BROWSER AS GAME ENGINE - EXPERIMENTAL APPROACH ON INVESTIGATING WAYS TO IMPLEMENT MUSIC ELEMENTS INTO GAMES WHICH ARE PROGRAMMED WITH MODERN WEB TECHNOLOGIES.

MEDIALOGY MASTER THESIS

BY: *KRISTJAN KALMUS*

MAIN SUPERVISOR: *JANNICK KIRK SØRENSEN*

CO-SUPERVISOR: *STEFANIA SERAFIN*

DEPARTMENT OF ARCHITECTURE, DESIGN AND MEDIA TECHNOLOGY

COPENHAGEN, SPRING 2014

Table of contents

Table of contents 3

Preface 6

1. Introduction 7

2. Background 9

2.1 The game audio history 9

2.2 The importance of sound in games 10

2.3 Types of sounds 12

3. Working with audio 14

3.1 Limitations 14

3.2 Practices to cope with the limitations 17

3.2.1 Saving file as mono *17*

3.2.2 Downsampling and compression *18*

3.2.3 Frequency analysis and concatenation *18*

3.2.4 Saving file at twice the speed *19*

3.2.5 Different playback speeds *20*

3.2.6 Reducing load time *20*

4. Audio implementation methods and file types 21

4.1 State of the art 21

4.1.1 Adobe Flash *21*

4.1.2 `<audio>`-tag *23*

4.1.3 Web Audio API *25*

4.1.4 Audio Data API *28*

4.2 Sound file types 28

4.2.1 WAV audio *28*

4.2.2 MP3 ..... *28*

4.2.3 OGG Vorbis *29*

4.2.4 WebM *29*

4.2.5 AAC *29*

5. Implementation and methodology 30

5.2 The technical solution of the implementation 30

5.3 Implementation to test looped background audio 33

5.4 Computers' specifications used for testing ... 33
5.5 Measuring and data logging ... 34
6. Implementation and testing ... 35
6.1 Data network speeds and download times ... 35
6.1.1 Introduction and test conditions ... 35
6.1.2 Ways of measuring ... 36
6.2.3 Testing and results ... 37
6.2 Web Audio API decoding times ... 41
6.2.1 Introduction and test conditions ... 41
6.2.2 Ways of measuring ... 42
6.2.3 Decoding times of MP3-audio ... 44
6.2.4 Decoding times of OGG-audio ... 44
6.2.5 Decoding times of AAC-audio ... 45
6.2.6 Decoding times and different audio qualities ... 46
6.2.7 Decoding times and different number of files ... 46
6.3 Looping sounds ... 47
6.3.1 Introduction and test conditions ... 47
6.3.2 Ways of measuring ... 48
6.3.3 Looping and MP3-files ... 48
6.3.4 Looping and OGG-files ... 51
6.3.5 Looping and MP4-files ... 51
6.4 Overcoming limitations of looping ... 52
6.4.1 Introduction, test conditions and measuring ... 52
6.4.2 Test results (audio-tag) ... 55
6.4.3 Test results (Web Audio API) ... 56
7. Reducing the usage of system resources ... 58
8. Conclusive analysis ... 60
9. Conclusion ... 64
10. Future works ... 66
References ... 68
List of figures, tables and code examples ... 72
Appendix A ... 74
Appendix B ... 75
Appendix C ... 76

Appendix D 77
Appendix E... 80
Appendix F 82
Appendix G 84
Appendix H 85
Appendix I... 88
Appendix J 89

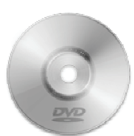
Preface

The thesis includes some informatory icons to make the reading and finding related additional materials easier.

Information in Appendix

When this icon is present it means that some of the relevant data has been included in the Appendix. The text on the right of the icon explains what kind of data the Appendix has, so that the reader could make an informed decision whether checking the Appendix immediately is beneficial to the topic at hand or not. Also it makes it easier later on to locate the place in the text where one or another appendix is relevant.

Information on DVD



This paper is accompanied by a DVD. When the DVD icon is present in the text, it means that visual or audio examples have been included to the DVD to illustrate the point made in the text. The text beside the icon explains the type of related data, where exactly it could be found from the disk and how it could be viewed (if applicable).

1. Introduction

Audio elements play an important role in the world of computer games. In the world of browser based gaming, the majority of games have required some sort of third party plug-in to overcome the limitations of the multimedia support in web browsers. The new HTML5 standard, introducing native audio-video support, has added new features to the web, making it possible to implement audio into web applications and to create multimedia games solely by using proprietary web technologies. The HTML5 audio-tag is the standardized way to include audio elements to web. In addition – the Web Audio API is in development, which should open up even more possibilities when it comes to audio, including audio panning, filtering and effects to name a few.

The goal of this paper is to look into the world of audio in games – the limitations, differences and possibilities of audio-tag and Web Audio API when it comes to implementing audio using web languages. This work covers the topics like the ease of use, the differences in implementation, the suitability for different purposes and how one method or another might affect the possible gaming experience (through usage of computer resources, loading times and differences in audio playback).

Problem formulation:

As a more specific problem formulation, the following postulation has been presented: **How well do audio-tag and Web Audio API perform compared to each other and how they can be used in a most optimal way to deliver the best user- and aural experience in browser based games.**

The process of analyzing the problem stated above will consist of different parts – the efficiency of using data delivery networks (loading times, file request times); implementation method specific characteristics; performance related aspects when using different types of audio files; how implementation methods and audio file types behave when used under possible real life condition (in a form of looped background audio). The results will be analyzed and some of the downsides will be looked into in more detail, to inspect possible workarounds to the limitations imposed by the implementation methods and audio encoding technologies.

The research question was tested using two web pages – one for each audio implementation method – which acted as frameworks and were modified based on the nature of each individual test. To make the tests comparable, a set of guidelines were put in place to which both frameworks had to comply with.

This paper is focused on the domain of PC-games and deals with audio which is recorded and sampled before implementation; the field of audio synthesis or MIDI-technology is beyond the scope of this paper. The reason is that limitations applying to the sampled audio are different from the limitations on synthesized or programmatically generated audio.

2. Background

This chapter gives an overview of the history and the evolution of audio in games, why the development of audio technologies is of great importance to the games and how audio can influence the gaming experience. The second half of the chapter will cover the list of different audio types used in games and their roles. The history of the development of the game audio is essential to understanding the current situation of the industry and what kind of expectations a developer might have when working with game audio.

2.1 The game audio history

Audio playback has always been limited. The best example is the hardware development of gaming consoles. First generation consoles didn't have any audio support or it was very limited. Magnavox Odyssey, the very first home video game system, had no sound. The following machines also started to implement sound – simple built-in sound speakers at first, later on more capable sound chips which generated sound for playback through TV-speakers [1]. One of the quite common solutions for sound generation was using 4 channel chips – 3 sound channels and 1 noise generator – which could be found from many gaming consoles throughout the 70's and 80's [2]. Over the decades manufacturers added more channels which enabled programmers and composers to create more complex musical pieces. With the Nintendo Entertainment System (NES) (released in 1983 in Japan and 1985 in US) the Programmable Sound Generator (PSG) was introduced to the gaming consoles and one of the PSG audio channels could have also been used to play audio samples [2, 3, 4]. This was a step forward in the direction of how majority of the audio has been implemented today – the audio consists of recorded and sampled CD-quality stereo audio and is not generated on fly by sound synthesis chips [5]. NES used 5 channels of monophonic audio, a year later (in 1986) Sega introduced sound generators which were able to generate sounds in four octaves each. In 1989 the NEC TurboGrafx-16 had 6 channels with stereo output, during the same time Sega Genesis brought 10 audio channels [5].

In the 90's games' audio started to put more demand on the system resources. This becomes evident when we look at the specification of the game consoles released since the early 90's – Play Station (released in 1994) already had 24 audio channels and 512KB

dedicated memory, Nintendo 64 (released 1996) used shared memory of 4MB, Sega Dreamcast (released 1998) had 64 channels and 16MB of shared memory, Sony Playstation 2 (released 2000) allowed programmers to work with 32MB of shared memory. A year later Microsoft released Xbox which had 64MB of memory, Xbox later version – Xbox360 (2005) – had 512MB of shared memory [2].

2.2 The importance of sound in games

The capabilities of audio have been varied a lot over the decades as seen from the previous quick overview of the game audio – the early game consoles lacked audio support or had primitive internal speakers to generate simple beep sounds. Later on multiple channel sound generation chips were added to the console boards, but there was still a long way to go until the technology allowed the usage of pre-recorded audio [2, 5]. It is understandable why audio has gotten and gets less attention in computer games since for players, the two most important features in computer games are playability and graphics, when they are choosing a game to buy [6].

Even though players don't consider audio as important, having a quality in-game audio can benefit the game in various ways. The game development studios have understood this and as a result they often have on-site sound engineers, working with games, to create the best possible aural environment for the games [1, 5].

One might ask why is audio so important to the games. Sound plays different roles in games – it separates the player from the surrounding distractions, reflects the game state, acts as a feedback medium for player actions, helps the storyline progression, and makes the fictional world seem more realistic [1, 5, 7, 8].

Lately more and more attention is given to the immersive qualities of games which are important in many ways. Immersion by definition is a state where entire player's attention is on the game [9, 10, 11], the sense of time is reduced [9, 11, 12] and in some sense the player becomes a part of the experience itself [10, 13]. On one side, when players are immersed then different shortcomings in usability and conflicts between expectations may often remain unnoticed [14]. On the other hand, the goal of player-centered game design approach, is to increase player enjoyment [15], making immersion a vital element for the success of a game. If players do not enjoy the game, they will not play it [11]. Studies have shown that sound plays vital part in the immersive qualities of a computer game [1, 9, 13,

16]. As noted above, immersion has been associated with number of features: lack of awareness of time, loss of awareness of the real world by being completely focused on the game at hand, involvement and a sense of being in the task environment. Also emotional involvement seems to be one of the key factors [13, 17] and sound/music is a very powerful medium to affect one's emotions [5, 18]. In addition it has been suggested that immersion correlates to the number of attentional sources (visual, auditory and mental) needed as well as the amount of each attentional type [9].

Gameplay immersion can have different dimensions [10] – this means that one could have an immersive experience with early "Pong" game where audio is very primitive and does not contribute that much to the overall immersive experience – but increasing the realism of a soundscape can increase the strength of the sense of immersion [16]. In addition, better audio quality improves the overall experience - sensory immersion is related to the audiovisual execution of the game, and audiovisual quality and style has been regarded one of the central aspects of a good digital game, meaning that often higher quality audio leads to the higher level of immersion [10]. As audio hardware has matured, the quality of audio has tried to keep up with the hardware improvements; this includes adding surround sound to games [1]. Even though web games may not include surround sound, it is still important to deliver a reasonably high quality audio, since obviously it adds to the overall look and feel of the game and contributes to the gaming experience.

Immersion in terms of audio, is a presentation of a soundscape in a way that listener has an impression of being entirely within a realistic sound environment [19] and it can be used to create the illusion that the world extends beyond the screen [6]. Sweetser and Wyeth have introduced a framework to rate the criterions of enjoyment in games [11] and some of the areas in which audio has a role to play are highlighted as follows:

- games should provide a lot of stimuli from different sources;
- players should receive immediate feedback on their actions;
- players should become less aware of their surroundings;
- games should have a high workload, while still being appropriate for the players' perceptual, cognitive, and memory limits.

As it becomes clear from the text above, sound and music have a distinct role to play in games. Even though players don't consider the sound to be very important when they

choose games for playing [20] it still has a lot to contribute to the gameplay experience. It has been even theoretized that video games will eventually become interactive movies where the psychological effects of music and sound will be dominant [5].

2.3 Types of sounds

To keep the topic from becoming too broad, I am focusing on the audio which is present only during gameplay (background music and gameplay-related sound effects), leaving out sounds played during menu screens, intro and credit sequences, and cinematic cutscenes.

In an average PC or console game[1], a player is exposed to multiple types of audio. In the movie industry, the sounds are most commonly divided into two categories – diegetic and non-diegetic. Diegetic sounds are part of the physical realm of the actors, non-diegetic sounds are external to the story world and often are there to bind the images and contribute to the overall mood like conventional background music [21].

Similarly to the film industry, diegetic sounds in games are those which have a physical source in the game environment and could be heard by the character in the game, i.e environmental sounds (wind, rain, thunder, birds etc), character sounds (breathing, footsteps), action sounds (sword swinging, gunshots, opening a door). The most commonly recognized non-diegetic sound in games is also background music. As in films it can convey the mood but it often gives feedback about the state of the game (i.e music changes during combat scenes or when time starts to run out) and therefore can influence the gameplay. Other types of non-diegetic sounds can include different audiocues which accompany banners or signs. The signs or banners are instructions, tips, and rules. These instructions are presented not as objects belonging to the fictional world but rather superimposed text, although part of the game [16, 22].

Since games are dynamic entities, the audio in games can broadly categorized as diegetic/non-diegetic, but within those categories it can be separated even further. In games we can also talk about interactive and adaptive sounds – environmental sounds and background music which react to the in-game day-night cycle, action sounds which are

[1] Author considers "an average PC or console game" as something which has meaningful game audio (gives feedback to the player), requires mouse/keyboard or game controller as an input device and requires some time commitment by the player.

played according to the player's actions and changing environmental sounds [1]. The game audio can be classified even in a more detailed level - in a literature another layer has been introduced to the game audio, dividing diegetic and non-diegetic sounds also into masking sounds (sound signal is diegetic but signifies a non-diegetic event) and symbolic sounds (sounds relate to the in-game events while signals remain non-diegetic) [22]. Since in this paper the implementation is not focusing on to 3D game world, most of the sounds are non-diegetic and can be regarded as symbolic sounds.

3. Working with audio

This chapter gives an overview of the limitations the audio designers have to work within, how those limitations might limit games/gameplay and how one could cope with them to some extent.

3.1 Limitations

Since the quality of the games are strictly in direct correlation with the amount of computing power at hand, programmers have often developed techniques to overcome certain limits or to have a best possible solution inside the system restrictions. The game audio programming is no different. Implementing audio in games has always had limitations and restrictions the developers have had to cope with. Even though it often feels that with the huge advancements in terms of processing power and system memory one should not worry about the system resources, it is still an issue. The number of sound channels used in games has grown from nothing to basically an infinite number of channels (dependent of the amount of system resources) and from the piezo-speakers to high quality surround sounds, but as technology advances new restrictions have emerged in the process. The system resources are shared between different processes and in games, audio processing is often not the main priority. In some cases it might even be necessary to discard a number of sound assets because of the limits stated above [23].

There is a paradoxical conflict between different variables of the audio. Stevens and Raybould in their book call it the triangle of compromise [7].

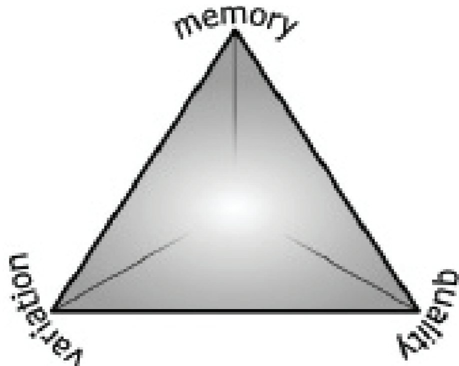


Figure 1 - The triangle of compromise in game audio by Stevens & Raybould

The "triangle of compromise" in game audio development consists of three elements – variation, memory and quality. The audio designer has to keep in mind that there is a constant battle between these three properties and a balance has to be found between them. For example having a huge number of different high quality effects is very demanding on the memory. If an amount of memory is limited, then one has to decrease the number of used samples or decrease their quality.

Theoretically each sound implemented in the game has a certain position in the "triangle of compromise" – having a sound element (for example sound of footsteps) with a small memory footprint[2] means that also the quality has to be low and not many variations can be used, meaning that the same low quality sound will be played over and over again. A good example here are the footsteps sounds in the game Final Fantasi XII – in real life when a person walks on the same surface the footsteps will still sound differently, but in Final Fantasi XII there is only one sample for each surface type, which eventually gets very annoying and tend to break immersion.

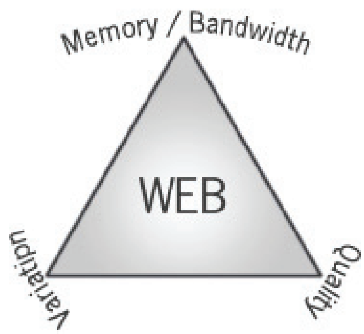


Figure 2 - Triangle of compromise for web conditions

The "triangle of compromise" also holds well in terms of web-based games. In case of the web the triangle should also include bandwidth, since even though memory still remains an issue, the usage of audio in web is also strongly influenced by the bandwidth limitations. Most of the audio features are more-less dependent on the JavaScript which introduces another layer of limitations [24].

JavaScript is the programming language of the web. In web JavaScript is mainly used alongside other web technologies such as HTML (used to describe the content of pages) and CSS (used to change the presentation of pages). Through JavaScript one can interact

[2] Memory footprint – the amount of memory software uses when running. (http://www.pcmag.com/encyclopedia/term/60598/memory-footprint)

with HTML and CSS and manipulate the web elements, thus making it the language which enables developers to specify the behavior of pages [25]. JavaScript makes it possible to add animations, interactivity and dynamic visual effects to the web page [26], so it is also the main language for creating visuals for the web based games. It is also vital when it comes to implementing and manipulating audio elements in browsers. Using the HTML5 audio-tag to include audio to the page doesn't require any scripting – by including the "`controls`" attribute to the audio-tag the default playback controls will be displayed on the page, which enables visitors to listen to the included audio. When the "`controls`" attribute has been left off, then usually the audio has been controlled through dedicated script [27]. In the current paper the HTML-page for presenting audio files with different qualities (can be found from the DVD) rely solely on the playback controls provided by the browser itself. The testing environments and the rest of the pages with audio use only JavaScript for controlling the playback, since many different audio manipulations are not supported natively (e.g fading one audio clip into another).

Audio and test environment examples *– A number of examples have been included to the DVD. Examples have also made available in the web and are accessible from http://www.webgamesaudio.com/masters/*

JavaScript is not as well optimized as some other programming languages and cannot take advantage of different performance improvement techniques – meaning that running a lot of JavaScript code can be quite CPU-intensive. There are number of bottlenecks, which could affect the performance and cause different glitches in audio playback [24].

Probably one of the most influential issues in web is latency – the time between user input event and corresponding audio playback. In Web Audio API specification documents, there are listed number of factors, which may cause latency: input device latency, internal buffering latency, digital signal processing (DSP) latency, output device latency, distance of user's ears from speakers etc. All the elements contribute to the total audio latency. Long audio latency is an unwanted property – it may affect timing, give the impression of sound lag or the game being non-responsive [24] thus directly affect the gameplay and lower the quality of the experience.

Figure 3 - The basics of latency

Multiple studies have made about audio-visual simultaneity and in case of films and videos, audio is considered to be out of sync with video when a sound is approximately 75ms early or 90ms late (some of the studies have came to different conclusions with longer times from 130ms early and up to 250ms late). [28] In games the early audio timing doesn't apply, since audio playback is dependent on the gamers' actions. I would theorize that in case of games, the latency should be smaller, since gamer usually can expect a sound based on his/her actions. Web Audio API documentation mentions latency from 3-6 milliseconds up to 25-50 milliseconds to be reasonable (of course it depends on the type of application) [24].

3.2 Practices to cope with the limitations

Because of the limited nature of audio processing, different techniques, solutions and practices have been developed over time to cope with the limitations and fully utilize the capabilities of the audio. In a book „The Game Audio Tutorial" Richard Stevens and Dave Raybould (with the help of numerous contributors) discuss different ways how to work with audio in games and to make audio memory footprint smaller by finding a compromise between audio quality and required resources, while giving gamers the best aural experience possible [7]. The following is a list of techniques represented, in the book, on how to reduce the usage of resources by audio elements. Each list element also includes a little analysis on if and how it would be possible to implement that specific method using the audio implementation methods for web.

3.2.1 Saving file as mono

Saving an audio file as mono is the quickest way to reduce the file size two times. Saving audio files as mono has been mentioned by different sources - Adobe Community Help suggests that if no compression has been used then it is a good practice to use mono sounds [29], game{closure} DevKit Docs suggest that mono files should be used when possible [30], sound effects and speech audio are usually saved as mono files [5].

3.2.2 Downsampling and compression

The easiest way to reduce the amount of used resources is to make sound files smaller by downsampling or by using some compression algorithms. Downsampling usually results in a loss of audio quality. Using compression algorithms means that to some extent it is possible to reduce the file size while maintaining the audio quality relatively well. The problem with compressed audio is that even though the file size is smaller, it often takes slightly more computing power to play it because it has to be decoded first. Additionally in some cases there might arise support problems with compressed audio – for example using compressed audio in web browsers can be tricky, since different browsers support different compression algorithms [27, 31, 32], making the implementation more complicated (programmer have to make different compression types available to the browser which then chooses the file it can play). The browser support for different file types will be covered in later chapters.

Downsampling and compression examples *– Examples have been included to the DVD, where audio has been saved on different qualities to give an overview how different rendering settings can affect audio quality. Examples have also made available in the web and are accessible from http://www.webgamesaudio.com/masters/*

3.2.3 Frequency analysis and concatenation

WAV-file with sample rate 44100 Hz, allows one to recreate all the possible frequencies in a human hearing range, which is approximately from 20Hz to 20000Hz. To have a digital representation of sound which has harmonics up to 20000Hz one has to have a sample rate at least twice the size [7, 33]. If the audio signal doesn't contain any high frequency content, it is possible to render the audio using lower sample rate thus making the file size smaller without losing quality. If some part of audio file has high frequency elements in the beginning or in the end of it, then one of the special techniques mentioned in the book is to cut the audio file up and save the part with the high frequency audio in a separate high sampling rate file, and the rest of the file with a low sampling rate. During playback one file will be played instantly after another, resulting in a seemingly one audio file but in total with smaller memory footprint.

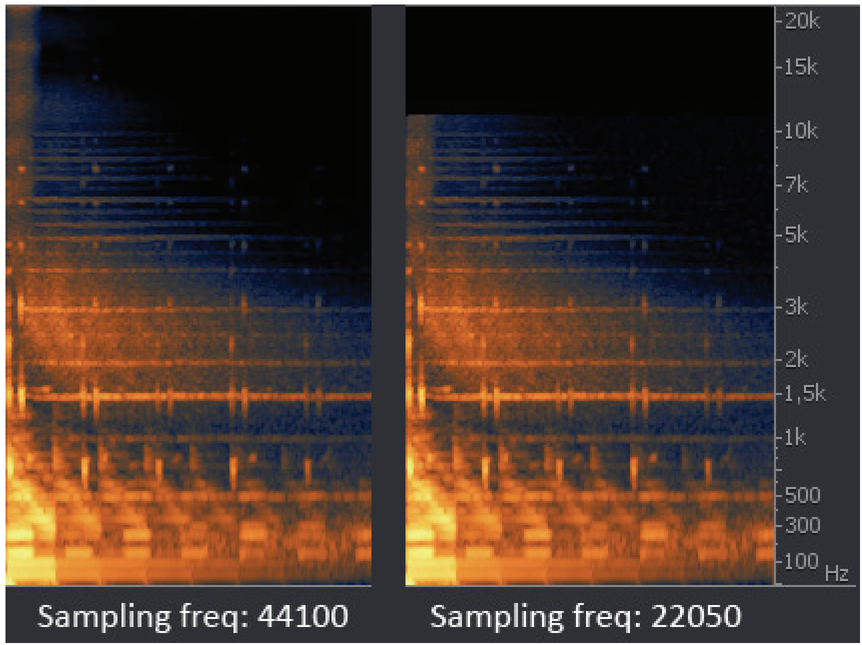


Figure 4 - Differences in frequency data when different sampling frequencies have been used. Colors represent the volume level of the sound on different frequencies (cyan is lowest, light orange is highest)

The previous figure represents the visual audio spectral data of a sound effect I created. As one can see, there are only a limited amount of high frequencies present in the beginning of sound file, which means that concatenation could be successfully used on this audio file. Also if the lack of high frequencies do not affect the audible quality of the audio, one can simply save audio with a lower sampling rate.

This is an advanced functionality that game engines are able to provide, but implementing it in the web can be more complicated, since this is not a native functionality that web languages could provide.

3.2.4 Saving file at twice the speed

One way of decreasing the file size but not sacrificing much of the quality is to increase the raw audio source playback 2 times before rendering it out as a game audio asset. It could be done in different audio editing programs with a time compression functionality which allows manipulating with the audio playback speeds. In the game itself, the file would be played back 2 times slower, thus creating the feel of the original sound effect. This technique makes it possible to save 50% of the audio file size compared to the original "unstretched" audio.

In the web both HTML5 audio-tag and Web Audio API support playback speed changes [24, 27].

3.2.5 Different playback speeds

One fairly common technique is to use the same audio file for multiple effects simply by changing the playback speed of the audio file. This allows a reducing of audio memory load and also reducing the number of files included in the game.

3.2.6 Reducing load time

One of the techniques used in computer games is to concatenate the audio effects into one single file. Reading a number of different audio files from hard drive or from optical disk induces delay – even though it may not be a very long delay it still can result in unwanted effects not acceptable by game developers. By including audio markers to a single file it is possible to start playing from different places. The problem in this case is the following – is it possible to make browsers to recognize media markers inside a sound file? The technique itself would benefit page loading time, because for each file the browser loads during the opening of a web page, it has to send a separate request to the server. Reducing the number of requests made by the browser decreases the page loading time (as we can see from the test results covered in later chapters).

4. Audio implementation methods and file types

This chapter talks about the ways the audio can be implemented in browser based games, covering the currently most widely used Adobe Flash and the native browser technologies including their technical possibilities and characteristics. In the second part of the chapter the sound file types, which can be used in browsers, have been covered.

4.1 State of the art

4.1.1 Adobe Flash

When talking about browser based games, then these games are more casual type of games. Casual games are commonly described as games which allow people to have a meaningful play experience within a short time frame [34]. For example one game round of Bejeweled Blitz, one of the most popular games in Facebook [35], lasts for one minute, while in case of hardcore games[3] one round may take up to an hour or sometimes even more. This has a lot to do with web browsers' and bandwidth limitations, therefore also the audio of browser based games tend to be rather limited, for example one background music loop plus a handful of sound effects.

Currently most of the games which can be played through browser will need the browser to support of Adobe Flash which has been the *de facto* standard for web-based gaming [31]. Adobe Flash has been on the web gaming scene for a long time. It was originally designed for doing web drawings and animations but has evolved a lot since [36]. Today one can do many things using Flash – it includes creating animations, 3d effects, play audio (in which this paper is most interested in), multimedia streaming, Flash can be used for presentations, creating interfaces for info kiosks, creating games, mobile and desktop applications [37].

In terms of Flash's audio capabilities most of it is available through a Flash-specific scripting language called ActionScript. ActionScript is a language used to program interactive Flash content. It has some similarities with JavaScript but also inherits some elements from languages like Java and C and it can be used to control animations, data, playing audio and

[3] Hardcore game is traditionally considered to require a large time commitment for a meaningful experience and to make demands on the skills and commitment of a player itself. [23]

video, user events etc and for accessing Flash libraries and APIs [38]. ActionScript was introduced in 2000, its latest third version was released 2006 and it introduced a number of new possibilities [37]. In Adobe Flash there are two types of sounds: event sounds (has to be downloaded before playing) and stream sounds (playback will start as soon as enough data has been downloaded). These types of sounds can be used in different ways: to have a sound played continuously or synchronize it with a specific animation. In Flash there is even a special event that can be used if one wants to trigger another event after the sound has finished playing. In Flash one can load sounds dynamically and have access to audio envelopes [29].

In Adobe Flash audio playback uses different classes. Each sound has to be encapsulated into Sound object, which also deals with loading and buffering audio data. The playback of Sound object is controlled through SoundChannel class. The panning and volume of a SoundChannel can be controlled with SoundTransform class. From there the sound is forwarded to SoundMixer class, which is the global mix of all played sounds. If overall volume and panning has to be changed, then it can be done through SoundMixer's own SoundTransform class. At any given point the maximum number of mixed SoundChannels is 32 [38].

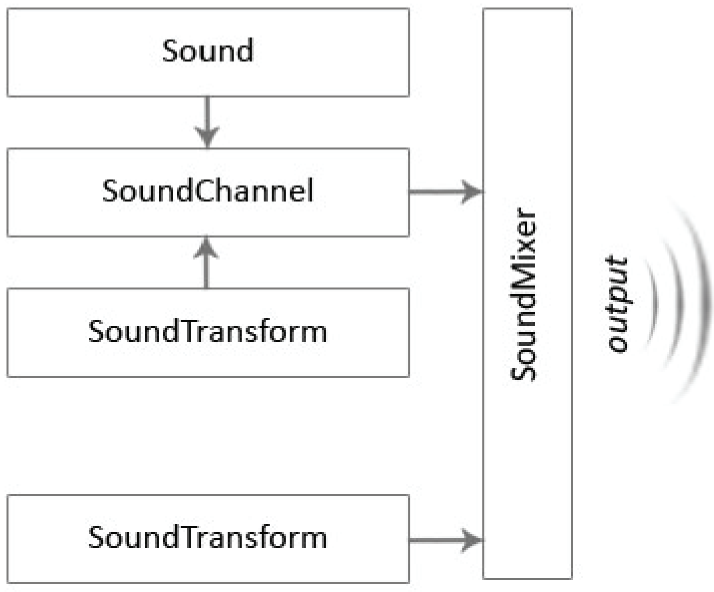


Figure 5 - The class structure of Adobe Flash sound system

The previous figure shows in overall the hierarchy of sound system classes in Adobe Flash. This kind of approach makes it possible to tweak each sound or group of sounds separately.

A list of functionality what Adobe Flash allows to do according to [38]:

- "Seek" functionality or in other words to determine the starting point of the playback in the audio file.
- Looping (the number of loops has to be set).
- Possibility to access MP3-files' metadata (i.e song name, artist, track number, album name etc).
- Display sound's waveform or frequency spectrum.
- Change audio playback sample rate/speed.
- Extract any portion of audio and modify its data.
- Audio synthesis and dynamic writing to audio buffer

In Flash there are multiple ways how to implement sound – it is possible to work with audio by using ActionScript or to add audio to the timeline. The way of implementation depends on the usage of the sound – for example when sound has to be exactly in sync with an animation then it is done by including a sound file to the animation timeline. If syncing is not a priority then it is also possible to stream audio, but if the audio has to match with some animations, then on slower connections it could result in a bad user experience [37].

Adobe Flash has set high standards to how one could work with audio in the web. Since usage of Flash for websites is declining quite rapidly [39] and Adobe is paying more and more attention on creating tools to allow content to be exported into web standards [40], it also means that a browsers native audio support has to keep up with the developments in this area.

4.1.2 `<audio>`-tag

Implementing sound elements to a web page has always been a problem. This is one of the reasons Flash become standard for web-based gaming. Before HTML5 there was no standardized native support for audio embedding to the page. The new tag included in HTML5 is `<audio>` tag [31]. The most basic code for embedding an audio file into a web page looks like the following:

```
<audio controls>
  <source src="myAudio.ogg" type="audio/ogg">
  <source src="myAudio.mp3" type="audio/mpeg">
Your browser does not support the audio element.
</audio>
```

Code 1 - The minimum amount of code necessary for adding audio to page. Source: W3Schools

This is how the previous block of code works:

- `<audio control>` defines sound content, the "`control`" part instructs the browser to show playback controls (i.e play-pause button, volume slider).
- Two `source` tags define media source file (of course more than two sources can be included). The reason why there are two media sources with different audio formats is because browsers do not support the same formats, mainly because of patent issues [27, 32].

Appendix A *- More information about the support of different file formats across browsers can be found from the Appendix A.*

- The text "Your browser does not support the audio element" will be displayed only when a browser doesn't recognize `<audio>` tag – older browsers will skip all the tags they do not understand (in this case `<audio>` and `<source>` tags) and since the only thing browser understands is the line of simple text, it will be displayed; in modern browsers audio controls will be displayed (if instructed) and the regular text inside the tags will be hidden as standard [41]. The text between the tags is just an illustrative example – in real life, a better practice would be to include a download link to the intended audio file, so those who use older browsers would still have an access to the audio.

The audio tag also has a number of global attributes which make it easier to access specific media element or set necessary parameters.

Appendix B *– The full list of audio-tag global attributes can be found from the Appendix B.*

There are also a number of media-specific attributes. Most useful tags are the following: src, preload, autoplay, mediagroup, loop, muted, controls, volume. Most of the attributes are self-explanatory but I will cover their functionality shortly: src – The URL of the

embeddable content; preload – attribute to hint browser whether it should download the content automatically or wait for a specific input from user; autoplay – file can be set to start playing as soon as enough data has been downloaded; mediagroup – a way to group more than one media file together (can be used to start the playback of multiple files simultaneously); loop – audio file will be looping; muted – media plays without the sound (user has to unmute manually); controls – if attribute present, then media playback controls will be shown in the browser; volume – sets or returns the volume of the audio [27, 41, 42]. The list of attributes is longer but it is not convenient to list them all here.

In HTML5 it is possible to seek through media (audio and video) and also specify the playback range (play only a portion of the media) [43]. The latter functionality should become handy in browser-based games. As mentioned previously in the "Working within the limits" chapter – concatenating audio files can reduce loading times and be especially beneficial in reducing the number of request made to the server.

`<audio>` element has also number of limitations. Among other things it is difficult to implement precise timing controls, the number of sounds playable at once is limited, pre-buffering a sound is not very reliable, no real-time audio effects, no audio analyzing capabilities [44].

In addition while going through the list of functionality and attributes one could notice that one of the most common parameter is missing – panning. While using audio-tag for audio playback it is not possible to pan audio sources to right or left (there is no reference to panning mentioned in the W3C documentation). If dynamic panning (panning audio elements based on the location of the audio source on the screen) is absolutely necessary then it has to be done through other means.

4.1.3 Web Audio API

Web Audio API is created to enable audio processing and synthesizing in web applications. Its modular structure can somewhat be compared with the Adobe Flash's sound system – the overall audio rendering is defined by number of connected AudioNode objects (the matter of AudioNodes will be covered in more detail later). The Web Audio API standard is currently in the state of Working Draft, which means that it is prone to change [24].

Web Audio API is completely separate from the audio-tag (although it has integration points with other web APIs). Web Audio API overall goal is to make the functionality found

in game engines and audio production applications (mixing, filtering and processing) available to web browsers. The API can be used for multiple cases – games, web audio applications, sound synthesis [44].

Hierarcically the Web Audio API is structured in the following way: all the magic happens inside the AudioContext where all the AudioNode objects are situated. AudioNodes can be connected with each other and any AudioNode's output can also be another one's input. There are four different types of nodes:

- Source node – sources can be audio buffers, live inputs (i.e microphone), audio-tags, oscillators
- Modification nodes – include filters, convolvers, panners etc
- Analysis nodes – analyzers
- Destination node – audio outputs and offline processing buffers. Audio doesn't have to be played, but can also only have a visual representation acting as a final destination.

The number of nodes can vary depending on the certain case. Separate sound sources can have a different number of nodes in the signal flow path [24, 44].

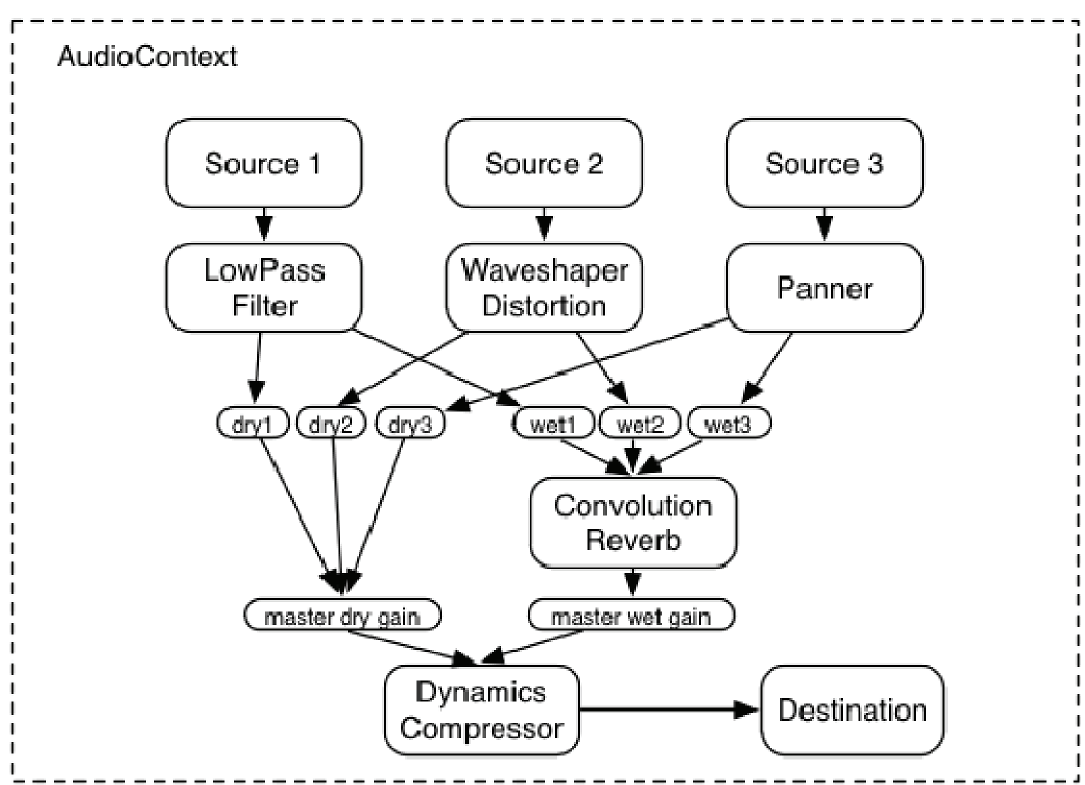


Figure 6 - AudioContext with different AudioNodes. Source: W3C [24]

Web Audio API W3C documentation lists a number of predefined features which can be used for processing the sound. Many of the features are made specifically for games to improve the gaming experience and increase the immersive qualities of the game. Some of the more relevant features for this paper are: low latency sound playback, automations, audio spatialization (different panning modes, distance attenuation, occlusion, obstruction, Doppler shift, source-listener[4] model), high quality room effects (small/large room, cathedral, concert hall, cave, comb filter effects to name a few), dynamics compression, audio filters. Most importantly this list of features supported by the Web Audio API makes it possible to use it for 3D games since audio sources can be placed in the 3D space and playback parameters being changed depending on the relative location of the audio source from the listener [24].

Since Web Audio API is meant to extend the capabilities of web browser, there are still some performance considerations which come with it. One of the most relevant issues is latency (discussed above in more details) – the time between user action and a sound being heard.

To deal with the resource limitations (especially with CPU power), there are some measures implemented into the Web Audio API which help to cope with the processing limitations. The Web Audio API offers a way to monitor CPU load to dynamically implement adjustments, preventing it from going too high. Another option offered by the API is voice-dropping – limiting the number of sounds played at the same time to keep the CPU usage in a reasonable range. This can be done either by setting a certain number of allowed voices or monitor and drop them dynamically. Other recommendations for conserving resources include simplifying audio effects used in the audio signal path and running audio rendering at a lower sample rate [24].

Web Audio API extends the possibilities of web to a greater extent, allowing programmers to create more complex audio solutions. Working with JavaScript in browsers has its limitations, but the API introduces different ways to cope with the limitations to deliver the best possible audio experience.

---

[4]Source-listener acts as a microphone-like device. It receives audio from any given audible sound source in the game space. (http://docs.unity3d.com/Documentation/Components/class-AudioListener.html)

4.1.4 Audio Data API

One has to remember not to confuse Web Audio API with the Audio Data API (sometimes also referenced as Audio API). Audio Data API was developed by Mozilla to extend HTML5 audio and video elements by exposing audio metadata and raw audio data (similarly to Web Audio API to enable working with sample data) but has since been deprecated and its usage is not recommended [27, 45].

4.2 Sound file types

As mentioned previously there are multiple limitations which affect the sound in games and especially when implementing it using web technologies. There are multiple types of sound that one can use. This chapter will cover the principles and differences related to the file types which can be used in web. When implementing audio in the web three audio types will be used – MP3-files, audio encoded with Vorbis inside the OGG-container and AAC encoded audio inside the MP4-container.

4.2.1 WAV audio

Audio files with WAV-extension usually consist of uncompressed soundform data in Pulse-Code Modulation (PCM) representation [27]. PCM sound data is a binary digital representation of an analog sound [1]. Files with .wav extension is a standard for PC, the equivalent file type for Mac-computers has .aiff extension. Since both of these file types are uncompressed it means large file sizes [5]. The large file sizes make the usage of WAV-audio inefficient; also WAV-audio has not been supported by all of the major browsers (without an equivalent file type to fall back to) which is the reason why this type of audio will not be used during testing.

4.2.2 MP3

MPEG-1 Audio Layer 3 is the most well-known audio compression method, commonly referred as MP3 because of the file extension (.mp3). MP3 is a lossy format – in other words it means that some of the audio data will go missing during compression [27].

MP3 compression method is based on psycho-acoustic principles – sounds that are hard to hear (high-frequency sounds or quieter sounds masked by other ones) are removed from the audio data resulting in a smaller file size. Since decoding MP3-files for playback takes

some processing time and power, it is not the best option for instantaneous playback. Using a high compression rates may also result in an unwanted audio artifacts [7, 46].

4.2.3 OGG Vorbis

OGG Vorbis is also a lossy audio compression similarly to the MP3. The OGG is the name of the container format (beside audio OGG can also contain video and metadata). Vorbis is the name of the compression scheme designed to be contained in OGG. The OGG Vorbis is completely open and patent free standard and according to the developers it should deliver better audio quality than MP3 file with the same size [27, 47].

4.2.4 WebM

The WebM is also a file container type, which is designed to specifically use only Vorbis audio codec. According to Mozilla developers network, WebM is preferred over Ogg, since it should provide a better compression to quality ratio [32]. WebM project homepage (http://www.webmproject.org/) states that this format has been developed specifically for delivering media over web (including live streaming) and is open source and patent free, so everybody could use it freely. WebM would not be in later implementations due to the lack of support for this audio format by audio editing programs.

4.2.5 AAC

The Advanced Audio Coding (AAC) belongs also into the family of lossy codecs' and was originally considered to be a successor to MP3 [27]. It uses same basic coding paradigm (as MP3), but delivers better sound quality – through different improvements AAC should reach the same quality as MP3 at about 70% of the bit-rate [48].

In Appendix A there are listed number of different audio file types which can be included to web under certain conditions.

Appendix A *– a list of audio codecs and containers with the corresponding browser supports can be found from Appendix A.*

Ogg and WebM containers can also be used for video, but discussing video related topics is not part of this paper.

5. Implementation and methodology

This chapter gives an overview of the methodology used to test different aspects of the performance of audio implementation methods in web – test approach, used test environments,

5.2 The technical solution of the implementation

Based on my understanding how game audio might be implemented in the web based games, and to make the tests more easily comparable across both implementation methods (audio-tag and Web Audio API), two web pages have been programmed to act as frameworks. Both pages have been programmed to comply with the following technical requirements:

- Uses compressed audio file to save bandwidth.
- Streamed over the Internet (audio data doesn't have to be fully loaded before playback).
- Possibility to change playback volume

Most of the tests were made with 1 minute long audio files; the number and the quality of audio files were changed based on the type of the test. For testing network speeds, a 1-minute long MP3 file (with approximate file size of 938KB) was used as a standard. For tests related to background audio it was made sure that the audio clips were also about 1 minute long, so the test outcomes could be compared more easily. All the audio is in stereo - this means bigger audio files but soundwise is more pleasurable to work with. Also the audio cross-over functionality has been added to the frameworks, since it is important for creating dynamic background music in a form of changing background audio, based on the game state or other parameters. The frameworks have been set to start audio playback automatically after the page has finished loading, which can be done through JavaScript by calling a "play"-method. In case of audio-tag all the audio included to the page are counted and the first audio element added to the page is played; in case of Web Audio API no specific tags have been added to the body of the page, but all audio files are listed in a JavaScript array and the first audio file in the list is played (the audio data also has to be decoded first before the playback could be initiated).

Also the user interface was standardized - both implementations include the following custom UI elements:

- Volume slider – enables user to change the playback volume. The default volume assigned during page load is 20% of the file's original volume. Volume slider will affect the playback volume of the currently playing file (the current file will be tracked by the script).
- Playback buttons – enables user to start or stop the sound. Buttons which cannot be pressed (or when a button press would not change the playback state) are disabled (e.g. when a sound is already playing then "Play" button is grayed out). In case of audio-tag there is no dedicated stop method, which means that the playback will be paused and the playback time set to 0. For WebAudipAPI the play and stop buttons will either create the connection from the source to the destination of the AudioContext class or destroy it.
- Crossfade button – enables to fade from one audio into another. When the button is pressed, a next audio file will start to play (with a playback volume of 0), then over a period of 1 second the next audio clip's playback volume will be increased to the level of the previous audio; at the same time the currently played audio file's volume will be decreased to 0. At the end of the crossfade the next audio clip will be assigned to be as the current audio clip and the previous audio clip's playback will be stopped. The approach to the crossfade is quite the same for both implementations but the complexity of the code is very different – in case of audio-tag, the referencing audio files are as easy as referencing to the id-attribute of the tag; in Web Audio API the way audio data has been referenced is completely different, making the implementation more complex (the current implementation method utilizes the possibilities of multidimensional arrays).
- Audio selection list – this area lists all the audio elements one could select between when doing a crossfade. The equivalent for the selection list are the changes in the game conditions or views (i.e background music changes when going from title window into the game) based on the user interactions. When audio-tag has been used for the implementation, the number of audio elements are counted, added to an array and then the selection list is dynamically created based on the array. A similar process takes place when Web Audio API has been used with the difference

that the audio files included to the page has already been listed as an array and the selection list is generated based on this.

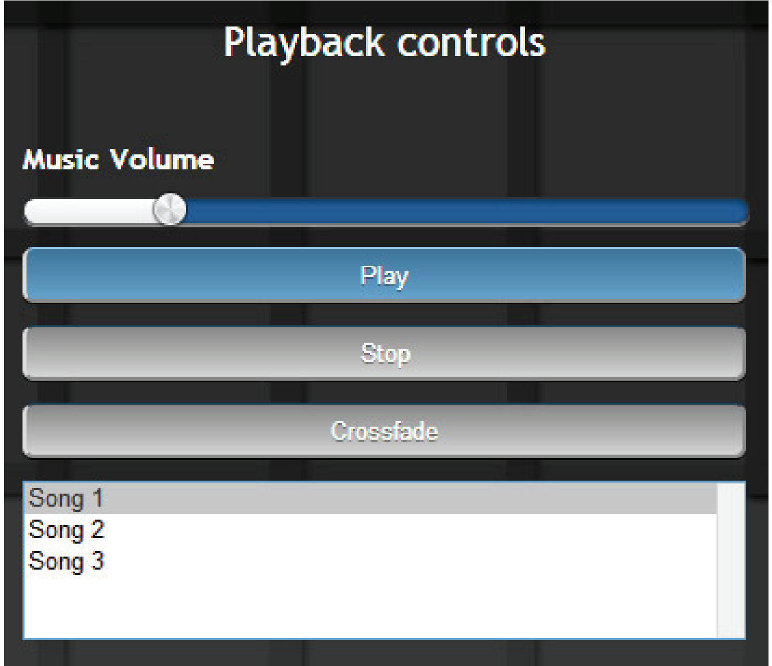


Illustration 1 – Part of the user interface of the test pages

Each page also includes some explanatory information about the current implementation and what type of processes take place during user interaction.

All test pages have been included to the DVD accompanying this paper. Because of the technological reasons Web Audio API-pages has to be run in a server environment. For that reason all the web examples have also made available in the web and are accessible from http://www.webgamesaudio.com/masters/

After standardization the basic implementation (leaving in only the necessary lines of code) the lenght of the implementation is 144 lines of code for the Audio-tag (the page includes 3 audio files) and 204 lines for Web Audio API (the number of audio files included does not change the lenght of the implementation). The implementation relies on jQuery[5] and a custom jQueryUI[6] libraries to provide necessary functionality for tracking user events, manipulating with UI-elements' attributes and controlling UI-widget logic.

[5] jQuery is a JavaScript library, which provides API for working with and manipulating HTML documents, handling events, animations and working with AJAX. (http://jquery.com)

[6] jQueryUI complement jQuery by simplifying the creation of certain user interface elements (http://jqueryui.com)

5.3 Implementation to test looped background audio

One of the most prevalent test cases is to find out the suitability of the audio implementation methods for implementing looped background audio and compare the performances. These tests should also bring out the possible bottlenecks of the technologies when sound effects are to be used and implemented. During testing only nondiegetic music has been used, which plays on the background and is not part of the possible game world itself (i.e environmental sounds like weather, birds or scenery), but also often environmental sounds may suffer the same problems as music (i.e clearly noticeable looping or moments of silence in between loops).

From the early history of games the background music has often been a looped sound sequence or a looped audio file. This was mostly because of the limited nature of system resources and therefore the same piece of music was used over and over again. The downside of this approach was the repetitive nature of the background music [1]. Many of the games use the approach of dynamic audio – the music changes over the course of the gameplay based on the location or game state thus eliminating the repetitiveness from it [5]. According to the game audio academicians the preferred length of the audio file is 3- to 4-minutes without a noticeable breakpoint, if audio is suppose to loop continuously [5]. In this case the lengths of the used audio clips are around 1 minute (as mentioned in the previous subchapter), cut in the way to create a possibly seamless transition at the breakpoint. A 1 minute loop is quite short but it was chosen to reduce the server load during testing.

5.4 Computers' specifications used for testing

Depending on the nature of the test, the patterns and the tendencies between conditions become evident only when tested on multiple systems with different capabilities. Some of tests were carried out on 3 different computers with diverse hardware setups. Hardware configuration data has been gathered using a free hardware identifier "CPU-Z"[7]; to put the CPU-performances into perspective the CPU-benchmark scores[8] have been included at the end of the computer specification details. The hardware specifications are following:

[7] http://www.cpuid.com/softwares/cpu-z.html

[8] The scores are based on the information at http://www.cpubenchmark.net

Computer number	*Hardware specification*
Computer no. 1	Dell Latitude D630 (laptop); CPU: Intel Mobile Core 2 Duo T7100 @ 1.8GHz; Cores: 2; Threads: 2; Memory: 2GB @ 333MHz (CPU benchmark score 1042)
Computer no. 2	Acer Aspire 7739 (laptop); CPU: Intel Core i3 380M @ 2.53GHz; Cores: 2; Threads: 4; Memory: 4GB @ 533 MHz (CPU benchmark score 2117)
Computer no. 3	HP Pavilion 500 (desktop) CPU: Intel Core i5 3350P @ 3.10GHz; Cores: 4; Threads: 4; Memort: 8GB @ 800MHz (CPU benchmark score 6143)

Table 1 - Hardware specifications of the computers used for testing

From computer one to three, the hardware gets better and enables us to see, how different hardware (and the amount of processing power available) affects the performance of different aspects of audio in web. All the tests have been carried out using the latest version of Google Chrome, since it can be considered to be the flagship of browsers (it gets the highest score in "HTML5 TEST"[9] and has the best score in HTML5 audio support). During testing, each computer run only the necessary programs and default background processes to get the most unified results across the systems.

Tests which results are theoretically not affected by the processing capabilities of a computer but by other factors instead, have been conducted only on one setup.

5.5 Measuring and data logging

During testing a number of diverse aspects have been measured. Google Chrome Developer Tools[10] offer a great variety of possibilities for developers to get an overview of the overall page loading times which are useful for optimizing web pages. By default the feedback data is limited as browsers' developer tools don't give any information about the execution of certain processes or separate JavaScript functions unless breakpoints have been set or some specific logging functionality has been programmed into the web page. Google Chrome provides developers with a Console API[11], which provides methods for outputting various data in the console window. The more specific usage of the logging solutions has been covered in more details alongside the explanation of each separate test.

[9] HTML5 TEST analyses browser support of various HTML-tags and attributes. Test can be found from http://beta.html5test.com/index.html

[10] Developer Tools window can be openedby clicking Customize and control menu -> Tools -> Developer tools or by right-clicking on a web page and then clicking on "Inspect element".

[11] Console API documentation: https://developers.google.com/chrome-developer-tools/docs/console-api

6. Implementation and testing

This chapter is focusing on the different aspects of implementing sounds in real life and how different variables influence the usage experience. Implementations involving audio-tag and Web Audio API have been covered separately (if applicable) to compare the ups and downs of each separate method.

6.1 Data network speeds and download times

6.1.1 Introduction and test conditions

Data networks (Internet) speeds and throughput is one of the first things that come into play and might influence the overall user experience. The following graph shows how much time it theoretically takes to download a 1 minute long 128kbps MP3-file. The size of the file would be ~938KB (128kbps * 1000 * 60sec / 8 bit / 1024 = 937,5 KByte).

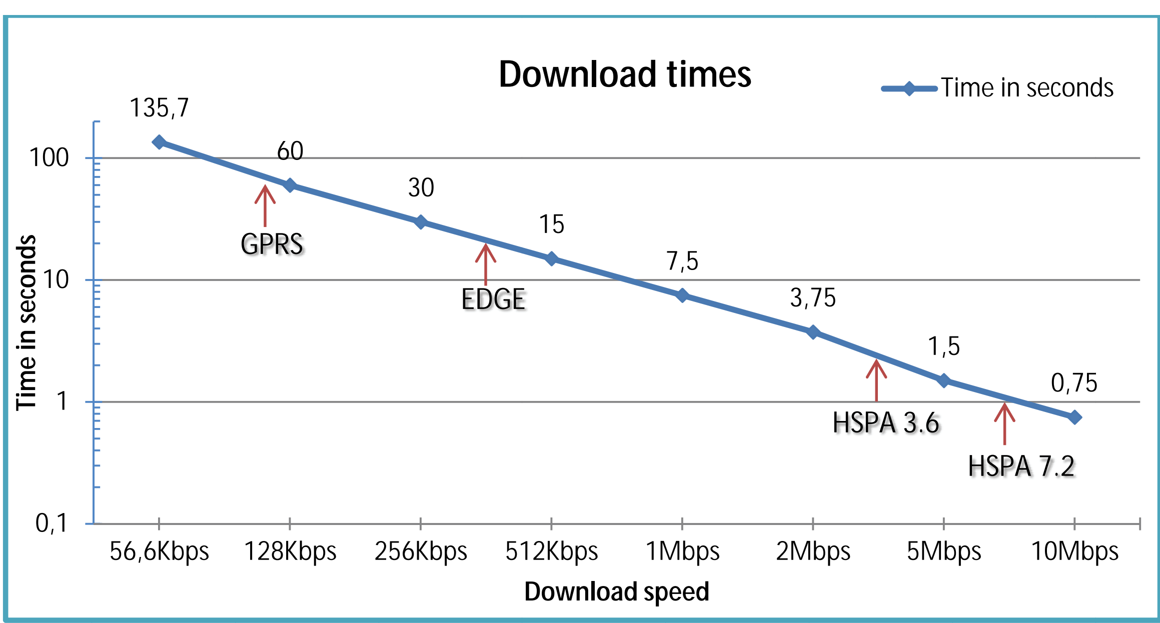


Figure 7 - Download times of 1 minute long MP3 file in case of different download speeds. Red arrows appoint to the approximate position, where mobile connections reside on the speed graph. (based on http://www.techspot.com/guides/272-everything-about-4g/).

The times are true only under the assumption, that the connection throughput is 100% at any given moment (a theoretical maximum); the graph doesn't take into account request times, network delays or other possible factors which affect the real connection speed.

The real loading times are usually slightly different from the idealistic model. To test the real download speed the implementation frameworks were changed to only download and

start playing a 1 minute long MP3-file (with size ~938KB). The test has been made under the following conditions:

- Average server ping time: 25ms
- Trace route analysis showed 7 hops to reach the destination server.
- The download speed on paper should be 5Mbps, average real download speed is ~4.9Mbps. (based on the Ookla Internet speed test at www.speedtest.net)

Among other things the relationship between the number of requested files, the lengths of audio files and server response times have been analyzed, which from the perspective of webpage optimization are important [49].

6.1.2 Ways of measuring

For measuring the speed Google Chrome developer's network tool has been used, which shows how much time it took to download the file and from which parts the total download time consists of. The data is visualized on a timeline, giving developers an overview of the downloaded files, loading order and timings.

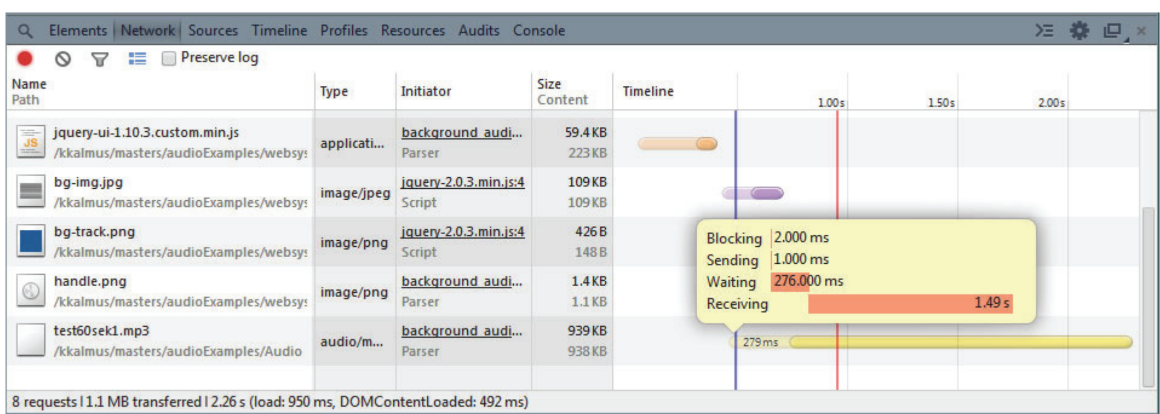


Illustration 2 - Google Chrome developer tools' network tab with timings information.

The download phases are described by the Google developer tools documentation[12]:

- Blocking – Time the request spent waiting for an already established connection to become available for re-use.
- Sending – Time spent sending request.
- Waiting – Time spent waiting for the initial response.
- Receiving – Time spent receiving the response data.

[12] https://developers.google.com/chrome-developer-tools/docs/network

When the total download time of multiple audio files have been measured, the Network tool's view does not provide accurate timings data. Analyzing the request-download timings of multiple files requires working with the Google Chrome Developer Tools' console window, because getting exact timing information from the Network tab's timeline area is rather impossible.

Illustration 3 - Google Chrome developers tools' network tab in case of three audio files. Tests showed that there is no consistency in the order of downloads.

From the visual representation it is difficult to accurately determinate the timings – the download times of each individual file is available, but often it is impossible to get the total download time from the download start of the first audio file until the finish of the last audio file. By entering a specific command into the developers tools console window it is possible to access the same raw timings data the visual representation relies on –the command "window.performance.getEntries()['entry number']" (where 'entry number' is the number of the file requested by the parser or by script) returns the full overview of the timings related to that specific file (the 'entry numbers' were initially determined by trial and error). By comparing the timings of audio files to each other it is possible to determine how much time it takes to request and download the files. The method is based on Google Chrome Developer Tools documentation (https://developers.google.com/chrome-developer-tools/docs/network).

6.2.3 Testing and results

The test has multiple stages to cover many possible loading cases. Each test case was made in two sets – loading audio for audio-tag and for Web Audio API. The download speeds were carried out 10 times for each type of audio to eliminate the possibility that the test would be influenced by random fluctuations of the connection.

The first test was to measure download times of 1 minute of MP3-audio. The timings data was based on the Google Chrome developers tools network window and the average results are shown in the following table:

Audio type \ Download phase	*Sending*	*Waiting*	*Receiving*
AUDIO-tag	1,4ms	281ms	1,49s
Web Audio API	0,9ms	118ms	1,481s

Table 2 - Average loading times of 1 minute long MP3-file in case of AUDIO-tag and Web Audio API

Appendix C *– The full table of all the test results can be found from Appendix C.*

The differences in the sending times are negligible and can be discarded from the measurements in the future. The reason for a clear difference in waiting times is unknown. It might be due to the fact that in case of audio-tag the audio element is part of the DOM (Document Object Model[13]) and the request to the server has been made by the parser[14], leading to a longer response time by the server. In the case of Web Audio API, the request to get the audio file has been made separately by the script and is independent from the loading of the rest of the page. The receiving times in both cases were very close to the theoretical download time (see Figure 7 above) and the differences were also negligible.

A web game might also include multiple audio files, therefore the same test was conducted multiple times with different number of audio files (each a minute long 128kbps MP3-file). The following table summarizes the waiting times when 2, 3 or 5 minutes of audio have been added to the page. All the times are in milliseconds.

Audio type \ Properties	*Request start*	*Response start*	*Response end*	*Total time*	*Request time*	*Receiving time*
AUDIO-tag 2x1min	577,8	948,6	4058,5	3480,7	370,8	3109,9
Web Audio API 2x1min	709,8	868,9	3942,9	3233,1	159,1	3074
AUDIO-tag 3x1min	681,6	1209,9	5849,3	5167,7	528,3	4639,4
Web Audio API 3x1min	883,3	1074	5806,1	4922,8	190,7	4732,1
AUDIO-tag 5x1min	859,3	1463,7	9800,1	8940,8	604,4	8336,4
Web Audio API 5x1min	961,9	1254	9092,3	8130,4	292,1	7838,3

Table 3 - Average loading times in milliseconds of 2 (2x1 minute), 3 (3x1 minute) and 5 (5x1 minute) minutes of audio data in case of AUDIO-tag and Web Audio API.

[13] DOM is a programming API for HTML and XML documents, which defines their logical structure and enables to access and modify elements and content. (http://www.w3.org/TR/1998/WD-DOM-19980720/introduction.html)

[14] Parser is responsible for taking the HTML-file, converting it into an DOM object and processing it (http://www.w3.org/html/wg/drafts/html/master/syntax.html#parsing)

***Appendix D** – The raw data of the tests can be found from Appendix D.*

By increasing the used amount of audio data step by step some of the patterns in loading times become more evident. When Web Audio API has been used, the time of the first request to fetch audio from the server has been made in average 145ms later (compared to audio-tag). The most reasonable explanation is that since for Web Audio API the request is sent by the script and not by the parser, the script will be triggered after the rest of the page has been completely loaded, delaying the start of the first request by a few hundred milliseconds.

With each additional audio file the initial waiting time before the request will be made (request start time) becomes longer. Also the request time becomes longer (time between request and server response) - not so much for Web Audio API but noticeably in case of audio-tag. When audio-tag has been used the request time for one audio file is in average 280ms, for 2 files 370ms, for 3 files 528ms and for 5 files 604ms. The request time seems to get longer when more audio files have been added to the page, but it is difficult to see any specific growth patterns behind it. Receiving times of the audio deviate slightly from the perfect download timings as shown on the Figure 7.

Whether the differences were conditioned by the increasing number of the requests made to the server, a similar test was carried out as a comparison using one 2 minutes long and one 3 minutes long MP3-file (at bitrate of 128kbps). The following table summarizes the test results:

Audio type \ Properties	*Request start*	*Response start*	*Response end*	*Total time*	*Request time*	*Receiving time*
AUDIO-tag 1x2min	530,8	810,2	3905,9	3375,1	279,4	3095,7
Web Audio API 1x2min	942,3	1105,3	4236,7	3294,4	163	3131,4
AUDIO-tag 1x3min*	540,33333	847,33333	16381,666	15841,33	307	15534,33
Web Audio API 1x3min	775,7	892,1	5633,6	4857,9	116,4	4741,5

Table 4 - Average loading times in milliseconds of 2 and 3 minutes (in a single file) of audio data.

*The test with 3 minutes of audio data (in a single file) in case of audio-tag was cancelled after third test round. After buffering 2MB of audio data (the total file size of 3 minutes long MP3-file is ~2.74MB) the download speed was reduced significantly probably because of how browser handles the buffering of bigger audio files (possibly implementing some

sort of network load optimization). This led to download time of ~15 seconds in average, making it impossible to compare it with the rest of the tests. A line from the W3C documentation states the following about how the resource fetching algorithms for media elements can be implemented into browsers: „The rate of the download may also be throttled automatically by the user agent, e.g. to balance the download with other connections sharing the same bandwidth" [41]. It also states that user agent may stop downloading content at any moment and wait for an user interaction and based on this decide what to do next (continue downloading or suspend the media element). Because of the load optimization the test was never carried out for 5 minutes of audio in a single audio file, since it would not have resulted in any reliable data.

***Appendix F** – the raw data of the previous tests (the download timings of 2 minutes and 3 minutes long MP3-files) can be found from Appendix F.*

This test shows that when it comes to the downloading times it does not matter much whether one has included 3 minutes of audio in a single file or uses three 1-minute long files. The timings are roughly the same, therefore the deviation form the ideal download timings has to be caused by other reasons – one of the most likely possibilities is that during the lifetime of a connection it will not be able to maintain the possible maximum speed and drops slightly during the download process.

In addition a quick set of tests with ten 3-seconds and ten 20-seconds long audio files was performed to see, how the number of the included audio files influences the request times. The following table concludes the average of the request times to make it easier to compare them.

Amount of time	*Audio-tag*	*Web Audio API*
1x1min	281	118
2x1min	370,8	159,1
3x1min	528,3	190,7
5x1min	604,4	292,1
1x2min	279,4	163
1x3min	307	116,4
10x3sec	525,6	253,1
10x20sec	609,8	302

Table 5 - Comparative table of request times. All times are in milliseconds.

The comparison shows that the times are not in a straightforward correlation of the total number of audio files included to the page, but also the lenght of the audio files influence the request times.

Appendix F *– raw data of the tests with 10 x 3 seconds and 10 x 20 seconds of audio and 5 x 1minutes of audio can be found from Appendix F.*

The data indicates that the most efficient way is to concatenate files and use only one file, but often this may not be the most convenient way, i.e when working with sound effects it most likely will be easier to use one audio file per sound effect than concatenate them together and then fine-tune manually the playback regions. The specific solution and used techniques depend on the nature of the game and used audio.

6.2 Web Audio API decoding times

6.2.1 Introduction and test conditions

As showed previously, in case of Web Audio API the server response times were smaller; file loading times were pretty much the same meaning that in overall the file loading times were slightly smaller. This does not mean that when using Web Audio API the overall user experience will be better due to the smaller loading times. While using audio-tag the playback starts when enough of the first audio element (which was set to start playing immediately after the page has been loaded) has been buffered. In case of Web Audio API there is a noticeable delay between the point when the page seems to be loaded and when audio playback begins. By design the Web Audio API normally works with buffer arrays

(there are some special cases which will be mentioned later) and files have to be loaded and decoded first, which introduces additional waiting time. The goal of the test is to analyze the factors that influence the decoding time.

The test was conducted in multiple parts: to test the decoding differences across different audio file types, the decoding timings were tested in case of 128kbps MP3, OGG and AAC (in MP4 container). A separate test was conducted to determine if the differences in the audio quality affects the decoding times, for which MP3-files were used with different bitrates (96kbps, 128kbps, 192kbps and 256kbps); for each bitrate there were 5 different test cases – 1, 2, 3 or 5 minutes of audio in the form of 1 minute long audio and to test if the decoding time is affected by the number of file requests made to the server the decoding time of one 5 minute long MP3 file was compared to time which takes to decode five 1 minute long files. To find out, how the computer hardware affects the decoding time, all test conditions were carried out on all 3 different hardware setups (systems detailed specifications can be found from Table 1).

Since Web Audio API requires the page to be in a web server environment, a local web server was created in each test computer using WampServer. WampServer is a web development environment, which enables developing web applications on a local computer without the necessity of having an online hosting solution.

6.2.2 Ways of measuring

To measure the delay time before the playback one has to measure the time it takes to execute the function which decodes the files. To measure the function execution time the `console.time(label)` and `console.timeEnd(label)` commands will be used. When `console.time(label)` is called, a timer will be started and run until `console.timeEnd(label)` will be called (with the same label) which stops the measuring and outputs the time value into console window.

```
BufferLoader.prototype.loadBuffer = function(url, index) {
    var request = new XMLHttpRequest();
    request.open("GET", url, true);
    request.responseType = "arraybuffer";
    var loader = this;
    request.onload = function() {
        console.time("AudioFile" + index);
        loader.context.decodeAudioData(
            request.response,
            function(buffer) {
                if (!buffer) {
                    alert('error decoding file data: ' + url);
                    return;
                }
                loader.bufferList[index] = buffer;
                player.bf[index] = loader.bufferList[index];
                console.timeEnd("AudioFile" + index);
                if (++loader.loadCount == loader.urlList.length)
                    loader.onload(loader.bufferList);
            }
        )
    }
    request.onerror = function() {
        alert('BufferLoader: XHR error');
    }
    request.send();
}
```

Code 2 - JavaScript code responsible for decoding audio data and creating array buffer in Web Audio API with the timer start and end commands (marked in red).

This technique can be used to get information about any JavaScript function performance. In this case those command lines have used and inserted into the code to measure decoding times of audio files. The function represented in Code 2 will be initiated every time an audio file is decoded. By inserting the timer commands in the specific positions (shown in red) it is possible to measure the decoding time of each separate file. To measure the overall length of the decoding process another timer has been set to start before the first call of this code block and stopped before the audio playback begins. After the implementation the console window returns the following data:

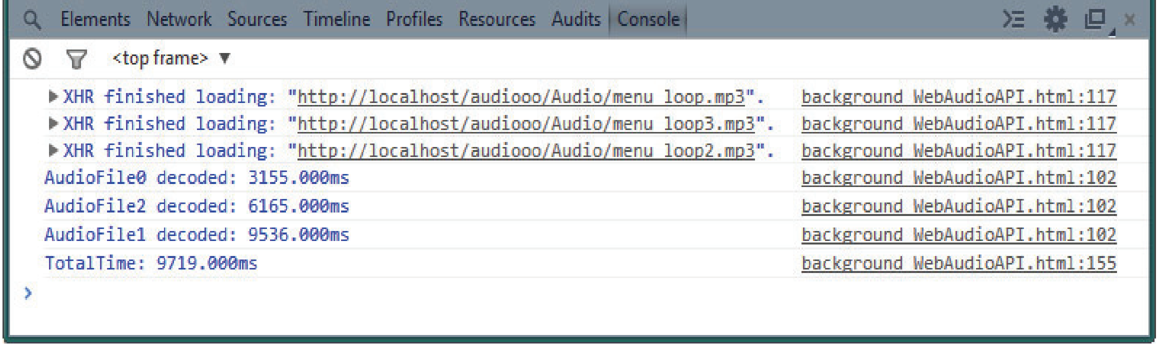


Illustration 4 - Google Chrome console window with the custom timings data.

When looking at the results it becomes evident, that the decoding of audio files are overlapping (decoded partly at the same time) since the sum on of each decoding time is way bigger than the total decoding time (3155 + 6165 + 9536 = 18856 while the total time is 9719). For that reason the decoding time analysis will be based only on the length of the total decoding time.

6.2.3 Decoding times of MP3-audio

The overall timing differences for MP3-files are the following (average decoding times):

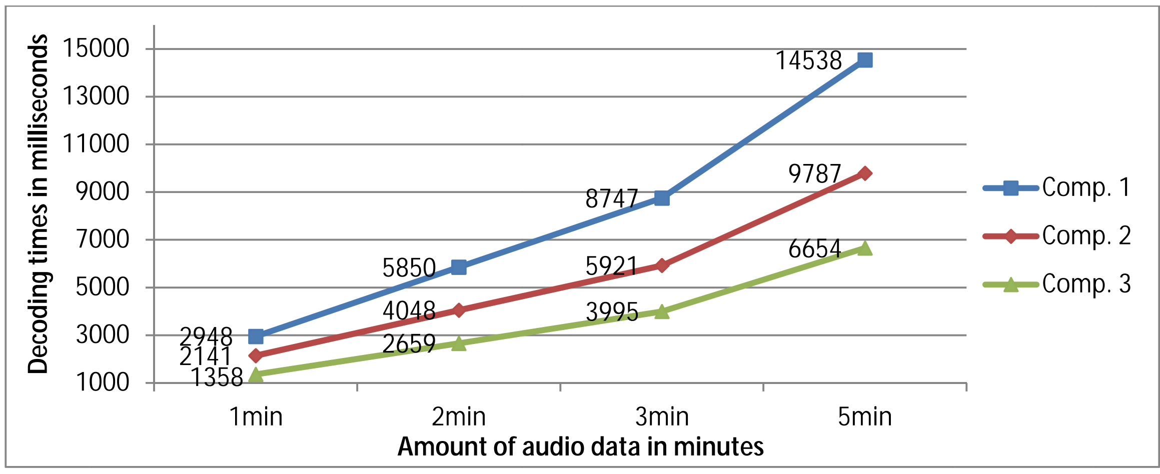


Figure 8 – Web Audio API average decoding times across 3 different computer setups (based on 128kbps MP3-s)

***Appendix H** – raw data of the test determining MP3 decoding times can be found from Appendix H.*

The decoding times' scaling is pretty close to linear regression. Tests showed that there is a constant amount of overhear present which affect the times (as mentioned earlier the decoding times and the overall function execution times are measured separately). It also becomes evident that in case of slower computers (and in the future possibly mobile devices) including lots of audio to the page using MP3 files makes the initial loading time considerably longer.

6.2.4 Decoding times of OGG-audio

The overall differences of decoding times for OGG-files are the following (average decoding times):

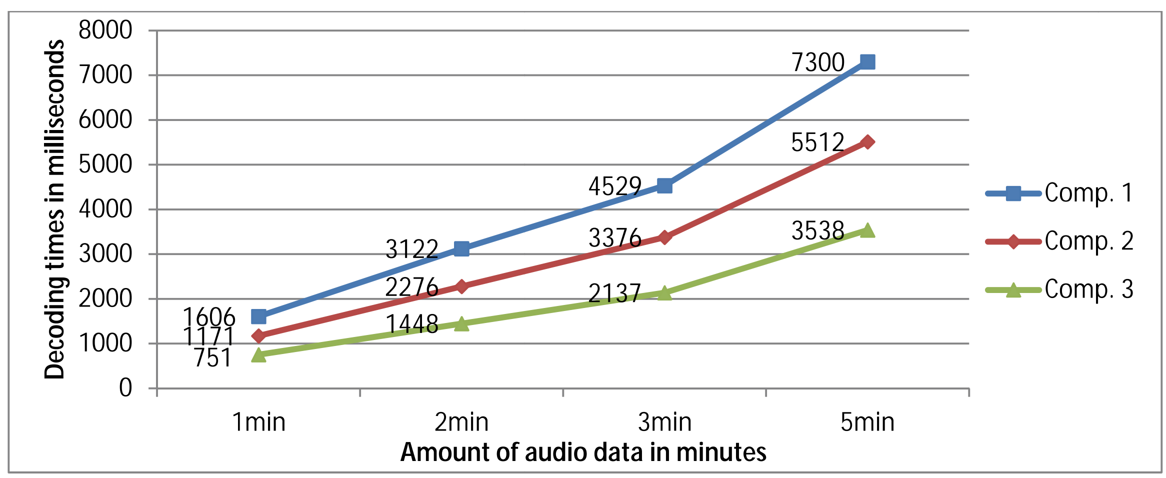


Figure 9 - Web Audio API average decoding times across 3 different computer setups (based on 128kbps OGG-s)

Appendix I *– raw data of the test determining OGG decoding times can be found from Appendix I.*

Decoding of OGG-files takes almost 50% less time that decoding of MP3-s. Also because of some other advantages mentioned later on, this makes the OGG a preferred file type whenever it is supported by the browser.

6.2.5 Decoding times of AAC-audio

The overall timing differences for AAC-audio in MP4 container are the following (average decoding times):

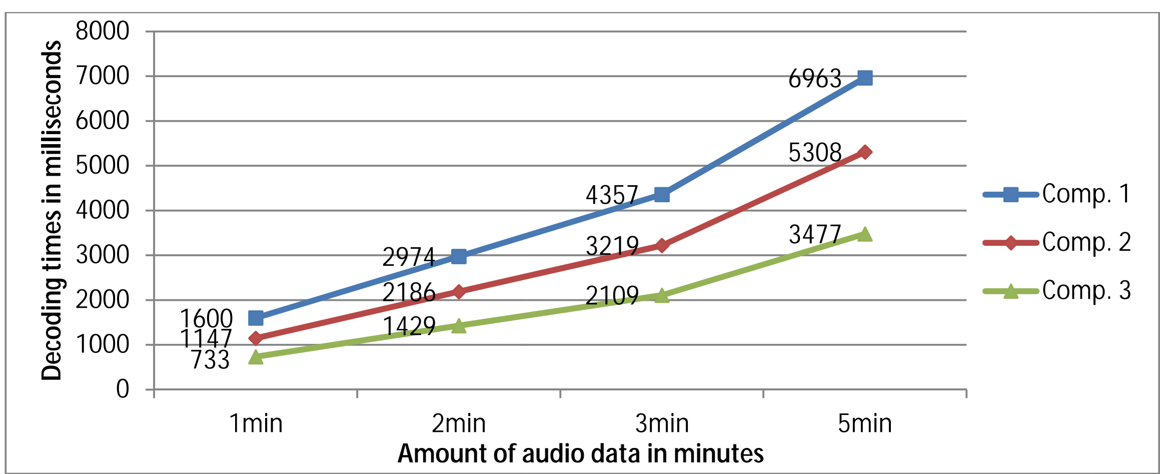


Figure 10 - Web Audio API average decoding times across 3 different computer setups (based on 128kbps AAC-audio)

Appendix J *– raw data of the test determining AAC/MP4 decoding times can be found from Appendix J.*

It is interesting to note that the decoding of AAC-audio (MP4 file) takes the least time out of the three tested file formats. This makes the AAC-audio to be a valid substitution to MP3

files to use with the Web Audio API, since the differences in decoding times are so significant.

6.2.6 Decoding times and different audio qualities

This part of the testing has been made using only MP3 files at different bitrates. In overall different bit-rates does seem to influence the decoding times – decoding 192kbit or 256kbps MP3 file takes longer time than when 128kbps files have been used, but the differences seem to be marginal. The average differences between average decoding times are summarized in the following table:

Audio data lenght	1 minutes	2 minutes	3 minutes	5 minutes
Average difference in ms	107,2ms	136,8ms	112,2ms	138,4ms
Average difference in %	2,843%	4,31%	1,779%	1,343%

Table 6 - Average differences of average decoding times between 128kbps and 256kbps files

Appendix H *– raw data of the tests to determine audio quality iinfluences decoding times can be found from Appendix H.*

The table above represents the average difference of the average decoding times differences between all three computer setups. When it comes to page loading times then 1/10th of a second is rather unnoticeable. Using higher quality audio files as the audio source doesn't change the waiting time as far as the decoding process is concerned. Average decoding time difference between bitrates of 128kbps and 256kbps is about 2,5% across all audio lengths, which is an equivalent of 123ms. When audio files saved at 192kbps is compared to audio of 128kbps, the difference falls somewhere in between. Comparing a 128kbps MP3 with 96kbps MP3 the difference between qualities is already rather small and the tests showed that no specific tendencies can be deducted from the measurements. To get more precise results the number of data points should be increased, but in terms of web game performance, decoding times should not be taken into consideration when one tries to decide whether to use high-quality or low-quality audio.

6.2.7 Decoding times and different number of files

The last step of the test was to determine if the there are differences in the decoding times when five 1-minute long files are decoded versus one 5-minute long audio file. For this test MP3-files with different bitrates have been used.

The following table concludes the results of the test.

Time & Bitrate / Computer	Computer no. 1		Computer no. 2		Computer no. 3	
5x1min / 1x5min 96kbps	14353	13867	9580	9626	6620	6477
5x1min / 1x5min 128kbps	14538	14168	9787	9931	6654	6620
5x1min / 1x5min 192kbps	14568	14379	9993	9902	6667	6690
5x1min / 1x5min 256kbps	14677	14527	9998	10086	6720	6730

Table 7 - Comparison of decoding times (in milliseconds): 5x1minute vs 1x5minute of audio data. Smaller times are better and marked as green.

Appendix H *– raw data of the tests to determine how the number of files used influences decoding times (as long as the total amount of audio remains the same) can be found from Appendix H.*

The average differences from computer 1 to 3 across all bitrates were 295.75, 46.75 and 36 milliseconds respectively. In case of computer no. 2 the decoding time in average was longer when one 5-minute long audio file was decoded. The number of audio file requests affected mostly the computer no. 1 (the slowest one; computer specifications can be found from Table 1), for other machines the timing differences seemed to be pretty random. Since the test was conducted using a web server solution at a local computer the seemingly huge difference of computer no. 1 times compared to other computer setups might be caused by the hard drive seeking times (and by the overall system slowness). By analyzing the structure of the JavaScript code responsible for decoding the audio data (see Code 2) and the audio decoding times from the console window (see Figure 8) it is logical to assume that the decoding process starts as soon as the audio data has been received by the browser, utilizing the processing power of the CPU to its fullest. Since the amount of CPU-usage remains relatively the same throughout the decoding process it can be said that the number of files decoded do not matter until the total amount of audio data remains the same.

6.3 Looping sounds

6.3.1 Introduction and test conditions

As mentioned previously looping sounds is a fairly common technique in games. This raises the question, how well can web deal with sounds which are set to be looped. In both cases (audio-tag and Web Audio API) looping is part of the default playback methods – in case of audio-tag one has to simply include a „loop“ parameter inside the tag (i.e <audio loop><source src="myRandomAudio.mp3"></audio>); to make a sound looping with Web

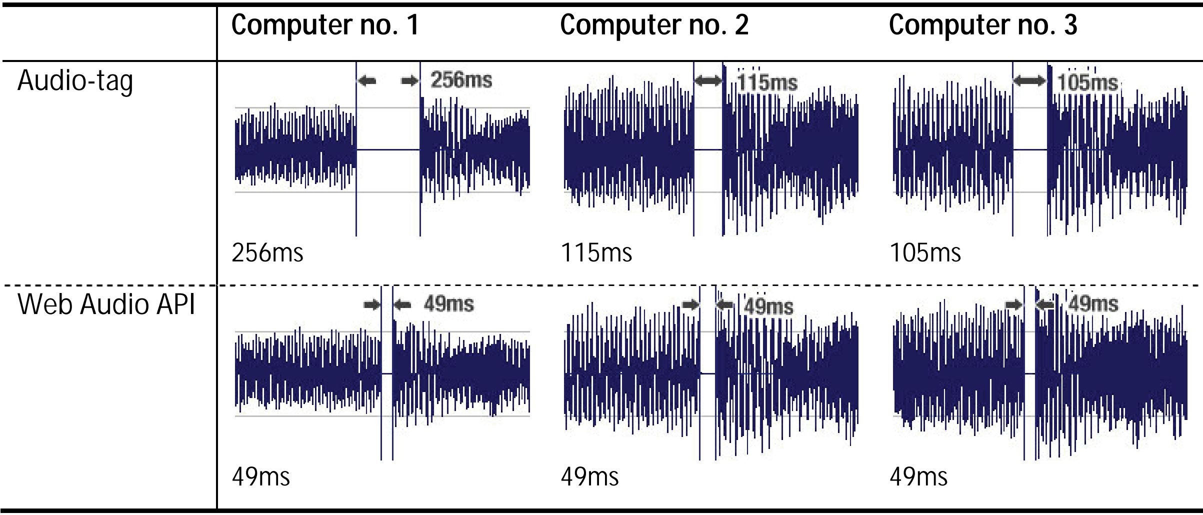
Audio API one has to set the audio source to be loopable (i.e source.loop = true). To approach the question about the performance of the looped audio a test was conducted on three different computer setups (the same setups also used in the previous tests) with three different audio file types (MP3, OGG and AAC/MP4). The main area of interest in measuring the quality of looped audio is to determine the amount of silence present at the breaking point of the loop. Among previously stated test goals this reveals and helps to analyze possible shortcomings in the designs of audio formats when it comes to using them in web.

6.3.2 Ways of measuring

Testing that kind of functionality means that in most cases the conventional developer tools do not have methods to test looping quality and playback delays, therefore the timings were measured manually. The process of testing playback delays included multiple steps: audio playback was initiated, the playback was internally recorded by an audio editing program (many soundcards enable users to record so called "stereo mix" which means that the final audio signal can be internally routed back to the computer and recorded), the recorded audio was later analyzed and the different timings were measured using audio selection tool inside the editing program. Manual measuring of timings was also used in cases where no direct audio recording was involved but only specific time ranges had to be measured.

6.3.3 Looping and MP3-files

The following table concludes how much of a silence is present at the breakpoint:



	Computer no. 1	Computer no. 2	Computer no. 3
Audio-tag	256ms	115ms	105ms
Web Audio API	49ms	49ms	49ms

Table 8 - The lenght of silence in breakpoints across different computer setups when using MP3-s.

Test pages are available on the DVD. (Web Audio API requires a web server to work); pages are also available in the web and can be accessed through http://sisters.ee/kkalmus/masters/audio_master.html

When looping a MP3-audio embedded to the web page with either audio-tag or Web Audio API, one can clearly hear a moment of silence in the breakpoint, when audio file ends and starts again. The small amount of silence leads to an undesirable effect since it is easily noticeable. The gap is more prominent when audio-tag has been used to include audio to the page (especially on slower computers).

The small amount of silence leads to an undesirable effect since it is easily noticeable (especially on slower computers). The gap is more prominent when audio-tag has been used to include audio to the page. There are two main reasons behind the gap when audio-tag is used with MP3 – firstly, as it becomes evident from the table, it is the processing power of the computer. To decode MP3-files it takes time and power [7] and the gap was smaller when a more powerful computer had been used. The second reason lies in the very nature of MP3, which introduces a delay both during encoding and decoding [50, 51].

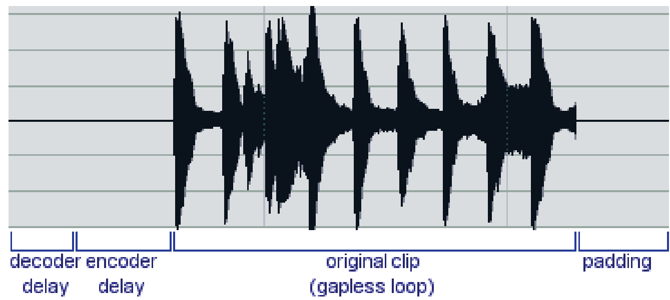


Figure 11 - Encoder & decoder delays and padding cause a "gap" in the loop when the track is played iteratively. Figure taken from http://www.compuphase.com/mp3/mp3loops.htm

By decoding the audio and creating a buffer array in the beginning, the Web Audio API solution can overcome the limitation of the playback of MP3 files at the cost of increasing the initial loading time, but the encoder delay and padding will still be present.

It's not the goal of this paper to get into details of the design of MP3 encoders/decoders, but some relevant points will be covered. By design the encoder includes a number of empty samples in the beginning of the MP3 file, which are necessary during decoding process and removing them may lead to different problems. Decoding process itself also introduces a delay to the playback (silence, which is independent from the silence encoded into the file). Some decoders have been programmed to remove the initial silence during

playback, but seemingly this is not the case with Google Chrome (see Table 6). The delay at the end of the file originates from the design of MP3 – MP3s are divided into frames, each frame consists of 1152 time samples. The MP3 file has to end with a full frame. When there is not enough audio data to fill the last frame (meaning that the number of samples in the song are not an exact multiple of 1152), then the last frame of data is padded with zeroes [50].

How the previous theory applies in reality? When opening a WAV file in a audio editing program (for example Audacity) and saving it as a MP3 file, on can notice an additional silence added at the beginning and at the end of the audio (see Table 7 below) in the freshly created MP3 file. The silence cannot be removed from the file and thus is also present during the audio playback in the web. When the audio file has set to be looped, then the end padding and encoder delay will be the main reasons for the gap between the end and the beginning.

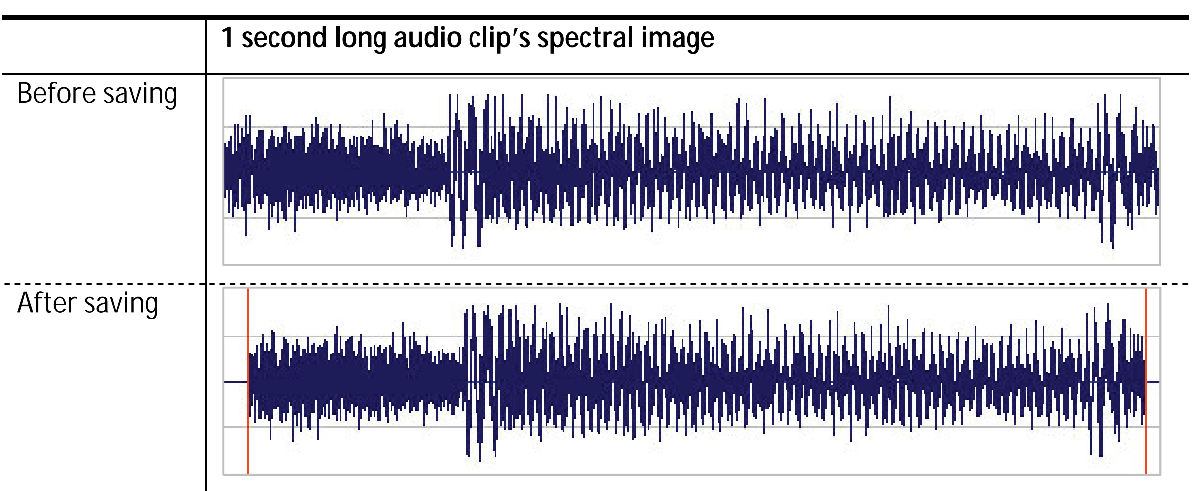
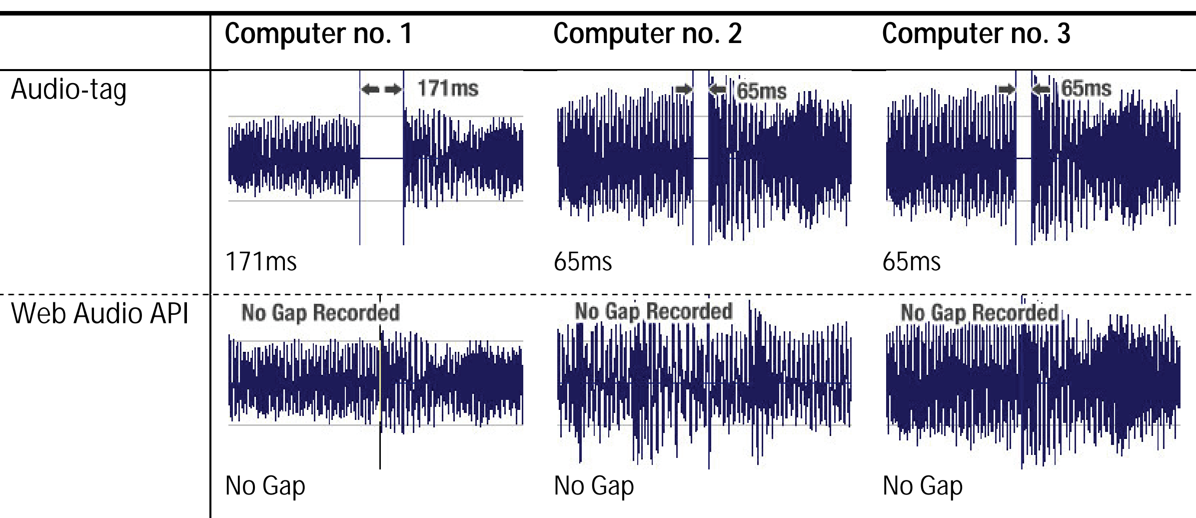
	1 second long audio clip's spectral image
Before saving	
After saving	

Table 9 - Spectral image before and after saving a piece of audio as MP3. On the "After saving" image, the red lines represent the positions of the gaps; the gap in the beginning is 27ms, at the end 17ms.

In Table 8 the amount of silence present when used Web Audio API is 49ms while the sum of the silence shown in the Table 9 is 44ms. This shows that the size of the gap can vary, which is also possible according to the architecture of the MP3 files: the total amount of silence in the beginning is fixed, but the silence at the end can vary depending on how many zeroes will be padded at the end of the last audio frame. The worst case scenario is that the silence at the end is as long as the silence in the beginning, resulting in a total of 54ms of silence.

6.3.4 Looping and OGG-files

OGG-files are completely different and do not have same issues as MP3-files. Saving an audio as an OGG does not introduce any silence or padding inside the file but instead it is saved as is. Since there is no encoder induced delays the gap when using audio-tag is smaller. Some of the improvements can also probably put down to how the decoder works. In case of Web Audio API the looping takes place seamlessly regardless of the test computer's processing capabilities. The following table shows the amount of silence in case of OGG files across all three test setups:

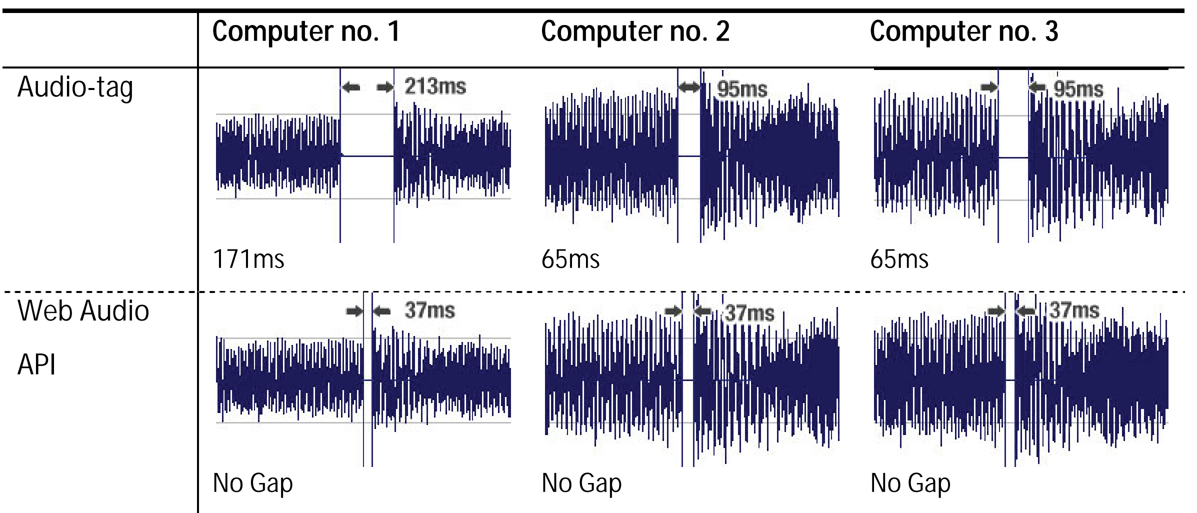


	Computer no. 1	Computer no. 2	Computer no. 3
Audio-tag	171ms	65ms	65ms
Web Audio API	No Gap	No Gap	No Gap

Table 10 - The lenght of silence in breakpoints across different computer setups when using OGG-s.

6.3.5 Looping and MP4-files

The following table shows the amount of silence during looping when AAC audio was used:



	Computer no. 1	Computer no. 2	Computer no. 3
Audio-tag	171ms	65ms	65ms
Web Audio API	No Gap	No Gap	No Gap

Table 11 - The lenght of silence in breakpoints across different computer setups when using AAC-audio (in MP4 container).

Using AAC audio (in MP4 container) resulted in a smaller gap compared to MP3 files. When looking at the browser compatibility with different audio file formats (Appendix A) then one

could see that the browsers which support MP3 also support MP4 format, making MP4 a good substitute for MP3. MP4 file doesn't have encoder induced delay in the beginning but it has the padding at the end. The amount of padding can also vary depending on the amount of audio data added to the last frame of the audio file.

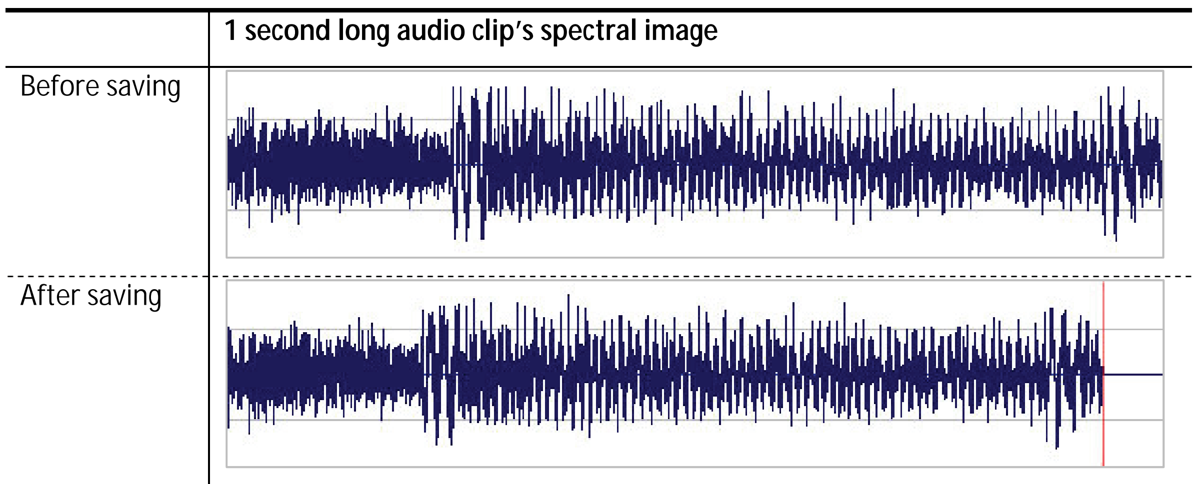
	1 second long audio clip's spectral image
Before saving	
After saving	

Table 12 - Spectral image before and after saving a piece of audio as MP4. On the "After saving" image, the red line represent the positions of the end padding; the gap at the end is 44ms.

When looking at the gap times in the previous tables, the data shows that the audible gap is also there because of how the browser handles the audio files – a certain amount of the silence will be introduced when the playback position is changed back to the beginning while looping. When OGG-file was used, the gap on a fastest computer was 65ms, with MP4 the gap was 95ms (which includes 23ms of encoded silence), and with MP3 the gap was 105ms (which includes 55ms of encoded audio). This shows that the performance of audio decoders and the amount of decoder induced silence is different. From the end point of view the OGG performs the best (across all 3 computers the looping gap was the smallest); the MP3 decoder seems to be most efficient when it comes to decoder induced silence but unfortunately the overall length of the silence is still the longest due to the silence encoded into the file, which makes MP4 to be a better option.

6.4 Overcoming limitations of looping

6.4.1 Introduction, test conditions and measuring

Looking at the previous data raises a question, whether there are any possible solutions to overcome the limitations? When using audio-tag the decoder induced delay is always present and especially prominent on slower computers. Web Audio API decodes the audio

and therefore doesn't suffer from decoder delays, but the silence inside the audio file itself will come along during the creation of buffer array.

One of the easiest solutions is to use fading whether the audio itself has been faded in and out on the audio file or fading has been created programmatically. Based on my own experience this technique has been used extensively even In high quality PC-games – one of the latest example is Hearthstone (a strategy card game by Blizzard) where audio fades out at some point and comes back later. One of the possibilities is to set the breakpoint to be in a position, where a moment of silence seems to be part of the audio and doesn't break the perceived consistency of the audio piece.

When none of these previously stated techniques can be used due to the nature of the audio design, and when audio-tag has been used, then theoretically another possibility is to loop the audio "manually" by change the playback position at a right time. In a test case created to investigate the possibility further, the same background audio was used as for the other looping tests but the amount of silence in the beginning and at the end of the file was increased up to 1 second (with the MP3 and MP4 files the end padding makes it difficult to get the timing exactly right); then an event listener was added to the audio element which changed the playback position to 1 second (where the actual audio data begins) whenever the playback position reached the end of the playable audio data (1 second from the end of the file). Another fixed amount of silence, which is set to be longer than the encoder delay, could also be a possibility.

Using this sort of manual looping should in theory eliminate the encoder delay and padding from the playback. The downside of this approach is that there will be a fixed amount of delay in the beginning when audio file's playback starts, but this could be dealt with for example including a small part of audio to the beginning of the file which will be left out from the loop later on. Theoretically, one could set the playback start position to the point where the silence in the beginning of the file ends, but this increases the complexity of the code – in order for this to work another method has to be included, which checks whether enough audio data has been buffered for the playback position to be changed and after that starts playback, otherwise when the command has been given to set the playback position to somewhere else in the file, that position may not yet exist, since not enough audio data has been fetched. In other words it is impossible to set the playback position to 1 second when only 500ms worth of audio data has been downloaded.

The following snippet of code is responsible for changing the playback time back to beginning with some data logging methods to give feedback about the timings. The code outputs two values into browser's console window – the time point where the script is set to make the break and when the break has actually been made.

```
currentAudio.addEventListener("timeupdate", function() {
      var duration = currentAudio.duration;
      var change = duration - 1.4;
      var position = currentAudio.currentTime;

      if ( change <= position) {
            console.log (ch);
            console.log (position);
            currentAudio.currentTime = 1;
      }
});
```

Code 3 – JavaScript event listener responsible for custom looping functionality with logging.

The test revealed some of the bottlenecks also mentioned earlier in this paper – the performance of JavaScript and its timing accuracy. The code can be tuned to compensate possible delays induced by JavaScript. In theory the audio file should have 1 second of silence at the end of the file (end padding still remains to be a problem), so the script should change the current playback time to the beginning when one second is remaining from the end of the file. Checking if exactly one second has left to be played is impossible (the conditional equation of "if playback time is equal to breaking point time, then make the break" did not work), therefore currently the event listener checks whether the current playback position has gone past the potential breaking point. To compensate the possible reaction delay the breaking point in the code example has been set to 1.4 seconds from the end of the file. The test was conducted under multiple compensational values from 1.2 to 1.4 seconds.

When using Web Audio API the problem related to the decoder is eliminated since the files have been decoded and buffer array has been used, but the gaps and silence already present in the source file remains, which is a problem when MP3- or MP4-files have been used. Using OGG-file as the source of the audio whenever possible is probably the best option since the OGG-files does not include any encoder induced silence, but not all platforms might support OGG. When the same approach has been taken (to include 1 second of silence to the beginning and at the end) then Web Audio API has a specific

attributes to determine the start and the end of the loop which can be used for this purpose.

```
var Play = function(bufferedData, looping, sourceGain) {
        this.source = context.createBufferSource();
        this.source.buffer = bufferedData;
        this.source.loop = looping;
        var duration = this.source.buffer.duration;
        this.source.loopStart = 1;
        this.source.loopEnd = duration - 1;
        this.source.gain.value = sourceGain;
        this.source.connect(gainNode);
        gainNode.gain.value = volumeVal / 100;
        gainNode.connect(context.destination);
        this.source.start(0);
    };
```

Code 4 - Function defining the flow of the signal and other playback parameters. The loop parameters are shown in red.

The implementation of this is fairly easy – before the playback the `loopStart` attribute is set to 1 and the `loopEnd` attribute is set dynamically to be one second from the end based on the length of the buffer. The previous snippet of the code (Code 4) represents the function responsible for connecting audio source to the destination and among other also defines looping parameters.

6.4.2 Test results (audio-tag)

This technique enables one to partly overcome the problem with the silence, but it greatly increases the complexity of the code and is not really reliable solution for the following reasons. Since browsers are different and the file support is different, then also the OGG file has to be presented inside the audio-tag. When OGG file has been used instead of MP3/MP4, the problem with the encoded silence does not exist, meaning that an extra amount of silence has to be put into the OGG files manually or an extra subroutine has to be programmed into the code, which keeps track of the file type currently used and changes the playback and looping conditions based on this information.

The timings data showed that there was no consistency in the timings when the actual break was made. Interestingly enough, by comparing the reaction times shown in Table 13, it can be seen that the fluctuation of timings between computer no. 1 (slowest) and computer no. 3 (fastest) are about the same. From the computer no 1 to 3 the standard deviations are 64,1852; 39,6832 and 76,4229 milliseconds respectively, which show that even on a powerful computer JavaScript is not reliable when it comes to perfectly timed executions.

Computer no. 1	*Computer no. 2*	*Computer no. 3*
240,02	141,33	116,68
106,31	220,99	216,99
194,15	159,94	236,21
73,87	207,38	140,33
61,87	188,77	9,01
154,15	171,55	66,06
178,15	149,33	226,99
153,14	188,77	116,11
221,58	83,28	97,89
73,87	141,33	206,38

Table 13 - Reaction times of JavaScript when audio has been set to be looped through manually created subroutine throughout the test computers. All given values are in milliseconds.

Appendix G *– raw data of the tests to determine the performance of the manual audio looping across the test systems can be found from Appendix G.*

This test shows clearly how unreliable JavaScript can be when used for such purposes. Basically the code is an infinite loop, which is set to check the current playback time and execute one conditional block when condition is true (see Code 3) which sole purpose is to set the playback position back to the beginning of file. The implementation already tries to cope with the JavaScript induced reaction delay, but since the reaction timings are so different sometimes a small section at the end of the loop will be cut off, creating unnaturally sounding breakpoint (which still seems to sound better than a brief moment of silence).

It is important to note that this type of looping implementation seems to work only when MP3 audio has been used. The "*timeupdate*" event will be not fired (for a detailed overview of the implementation see Code 3) when OGG or MP4 has been used as audio source. The broken functionality might be a result of a bug in the Google Chrome's implementation of audio-tag.

6.4.3 Test results (Web Audio API)

With some minor modifications in the parameters of the start-method and some fine tuning, the playback worked smoothly. Because the last audio frames of MP3- and MP4-

files have some padding, then using such a simplistic code to determine the end of the loop may not give the best results when one unified function has to handle multiple audio files (this does not apply to OGG files, since the encoder does not alter the original source data in sense of encoded silence). When the number of used audio clips is low, one of the options is to specify the loop regions manually and save them alongside the buffer data. When one decides to take the approach of manual fine tuning, the technique of adding the extra silence in the beginning and at the end becomes redundant – when using manually selected loop selections it is also possible to concatenate the different audio files into single file and define exactly when specific audio clips start and end (as showed in chapter 8 the decoding time remains the same and overall loading time might benefit from reduced request time). This makes it possible to have perfect loops without any delays and unwanted silence.

Manual looping *– An examples has been included to the DVD, where an audio playback has been manually set to be looped at specificed times (both for audio-tag and Web Audio API). Example have also made available in the web and are accessible from http://www.webgamesaudio.com/masters/*

7. Reducing the usage of system resources

In chapter 3 some of the ways were covered which have been used in games to reduce the usage of system resources by audio. In this chapter the list will be gone through and analyzed the suitability of the techniques in web environment in the light of the previous tests.

One of the most viable options mentioned in chapter 3 is to use mono audio. This reduces the size of the files 2 times and subsequently reduces the data network usage, also when Web Audio API has been used, using mono files their decoding time will be reduced to half. This is a simple way to improve the overall performance significantly when usage of stereo audio is not crucial.

Concatenation (loading multiple audio parts as one file) is also one of the techniques mentioned. By using concatenation the web game could theoretically benefit from the reduced overall load times, but technically it could work out only when Web Audio API has been used to deliver audio. Audio-tag lack reliable default support for playing only parts of the file and therefore requires a custom functions to make it work, which may not be the most efficient way of implementing audio.

One of the most interesting techniques mentioned is the way of saving file twice the original speed and then playing it back on half of the speed, which should result in a smaller source audio file. It won't be looked into how game engines handle this technique, but browsers do not seem to do very well. There are multiple ways of shortening the length of an audio file – one of the ways is to basically save the file at twice the playback speed which also results in a change of pitch, another way is to use time-stretch technique, which leaves the pitch intact but changes the speed [52]. When changing the playback speed in case of audio-tag, the browser handles the playback speed by the standards of time-stretch. When listening to the examples it becomes evident that audio-tag is not capable of incorporating this technique in purpose of saving bandwith and reducing loading time. There are too many artifacts present in the audio for this to be an option.

Changing playback speed *– Examples has been included to the DVD, where an audio playback speed has been changed (both for audio-tag and Web Audio API). Examples have also made available in the web and are accessible from http://www.webgamesaudio.com/masters/*

Web Audio API handles changes in playback speed differently and does not apply any time-streching to the buffer when playback speed is changed – when playback speed is changed it also affects the pitch (it is similar to the effect achieved when changing the playback speed on a gramophone or on a turntable). The test showed that when a source audio was saved at twice the playback speed, and then played back on half of the speed Web Audio API did a relatively good job. The viability of this technique in web remains under a question since there are other methods to get the same audio quality with the same changes in file size (i.e downsampling) – when audio has been speed up and saved as 128kbps 44100 Hz audio file, the perceived quality is the same as audio saved at normal speed but at 22050hz. Using different playback speeds for audio effects could be useful but using the technique for background audio is not worth the hassle.

8. Conclusive analysis

In this chapter a wider look to the results will be taken and analyzed; pros and cons of different usage methods will be brought out in a process. Also it will be looked into, how different aspects might influence the user experience in real life conditions and cases.

Page and resource loading times have always been something developers have had to reckon with. From the tests related to download timings can be seen that the number of requests made to the server has an impact to the download timings. Much as reducing the number of audio files included to the web game can improve the performance, it cannot be suggested that one could only benefit from it. The balance between the number of audio files included and the length of the audio files have to be found based on the project at hand – as also discussed previously, concatenating audio files together can lead to more complex codebase since extra routines have to be programmed into the system to handle the audio playback. When comparing audio-tag and Web Audio API in this matter, then Web Audio API is clearly more flexible as it offers a better variety of methods for working with audio – using concatenated audio increases greatly the implementation complexity with audio-tag, but Web Audio API have the necessary functionality to make using concatenated audio a solid option. In some cases this could also result in a more efficient data network usage.

When the number of audio assets in a game starts to become a problem, then to optimize the performance it might be a good idea to load the necessary audio assets only when required (i.e when game is initially loaded, only audio used during menu screen will be downloaded). This approach is beneficial in both implementation cases (audio-tag and Web Audio API) because when game has been opened but the player leaves the page without playing it further, no bandwidth will be wasted on loading unnecessary assets. With Web Audio API a certain amount of decoding time will be added to the loading times, which means that loading and decoding huge amount of audio during initial loading of the page increases waiting time before user could interact with the page. Utilizing the possibilities of asynchronous data retrieval it would be possible to find a balance between loading times and number of files loaded at the time. In addition to the audio assets, games also include number of graphic assets and script files which also have to be loaded and therefore it is

especially important to balance the performance – minimizing the number of HTTP request is in the top of the list of techniques which helps to maximize web page display speed [49]. In the field of optimizing the loading of audio files, audio-tag seems to surpass Web Audio API because of the optimization techniques built into the browsers – browser may decide not to download a piece of audio when it is not needed and in case of large audio files the network optimization happens automatically by default without the necessity of implementing custom functions to deal with it. With Web Audio API, the optimization techniques have to be designed and implemented manually by developer.

Browsers have gone through a tremendous development over the past years. With the rise of HTML5 a set on new possibilities has opened up to the developers and the cross-browser compatibility has also improved. The same multimedia functionality which once required plug-ins is now supported by browsers by default. The new possibilities are welcome, but using them to create multimedia solutions does not come without a hassle. Using Adobe Flash for creating a multimedia application means that the experience is about the same across different browsers and developers do not have to worry about the browser support or any special cases. This is not the case when using native web technologies – the browser support is different and often dictated by patent issues or corporate policies. Including an audio file to a page means that multiple audio files have to be made available to offer the same experience for the users' of different browsers. Fortunately, when audio-tag has been used, browsers will choose appropriate file type automatically – from the files listed between audio-tag a browser will use an audio file it supports, which helps out a lot. Since the Web Audio API have established more ground than Audio Data API (once developed by Mozilla) and possibly becomes more widely supported over time means that multiple audio files have to be provided (similarly when audio-tag has been used). In case of Web Audio API there is a slight difference – Web Audio API does not have such an automatic file selection method as audio-tag has and the file type selection has to be specifically programmed by the developer. This means that instead of working on the final product itself, one has to spend time to cover the extra cases and to deal with the browsers' characteristics, making the implementation probably more time consuming than it would with Adobe Flash.

Another set of issues come along with the characteristics of different types of audio files. Since audio encoding technologies are different, audio files behave differently (as seen in

the test with looped audio). It is not that much of an issue when simply a song has been included to a page for visitors to be listened, but when timings and performance are crucial (as they are for games) then delivering a standardized user experience is not that easy. Creating specific functions to cope with the problems induced by the audio encodings (for example silence present in the beginning of MP3-files and padding at the end of MP3/MP4-files) often create an unnecessary overhead and complexity. In Web Audio API it is theoretically possible to remove the silence from buffer array, but that requires manipulating with the arrays directly, which is CPU-heavy process and may not be worth it; there are better ways to get the wanted result, like specifying playback regions. For audio-tag the audio is implemented "as is" and cannot be altered directly.

When analyzing the applicability of different audio file types in the light of the test results, then most suitable file type out of the three seems to be OGG. Working with it includes the least amount of hassle – the file will be saved as is (encoder doesn't alter the underlying audio signal by adding unwanted silence) without any unwanted side effects, the decoding process when used with the Web Audio API is rather quick (MP4 was decoded slightly faster) and the bitrate-to-quality ratio is good. Unfortunately not all browsers support it, meaning that still another file has to be included alongside OGG. MP3 is a widely used compression format, but based on the results I would suggest to discard MP3-files completely when adding audio to web games and use AAC/MP4-files instead. With its encoder induced delay in the beginning and padding at the end MP3 file brings along additional complications. MP4 files also have end padding which requires some effort to get around of, but the audio quality is better than MP3's [48] and the WebAudipAPI's decoding process will take about twice as less time than for MP3-s. When reading through different books and looking at various audio-tag usage examples, one could notice a pattern emerging – MP3 file has always the first source file included to the audio-tag. By changing the audio source order and making the OGG file to be the first in the list guarantees, that when browser support includes OGG files it will be used (browser picks the first audio source it can deal with [27, 43]). This makes sure that the benefits of the OGG-files can be utilized of all possible cases.

When coming to the world of sound effects the previous tests also cover some of aspects related to implementing sound effects. With audio-tag I would suggest that the easiest way of including sound effects to the game is to have one tag-block for each effect in use since it

is simpler than concatenating them and then trying to program extra functionality to deal with the vast number of playback regions for one audio file. The downside of this is the increased number of request which will be made to the server. It has been estimated that each additional object will add extra 40ms of latency to the load time of the page; the latency is also dependent on the location of web servers and the number of "hops" data has to take to get from the source to the destination [49]. Analyzing the absolutely necessary number of included audio effects and its effects on the page performance should be part of the preliminary game design process when audio-tag has been used. When implementing sound effects using audio-tag, then effects' playback latency is something that is heavily dependent on the processing power of the computer, meaning that developers have to accept that the audio playback timings and thus user experience can be varied across different systems.

Web Audio API offers better possibilities to the implementation of sound effects, since it have methods for determining playback regions, making a concatenation of sound effects to be a convenient solution (whenever applicable) – one file means small request delay and better network efficiency. Also the playback timing is more consistent. Tests with audio looping showed that even slower computers can effectively work with audio buffer assuring an identical user experience across systems.

It has to be remembered that the tests were conducted only on Google Chrome and the results may vary across different browser for better or for worse, but the suggestions and proposed techniques should still remain valid.

9. Conclusion

From those two implementations covered in this paper it is clear that audio-tag is clearly simpler and implementing audio using audio-tag is easy. The list of audio events and properties seems to be sufficient to satisfy the needs of a less demanding project. As upsides it is worth mentioning the facts that audio playback can be triggered as soon as enough of the audio data has been buffered, a suitable audio file will be selected by the browser automatically (assuming that at least one of the supported file types has been provided within the audio-tag) and all major browsers support it one way or another. It should be kept in mind that as the games get sonically more demanding audio-tag may not be up to the task and it does not contest the possibilities of Adobe Flash, which still seems to be the main way for delivering multimedia content over the Internet. When more manipulation possibilities are needed but not necessarily with a precise playback timings, then with some work one could use audio element as the input source to the Web Audio API to apply additional processing to the audio (an example of it has been provided on the following page: http://updates.html5rocks.com/2012/02/HTML5-audio-and-the-Web-Audio-API-are-BFFs).

Web Audio API is still a rather young compared to some other web technologies and therefore is prone to changes (hopefully to the better). Public W3 Audio Working Group's discussion archive shows a notable number of letters exchanged on various audio related topics (http://lists.w3.org/Archives/Public/public-audio/) which encourages believing in the sustainability of Web Audio API and in future improvements. A quick search around the Internet reveals that there are also a number of Web Audio API-related bugs out there, which influence the usage of that technology to some extent (for example a list of open bugs can be found from https://bugzilla.mozilla.org/buglist.cgi?component=Web%20Audio&product=Core&bug_status=__open__). Also the support is still rather limited and it may take some time before all major browser vendors decide to make the functionality available to their browsers; for example the latest Internet Explorer version (IE11, which has not been out very long at the time of this writing) does not support it, even though the requests to include it to this version was made by the community some time ago (http://connect.microsoft.com/IE/feedback/details/799529/web-audio-api-support).

Implementing Web Audio API does require some finesse – trying to get a hold on the technology can be quite time consuming in the beginning, but it does pack a rather impressive list of methods to work with from dynamic audio generation to filters/effects to audio visualization. Some of the cons of Web Audio API are that the mechanism to use a supported audio file has to be programmed manually and is not done automatically by the browser which means extra time spent not working with the solution itself. On the other hand Web Audio API works wonderfully when the audio playback has to be timed rather precisely. Still, it has to be remembered that the audio playback initiation relies on JavaScript and if most of the system CPU has been consumed by dealing with game logic the playback may be delayed because of the performance issues of JavaScript. All things considered, after the initial waiting time to get the audio data decoded the implementation seems to do pretty much what it was set out to do and could be a solid substitute for Adobe Flash. When recorded audio will be used one still has to keep track on the amount of used audio since decoding audio taxes hardware quite much and some cases can introduce long waiting times. Decoding necessary audio assets into audio buffer at different times over the lifetime of a gaming session could be a solution.

To conclude the results in a final compact form, a set of recommendations have been generated, which can be useful when implementing audio to web based games.

- Analyze the technical requirements for audio – using audio-tag is easier but Web Audio API offers more possibilities.
- Consider using not too many audio assets, as it can hurt loading times.
- When choosing audio asset's quality, check the audio's spectral data to make the most optimal decision.
- When the number of assets cannot be limited, don't try to load all audio at once (especially when using Web Audio API due to the decoding time) – take advantage of asynchronous loading and load them when necessary or on background.
- Consider using OGG and AAC/MP4 audio instead of MP3 whenever possible.
- Looping background audio with audio-tag works best when the piece of music has set to be faded in/out or composed in a way which masks the break point.
- JavaScript is not as optimized as ActionScript, therefore the performance and execution timings can vary even on more powerful devices – take into account that game might be played on different devices with various processing capabilities.

10. Future works

This paper covers the topic of sound effects mostly on the theoretical level and analyses their implementation possibilities-limitations based on the results of mostly background audio related tests. The tests revealed some potential problematic aspects in the performance of sound effects in web – playback delays, increase in file request times when increasing the number of audio files, processing speeds and performance of JavaScript to conclude the most prominent limitations. One of the possible future works is to create a web game which incorporates a decent number of sound effects and conduct a qualitative study to analyze the perceived performance of audio implementation methods in case of audio effects in web games.

Since the tests were made only using Google Chrome, differences in performance across browsers could and should be looked into. Based on my personal experience I would say that even when certain elements or attributes are supported among all browsers, often there are still some differences how one browser or another deals with elements or executes specific snippets of code. With the increasing support of Web Audio API, browser specific quirks and differences in handling Web Audio API's methods could be analyzed in the future (and a similar analysis could be made for audio-tag, though the latter seems to work rather uniformly across browsers). Making a game to be played only on one platform (let say a Google Chrome) and thus tying users to one browser or platform is not beneficial in a long run; instead one should cover as many browsers and platforms as possible and try to offer the same experience across them – to do that the differences in handling the audio across browsers have to be looked into and dealt with accordingly.

The tests covered in this paper included only clean audio implementations (meaning that all other aspects – codebase and graphic assets – were kept to a minimum) and in order to find out how audio implementation methods perform under real conditions, a more complex test environment is necessary which also includes a decent amount of graphic elements and where system resources have to be shared among audio, visuals and game logic. This would reveal how audio implementation methods perform under stress.

Using web as an implementation environment also gives the benefit of cross-device compatibility which brings us to the world of mobile devices. How audio implementations

using audio-tag or Web Audio API perform on mobile devices is another topic that could be looked into in the future. Mobiles are getting more powerful in terms of processing power but they are still not as powerful as desktop computers or laptops. As seen from the tests, the processing power of a device can in many cases directly influence either user experience or game performance at large and mobile devices are no different. Also the possibility of overheating and the battery consumption are some of the elements which could theoretically act as additional limitations and could be looked into.

References

1. Collins, Caren (2008) *Game sound – An Introduction to the history, theory, and practice of video game music and sound design*. The MIT Press
2. Video Game Console Library, http://www.videogameconsolelibrary.com
3. Baldwin, Neil (2009-2010) *NTRQ: NES Tracker*
4. NES Specifications. http://nocash.emubase.de/everynes.htm
5. Marks, Aaron (2009) *The complete guide to game audio. For composers, musicians, sound designers, and game developers. Second edition*. Focal Press
6. Garcia, Juan M. (2006) *From heartland values to killing prostitutes: An overview of sound in the video game Grand Theft Auto Liberty City Stories*, Audio Mostly 2006, Piteå, Sweden, (October 11—12, 2006).
7. Stevens, Richard; Raybould, Dave (2011) *The game audio tutorial. A practical guide to sound and music for interactive games.* Elsevier Inc.
8. Perron, Bernard; Wolf, Mark J. P. (2009) *The Video Game Theory Reader 2*.
9. Brown, Emily; Cairns, Paul (2008) *A Grounded Investigation of Game Immersion*
10. Ermi, Laura; Mäyrä, Frans (2005) *Fundamental components of the gameplay experience: Analysing immersion*, Changing Views – Worlds in Play, Toronto, (June 16—20, 2005).
11. Sweetser, Penelope; Wyeth, Peta (2005) *GameFlow: A Model for Evaluating Player Enjoyment in Games*
12. Sanders, Timothy; Cairns, Paul (2010) *Time perception, immersion and music in videogames.*
13. Grimshaw, Mark; Lindley, Craig A.; Nacke, Lennart (2008) *Sound and Immersion in the First-Person Shooter: Mixed Measurement of the Player's Sonic Experience.*
14. Cheng, Kevin; Cairns, Paul A. (2005), *Behaviour, Realism and Immersion in Games*. CHI '05 Extended Abstracts on Human Factors in Computing Systems Pages 1272-1275
15. Pivec, Paul; Pivec, Maja (2009) *Immersed, but How? That Is the Question*
16. Jørgensen, Kristine (2006) *On the Functional Aspects of Computer Game Audio*
17. Jennett, Charlene; Cox, Anna L.; Cairns, Paul; Dhoparee, Samira; Epps, Andrew; Tijs, Tim; Walton, Alison (2008) *Measuring and Defining the Experience of Immersion in*

Games. Volume 66 Issue 9, September, 2008, International Journal of Human-Computer Studies

18. Panksepp, Jaak; Bernatzky, Günther (2001) *Emotional sounds and the brain: the neuro-affective foundations of musical appreciation*. Behavioural Processes 60 (2002) 133_/155
19. Karjalainen, Matti (1999) *Immersion and content – a framework for audio research.*
20. Cunningham, Stuart; Grout, Vic; Hebblewhite, Richard (2006) *Computer Game Audio: The Unappreciated Scholar of the Half-Life Generation.* Proceedings of the Audio Mosty Conference a Conference on Sound in Games
21. Chion, Michel (1994) *Audio-Vision: Sound on Screen*. Columbia University Press
22. Ekman, Inger (2005) *Meaningful Noise: Understanding Sound Effects in Computer Games*
23. Kaye, Tomasz (2013) *ibb & obb. Sound design post-mortem.* http://www.gamasutra.com/blogs/TomaszKaye/20131028/202776/ibb__obb_Sound_design_postmortem.php
24. W3C Working Draft. *Web Audio API*. http://www.w3.org/TR/webaudio/
25. Flanagan, David (2011) *JavaScript: The Definitive Guide (6th Edition).* O'Reilly Media
26. McFarland, David S. (2012) *JavaScript & jQuery: The Missing Manual, Second Edition*. O'Reilly Media
27. Powers, Shelley (2011) *HTML5 Media*. O'Reilly Media
28. Levitin, Daniel J.; MacLean, Karon; Mathews, Max; Chu, Lonny (1999) *The Perception of Cross-Modal Simultaneity.*
29. Adobe Community Help, *Flash Professional – Using Sounds in Flash.* http://help.adobe.com/en_US/flash/cs/using/WSd60f23110762d6b883b18f10cb1fe1af6-7ce8a.html
30. Game{closure} DevKit Docs. *Creating Audio Assets.* http://doc.gameclosure.com/guide/audio-assets.html#stereo-and-mono-files
31. Harris, Andy (2013) *HTML5 Game Development for Dummies*. John Whiley & Sons, Inc.
32. Mozilla developer network. *Media formats supported by the HTML audio and video elements.* https://developer.mozilla.org/en-US/docs/HTML/Supported_media_formats

33. Fitzgerald, Chriz (2013) *Sample Rate Explained*. http://learningcenter.berklee.edu/blog/sample-rate-explained Berklee College of Music – Learning Center.
34. Juul, Jesper (2012) *A Casual Revolution: Reinventing Video Games and Their Players*. The MIT Press
35. Facebook Application Center. https://www.facebook.com/appcenter/category/games/'
36. Gay, Jonathan. *The history of Flash.* http://www.adobe.com/macromedia/events/john_gay/page04.html
37. Grover, Chris (2012) *Flash CS6: The Missing Manual*. O'Reilly Media
38. Braunstein, Roger (2010) *ActionScript 3.0 Bible*. Wiley Publishing
39. W3Techs - Web Technology Surveys. *Usage of Flash for websites.* http://w3techs.com/technologies/details/cp-flash/all/all
40. Adobe & HTML. http://html.adobe.com/mission/
41. W3C Working Draft (29 October 2013). *HTML5 5.1 – A vocabulary and associated APIs for HTML and XHTML.* http://www.w3.org/TR/html51/embedded-content-0.html#the-audio-element
42. W3C Schools. *HTML Audio and Video DOM Reference.* http://www.w3schools.com/tags/ref_av_dom.asp
43. Mozilla developer network. *Using HTML5 audio and video.* https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Using_HTML5_audio_and_video
44. Smus, Boris (2013) *Web Audio API.* O'Reilly Media
45. Mozilla developer network. *Introducing the Audio API extension.* https://developer.mozilla.org/en-US/docs/Introducing_the_Audio_API_Extension
46. Hacker, Scot (2000) *MP3: The definitive guide.* O'Reilly Media
47. Vorbis Homepage. http://www.vorbis.com/faq/
48. Brandenburg, Karlheinz (1999) *MP3 and AAC explained*. AES 17th International Conference on HighQuality Audio Coding
49. King, Andrew B. (2008) *Website Optimization*. O'Reilly Media
50. Taylor Mark (2000) *Lame Technical FAQ*. http://lame.sourceforge.net/tech-FAQ.txt

51. Lutzky, Manfred; Schuller, Gerald; Gayer, Marc; Krämer, Ulrich; Wabnik, Stefan (2004) *A guideline to audio codec delay.* Audio Engineering Society, Convention Paper 6062
52. Bernsee, Stephan (1999) *Time Stretching And Pitch Shifting of Audio Signals – An Overview*. http://www.dspdimension.com/admin/time-pitch-overview/

List of figures, tables and code examples

Figure 1 - The triangle of compromise in game audio by Stevens & Raybould 14

Figure 2 - Triangle of compromise for web conditions 15

Figure 3 - The basics of latency 17

Figure 4 - Differences in frequency data when different sampling frequencies have been used. Colors represent the volume level of the sound on different frequencies (cyan is lowest, light orange is highest) 19

Figure 5 - The class structure of Adobe Flash sound system 22

Code 1 - The minimum amount of code necessary for adding audio to page. Source: W3Schools 24

Figure 6 - AudioContext with different AudioNodes. Source: W3C [24] 26

Table 1 - Hardware specifications of the computers used for testing............................... 34

Figure 7 - Download times of 1 minute long MP3 file in case of different download speeds. Red arrows appoint to the approximate position, where mobile connections reside on the speed graph. (based on http://www.techspot.com/guides/272-everything-about-4g/). . 35

Table 2 - Average loading times of 1 minute long MP3-file in case of AUDIO-tag and Web Audio API 38

Table 3 - Average loading times in milliseconds of 2 (2x1 minute), 3 (3x1 minute) and 5 (5x1 minute) minutes of audio data in case of AUDIO-tag and Web Audio API......................... 38

Table 4 - Average loading times in milliseconds of 2 and 3 minutes (in a single file) of audio data.. 39

Table 5 - Comparative table of request times. All times are in milliseconds..................... 41

Code 2 - JavaScript code responsible for decoding audio data and creating array buffer in Web Audio API with the timer start and end commands (marked in red)........................ 43

Illustration 4 - Google Chrome console window with the custom timings data. 43

Figure 8 – Web Audio API average decoding times across 3 different computer setups (based on 128kbps MP3-s) 44

Figure 9 - Web Audio API average decoding times across 3 different computer setups (based on 128kbps OGG-s) 45

Figure 10 - Web Audio API average decoding times across 3 different computer setups (based on 128kbps AAC-audio) 45

Table 6 - Average differences of average decoding times between 128kbps and 256kbps files 46
Table 7 - Comparison of decoding times (in milliseconds): 5x1minute vs 1x5minute of audio data. Smaller times are better and marked as green. 47
Table 8 - The lenght of silence in breakpoints across different computer setups when using MP3-s........ 48
Figure 11 - Encoder & decoder delays and padding cause a "gap" in the loop when the track is played iteratively. Figure taken from http://www.compuphase.com/mp3/mp3loops.htm 49
Table 9 - Spectral image before and after saving a piece of audio as MP3. On the "After saving" image, the red lines represent the positions of the gaps; the gap in the beginning is 27ms, at the end 17ms........ 50
Table 10 - The lenght of silence in breakpoints across different computer setups when using OGG-s. 51
Table 11 - The lenght of silence in breakpoints across different computer setups when using AAC-audio (in MP4 container)........ 51
Table 12 - Spectral image before and after saving a piece of audio as MP4. On the "After saving" image, the red line represent the positions of the end padding; the gap at the end is 44ms. 52
Code 3 – JavaScript event listener responsible for custom looping functionality with logging. 54
Code 4 - Function defining the flow of the signal and other playback parameters. The loop parameters are shown in red. 55
Table 13 - Reaction times of JavaScript when audio has been set to be looped through manually created subroutine throughout the test computers. All given values are in milliseconds. 56

Appendix A

Browser compatibility with `<audio>` tag according to [32].

Feature	Chrome	Firefox	Internet Explorer	Opera	Safari
Basic support	3.0	3.5	9.0	10.50	3.1
<audio>: PCM in WAVE	Yes	3.5	Not Supported	Not Supported	3.1
<audio>: Vorbis in WebM	Yes	4.0	Not Supported	10.60	3.1 (must be installed separately)
<audio>: Vorbis in Ogg	Yes	3.5	Not Supported	10.50	3.1 (must be installed separately)
<audio>: MP3	Yes	Partial*	9.0	Not Supported	3.1
<audio>: AAC in MP4	Yes	Partial*	9.0	Not Supported	3.1
<audio>: Opus in Ogg	27.0	15.0	Unknown	Unknown	Unknown

*Firefox supports MP3 and AAC partially. Because of the patent issues the support is not built directly into Firefox but instead it relies on support from the operating system or hardware. Therefore Firefox supports these formats on the following platforms: Windows 7+ (Firefox version 21.0), Windows Vista (version 22.0), Android (version 20.0).

Appendix B

The global attributes supported by the audio tag according to [27].

- accesskey – makes possible to access media element with specifically named keyboard key.
- class – element class name.
- contenteditable – if the attribute's value is true, content can be edited.
- dir – the directionality of the element's text.
- draggable – determines, if the element can be dragged.
- dropzone – defines the action when an item is dropped on the zone.
- hidden – boolean attribute which determine if the element will be rendered.
- Id – a unique identifier for the element.
- lang – specifies the primary language of the content.
- spellcheck – used to enable spell and grammar checking of the element's contents.
- style – inline CSS styling.
- tabindex – determines the element's order in tabbing sequence.
- title – tooltip info.

Appendix C

The results of the test to determine whether there are any differences in loading times when audio clip has been implemented using <audio>-tag or Web Audio API. Data is based from the information gathered using Google Chrome Developer tools' network timeline view.

1 minute of audio	*Sending*	*Waiting*	*Receiving*
<audio>-tag	1ms	292ms	1,49s
	1ms	289ms	1,49s
	1ms	274ms	1,49s
	1ms	275ms	1,49s
	2ms	288ms	1,49s
	1ms	284ms	1,49s
	0ms	279ms	1,49s
	4ms	279ms	1,49s
	2ms	284ms	1,49s
	1ms	268ms	1,48s
Web Audio API	1ms	113ms	1,42s
	2ms	125ms	1,47s
	1ms	110ms	1,49s
	1ms	115ms	1,50s
	1ms	113ms	1,50s
	1ms	114ms	1,50s
	0ms	118ms	1,50s
	0ms	118ms	1,49
	1ms	134ms	1.46s
	1ms	122ms	1.48s

Appendix D

The following data has been gathered using a specific command (“`window.performance.getEntries()['entry number']`”) in Google Chrome Developer tools' console window, which return raw timings data. All times in the following tables are in milliseconds. Values in the “Request start”, “Response start” and “Response end” columns are referencing to the time points when those specific events took place during the page load.

The following table represents download timings' data for 2-minutes of audio (2 x 1 minute audio files)

2x1 minutes of audio	*Request start*	*Response start*	*Response end*	*Total time*	*Request time*	*Receiving time*
<audio>-tag	547	938	4423	3876	391	3485
	945	1310	4424	3479	365	3114
	538	895	3916	3378	357	3021
	521	888	3878	3357	367	2990
	556	941	3939	3383	385	2998
	501	885	3881	3380	384	2996
	625	962	3989	3364	337	3027
	493	865	3867	3374	372	3002
	532	906	4376	3844	374	3470
	520	896	3892	3372	376	2996
Web Audio API	302	506	3548	3246	204	3042
	702	868	3914	3212	166	3046
	724	884	3933	3209	160	3049
	767	912	3975	3208	145	3063
	800	962	4022	3222	162	3060
	734	860	3977	3243	126	3117
	700	891	3950	3250	191	3059
	790	940	4036	3246	150	3096
	865	1012	4115	3250	147	3103
	714	854	3959	3245	140	3105

The following table represents download timings' data for 3-minutes of audio (3 x 1 minute audio files)

3x1 minutes of audio	*Request start*	*Response start*	*Response end*	**Total**	*Request time*	*Receiving time*
<audio>-tag	610	1023	5606	4996	413	4583
	980	1661	6318	5338	681	4657
	562	1057	5618	5056	495	4561
	606	1301	5873	5267	695	4572
	717	1162	5696	4979	445	4534
	562	1034	5589	5027	472	4555
	1125	1767	6418	5293	642	4651
	669	1135	6054	5385	466	4919
	490	976	5551	5061	486	4575
	495	983	5770	5275	488	4787
Web Audio API	1110	1243	5935	4825	133	4692
	772	914	5817	5045	142	4903
	875	1120	5678	4803	245	4558
	917	1118	5721	4804	201	4603
	1365	1499	6320	4955	134	4821
	785	972	5702	4917	187	4730
	747	1001	5718	4971	254	4717
	780	986	5809	5029	206	4823
	748	986	5763	5015	238	4777
	734	901	5598	4864	167	4697

The following table represents download timings' data for 5-minutes of audio (5 x 1 minute audio files)

5x1 minutes of audio	*Request start*	*Response start*	*Response end*	**Total**	*Request time*	*Receiving time*
<audio>-tag	540	1143	11069	10529	603	9926
	601	1262	9219	8618	661	7957
	936	1556	9511	8575	620	7955
	644	1202	9302	8658	558	8100
	1730	2133	9908	8178	403	7775
	606	1153	8858	8252	547	7705
	556	1162	10653	10097	606	9491
	1859	2982	11898	10039	1123	8916
	561	890	8823	8262	329	7933
	560	1154	8760	8200	594	7606
Web Audio API	1973	2224	10449	8476	251	8225
	937	1415	9084	8147	478	7669
	796	1047	8890	8094	251	7843
	893	1169	8995	8102	276	7826
	840	1065	8902	8062	225	7837
	901	1157	8975	8074	256	7818
	854	1055	8889	8035	201	7834
	752	1044	8908	8156	292	7864
	794	1208	8898	8104	414	7690
	879	1156	8933	8054	277	7777

Appendix E

Raw data of the tests which measure how the number of files influences the initial response time. All values are in milliseconds.

10 x 3 sec of audio	*Request start*	*Response start*	*Request time*
<audio>-tag	575	1221	646
	666	1274	608
	556	880	324
	652	1113	461
	2136	2761	625
	809	1265	456
	980	1437	457
	841	1433	592
	677	1142	465
	631	1253	622
Web Audio API	952	1115	163
	1264	1608	344
	1362	1688	326
	1026	1234	208
	831	1023	192
	888	1166	278
	1440	1756	316
	1141	1449	308
	838	982	144
	1617	1869	252

10 x 20 sec of audio	*Request start*	*Response start*	*Request time*
<audio>-tag	641	1142	501
	2380	3060	680
	629	1616	987
	686	1330	644
	597	1239	642
	573	1227	654
	655	1088	433
	601	958	357
	541	1140	599
	509	1110	601
Web Audio API	826	1117	291
	1321	1813	492
	911	1218	307
	993	1296	303
	883	1059	176
	907	1208	301
	847	1092	245
	918	1221	303
	892	1187	295
	874	1181	307

Appendix F

The following table represents download timings' data for 3-minutes of audio (1x3 minute audio file).

3 minutes of audio	*Request start*	*Response start*	*Response end*	**Total**	*Request time*	*Receiving time*
<audio>-tag*	556	884	15032	14476	328	14148
	531	817	17038	16507	286	16221
	534	841	17075	16541	307	16234
Web Audio API	741	859	5648	4907	118	4789
	762	879	5666	4904	117	4787
	729	847	5603	4874	118	4756
	756	870	5581	4825	114	4711
	705	821	5531	4826	116	4710
	686	801	5490	4804	115	4689
	690	809	5519	4829	119	4710
	1028	1144	5921	4893	116	4777
	965	1081	5845	4880	116	4764
	695	810	5532	4837	115	4722

*Stopped testing with audio-tag after third trial due to the fact that after buffering 2MB of audio data the download speed dropped significantly, resulting in a total download time up to 16 seconds. The reason behind this might be related to the way how browser handles the buffering of bigger audio files (which might include utilizing some sort of network traffic optimization).

The following table represents download timings' data for 2-minutes of audio (1x2 minute audio file).

2 minutes of audio	*Request start*	*Response start*	*Response end*	**Total**	*Request time*	*Receiving time*
<audio>-tag	541	775	3888	3347	234	3113
	553	847	3938	3385	294	3091
	644	880	3970	3326	236	3090
	503	756	3874	3371	253	3118
	504	814	3909	3405	310	3095
	474	770	3861	3387	296	3091
	538	832	3924	3386	294	3092
	493	786	3878	3385	293	3092
	539	836	3921	3382	297	3085
	519	806	3896	3377	287	3090
Web Audio API	1080	1201	4312	3232	121	3111
	1442	1845	4945	3503	403	3100
	1053	1341	4438	3385	288	3097
	789	905	4057	3268	116	3152
	1268	1384	4489	3221	116	3105
	782	905	4027	3245	123	3122
	717	831	3994	3277	114	3163
	689	803	3903	3214	114	3100
	757	878	4048	3291	121	3170
	846	960	4154	3308	114	3194

Appendix G

The following table shows raw timings the data in Table 13 is based on. The first column shows the time in seconds, when the custom looping function was set to cut the audio and change playback position back to the beginning of the file. The rest of the columns show the values across test systems (from computer no 1 to 3) when the playback position was really changed.

Ideal break time	*Break time/ Comp. no 1*	*Break time/ Comp. no 2*	*Break time/ Comp. no 3*
68,623438	68,763462	68,764766	68,740115
68,623438	68,629753	68,844426	68,840426
68,623438	68,717583	68,783376	68,859646
68,623438	68,597312	68,830816	68,763766
68,623438	68,585312	68,812206	68,632446
68,623438	68,677583	68,794986	68,689496
68,623438	68,701583	68,772766	68,850426
68,623438	68,676583	68,812206	68,739546
68,623438	68,745022	68,706717	68,721327
68,623438	68,597312	68,764766	68,829816

Appendix H

Raw data of MP3 decoding times in case of Web Audio API (times in milliseconds). ☐Computer no.1 ☐Computer no. 2 ☐Computer no. 3

The average value of each column is shown on the row between the triple lines.

1x1min 96kbps	2x1min 96kbps	3x1min 96kbps	5x1min 96kbps	1x1min 128kbps	2x1min 128kbps	3x1min 128kbps	5x1min 128kbps	1x1min 192kbps	2x1min 192kbps	3x1min 192kbps	5x1min 192kbps
3030	5900	8877	14284	3168	5762	8869	14671	3017	5911	8849	14662
2879	5979	8868	14441	3177	5970	8717	14452	3177	5954	8613	14754
2945	5801	8569	14067	2957	5974	8672	14604	3208	5889	8963	14639
3001	5941	8880	14444	3102	5766	8834	14493	3047	5973	8626	14332
2856	5854	8834	14381	2899	5949	8551	14591	3215	5816	8831	14599
2965	5779	8716	14456	3174	5808	8901	14437	3021	5845	8897	14485
2960	5834	8834	14402	2964	5725	8691	14522	3063	6084	8676	14508
3026,429	5869,714	8796,857	14353,57	2948	5850,571	8747,854	14538,57	3106,857	5924,571	8779,285	14568,43
2116	4047	5890	9492	2101	4021	5889	9944	2255	4165	6017	10160
2171	4105	5893	9650	2096	4071	5875	9660	2112	4247	5993	9972
2199	4030	6004	9512	2072	4045	5957	9838	2238	3998	6174	9956
2118	4077	5774	9661	2204	4001	5949	9696	2296	4200	6098	9969
2196	4140	6007	9558	2262	4150	5985	9868	2107	4219	6062	9968
2188	3956	5767	9720	2115	4022	5903	9602	2227	4077	6109	9966
2088	4064	5918	9467	2143	4029	5894	9906	2250	4105	5873	9960
2153,714	4059,857	5893,286	9580	2141,857	4048,429	5921,714	9787,714	2212,143	4144,429	6046,571	9993

1x1min 96kbps	2x1min 96kbps	3x1min 96kbps	5x1min 96kbps	1x1min 128kbps	2x1min 128kbps	3x1min 128kbps	5x1min 128kbps	1x1min 192kbps	2x1min 192kbps	3x1min 192kbps	5x1min 192kbps
1387	2678	3983	6635	1351	2664	4003	6684	1387	2696	4003	6666
1368	2677	3928	6713	1325	2683	3984	6646	1377	2707	4019	6670
1331	2681	3970	6604	1383	2695	3993	6649	1387	2701	4019	6681
1362	2676	3971	6598	1384	2613	3999	6664	1384	2687	4012	6647
1374	2677	3966	6594	1366	2658	3991	6670	1388	2691	4002	6662
1374	2616	4004	6595	1364	2660	3995	6617	1382	2703	4009	6673
1373	2664	3952	6601	1336	2645	4000	6652	1380	2702	4005	6673
1367	2667	3967,714	6620	1358,428	2659,714	3995	6654,571	1383,571	2698,143	4009,857	6667,429

1x1min 256kbps	*2x1min 256kbps*	*3x1min 256kbps*	*5x1min 256kbps*	*1x5min 96kbps*	*1x5min 128kbps*	*1x5min 192kbps*	*1x5min 256kbps*
3045	6223	8689	14662	13943	14211	14440	14476
3217	6156	8937	14740	13835	14177	14335	14593
3238	5935	8754	14605	13895	14127	14329	14576
2986	6226	9034	14657	13743	14237	14447	14429
3199	5949	8900	14821	13881	14131	14326	14548
3027	6208	8674	14684	13928	14135	14414	14500
3226	6150	9010	14574	13844	14161	14367	14567
3134	6121	8856,857	14677,57	13867	14168,43	14379,71	14527
2212	4063	6054	10071	9586	9980	9733	10110
2217	4174	6169	10055	9725	9912	10073	9863
2234	4125	6033	9820	9746	9946	9767	10110
2293	4158	6281	10068	9658	9975	10045	10063
2214	4141	6044	9913	9451	9940	9843	10166
2245	4114	6102	10066	9638	9831	10033	10125
2281	4142	6045	9994	9580	9938	9825	10167
2242,286	4131	6104	9998,143	9626,286	9931,714	9902,714	10086,29

1x1min 256kbps	*2x1min 256kbps*	*3x1min 256kbps*	*5x1min 256kbps*	*1x5min 96kbps*	*1x5min 128kbps*	*1x5min 192kbps*	*1x5min 256kbps*
1402	2715	4043	6730	6482	6663	6713	6715
1392	2724	4046	6700	6467	6642	6697	6745
1387	2722	4065	6731	6458	6640	6670	6723
1391	2728	4020	6748	6495	6572	6678	6702
1389	2706	4053	6733	6495	6621	6662	6718
1398	2707	4049	6708	6487	6606	6685	6764
1396	2718	4006	6692	6456	6596	6725	6747
1393,571	2717,143	4040,286	6720,286	6477,143	6620	6690	6730,571

Computer no. 1 – Dell Latitude D630 (laptop), CPU: Intel Mobile Core 2 Duo T7100 @ 1.8GHz, Cores: 2, Threads: 2, Memory: 2GB @ 333MHz

Computer no. 2 – Acer Aspire 7739 (laptop), CPU: Intel Core i3 380M @ 2.53GHz; Cores: 2, Threads: 4, Memory: 4GB @ 533MHz

Computer no. 3 – HP Pavilion 500 (desktop), CPU: Intel Core i5 3350P @ 3.10GHz, Cores: 4, Threads: 4, Memory: 8GB @ 800MHz

Appendix I

Raw data of OGG decoding times in case of Web Audio API (times in milliseconds). The average value of each column is shown on the row between the triple lines.

1x1min	2x1min	3x1min	5x1min	1x1min	2x1min	3x1min	5x1min	1x1min	2x1min	3x1min	5x1min
1561	3064	4403	7309	1209	2242	3359	5541	749	1467	2146	3544
1607	3237	4424	7312	1152	2350	3416	5531	756	1449	2142	3522
1595	3123	4606	7314	1169	2265	3338	5419	755	1444	2145	3538
1528	3219	4455	7289	1110	2288	3394	5479	750	1435	2141	3525
1640	2995	4642	7178	1170	2230	3402	5477	747	1455	2148	3546
1702	3138	4566	7374	1189	2275	3368	5572	748	1440	2119	3543
1651	3200	4566	7333	1179	2298	3332	5525	753	1445	2116	3527
1615	3060	4444	7299	1177	2345	3370	5552	748	1454	2146	3542
1597	3150	4542	7303	1200	2252	3378	5512	751	1443	2133	3559
1565	3037	4649	7295	1160	2223	3406	5521	754	1453	2139	3536
1606,1	3122,3	4529,7	7300,6	1171,5	2276,8	3376,3	5512,9	751,1	1448,5	2137,5	3538,2

Computer no. 1 – Dell Latitude D630 (laptop), CPU: Intel Mobile Core 2 Duo T7100 @ 1.8GHz, Cores: 2, Threads: 2, Memory: 2GB @ 333MHz

Computer no. 2 – Acer Aspire 7739 (laptop), CPU: Intel Core i3 380M @ 2.53GHz; Cores: 2, Threads: 4, Memory: 4GB @ 533MHz

Computer no. 3 – HP Pavilion 500 (desktop), CPU: Intel Core i5 3350P @ 3.10GHz, Cores: 4, Threads: 4, Memory: 8GB @ 800MHz

Appendix J

Raw data of AAC/MP4 decoding times in case of Web Audio API (times in milliseconds). The average value of each column is shown on the row between the triple lines.

1x1min	2x1min	3x1min	5x1min	1x1min	2x1min	3x1min	5x1min	1x1min	2x1min	3x1min	5x1min
1635	3058	4224	7063	1085	2201	3167	5284	743	1437	2117	3516
1631	3030	4352	7019	1142	2186	3219	5341	742	1414	2082	3472
1644	2942	4421	6857	1122	2196	3242	5335	742	1431	2103	3449
1397	3003	4353	6937	1212	2193	3207	5262	737	1418	2135	3479
1607	2891	4218	6987	1174	2164	3196	5332	703	1425	2108	3480
1638	3082	4395	6876	1136	2184	3209	5364	737	1439	2090	3472
1632	2902	4330	7018	1136	2188	3162	5316	740	1426	2116	3490
1525	3053	4460	6877	1185	2187	3262	5295	744	1421	2117	3466
1656	2882	4368	6977	1125	2188	3250	5335	740	1448	2108	3464
1635	3058	4224	7063	1161	2178	3282	5221	702	1435	2119	3488
1600,3	2974,6	4357,5	6963,5	1147,8	2186,5	3219,6	5308,5	733	1429,4	2109,5	3477,6

Computer no. 1 – Dell Latitude D630 (laptop), CPU: Intel Mobile Core 2 Duo T7100 @ 1.8GHz, Cores: 2, Threads: 2, Memory: 2GB @ 333MHz

Computer no. 2 – Acer Aspire 7739 (laptop), CPU: Intel Core i3 380M @ 2.53GHz; Cores: 2, Threads: 4, Memory: 4GB @ 533MHz

Computer no. 3 – HP Pavilion 500 (desktop), CPU: Intel Core i5 3350P @ 3.10GHz, Cores: 4, Threads: 4, Memory: 8GB @ 800MHz