Tutoriel WEB Audio API
Anthony Biga & iliasse Hassala
Etudiants M1 info / MBDS
UNSA
Projet Web 2.0

sources :
Html5rocks
Spécification webAudio W3C

Browsers supportés (Avril 2012) :

- Chrome (version stable ou *Canary*)
- Safari (en faisant un petit bidouillage, *cf "Astuce" en fin de rapport*)

WEB AUDIO API

HTML 5 - JavaScript

Web Audio est une API JavaScript de haut niveau pour le traitement et la génération de son dans des application web en HTML5.

Table des matières :

1.Démarrer avec AudioContext

Pour jouer et gérer du son dans une page HTML5 il faut définir un “contexte audio” c’est à dire une instance de *AudioContext*. La page est associée au contexte grâce à ce code JavaScript :

Code :

```
var context;
window.addEventListener('load', init, false);
function init() {
  try {
    context = new webkitAudioContext();
  }
  catch(e) {
    alert('Web Audio API is not supported in this browser');
  }
}
```

NB : Pour les navigateurs basés sur WebKit, il faut utiliser le préfixe “webkit”, comme avec webkitAudioContext.

2.Charger un son

Pour des sons courts ou de durée moyenne, l'API Audio utilise un AudioBuffer et récupère les fichiers audio avec *XMLHttpRequest*. Selon les navigateurs, les formats supportés varient du WAV au MP3 en passant par l'AAC ou encore l'OGG. Le code suivant montre comment charger un son dans une page :

Code :

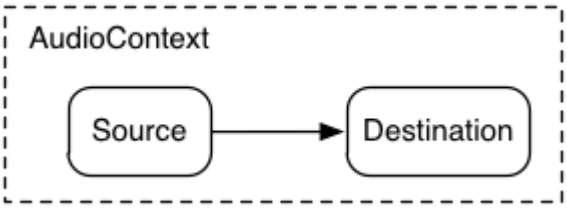
```
var dogBarkingBuffer = null;
var context = new webkitAudioContext();

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

  // Decode asynchronously
  request.onload = function() {
    context.decodeAudioData(request.response, function(buffer) {
      dogBarkingBuffer = buffer;
    }, onError);
  }
  request.send();
}
```

Comme un fichier audio contient des données binaires et non du texte, on définit responseType comme un *ArrayBuffer*.
Une fois le fichier audio reçu, il peut être mis de côté et décodé plus tard ou bien directement avec la méthode decodeAudioData() de *AudioContext*. Cette méthode prend le ArrayBuffer du fichier audio, sauvegardé dans request.response, et le décode de manière asynchrone de façon à ne pas bloquer l'exécution du thread JavaScript principal. Quand decodeAudioData() a fini de s'exécuter, une fonction de call-back est appelée qui fournie les données décodées sous forme d'un AudioBuffer.

3.Jouer un son



Un simple graphe audio

Une fois chargé, le son est prêt à être joué :

Code :

```
var context = new webkitAudioContext();
function playSound(buffer) {
// on crée une source sonnore
  var source = context.createBufferSource();
// on lui dit quel son jouer
  source.buffer = buffer;
// on la connecte à la destination du contexte (le hauts parleurs)
  source.connect(context.destination);
// et on joue le son
source.noteOn(0);
}
```

NB : Cette fonction playSound peut être appelée chaque fois qu'un utilisateur appuie sur une touche de son ou clique sur quelque chose avec sa souris.

La fonction noteOn(time) facilite l'ordonnancement précis des sons joués pour les jeux et autres application à durée critique. Cependant, pour que l'ordonnancement fonctionne correctement, il faut s'assurer que les buffers utilisés ont bien été pré-chargés.

4.Comment rendre l'API Audio plus absraite

Il est préférable de créer un système de chargement plus général qui ne soit pas codé en dur pour un son spécifique. Il y a plusieures approches dont celle présentée ici, à savoir l'utilisation de la classe *BufferLoader*. Le code qui suit est un exemple qui montre comment utiliser cette classe pour créer deux AudioBuffer qui seront joués en même temps dès qu'ils seront chargés :

Code :

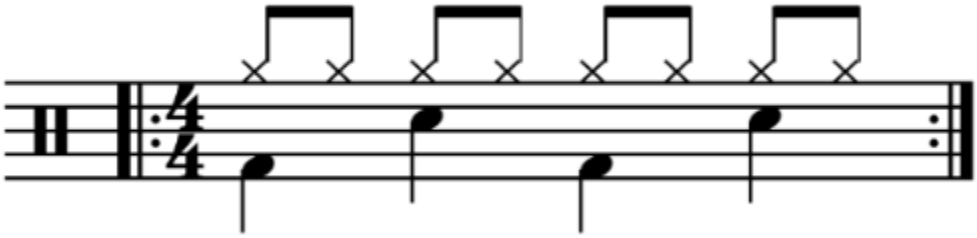
```
window.onload = init;
var context;
var bufferLoader;
function init() {
  context = new webkitAudioContext();
  bufferLoader = new BufferLoader(
    context,
    [
      '../sounds/hyper-reality/br-jam-loop.wav',
      '../sounds/hyper-reality/laughter.wav',
    ],
    finishedLoading
    );

  bufferLoader.load();
}
function finishedLoading(bufferList) {
  // crée deux sources sonnores et les joue ensemble.
  var source1 = context.createBufferSource();
  var source2 = context.createBufferSource();
  source1.buffer = bufferList[0];
  source2.buffer = bufferList[1];
  source1.connect(context.destination);
  source2.connect(context.destination);
```

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

5.Introduire du rythme

L'API Audio laisse le développeur ordonnancer précisément les sons qu'il souhaite jouer. Pour le démontrer, on peut générer une piste rythmique simple. Le kit de batterie le plus connu au monde est probablement le suivant :



Un modèle simple de rythme de batterie rock

dans lequel un charleston est jouée chaque croche, et le kick et la caisse claire sont joués en alternance tous les trois temps, en 4 temps.

En supposant que le charleston, le kick et la caisse claire ont été préalablement chargés, le code pour jouer la partition ci-dessus est simple :

Code :

```
for (var bar = 0; bar < 2; bar++) {
  var time = startTime + bar * 8 * eighthNoteTime;
  // jouer le kick aux temps 1 et 5
  playSound(kick, time);
  playSound(kick, time + 4 * eighthNoteTime);
```

```
        // jouer la caisse claire aux temps  3 et 7
        playSound(snare, time + 2 * eighthNoteTime);
        playSound(snare, time + 6 * eighthNoteTime);

        // jouer le charleston tous les 2 temps.
        for (var i = 0; i < 8; ++i) {
          playSound(hihat, time + i * eighthNoteTime);
        }
      }
```

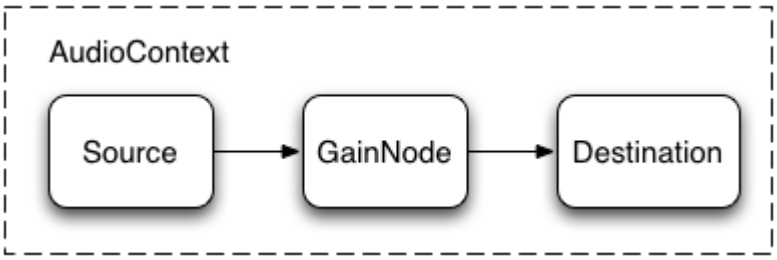
NB : Ici on ne fait qu'une seule répétition au lieu de la boucle infinie que l'on voit sur la partition. La fonction playSound est une méthode qui joue un buffer à un moment précis :

Code :

```
      function playSound(buffer, time) {
        var source = context.createBufferSource();
        source.buffer = buffer;
        source.connect(context.destination);
        source.noteOn(time);
      }
```

6.Changer le volume d'un son

Une des opérations les plus basique consiste à modifier le volume du son que l'on veut jouer. En utilisant l'API Web Audio on peut router une source sonore vers sa destination à travers un *AudioGainNode* dans le but de manipuler son volume :



Audio-graphe avec un noeud de gain

Cette connexion peut être réalisée de la manière suivante :

Code :

```
// créer un noeud de gain.
var gainNode = context.createGainNode();
// connecter la source au noeud de gain.
source.connect(gainNode);
// connecter le noeud de gain à la destination.
gainNode.connect(context.destination);
```

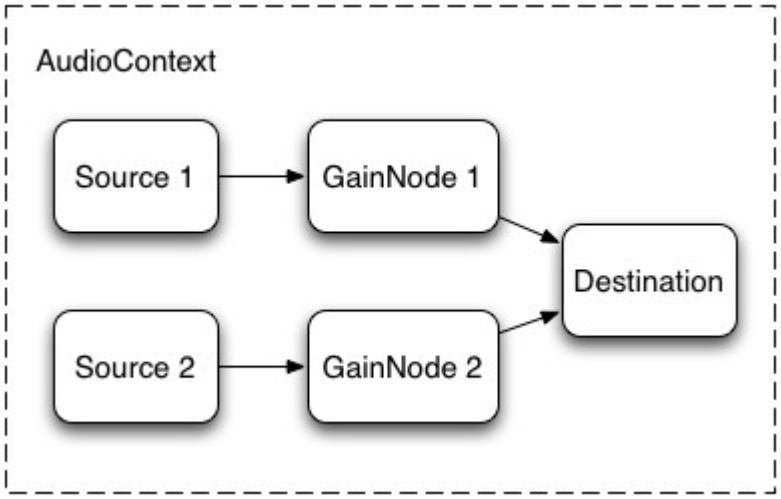
NB: Une fois le graphe mis en place on peut changer le volume en manipulant la valeur du noeud du gain (gainNode.gain.value) :

Code :

```
// réduire le volume.
gainNode.gain.value = 0.5;
```

7.CROSS-FADING entre deux sources sonores

Maintenant, supposons que l'on ait deux sources sonores, chacune produisant un son distinct et que l'on souhaite "mixer" les deux. Un peu à la manière d'un Disc-Jockey. Le schéma suivant illustre bien la situation.



Deux sources audio convergeant vers la même destination après passage dans un amplificateur

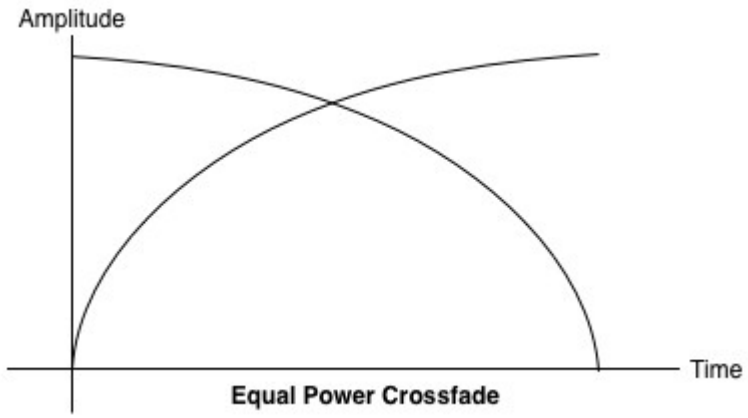
Afin d'implémenter une telle situation, il suffit simplement de créer deux *AudioGainNodes* et connecter les deux sources a ces derniers, ce qui donne quelque chose de semblable à cette fonction :

Code :

```
function createSource(buffer) {
  var source = context.createBufferSource();
  // Create a gain node.
  var gainNode = context.createGainNode();
  source.buffer = buffer;
  // Turn on looping.
  source.loop = true;
  // Connect source to gain.
  source.connect(gainNode);
  // Connect gain to destination.
  gainNode.connect(context.destination);

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

Reste le problème de la différence de volume lorsque l'on "switch" entre les différentes sources (physiquement parlant on pourrait se retrouver dans le cas où : Son + Son = Silence). Le schéma suivant propose une solution à ce problème là :

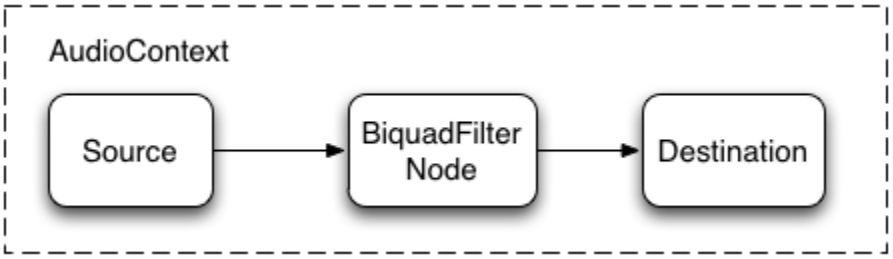


Courbes de gains non linéaires

En Effet, en égalisant les puissances de sortie du point de vue de l'amplitude où les gains apportés ne sont plus linéaires mais légèrement courbés, on minimise considérablement le "plongeon" de son qu'il y avait quand on passait d'un son à l'autre. Ce qui permet d'avoir une sorte de fondu enchaîné entre différentes régions sonores.
D'ailleurs, on pourrait utiliser cette technique si on désirait par exemple faire une application de lecture de mp3 en continu; le fondu enchaîné peut être réalisé en mettant un gain décroissant à la fin de la piste en lecture et un gain croissant juste au début de la prochaine piste à lire.
La Web Audio API fournit un nombre assez considérable de fonctions de traitement de son dans le cas décrit précédemment tel que *linearRampToValueAtTime* et *exponentialRampToValueAtTime*.

NB: Nous avons pris l'exemple de deux sources sonores, mais ceci pourra être généralisé à plusieurs sources à la fois.

8.Comment appliquer un simple filtre audio à une source sonore ?



Un graphe audio avec un filtre biquadratique

L'API Web Audio nous permet "d'aiguiller" le son d'un noeud à un autre très facilement, ce qui nous permet de pouvoir créer beaucoup de noeuds intermédiaires de traitement de son.
Sur le schéma, nous avons choisi de positionner un filtre biquadratique entre la source et la destination. Un tel filtre permet en effet de créer des égaliseurs graphiques (graphic equalizers) et d'autres effets intéressants.
Par ailleurs il existe beaucoup de type de *filtres* dans l'API Web Audio comme *Le filtre passe-bas, passe-haut* et *passe-bande etc.* Et bien sur tous les filtres sont finement *paramétrables*. Paramètres qui permettent de spécifier le *gain*, la (ou les) fréquence a laquelle s'applique le filtre ainsi que le *facteur de qualité*.
Voici un exemple d'implémentation d'un filtre passe bas (ce qui nous permet de n'extraire que les basses fréquences) :

Code :

```
// Create the filter
var filter = context.createBiquadFilter();
// Create the audio graph.
source.connect(filter);
filter.connect(context.destination);
// Create and specify parameters for the low-pass filter.
filter.type = 0; // Low-pass filter. See BiquadFilterNode docs
filter.frequency.value = 440; // Set cutoff to 440 HZ
// Playback the sound.
source.noteOn(0);
```

NB : Les réglages de fréquences doivent prendre en compte l'utilisation d'une échelle logarithmique, puisque l'oreille humaine fonctionne sur le même principe (le "La 4" est à 440 Hz et le "La 5" est à 880 Hz)

Finalement, on peut toujours déconnectez un filtre d'un noeud pour le rediriger vers un autre et changer ainsi dynamiquement le *AudioContext graph* (ceci se fait grâce à l'appel à la méthode *node.disconnect(outputNumber)*).

L'exemple suivant permet de re-router le graphe, pour qu'au lieu de traverser un noeud il se connecte directement à la source.

Code :

```
// Disconnect the source and filter.
source.disconnect(0);
filter.disconnect(0);
// Connect the source directly.
source.connect(context.destination);
```

Astuce : Mac OS X

Web Audio is available in Apple Safari nightly builds. Before running, it must be enabled by going into the "Terminal" application and typing:

```
$ defaults write com.apple.Safari WebKitWebGLEnabled -bool YES
$ defaults write com.apple.Safari WebKitWebAudioEnabled -bool YES
$ defaults write com.apple.Safari
com.apple.Safari.ContentPageGroupIdentifier.WebKit2WebAudioEnabled -bool YES
$ defaults write com.apple.Safari
com.apple.Safari.ContentPageGroupIdentifier.WebKit2WebGLEnabled -bool YES
```