8 Video and audio: playing media in the browser

This chapter covers

- Navigating the cross-browser and cross-device issues inherent in video
- Converting between different audio and video formats
- Controlling video playback
- Performing video post-processing in the browser using the `<canvas>` element
- Integrating video playback with other content

Clearly, the web is about more than text, but until HTML5 came along we had no built-in way to play audio and video in the HTML standard. Instead, browsers had to depend on third-party applications known as plug-ins.

Not so today. The web is increasingly being used as a replacement for traditional broadcast media. Services like Netflix, YouTube, Spotify, last.fm, and Google Music seek to replace your DVD and CD collections with online players. With HTML5, video and audio become first-class citizens of web content. Rather than handing responsibility for playing media to a third-party application, it's played within the browser, allowing you to control and manipulate media from within your web application.

In this chapter you'll learn to use HTML5's Media Element Interface while building a video telestrator jukebox. A telestrator, made famous by U.S. football coach and announcer John Madden, allows the user to draw directly onto a playing video; the term comes from television sports broadcasting (television + illustrate = telestrate).

Why build the video telestrator jukebox?

These are the benefits:

- You'll learn to use the `<video>` element to add a video to a web page.
- You'll see how to control video playback with JavaScript using Media Element Interface.
- You'll discover how to support different browsers with different file formats using the `<source>` element.

As you move through the chapter, you'll do the following:

- Build the basic jukebox framework
- Add videos to the web page with HTML5
- Use the HTMLMediaElement interface to load and play videos based on user selection
- Attach event handlers to provide user feedback, enable UI options, and start playback
- Use the `<source>` element to provide multiple videos in different formats to support all browsers
- Control video from JavaScript with the HTMLMediaElement interface
- Combine playing video with other web content

We'll show you the application and help you get your prerequisites in order, and then we'll get you started building the basic video player.

8.1 Playing video with HTML5

Placing a video in HTML5 markup is simple, and no more complex for any given browser than placing an image. In this section you'll take full advantage of the built-in browser support to build the simplest possible video jukebox.

We'll show you what the finished product will look like and help you get your prerequisites aligned. Next, you'll lay the application's basic framework and then use the `<video>` element to add videos to the web page.

8.1.1 Application preview and prerequisites

The sample player you'll be building in this chapter is shown in figure 8.1.

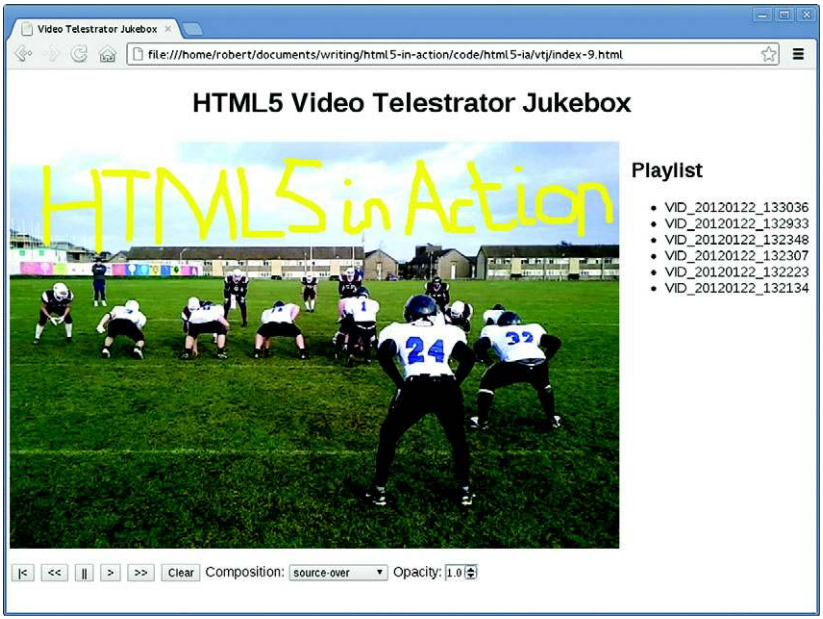

Figure 8.1 The finished telestrator jukebox application, showing a video, some artistic telestration, a playlist of videos to choose from, and, underneath the video, a toolbar for controlling the playback

The figure shows the four main components of the player:

- The video itself, showing American football action
- Some artistic telestration saying "HTML5 in Action"
- A playlist of videos to choose from on the right side
- A toolbar to control the playback below the video

WHICH BROWSER TO USE?

For this section please use Chrome, Safari, or Internet Explorer. For the time being you'll have to avoid Firefox and Opera because of the cross-browser video file format issues. We'll discuss these issues, and perform a few tricks to make everything work in Firefox and Opera, in section 8.1.3.

`<video>/<audio>` **elements**	3	3.5	9	10.5	4.0

PREREQUISITES

Before you begin, download the set of sample videos from this book's website and the latest version of jQuery from http://jquery.com/. Put the videos in a directory of the same name in your working directory, and place jQuery in the working directory itself.

You'll also need the requestAnimationFrame polyfill from https://gist.github.com/1579671 for the later sections. The code at that URL will go in the script section when you start animating in section 8.4.1.

With those preliminaries out of the way, you're ready to build the framework.

8.1.2 Building the basic jukebox framework

Listing 8.1 shows the framework around which you'll be building the application. It creates a simple layout and has placeholders for the video player and the playlist, the major components you'll be adding in the later sections.

Create a new HTML page in your working directory called index.html, with the following listing as its contents.

Listing 8.1 index.html—Basic jukebox layout

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Video Telestrator Jukebox</title>
    <script src="jquery-1.8.2.min.js"></script>
    <script src="raf-polyfill.js"></script>
    <style>
        body {
            font-family:  sans-serif;
            border: 0;
            margin: 0;
            padding: 0;
        }
        header {
            text-align: center;
        }
        #player {
            display: table;
            width: 100%;
            padding: 4px;
        }
        #player > div, #player > nav {
            display: table-cell;
            vertical-align: top;
        }

        #player canvas {
            display: block;
        }
        #player menu, #player label {
            display: inline-block;
            padding: 0;
        }
        input[type=number] {
            width: 36px;
        }
    </style>
```

Latest version of jQuery.

requestAnimationFrame polyfill from https://gist.github.com/1579671.

Basic CSS to lay everything out.

```
</head>
<body>
    <header>
        <h1>HTML5 Video Telestrator Jukebox</h1>
    </header>
    <section id="player">
        <div>
            <!-- The video will appear here-->
        </div>
        <nav>
            <h2>Playlist</h2>
            <ul>
                <!-- The video playlist will appear here-->
            </ul>
        </nav>
    </section>
</body>
</html>
```

You'll add a <video> element here in section 8.1.3.

You'll add a playlist here in section 8.2.

Now, with the foundation laid, let's get to the fun parts of the application by adding a video to the page.

8.1.3 Using the video element to add videos to web pages

The goal in designing HTML5's `<video>` element was to make the embedding of video within a web page as straightforward as embedding an image. Although you'll encounter additional complexities due to video file formats being more feature-rich than image formats, the design goal has been attained. Figure 8.2 shows the `<video>` element applied in Google Chrome.

Core API

The next listing shows all of the code required to display the video in figure 8.2. As you can see, it's not complicated. Insert this code in place of the first comment in listing 8.1, and refresh the page to reproduce figure 8.2.

Figure 8.2 Basic HTML5 video player in Chrome

Listing 8.2 index.html—Embed a video

```
<video src="videos/VID_20120122_133036.mp4"
       controls
       width="720" height="480">
    Your browser does not support the video element, please
    try <a href="videos/VID_20120122_133036.mp4">downloading
    the video instead</a>
</video>
```

Show the standard play/pause/fast forward controls to the user.

The src attribute specifies the video to display, like the <img> element.

The width and height don't have to match the video—the browser will scale everything to fit, as with images.

Browsers that don't support the <video> element will display the fallback content.

Core API

You used four attributes, `src`, `controls`, `width`, and `height`, in the code in listing 8.2. Table 8.1 summarizes those attributes; for a full list of attributes see appendix B.

Table 8.1 Media element attributes

Attribute	Description
`src`	The video to play.
`controls`	A Boolean attribute. If you add it, the browser will provide a standard set of controls for play/pause/seek/volume, and so on. If you leave the attribute out, your code has to control the player (see section 8.3.2).
`width`	The width of the media (video only).
`height`	The height of the media (video only).

For your application, displaying a single video isn't enough. You need more videos and the ability to switch between them and control their playback in response to user commands. To do this you'll need to learn about the HTMLMediaElement interface—a collection of attributes and functions for both `<video>` and `<audio>` elements, which can be used to start playing the media, pause the media, and change the volume, among other things. We'll tackle that in the next section.

Where's the audio?

Perhaps you've already noticed, but in this chapter you'll be considering and using the `<video>` element rather than the `<audio>` element. This isn't because the `<audio>` element is less important (it isn't) or because it's more complex (it's not) but because this is a book. Although a book may not be an ideal medium for presenting moving pictures, it's an even worse one for invisible sound. But both elements share a single API, the HTMLMediaElement interface, and it's this API that's the focus of this chapter. The only differences between the `<audio>` and `<video>` elements are related to visual properties. The `<video>` element allows you to specify a width and a height for the media, the `<audio>` element does not.

Figure 8.3 A video playing in IE9 selected from the playlist. The videos have been taken directly off of author Rob Crowther's mobile phone, default names included.

8.2 Controlling videos with the HTMLMediaElement interface

Now that you have a video playing, let's start implementing the jukebox feature by allowing users to select from a list of videos, which will appear alongside the `<video>` element (figure 8.3).

Over the next two sections you'll work through five steps, writing code that allows you to do the following:

- Step 1: Load a list of videos.
- Step 2: Start a video when selected.
- Step 3: Change between videos.
- Step 4: Use event handlers to handle the changing of video in greater detail.
- Step 5: Provide multiple video formats to support all browsers.

As we mentioned, in this section you'll be making use of the HTMLMediaElement interface from JavaScript; as usual with HTML5, the markup only gets you so far. Most of the interesting stuff is done with JavaScript!

Step 1: Load a list of videos

First, let's hardcode a list of videos into the playlist and hook up everything so that when a user clicks a video it starts playing. Listing 8.3 shows the markup for the playlist; insert it in place of the second comment placeholder in listing 8.1. In a real application you'd almost certainly be generating this list dynamically, but we're going to avoid requiring backend code in this chapter.

Listing 8.3 index.html—Markup for the video playlist

```
<h2>Playlist</h2>
<ul class="playlist">
    <li>VID_20120122_133036.mp4</li>
    <li>VID_20120122_132933.mp4</li>
    <li>VID_20120122_132348.mp4</li>
    <li>VID_20120122_132307.mp4</li>
    <li>VID_20120122_132223.mp4</li>
    <li>VID_20120122_132134.mp4</li>
</ul>
```

Slot this code in the placeholder section in listing 8.1.

The videos listed are available in the book's code download.

STEP 2: START A VIDEO WHEN SELECTED

In order to start a video when the user clicks one of the list items, you'll need to know one property and one method of the HTMLMediaElement interface, both of which are summarized in table 8.2.

Table 8.2 HTMLMediaElement interface

Attribute/method	Description
`.src`	Read/write, reflects the value of the `src` attribute; use it to select a new video.
`.play()`	Start playing the current media.

STEP 3: CHANGE BETWEEN VIDEOS

Core API

You'll also need the `change_video` function, shown in the next listing. As you can see, it uses both the `src` property and the `play()` method to change the video being played. Include the listing in a `script` block at the end of your code's `head` section.

Listing 8.4 index.html—Handling the user clicking the playlist

```
function change_video(event) {

    var v = $(event.target).text().trim();

    var p = $('#player video:first-of-type')[0];

    p.src = 'videos/' + v;

    p.play();

}

$(document).ready(

    function() {

        $('.playlist').bind('click', change_video);

    }

)
```

The function that handles the click events on the playlist.

The video name is the text content of the clicked-on item; if you want a more user-friendly interface, you could put in a more readable text label and have the filename on a data-* attribute.

Get a reference to the <video> element.

Set the src value to the new filename.

Start playing the file.

Bind the handler to the click event of the playlist.

STEP 4: USE EVENT HANDLERS TO HANDLE THE CHANGING OF VIDEO IN GREATER DETAIL

In the previous code, the `src` of the `<video>` element is set, and the `play()` method is called immediately. This works well because all of the videos are relatively small and everything is being loaded off the local disk. If you had a much larger video, it's likely that not enough of it will have loaded to start playback if the `play()` method is called immediately, leading to an error. A more reliable approach would be to wait until the video is loaded before starting to play. The HTMLMediaElement interface includes a number of events that fire as the media is loading. The events fired during the loading of a media file are listed in table 8.3 (all of them will fire during the loading of the media).

Table 8.3 Media element events

Event	Occurs when
`loadedmetadata`	The browser has determined the duration and dimensions of the media resource and the text tracks are ready.
`loadeddata`	The browser can render the media data at the current playback position for the first time.
`canplay`	The browser can resume playback of the media but estimates that if playback were to be started, the media couldn't be rendered at the current playback rate up to its end, without having to stop for further buffering of content.
`canplaythrough`	The browser estimates that if playback were to be started, the media could be rendered at the current playback rate all the way to its end, without having to stop for further buffering.

If you were loading a large media file across the network, then you'd have time to display a notification to the user as each of these events occurred. In this section you'll bind event listeners to each of these events and start the playback on `canplaythrough`. But first, let's look at the network-related information available through the HTMLMediaElement interface.

DETERMINING THE STATE OF MEDIA RESOURCES WITH .NETWORKSTATE AND .READYSTATE

The HTMLMediaElement interface includes two useful properties that allow you to determine the state that the media resource is in: `.networkState` and `.readyState`. In a real application you could use the information provided by these properties to give visual feedback about the state of the loading media resource; for example, a progress bar or a loading spinner. Table 8.4 lists the values each property can assume. The `.networkState` is similar to the `.readyState` property on the request object in an `XMLHTTPRequest` and the media `.readyState` corresponds closely to the events listed in table 8.3.

Table 8.4 HTMLMediaElement interface properties and values

Property/values	Description
`.networkState`	Returns the current network state of the element; the value returned is one of the four shown next.
`NETWORK_EMPTY`	Numeric value: `0` (no data yet).

Table 8.4 HTMLMediaElement interface properties and values *(continued)*

Property/values	Description
NETWORK_IDLE	Numeric value: 1 (the network is temporarily idle).
NETWORK_LOADING	Numeric value: 2 (the network is currently active).
NETWORK_NO_SOURCE	Numeric value: 3 (no source has been set on the media element).
.readyState	Returns a value that expresses the current state of the element, with respect to rendering the current playback position.
HAVE_NOTHING	Numeric value: 0 (no data has yet been loaded).
HAVE_METADATA	Numeric value: 1 (enough data has loaded to provide media metadata).
HAVE_CURRENT_DATA	Numeric value: 2 (enough data is available to play the current frame, but not enough for continuous streaming).
HAVE_FUTURE_DATA	Numeric value: 3 (enough data is available to play several frames into the future).
HAVE_ENOUGH_DATA	Numeric value: 4 (enough data is available and continuing to become available that the media can be streamed).

PLAYING VIDEO ON THE CANPLAYTHROUGH EVENT

The next listing shows a simple example of how to use the HTMLMediaEvent interface events and investigate the `networkState` and `readyState`. Insert this code in place of the `$(document).ready` part of listing 8.4.

Listing 8.5 index.html—Capturing HTMLMediaElement interface events

```
function play_video(event) {
    event.target.play();
}
function log_state(event) {
    console.log(event.type);
    console.log('networkState: ' + event.target.networkState);
    console.log('readyState: ' + event.target.readyState);
}
$(document).ready(
    function() {
        $('.playlist').bind('click', change_video);
        var v = $('#player video:first-of-type')[0];
        v.addEventListener('loadedmetadata', log_state);
        v.addEventListener('loadeddata', log_state);
        v.addEventListener('canplay', log_state);
        v.addEventListener('canplaythrough', log_state);
        v.addEventListener('canplaythrough', play_video);
    }
)
```

You'll use this function to start playing the video as soon as it hits the canplaythrough event; this replaces p.play() in listing 8.4. (This is functionally equivalent to adding the autoplay attribute.)

This generic function will log some information about each event as it fires.

Bind all four events to the log_state function.

TRY IT OUT

Apart from the video playing automatically, the previous listing shouldn't work any differently from listing 8.4, which allowed you to switch between videos. But if you open

up your browser's console, you should see output similar to that shown in the following listing (exact values may vary from browser to browser).

Listing 8.6 Console output from listing 8.5

```
loadedmetadata
networkState: 1
readyState: 4
loadeddata
networkState: 1
readyState: 4
canplay
networkState: 1
readyState: 4
canplaythrough
networkState: 1
readyState: 4
```

Remember that `networkState: 1` is `NETWORK_IDLE` and `readyState: 4` is `HAVE_ENOUGH_DATA`. With all of the videos on local disk you shouldn't expect too much else, although you may see a `networkState` of `2` on IE. If you have some larger videos online, you should see some different values in each event.

PROGRESS CHECK!

If you've been following along in Chrome, Safari, or IE9 as we recommended at the start of this chapter, you should now have a simple interface, which allows you to click a list of videos and see them play. Figure 8.4 shows what you should be seeing; compare your code to the file index-2.html in the chapter's code download if you're having any problems.

Figure 8.4 What your app should look like in the browser at this point

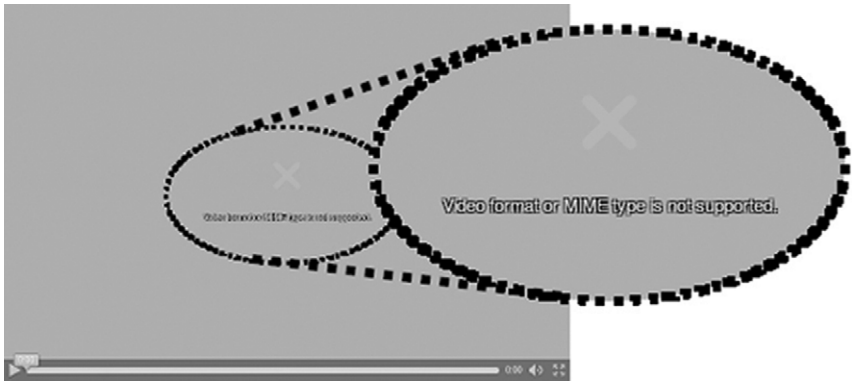

Figure 8.5 An MP4 video in Firefox, where video format or MIME type isn't supported

USING FIREFOX OR OPERA?

If you've tried out the page in Firefox or Opera, you've probably seen a gray screen similar to the one in figure 8.5, which says "Video format or MIME type is not supported."

The issue illustrated in figure 8.5 is that neither Firefox nor Opera supports the MP4 video format even though they support the `<video>` element itself.[1] But the `<video>` and `<audio>` elements provide a workaround for this issue: It's possible to specify multiple media files by using the `<source>` element.

8.3 Specifying multiple formats with the <source> element

Each `<video>` element can have multiple `<source>` elements as children. Each `<source>` specifies a video, and the browser tries each one in turn and uses the first video format it can support. Figure 8.6 shows the same video player in Firefox we showed you earlier after `<source>` elements have been added, instead of using the `src` attribute.

STEP 5: PROVIDE MULTIPLE VIDEO FORMATS TO SUPPORT ALL BROWSERS

Core API

Now let's implement. The next listing shows the new markup for the `<video>` element, using child `<source>` elements. Insert the code in place of the existing `<video>` element in your working file.

Listing 8.7 index.html—Adding the `<source>` element

```
<video controls
 width="720" height="480">
    <source src="videos/VID_20120122_133036.mp4"
            type="video/mp4">
    <source src="videos/VID_20120122_133036.webm"
            type="video/webm">
    Your browser does not support for video element, please
    try <a href="videos/VID_20120122_133036.mp4">downloading
    the video instead</a>
</video>
```

The original .mp4 video.

A version of the video in .webm format.

[1] Recent versions of Firefox will play MP4 videos on Windows using the support available in the OS.

Figure 8.6 `<video>` element in Firefox with multiple sources

CODE CHECK!

This is a good time to stop and check your progress in the browser. You can find the code to this point in the build in the code download, in a file named index-3.html. Compare your index.html code with that code if you have any problems.

8.3.1 Discovering which video is playing with .currentSrc

With the new code, Firefox will now load the video it's able to play. This does introduce a problem for your jukebox feature. Before, you were able to set the `.src` property to change the video, but now you need to set the `.src` differently depending on what video file the browser selected to play. Unfortunately, you can't replace all of the child `<source>` elements with a new set; to change the playing video you have to set the `.src` property.

Core API

To solve this problem you need to know about another property of the HTMLMediaElement interface: `.currentSrc`. This property tells you the filename of the currently selected media.

Because all of your video files are consistently named, you can remove the file extension for all of the `<li>` elements in the playlist (do this now). Instead of getting the complete filename from the `<li>` elements, the `change_video` method can copy the file extension from the `.currentSrc` property and use that to compose the filename of the selected video. The following listing shows the updated `change_video` function, which used this approach; use it to replace the existing one in your file.

Listing 8.8 index.html—Using `currentSrc` to determine the video type

```
function change_video(event) {
    var v = $(event.target).text().trim();
    var p = $('#player video:first-of-type')[0];
    var ext = p.currentSrc.slice(
                  p.currentSrc.lastIndexOf('.'),
                  p.currentSrc.length);
    p.src = 'videos/' + v + ext;
}
```

The playlist should now contain only extension-less entries like VID_20120122_132933.

Slice the file extension from the value of currentSrc starting at the last period.

Combine the file extension with the name to set the new source.

A workaround for IE9's currentSrc bug

The code in listing 8.8 is straightforward, but you may find that it doesn't work properly in IE9. The problem is a bug in IE9: Once a `<source>` element is added, it immediately takes priority over the `src` attribute and the `currentSrc` property of the `<video>` element. This means that if you run the app in IE9, then instead of selecting a new video when you click the playlist, you'll see the first video repeated.

Another limitation of IE9 is that updating `<source>` elements with JavaScript has no effect. If you want to update the playing video in IE9 when you've used `<source>` elements, then the only workable solution is to replace the entire `<video>` element. The following snippet shows just such an approach:

```
function change_video(event) {
    var v = $(event.target).text().trim();
    var vp = $('#player video:first-of-type');
    var p = vp[0];
    var ext = p.currentSrc.slice(
                  p.currentSrc.lastIndexOf('.'),
                  p.currentSrc.length);
    var nv = $('<video controls src="videos/' + v + ext + '" ' +
     'width="720" height="480">' +
     'Your browser does not support the video element, please ' +
     'try <a href="videos/' + v + ext+ '">downloading ' +
     'the video instead</a></video>');
    vp.parent().append(nv);
    vp.remove();
    nv[0].play();
}.
```

For this workaround you'll need a reference to both the actual <video> element and the jQuery object.

Add the new <video> element alongside the current one.

Remove the current <video> element, leaving only the new one.

Instead of updating currentSrc, create a new <video> element with the correct src attribute.

Fortunately this bug is fixed in IE10. Because of this, and to avoid the code complexity getting in the way of learning about the APIs, not to mention that this approach will create new issues in other browsers (which will require further workarounds), the rest of the code in this chapter will ignore this issue. If you're using IE9, then please check the code download files for versions that have been fixed to work in IE9 (they have IE9 in the filename).

You now have a working video jukebox, but you probably still have questions:

- What are these different video formats such as .mp4 and .webm?
- How many different formats do I need to provide to support all browsers?
- If I don't have a particular video in a certain format, how can I convert between them?

We'll discuss changing video formats in the next section. Before we do, we want to answer the first two questions by looking at which browsers support which video and audio formats; table 8.5 summarizes this information.

Table 8.5 Browser video and audio format support

Video formats/ codecs						For broad desktop support, you should provide at least two versions of your media.
MPEG-4/H.264	3	~	9	~	3.2	For video, your best bet is to provide MPEG-4/H.264 and WebM/VP8, at minimum, to cover all current browsers.
Ogg/Theora	3	3.6	~	10.5	*	
WebM/VP8	6	4	*	10.6	*	

* IE and Safari will play additional formats if users install the codec within Windows Media Player or QuickTime, respectively. Currently there's no compatible Ogg/Theora codec for Windows.

Audio formats/ codecs						
MP3	3	~	9	~	3.2	For audio, we recommend that you provide MP3 and Ogg, at minimum, to cover all current browsers.
AAC	3	~	9	~	3.2	
Ogg	3	3.6	~	10.5	*	
WAV	3	3.6	~	10.5	3.2	

* Safari will play additional formats if users install the codec within QuickTime.

As you can see, no single format is universally adopted across all browsers. For broad desktop support, you need to provide at least two versions of your media: for video at least WebM/VP8 and MPEG-4/H.264, for audio MP3 and OGG.

Media format support is something of a contentious issue in the HTML5 world. The sidebar "Why doesn't HTML5 mandate a format that all browsers support?" explains why.

Why doesn't HTML5 mandate a format that all browsers support?

Initially, the HTML5 specification mandated the Ogg/Theora video format. This seemed like a good choice because it's an open source format and the codec is royalty free. But Apple and Microsoft refused to implement Ogg/Theora, preferring instead the MP4/h.264 combination. MPEG LA, LLC, administers MP4/h.264 and

(continued)

sells licenses for encoders and decoders on behalf of companies that hold patents covering the h.264 codec. (Apple and Microsoft are two such companies.)

Supporters of h.264 argue that Ogg/Theora is technically lower quality, has no hardware support (important on battery-powered devices with low-end CPUs), and is more at risk from patent trolls because the obvious way to make money out of infringers is to sue them, whereas submarine patents affecting h.264 can be monetized through MPEG LA.

Supporters of Ogg/Theora argue that the openness of the web requires an open video format. Mozilla couldn't distribute its source code if it contained an h.264 decoder because then everyone who downloaded the code would require a license from MPEG LA. Google avoided this issue by splitting its browser into free parts (the open source Chromium project) and closed parts.

Because the vendors were divided on which format to make standard, and because one of the goals of HTML5 is to document the reality of the implementation, the requirement for supporting any particular codec was removed from the specification. This isn't without precedent in the HTML world—the `<img>` element doesn't specify which image formats should be supported. We can see some light at the end of the tunnel: Google subsequently released the WebM format as open source with an open license. As the owner of the number-one video site on the web, YouTube, and a provider of the Android mobile OS, it's well-positioned to overcome h.264's commercial advantages.

8.3.2 Converting between media formats

For practical purposes, what you need to know is how to convert a video in one of the supported formats to a different format. A tool called a *transcoder* can convert between different container formats and encodings. There are several online and downloadable tools that convert individual media files; several are listed in the links and resources in appendix J. But for batch processing large numbers of files you'll need to use a command-line tool. Appendix H explains how to use `ffmpeg` to transcode the video files used in this chapter.

You're at the point where you can play a video in every browser that supports the `<video>` element, thanks to the `<source>` element. You also know which video formats you need to provide to support which browsers. Now it's time to create the telestrator feature, which will let you draw directly onto the playing video.

8.4 Combining user input with video to build a telestrator

As we mentioned earlier, the telestrator allows the user to draw directly on a playing video to illustrate the action to the television audience. To create this feature in your application, you'll need a way to combine the video with other image data. For this you'll use the `<canvas>` element. You should be familiar with Canvas from chapter 6.

In that chapter you learned about the drawing capabilities of Canvas to create an interactive game. In this chapter you'll concentrate on the general-purpose, image data-manipulation features to combine images and other content with a video feed.

In this section, you'll learn

- How to use the `<canvas>` element to play a video
- How to create controls for video playback (because the `<canvas>` element renders the video image data, not the `<video>` element)
- How to combine the video on the canvas with other content, such as images
- How to perform basic image-processing using the `<canvas>` element
- How to capture the user's drawings (telestrations) and add them to the video during playback

Your work on the telestrator will happen in three groups of steps:

Group 1: Playing video through a `<canvas>` element	Group 2: Manipulating video as it's playing	Group 3: Building the telestrator feature
▪ Step 1: Add the `<canvas>` element. ▪ Step 2: Grab and display image data. ▪ Step 3: Add markup for and implement video player controls.	▪ Step 1: Add a frame image to the video. ▪ Step 2: Adjust how the frame and video combine on the canvas. ▪ Step 3: Adjust the opacity of the video. ▪ Step 4: Grayscale the video being played back.	▪ Step 1: Capture mouse movement. ▪ Step 2: Display the captured path over the video. ▪ Step 3: Add a "clear" button so users can remove telestrations and start again.

Let's start with how to play video through the `<canvas>` element.

8.4.1 Playing video through the <canvas> element

The first requirement is to be able to modify the video as it's being played back. You could do this by layering elements on the page and hiding and showing things at the required time. If you were stuck using plug-ins to render the video, that would be your only option for modifying the video from HTML. But the `<video>` element makes its data available as images. You can access each frame of the video as it's ready and treat it as image data. It's then quite straightforward to use the `<canvas>` element to grab that image data and display it.

STEP 1: ADD THE <CANVAS> ELEMENT

The following listing shows the basic setup required in the markup. The `<style>` element should be placed in the head section of the document, or you can add the rule to your existing `<style>` element. The `div` replaces the existing one, where your `<video>` element is located.

Listing 8.9 index.html—Adding a `<canvas>` element to display video

```
<style>
    #player video:first-of-type {
        display: none;
    }
</style>

<div>
    <canvas width="720" height="480"></canvas>
    <video controls
           width="720" height="480">
        <source src="videos/VID_20120122_133036.mp4"
                type="video/mp4">
        <source src="videos/VID_20120122_133036.webm"
                type="video/webm">
        Your browser does not support for the video element, please
        try <a href="videos/VID_20120122_133036.mp4">downloading
        the video instead</a>
    </video>
</div>
```

CSS is used to hide the <video> element.

Add a <canvas> element with the same dimensions as the video.

The <video> element remains as it was, although now that it's invisible, the controls parameter and fallback content aren't strictly necessary.

STEP 2: GRAB AND DISPLAY IMAGE DATA

Core API

Now you need to listen for the `play` event on the `<video>` element and use that as a trigger to start grabbing video frames and rendering on the canvas. The `$(document).ready` in the next listing should replace the existing function you added previously in listing 8.8.

Listing 8.10 index.html—Adjusting the `draw()` function to use the `<canvas>` element

```
$(document).ready(
    function() {
        $('.playlist').bind('click', change_video);
        var v = $('#player video:first-of-type')[0];
        var canvas = $('#player canvas:first-of-type')[0];
        var context = canvas.getContext('2d');
        function draw() {
            if(v.paused || v.ended) return false;
            context.drawImage(v,0,0,720,480);
            requestAnimationFrame(draw);
        }
        v.addEventListener('play', draw);
    }
)
```

This part of the code remains the same as before.

The draw() function will draw the video frames one by one on the canvas; a closure is used to cache references to the video and the canvas context.

If the video has stopped playing, don't do any additional work.

A recursive call is made to the draw() function using the requestAnimationFrame polyfill (see listing 8.1).

Listen for the play event on the <video> element to kick off the draw function.

Now you're able to play back the video through the `<canvas>` element, but you'll notice that something is missing. The controls you got for free as part of the `<video>` element are no longer accessible now that the video is being played through `<canvas>`. The next section deals with creating your own controls.

Figure 8.7 Custom playback buttons in Opera

8.4.2 Creating custom video playback controls

In this section you'll create a simple menu of buttons to control video playback. Figure 8.7 shows the final effect. Obviously, we're not aiming to win any points for design here; it's the functionality we're interested in.

STEP 3: ADD MARKUP FOR AND IMPLEMENT VIDEO PLAYER CONTROLS

The simple markup for the controls we're adding—return to start, slow down playback, pause, play, and speed up playback—is shown here; add this code directly after the `<canvas>` element.

Listing 8.11 index.html—Creating video player controls

```
<menu>
    <button>|&lt;</button>
    <button>&lt;&lt;</button>
    <button>||</button>
    <button> &gt; </button>
    <button>&gt;&gt;</button>
</menu>
```

Return to start.
Slow down playback.
Pause.
Play.
Speed up playback.

Core API

To make the buttons functional, you'll have to learn about a few more properties and methods on the HTMLMediaElement interface. A summary of these methods is shown in table 8.6.

Table 8.6 More HTMLMediaElement interface methods

Attribute/method	Description
`.currentTime`	Read/write the current position (in seconds) of the playback
`.duration`	The length of the media in seconds

Table 8.6 More HTMLMediaElement interface methods *(continued)*

Attribute/method	Description
`.defaultPlaybackRate`	The speed, expressed as a multiple of the standard playback speed of the media
`.playbackRate`	The rate at which the media is currently playing back as a positive multiple of the standard playback speed of the media (less than 1 is slower; greater than 1 is faster)
`.pause()`	Pauses the currently playing media

With these properties and methods you have enough information to implement the five buttons. In the `$(document).ready` function you added in listing 8.10, you'll need to bind a handler to the menu, like the one shown next. It can be added anywhere in that function as long as it's after the declaration for the v variable. If you're not sure, add it at the end.

Listing 8.12 index.html—Handler function for the control menu

```
$('menu').bind('click', function(event) {
    var action = $(event.target).text().trim();
    switch (action) {
        case '|<':
            v.currentTime = 0;
            break;
        case '<<':
            v.playbackRate = v.playbackRate * 0.5;
            break;
        case '||':
            v.pause();
            break;
        case '>':
            v.playbackRate = 1.0;
            v.play();
            break;
        case '>>':
            v.playbackRate = v.playbackRate * 2.0
            break;
    }
    return false;
})
```

For simplicity, you can use the text content of the buttons to determine which one was clicked.

To go back to the start of the video, set the currentTime to 0.

Repeatedly hitting the fast or slow buttons will multiply the playback rate, but hitting play will reset it to 1.

pause() and play() do exactly what it says on the tin.

CODE CHECK!

You've now restored basic functionality to your video player. The working code to this point in the chapter is in the file index-5.html in the code download, so you can compare what you've written. For extra credit, consider how you might use `.currentTime` and `.duration` in concert with a `<meter>` element (see section 2.3.3) to reproduce the seek bar. Otherwise, move on to the next section, where you'll explore the effects you can achieve now that playback is occurring through a `<canvas>` element.

Figure 8.8 Grayscale video playback through `canvas` combined with an image at 90 percent opacity

8.4.3 Manipulating video as it's playing

The point of playing the video through the `<canvas>` element wasn't to merely replicate the behavior you get for free with the `<video>` element but to process the video output. In this section you'll learn basic techniques for processing the video, ending up with something that looks like figure 8.8. You'll use these same techniques in later sections to build the telestrator.

Figure 8.8 also shows the result of the next group of four steps you'll walk through:

- Group 2: Manipulating video as it's playing
  - Step 1: Add a frame image to the video.
  - Step 2: Adjust how the frame and video combine on the canvas.
  - Step 3: Adjust the opacity of the video.
  - Step 4: Grayscale the video being played back.

STEP 1: ADD A FRAME IMAGE TO THE VIDEO

You learned about drawing images on canvas in chapter 6; the basic approach is the same for this step. First, you need an image on the page. It can go anywhere inside the `<#player>` element (hide it with CSS `display: none`):

```
<img src="images/frame.svg">
```

To give users the ability to turn the frame on and off, you'll need a button in the menu from listing 8.11:

```
<button>Framed</button>
```

Because it's on the menu, you can take advantage of the existing click-handling code for that—the additional cases for the `switch` statement are shown in the following listing—and add them to the handler from listing 8.11.

Listing 8.13 index.html—Handler for the Frame button

```
case 'Framed':
    framed = false;
    $(event.target).text('Frame');
    break;
case 'Frame':
    framed = true;
    $(event.target).text('Framed');
    break;
```

You'll set up the framed variable in listing 8.14.

With this next listing, you need to adjust the `draw()` function to draw the frame.

Listing 8.14 index.html—Adjust the `draw()` function to show the frame

```
var framed = true;
var frame = $('#player img:first-of-type')[0];
//...
function draw() {
    if(v.paused || v.ended) return false;
    context.drawImage(v,0,0,720,480);
    if (framed) {
        context.drawImage(frame,0,0,720,480);
    }
    requestAnimationFrame(draw);
    return true;
}
```

This is the framed variable you were promised in listing 8.13.

For brevity, all the other declarations have been left out; leave them as they are in your code.

Draw the frame only if the user has requested it.

The drawImage function is as you remember it; note that the frame gets drawn after (on top of) the video.

And that's it! You should now be able to get a frame to appear over the video playback at the click of a button. In the next step you'll learn how to adjust how the two images, the frame and video, are composed (combined) together on the Canvas.

STEP 2: ADJUST HOW THE FRAME AND VIDEO COMBINE ON THE CANVAS

Core API

By default, things you draw on the Canvas layer on top of each other; each new drawing replaces the pixels below it. But it's possible to make this layering work differently with the `.globalCompositeOperation` property of the context.

Figure 8.9 provides an example of each composition mode available to you.

To allow you to experiment, we've created a `<select>` element with all of the possible modes in listing 8.15. The composition operations split the world into two segments:

- Destination, what's already drawn
- Source, the new stuff you're trying to draw

Add the code from the following listing (place it after the `<menu>` element you added in listing 8.11).

Listing 8.15 index.html—<select> element for composition mode

```
<label>
    Composition:
    <select>
        <option>copy</option>
        <option>destination-atop</option>
        <option>destination-in</option>
        <option>destination-out</option>
        <option>destination-over</option>
        <option>source-atop</option>
        <option>source-in</option>
        <option>source-out</option>
        <option selected>source-over</option>
        <option>lighter</option>
        <option>xor</option>
    </select>
</label>
```

Display the source, where source and destination overlap.

Display the source in the transparent parts of the destination.

Add the source only where it overlaps destination, but put the destination on top.

Set the overlap of destination and source to transparent; elsewhere display the destination.

Where the two overlap, display the destination; elsewhere display the source.

Display the source where it overlaps the destination; show the destination elsewhere.

Set the destination to transparent. Set the overlap of source and destination to transparent; elsewhere display the source.

The default; draw the new stuff over the old.

Add the source and destination colors together.

Parts are transparent where both overlap; elsewhere display destination or source.

Add the source where it overlaps the destination, with the source on top; elsewhere, the destination is transparent.

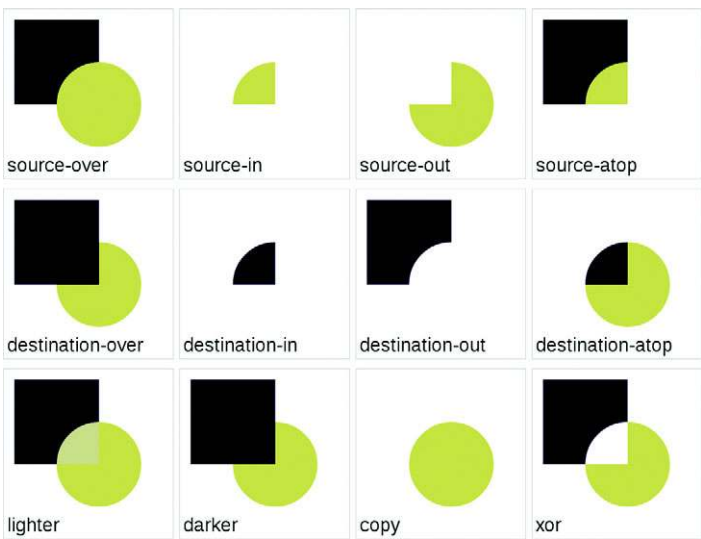

Figure 8.9 Canvas composition modes. The code used to generate this figure is in the source code download in a file called canvas-composition-modes.html.

Now, so that your application can respond to changes, you need to bind the `<select>` element to an event handler. The next listing has code that replaces your existing `draw()` function.

Listing 8.16 index.html—Change the composition mode in the `draw()` function

```
var c_mode = 'source-over';
$('select').bind('change', function(event) {
    c_mode = event.target.value;
})
function draw() {
    if(v.paused || v.ended) return false;
    context.clearRect(0,0,720,480);
    context.globalCompositeOperation = c_mode;
    context.drawImage(v,0,0,720,480);
    if (framed) {
        context.drawImage(frame,0,0,720,480);
    }
    requestAnimationFrame(draw);
    return true;
}
```

Create a variable to keep track of the state as before; saves expensive DOM lookups in the video playback loop.

You've used the JavaScript names in the select options, so this bit is easy.

Set the mode.

If you don't clear the canvas, each successive frame of the video will be composited with the previous one.

Video isn't the ideal format to experiment with composition modes because it's always a fully opaque image, and in this example it's taking up all the pixels. But this simple implementation will allow you to experiment and consider where you might use them in your own projects.

STEP 3: ADJUST THE OPACITY OF THE VIDEO

Core API

The opacity is set with the `.globalAlpha` property. It should be a value between 0 and 1; in common with CSS, 1 is fully opaque and 0 is completely transparent. In your application you can add an item to let the user set the value with a number input; add this code after the `<menu>` element:

```
<label>
    Opacity:<input type="number" step="0.1" min="0" max="1" value="1.0">
</label>
```

As before, you need to attach an event handler to this input and feed the results into the `draw()` function through a variable. The following listing has the additional code to capture the opacity and another new `draw()` function. Replace the `draw()` function from listing 8.15 with this new code (retaining the composition mode binding to `$('select')`):

Listing 8.17 index.html—Change the opacity in the `draw()` function

```
var c_opac = 1;
$('input[type=number]').bind('input', function(event) {
    c_opac = event.target.value;
})
function draw() {
    if(v.paused || v.ended) return false;
    context.clearRect(0,0,720,480);
```

The default opacity is 1 (fully opaque).

Set the variable when the user changes the value.

```
    context.globalCompositeOperation = c_mode;
    context.globalAlpha = c_opac;
    context.drawImage(v,0,0,720,480);
    if (framed) {
        context.drawImage(frame,0,0,720,480);
    }
    requestAnimationFrame(draw);
    return true;
}
```

Use the variable to set the opacity within the draw() function. You can use opacity to create interesting effects when used in combination with the composition mode.

STEP 4: GRAYSCALE THE VIDEO BEING PLAYED BACK

Core API

The `<canvas>` element is also a general-purpose, image-processing tool, thanks to its `.getImageData` and `.putImageData` methods. These methods directly access the array of pixels making up the canvas. Once you have the pixels, you can implement standard image-processing algorithms in JavaScript. The next listing is a JavaScript implementation of an algorithm to turn an image gray. This code can be included anywhere inside your `<script>` element.

Listing 8.18 index.html—A function to make an image grayscale

```
function grayscale(pixels) {
    var d = pixels.data;
    for (var i=0; i<d.length; i+=4) {
      var r = d[i];
      var g = d[i+1];
      var b = d[i+2];
      var v = 0.2126*r + 0.7152*g + 0.0722*b;
      d[i] = d[i+1] = d[i+2] = v
    }
    return pixels;
};
```

NOTE The grayscale function in listing 8.18 is adapted from the HTML Rocks article on image filters; see www.html5rocks.com/en/tutorials/canvas/imagefilters/ for more details.

With the complex math all safely hidden in a general-purpose function, all that remains is to apply it to the canvas. Listing 8.19 shows how you'd call the `grayscale()` function from within your `draw()` function. For this to work, you need to declare a variable `grayed` alongside the `framed` one you created in listing 8.14 and set it to an initial value of `false`.

Listing 8.19 index.html—Use the `grayscale()` function within `draw()`

```
context.drawImage(v,0,0,720,480);
if (grayed) {
    context.putImageData(
        grayscale(context.getImageData(0,0,720,480))
        ,0
        ,0
    );
}
```

You have to first draw the video as an image to the canvas before you can start processing it.

Get the image data from the canvas and pass it through the grayscale function.

Draw the results back on to the canvas starting at the top left (0,0).

NOTE The `getImageData()` method will trigger a security error if you access the example from a file:// URL. If you run into any problems, try accessing the file using a local web server. In Chrome there's also a bug that causes a security violation when `getImageData()` is called after an SVG image has been drawn on the canvas. Check https://code.google.com/p/chromium/issues/detail?id=68568 for updates.

You will also need a Grayed button inside the menu and a handler in the switch statement. This will work analogously to the Framed button you created in listing 8.13, so we won't repeat the code here.

CODE CHECK!

The file index-6.html in the book's code download is a working version of the code to this point (but see section 8.3.1 if you're using IE9).

NOTE Image processing works pixel by pixel, which means it becomes increasingly more expensive the higher the quality of the video. Unless you're building an application to preview video processing results, your users will usually be grateful if you do expensive real-time processing on the server, instead of in their browser.

8.4.4 Building the telestrator features

Using the techniques from the previous section of rendering the video through a `<canvas>` element and overlaying graphics on that video, you can now add the telestration feature. The results, demonstrating the artistic abilities of the authors, are shown in figure 8.10.

Figure 8.10 After working through this final section, you'll be ready to telestrate!

It will take just three remaining steps to get you there:

- Group 3: Building the telestrator feature
  - Step 1: Capture mouse movement.
  - Step 2: Display the captured path over the video.
  - Step 3: Add a "clear" button so users can remove telestrations and start again.

STEP 1: CAPTURE MOUSE MOVEMENT

To capture mouse movement, you'll need to modify your `$(document).ready` function to include the following code. It doesn't matter where you add it; in the downloadable example it's between the initial declarations and the `draw()` function.

Listing 8.20 index.html—Capturing the mouse movement

```
var clickX = new Array();
var clickY = new Array();
var clickDrag = new Array();
var paint = false;

var canvas = $('#player canvas:first-of-type');
var pos = canvas.position();
canvas.bind('mousedown', function(event) {
    var mouseX = event.pageX - pos.left;
    var mouseY = event.pageY - pos.top;
    paint = true;
    addClick(mouseX, mouseY);
}).bind('mousemove', function(event) {
    if(paint){
        var mouseX = event.pageX - pos.left;
        var mouseY = event.pageY - pos.top;
        addClick(mouseX, mouseY, true);
    }
}).bind('mouseup', function(event) {
    paint = false;
}).bind('mouseleave', function(event) {
    paint = false;
});

function addClick(x, y, dragging) {
    clickX.push(x);
    clickY.push(y);
    clickDrag.push(dragging);
}
```

This variable determines whether you're currently recording mouse movements.

Set up global variables to record the movement of the mouse.

You are now using jQuery to attach event handlers to the canvas, so you will use the jQuery reference in code rather than the DOM reference as before. Remember to update the assignment which gets the context to use canvas[0] instead of canvas.

Cache the position of the <canvas> element on the page so you don't have to do expensive DOM queries.

As the user moves the mouse around, and if you're currently painting, add further positions.

When the user presses the mouse button, set the paint variable to true and record the initial position with the addClick function.

If the user releases the button or moves off the canvas, set the paint variable to false.

The addClick function populates the variables created in the first step in this listing.

> **NOTE** To keep the `draw()` function simple, in this section we've removed the code and buttons for `Grayed` and `Framed`. Leaving them in your code won't harm anything, but bear this in mind as you follow the instructions to replace and include code in this section.

STEP 2: DISPLAY THE CAPTURED PATH OVER THE VIDEO

The next step is to display the path within the `draw()` function. The following listing has yet another new `draw()` function.

Listing 8.21 index.html—Modifying the `draw()` function to show the path

```
function draw() {
    if(v.paused || v.ended) return false;
    context.clearRect(0,0,720,480);
    context.globalCompositeOperation = c_mode;
    context.globalAlpha = c_opac;
    context.drawImage(v,0,0,720,480);
    context.strokeStyle = "#ffff00";
    context.lineJoin = "round";
    context.lineWidth = 8;
    for(var i=0; i < clickX.length; i++) {
        context.beginPath();
        if(clickDrag[i] && i){
            context.moveTo(clickX[i-1], clickY[i-1]);
        } else {
            context.moveTo(clickX[i]-1, clickY[i]);
        }
        context.lineTo(clickX[i], clickY[i]);
        context.closePath();
        context.stroke();
    }
    requestAnimationFrame(draw);
    return true;
}
```

Note that to keep things simple, if the video is paused, nothing will be drawn, even though new telestrations will continue to be recorded.

We will telestrate in a nice, visible yellow.

Loop through the coordinates stored in the path.

Special handling for the first coordinate because you can't access element <-1> of an array.

STEP 3: ADD A CLEAR BUTTON SO USERS CAN REMOVE TELESTRATIONS AND START AGAIN

As a final step you need to add a Clear button so users can remove their telestrations and start again. An easy place to put this is in the controls menu you already have, by adding another button:

```
<button>Clear</button>
```

The new case for your big `switch` statement is shown in the next listing.

Listing 8.22 index.html—Process the clear action

```
case 'Clear':
    clickX = new Array();
    clickY = new Array();
    clickDrag = new Array();
    paint = false;
    break;
```

Reset all the stored path data.

Stop capturing new drawing data.

With that you should have a fully functioning video jukebox telestrator and be well on your way to adding your own garish yellow annotations to the videos of your choice. Figure 8.11 shows the authors' feeble attempt at a John Madden impersonation along with the Clear button ready to consign that attempt to history.

CODE CHECK!

In the code download you'll find a working version of the code from this section in the file index-9.html. There's also an index-10.html file, which includes the code from

Figure 8.11 The finished application in Firefox

this section as well as the `Grayed` and `Framed` functionality from the previous section we took out to simplify the listings.

8.5 Summary

In this chapter you've learned how HTML5 makes it as straightforward a process to add video and audio to web pages as it is to add images. You've taken the news of browser incompatibilities in format support in stride and learned how to convert between video formats, and you've learned how to control media elements with JavaScript. The added bonus of having video within HTML5 is that you can use it as input for other content, in particular the `<canvas>` element. You've also learned how to combine video with images and, finally, how to combine it with live drawing. We hope that in addition to all the technical knowledge you've gained, you've also thought of ideas on how to incorporate media within your web applications, as well as playing media on your page.

In the next chapter, you'll continue to learn about exciting visual effects you can create with HTML5 as you learn about WebGL. The WebGL format allows you direct access to the computer's graphics hardware from JavaScript, raising the possibility of implementing real 3D games and data visualizations.

Chapter 9 at a glance

Topic	Description, methods, and so on	Page
Engine creation	Creating a WebGL engine from scratch	
	▪ Time-saving scripts	274
	▪ Basic engine pattern	277
	▪ Default entity class	279
	▪ Helper methods	280
Graphics cards	Interacting with a graphics card	
	▪ OpenGL	282
	▪ Creating shaders	284
	▪ Attaching 3D data to entities	283
	▪ Outputting shapes	288
	▪ Matrices usage	288
WebGL app	Putting everything together to create an app	
	▪ 2D triangle in 3D	296
	▪ 3D basics	297
	▪ Large complex polygons	300
	▪ Cubes	305
	▪ Particle generation	308

Look for this icon 

throughout the chapter to quickly locate the topics outlined in this table.