
Edinburgh Napier University

## SET08101/SET08702 - Web Tech

---

# Lab 10 - Sound & Vision

---

**Dr Simon Wells**

# 1 Aims

At the end of the practical portion of this topic you will be able to:

- Play existing audio files on your pages
- Create sounds from scratch using the AudioContext
- Create 2D graphics from scratch using the HTML canvas & 2D graphics context

# 2 Activities

## 2.1 Playing an Audio File

First find yourself a suitable annoying audio file to experiment with. I use a one second clown horn[1]. Download it, or else find something else that you like and place it in the folder where you will do today's lab exercises.

The first thing we will do is investigate the use of the <audio> HTML5 element. Create the following file and name it audiocontrol.html

```
<!DOCTYPE html>
<html>
    <body>
        <audio id="my_audio_control" autoplay loop controls>
            <source src="horn.mp3" type="audio/mpeg">
            Your browser does not support the audio element.
        </audio>
    </body>
</html>
```

Now try using different audio files that you have available. Notice how easy it is to play a media file and to give your users shuttle controls for manipulating it. You should experiment with the removing each of the autoplay, loop, and controls attributes to see what the effects of each are for you. The <audio> element can do more than this so you should investigate the associated MDN documentation[2].

You can also manipulate the <audio> element from JS. Let's give that a try now. Create a file called audiocontroljs.html and add the following code:

```
<!DOCTYPE html>
<html>
    <body>
        <audio id="my_audio" autoplay="autoplay" loop controls>
            <source src="horn.mp3" type="audio/mpeg">
            Your browser does not support the audio element.
        </audio>
        <button onclick="toggle_loop()" type="button">Toggle looping</button>

        <script>
            var ao = document.getElementById("my_audio");

            function toggle_loop() {
                if (ao.loop) { ao.loop = false; }
                else { ao.loop = true; }
                ao.load();
            }
        </script>
    </body>
</html>
```

[1] Available from my Github Repo: `https://github.com/siwells/set08101/blob/master/resources/horn.mp3?raw=true`

[2] `https://developer.mozilla.org/en-US/docs/Web/HTML/Element/audio`

In this example We then have added a button to the HTML interface which calls a JS function to toggle the loop attribute fot he <audio> element. In the JS we've then used the <audio> element's ID to retrieve a reference to it so that we can set the value of loop to either true or false depending upon the current state of the <audio> control.

Now try the following:

1. Add buttons to toggle other aspects of the <audio> element such as the controls and mute.
2. Try using other HTML controls to build your own custom user interface for the <audio> element, for example enabling the user to set the time to start playing from (you might need a longer MP3 file than my clown horn for that though. Also investigate using your own buttons to play and pause the audio track.
3. Build your own soundboard web page. This should include a number of buttons and should play a different sound effect when each button is clicked. You might have to download a few different sound files or else record yourself (or others) making appropriate noises.

## 2.2 Audio Synthesis

Audio synthesis is the process of creating sound from first principles through code. Let's try that out now. Create a file called audiocontext.html and add the folowing code:

```
<!DOCTYPE html>
<html>
    <body>
        <script>
            var context = new (window.AudioContext || window.webkitAudioContext)();
            var oscillator = context.createOscillator();
            oscillator.connect(context.destination);
            oscillator.start();
        </script>
    </body>
</html>
```

When you open this in your browser you should hear an annoying tone being played. All we've done is create an AudioContext, which is a graph structure for building sound graphs. We've then added an oscillator to it, using the default setting of the oscillator, and connected the oscillator to a default destination, which is the speakers or other default audio output for your machine. When we start the oscillator the sound is then pumped to the destination and we hear it.

This example would probably be much less annoying if we can control it. Let's add a mute button by creating the audiocontext_mute.html example:

```
<!DOCTYPE html>
<html>
    <body>
        <button onclick="mute()" type="button">MUTE</button>
        <script>
            var context = new (window.AudioContext || window.webkitAudioContext)();
            var gain = context.createGain();
            gain.connect(context.destination);
            gain.gain.setValueAtTime(0, context.currentTime);
            var oscillator = context.createOscillator();
            oscillator.connect(gain);
            oscillator.start();

            function mute() {
                if(gain.gain.value == 0) { gain.gain.setValueAtTime(1, context.
                    currentTime); }
                else {gain.gain.setValueAtTime(0, context.currentTime)}
            }
        </script>
    </body>
</html>
```

Note that I've set this to be muted by default. You'll have to press the button once to hear any sound.

We can also change the sound that is generated by our oscillator. We can set the type, which affects the shape of the waveform, using the 'square', 'sawtooth', and 'triangle' types. We can also set the frequency that is played. Create a new file called audiocontext_freq.html and add the foll0wing code:

```
<!DOCTYPE html>
<html>
    <body>
        <button onclick="mute()" type="button">MUTE</button>
        <script>
            var context = new (window.AudioContext || window.webkitAudioContext)();

            var gain = context.createGain();
            gain.connect(context.destination);
            gain.gain.setValueAtTime(0, context.currentTime);


            var oscillator = context.createOscillator();
            oscillator.type = 'sine';
            oscillator.frequency.value = 220;
            oscillator.connect(gain);
            oscillator.start();

            function mute() {
                if(gain.gain.value == 0) { gain.gain.setValueAtTime(1, context.
                    currentTime); }
                else {gain.gain.setValueAtTime(0, context.currentTime)}
            }
        </script>
    </body>
</html>
```

Experiment with this by trying different values for the oscillator frequency and type. Perhaps experiment with using different HTML elements to control the type and frequency, such as radio buttons for the type and a slider for the frequency.

Specific musical notes are just particular frequencies of sound. Wikipedia has a useful table of frequencies and their musical names[3]. You could use this knowledge to start thinking about how to build web-based versions of musical instruments.

Let's round out this section by looking at how to play a bittune from first principles. Create a file called audiocontext_bittune.html and add the following code:

```
<!DOCTYPE html>
<html>
    <body>
        <script>
            window.onload = function() {
                var audio = new (window.AudioContext || window.webkitAudioContext)
                    ();
                position = 0,
                scale = {
                    b: 233,
                    c: 261,
                    d: 293,
                    e: 329,
                    f: 349,
                    g: 391,
                    A: 440,
                    B: 493,
                    C: 523,
                    D: 587,
                    E: 659,
                    F: 698,
                    G: 783,
                    a: 880
                },
                song = "EE-E-CE-G---g--C-g--e--A-B-BA-gEGaFG-E-CDB--C--g--eA-B-BA-
                    gEGaFG-E-CDB";

```

[3] https://en.wikipedia.org/wiki/Piano_key_frequencies

```
26              setInterval(play, 1000 / 4);
27
28              function createOscillator(freq) {
29                  var attack = 10,
30                      decay = 250,
31                      gain = audio.createGain(),
32                      osc = audio.createOscillator();
33
34                  gain.connect(audio.destination);
35                  gain.gain.setValueAtTime(0, audio.currentTime);
36                  gain.gain.linearRampToValueAtTime(1, audio.currentTime + attack /
                        1000);
37                  gain.gain.linearRampToValueAtTime(0, audio.currentTime + decay /
                        1000);
38
39                  osc.frequency.value = freq;
40                  osc.type = "square";
41                  osc.connect(gain);
42                  osc.start(0);
43
44                  setTimeout(function() {
45                      osc.stop(0);
46                      osc.disconnect(gain);
47                      gain.disconnect(audio.destination);
48                  }, decay)
49              }
50
51              function play() {
52                  var note = song.charAt(position),
53                      freq = scale[note];
54                  position += 1;
55                  if(position >= song.length) {
56                      position = 0;
57                  }
58                  if(freq) {
59                      createOscillator(freq);
60                  }
61              }
62          };
63          </script>
64      </body>
65  </html>
```

Note how we've defined a dictionary containing the note names and frequencies for some common notes. We've also described a "song" using a string. We've then used the setInterval function to play new notes every 250ms. This function calls our play() function which gets the not from our song string and determines the frequency to play. We then call our createOscillator() function which sets up a new oscillator which play for 250ms. We use the setTimeout() function to limit how long our oscillator runs before it is stopped. This is because musical notes usually stop playing after a set length of time.

Now try the following:

1. Build a simple piano that responds to mouseclicks to play the appropriate corresponding sound. You could use buttons as piano keys initially, but you're free to come up with other solutions.

2. Try to build a simple theremin page which uses your touchpad or mouse to alter the volume and pitch of the sound played. You might want to investigate how the HTML multitouch controls can be used to provide a better user experience[4].

3. Build a match the noise game. The interface should have an even number of buttons with noises randomly assigned to the buttons. Your user must click buttons to play sounds, for every pair of sounds that match, they get a point and those buttons become inactive. The winner is the player to match the most sounds by the end.

4. Investigate the sonification options for the audio context. These enables you to produce visualisations that are related to the current sounds being produced. How could you incorporate such a feature into your piano or theremin?

[4] https://developer.mozilla.org/en-US/docs/Web/API/Touch_events/Multi-touch_interaction

## 2.3 Drawing on your webpages with the HTML <canvas> & 2D Graphics Context

Now it's time to satisfy a differnt sense. Instead of our hearing, let's satisfy our vision. Let's make some pretty pictures (after a fashion) from first principles using the HTML <canvas> element and the 2D Graphics context. Create a new file called 2dgraphics.html and add the following code:

```
<!DOCTYPE HTML>
<html>
   <head>
      <style>
         #my_canvas{border:1px solid tomato;}
      </style>
   </head>

   <body>
      <canvas id = "my_canvas" width = "100" height = "100"></canvas>
   </body>
</html>
```

This gives us our canvas, and shows us where it is on the page because of the beautiful tomato coloured border, but it doesn't really do anything. Let's add some boilerplate code for setting up the graphics context, so that we're ready to start drawing. Amend 2dgraphics.html so it looks liek the following:

```
<!DOCTYPE HTML>
<html>
   <head>
      <style>
         #my_canvas{border:1px solid tomato;}
      </style>
   </head>

   <body>
      <canvas id = "my_canvas" width = "100" height = "100"></canvas>
      <script>
        var canvas  = document.getElementById("my_canvas");
        if (canvas.getContext) {
            var ctx = canvas.getContext('2d');
            // Add you drawing code here...
        } else {
            // Do something else if user's browser doesn't
            // support the canvas element, i.e. retrieve
            // an image file as a replacement or a message
        }
      </script>
   </body>
</html>
```

The output from this isn't actually any different from before, but we now have a place where we can start adding some drawing code (indicated by our useful comment line) and a place to output a message for users on browsers that don't support the 2d graphics context. We've also used the canvas.getContext() function to create a new context object which we've stored in the variable 'ctx'.

We can draw into this graphics context. Let's try using the fillRect() function to draw a rectangle. Create a file called rect.html and add the following code:

```
<!DOCTYPE HTML>
<html>
   <head>
      <style>
         #my_canvas{border:1px solid tomato;}
      </style>
   </head>
   <body>
      <canvas id = "my_canvas" width=360 height=240 ></canvas>
      <script>
        var canvas  = document.getElementById("my_canvas");
        var ctx = canvas.getContext('2d');
```

```
        ctx.fillStyle = "tomato";
        ctx.fillRect(25,25,100,100);
      </script>
   </body>
</html>
```

As well as rectangles, we can draw curves and arcs. Create a file called smiley.html and add the following code:

```
<!DOCTYPE HTML>
<html>
   <head>
      <style>
         #my_canvas{border:1px solid tomato;}
      </style>
   </head>
   <body>
      <canvas id = "my_canvas" width=360 height=240 ></canvas>
      <script>
        var canvas  = document.getElementById("my_canvas");
        var ctx = canvas.getContext('2d');
        ctx.strokeStyle = "tomato";
        ctx.lineWidth = 1
        ctx.beginPath();
        ctx.arc(75,75,50,0,Math.PI*2,true);
        ctx.moveTo(110,75);
        ctx.arc(75,75,35,0,Math.PI,false);
        ctx.moveTo(65,65);
        ctx.arc(60,65,5,0,Math.PI*2,true);
        ctx.moveTo(95,65);
        ctx.arc(90,65,5,0,Math.PI*2,true);
        ctx.stroke();
      </script>
   </body>
</html>
```

We can also draw arbitrary lines as well. Let's try that out. Create a files called lines.html and add the following code:

```
<!DOCTYPE HTML>
<html>
   <head>
      <style>
         #my_canvas{border:1px solid tomato;}
      </style>
   </head>
   <body>
      <canvas id = "my_canvas" width=360 height=240 ></canvas>
      <script>
        var canvas  = document.getElementById("my_canvas");
        var ctx = canvas.getContext('2d');
        ctx.strokeStyle = "green";
        ctx.lineWidth = 1;
        for (i=0;i<10;i++){
            ctx.lineWidth = 1+i;
            ctx.beginPath();
            ctx.moveTo(5+i*14,5);
            ctx.lineTo(5+i*14,140);
            ctx.stroke();
        }
      </script>
   </body>
</html>
```

We can build lots of images from first principles just using combinations of lines, curves, and rectangles. The 2D graphics context can do a lot more though[5] and you're only really limited by your creativity

[5]Investigte the MDN documentation here: `https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API`

Now try the following:

1. Combine the three techniques we've seen to draw a childrens style drawing of a house with the sun shining. You might need to research additional features of the 2d graphics context.