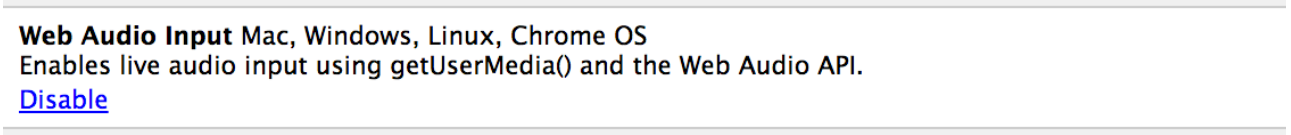

Рисунок 7-1. Включение эксперимента Web Audio Input в Chrome

После включения этой функции можно использовать узел Web Audio — `MediaStreamSourceNode`. Этот узел оборачивает объект аудиопотока, доступный после его инициализации. Это полностью аналогично тому, как узлы `MediaElementSourceNode` оборачивают элементы `<audio>`. В следующем примере мы визуализируем живой аудиопоток, обработанный *режекторным фильтром* (notch filter):

```
function getLiveInput() {
  // Запрашиваем только аудиопоток
  navigator.webkitGetUserMedia(
    { audio: true },
    onStream,
    onStreamError
  );
};

function onStream(stream) {
  // Оборачиваем MediaStreamSourceNode вокруг аудиопотока
  var input = context.createMediaStreamSource(stream);

  // Подключаем вход к фильтру
  var filter = context.createBiquadFilter();
  filter.frequency.value = 60.0;
  filter.type = filter.NOTCH;
  filter.Q = 10.0;

  var analyser = context.createAnalyser();

  // Подключаем граф
  input.connect(filter);
  filter.connect(analyser);

  // Настраиваем анимацию
  requestAnimationFrame(render);
};

function onStreamError(e) {
  console.error(e);
};

function render() {
  // Визуализируем аудиопоток
  requestAnimationFrame(render);
};
```

Другой способ создания потоков основан на `WebRTC PeerConnection`. Подключив поток связи к Web Audio API, вы, например, сможете пространственно расположить несколько участников видеоконференции.

Управление аудио при переключении вкладок

Разрабатывая веб-приложение, в котором есть воспроизведение звука, важно учитывать состояние страницы. Классический пример проблемы: когда один из множества открытых табов воспроизводит звук, но непонятно, какой именно. Для музыкального приложения это может быть допустимо, ведь музыка должна продолжать играть независимо от того, видна ли страница. Однако для игры чаще всего нужно приостанавливать игровой процесс (и воспроизведение звука), если страница перестаёт быть активной.

К счастью, API видимости страницы `Page Visibility API` предоставляет возможность определять, скрыта страница или видна. Состояние можно узнать через булево свойство `document.hidden`. Событие, которое срабатывает при изменении видимости, называется `visibilitychange`. Так как API до сих пор считается экспериментальным, все названия свойств и событий пишутся с префиксом `webkit`. С учётом этого, приведённый ниже код останавливает узел-источник, когда страница скрывается, и возобновляет его работу, когда страница снова становится видимой:

```
// Слушаем событие изменения видимости страницы
document.addEventListener('webkitvisibilitychange',
onVisibilityChange);

function onVisibilityChange() {
  if (document.webkitHidden) {
    source.stop(0);
  } else {
    source.start(0);
  }
}
```

Заключение

Спасибо за чтение этой книги по Web Audio API. Если вы новичок в цифровом аудио, я надеюсь, что я помог вам разобраться в некоторых фундаментальных концепциях. Если вы уже были знакомы с Web Audio API, я надеюсь, что вы узнали что-то новое.

Перед тем как закончить, я хочу порекомендовать вам несколько отличных книг и ресурсов — они показались мне крайне интересными и полезными, и на них я опирался при подготовке этой книги. Мой топ-5 включает в себя:

1. Спецификация Web Audio API от Chris Rogers.
2. *Real Sound Synthesis for Interactive Applications* от Perry R. Cook (A K Peters, 2002).
3. *Mastering Audio: The Art and the Science* от Bob Katz (Focal Press, 2002).
4. Туториалы от Andy Farnell: “Practical Synthetic Sound Design”.
5. “All About Decibels, Part I: What’s your dB IQ?” от Lionel Dumond.

Устаревшие возможности API

Web Audio API всё ещё развивается, и некоторые методы добавляются, удаляются и переименовываются. В этом разделе описаны некоторые из последних изменений, внесённых в API:

- `AudioBufferSourceNode.noteOn()` был изменён на `start()`.
- `AudioBufferSourceNode.noteGrainOn()` был изменён на `start()`.
- `AudioBufferSourceNode.noteOff()` был изменён на `stop()`.
- `AudioContext.createGainNode()` был изменён на `createGain()`.
- `AudioContext.createDelayNode()` был изменён на `createDelay()`.
- `AudioContext.createJavaScriptNode()` был изменён на `createScriptProcessor()`.
- `OscillatorNode.noteOn()` был изменён на `start()`.
- `OscillatorNode.noteOff()` был изменён на `stop()`.
- `AudioParam.setTargetValueAtTime()` был изменён на `setTargetAtTime()`.

Кроме этих изменений, многие константы в Web Audio API меняются с переменных на строковые перечисления. Например, типы фильтров меняются с `filter.LOWPASS` на `lowpass`, типы осцилляторов меняются с `osc.SINE` на `sine` и др.

В книге я использовал новые версии всех API, поэтому тем, кто работает со старыми версиями, может потребоваться вернуться к прежним методам и именам констант.

Для получения наиболее актуальной информации о изменениях в нейминге, обратитесь к спецификации Web Audio.

Глоссарий

Аудиоконтекст

Контейнер для всех аудиоузлов внутри графа Web Audio API.

Глубина битности

Количество битов, выделенных для каждого значения в потоке аудиоданных.

Битрейт

Количество битов в секунду, которое будет выводить сжатый аудиофайл.

Центы

Логарифмическая единица измерения музыкальных интервалов. В равномерно темперированном строе октава делится на 12 полутонов, по 100 центов каждый.

Клиппинг

Искажение звуковой волны, возникающее, когда её уровень превышает максимально допустимое значение (`0 dBFS`).

Децибелы

Относительная единица измерения интенсивности звукового сигнала.

dBFS

Уровень звука относительно полной шкалы (номинально `0 dBFS`). Это значение будет отрицательным, пока звук не находится в состоянии клиппинга.

dBSPL

Уровень звукового давления относительно порога слышимости человека.

Быстрое преобразование Фурье

Алгоритм, разбивающий звуковую волну на составляющие синусоиды.

Герцы

Единица измерения частоты. Число событий в секунду.

Частота Найквиста

Половина частоты дискретизации аудиобуфера.

Импульсно-кодовая модуляция (Pulse Code Modulation, PCM)

Способ хранения звуковых волн в виде массива чисел.

Скорость воспроизведения

Скорость, с которой воспроизводится аудиобуфер.

Частота дискретизации

Число измерений аналогового звука в секунду при квантовании, то есть в процессе преобразования аналогового сигнала в цифровой.

Об авторе

Борис Смусь — фронтенд-инженер в команде Google Chrome Developer Relations, специализируется на мобильной веб-разработке и Web Audio *(прим. переводчика: на момент написания оригинальной книги в 2013 году)*. До работы в Google он был исследователем в области человеко-компьютерного взаимодействия в Университете Карнеги-Меллон и инженером-программистом в Apple.