Load and play sound samples

USE AN `AUDIOBUFFERSOURCENODE` AS THE SOURCE OF THE SOUND SAMPLE IN THE WEB AUDIO GRAPH

There is a special node in Web Audio for handling sound samples, called an `AudioBufferSourceNode`.

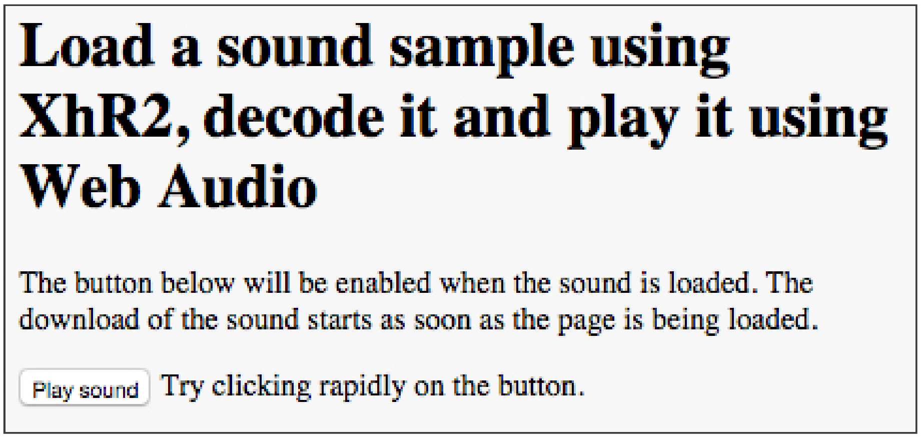
This node has different properties:

- `buffer`: the decoded sound sample.
- `loop`: if true, the sample will be played as an infinite loop when the sample has been played to its end (default), it also depends on the two next properties.
- `loopStart`: a double value indicating, in seconds, where in the buffer sample, the restart of the play must happen. Its default value is 0.
- `loopEnd`: a double value indicating, in seconds, where in the buffer sample, the replay of the play must stop (and eventually loop again). Its default value is 0.
- `playbackRate`: the speed factor at which the audio asset will be played. Since no pitch correction is applied on the output, this can be used to change the pitch of the sample.
- `detune`: not relevant for this course.

LOADING AND DECODING A SOUND SAMPLE

You need to load a sample using Ajax before using it, and set it to the `buffer` property of an`AudioBufferSourceNode` once decoded.

Try the example at JSBin:



Load a sound sample using XhR2, decode it and play it using Web Audio

The button below will be enabled when the sound is loaded. The download of the sound starts as soon as the page is being loaded.

Try clicking rapidly on the button.

In this example, as soon as the page is loaded, we send an Ajax request to a remote server in order to get the file shoot2.mp3. When the file is loaded, we decode it. Then we enable the button (before the sample was not available, and could not be played), and you can click on the button to play it.

Notice in the code that each time we click on the button, we rebuild the audio graph.

This is because AudioBufferSourceNodes can be used only once!

But don't worry, Web Audio is optimized for handling thousands of nodes...

HTML code extract:

```
<button id="playButton"disabled=true>Play sound</button>
```

JavaScript source code:

```
var ctx;

var soundURL =
 'http://mainline.i3s.unice.fr/mooc/shoot2.mp3';
var decodedSound;

window.onload = function init() {
   // The page has been loaded
   // To make it work even on browsers like Safari, that still
   // do not recognize the non prefixed version of AudioContext
 var audioContext = window.AudioContext|| window.webkitAudioContext;

   ctx = new audioContext();

   loadSoundUsingAjax(soundURL);
   // By default the button is disabled, it will be
   // clickable only when the sound sample will be loaded
   playButton.onclick = function(evt) {
      playSound(decodedSound);
   };
};

function loadSoundUsingAjax(url) {
   var request = new XMLHttpRequest();
   request.open('GET', url, true);
   // Important: we're loading binary data
   request.responseType = 'arraybuffer';

   // Decode asynchronously
   request.onload = function() {
      console.log("Sound loaded");
      // Let's decode it. This is also asynchronous
      ctx.decodeAudioData(request.response,
          function(buffer) { // success
             console.log("Sound decoded");
             decodedSound = buffer;
             // we enable the button
             playButton.disabled = false;
          },
          function(e) { // error
             console.log("error");
          }
      ); // end of decodeAudioData callback
```

```
    };    // end of the onload callback
    // Send the request. When the file will be loaded,
    // the request.onload callback will be called (above)
    request.send();
}

function playSound(buffer){
    // builds the audio graph, then start playing the source
    var bufferSource =ctx.createBufferSource();
58.     bufferSource.buffer = buffer;
    bufferSource.connect(ctx.destination);
    bufferSource.start(); // remember, you can start() a
source only once!
}
```

Explanations:

- When the page is loaded, we first call the `loadSoundUsingAjax` function for loading and decoding the sound sample (*line 16*), then we define a click listener for the button. Loading and decoding the sound can take some time, so it's asynchronous. This means that the call to `loadSoundUsingAjax` will return while the downloading and decoding is still in progress. We can define a click listener on the button anyway, as this one is disabled by default (see the HTML code). Only once the sample has been loaded and decoded, will the button be enabled (*line 42*).
- The `loadSoundUsingAjax` function will first create an`XmlHttpRequest` using the "new version of Ajax called XhR2" (described in detail during week 3). First we create the request (*lines 26-30*): notice the use of '`arrayBuffer`' as a `responseType` for the request. This has been introduced by Xhr2 and is necessary for binary file transfer. Then the request is sent (*line 52*).
- Ajax is an asynchronous process: once the browser receives the requested file, the `request.onload`callback will be called (it is defined in *line 33*), and we can decode the file (an mp3, the content of which must be uncompressed in memory). This is done by calling`ctx.decodeAudioData(file, successCallback, errorCallback).` When the file is decoded, the success callback is called (*line 38-43*). We store the decoded buffer in the variable `decodedSound`, and we enable the button.

- Now, if someone clicks on the button, the `playSound`function will be called (*lines 55-61*). This function builds a simple audio graph: it creates an`AudioBufferSourceNode` (*line 57*), sets its `buffer`property with the decoded sample, connects this source to the speakers (*line 59*) and plays the sound. **Source nodes can only be used once ("fire and forget" philosophy), so to play the sound again, we have to rebuild a source node and connect it to the destination again. This seems strange when you learn Web Audio, but don't worry - it's a very fast operation, even with hundreds of nodes.**

LOADING AND DECODING MULTIPLE SOUNDS: THE BUFFERLOADER UTILITY

The problem: AJax requests are asynchronous

The asynchronous aspect of Ajax has always been problematic for beginners. For example, if our applications use multiple sound samples and we need to be sure that all of them are loaded and decoded, using the code we presented in the earlier example will not work as is. We cannot call:

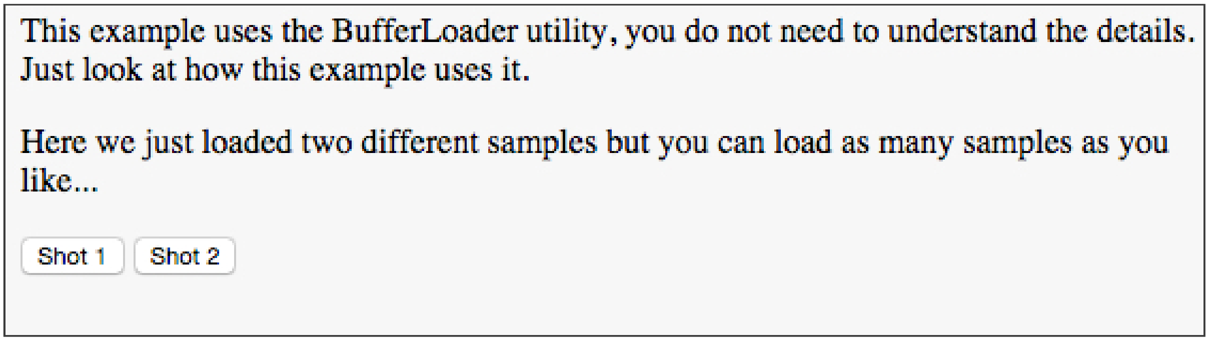
```
loadSoundSample(urlOfSound1);
loadSoundSample(urlOfSound2);
loadSoundSample(urlOfSound3);
etc...
```

As we will never know exactly when all the sounds have finished being loaded and decoded. All these calls will run operations in the background and return instantly.

The BufferLoader utility object: useful for preloading sound and image assets

There are different approaches for dealing with this problem. During the HTML5 Part 1 course, we already presented some utility functions for loading multiple images. Here we use the same approach and we have packaged the code into an object called `BufferedLoader`.

Example at JSBin that uses the BufferLoader utility:


This example uses the BufferLoader utility, you do not need to understand the details. Just look at how this example uses it.

Here we just loaded two different samples but you can load as many samples as you like...

HTML code:

```
<button id="shot1Normal"disabled=true>Shot 1</button>
<button id="shot2Normal"disabled=true>Shot 2</button>
```

JavaScript code extract (does not contain the BufferLoader utility code):

```
var listOfSoundSamplesURLs = [
 'http://mainline.i3s.unice.fr/mooc/shoot1.mp3',
 'http://mainline.i3s.unice.fr/mooc/shoot2.mp3'
];

window.onload = function init() {
  // To make it work even on browsers like Safari, that still
  // do not recognize the non prefixed version of AudioContext

var audioContext = window.AudioContext|| window.webkitAudioContext;

  ctx = new audioContext();
  loadAllSoundSamples();
};

function playSampleNormal(buffer){
  // builds the audio graph and play
  var bufferSource =ctx.createBufferSource();
  bufferSource.buffer = buffer;
  bufferSource.connect(ctx.destination);
21.   bufferSource.start();
    }
     

    function onSamplesDecoded(buffers){
       console.log("all samples loaded and decoded");
       // enables the buttons
       shot1Normal.disabled=false;
       shot2Normal.disabled=false;
       // creates the click listeners on the buttons
       shot1Normal.onclick = function(evt) {
33.        playSampleNormal(buffers[0]);
       };
       shot2Normal.onclick = function(evt) {
          playSampleNormal(buffers[1]);
       };
    }

    function loadAllSoundSamples() {
      // onSamplesDecoded will be called when all samples
43.   // have been loaded and decoded, and the decoded sample will
      // be its only parameter (see function above)

    bufferLoader = newBufferLoader(ctx, listOfSoundSamplesURLs,onSamplesDecoded);
      // starts loading and decoding the files
      bufferLoader.load();
49. }
```

The call to `loadAllSoundSamples()` (*line 13*) will result in a call to `onSamplesDecoded(decodedSamples)`, located at *line 25*, when all the sound sample files have been loaded and decoded. The array of decoded samples is the parameter of the `onSamplesDecoded` function.

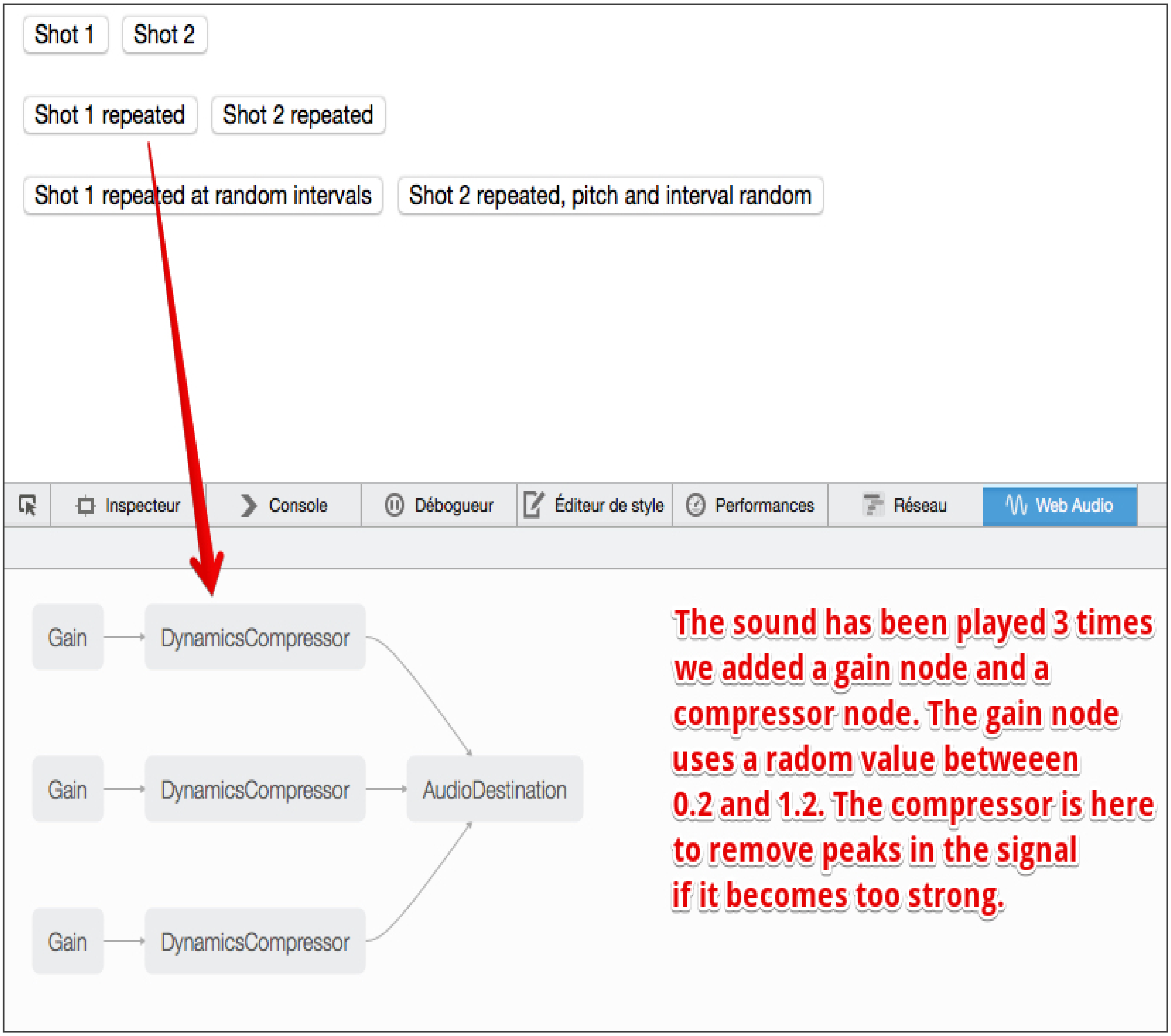
The `BufferLoader` utility object is created at *line 45* and takes as parameters: 1) the audio context, 2) an array with the URLs of the different audio files to be loaded and decoded, and 3) the function that should be called when all files are loaded and decoded.

This function should accept an array as a parameter: the array of the decoded sound files.

For the source of the BufferLoaded object definition, look at the JavaScript tab in the example at JSBin.

A VARIANT OF A PREVIOUS EXAMPLE: PLAY THE TWO SOUND SAMPLES AT VARIOUS PLAYBACK RATES, REPEATEDLY

Example at JSBin:



The sound has been played 3 times we added a gain node and a compressor node. The gain node uses a radom value betweeen 0.2 and 1.2. The compressor is here to remove peaks in the signal if it becomes too strong.

In this example, we added a function (borrowed and adapted from this article on HTML5Rocks):

- `makeSource(buffer, intervalBetweenSounds, nbSoudsToBePlayed, randomIntervalTimeFactor, randomPitchFactor)`

Here is the source code of this function:

```
function makeSource(buffer) {
   // build graph source -> gain -> compressor -> speakers
   // We use a compressor at the end to cut the part of the
signal
   // that would make peaks
   // create the nodes
   var source = ctx.createBufferSource();
   var compressor =ctx.createDynamicsCompressor();
   var gain = ctx.createGain();
   // set their properties
   // Not all shots will have the same volume
   gain.gain.value = 0.2 + Math.random();
   source.buffer = buffer;
   // Build the graph
   source.connect(gain);
  gain.connect(compressor);
  compressor.connect(ctx.destination);
 return source;
}
```

And this is the function that plays different sounds in a row, eventually creating random time intervals between them and random pitch variations:

```
function playSampleRepeated(buffer,rounds, interval, random,
random2) {
   if (typeof random == 'undefined') {
      random = 0;
   }
```

```
    if (typeof random2 == 'undefined') {
        random2 = 0;
    }
    var time = ctx.currentTime;
10.    // Make multiple sources using the same buffer and play in
quick succession.
    for (var i = 0; i < rounds; i++) {
        var source = makeSource(buffer);
        source.playbackRate.value = 1 +Math.random() * random2;

source.start(time + i * interval +Math.random() * random);
    }
}
```

Explanations:

- *Lines 11-15*: we make a loop for building multiple routes in the graph. The number of routes corresponds to the number of times that we want the same buffer to be played. Note that the `random2` parameter enables us torandomize the playback rate of the source node that corresponds to the pitch of the sound.
- *Line 14*: this is where the sound is being played. Instead of calling source.start(), we call source.start(delay), this tells the Web Audio scheduler to play the sound after a certain time.
- The makeSource function builds a graph from one decoded sample to the speakers. A gain is added that is also randomized in order to generate shot sounds with different volumes (between 0.2 and 1.2 in the example). A compressor node is added in order to limit the max intensity of the signal in case the gain makes it peak.

KNOWLEDGE CHECK 1.5.9

Video games often need to play sound samples in rapid sequence, with different effects, pitch, etc. What best practice has been presented in the course?

○ They should be loaded and decoded, before being used.

○ All sound samples must be loaded before being used.