Safari HTML5 Audio and Video Guide

Audio, Video, & Visual Effects

2010-03-18

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction

If you embed audio or video media in your website, you should read this document.

HTML5 is the next major version of HTML, the primary standard that determines how web content interacts with browsers. HTML5 enables audio and video to play natively in the browser without requiring proprietary plug-ins. With HTML5, you can add media to a website with just a few lines of code. You can also create customized media controllers for rich interactivity using standard web technologies.

The HTML5 `<audio>` and `<video>` tags make adding media to a website simple. Just use the `src` attribute to identify the media source and include a `controls` attribute so the user can play and pause the media.

Example:

```
<video src="http://Myserver.com/Path/Mymovie.mp4" controls> </video>
```

You can set additional attributes, such as `autoplay` or `loop`, optionally supply a height and width, or simply let the browser allocate the space it needs based on the media. In any case, it generally requires only a single line of code.

Safari on iPhone OS 3.0 and later (including iPad) and Safari on the desktop 3.1 and later (for Mac OS X and Windows) support the `<video>` and `<audio>` elements. Not all browsers support these tags yet, and not all types of encoded media play back in all browsers. However, there are simple ways to provide fallbacks if the browser doesn't yet support the `<video>` or `<audio>` tag, and simple methods to provide alternate media sources if the browser supports the tags but can't play a particular media file.

Because the `<audio>` and `<video>` elements are standard HTML they integrate automatically with CSS and JavaScript. You can combine CSS properties, JavaScript functions, and DOM events to create sophisticated movie controllers customized to your needs and aesthetics, yet entirely based on W3C standards and independent of third-party plug-in APIs.

Standard JavaScript methods allow you to play, pause, jump to a given point, or specify a new source, and DOM events notify you when the media loads, plays, pauses, or completes. You can apply CSS properties and use DOM events to modify `<video>` elements just as you would any other element—change the opacity, overlay text, move the element smoothly across the page, add a reflection, rotate the element in 3D, and so on. This document includes several examples.

Important: This is a preliminary document. Although it has been reviewed for technical accuracy, it is not final. Apple is supplying this information to help you adopt the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be vetted against final documentation. For information about updates to this and other developer documentation, you can check the ADC Reference Library Revision List. To receive notification of documentation updates, you can sign up for a free Apple Developer Connection Online membership and receive the bi-weekly ADC News e-mail newsletter. See http://developer.apple.com/products/for more details about ADC membership.)

Organization of This Document

This document shows you how to use the HTML5 `<audio>` and `<video>` tags in the following ways:

- "Audio and Video Tag HTML" (page 9)—How to use the `<audio>` and `<video>` tags, taking device-specific considerations into account; how to specify multiple sources; and how to provide fallback content for browsers that don't support the `<audio>` and `<video>` elements.
- "Controlling Media With JavaScript" (page 19)—How to control media using JavaScript and DOM events.
- "Adding CSS Styles" (page 27)—How to enhance media presentation using CSS.

See Also

- *Safari DOM Additions Reference*—DOM events added to Safari to support HTML5 audio and video, touch events and gestures, and CSS transforms and transitions.
- *Safari CSS Visual Effects Guide*—How to use CSS transitions and effects in Safari.
- *Safari CSS Reference*—Complete list of CSS properties, rules, and property functions supported in Safari, with syntax and usage.
- *Safari HTML Reference*—The HTML elements and attributes supported by different Safari and WebKit applications.
- *WebKit DOM Programming Topics*—How to get the most out of using DOM events in Safari.
- *Safari Web Content Guide*—How to create web content that works well in Safari on the desktop and Safari on iPhone OS.
- *iPhone Human Interface Guidelines for Web Applications*—User interface guidelines for designing webpages and web applications for Safari on iPhone OS.

Audio and Video HTML

In its simplest form, the `<audio>` and `<video>` tags require only a `src` attribute to identify the media, although you generally want to set the `controls` attribute as well, so the user can play and pause the media. The browser allocates space, provides a default controller, loads the media, and plays it when the user clicks the play button. It's all automatic.

There are optional attributes as well, such as `autoplay`, `loop`, `height`, and `width`.

Basic Syntax

The attributes for the `<audio>` and `<video>` tags are summarized in Table 1-1 (page 9). The only difference between the `<audio>` and `<video>` tag attributes is the option to specify a height, width, and poster image for video.

Table 1-1 Attributes of the `<audio>` and `<video>` elements

Attribute	Value	Description
autobuffer	Boolean—any value sets this to `true`	If present, asks the browser to begin loading the media immediately.
autoplay	Boolean—any value sets this to `true`	If present, asks the browser to play the media automatically.
controls	Boolean—any value sets this to `true`	If present, causes the browser to display the default media controls.
height (video only)	*pixels*	The height of the video player.
loop	Boolean—any value sets this to `true`	If present, causes the media to loop indefinitely.
poster (video only)	*url of image file*	If present, shows the poster image until the first frame of video has downloaded.
src	*url*	The URL of the media.
width (video only)	*pixels*	The width of the video player.

Important: Several of the attributes are boolean. Although they can be set to `false` using JavaScript, *any* use of them in the HTML tag sets them to `true`. `Controls="controls"`, for example, is the same as `controls=true` or simply `controls`. Even `controls=false` sets controls to `true` in HTML. To set these attributes to `false` in HTML, omit them from the tag.

Listing 1-1 shows an HTML page that autoplays a video with user controls.

Listing 1-1 Creating a simple movie player

```
<!DOCTYPE html>
<html>
   <head>
      <title>Simple Movie Player</title>
   </head>
   <body>
      <video src="http://Domain.com/path/My.mov"
             controls="controls"
             autoplay="autoplay"
             height=270 width=480
      >
      </video>
  </body>
</html>
```

Device-Specific Considerations

There are a handful of device-specific considerations you should be aware of when embedding audio and video using HTML5.

Optimization for Small Screens

Currently, Safari optimizes video presentation for the smaller screen on iPhone or iPod touch by displaying audio or video using the full screen—video controls appear when the screen is touched, and the video is scaled to fit the screen in portrait or landscape mode. Neither video nor audio is presented within the webpage. The `height` and `width` attributes affect only the space allotted on the webpage, and the `controls` attribute is ignored. This is true only for Safari on devices with small screens. On Mac OS X, Windows, and iPad, Safari displays audio and video inline, embedded in the webpage.

User Control of Downloads Over Cellular Networks

In Safari on iPhone OS (for all devices, including iPad), where the user may be on a cellular network and be charged per data unit, autobuffering and autoplay are disabled. No data is loaded until the user initiates it. This means the JavaScript `play()` and `load()` methods are also inactive until the user initiates playback, unless the `play()` method is triggered by user action. In other words, a user-initiated Play button works, but an `onLoad` play event does not.

This plays the movie: `<input type="button" value="Play" onClick="document.myMovie.play()">`

This does nothing on iPhone OS: `<body onLoad="document.myMovie.play()">`

Because the native dimensions of a video are not known until the movie metadata loads, a default height and width of 150 x 300 is allocated on devices running iPhone OS if the height or width is not specified. Currently, the default height and width do not change when the movie loads, so you should specify the preferred height and width for the best user experience on iPhone OS, especially iPad.

iPhone Video Placeholder

On iPhone and iPod touch, a placeholder with a play button is shown until the user initiates playback, as shown in Figure 1-1. The placeholder is translucent, so the background or any poster image shows through. The placeholder provides a way for the user to play the media.

Figure 1-1 The iPhone video placeholder

On the desktop and iPad, the first frame of a video displays as soon as it becomes available. There is no placeholder.

Media Playback Controls

Controls are always supplied during fullscreen playback on iPhone and iPod touch, and the placeholder allows the user to initiate fullscreen playback. On the desktop or iPad, you must either include the `controls` attribute or provide playback controls using JavaScript. It is especially important to provide user controls on iPad because autoplay is disabled.

Supported Media

Safari on the desktop supports any media the installed version of QuickTime can play. This includes media encoded using codecs QuickTime does not natively support, provided the codecs are installed on the user's computer as QuickTime codec components.

Safari on iPhone OS (including iPad) currently supports uncompressed WAV and AIF audio, MP3 audio, and AAC-LC or HE-AAC audio. HE-AAC is the preferred format.

Safari on iPhone OS (including iPad) currently supports MPEG-4 video (Baseline profile) and QuickTime movies encoded with H.264 video (Baseline profile) and one of the supported audio types.

iPad and iPhone 3G and later support H.264 Baseline profile 3.1. Earlier versions of iPhone support H.264 Baseline profile 3.0.

Video Overlays

Currently, all devices running iPhone OS are limited to playback of a single video stream at any time. Playing more than one video—side by side, partly overlapping, or completely overlaid—is not currently supported on iPhone OS devices. You can change the video source dynamically, however. See "Replacing a Media Source Sequentially" (page 22) for details.

Working with Attributes

There are several ways you can control media playback directly in HTML by setting attributes appropriately.

Resizing the Video

In Safari on iPhone OS, the native size of the video is unknown until the user initiates a download. If no height or width is specified, a default size of 150 x 300 pixels is allocated in the webpage. On iPad, the video currently plays in this default space. On iPhone or iPod touch, the video plays in full-screen mode once the user initiates it, and the default space is allocated to a placeholder on the webpage.

In Safari on the desktop, the movie metadata is downloaded automatically whenever possible, so the native video size is used as the default. If only the `height` or `width` parameter is specified, the video is scaled up or down to that height or width while maintaining the native aspect ratio of the movie. If both height and width are specified, the video is presented at that size. If neither is specified, the video is displayed at its native size.

Enabling the Controller

In Safari, the default video controller is slightly translucent and is overlaid on the bottom 18 pixels of the video. The controller is not normally visible when the movie is playing—it appears only when the movie is paused, when the user touches the video, or when the mouse pointer hovers over the playing movie. In cases where it is crucial that the bottom of the video never be obscured, omit the `controls` attribute. (You cannot set the attribute to `false` explicitly in HTML—you set it to `false` *implicitly* by leaving the attribute out.)

If you do not set the `controls` attribute, you must either set the `autoplay` attribute, create a controller using JavaScript, or play the movie programmatically from JavaScript. Otherwise the user has no way to play the movie.

Warning: To prevent unsolicited downloads over cellular networks at the user's expense, embedded media cannot be played automatically in Safari on iPhone OS—the user always initiates playback. A controller is automatically supplied on iPhone or iPod touch once playback in initiated, but for iPad you must either set the `controls` attribute or provide a controller using JavaScript.

Autobuffering

If you set the `autobuffer` attribute, it tells the browser you want the specified media file to start buffering immediately, making it more likely that it will begin promptly and play through smoothly when the user plays it.

If you have multiple movies on the page, you should leave the `autobuffer` attribute unset, to prevent all the movies from downloading at once.

Note: The `autobuffer` attribute is not supported in Safari 4.0.4. Safari on the desktop always autobuffers. Safari on iPhone OS never autobuffers.

Playing Background Music

To play an audio file in the background, set the `autoplay` attribute but not the `controls` attribute, as shown in Listing 1-2. To play the file continuously, set the `loop` attribute.

Listing 1-2 Playing Background audio

```
<!DOCTYPE html>
<html>
   <head>
      <title>Background Audio Player</title>
   </head>
   <body>
      <audio src="http://Domain.com/path/My.mp3"
             autoplay
             loop
            <!-- values are optional for boolean attributes -->
      >
      </audio>
<p>If the sound is turned on, you should hear music playing...</p>
  </body>
</html>
```

Note: On iPhone or iPod touch, the audio is played in full-screen mode. On iPad, the audio does not play unless you set the `controls` attribute or provide a JavaScript control.

Showing a Poster

Setting a poster image normally has a transitory effect—the poster image is shown only until the first frame of the video is available, which is typically a second or two. On iPhone and iPod touch, however, the first frame is not shown until the user initiates playback, and a poster image is recommended, as shown in Listing 1-3.

Listing 1-3 Showing a poster

```
<!DOCTYPE html>
<html>
```

```
   <head>
      <title>Movie Player with Poster</title>
   </head>
   <body>
      <video src="http://Domain.com/path/My.mov"
             controls="controls"
             autoplay="autoplay"
             height=340 width=640
             poster="http://Domain.com/path/Poster.jpg"
      >
      </video>
  </body>
</html>
```

Providing Multiple Sources

Not all types of audio and video can play on all devices and browsers. Fortunately, the `<audio>` and `<video>` elements allow you to list as many `<source>` elements as you like. The browser iterates through them until it finds one it can play or runs out of sources. Instead of using the `src` attribute in the media element itself, follow the `<audio>` or `<video>` tag with one or more `<source>` elements, prior to the closing tag.

```
<audio controls=controls>
<source src="mySong.aac">
<source src="mySong.oga">
</audio>
```

The `<source>` element can have a `type` attribute, specifying the MIME type, to help the browser decide whether or not it can play the media without having to load the file.

```
<audio controls=controls>
<source src="mySong.aac" type="audio/mp4">
<source src="mySong.oga" type="audio/ogg">
</audio>
```

The `type` attribute can take an additional `codecs` parameter, to specify exactly which versions of which codecs are needed to play this particular media.

```
<audio controls=controls>
<source src="mySongComplex.aac" type="audio/mp4; codecs=mp4a.40.5">
<source src="mySongSimple.aac" type="audio/mp4; codecs=mp4a.40.2">
</audio>
```

Specifying Multiple Media Formats

List the alternate media sources in the order of most desirable to least desirable. The browser chooses the first listed source that it thinks it can play. For example, if you have an AAC audio file, an Ogg Vorbis audio file, and a WAVE audio file, listed in that order, Safari plays the AAC file. A browser that cannot play AAC but can play Ogg plays the Ogg file. A browser that can play neither Ogg nor AAC might still be able to play the WAVE.

Listing 1-4 a simple example of using multiple sources for an audio file:

Listing 1-4 Specifying multiple audio sources

```
<!DOCTYPE html>
<html>
   <head>
      <title>Multi-Source Audio Player</title>
   </head>
   <body>
      <audio
             controls="controls"
             autoplay="autoplay"
      >
             <source src="http://Domain.com/path/MyAudio.m4a">
             <source src="http://Domain.com/path/MyAudio.oga">
             <source src="http://Domain.com/path/MyAudio.wav">
      </audio>
  </body>
</html>
```

Notice that you don't have to know which browsers support what formats, and then sniff the user agent string for the browser name to decide what to do. Just list your available formats, preferred format first, second choice next, and so on. The browser plays the first one it can.

Currently, most common browsers support low-complexity AAC and MP3 audio and basic profile MPEG-4. Most browsers that do not support these formats support Theora video and Vorbis audio using the Ogg file format. Generally speaking, if you provide media in MPEG-4 (basic profile) and Ogg formats, your media can play in all common browsers that support HTML5 media.

Note: Safari on iPhone OS supports low-complexity AAC audio, MP3 audio, AIF audio, WAVE audio, and baseline profile MPEG-4 video. Safari on the desktop (Mac OS X and Windows) supports all media supported by the installed version of QuickTime, including any installed third-party codecs.

Specifying Multiple Delivery Schemes

You can also use multiple source files to specify different delivery schemes. Let's say you have a large real-time video streaming service that uses RTSP streaming, and you want to add support for Safari on iPhone OS, including iPad, using HTTP Live Streaming, along with a progressive download for browsers that can't handle either kind of streaming. As shown in Listing 1-5, with HTML5 video it's quite straightforward.

Listing 1-5 Specifying multiple delivery schemes

```
<!DOCTYPE html>
<html>
   <head>
      <title>Multi-Scheme Video Player</title>
   </head>
   <body>
      <video controls autoplay >
             <source src="http://HttpLiveStream.4gu">
             <source src="rtsp://LegacyStream.3gp">
             <source src="http://ProgressiveDownload.m4v">
      </video>
  </body>
</html>
```

Again, the browser picks the first source it can handle. Safari on the desktop plays the RTSP stream, while Safari on iPhone OS plays the HTTP Live stream. Browsers that support neither 4gu playlists nor RTSP URLs play the progressive download version.

Multiple Data Rate Sources

HTML5 does not support selection from multiple sources based on data rate. If you supply multiple sources, the browser chooses the first it can play based on scheme, file format, profile, and codecs. Bandwidth is not tested. To provide multiple bandwidths, you must provide a `src` attribute that specifies a source capable of supporting data rate selection itself.

For example, if one of your sources is an HTTP Live Stream, the playlist file can specify multiple streams, and Safari selects the best stream for the current bandwidth dynamically as network bandwidth changes.

Similarly, if the source is a QuickTime reference movie, it can include alternate sources for progressive download, and Safari chooses the best reference movie for the bandwidth when the video is first requested (though it does not dynamically switch between sources if available bandwidth subsequently changes).

Specifying Fallback Behavior

It's easy to specify fallback behavior for browsers that don't support the `<audio>` or `<video>` elements—just put the fallback content between the opening and closing media tags, after any `<source>` elements. See Listing 1-6.

Listing 1-6 Adding simple fallback behavior

```
<!DOCTYPE html>
<html>
   <head>
      <title>Simple Movie Player with Trivial Fallback Behavior</title>
   </head>
   <body>
      <!-- opening tag -->
      <video src="http://Domain.com/path/My.mov"
             controls="controls"
      >

       <!-- fallback content -->
       Your browser does not support the video element.

      <!-- closing tag -->
      </video>
  </body>
</html>
```

Browsers that don't support the `<audio>` or `<video>` tags simply ignore them. Browsers that support these elements ignore all content between the opening and closing tags (except `<source>` tags).

Note: Browsers that understand the `<audio>` and `<video>` tags do not display fallback content, even if they cannot play any of the specified media. To provide fallback content in case no specified media is playable, see "Using JavaScript to Provide Fallback Content" (page 23).

Fall Back to the QuickTime Plug-in

There is a simple way to fall back to the QuickTime plug-in that works for nearly all browsers—download the prebuilt JavaScript file provided by Apple, `AC_QuickTime.js`, from http://developer.apple.com/internet/licensejs.html and include it in your webpage by inserting the following line of code into your HTML head:

```
<script src="AC_QuickTime.js" type="text/javascript">
</script>
```

Once you've included the script, add a call to `QT_WriteOBJECT()` between the opening and closing tags of the `<audio>` or `<video>` element, passing in the URL of the movie, its height and width, and the version of the activeX control for Internet Explorer (just leave this parameter blank to use the current version). See Listing 1-7 for an example.

Listing 1-7 Falling back to the QuickTime plug-in

```
<!DOCTYPE html>
<html>
   <head>
      <title>Simple Movie Player with QuickTime Fallback</title>
      <script src="AC_QuickTime.js" type="text/javascript">
      </script>
   </head>
   <body>
      <video controls="controls">
         <source src="myMovie.mov" type="video/quicktime">
         <source src="myMovie.3gp" type="video/3gpp">
       <!-- fallback -->
       <script type="text/javascript">
        QT_WriteOBJECT('My.mov' , '320', '240', '');
       </script>
      </video>
  </body>
</html>
```

You can pass more than twenty additional parameters to the QuickTime plug-in. For more about how to work with the QuickTime plug-in and the `AC_QuickTime.js` script, see *HTML Scripting Guide for QuickTime*.

Fall Back to Any Plug-In

Since most browsers now support the `<audio>` and `<video>` elements, you can simplify the process of coding for plug-ins by including only the version of the `<object>` tag that works with Internet Explorer as your fallback for HTML5 media.

The example shListing 1-8 uses HTTP Live Streaming for browsers that support it, MPEG-4 by progressive download for most others, Ogg Vorbis for browsers that support only open-source media, and falls back to a plug-in for versions of Internet Explorer that don't support HTML5.

Listing 1-8 Falling back to a plug-in for IE

```
<!DOCTYPE html>
<html>
   <head>
      <title>Simple Movie Player with Plug-In Fallback</title>
   </head>
   <body>
       <video controls="controls">
        <source src="HttpLiveStream.m3u8 type="vnd.apple.mpegURL">
        <source src="ProgressiveDowload.mp4" type="video/mp4">
        <source src="OggVorbis.ogv" type="vido/ogg">
<!-- fallback -->
        <OBJECT CLASSID="clsid:02BF25D5-8C17-4B23-BC80-D3488ABDDC6B"
         CODEBASE="http://www.apple.com/qtactivex/qtplugin.cab"
         HEIGHT="320"
         WIDTH="240"
        >
          <PARAM NAME="src" VALUE="ProgressiveDownload.mp4">
        </object>
      </video>
  </body>
</html>
```

TListing 1-8 uses the QuickTime plug-in. To use a different plug-in, change the `CLASSID` and `CODEBASE` parameters to those of your preferred plug-in and provide the `PARAM` tags the plug-in requires.

Controlling Media with JavaScript

Because the `<audio>` and `<video>` elements are part of the HTML5 standard, there are standard JavaScript methods, properties, and DOM events associated with them.

There are methods for loading, playing, pausing, and jumping to a time, for example. There are also properties you can set programmatically, such as the `src` URL and the height and width of a video, as well as read-only properties such as the video's native height. Finally, there are DOM events you can listen for, such as load progress, media playing, media paused, and media done playing.

This chapter shows you how to do the following:

- Use JavaScript to create a simple controller.
- Change the size of a movie dynamically.
- Display a progress indicator while the media is loading.
- Replace one movie with another when the first finishes.
- Provide fallback content using JavaScript if none of the media sources are playable.

For a complete description of all the methods, properties, and DOM events associated with HTML5 media, see *Safari DOM Additions Reference*. All the methods, properties, and DOM events associated with `HTMLMediaElement`, `HTMLAudioElement`, and `HTMLVideoElement` are exposed to JavaScript.

A Simple JavaScript Media Controller and Resizer

Any of the standard ways to address an HTML element in JavaScript can be used with `<audio>` or `<video>` elements. You can assign the element a name or an id, use the tag name, or use the element's place in the DOM hierarchy. The example in Listing 2-1 uses the tag name. The example creates a simple play/pause movie control in JavaScript, with additional controls to toggle the video size between normal and doubled. It is intended to illustrate, in the simplest possible way, addressing a media element, reading and setting properties, and calling methods.

Listing 2-1 Adding simple JavaScript controls

```
<!DOCTYPE html>
<html>
  <head>
    <title>Simple JavaScript Controller</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <script type="text/javascript">

       function playPause() {
        var myVideo = document.getElementsByTagName('video')[0];
         if (myVideo.paused)
            myVideo.play();
```

```
        else
           myVideo.pause();
      }

      function makeBig() {
         var myVideo = document.getElementsByTagName('video')[0];
         myVideo.height = (myVideo.videoHeight * 2 ) ;
       }
      function makeNormal() {
         var myVideo = document.getElementsByTagName('video')[0];
         myVideo.height = (myVideo.videoHeight) ;
       }

    </script>
  </head>

  <body>
     <div class="video-player" align="center">
      <video src="myMovie.m4v" poster="poster.jpg" ></video>
      <br>
      <a href="javascript:playPause();">Play/Pause</a> <br>
      <a href="javascript:makeBig();">2x Size</a> |
      <a href="javascript:makeNormal();">1x Size</a> <br>
     </div>
  </body>
</html>
```

The previous example gets two read-only properties: `paused` and `videoHeight` (the native height of the video). It calls two methods: `play()` and `pause()`. And it sets one read/write property: `height`. Recall that setting only the height or width causes the video to scale up or down while retaining its native aspect ratio.

Note: Safari for iPhone OS version 3.2 does not support dynamically resizing video on the iPad.

Using DOM Events to Monitor Load Progress

One of the common tasks for a movie controller is to display a progress indicator showing how much of the movie has loaded so far. One way to do this is to constantly poll the media element's `buffered` property, to see how much of the movie has buffered, but this is a waste of time and energy. It wastes processor time and often battery charge, and it slows the loading process.

A much better approach is to create an event listener that is notified when the browser has something to report. Once the movie has begun to load, you can listen for `progress` events. You can read the `buffered` value when the browser reports `progress` and display it as a percentage of the movie's `duration`.

Another useful DOM event is `canplaythrough`, a logical point to begin playing programmatically.

Listing 2-2 loads a large movie from a remote server so you can see the progress changing. It installs an event listener for `progress` events and another for the `canplaythrough` event. It indicates the percentage loaded by changing the inner HTML of a paragraph element.

You can copy and paste the example into a text editor and save it as HTML to see it in action.

Listing 2-2 Installing DOM event listeners

```
<!DOCTYPE html>
<html>
 <head>
    <title>JavaScript Progress Monitor</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <script type="text/javascript">


       function getPercentProg() {
         var myVideo = document.getElementsByTagName('video')[0];
         var soFar = parseInt(((myVideo.buffered.end(0) / myVideo.duration) *
100));
         document.getElementById("loadStatus").innerHTML =  soFar + '%';
        }

       function myAutoPlay() {
          var myVideo = document.getElementsByTagName('video')[0];
          myVideo.play();
        }

        function addMyListeners(){
         var myVideo = document.getElementsByTagName('video')[0];
         myVideo.addEventListener('progress',getPercentProg,false);
         myVideo.addEventListener('canplaythrough',myAutoPlay,false);
        }

   </script>
  </head>

  <body onLoad="addMyListeners()">
     <div align=center>
        <video controls
        src="http://homepage.mac.com/qt4web/sunrisemeditations/myMovie.m4v" >
        </video>
        <p ID="loadStatus">
         MOVIE LOADING...
        </p>
      </div>
  </body>
</html>
```

Note: On the iPad, Safari does not begin downloading until the user clicks the poster or placeholder. Currently, downloads begun in this manner do not emit `progress` events.

The `buffered` property is a `TimeRanges` object, essentially an array of start and stop times, not a single value. Consider what happens if the person watching the media uses the time scrubber to jump forward to a point in the movie that hasn't loaded yet—the movie stops loading and jumps forward to the new point in time, then starts buffering again from there. So the `buffered` property can contain an array of discontinuous ranges. The example simply seeks to the end of the array and reads the last value, so it actually shows the percentage into the movie duration for which there is data. To determine precisely what percentage of a movie has loaded, taking possible discontinuities into account, iterate through the array, summing the seekable ranges, as illustrated in Listing 2-3

Listing 2-3 Summing a TimeRanges object

```
var myBuffered = myVideo.buffered;
var total = 0;
for (ndx = 0; ndx < myBuffered.length; ndx++)
   total += (seekable.end(ndx) - seekable.start(ndx));
```

The `buffered`, `played`, and `seekable` properties are all `TimeRanges` objects.

Replacing a Media Source Sequentially

Another common task for a website programmer is to create a playlist of audio or video—to put together a radio set or to follow an advertisement with a program, for example. To do this, you can provide a function that listens for the `ended` event. The `ended` event is triggered only when the media ends (plays to its complete duration), not if it is paused.

When your listener function is triggered, it should change the media's `src` property, then call the `load` method to load the new media and the `play` method to play it, as shown in Listing 2-4.

Listing 2-4 Replacing media sequentially

```
<!DOCTYPE html>
<html>
  <head>
   <title>Sequential Movies</title>
   <meta http-equiv="content-type" content="text/html; charset=utf-8">
   <script type="text/javascript">

       // listener function changes src
       function myNewSrc() {
          var myVideo = document.getElementsByTagName('video')[0];

myVideo.src="http://homepage.mac.com/qt4web/sunrisemeditations/myMovie.m4v";
          myVideo.load();
          myVideo.play();
        }
       // function adds listener function to ended event -->
        function myAddListener(){
         var myVideo = document.getElementsByTagName('video')[0];
         myVideo.addEventListener('ended',myNewSrc,false);
        }
   </script>
  </head>
  <body onload="myAddListener()">
      <video controls
        src="http://homepage.mac.com/qt4web/sunrisemeditations/A-chord.m4v"
       >
      </video>
  </body>
</html>
```

The previous example works on both Safari for the desktop and Safari for iPhone OS, as the `load()` and `play()` methods are enabled even on cellular networks once the user has started playing the first media element.

Using JavaScript to Provide Fallback Content

It's easy to provide fallback content for browsers that don't support the `<audio>` or `<video>` tag using HTML (see "Fallback" (page 16)). But if the browser understands the tag and can't play any of the media you've specified, you need JavaScript to detect this and provide fallback content.

To do this, you need to iterate through your source types using the `canPlayType` method.

Important: The HTML5 specification has changed. Browsers conforming to the earlier version of the specification, including Safari 4.0.4 and earlier, return "no" if they cannot play the media type. Browsers conforming to the newer version return an empty string (“”)instead. You need to check for either response, or else check for a positive response rather than a negative one.

If the method returns “no” or the empty string (“”) for all the source types, the browser knows it can't play any of the media, and you need to supply fallback content. If the method returns “maybe” or “probably” for any of the types, it will attempt to play the media and no fallback should be needed.

The following example creates an array of types, one for each source, and iterates through them to see if the browser thinks it can play any of them. If it exhausts the array without a positive response, none of the media types are supported, and it replaces the video element using `innerHTML`. Listing 2-5 displays a text message as fallback content. You could fall back to a plug-in or redirect to another page instead.

Listing 2-5 Testing for playability using JavaScript

```
<!DOCTYPE html>
<html>
 <head>
   <title>JavaScript Fallback</title>
   <meta http-equiv="content-type" content="text/html; charset=utf-8">
   <script type="text/javascript">

     function checkPlaylist() {
       var playAny = 0;
       myTypes = new Array ("video/mp4","video/ogg","video/divx");
       var nonePlayable = "Your browser cannot play these movie types."
       var myVideo = document.getElementsByTagName('video')[0];
       for (x = 0; x < myTypes.length; x++)
       { var canPlay = myVideo.canPlayType(myTypes[x]);
         if ((canPlay=="maybe") || (canPlay=="probably"))
            playAny = 1;
       }
       if (playAny==0)
         document.getElementById("video-player").innerHTML = nonePlayable;
     }

   </script>
 </head>
 <body onload="checkPlaylist()" >
   <div id="video-player" align=center>
     <video controls height="200" width="400">
     <source src="myMovie.m4v" type="video/mp4">
     <source src="myMovie.oga" type="video/ogg">
     <source src="myMovie.dvx" type="video/divx">
```

```
    </video>
  </div>
 </body>
</html>
```

Handling Playback Failure

Even if a source type is playable, that's no guarantee that the source *file* is playable—the file may be missing, corrupted, misspelled, or the `type` attribute supplied may be incorrect. If Safari 4.0.4 or earlier attempts to play a source and cannot, it emits an `error` event. However, it still continues to iterate through the playable sources, so the `error` event may indicate only a momentary setback, not a complete failure. It's important to check which source has failed to play.

Important: If you install an event listener for the `error` event, it should not trigger fallback behavior unless the `currentSrc` property contains the last playable source filename.

Changes in the HTML5 specification now require the media element to emit an error only if the *last* playable source fails, so this test should not be necessary in the future, except for compatibility with current browsers.

TThe example in Listing 2-5 iterates through the source types to see if any are playable. It saves the filename of the last playable source. If there are no playable types, it triggers a fallback. If there are playable types, it installs an `error` event listener. The event listener checks to see if the current source contains the last playable filename before triggering a failure fallback. (The `currentSrc` property includes the full path, so test for inclusion, not equality.)

Notice that when adding a listener for the `error` event you need to set the `capture` property to `true`, whereas for most events you set it to `false`.

Listing 2-6 Testing for failure using JavaScript

```
<!DOCTYPE html>
<html>
 <head>
   <title>JavaScript Fallback</title>
   <meta http-equiv="content-type" content="text/html; charset=utf-8">
   <script type="text/javascript">

     var lastPlayable;
     myTypes = new Array ("video/mp4","video/ogg","video/divx");
     mySrc = new Array ("myMovie.mp4","myMovie.oga","myMovie.dvx");

     function errorFallback() {
        var errorLast = "An error occurred playing " ;
        var myVideo = document.getElementsByTagName('video')[0];
        if (myVideo.currentSrc.match(lastPlayable))
         { errorLast = errorLast + lastPlayable ;
           document.getElementById("video-player").innerHTML = errorLast;
         }
      }

     function checkPlaylist() {
       var noPlayableTypes = "Your browser cannot play these movie types";
```

```
        var myVideo = document.getElementsByTagName('video')[0];
        var playAny = 0;
        for (x = 0; x < myTypes.length; x++)
        {
          var canPlay = myVideo.canPlayType(myTypes[x]);
          if ((canPlay=="maybe") || (canPlay=="probably"))
             { playAny = 1;
               lastPlayable=mySrc[x];
             }
         }
        if (playAny==0)
        {
        document.getElementById("video-player").innerHTML = noPlayableTypes;
        } else {
        myVideo.addEventListener('error',errorFallback,true);
        }
       }

   </script>
 </head>
 <body onload="checkPlaylist()" >
   <div id="video-player" align=center>
     <video controls >
     <source src="myMovie.mp4" type="video/mp4">
     <source src="myMovie.oga" type="video/ogg">
     <source src="myMovie.dvx" type="video/divx
     </video>
   </div>
  </body>
</html>
```

Resizing Movies to Native Size

If you know the dimensions of your movie in advance, you should specify them. Specifying the dimensions is especially important for delivering the best user experience on iPad. But you may not know the dimensions when writing the webpage. For example, your source movies may not be the same size, or sequential movies may have different dimensions. If you install a listener function for the `loadedmetadatata` event, you can resize the video player to the native movie size dynamically using JavaScript, as soon as the native size is known. If sequential movies may be different sizes, you can do this any time you change the source. Listing 2-7 shows how.

Listing 2-7 Resizing movies programmatically

```
<!DOCTYPE html>
<html>
  <head>
   <title>Resizing Movies</title>
   <meta http-equiv="content-type" content="text/html; charset=utf-8">
   <script type="text/javascript">

       // set height and width to native values
       function naturalSize() {
          var myVideo = document.getElementsByTagName('video')[0];
          myVideo.height=myVideo.videoHeight;
```

```
          myVideo.width=myVideo.videoWidth;
        }
       // install listener function on metadata load
        function myAddListener(){
         var myVideo = document.getElementsByTagName('video')[0];
         myVideo.addEventListener('loadedmetadata',naturalSize,false);
        }
   </script>
  </head>
  <body onload="myAddListener()" >

      <video src="http://homepage.mac.com/qt4web/myMovie.m4v"
             controls >
      </video>

  </body>
</html>
```

Adding CSS Styles

Because the `<video>` element is standard HTML, you can modify its appearance and behavior using CSS styles. You can modify the opacity, add a reflection, rotate the video in three dimensions, and much more. You can also use CSS styles to enhance elements that interact with audio or video, such as custom controllers and progress bars.

Note: You may want to create your own stylish movie controller—including a progress bar and a time scrubber—that slides smoothly out of the way when not in use. To see an example of just that, download the *ConcertDemo* sample code from `http://developer.apple.com/safari/library/samplecode/HTML5VideoPlayer/index.html`. The sample code contains complete HTML, JavaScript, CSS, and video.

This chapter illustrates some methods of adding CSS styles to video, as well as how to add a styled progress bar and play/pause button. For more information on using CSS styles in Safari, see *Safari CSS Visual Effects Guide*, *Safari Graphics, Media, and Visual Effects Coding How-To's*, and *Safari CSS Reference*.

Adding CSS Styles to Video

The example in Listing 3-1 (page 30) shows how to rotate video in three dimensions, hide and reveal content underlying the video, and change the opacity to composite video over a background. All the changes can be applied dynamically while the video is playing. The example uses a combination of HTML, CSS, and JavaScript to achieve these effects. Each part is explained individually in the following sections.

The Example HTML Code

In the body of the document, the example first defines content to go under the video:

```
<div id="back">
<p style="position:relative;top:50%;" align="center">
This is hidden behind the video
</p>
</div>
```

The content is a `div` element just a little smaller than the video with text centered horizontally and vertically.

The example then adds the video, inside a `div` element named “overlay”:

```
<div id="overlay" >
<video class="video-player" id="player"
       height="348" width="680" controls
       src="myMov.mp4"
 >
</video>
```

Because the overlay is declared after the background, CSS naturally puts it in the layer above the background if they overlap later.

Finally, the example adds buttons to flip the video, reveal the hidden content, and fade the video in or out by changing its opacity, along with a button that displays the current opacity:

```
<input type=button value="flip backward" onclick="flipVideo()"
                  id="flipflop">
<input type=button value="show back" onclick="toggleBack()"
                  id="backer">
<input type=button value="fade out" onclick="fadeTo(0)">
<input type=button value="fade mid" onclick="fadeTo(.5)">
<input type=button value="fade in" onclick="fadeTo(1)">
<input type=button value="opacity: 100%" id="showfade">
</div>
```

The "overlay" `div` element is then closed—it includes the buttons and the video, so the buttons stay with the video when they are repositioned.

The Example CSS Styles

In the `style` section of the head, the example sets the background content to the size of the video, gives it a background color, and makes it hidden:

```
#back{
  height:330px;
  width:670px;
  background-color:gray;
  visibility:hidden; }
```

The example then positions the `div` containing the video and buttons at the top of the page (with a 5-pixel pad for aesthetics), covering the background content:

```
#overlay{
 position:absolute;
 top:5px; }
```

Finally, the example adds styles to the video itself. First it adds a background color. Then it specifies that any change in the `webkit-transform` or `opacity` properties should be animated by specifying them in `webkit-transition-property`. The animation timing is set to start gradually (ease in) and take one second to complete:

```
#player {
 background-color:black;
 -webkit-transition-property: -webkit-transform, opacity;
 -webkit-transition-timing-function: ease-in;
 -webkit-transition-duration: 1s; }
```

The CSS style declarations begin with # because they are each applied to a single element, based on its unique `id` attribute. Style declarations that begin with a dot (for example `.back`) are applied based on the `class` attribute, which multiple elements may share.

The Example JavaScript Functions

The JavaScript section of the head defines three functions: `flipVideo()`, `toggleBack()`, and `fadetTo(val)`.

The `flipVideo()` function rotates the video in 3D, flipping it 180° around the vertical axis by setting the `style.webkitTransform` property to `rotateY(0deg)`, or `rotateY(180deg)`. Because the function is a toggle, it checks the button text to see what it should do—flip forward or flip backward—then changes the text of the button to show what it does next time.

```
function flipVideo() {
  var myVideo = document.getElementById('player');
  var myButton = document.getElementById('flipflop');
  if (myButton.value=="flip backward"){
    myVideo.style.webkitTransform = "rotateY(180deg)";
    myButton.value="flip forward";
  }else{
    myVideo.style.webkitTransform = "rotateY(0deg)";
    myButton.value="flip backward";
} }
```

The `toggleBack()` function also rotates the video in 3D, this time 90° around the vertical axis, so it is edge-on to the viewer and therefore invisible. It also changes the `style.visibility` property of the background content from `hidden` to `visible`. Because the function is a toggle, it then changes the text of the button from "show back" to "hide back". When called again, it resets the Y rotation to 0° and sets the background content back to `hidden`:

```
function toggleBack() {
  if (myButton.value=="show back"){
    myVideo.style.webkitTransform = "rotateY(90deg)";
    myButton.value="hide back";
    backContent.style.visibility="visible";
   }else{
    myVideo.style.webkitTransform = "rotateY(0deg)";
    myButton.value="show back";
    backContent.style.visibility="hidden";
 } }
```

The `fadeTo(opacityVal)` function takes one parameter: the desired opacity value. It sets the video's `style.opacity` property to that value and changes the text in a button to display the current opacity as a percent.:

```
function fadeTo(opacityVal) {
  document.getElementById('player').style.opacity=opacityVal;
  var showPercent= ("opacity "+ (opacityVal * 100) + "%");
  document.getElementById('showfade').value= showPercent;
}
```

Notice that JavaScript is not used to animate the effects on the video. JavaScript just changes the style property values. The animation parameters are specified in CSS and the effects are rendered automatically.

Code Example: Adding Style to Video

Listing 3-1 is a complete working example, incorporating all the material in the previous three section, illustrating how to combine HTML, CSS, and JavaScript to add style to video dynamically.

Listing 3-1 Adding dynamic 3D rotation, hiding, and opacity to video

```
<!DOCTYPE html>
<html>
  <head>
    <title>Flipping and Fading Video</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">

    <style>

    #back{
      height:330px;
      width:670px;
      background-color:gray;
      visibility:hidden;
      }

    #overlay{
     position:absolute;
     top:5px;
     }

    #player {
     background-color:black;
     -webkit-transform:rotateY(0deg);
     -webkit-transition-property: -webkit-transform, opacity;
     -webkit-transition-duration: 1s;
     -webkit-transition-timing-function: ease-in;
     }

    </style>

    <script type="text/javascript">

         function flipVideo() {
           var myVideo = document.getElementById('player');
           var myButton = document.getElementById('flipflop');
           if (myButton.value=="flip backward"){
             myVideo.style.webkitTransform = "rotateY(180deg)";
             myButton.value="flip forward";
           }else{
             myVideo.style.webkitTransform = "rotateY(0deg)";
             myButton.value="flip backward";
           }
         }

         function toggleBack() {
           var backContent=document.getElementById('back');
           var myVideo = document.getElementById('player');
           var myButton = document.getElementById('backer');
           if (myButton.value=="show back"){
             myVideo.style.webkitTransform = "rotateY(90deg)";
             myButton.value="hide back";
             backContent.style.visibility="visible";
            }else{
             myVideo.style.webkitTransform = "rotateY(0deg)";
             myButton.value="show back";
             backContent.style.visibility="hidden";
```

```
            }
           }

          function fadeTo(opacityVal) {
            document.getElementById('player').style.opacity=opacityVal;
            var showPercent= ("opacity "+ (opacityVal * 100) + "%");
            document.getElementById('showfade').value= showPercent;
          }

    </script>

  </head>
  <body style="background-color:#C0C0C0;">
       <div id="back">
       <p style="position:relative;top:50%;" align="center">
       This is hidden behind the video
       </p>
       </div>
       <div id="overlay">
       <video id="player"
              height="348" width="680" controls
              src="myMov.m4v"
        >
       </video>
       <br>
     <input type=button value="flip backward" onclick="flipVideo()"
                      id="flipflop">
     <input type=button value="show back" onclick="toggleBack()"
                      id="backer">
     <input type=button value="fade out" onclick="fadeTo(0)">
     <input type=button value="fade mid" onclick="fadeTo(.5)">
     <input type=button value="fade in" onclick="fadeTo(1)">
     <input type=button value="opacity: 100%" id="showfade">
     </div>
  </body>
</html>
```

Adding a Styled Progress Bar

As shown in "Using DOM Events to Monitor Load Progress" (page 20), it's fairly easy to monitor the movie loading process and show what percentage has downloaded using JavaScript alone. By integrating CSS styles, you can create a real progress bar.

The following example, Listing 3-2, creates a `div` element named `video-player` whose height is the height of the video plus the progress bar. It puts the `video` element inside the `video-player`, then adds a background bar by creating a nested `div` element and styling it with a size, color, and shadow. Inside the bar it adds a partly transparent progress indicator, positioning it on top of the background bar but setting its width to 0%. Because the bar and progress indicator are inside the `video-player` element, they inherit its width. That way when the progress indicator's width goes from 0 to 100%, it grows to the width of the video. The progress indicator is translucent, so the background bar shows through.

A listener function on the `progress` event updates the `width` property of the progress indicator. An installer function installs the listener and is called on page load.

Listing 3-2 Adding a styled progress bar

```
<!DOCTYPE html>
<html>
  <head>
    <title>Styled Progress Bar</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">

    <style>
    #video-player {
     width: 680px;
     height: 366px;
     background-color: black;
     }
    #background-bar {
     height: 18px;
     width: 680px;
     background-color: #D4D4D4;
    -webkit-box-shadow: 0px 2px 4px rgba(0,0,0,0.8);
    }
    #loader {
    height: 18px;
    width: 1%;
    opacity:.25;
    background-color: blue;
    }
    </style>

    <script type="text/javascript">
    function showLoad() {
    var myVideo = document.getElementsByTagName('video')[0];
    var soFar = parseInt(((myVideo.buffered.end(0) / myVideo.duration) * 100));
    myLoader=document.getElementById("loader");
    myLoader.style.width=(soFar + '%');
    }

    function myAddListener() {
    var myVideo = document.getElementsByTagName('video')[0];
    myVideo.addEventListener('progress',showLoad,false);
    }

    function playVideo() {
    var myVideo = document.getElementsByTagName('video')[0];
    myVideo.play();
    }

    </script>

  </head>
  <body onload="myAddListener()" >
      <div id="video-player">
       <video height="348" width="680" autoplay
        src="http://homepage.mac.com/qt4web/myMovie.m4v"
        >
       </video>
       <br>
       <div id="background-bar">
         <div id="loader">
```

```
        </div>
      </div>
    <div>
      <p> </p>
      <input type="button" value="play" onclick="playVideo()">
    </div>
  </body>
</html>
```

Note that the previous example includes a simple JavaScript play button. If the `controls` attribute is omitted and no JavaScript controls are provided, a user on the iPad has no way to play the movie.

Adding A Styled Play/Pause Button

The final example, Listing 3-3, creates a video-player element that surrounds the video with a black background and puts a play/pause button under the video, centered horizontally. The play/pause button is styled with 24-point white text on a dark grey background, at 50% opacity.

When the user presses the play/pause button, it plays or pauses the movie. It also toggles the button's value between ">" and "||". The opacity is set to 50% when the video is paused and 20% when it's playing, so the button is less distracting from the playing video.

Finally, a call to reset the play/pause button is added as an event listener to the video element's `ended` event.

Listing 3-3 Adding a styled play/pause button

```
<!DOCTYPE html>
<html>
  <head>
    <title>Styled Play/Pause Button</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">

    <style>
    #video-player {
     width:50%;
     margin-left:25%;
     margin-right:25%;
     background-color: black;
     }

   #videobutton {
    line-height: 24pt;
    border: 3px solid white;
    -webkit-border-radius: 16px;
    opacity: 0.5;
    font-size: 24pt;
    color: white;
    background-color: #404040;
    cursor: pointer;
    text-align: center;
    z-index: 1;
    opacity:.5;
    }
 </style>
```

```
 <script type="text/javascript">

         function pauseButton() {
             var myButton = document.getElementById("videobutton");
                myButton.value = "||";
                myButton.style.opacity=".2";
            }

         function playButton() {
             var myButton = document.getElementById("videobutton");
                myButton.value = ">";
                myButton.style.opacity=".5";
            }

         function playPause() {
              var myVideo = document.getElementsByTagName('video')[0];
              if (myVideo.paused) {
                myVideo.play();
                pauseButton();
              }else{
             myVideo.pause();
             playButton();
            }
         }

         function myAddListener() {
         var myVideo = document.getElementsByTagName('video')[0];
         myVideo.addEventListener('ended',playButton,false);
         }

</script>
</head>

<body onload="myAddListener()">
  <div id="video-player">
     <video src="http://homepage.mac.com/qt4web/myMovie.m4v"
            height="270" width="480">
     </video>
     <P align=center>
     <input id="videobutton" type=button onclick="playPause()" value="&gt;">
     </p>
  </div>
  </body>
</html>
```

Document Revision History

This table describes the changes to *Safari HTML5 Audio and Video Guide*.

Date	Notes
2010-03-18	Corrected typos in sample code and text. Minor changes throughout.
2010-03-09	New document showing how to use the HTML5 <audio> and <video> elements using HTML, JavaScript methods, DOM events, and CSS styles.

Document Revision History