Université Paris 8

Moteurs de Jeu [1]

Nicolas JOUANDEAU
n@up8.edu

septembre 2022

1. A la découverte des fonctions proposées par les moteurs et les jeux possibles.

6 Sons

6.1 Jouer des sons avec `Audio`

En reprenant le programme de la section 1.6, on ajoute deux sons pour des rebonds de la balle sur le sol :

- un son `long-pop-2358` à chaque rebond pair
- un son `soap-bubble-2925` à chaque rebond impair
- un son correspondant à `f.wav` est chargé avec `new Audio("./f.wav")`
- quand la position de la balle est au point de rebond, on joue un son avec `play()` ; pour assurer que le fichier audio associé à `pocAudio1` est suffisament chargé, on réalise les tests lignes 16-17 ; ajouter `pause()` en remettant la propriété `currentTime` à 0 permet de rejouer le son si celui ci n'est pas fini

```
1  let cnv = document.getElementById("myCanvas");
2  let ctx = cnv.getContext("2d");
3  let ballSize = 30, vX = 2.0, hY = 200.0;
4  let posX = 0, posY = 400-1.0*Math.abs(hY*Math.sin(0.0));
5  let pocAudio1 = new Audio("./assets/mixkit-long-pop-2358.wav");
6  let pocAudio2 = new Audio("./assets/mixkit-soap-bubble-2925.wav");
7  pocAudio1.volume = 1.0;
8  pocAudio2.volume = 1.0;
9  let pocAudioId = 0;
10 function draw() { ... }
11 function update_pos() {
12   posX += vX;
13   posY = 400.0-1.0*Math.abs(hY*Math.sin(Math.PI*posX/60));
14   if(posX <= vX || posY >= 399.0) {
15     if(pocAudioId == 0) {
16       if(pocAudio1.readyState == HTMLMediaElement.HAVE_FUTURE_DATA ||
17          pocAudio1.readyState == HTMLMediaElement.HAVE_ENOUGH_DATA) {
18         pocAudio1.pause();
19         pocAudio1.currentTime = 0;
20         pocAudio1.play();
21       }
22       pocAudioId = 1;
23     } else {
24       if(pocAudio2.readyState == HTMLMediaElement.HAVE_FUTURE_DATA ||
25          pocAudio2.readyState == HTMLMediaElement.HAVE_ENOUGH_DATA) {
26         pocAudio2.pause();
27         pocAudio2.currentTime = 0;
28         pocAudio2.play();
29       }
30       pocAudioId = 0;
31     }
32   }
33   if(posX >= cnv.width) posX = 0;
34 }
```

```
let previousTimeStamp = undefined;
let updateTime = 10, elapsed = updateTime+1;
function update(timestamp) { ... }
requestAnimationFrame(update);
```

Les formats wav, mp3, aac et ogg sont supportés ; en fixant la propriété `loop` à `true`, on peut jouer en boucle un son ; pour jouer automatiquement une musique de fond en boucle à la fin de son chargement, on pourra faire comme suit :

```
let anotherAudio = new Audio('f.wav');
anotherAudio.addEventListener("canplaythrough", event => {
  anotherAudio.loop = true;
  anotherAudio.play();
});
```

6.2 Mixer des sons avec l'API `WebAudio`

L'API `WebAudio` permet de manipuler plusieurs sources, de les mixer, d'appliquer des effets et de les visualiser :

- `XMLHttpRequest` permet de charger un son existant
- `AudioContex` permet de définir un graphe de traitement des sons
  - `createBufferSource` déclare un noeud source
  - `createBiquadFilter` déclare un noeud filtre
  - `createGain` déclare un noeud gain
  - `A.connect(B)` connecte `A` vers `B`
- terminer le graphe en la propriété `destination` de l'`AudioContex` courant permet d'entendre le résultat
- il est courant de connecter plusieurs noeuds sur l'`AudioContex` courant
- `A.start()` permet de jouer la source `A`
- `A.stop()` arrête la source `A`
- chaque lecture implique de redéfinir le graphe de traitement des sons (les références sont automatiquement désalloués par le ramasse-miette)

Pour un `audio_ctx` prédéfini, on pourra jouer un son sans option de filtrage comme suit :

```
function playSound(sound_buffer, gain) {
  let source_node = audio_ctx.createBufferSource();
  source_node.buffer = sound_buffer;
  let gain_node = audio_ctx.createGain();
  gain_node.gain.value = gain;
  source_node.connect(gain_node);
  gain_node.connect(audio_ctx.destination);
  source_node.start();
}
```

Pour ajouter une option de filtrage, on peut jouer un son comme suit :

```
function playSound(sound_buffer, filter_on, detune, freq,
                   filter_gain, q, type, gain) {
  let source_node = audio_ctx.createBufferSource();
  source_node.buffer = sound_buffer;
  let filter_node = audio_ctx.createBiquadFilter();
  filter_node.detune.value = detune;
  filter_node.frequency.value = freq;
  filter_node.gain.value = filter_gain;
  filter_node.Q.value = q;
  filter_node.type = type;
  let gain_node = audio_ctx.createGain();
  gain_node.gain.value = gain;
  if(filter_on) {
    source_node.connect(filter_node);
    filter_node.connect(gain_node);
  } else {
    source_node.connect(gain_node);
  }
  gain_node.connect(audio_ctx.destination);
  source_node.start();
}
```

Utiliser le paramètre de la fonction `start` permet d'obtenir deux sources séquencielles (*i.e.* jouant deux sons en séquence) :

```
source.start(audio_ctx.currentTime);
source2.start(audio_ctx.currentTime + sound_buffer1.duration);
```

`numberOfInputs` permet de connaître le nombre d'entrées connectés à un noeud :

```
console.log(audio_ctx.destination.numberOfInputs);
```

Il est également possible de créer des sons à l'aide de fonctions mathématiques avec `OscillatorNode` ; l'effet de réverbération s'obtient avec `ConvolverNode`.

Pour les filtres, on a les propriétés suivantes :

- `detune` définit le décallage des oscillations des sons élémentaires en cent [7], de valeur par défaut 100
- `frequency` définit la fréquence du filtre en Hz, de valeur par défaut 350, de valeur attendue entre 10 et la moitié de la fréquence d'échantillonnage
- `gain` définit le volume du son en dB, de valeur par défaut 0, de valeur attendue entre -40 et $+40$, une valeur négative étant une atténuation, et une valeur positive étant une augmentation du son associé
- `Q` définit le facteur de qualité, de valeur par défaut 1, de valeur attendue entre 0.0001 et 1000 sur une échelle logarithmique

7. unité pour les intervalles musicaux, basée sur une échelle logarithmique par rapport à la fréquence fondamentale d'un son ; un octave est de 1200 cents.

— `type` définit le type du filtre
  — `lowpass` réalise un filtre passe-bas résonnant du second ordre avec une atténuation de 12 dB/octave ; les fréquences inférieures à la valeur `frequency` passent et les supérieures sont atténuées ; `Q` définit le comportement à proximité de `frequency` ; `gain` n'est pas pris en compte
  — `highpass` réalise un filtre passe-haut (symétrique à `lowpass`) ; les fréquences supérieures à la valeur `frequency` passent et les inférieures sont atténuées ; `Q` et `gain` sont similaires au filtre passe-bas
  — `bandpass` réalise un filtre passe-bande du second ordre ; seules les fréquences dans la plage attendue passent ; `frequency` définit le centre de la plage et `Q` définit la largeur de la plage
  — `lowshelf` définit un filtre des basses fréquences ; les fréquences en dessous de `frequency` sont amplifiées ou atténuées, les autres sont inchangées ; `Q` n'est pas pris en compte
  — `highshelf` définit le filtre des hautes fréquences (symétrique à `lowshelf`)
  — `peaking` définit une plage de modification de fréquences ; comme pour `bandpass`, `Q` et `frequency` définissent la plage ; `gain` définit si les fréquences sont atténuées ou amplifiées
  — `notch` définit un filtre coupe-bande, qui est le contraite de `bandpass` ; les fréquences dans la plage attendue ne passent pas
  — `allpass` définit un filtre passe-tout du second ordre ; le déphasage des fréquences est modifié ; `frequency` définit le point de déphasage maximal ; `Q` définit la transition vers la fréquence moyenne ; `gain` n'est pas pris en compte

Pour définir un rythme de batterie, on définit une grille, dans laquelle clicker une fois fait apparaitre un carré gris (son avec un gain de 0.25) et clicker deux fois fait apparaitre un carré noir (son avec un gain de 1) ; on définit une portée composée de 6 mesures ; une mesure est copmposée de 4 temps ; on se limite aux notes des demi-temps ; à intervalle de 3 lignes, on définit le rythme des notes de `hi-hat`, `snare` et `kick` [8].

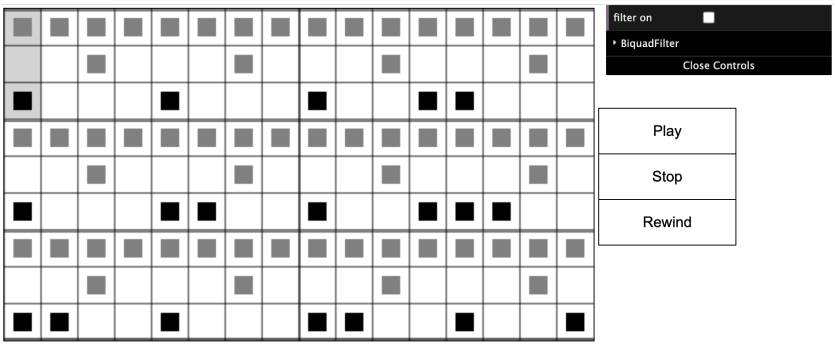

Fig. 66 – Rythme et batterie.

8. Une batterie est classiquement composée de fûts, peaux et cymbales : des toms, une grosse caisse (`kick`), une caisse claire (`snare`) et une charleston (`hi-hat`).

Pour charger les sons, on utilise la classe `BufferLoader`.

```
class BufferLoader {
  constructor(context, urlList, callback) {
    this.context = context;
    this.urlList = urlList;
    this.onload = callback;
    this.bufferList = new Array();
    this.loadCount = 0;
  }
  loadBuffer(url, index) {
    let request = new XMLHttpRequest();
    request.open("GET", url, true);
    request.responseType = "arraybuffer";
    let loader = this;
    request.onload = function() {
      loader.context.decodeAudioData(
        request.response,
          function(buffer) {
            if (!buffer) {
              console.log('error decoding file data: ' + url);
              return;
            }
            loader.bufferList[index] = buffer;
            if (++loader.loadCount == loader.urlList.length)
              loader.onload(loader.bufferList);
          },
        function(error) {
          console.error('decodeAudioData error', error);
        });
    }
    request.onerror = function() {
      console.log('BufferLoader: XHR error');
    }
    request.send();
  }
  load() {
    for(let i = 0; i < this.urlList.length; ++i)
      this.loadBuffer(this.urlList[i], i);
  }
};
```

```
let cnv = document.getElementById("myCanvas");
let cnv_left = cnv.getBoundingClientRect().left;
let cnv_top  = cnv.getBoundingClientRect().top;
let ctx = cnv.getContext("2d");
let nbL = 9;
let nbC = 16;
let posX = 0, posY = 0;
let mouseX = 0, mouseY = 0, mouseIn = false;
let playOn = false;
let partition = [];
for(let i = 0; i < nbL; i+=1) {
  let piste = [];
  for(let j = 0; j < nbC; j+=1) {
    piste.push(0);
  }
  partition.push(piste);
}
let load_at_start = true;
if(load_at_start) {
  for(let i = 0; i < nbL; i+=3)
    for(let j = 0; j < nbC; j+=1)
      partition[i][j] = 1;
  let vol1_line = [1,1,1,1,4,4,4,4,7,7,7,7];
  let vol1_col = [2,6,10,14,2,6,10,14,2,6,10,14];
  for(let i = 0; i < vol1_line.length; i+=1)
    partition[vol1_line[i]][vol1_col[i]] = 1;
  let vol2_line = [2,2,2,2,2,5,5,5,5,5,5,8,8,8,8,8,8,8];
  let vol2_col = [0,4,8,11,12,0,4,5,8,11,12,13,0,1,4,8,9,12,15];
  for(let i = 0; i < vol2_line.length; i+=1)
    partition[vol2_line[i]][vol2_col[i]] = 2;
}
document.getElementById("play").addEventListener("click", playFun);
document.getElementById("stop").addEventListener("click", stopFun);
document.getElementById("rewind").addEventListener("click", resetPos);
function playFun() { playOn = true; }
function stopFun() { playOn = false; }

let audio_ctx = new AudioContext();
let bufferLoader = new BufferLoader( audio_ctx,
  ["./assets/R8/hihat.wav","./assets/R8/snare.wav","./assets/R8/kick.wav"],
  finishedLoading
);
bufferLoader.load();
function finishedLoading() { }
```

```
class FilterNodeParam {
  constructor(on=false) {
    this.on = on;
    this.detune = 100;
    this.frequency = 350;
    this.gain = 0;
    this.q = 1;
    this.type = "allpass";
    }
}
let filter_param = new FilterNodeParam();
let gui = new dat.gui.GUI();
gui.add(filter_param, 'on').name("filter on");
let filter_folder = gui.addFolder('BiquadFilter');
filter_folder.add(filter_param, 'detune').min(10).max(1000).step(1);
filter_folder.add(filter_param, 'frequency').min(10).max(1000).step(1);
filter_folder.add(filter_param, 'gain').min(-40).max(40).step(1);
filter_folder.add(filter_param, 'q').min(0.0001).max(1000);
filter_folder.add(filter_param, 'type', ["lowpass", "highpass", "bandpass",
  "lowshelf", "highshelf", "peaking", "notch", "allpass"]);

function playSound(sound_id, sound_gain) {
  let source_node = audio_ctx.createBufferSource();
  source_node.buffer = bufferLoader.bufferList[sound_id];
  let filter_node = audio_ctx.createBiquadFilter();
  filter_node.detune.value = filter_param.detune;
  filter_node.frequency.value = filter_param.frequency;
  filter_node.gain.value = filter_param.gain;
  filter_node.Q.value = filter_param.q;
  filter_node.type = filter_param.type;
  let gain_node = audio_ctx.createGain();
  gain_node.gain.value = sound_gain;
  if(filter_param.on) {
    source_node.connect(filter_node);
    filter_node.connect(gain_node);
  } else {
    source_node.connect(gain_node);
  }
  gain_node.connect(audio_ctx.destination);
  source_node.start();
}
```

```
function drawLine(xi,yi,xf,yf) {
  ctx.beginPath();
  ctx.moveTo(xi,yi);
  ctx.lineTo(xf,yf);
  ctx.stroke();
  ctx.closePath();
}
function drawSquare(fillColor, x, y, dx, dy) {
  ctx.beginPath();
  ctx.fillStyle = fillColor;
  ctx.fillRect(x, y, dx, dy);
  ctx.closePath();
}
function draw() {
  ctx.clearRect(0, 0, cnv.width, cnv.height);
  drawSquare("lightgray", posX*cnv.width/nbC, posY*cnv.height/nbL,
             cnv.width/nbC, cnv.height*3/nbL);
  for(let i = 0; i < cnv.width; i+=cnv.width/nbC) {
    drawLine(i,0,i,cnv.height);
  }
  for(let i = 0; i < cnv.height; i+=cnv.height/nbL) {
    drawLine(0,i,cnv.width,i);
    if((i%3)==0) drawLine(0,i+2,cnv.width,i+2);
  }
  drawLine(0,cnv.height-1,cnv.width,cnv.height-1);
  drawLine(0,cnv.height-2,cnv.width,cnv.height-2);
  drawLine(cnv.width/2+1,0,cnv.width/2+1,cnv.height);
  let current_C = Math.floor(mouseX/(cnv.width/nbC));
  let current_L = Math.floor(mouseY/(cnv.height/nbL));
  if(mouseIn) {
    drawSquare("red", current_C*cnv.width/nbC, current_L*cnv.height/nbL,
               cnv.width/nbC, cnv.height/nbL);
  }
  for(let i = 0; i < nbL; i+=1) {
    for(let j = 0; j < nbC; j+=1) {
      if(partition[i][j] == 1) {
        drawSquare("gray", 10+j*cnv.width/nbC, 10+i*cnv.height/nbL,
                   cnv.width/nbC-20, cnv.height/nbL-20);
      }
      if(partition[i][j] == 2) {
        drawSquare("black", 10+j*cnv.width/nbC, 10+i*cnv.height/nbL,
                   cnv.width/nbC-20, cnv.height/nbL-20);
      }
    }
  }
}
```

```
function resetPos() {
  posX = 0; posY = 0;
}
function playPartition() {
  for(let i = 0; i < 3; i+=1) {
    if(partition[posY+i][posX] == 1) playSound(i,0.25);
    if(partition[posY+i][posX] == 2) playSound(i,1.0);
  }
}
function updatePos() {
  posX += 1;
  if(posX == nbC) {
    posX = 0;
    posY += 3;
    if(posY >= nbL) resetPos();
  }
}
let previous_time_stamp = undefined;
let update_time = 250, elapsed = update_time+1;
function update(timestamp) {
  if(previous_time_stamp != undefined)
    elapsed = timestamp-previous_time_stamp;
  if(elapsed > update_time) {
    previous_time_stamp = timestamp;
    if(playOn == true) {
      playPartition(); updatePos();
    }
  }
  draw();
  requestAnimationFrame(update);
}
cnv.addEventListener("mousemove", mousemove_fun);
cnv.addEventListener("click", mouseclick_fun);
function mousemove_fun(e) {
  mouseX = e.clientX - cnv_left;
  mouseY = e.clientY - cnv_top;
  mouseIn = true;
  if(mouseX <= 5 || mouseX >= cnv.width-5) mouseIn = false;
  if(mouseY <= 5 || mouseY >= cnv.height-5) mouseIn = false;
}
function mouseclick_fun(e) {
  let current_C = Math.floor(mouseX/(cnv.width/nbC));
  let current_L = Math.floor(mouseY/(cnv.height/nbL));
  partition[current_L][current_C] += 1;
  if(partition[current_L][current_C] == 3)
    partition[current_L][current_C] = 0;
}
requestAnimationFrame(update);
```