Week 7 – Audio and Video

Introduction

For a long time Web designers had to rely on Adobe Flash or other clunky, third party plugins in order to deal with audio and video playback online. HTML5 has changed this. The HTML specification includes new tags: `<audio>` and `<video>`. Browsers that support these tags offer native playback for audiovisual and audio-only media files. For older browsers, the `<audio>` and `<video>` tags offer a 'fall back' – text that is displayed telling the user that their browser is outdated.

There are a number of formats available for media files, and a problem exists with interoperability. Some browsers only support some formats. Luckily, the `<audio>` and `<video>` tags allow us to stipulate multiple formats of an individual file, and the browser will play whichever one it can.

Things have gotten better recently, and as a general rule the majority of modern browsers will support the following file formats:

Audio	Video
.mp3	.mp4
.ogg	.ogv*
	.webm

*Note, videos sometimes use the .ogg extension. This is perfectly okay.

Why is HTML5 better than Flash and what can I do with it?

The benefit of HTML5 over Flash (and any other plugin) is that the user doesn't need any third party plugins, everything is provided by the browser. Regardless of whether or not you think Apple was wrong not to support Flash in iOS, the reality is that is doesn't, meaning that Flash is a big no, no for the millions of iPhone and iPad users out there. Further, the HTML5 `<audio>` and `<video>` tags are easily accessed and controlled by JavaScript (and therefore jQuery). We can also trigger events according to what's happening in the media file. For example, you could make a video disappear from the page once it had finished playing, or you could make a Google Map appear when the video gets 3 minutes and 27 seconds into its playback. Sure, theoretically you could achieve these same things with Flash, but take my word for it; it would be a hassle. HTML5 and jQuery opens some great opportunities for designers to produce rich content and this is what we're hoping you'll do with the major projects.

Adding Media to a Web Page

HTML5 makes it easy to add video and audio to web pages using the `<video>` and `<audio>` tags. We'll look at each one in turn.

<video>

This is example code of how to add video to a web page. In this example we assume that the video files (one encoded as MP4 and the other as OGG) are stored in a subfolder called videos.

```
<video>
  <source src="videos/trailer.mp4">
  <source src="videos/trailer.ogg">
  <p>Your browser doesn't support HTML5. Please upgrade.</p>
</video>
```

Note that there are two `<source>` tags nested inside the `<video>` tag. Each one refers to the video in different formats so that it can be played on all browsers. A browser will move through the `<source>` tags until it finds a format it supports.

Safari is a bit fussy when it comes to the <video> tag; this is why it's important to put the `<source>` tag for the MP4 format first.

The content in the `<p> </p>` tags is the *fallback* text. This is displayed if the user's browser is old and doesn't support the HTML5 `<video>` tag.

As it currently stands, the code above will add a video to the page but it won't play automatically and further, the user won't be able to start the video. The `<video>` tag accepts some attributes that can help us here. For example, the `autoplay` attribute will start the video once the page loads, and the `controls` attribute displays the media player controls. For example:

```
<video controls autoplay>
  <source src="videos/trailer.mp4">
  <source src="videos/trailer.ogg">
  <p>Your browser doesn't support HTML5. Please upgrade.</p>
</video>
```

The `controls` and `autoplay` attributes can be placed in any order, and you don't have to use both of them. For example, you might not want the video to play automatically, but want to provide the controls so that the user can press play when they are ready to watch the video. Also note that unlike other attributes we've used, these don't require a value.

<audio>

Usage of the `<audio>` tag follows the same approach as the `<video>` tag. For example:

```
<audio>
  <source src="audio/song.mp3">
  <source src="audio/song.ogg">
  <p>Your browser doesn't support HTML5. Please upgrade.</p>
</audio>
```

And of course, you can use the `controls` and `autoplay` attributes.

Acting on Events

Now that you've seen how easy it is to add media to your web page, let's get a bit more complex.

There are number of *events* available to the `<video>` and `<audio>` tags. In this context, an *event* is an announcement of information about the media file such as whether it has loaded enough data to play, or whether the media file has finished playing. There are many events available, but today we're just going to look at a couple, `ended` and `timeupdate`.

ended

The `ended` event lets the browser know when the file has finished playing. We can use jQuery to listen out for that event, and when it is triggered, we could for example, make a video disappear or display some "thanks for watching" text.

In order make use of events, we use jQuery to *act on* the event. Basically, this means we tell the browser to *listen* for when the event occurs and then do something.

```
$('video').on('ended', function() {
   $('#thxText).show();
});
```

In this example, we're telling the browser to listen out for when the ended event is announced and then to display a specified HTML element.

timeupdate

The `timeupdate` event is fired approximately four times per second or every 250ms (it differs slightly between browsers). We can use jQuery to listen out for the `timeupdate` event and retrieve the current playback time using the `currentTime` property. It's a little more complex than using the `ended` event, but don't let that stop us. This is useful for syncing actions to media playback. For example, when a video reaches the 40 second mark, you could make some text appear on the page, and when the video reaches the 55 second mark, you could hide that text. In order to achieve such things, we need to be able to retrieve the current playback time of the media file. Here is some sample code:

HTML

```
<video controls autoplay>
   <source src="videos/trailer.mp4">
   <source src="videos/trailer.ogg">
   <p>Your browser doesn't support HTML5. Please upgrade.</p>
</video>
```

The HTML is the same as before.

jQuery

```
$('video').on('timeupdate', function() {
   var playTime = Math.round( $(this).get(0).currentTime );
   if(playTime == 5) {
      $('#txt1').show();
   }
   if(playTime == 10) {
      $('#txt1').hide();
   }
});
```

Looks complex huh? Let's break it down.

```
$('video').on('timeupdate', function() {
   var playTime = Math.round( $(this).get(0).currentTime );
   if(playTime == 5) {
      $('#txt1').show();
   }
   if(playTime == 10) {
      $('#txt1').hide();
   }
});
```

Just like the `ended` example above, this tells the browser to listen out for when the video element fires off the `timeupdate` event, which happens approximately every 250ms.

```
$('video').on('timeupdate', function() {
   var playTime = Math.round( $(this).get(0).currentTime );
   if(playTime == 5) {
      $('#txt1').show();
   }
   if(playTime == 10) {
      $('#txt1').hide();
   }
});
```

Here, we're creating a variable called playTime to hold the value of the current playback position.

```
$(this).get(0).currentTime
```

`$(this)` refers to the `$('video')` in the previous line. The `.get(0)` is a bit difficult to explain, so for now just know that you have to use it. `.currentTime` is a property of the `<video>` element that tells us the current playback position. It returns a very specific fractional number, e.g. 4.17334578. This is hard to work with so we're using `Math.round()` a JavaScript method for rounding up fractions to whole numbers, so 4.17334578 becomes 4 – much easier to work with.

```
$('video').on('timeupdate', function() {
   var playTime = Math.round( $(this).get(0).currentTime );
   if(playTime == 5) {
      $('#txt1').show();
   }
   if(playTime == 10) {
      $('#txt1').hide();
   }
});
```

Next are a couple of IF statements. The IF statement compares the value of the variable playTime against another value. In the first instance it asks "does the value of playTime equal 5?", if it does, then it does whatever is contained between the `{}`, in this case it's telling an HTML element with the ID `txt1` to show. You could, however, get the IF statement to trigger anything, for example, show an image, play a sound, open a pop-up window, load a Google Map – anything you can think of.

It's important to understand that the portion of the code highlighted below is triggered by the `timeupdate` event:

```
$('video').on('timeupdate', function() {
   var playTime = Math.round( $(this).get(0).currentTime );
   if(playTime == 5) {
      $('#txt1').show();
   }
   if(playTime == 10) {
      $('#txt1').hide();
   }
});
```

So basically, every 250ms we retrieve the current playback position (using `$(this).get(0).currentTime`) and round it up to a whole number (using `Math.round()`) and assign it to a variable called `playTime`.

The value of `playTime` is then compared against set values in the IF statement every 250ms and when there's a match, it run whatever code is specified by the IF statement.

It will take time to get your head around how this code works, but we're happy to get through it over and over until you understand it.