ARCADE 3D-audio codec: an implementation for the web

François BECKER
Coronal Encoding
Lyon, France
francois@coronal.audio

Benjamin BERNARD
Coronal Encoding
Lyon, France
benjamin@coronal.audio

Clément CARRON
Coronal Encoding
Lyon, France
clement@coronal.audio

ABSTRACT

This poster introduces the implementation of the ARCADE 3D-audio codec for web browsers.

ARCADE can embed a full 3D audio scene in a simple stereo-compatible audio stream that can be further compressed with standard lossy compression schemes, aired to analog or digital radio receivers or even stored on analog supports. An ARCADE-encoded stream can be decoded to any 2D or 3D-audio rendering format, for instance using Vector-Based Amplitude Panning (VBAP), Higher Order Ambisonics (HOA), or personalized binaural with headtracking.

ARCADE adapts seamlessly to the audio industry needs, from storage to production, distribution/delivery, and rendering. It finds uses in Virtual or Augmented Reality (VR/AR), movies, gaming, music, telepresence & teleconferencing.

We present a JavaScript (JS) and Web Audio API implementation of the ARCADE decoder, which was originally written in C++11, along with technical details of the porting operations. Live demos of 3D-audio content transmission, decoding and dynamic binaural rendering will be given during the poster session.

1. INTRODUCTION

1.1 Overview of the ARCADE codec

The codec [1] allows transporting a 3D audio scene through two raw audio channels without metadata. Any 3D audio format such as Ambisonics or channel-based audio (Auro-3D, Atmos, standard 5.1 or 7.1 surround…) can be encoded and transported within standard stereo streams.

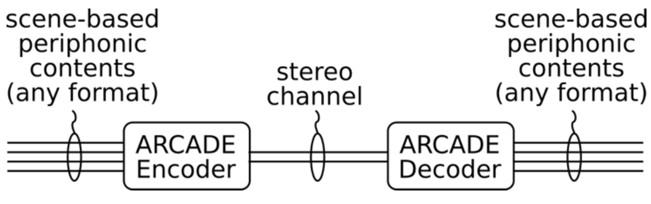
scene-based periphonic contents (any format)
stereo channel
scene-based periphonic contents (any format)
ARCADE Encoder
ARCADE Decoder

Figure 1. ARCADE general workflow

The encoding stage analyses the scene-based audio in a perceptual manner and in the frequency domain. The encoded signal, converted back to the time domain, is stereo-compatible, allowing it to be listened to through a stereo speaker setup. It can be distributed through analog or digital channels as it is robust to signal degradation and data compression (Advanced Audio Coding [AAC], MPEG Layer-3 [MP3], Ogg Vorbis and Opus…).

The decoding stage restores the audio scene in the frequency domain, then renders it. Virtually any rendering technique can be used to distribute the signal to various audio listening systems: headphones (e.g. binaural, for VR/AR), surround (2D) or periphonic (3D) speaker layouts (e.g. VBAP or Spherical Harmonics, for home cinema, car audio systems), etc. A binaural renderer, outside the scope of the present work, is used for the decoded scene-based or multichannel content demonstrations.

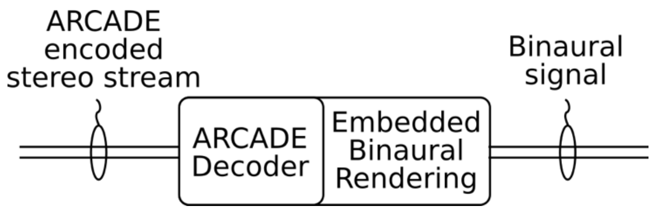

Figure 2. Decoding to binaural

1.2 Codec DSP operations

The ARCADE codec notably works in the frequency domain by analyzing the magnitude and phase relationships between the input signal components and using the extracted information to encode (resp. reconstruct) the original scene-based soundfield, which mandates frequent use of a real-to-complex Fast Fourier Transform (FFT) (resp. its complex-to-real inverse counterpart). One of the major challenges encountered during the present work was to achieve a good performance for said transforms, which is detailed in section 2.3.

Other DSP operations are quite standard ones and don't need specific libraries.

1.3 Porting options and scope

The codec library was originally developed in C++11 [2], a standard for audio processing algorithms. The codec being able to convert a full 3D soundfield to a (optionally further compressed) two-channel stereo stream, it finds uses in web content distribution and its availability in web browsers is an important feature. We had three options to port the codec to the web:

- rewrite the codec codebase as JS code, using the native Web Audio API primitives whenever possible;
- compile the ARCADE C++11 codebase as a binary blob to be embedded within a web browser plugin;

- use a source-to-source compiler such as Emscripten [3], PNaCl [4] or cheerp [5] to generate JS code from the ARCADE C++11 codebase.

While the encoder could also be rewritten in JS and run in the browser or on a server (using for example Node.js), the scope of work was, mainly for the reasons exposed in section 2.4, restricted to the decoder.

2. PRELIMINARY EXPLORATIONS

2.1 Porting options review

A total rewriting of the codec codebase was out of question, due to the limited time allocated to this project on the one hand, and on the other due to the fact that the Web Audio API didn't offer the needed native nodes, especially for complex-valued FFT. Additionally, sharing the same codebase between desktop and web version had substantial advantages over a code split, especially regarding maintenance and future evolutions.

The blob option was also deemed impractical because of the large number of platforms and browsers to be supported, and because it needed manual installation on the client side. Moreover, mobile browsers, for instance Safari Mobile, don't support binary plug-ins.

The only remaining approach was the source-to-source option, which we detail in this presentation. We chose to use Emscripten over NaCl or Cheerp, mainly for its performance and proven track record in porting audio-related code.

2.2 Emscripten

Emscripten allows compiling LLVM (Low Level Virtual Machine) bytecode to JS, relying on asm.js, a strict subset of JS where all operations are clearly statically typed. It can translate C/C++ code, but also any LLVM-supported language such as Python or Ruby, to bytecode that web browsers can understand.

Emscripten seemed to be an interesting choice for porting the decoder, as it had already been used for similar tasks, especially during the Faust port [10] or for using DAW plugins in browsers [11].

2.3 Choice of a FFT library

The ARCADE codebase originally used the Intel Integrated Performance Primitives (IPP) [6], a software library for multimedia and data processing applications, especially for its FFT operations. The IPP library being closed-source and using CPU-dependent instructions, its processing with a source-to-source compiler such as Emscripten was impossible.

We hence had to remap all calls to the IPP FFT routines to another library. Emscripten allowing to call either a C/C++ or a native JS function, the panel of choice included both native JS FFT libraries (Nokert fft.js, dntj jsfft, Project Nayuki's fft) and C-based libraries (KissFFT, FFTW) compiled with Emscripten.

Performance tests for all these libraries were readily available online [7], theoretically allowing us to test them within different web browsers (Firefox, Safari, Safari Mobile, Microsoft Edge, Google Chrome), while using different platforms (desktops, mobile phones or tablets) and operating systems (macOS, iOS, Windows and Android).

However, we decided to modify the scripts from [7] so that the testing conditions exactly replicated the ARCADE prototype use case: either double precision floating point inputs and outputs (the native JS format) or 32-bit floats, fixed 2048 sample block-size, real-to-complex (forward) and complex-to-real (inverse) transforms. The performance test metric was chosen to be the mean iteration rate per second, when running 8000 iterations of the forward and inverse transforms.

Special attention was given to these tests in order to get as comparable as possible results across libraries:

- as some FFT implementations include initialization routines, such as look-up tables precomputations, whose overhead can bias the initial results, the performance for the first half of iterations was discarded and only the remaining 4000 iterations were retained;
- when libraries only offered complex-complex forward (resp. inverse) transforms, they were used as a fallback solution by zeroing (resp. ignoring) the imaginary part of the input (resp. output);
- adequate scaling was applied to the forward and inverse transforms when needed by some libraries, so that the forward-inverse transform round-trip yielded a unitary gain.
- when optimized real-to-complex forward transforms (FFTW, KissFFT…) were available, they were used although they only returned the positive frequency bins.

We are aware this comparison isn't balanced for some libraries, due to the difference in input types (as mentioned above, native JS libraries use double precision, while some C libraries can use single precision floats). Our goal was to "blackbox-benchmark" the current offerings and find, on average, the best performing library on all target platforms.

Results details, as found in Table 1 and summarized in Figure 3 and Figure 4, show interesting behaviors:

- using a single precision inputs and processes instead of the native double precision has a slight impact on performance: the 32-bit Nayuki C++ implementation was 13% less performant than the native, 64-bit one. This was expected as Emscriptem-compiled code stores floats as float32, but uses float64 arithmetic, causing a small CPU overhead [8].
- the Nayuki JS implementation is the best performing JS library on all platforms and hardware.
- the well-known FFTW library shows very inconsistent results across browsers and platforms: while it gets the best results on Mozilla browsers, it also has a very poor performance on Blink/Webkit-based ones, such as Safari and Safari Mobile. Limiting factors might be the size of the Emscriptem-compiled FFTW (around 3 MB, while other C/C++ implementations average at 62 KB), or the call stack size limits of the various JS engines, but we couldn't find a definitive answer as to these significant discrepancies.
- the optimized real KissFFT performance is also quite low, mainly due to the performance of its inverse transform.
- the library that consistently showed the most performance across desktop and mobile platforms, and browsers, was the C-based KissFFT complex transform [9], which we selected for the rest of this work.

Table 1. FFT performance comparison, iterations / sec

			Implementation	Nayuki	Nayuki	Nayuki	KissFFT	KissFFT	Cross	FFTW	Nockert	dntj
			Language	JS	C/C++	C/C++	C/C++	C/C++	C/C++	C/C++	JS	JS
			Precision	64	64	32	32	32	64	32	64	64
			Type	Complex	Complex	Complex	Real	Complex	Real	Real	Real	Complex
			Size	8 KB	61 KB	61 KB	64 KB	64 KB	60 KB	3.3 MB	41 KB	10 KB
Hardware	**CPU**	**OS**	**Browser**									
MacBookPro11,3	2.3 GHz Intel Core i7	macOS 10.12	Firefox 54	5863	9569	7944	4613	9213	8427	21132	2957	937
			Chrome 59	5107	5885	4597	4387	9672	6504	8237	1599	4721
			Safari 10.1.1	8190	8882	7397	4496	11046	9157	928	648	1397
	2.3 GHz Intel Core i7	Windows 10	Firefox 54	4172	5790	6295	4129	10731	5562	12775	2120	366
			Chrome 59	5742	6501	5633	4719	10360	6294	9059	1553	4817
			Edge 38	3459	8299	7249	4016	6592	6333	11426	759	1037
iPhone 5	Apple A6 1.3 GHz	iOS 10.3	Safari Mobile	539	756	530	1719	1225	727	69	72	204
iPhone 7	Apple A10 Fusion 2.3 GHz	iOS 10.3	Safari Mobile	8558	9558	8917	22896	11034	8511	875	599	1395
iPad 4	Apple A6X 1.4 GHz	iOS 10.3	Safari Mobile	646	898	705	2002	1341	874	89	88	262
iPad Air 2	Apple A8X 1.5 GHz	iOS 10.2.1	Safari Mobile	4174	5489	4733	12266	5738	4439	167	255	756
Ipad Mini 2	Apple A7 1.3 GHz	iOS 10.3.1	Safari Mobile	3117	3787	3180	6693	4473	3441	62	196	570
Asus K013	Intel Atom Z3745 1.86 GHz	Android 5.0	Chrome 58	929	1144	951	589	2083	1264	979	445	935
			Firefox 52	961	1320	1190	598	2348	1286	1290	553	79
Samsung Galaxy S6	Samsung Exynos 7 Octa 2.1 GHz	Android 7.0	Firefox 52	897	956	1206	4032	1620	1117	1859	467	0
			Chrome 57	1424	1554	1707	5713	2720	1711	2035	378	1534
Desktop average				**5422**	**7488**	**6519**	**4393**	**9602**	**7046**	**10593**	**1606**	**2213**
Mobile average				**2361**	**2829**	**2569**	**6279**	**3620**	**2597**	**825**	**339**	**637**

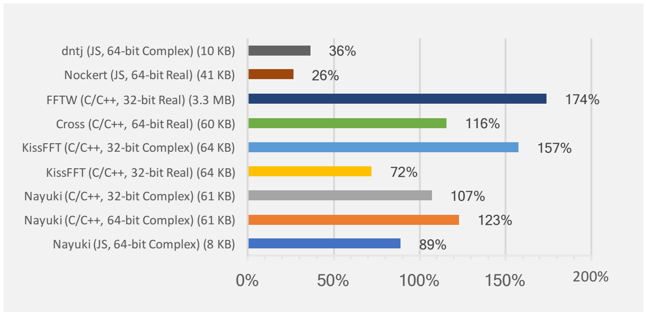

Figure 3. Relative performance on desktop systems. Average performance is 6098 iterations/s

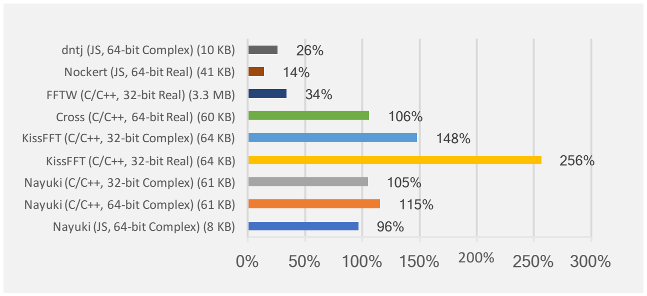

Figure 4. Relative performance on mobile platforms. Average performance is 2451 iterations/s

2.4 Intellectual property concerns

Intellectual property (IP) protection is a major concern in web-based software: as web technologies mostly rely on open standards, client-side code can be easily accessed and read. That's the reason why service took precedence over bare code, a paradigm shift that got momentum with web-based applications.

If ARCADE were to be ported to native JS, an easy option to prevent reverse-engineering would be to obfuscate JS code, but many tools exist that allow the user to remove this protection layer. This fact, to which one can add time-to-market and performance concerns, led us to go the Emscripten way. However, while reverse-engineering Emscripten-generated code would probably be a serious challenge, the final code is still embeddable as a whole and can hence be used in non-authorized manners.

In the absence of valuable protection offerings, another option would be to port as little IP as possible to the client side, where it could be ill-intentionally reused. However, the principal use case for ARCADE being the decoding and consumption of immersive audio content, be it within a web browser or in other scenarios, the decoder was the priority porting task, which technically limited IP dissemination.

3. IMPLEMENTATION

3.1 Integration details

The decoder has been integrated into a web player that allows playing audio content available on the web, and decode the stereo content using the ARCADE decoder. A binaural rendering is done at the rendering stage of the decoder, that allows an immersive experience. The accepted formats (MP3, Vorbis…) depend on the browser, as it is responsible for decoding the audio streams.

The running Web Audio node graph is a simple three nodes chain, with an AudioBufferSource node providing the raw PCM audio data, followed by a ScriptProcessor node running the ARCADE decoder including its binaural rendering stage, and finally an AudioDestination node provided by the AudioContext.

Device orientation tracking through the adequate browser primitives allows a user experience that mimicks that of headtracking, substantially improving the localization by the listener of the sources in the content: the user is then able to pan into the soundfield by moving the mobile device around.

The Spheroscope, a visual representation of the decoded soundfield whose code was originally in C++, has been implemented in JS via Emscripten, allowing the user to visualize the spatial audio content as it is decoded and played back. Figure 5 shows the Spheroscope displaying spatial audio content using a Lambert azimuthal equal-area projection of the sphere. A simple continuous color scheme has been used to ease the visualization of the spatial audio spectrum (lowest frequencies are displayed as red points, highest frequencies appear as blue points, while all color nuances are used for in-between frequency bands). The pixel intensity is directly linked to the content magnitude of the frequency band.

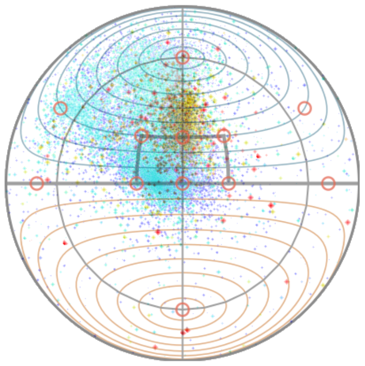

Figure 5. Spheroscope soundfield visualization

3.2 Browser compatibility issues

The stock Safari and Safari Mobile browsers still use the deprecated *webkitAudioContext*, as well as deprecated properties and method names, instead of following the Web Audio API-compliant AudioContext specification. We used the monkeypatch JS library [13] that transparently allows aliasing the deprecated names to the standard specification, ensuring wide compatibility with most browsers.

As for the device orientation tracking, we used the gyronorm.js [14] JS library that unifies browsers' behaviour regarding gyroscope data.

4. PERFORMANCE ASSESSMENT

4.1 Denormals

It has been reported in [10] that the absence of denormals management in JS was prone to cause issues in the Web Audio API, especially when dealing with feedback loops where denormals usually occur (e.g. reverbs, recursive filters…). The reason is that browsers don't provide an interface for setting to the adequate CPU flags (DAZ & FTZ). To solve the issue, a manual flush to zero has been manually implemented in code locations where denormals are prone to be found, in both the time and frequency domains.

4.2 Performance figures

We have run the ARCADE decoder library in the player on multiple desktop and mobile platforms. Desktop implementations run with no stuttering whatsoever, as do high-end mobile devices like the latest iPhones. However, on middle- or low-end mobile platforms like the iPhone 5, some buffer underruns occasionally occur, worsened by the activation of the device orientation tracking. The underlying cause is well known: the threading model behind the ScriptProcessorNode, which runs asynchronously in the main thread, and can even be interrupted by DOM manipulations, incited the Web Audio community to propose the (never implemented) AudioWorker model, followed by the (long awaited) AudioWorklet whose implementation is still pending at the time of writing. Nevertheless, the empirical evaluation that we have at the time of writing looks promising.

5. CONCLUSIONS AND FUTURE WORK

This paper has introduced an implementation for the web of the ARCADE 3D-audio codec decoder. The design decisions were presented, as well as the challenges encountered during the porting operations. The code performance was evaluated in terms of usability on common end-user platforms. The results show that the ARCADE decoder can be used on both desktop and mobile platforms in order to decode and render a full-3D soundfield in the browser, while using limited network bandwidth.

As soon as the AudioWorklet proposal gets implemented in major browsers, we'll obviously switch to that model, due to the audio glitches the ScriptProcessor threading model generates.

Our future work will deal with the switch from Emscripten to WebAssembly [12] (wasm), a recently proposed intermediate representation language (IRL) whose support in browsers is yet partially based on asm.js (and Google's PNaCl). The wasm development community plans on delivering close-to-native performance and improved compacity compared to the current asm.js IRL. Available as a LLVM backend, it will eventually become a serious option for real-time signal processing once all browsers natively support wasm bytecode. At the moment, while Emscripten already supports wasm, the benefits of using the latter are not obvious as it still relies on an asm.js intermediate representation. Nevertheless, we plan on using wasm as the WebAssembly project matures.

6. REFERENCES

[1] Coronal Encoding SAS. 2016. *ARCADE: 3D audio codec over stereo*. http://www.coronal.audio/2016/10/arcade-3d-audio-codec-stereo/.

[2] ISO. 2012. *ISO/IEC 14882:2011 Information technology --- Programming languages --- C++*. International Organization for Standardization, Geneva, Switzerland.

[3] Zakai, A. 2011. *Emscripten: an LLVM-to-JavaScript compiler*. Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (pp. 301-312), ACM.

[4] Donovan, A., Muth, R., Chen, B., & Sehr, D. 2010. *Pnacl: Portable native client executables*. Google White Paper.

[5] Leaning Technologies Limited. 2017. *Cheerp - C++ for the Web*. http://leaningtech.com/cheerp/index.html.

[6] Intel Corporation. 2017. *Intel® Integrated Performance Primitives (Intel® IPP) | Intel® Software*. https://software.intel.com/en-us/intel-ipp.

[7] Cannam, C. 2017. *Javascript FFT speed test*. http://all-day-breakfast.com/js-dsp-test/fft/.

[8] Bouvier, B. 2013. *Efficient float32 arithmetic in JavaScript*. https://blog.mozilla.org/javascript/2013/11/07/efficient-float32-arithmetic-in-javascript/.

[9] Borgerding, M. 2017. *Kiss FFT download | SourceForge.net*. http://kissfft.sourceforge.net.

[10] Borins, M. 2014. *From Faust to Web Audio: Compiling Faust to JavaScript using Emscripten*. Linux Audio Conference, Karlsruhe, Germany.

[11] Kleimola, J. 2015. *Daw plugins for web browsers*. 1st Web Audio Conference, IRCAM, Paris.

[12] The WebAssembly Community. 2017. *WebAssembly*. http://webassembly.org

[13] Wilson, C. 2015. *Monkeypatch to use proper AudioContext naming on prefixed/deprecated named systems*. https://github.com/cwilso/AudioContext-MonkeyPatch.

[14] Eker, D. 2017. *JavaScript project for accessing and normalizing the accelerometer and gyroscope data on mobile devices*. https://github.com/dorukeker/gyronorm.js