Developing game audio with the Web Audio API

- Game Audio on the Web
- Background music
  - The missing link: Audio tag to Web Audio
- Sound effects
- 3D positional sound
  - Advanced: sound cones
- Room effects and filters
- The final countdown
  - Detect clipping
  - Prevent clipping
  - Add a bit of sugar
- Conclusion

Audio is a huge part of what makes multimedia experiences so compelling. If you've ever tried watching a movie with the sound off, you've probably noticed this.

Games are no exception! My fondest video game memories are of the music and sound effects. Now, in many cases nearly two decades after playing my favorites, I still can't get Koji Kondo's Zelda compositions and Matt Uelmen's atmospheric Diablo soundtrack out of my head. The same catchiness applies to sound effects, such as the instantly recognizable unit click responses from Warcraft, and samples from Nintendo's classics.

Game audio presents some interesting challenges. To create convincing game music, designers need to adjust to potentially unpredictable game state a player finds themselves in. In practice, parts of the game can go on for an unknown duration, sounds can interact with the environment and mix in complex ways, such as room effects and relative sound positioning. Finally, there can be a large number of sounds playing at once, all of which need to sound good together and render without introducing performance penalties.

Game Audio on the Web

For simple games, using the `<audio>` tag may be sufficient. However, many browsers provide poor implementations, which result in audio glitches and high latency. This is hopefully a temporary problem, since vendors are working hard to improve their respective implementations. For a glimpse into the state of the `<audio>` tag, there is a nice test suite at areweplayingyet.org.

Looking deeper into the `<audio>` tag specification, however, it becomes clear that there are many things that simply can't be done with it, which isn't surprising, since it was designed for media playback. Some limitations include:

- No ability to apply filters to the sound signal
- No way to access the raw PCM data
- No concept of position and direction of sources and listeners
- No fine-grained timing.

In the rest of the article, I'll dive into some of these topics in the context of game audio written with the Web Audio API. For a brief introduction to this API, take a look at the getting started tutorial.

Background music

Games often have background music playing on a loop.

It can get very annoying if your loop is short and predictable. If a player is stuck in an area or level, and the same sample continuously plays in the background, it may be worthwhile to gradually fade out the track to prevent further frustration. Another strategy is to have mixes of various intensity that gradually crossfade into one another, depending on the context of the game.

For example, if your player is in a zone with an epic boss battle, you might have several mixes varying in emotional range from atmospheric to foreshadowing to intense. Music synthesis software often allows you to export several mixes (of the same length) based on a piece by picking the set of tracks to use in the export. That way you have some internal consistency and avoid having jarring transitions as you cross-fade from one track to another.

> *Garageband*

Then, using the Web Audio API, you can import all of these samples using something like the BufferLoader class via XHR (this is covered in depth in the introductory Web Audio API article. Loading sounds takes time, so assets that are used in the game should be loaded on page load, at the start of the level, or perhaps incrementally while the player is playing.

Next, you create a source for each node, and a gain node for each source, and connect the graph.

After doing this, you can play back all of these sources simultaneously on a loop, and since they are all the same length, the Web Audio API will guarantee that they will remain aligned. As the character gets nearer or further from the final boss battle, the game can vary the gain values for each of the respective nodes in the chain, using a gain amount algorithm like the following:

```
// Assume gains is an array of AudioGainNode, normVal is the intensity
// between 0 and 1.
var value = normVal - (gains.length - 1);
// First reset gains on all nodes.
for (var i = 0; i < gains.length; i++) {
    gains[i].gain.value = 0;
}
// Decide which two nodes we are currently between, and do an equal
// power crossfade between them.
var leftNode = Math.floor(value);
// Normalize the value between 0 and 1.
var x = value - leftNode;
var gain1 = Math.cos(x - 0.5*Math.PI);
var gain2 = Math.cos((1.0 - x) - 0.5*Math.PI);
// Set the two gains accordingly.
gains[leftNode].gain.value = gain1;
// Check to make sure that there's a right node.
if (leftNode < gains.length - 1) {
    // If there is, adjust its gain.
    gains[leftNode + 1].gain.value = gain2;
}
```

In the above approach, two sources play at once, and we crossfade between them using equal power curves (as described in the intro).

The missing link: Audio tag to Web Audio

Many game developers today use the `<audio>` tag for their background music, because it is well suited to streaming content. Now you can bring content from the `<audio>` tag into a Web Audio context.

This technique can be useful since the `<audio>` tag can work with streaming content, which lets you immediately play the background music instead of having to wait for it all to download. By bringing the stream into the Web Audio API, you can manipulate or analyze the stream. The

following example applies a low pass filter to the music played through the `<audio>` tag:

```
var audioElement = document.querySelector('audio');
var mediaSourceNode = context.createMediaElementSource(audioElement);
// Create the filter
var filter = context.createBiquadFilter();
// Create the audio graph.
mediaSourceNode.connect(filter);
filter.connect(context.destination);
```

For a more complete discussion about integrating the `<audio>` tag with the Web Audio API, see this short article.

Sound effects

Games often play back sound effects in response to user input or changes in game state. Like background music, however, sound effects can get annoying very quickly. To avoid this, it's often useful to have a pool of similar but different sounds to play. This can vary from mild variations of footstep samples, to drastic variations, as seen in the Warcraft series in response to clicking on units.

Another key feature of sound effects in games is that there can be many of them simultaneously. Imagine you're in the middle of a gunfight with multiple actors shooting machine guns. Each machine gun fires many times per second, causing tens of sound effects to be played at the same time. Playing back sound from multiple, precisely timed sources simultaneously is one place the Web Audio API really shines.

The following example creates a machine gun round from multiple individual bullet samples by creating multiple sound sources whose playback is staggered in time.

```
var time = context.currentTime;
for (var i = 0; i < rounds; i++) {
    var source = this.makeSource(this.buffers[M4A1]);
    source.noteOn(time + i - interval);
}
```

Now, if all of the machine guns in your game sounded exactly like this, that would be pretty boring. Of course they would vary by sound based on distance from the target and relative position (more on this later), but even that might not be enough. Luckily the Web Audio API provides a way to easily tweak the example above in two ways:

1. With a subtle shift in time between bullets firing
2. By altering playbackRate of each sample (also changing pitch) to better simulate randomness of the real world.

For a more real-life example of these techniques in action, take a look at the Pool Table demo, which uses random sampling and varies playbackRate for a more interesting ball collision sound.

3D positional sound

Games are often set in a world with some geometric properties, either in 2D or in 3D. If this is the case, stereo positioned audio can greatly increase the immersiveness of the experience. Luckily, Web Audio API comes with built-in hardware accelerated positional audio features that are quite straight forward to use. By the way, you should make sure that you've got stereo speakers (preferably headphones) for the following example to make sense.

In the above example, there is a listener (person icon) in the middle of the canvas, and the mouse affects the position of the source (speaker icon). The above is a simple example of using AudioPannerNode to achieve this sort of effect. The basic idea of the sample above is to respond to mouse movement by setting the position of the audio source, as follows:

```
PositionSample.prototype.changePosition = function(position) {
    // Position coordinates are in normalized canvas coordinates
    // with -0.5 < x, y < 0.5
    if (position) {
    if (!this.isPlaying) {
        this.play();
    }
    var mul = 2;
    var x = position.x / this.size.width;
    var y = -position.y / this.size.height;
    this.panner.setPosition(x - mul, y - mul, -0.5);
    } else {
    this.stop();
    }
};
```

Things to know about Web Audio's treatment of spatialization:

- The listener is at the origin (0, 0, 0) by default.
- Web Audio positional APIs are unitless, so I introduced a multiplier to make the demo sound better.

- Web Audio uses the y-is-up cartesian coordinates (the opposite of most computer graphics systems). That's why I'm swapping the y-axis in the snippet above

Advanced: sound cones

The positional model is very powerful and quite advanced, largely based on OpenAL. For more details, see sections 3 and 4 of the above-linked spec.

Position model

There is a single AudioListener attached to the Web Audio API context which can be configured in space through position and orientation. Each source can be passed through an AudioPannerNode, which spatializes the input audio. The panner node has position and orientation, as well as a distance and directional model.

The distance model specifies the amount of gain depending on proximity to the source, while the directional model can be configured by specifying an inner and outer cone, which determine amount of (usually negative) gain if the listener is within the inner cone, between the inner and outer cone, or outside the outer cone.

```
var panner = context.createPanner();
panner.coneOuterGain = 0.5;
panner.coneOuterAngle = 180;
panner.coneInnerAngle = 0;
```

Though my example is in 2D, this model generalizes easily to the third dimension. For an example of sound spatialized in 3D, see this positional sample. In addition to position, the Web Audio sound model also optionally includes velocity for doppler shifts. This example shows the doppler effect in more detail.

For more information on this topic, read this detailed tutorial on [mixing positional audio and WebGL][webgl].

Room effects and filters

In reality, the way sound is perceived greatly depends on the room in which that sound is heard. The same creaky door will sound very different in a basement, compared to an large open hall. Games with high production value will want to imitate these effects, since creating a separate set of samples for each environment is prohibitively expensive, and would lead to even more assets, and a larger amount of game data.

Loosely speaking, the audio term for the difference between the raw sound and the way it sounds in reality is the impulse response. These impulse responses can be painstakingly recorded, and in fact there are sites that host many of these pre-recorded impulse response files (stored as audio) for your convenience.

For a lot more information about how impulse responses can be created from a given environment, read through the "Recording Setup" section in the Convolution part of the Web Audio API spec.

More importantly for our purposes, the Web Audio API provides an easy way to apply these impulse responses to our sounds using the ConvolverNode.

```
// Make a source node for the sample.
var source = context.createBufferSource();
source.buffer = this.buffer;
// Make a convolver node for the impulse response.
var convolver = context.createConvolver();
convolver.buffer = this.impulseResponseBuffer;
// Connect the graph.
source.connect(convolver);
convolver.connect(context.destination);
```

Also see this demo of room effects on the Web Audio API spec page, as well as this example which gives you control over dry (raw) and wet (processed via convolver) mixing of a great Jazz standard.

The final countdown

So you've built a game, configured your positional audio, and now you have a large number of AudioNodes in your graph, all playing back simultaneously. Great, but there's still one more thing to consider:

Since multiple sounds just stack on top of one another with no normalization, you may find yourself in a situation where you are exceeding the threshold of your speaker's capability. Like images exceeding past their canvas boundaries, sounds can also clip if the waveform exceeds its maximum threshold, producing a distinct distortion. The waveform looks something like this:

Clipping

Here's a real example of clipping in action. The waveform looks bad:

Clipping

It's important to listen to harsh distortions like the one above, or conversely, overly subdued mixes that force your listeners to crank up the volume. If you're in this situation, you really need to fix it!

Detect clipping

From a technical perspective, clipping happens when the value of the signal in any channel exceeds the valid range, namely between -1 and 1. Once this is detected, it's useful to give visual feedback that this is happening. To do this reliably, put a JavaScriptAudioNode into your graph. The audio graph would be setup as follows:

```
// Assume entire sound output is being piped through the mix node.
var meter = context.createJavaScriptNode(2048, 1, 1);
meter.onaudioprocess = processAudio;
mix.connect(meter);
meter.connect(context.destination);
```

And clipping could be detected in the following `processAudio` handler:

```
function processAudio(e) {
    var buffer = e.inputBuffer.getChannelData(0);

    var isClipping = false;
    // Iterate through buffer to check if any of the |values| exceeds 1.
    for (var i = 0; i < buffer.length; i++) {
    var absValue = Math.abs(buffer[i]);
    if (absValue >= 1) {
        isClipping = true;
        break;
    }
    }
}
```

In general, be careful not to overuse the `JavaScriptAudioNode` for performance reasons. In this case, an alternative implementation of metering could poll a `RealtimeAnalyserNode` in the audio graph for `getByteFrequencyData`, at render time, as determined by `requestAnimationFrame`. This approach is more efficient, but misses most of the signal (including places where it potentially clips), since rendering happens at most at 60 times a second, whereas the audio signal changes far more quickly.

Because clip detection is so important, it's likely that we will see a built-in `MeterNode` Web Audio API node in the future.

Prevent clipping

By adjusting the gain on the master AudioGainNode, you can subdue your mix to a level that prevents clipping. However, in practice, since the sounds playing in your game may depend on a huge variety of factors, it can be difficult to decide on the master gain value that prevents clipping for all states. In general, you should tweak gains to anticipate the worst case, but this is more of an art than a science.

Add a bit of sugar

Compressors are commonly used in music and game production to smooth over the signal and control spikes in the overall signal. This functionality is available in the Web Audio world via the `DynamicsCompressorNode`, which can be inserted in your audio graph to give a louder, richer and fuller sound, and also help with clipping. Directly quoting the spec, this node

...lowers the volume of the loudest parts of the signal and raises the volume of the softest parts... It is especially important in games and musical applications where large numbers of individual sounds are played simultaneous to control the overall signal level and help avoid clipping.

Using dynamics compression is generally a good idea, especially in a game setting, where, as previously discussed, you don't know exactly what sounds will play and when. Plink from DinahMoe labs is a great example of this, since the sounds that are played back completely depends on you and other participants. A compressor is useful in most cases, except some rare ones, where you're dealing with painstakingly mastered tracks that have been tuned to sound "just right" already.

Implementing this is simply a matter of including a DynamicsCompressorNode in your audio graph, generally as the last node before the destination.:

```
// Assume the output is all going through the mix node.
var compressor = context.createDynamicsCompressor();
mix.connect(compressor);
compressor.connect(context.destination);
```

For more detail about dynamics compression, this Wikipedia article is very informative.

To summarize, listen carefully for clipping and prevent it by inserting a master gain node. Then tightening the whole mix by using a dynamics compressor node. Your audio graph might look something like this:

Final result

Conclusion

That covers what I think are the most important aspects of game audio development using the Web Audio API. With these techniques, you can build truly compelling audio experiences right in your browser. Before I sign off, let me leave you with a browser-specific tip: be sure to pause sound if your tab goes to the background using the page visibility API, otherwise you will create a potentially frustrating experience for your user.

For additional information about Web Audio, take a look at the more introductory getting started article, and if you have a question, see if it's already answered in the web audio FAQ. Finally, if you have additional questions, ask them on Stack Overflow using the web-audio tag.

Before I sign off, let me leave you with some awesome uses of the Web Audio API in real games today:

- Field Runners, and a writeup about some of the technical details.
- Angry Birds, recently switched to Web Audio API. See this writeup for more information.
- Skid Racer, which makes great use of spatialized audio.

□□□ #3
□ □□□ □□□ 2026-10-31 22:27:53 UTC
□ cocobook-sync □□□ 2026-11-04 17:38:28 UTC