HTML5 Video

Lazy Loading Youtube Embeds

After images the biggest payloads in my pages are Youtube embedded videos and since I already lazy load images I've decided to explore how to best lazy load embedded Youtube iframes; I embed Vimeo videos far less often so I'm not going to lazy load them.

> I have the following requirements for any lazy loading solution:
>
> - The video must be responsive
> - Any external script, stylesheets and assets must be less than 5kb in size

The original

This is what I normally do when working with video on the web and in my Wordpress blogs. I take the `iframe` element directly from Youtube and wrap it around a div with a class of `video` to style it accordingly

```html
<div class="video">
<iframe width="560" height="315"
  src="https://www.youtube.com/embed/w8zkLGwzP_4?rel=0"
  frameborder="0" allowfullscreen></iframe>
</div>
```

The CSS is simple and styles both the `iframe` and paragraphs inside the div.

```css
div.video iframe {
  clear: both;
  display: block;
  margin: 1em auto;
  max-width: inherit;
  text-align: center
}
```

```
div.video p {
  font-style: italic;
  font-weight: 700;
  margin-top: -.125em
}

iframe {
  margin: 1.5em 0
}
```

The issue we try to resolve with lazy loading is that each embedded video loads many additional resources whether we are interacting with the video or not. Those additional HTTP connections will slow down overall page load times and gives a lower Page Speed Score.

Furthermore the longer it takes for a page to display content to the user and become interactive the more likely the user it is to leave the page and not come back. Hence this experiment… Using digital inspiration’s lazy loading method as a model we’ll explore how to use it, what it does and whether it’s actually worth using it or not.

The lazy loaded version

> Lazy loading is a design pattern commonly used in computer programming to defer initialization of an object until the point at which it is needed. It can contribute to efficiency in the program’s operation if properly and appropriately used.
>
> — Wikipedia

In the context of a web page/application I use lazy loading to prevent an item from loading until it becomes visible on the screen (comes into the viewport). Once the video comes into view we wait for user’s input by either clicking or taping the video before playback begins.

We use the HTML below to indicate where to place the video and what video (using the video Youtube ID) to place there.

```
<div class="youtube-player" data-id="VIDEO_ID"></div>
```

The Javascript below will build three elements:

- A div element to hold the video iframe
- An img element containing the poster image and a play button
- The `iframe` element that contains the video and attributes like autoplay and no related videos at the end

The first part of the video creates the event listener, creates variable place holders and creates the first element a div with the id of the video we created.

```
document.addEventListener("DOMContentLoaded",
  function() {
    let div;
    let n;
    let v = document.getElementsByClassName("youtube-player");
    for (n = 0; n < v.length; n++) {
        div = document.createElement("div");
        div.setAttribute("data-id", v[n].dataset.id);
        div.innerHTML = loadThumb(v[n].dataset.id);
        div.onclick = loadIframe;
        v[n].appendChild(div);
    }
  });
```

`loadThumb` loads the poster image and the play overlay.

```
function loadThumb(id) {
  let thumb = '<img src="https://i.ytimg.com/vi/ID/hqdefault.jpg">';
  let play = '<div class="play"></div>';
  return thumb.replace("ID", id) + play;
  };
```

`loadIframe` builds the iframe element and populates it with the video we want, again by referencing its ID. It also provides attributes to control the frame around the video and whether we can go full screen with it.

```
function loadIframe() {
  let iframe = document.createElement("iframe");
  let embed = "https://www.youtube.com/embed/ID?autoplay=1&rel=0";
  iframe.setAttribute("src", embed.replace("ID", this.dataset.id));
  iframe.setAttribute("frameborder", "0");
  iframe.setAttribute("allowfullscreen", "1");
  this.parentNode.replaceChild(iframe, this);
  }
```

Finally, we use CSS to style the video and programmatically add the play button. as an overlay.

```
.youtube-player {
  position: relative;
  padding-bottom: 56.23%;
  /* Use 75% for 4:3 videos */
  height: 0;
  overflow: hidden;
  max-width: 100%;
  background: #000;
  margin: 5px;
}

.youtube-player iframe {
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  z-index: 100;
  background: transparent;
}

.youtube-player img {
  bottom: 0;
  display: block;
  left: 0;
  margin: auto;
  max-width: 100%;
  width: 100%;
  position: absolute;
  right: 0;
  top: 0;
  border: none;
  height: auto;
  cursor: pointer;
  -webkit-transition: .4s all;
  -moz-transition: .4s all;
  transition: .4s all;
}

.youtube-player img:hover {
  -webkit-filter: brightness(75%);
}

.youtube-player .play {
  height: 72px;
  width: 72px;
  left: 50%;
  top: 50%;
  margin-left: -36px;
  margin-top: -36px;
  position: absolute;
  background: url("../images/play.png") no-repeat;
  cursor: pointer;
}
```

What’s next?

This is the kind of project that would benefit greatly from ES6’s Template Literals or maybe even building a custom element for this. I may also want to turn this into a Wordpress function for my blogs.

Custom controls

One of the cool things about HTML5 video is that you can fully customize it to suit your needs and preferences. Dudley Storey has an interesting demo and tutorial on how to add custom controls for a video. I will take his idea and take it in a different direction… Rather than attach the controls to the video I'll create a control-panel like interface for a video.

When working in projects like this I create a minimal set of requirements. In this case the requirements are:

- Video still works if Javascript is not available
- Video is captioned and the captions can be toggled on and off
- Video can be played through keyboard commands

Some additional enhancements and ideas are at the end of the post. Now let's dive into code :-)

HTML

The HTML code is pretty straightforward. The root element for the demo is a section element with a class of `display-wrap`; it's purpose is to serve as the root of our flexbox layout.

The `videoPlayer` class div holds our video element. This is the core of the experiment with `mp4` and `webm` versions of the video and an Eglish version of our `vtt` captions. These are the elements that we'll manipulate with Javascript.

```
<section class='display-wrap'>
  <div class='videoPlayer'>
    <video id='video' controls poster='video/hololens.jpg'>

        <source
                src='video/hololens.mp4'
                type='video/mp4'>
        <source
                src='video/hololens.webm'
                type='video/webm'>
        <track kind='captions' src='video/hololens.en.vtt' srclang='en'
               label='English' id='english'>
    </video>
</div>
```

The last portion, the `control-panel` div holds the icons for the player controls. I wasn't able to find an icon for both captions on and off so I left them as text links. This is also a flexbox layout.

```
<div class='control-panel'>
    <h2>Video Control Panel</h2<h2>
    <div class='player-icons'>
        <a href='#' id='rewind'><img<div class='player-icons'>icons/rewin
        <a href='#' id='play'><img class='icon' id='playIcon' src='images/
        <a href='#' id='myStop'><img cl</a>'icon' src='images/icons/stop.
        <a href='#' id='forward'><img class='icon' src='images/icons/fast-
    </div>
    <div class='player-icons'>
        </a><!--<a href='#' id='reset'-->
        <a href='#' id='showCaptions'>Enable Captions</a>
        <a href='#' id='disableCaptions'>Disable Captions</a>
    </div>
  </div>
</section>
```

Javascript

The Javascript is very event driven and works by reacting to events that you've triggered. As usual, I'll break the Javascript in sections and describe what each one does or any relevant thing about the code.

The first section holds place holders for all the objects that will afect the video player.

```
// Event Listeners For Play/Pause Button
```

```
let video = document.getElementById('video');

let play = document.getElemen'video'play');
let playIcon = document.getElementById('playIcon');

let myStop = document.getElementById('myStop');

let rewind = document.getElementById('rewind');
let forward ='playIcon'getElementById('forward');

let showCaptions = document.getElementById('showCaptions');
let hideCaptions = document.getElementById('hideCaptions');

'forward'// Remove native controls
video.removeAttribute('controls');
```

The second part is our keyboard navigation. Using a keydown event listener we intercept multiple keys using a switch statement.

If the key code is 32 (space bar) or 13 (enter) then we trigger the play sequence: if the video is not playing (represented by the pause state) then we play the video and changge the icon to the pause icon. if the video is playing then we pause it and change the icon to play.

If the key code is 37 (left arrow) then we seek back 30 second in the video.

If the key code is 39 (right arrow) then we seek forward 30 seconds in the video.

In my original code I was using `keypress` instead of keydown. For some reason the code for left and right arrows was not working. Trying to figure out why.

```
// Event handler for keyboard navigation
window.addEventListener('keydown', (e) => {
  switch (e.which) {
    case 32:
    case 13:
      e.preventDefault();
      if (video.paused) {
          video.play();
          playIcon.src = 'images/icons/pause.svg';
      } else {
          video.pause();
          playIcon.src = 'images/icons/play-button.svg';
      }
      break;

    case 37:
      skip(-30);
      break;

    case 39:
      skip(30);
      break;
  }

});
```

What happens when the user click the play button is similar to what happens when they press the space bar or enter key:

- If the video is not playing (represented by the pause state) then we play the video and changge the icon to the pause icon
- If the video is playing then we pause it and change the icon to play

```
// play handler
play.addEventListener('click', (e) =>'click'// Prevent Default Click Actio
  e.preventDefault();
  if (video.paused) {
      video.play();
      playIcon.src = 'images/icons/pause.svg';
  } else {
      video.pause();
      playIcon.src = 'images/icons/play-button.svg';
  }
});
```

Working with captions proved very difficult. At first I had done something like the commented code below. The first part worked properly but not the second one... so I had to break it into separate items

```
//  captions.addEventListener('click', (e) => {
//
//      e.preventDefault();
//      console.log(video.textTracks[0].mode);
//      if (video.textTracks[0].mode = 'showing') {
//          video.textTracks[0].mode = 'hidden';
//          console.log(video.textTracks[0].mode);
//      } else {
//          video.textTracks[0].mode = 'showing';
//          console.log(video.textTracks[0].mode);
//      }
//
//  })
```

Each of the links uses attributes from the VTT Text Track to show or hide the video where appropriate. I'm still looking at merging these two into a single event listener like I did for play but haven't been able to find an easy way to do it.

```
// Show captions
showCaptions.addEventListener('click', (e) => 'click'eventDefault();
 video.textTracks[0].mode = 'showing';
});

'showing'// Hide captions
disableCaptions.addEventListener('click', (e) => {
  e.preventDefault();
  video.textTracks[0].mode = 'hidden';
});
```

In this demo I've made a very important difference between play/pause and stop. Pause will keep the play head at the current location so clicking the button again will resume play without interruption.

The stop button will stop playback, change the play button icon to play (regardless of its previous status) and reset the playback to the beginning of the video.

```
// Stop and reset
myStop.addEventListener('click', (e) => {
  video.pause();
  playIcon.src = 'images/icons/play-button.svg';
  video.currentTime = 0;
  video.load();
});
```

The seeking functions both forwards and backwards using a small convenience function, `skip` to adjust the timeline forward or backwards by the specifiec ammount.

```
// Back 30
rewind.addEventListener('click', ('click'
  skip(-30);
});

// Forward 30
forward.addEventListener('click', (e) => {
  skip(30);
});

function skip(value) {
  video.currentTime += val'click'
```

Most of the CSS deals with layout and making the video a responsive one. We first define our outer display as a `flex` container and give some basic styles to look like a separate unit… we also make the control panel take 1 portion of the flex layout.

```
.display-wrap {
    display: flex;
    flex-flow: row wrap;
}

.control-panel {
    border: 1px solid black;
    height: 20vh;
    flex: 1;
    padding: 1em;
}
```

The `.videoPlayer` classes make the video responsive and size it to be 16:9. This also takes 4 units in the flex layout.

```
.videoPlayer {
    position: relative;
    padding-bottom: 56.23%;
    /* Use 75% for 4:3 videos */
    height: 0;
    overflow: hidden;
    max-width: 100%;
    background: #000;
    margin: 5px;
    flex: 4;
}

.videoPlayer #video {
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    z-index: 100;
    background: transparent;
}
```

The last piece of CSS we use is to lyout the icons using flexbox and size each individual icon appropriate. Initially I used SVG icons but I could not get them to size like I wanted inside a flexbox grid so I reverted back to using PNG. I may revisit this using the icons as background images.

```css
.player-icons {
    display: flex;
    flex-flow: row;
    justify-content: space-between;
}

.icon {
    border: 0;
    flex: 1;
    height: auto;
    width: 32px;
}
```

Changes, refinements and future ideas

The code in the page works fine for a single video in a page. In future iterations of the project I'd like to do a few things:

Convert this into a class for better reusability. One thing I'd like to change is moving the code into a class so I can instantiate one per video in a page. Some of the challenges are learning more about classes and how to instantiate event handlers from inside a class (if it's possible at all).

Naming conventions for multiple videos. If using more than one video for the page then I need to come up with a convention for the IDs, ideally onne that would allow me to use string literal templates when triggering the events. This goes together with using classes.

Using background images instead of just regular icons. I really want to use SVG for the icons but was unable to use it. As part of a later iteration I want to explore using background images (inserted in the CSS code) instead of regular icons. This will make the code harder to read and may cause accessibility problems but it's worth researching.

Playlists

Another idea is how to create playlists like those on Youtube but without having to code the entire interface from scratch. Dudley Storey, again, proposed a solution using CSS `display: table` and a little Javascript magic.

I’ve taken the layout from Storey’s article as is (I’m still learning about `display: table` and related CSS) and enhanced the Javascript with some of my working ideas. The two main constraints:

- It must work without Javascript; The user must be able to view the videos when there Javascript is not available
- It must work without the mouse using only keyboard

THe HTML uses a figure as the container for the playlist. The two children are a `video` element with the traditional sources. We make sure to leave the controls visible so people who choose to work with the standard video controls.

In the `figcaption` element we add links and images for the other videos available in the playlist.

```
<!DOCTYPE html>
<html lang="en"<html lang="en">ta charset="UTF-8">
    <title>Video Playlist</title>
    <link rel="st<title>t" href="styles/styles.css">

</head>
<body>
<figure id="video_player">
    <video controls poster="images/SAO-Ordinal-Scale-Trailer1.jpg" id="vi
    </head>urce src="video/SAO-Ordinal-Scale-Trailer1.mp4" type="video/mp
        <source src="video/SAO-Ordinal-Scale-Trailer1.webm" type="video/we
    </video>
    <figcaption>
        <a href="video/SAO-Ordinal-Scale-Trailer2.mp4"><img src="images/S
        <a href="video/SAO-Ordinal-Scale-Tr<source src="video/SAO-Ordinal-
        <a href="video/SAO-Ordinal-Scale-Trailer4.mp4"><img src="images/S
    </figcaption>
</figure>
<script src="scripts/script.js"></a></script>
</body>
</html>
```

Javascript

The first portion of the script is a shortcut. Rather than attach the same event to multiple elements manually we define the elements we want to attach the event to, in this case all the a elements inside `figcaption` and loop through them attacking the handler function to the `onclick` event for each link.

```
let links = [...document.querySelectorAll('figcaption a')];

for (let i=0; i<links.length; i++) {
    links[i].onclick = handler;
}
```

Next we define the `handler` function that we've attached to the anchors in the page.

For each element we:

1. Prevent the default click. We want this function to handle the click rather than the browser's default link handling mechanism.
2. Capture a reference to the link's `href` attribute
3. Extract the file name by creating a substring of the `href` attribute we generated in the prvious step
4. Create a reference to the video element and remove the poster attribute. We don't want the poster from the previous movie to overlap the new video
5. We create a node list (**not an array**) of the source children elements
6. Assign the mp4 version of the video to the first child and the webm version of the video to the second child source element
7. Load the video
8. Play

```
function handler(e) {
    e.preventDefault(); // 1
    let videotarget = this.getAttribute("href"); "href"// 2et filename =
 '.'// 3video = document.getElementById("video"); // 4
    "video"// 4eAttribute("poster"); // 4
    let sou"poster"// 4nt.querySelectorAll("#video_player video source");
    source[0].src ="#video_player video source"// 5rce[1].src = filename
```

```
  ".webm"// 6
    source[1].src = filename + ".webm"; // 6
    video.load(); // 7
    video.play(); // 8
}
```

I've copied the keyboard event handler from another project to make sure we meet the keyboard accessibility requirement.

1. If the user presses either the space bar or enter key then toggle playback, if the video is paused then play it and if it's playing then pause it.
2. If the user presses the left key then seek 5 seconds backwards on the video
3. If the user presses the right key then seek 5 seconds forward on the video

For the arrow keys we use a utility function to seek the video.

```
// Event handler for keyboard navigation
window.addEventListener('keydown', (e) => {
    switch (e.which)'keydown'  case 32:  // 1
        case 13:  // 1
            e.preventDefault();
            if (video.paused) {
                video.play();
            } else {
                video.pause();
            }
            break;

        case 37:  // 2
            skip(-5);
            break;

        case 39:  // 3
            skip(5);
            break;
    }

});
```

```
function skip(value) {
    video.currentTime += value;
}
```

CSS

The CSS uses `display: table` to layout the content in a way that is backwards compatible. The video (the `#video_player` element)is the main component of our ‘table’ layout and will take 2/3rd of the space while each of the video thumbnails (`figcaption a` elements) are stacked in the remaining width of the element.

This is similar to using a table but not quite the same: The table element in HTML is a semantic structure. The `table` value for display is an indication of how the content should be displayed and has nothing to do with the structure of the content like the `table` tag does

```
#video_player {
    display: table;
    line-height: 0;
    font-size: 0;
    background: #fff;
}
#video_player video,
#video_player figcaption {
    display: table-cell;
    vertical-align: top;
}
#video_player figcaption {
    width: 25%;
}
#video_player figcaption a {
    display: block;
    opacity: .5;
    transition: 1s opacity;
}
```

```css
#video_player figcaption a img,
figure video {
    width: 100%;
    height: auto;
}
#video_player figcaption a:hover {
    opacity: 1;
}
```

One further refinement

Right now there is no way to navigate between videos and now way to play the first video again after you navigate to the thumbnails. It'll require some additional Javascript like the one in this post by Dudley Storey.

Morphing play button for video

Youtube has this little UI trick that I love. When you click the play button on a video the button will toggle morphing between play and pause icons. It's not a big thing but it's an additional cue for the user about the status of the video.

When I started to research this part of the video project project I found that most examples use SMIL inside a button element. SMIL is a companion to SVG that allows for animations and fairly complex animations and UI effects.

The downside of SMIL is that its shelf life is fairly limited. Chrome put a deprecation notice for SMIL in the console since version 45 (and Opera 32) dated April, 2015. The deprecation notice doesn't indicate when they will remove the feature which makes it harder to plan when/how to implement fallbacks... Yes, I know that we should work on implementing this correctly from the beginning but it's another library to add to the page and I've never been a fan of doing that just for cosmetic UI effects but in this case I will make an exception as I'll be using it for search bar tool animations too.

Equivalent APIs like the Web Animations API that would replace SMIL with an equivalent set of features is still at the Working Draft stage in the W3C process so, I hope, SMIL will survive until the spec is finalized and maybe longer.

The original

Dudley Storey's post was the first implementation of the feature as I've wanted to use it. I've taken the example and tweaked it with additional functionality.

HTML

```
<section class='display-wrap'>
  <div id='videoPlayer'>
    <video id='video' controls poster='video/hololens.jpg'>
      <source
        src='video/hololens.webm'
        type='video/webm'>
      <source
        src='video/hololens.mp4'
        type='video/mp4'>

      <track kind='subtitles' src='video/hololens.en.vtt' srclang='en'
       label='English' id='english'>
    </video>
    <button>
      <svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 50 50"
        id="playpause" xmlns:xlink="http://www.w3.org/1999/xlink">
        <title>Play</title>

        <polygon points="12,0 25, 11.5 25, 39 12, 50" id="leftbar" />
        <polygon points="25,11.5 39.7,24.5 41.5,26 39.7,27.4 25,39" id="r

        <animate to="7,3 19,3 19,47 7,47" id="lefttopause" xlink:href="#le
          attributeName="points" dur=".3s" begin="indefinite" fill="freeze
        <animate to="31,3 43,3 43,26 43,47 31,47" id="righttopause" xlink
          attributeName="points" dur=".3s" begin="indefinite" fill="freeze

        <animate to="12,0 25,11.5 25,39 12,50" id="lefttoplay" xlink:href=
          attributeName="points" dur=".3s" begin="indefinite" fill="freeze
        <animate to="25,11.5 39.7,24.5 41.5,26 39.7,27.4 25,39" id="right
          attributeName="points" dur=".3s" begin="indefinite" fill="freeze
      </svg>
    </button>
  </div>
</section>
```

Javascript

```
let container = document.getElementById('videoPlayer');
let video = document.getElementById('video');

var playpause = document.getElementById("p'video'e");
var lefttoplay = document.getElementById("lefttoplay");
var righttoplay = document.getElementById("righttoplay");
var lefttopause = document.getElementById("lefttopause");
var righttopause = do"lefttoplay"ementById("righttopause");

"righttopause"// Remove native controlseAttribute('controls');

// play handle'controls'// play handler for control paneltener('click',
  playpause.style.display = "block";

  if (video.paused) {
    video.play();
    playpause.classList.add("playing");
    lefttopause.beginElement();
    righttopause.beginElement();
  } else {
    video.pause();
    lefttoplay.beginElement();
    righttoplay.beginElement();
    playpause.classList.remove("playing");
  }
});
// Event handler for key'click'// Event handler for keyboard navigation
window.addEventListener('keydown', (e) => {
  switch (e.which) {
    case 32:
    case 13:
      e.preventDefault();
      if (video.paused) {
        video.play();
        playpause.classList.add("playing");
        lefttopause.beginElement();
        righttopause.beginElement();
      } else {
        video.pause();
        lefttoplay.beginElement();
        righttoplay.beginElement();
        playpause.classList.remove("playing");        }
      break;
  }

});
```

CSS

```
* {
  box-sizing: border-box;
}
body {
  background: #333;
  margin: 3rem;
}
#videoPlayer {
  position: relative;
  font-size: 0;
  width: 50%;
  margin: 0 auto;
}
#videoPlayer video,
#videoPlayer button {
  width: 100%;
  height: auto;
}
```

```
#videoPlayer button svg {
  width: 55%;
  margin: 0 auto;
}

#videoPlayer video,
#videoPlayer button {
  position: absolute;
  top: 0;
}

#videoPlayer button {
  background: transparent;
  outline: none;
  border: none;
  cursor: pointer;
}

#videoPlayer button svg {
  fill: #fff;
  padding: 3rem;
  transition: .6s opacity;
}

#playpause {
  display: none;
}

.playing {
  opacity: 0;
}
```

Vestibular Disorders, Reduced Motion Media Query and Video Backgrounds

I wasn't aware of Vestibular Disorders being an issue with animations on the web but it's a big part of the disabilities we need to consider when working on the web. It's important to remember that…

> People with vestibular disorders have a problem with their inner ear. It affects their balance as well as their visual perception of their world around them.
>
> Sometimes the sensation lasts only a short while, but others can suffer it for years. Walking becomes a challenge and they have a constant risk of falling. Concentration is diminished leaving the sufferer unfocused and often unproductive. It is often viewed as a "hidden" disability because it has no outward showing symptoms.

So we really want to avoid that kind of problems for our users.

Currently we can use the `preferes-reduced-motion` media query to test if the reduced motion settings are enabled (Safari only) and reduce or disable animations from your page. If the browser doesn't understand the query it'll skip the query and its content altogether.

```css
@media (prefers-reduced-motion) {
  .background {
    animation: none;
  }
}
```

Val Head's [Desgigning Safer Web Animations For Motion Sensitivity] she outlines some examples of sites that cause issues for users who experience Vestibular Disorders, some of the issues that trigger Vestibular Disorders and some solutions to address these problems.

Perhaps the most important thing to learn about learning about accessibility and the web is the closing quote on Val's article:

> On the web, more than in any other medium, the flexibility and control are there for you to design creatively and responsibly at the same time. We absolutely can innovate and push the web forward designing kick-ass interface animations while still being responsible designers. As a web animator, you can have your animation cake and eat it too—with a little creative thinking.

The `playsinline` attribute and iOS

Moving on we'll look at a (sort of) new attribute for iOS video playback: `playsinline`.

Older versions of iOS required you to tap the video before playback would begin. This was done to prevent unnecessary battery usage and to avoid random, and sometimes multiple, video playback on the page if autoplay was enabled for the videos.

iOS 7 introduced `webkit-playsinline` as a way to relax the requirement for video playback with a gesture.

With iOS 10 Apple has further relaxed the requirements for automatic video playback. The short version of the new video policies for iOS 10 is as follows:

- `<video autoplay>` elements will now honor the autoplay attribute, for elements which meet the following conditions:
    - `<video>` elements will be allowed to autoplay without a user gesture if their source media contains no audio tracks
    - `<video muted>` elements will also be allowed to autoplay without a user gesture
    - If a `<video>` element gains an audio track or becomes un-muted without a user gesture, playback will pause
    - `<video autoplay>` elements will only begin playing when visible on-screen such as when they are scrolled into the viewport, made visible through CSS, and inserted into the DOM
    - `<video autoplay>` elements will pause if they become non-visible, such as by being scrolled out of the viewport

- `<video>` elements will now honor the play() method, for elements which meet the following conditions:
    - `<video>` elements will be allowed to `play()` without a user gesture if their source media contains no audio tracks, or if their muted property is set to true
    - If a `<video>` element gains an audio track or becomes un-muted without a user gesture, playback will pause
    - `<video>` elements will be allowed to play() when not visible on-screen or when out of the viewport
    - `video.play()` will return a Promise, which will be rejected if any of these conditions are not met
- On iPhone, `<video playsinline>` elements will now be allowed to play inline, and will not automatically enter fullscreen mode when playback begins
    - `<video>` elements without playsinline attributes will continue to require fullscreen mode for playback on iPhone
    - When exiting fullscreen with a pinch gesture, `<video>` elements without playsinline will continue to play inline

So, now that we know how to auto play a video in mobile (at least some of them) and desktop we'll dive into background videos.

Chrome for Android

As of version 53 Chrome for Android supports muted autoplay on mobile. This means that the background video will work in Chrome as wel as in Firefox and UC Browsers where it has worked without a problem (Chrome was the only browser that restricted video autoplay in Android devices) but now that Chrome plays the same game we get wider support.

> Some things to remember when testing the feature with Chrome in Android:

- From an accessibility viewpoint, autoplay can be particularly problematic. Chrome 53 and above on Android provides a setting to disable autoplay completely at the OS level
- Autoplay for audio is disabled on Chrome on Android, muted autoplay doesn't make sense for audio
- There is no autoplay if Data Saver mode is enabled. If Data Saver mode is enabled, autoplay is disabled in Media settings
- Muted autoplay will work for any visible video element in any visible document, iframe or otherwise

- To take advantage of the new behaviour, you’ll need to add muted as well as autoplay

Background videos

The idea of using a video as a background to text is intriguing. I like the idea of providing additional context using motion video rather static images. At the same time we also have to be careful and mindful of how we use the video so as not to trigger vestibular disorders and keeping in mind that the video will not have audio so we can’t rely on an audio content for the background.

Given those constraints we can still do video background.

We build the video element with playsinline, autoplay and muted attributes. These attributes will make sure that the video will autoplay in iOS by respecting the requirements for autoplay in iOS 10. I guess if I wanted to be absolutely sure I’d also include `webkit-playsinline` to account for older versions of iOS.

```html
<video poster="images/tron-bg.jpg" id="bgvid"
       playsinline autoplay muted>
  <source src="video/tron-bg.webm" type="video/webm">
  <source src="video/tron-bg.mp4" type="video/mp4">
</video>
```

This is half the magic, the other half happens in CSS.

CSS

In CSS we style the content to make sure that it works reliably across browsers. We do a quick margin reset and set the background for the page to white. Users should not see the white background under any circumstance so the big red flag appears when they do.

```css
body {
    margin: 0;
    background: #fff;
}
```

We then set the video in the page and make it full width and full height. This is a combination of methodologies to center content:

- We center the content using absolute positioning and then translate the content up and to the left
- We set width and height to auto and constrain it to 100% of the width and height of the window

It makes the content fixed so that it won’t matter how large the content is the video will not scroll.

It gives the video a negative z-index lower than any other content on the page.

finally we give the video a background element and make it the same as the poster and the first frame of the video.

```
video {
    position: fixed;

    top: 50%;
    left: 50%;
    transform: translateX(-50%) translateY(-50%);

    min-width: 100%;
    min-height: 100%;
    width: auto;
    height: auto;

    z-index: -1000;

    background: url('../images/tron-bg.jpg') no-repeat;
    background-size: cover;
    transition: 1s opacity;
}
```

`stopfade` is a class that only holds opacity. When we get to Javascript we’ll toggle the class to produce an animation like effect of transforming opacity.

```
.stopfade {
    opacity: .5;
}
```

The remaining selectors control the content area laid over the video.

```
#content {
    font-family: "Roboto""Roboto"Sans", sans-serif;
    font-weight:100;
    background: rgba(0,0,0,0.3);
    color: white;
    padding: 2rem;
    width: 50%;
    margin:2rem;
    float: right;
    font-size: 1.2rem;
}
h1 {
    font-size: 3rem;
    text-transform: uppercase;
    margin-top: 0;
    letter-spacing: .2rem;
}
#content button {
    display: block;
    width: 80%;
    padding: .4rem;
    border: none;
    margin: 1rem auto;
    font-size: 1.3rem;
    background: rgba(255,255,255,0.23);
    color: #fff;
    border-radius: 3px;
    cursor: pointer;
    transition: .3s background;
}
#content button:hover {
    background: rgba(0,0,0,0.5);
}

a {
    display: inline-block;
    color: #fff;
    text-decoration: none;
    background:rgba(0,0,0,0.5);
    padding: .5rem;
    transition: .6s background;
}

a:hover{
    background:rgba(0,0,0,0.9);
}
```

Finally we use two media queries to control what happens when the width of the window hits certain break points. If the window is smaller than 500 pixels wide we change the width of the content area to 70% of the width of the window.

When the width of the device is no more than 800 pixes then we remove the video from the page and add the poster image as a background element for the root element of the page (HTML)

```
@media screen and (max-width: 500px) {
    div{width:70%;}
}

@media screen and (max-device-width: 800px) {
    html {
      background: url('../images/tron-bg.jpg')
                  #000 no-repeat center center fixed;
    }

    #bgvid {
      display: none;
    }
}
```

Javascript

The Javascript handles interactivity and the special case of reduced motion.

After defining constants for the video and the play/pause button we handle the reduced motion media query match. `window.matchMedia` is the programmatic way to test if a media query matches the current environment. if it does then we remove the `autoplay` attribute of the video element; we don't want the video to play automatically if it may cause problems for our users; we then set the button's text to paused to indicate the video is not playing.

```
const vid = document.getElementById('bgvid');
const pauseButton = document.querySelector('#content button');

if (window.matchMedia('(prefers-reduced-motion)').matches) {
  vid.removeAttribute('autoplay');
  vid.pause();
  pauseButton.innerHTML = 'Paused';
}

function vidFade() {
  vid.classList.add('stopfade');
}
```

The rest of the script sets up events for the user to interact with.

When the video ends we want to pause the video and call the `vidFade()` function to change the opacity of the video by toggling a CSS class on and off.

```
vid.addEventListener('ended', function()
{
// only functional if 'loop' is removed
  vid.pause();
// to capture IE10
  vidFade();
});
```

Next we register a click event handler for the video and toggle between play and paused states using the button at the end of our content session. We also register a `keypress` event to handle keyboard pausing using space and enter.

```
pauseButton.addEventListener('click', () => {
  vid.classList.toggle('stopfade');
  if (vid.paused) {
   'stopfade');
    pauseButton.innerHTML = 'Pause';
  } else {
    vid.pause();
    pauseButton.innerHTML = 'Paused';
  }
});


'Pause'// Event handler for keyboard navigation
window.addEventListener('keypress', (e) => {
  switch (e.which) {
    case 32:
    case 13:
      e.preventDefault();
      if (vid.paused) {
        vid.play();
        pauseButton.innerHTML = 'Pause';
      } else {
        vid.pause();
        pauseButton.innerHTML = 'Paused';
      }
      break;
  }
});
```

Final thoughts

This is a nice way to enhance the the content of a page however there are a few things we need to keep in mind:

- Don’t just use this technique because you can. The video should enhance the message of your content or it’s just a distraction

- The video will autoplay, but it should be muted by default; ideally, it should not include sound at all
- Be mindful of mobile devices: many phones and tablets disable autoplay on videos to save bandwidth and battery. See the section on `playsinline` for a discussion on how this requirement is relaxed in iOS 10 and `Chrome for Android` for what Chrome supports
- Consider the video's length is important
  - If it's too short a video can feel repetitive (as most such videos will be set to loop)
  - If the video is too long it becomes a narrative unto itself, and deserves to be a separate design element
- Accessibility is essential: Make sure that any text you place on top of the video has a high contrast ration to the video
  - Users should have easy access to a UI control to pause the video
  - Ideally, the video should play through only once.
- Bandwidth is a big deal. The video needs to be small, and compressed as effectively as possible
  - At the same time, it needs to scale across different devices and their associated screens
  - For high end experiences you may consider (unencrypted) DASH video with multiple bitrates to serve different devices

Links and resources

- Understanding Vestibular Disorders
- Designing Safer Web Animation For Motion Sensitivity
- An Introduction to the Reduced Motion Media Query
- The A11Y Project

Revisiting Video Encoding: MP4 and WebM

Introduction

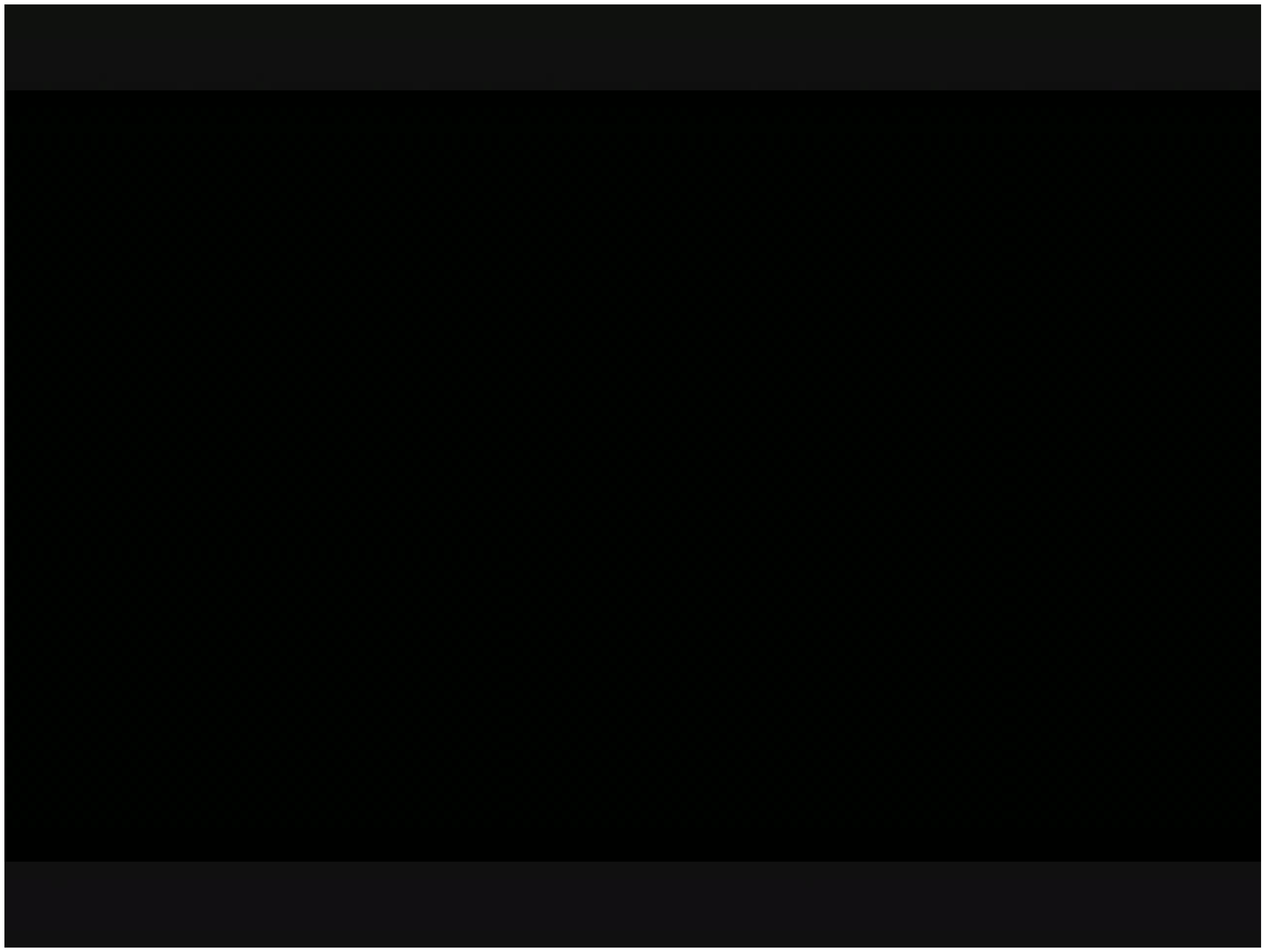
When HTML first introduced the `video` tag I was pumping my fist in joy. No more plugins to play video content. It was as simple as creating marlup like the one below to play and video in an MP4/ACC container with Egnlish and Swedish subtitles that can be changed as needed.

```html
<!-- Video with subtitles -->
<video src="foo.mp4" poster="foo-poster.png"
       width="640" height="480" controls>
  <track kind="subtitles" src="foo.en.vtt" srclang="en" label="English">
  <track kind="subtitles" src="foo.sv.vtt" srclang="sv" label="Svenska">
</video>
```

But it wass never as simple as it looked. Because there was no standard video format for HTML5 video, different browsers supported different container formats and different audio and video codecs. So the video turned into something like this:

```html
<video height="480" width="640" controls
  poster="https://archive.org/download/WebmVp8Vorbis/webmvp8.gif" >
  <source
    src="https://archive.org/download/WebmVp8Vorbis/webmvp8.webm"
    type="video/webm">
  <source
    src="https://archive.org/download/WebmVp8Vorbis/webmvp8_512kb.mp4"
    type="video/mp4">
  <source
    src="https://archive.org/download/WebmVp8Vorbis/webmvp8.ogv"
    type="video/ogg">
  Your browser doesn't support HTML5 video tag.
</video>
```

which produces the following video player:



Your browser doesn't support HTML5 video tag.

Each `source` element loads a different version of the video encoded with a different set of audio and video codecs. These files must be encoded separately and hosted separately.

There are also patent issues around MP4/h264 and ACC codecs. The MPEG Licensing Authority create a “patent pool” of essential technologies for MP4 encoding and decoding.

I had hoped that the new HEVC/h265 technology would not be encumbered by MPEG-LA style patent trolls but it was too much, apparently, as MPEG-LA already has an HVEC patent pool

So the fight has remained a stalemate with Mozilla and Opera on one side who refuse to pay the MP4 licensing fee and Microsoft, Google and Apple who have caved in and support MPEG4 playback as part of their HTML5 video implementations.

So, if it’s not MPEG4 or HVEC/h265 then what alternatives do we have available?

While Google implements MPEG4 in Chrome it has not remained static in the video codec front. In 2009 Google purchased On2 Technologies and have worked hard to make VP8, VP9 and its successor, WebM

MPEG-LA must have seen the benefit of VP8 becausse they began forming a patent pool for the technology. Google didn’t like that and the conflict ended with an agreement that would remove the MPEG-LA as a factor in VP8 licensing so that Google can continue to offer the code free and unencumbered for personal and commercial use, for now.

Why is this important?

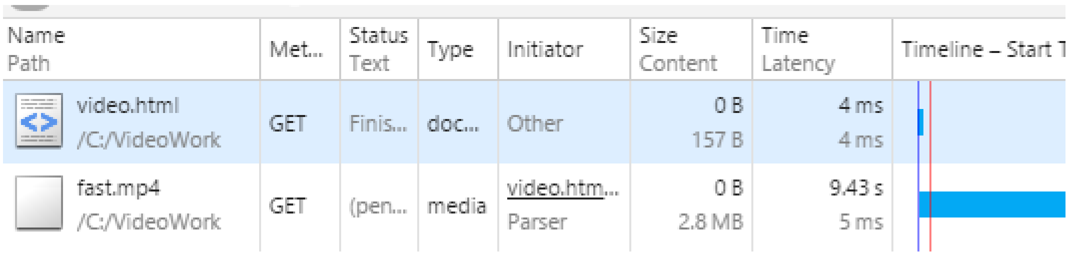
> [B]ecause this means that VP8 is a hell of lot safer and more free from possible legal repercussions than H.264 itself. What many H.264 proponents do not understand, either wilfully or out of sheer ignorance, is that those H.264 licenses embedded in Windows, OS X, iOS, your ‘professional’ camera, and so on, do not cover commercial use. If you shoot a video with your camera in H.264, upload it to YouTube, and get some income from advertisements, you’re in violation of the H.264 license (and the MPEG-LA made it clear they had no qualms about going after individual users). The extension the MPEG-LA announced (under pressure from VP8 and WebM) changed nothing about that serious legal limitation.
>
> — Google called the MPEG-LA’s bluff, and won

Why Our Civilization’s Video Art and Culture is Threatened by the MPEG-LA

The other codec worth looking at (mostly because it’s supported by Firefox) is OGG Theora from the Xiph.org Foundation. Like VP8 and WebM Theora is free and unencumbered by patents.

MP4 containers can be optimized for a kind of pseudo streaming by re-arranging the “atoms” of the movie (atoms, in this context, are the chunks of data that make up the movie). The video player is looking or the `moov` atom and will not play the movie until it finds it.

If your server is configured for HTTP Range Requests it will request smaller chunks until it find the atom it needs.

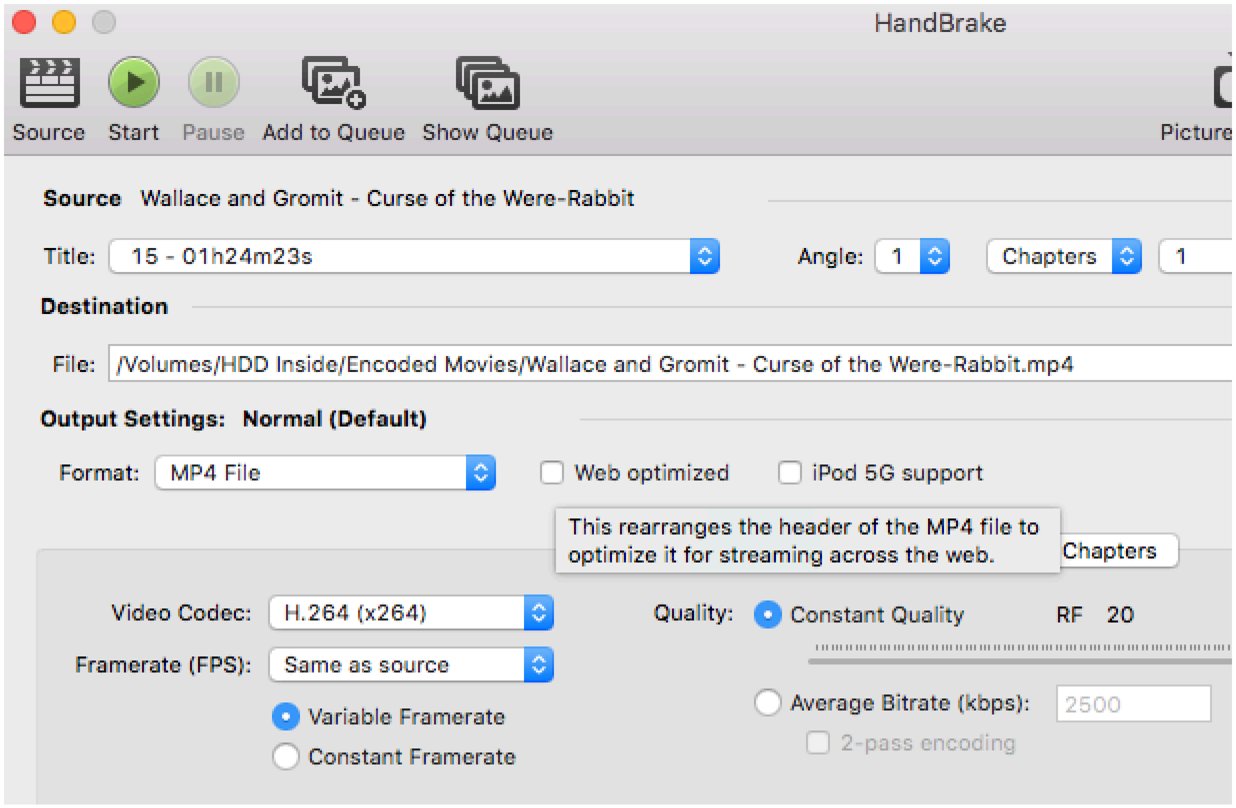


Name Path	Met...	Status Text	Type	Initiator	Size Content	Time Latency	Timeline – Start T
video.html /C:/VideoWork	GET	Finis...	doc...	Other	0 B 157 B	4 ms 4 ms	
fast.mp4 /C:/VideoWork	GET	(pen...	media	video.htm... Parser	0 B 2.8 MB	9.43 s 5 ms	

Figure 1: Different requests for the same video

Unfortunately for on-demand movies the moov atom is at the end of the file. So if the server is not configured to handle range requests then the player will have to download the complete file before it can start playing it.

If you're using already made content or don't want to re-encode the video you can use tools like Handbrake to optimize the video file for streaming across the web by moving the moov atom to the beginning of the file.



Source Wallace and Gromit - Curse of the Were-Rabbit

Title: 15 - 01h24m23s Angle: 1 Chapters 1

Destination

File: /Volumes/HDD Inside/Encoded Movies/Wallace and Gromit - Curse of the Were-Rabbit.mp4

Output Settings: Normal (Default)

Format: MP4 File ☐ Web optimized ☐ iPod 5G support

This rearranges the header of the MP4 file to optimize it for streaming across the web.

Chapters

Video Codec: H.264 (x264) Quality: ● Constant Quality RF 20

Framerate (FPS): Same as source ○ Average Bitrate (kbps): 2500

● Variable Framerate ☐ 2-pass encoding

○ Constant Framerate

Figure 2: Using Handbrake to re-arrange the video atoms

If you're working with multiple files or are more comfortable you can use ffmpeg to encode the file or add the appropriate flag to fast start playback. In the example below we add the `faststart` flag and use the same audio and video codecs as t he original file.

```
ffmpeg -i input.mp4 -movflags faststart \
-acodec copy \
-vcodec copy \
output.mp4
```

You can do something similar with WebM videos. The format is based around the Matroska container, either VP8 or VP9 video codecs and either Opus or Vorbis audio codecs. Matroska files, usually just called MKV files, use a kind of binary XML called EBML to store different things like video tracks, audio tracks, subtitles, and other data. These data chunks are called elements and they are similar in concept to the atoms in an MP4 file.

As with all video formats to start playing a WebM video, a browser has to know where the audio and video data is stored in elements. The element we're looking for is SeekHead. By default most video creation tools put a SeekHead element at the start of the video. The problem is that each video can have an unlimited number of SeakHead. In this case, the first SeekHead will container a pointer to a second SeekHead located at the end of the file.

Even if the first SeekHead contains pointers to the video and audio tracks, the browser still must go fetch the second SeekHead element, to see if there are additional video or audio tracks in the file, and determine which one has preference. Even if the second SeekHead is completely empty the browser must download and parse all SeekHead elements in the WebM file before it can play video content.

When playing a WebM video locally we don't need to worry about the file structure since we have all the content available for playback. When streaming a video over HTTP the order of elements does matter because the browser doesn't have the complete file yet. If the browser doesn't get certain elements at the beginning of the file it has to send range HTTP requests until it finds the data it needs. This can have impact on how quickly the file starts playing and overall page performance. The discussion below is all about rearranging the elements in the container.

Another aspect of WebM streaming performance is to optimize for seeking inside a video. This is another element, `Cues`. For the same reasons we are optimizing for fast start we want the `Cues` element downloaded as early as possible so that, if a user fast forwards the video, they will get a few HTTP downloads as possible.

To accomplish both goals, fast playback and fast seeking we'll use a single tool, mkclean, a tool specifically designed to address both the fast start and the fast seek problems. Using `original.webm` we run the following command to create the resulting `optimized.web` ready for the web

```
mkclean --doctype 4 \
--keep-cues \
--optimize \
original.webm optimized.webm
```

Revisting Video Encoding: DASH

Although DASH is designed for both on-demand and live streaming events, I'll concentrate on on-demand content.

Also important to note. Even though EME is part of DASH we will not work with EME extensions as I don't believe they should be part of the web platform.

Because our ecosystem for playing video on the web has changed considerably and now smaller devices (phone, tablets) access our content over unreliable networks subject the way we deliver video has changed. We need to account for these elements in how we deliver our video.

We'll work with two specs for streaming video for web delivery are DASH (also known as MPEG-DASH) and Apple's HSL.

Dynamic Adaptive Streaming over HTTP (DASH) is an adaptive bit-rate streaming technique delivered from conventional HTTP web servers.

MPEG-DASH breaks the content into a sequence of small HTTP-based file segments, each segment containing a short interval of playback time of content that is potentially many hours in duration, such as a movie or the live broadcast of a sports event.

Another important consideration is your audience. How many different streams will you provide for your audience? Are these videos encoded and packaged properly?

When creating DASH content you can choose what bit rates you make the content available for by providing videos encoded to those bitrates, the packager creates alternative segments encoded at the target bit rates covering the same, short, intervals of play back time.

While the content is being played back by a DASH client (in this case the web browser), the client automatically selects from the alternatives the next segment to download and play back based on current network conditions. The client selects the segment with the highest bit rate possible that can be downloaded in time for play back without causing stalls or re-buffering events in the playback. Thus, a browser playing back DASH content can seamlessly adapt to changing network conditions, and provide high quality play back with fewer stalls or re-buffering events.

DASH doesn’t resolve the HTML5 codec issue. DASH is codec agnostic which means that it can be implemented in either H.264 or WebM. This means that we’re back at square one in terms of what code we use and that will mean an increase in costs associated with storage and, potentially, bandwidth delivery.

MPEG-LA has a DASH Patent Pool and t his has definite impact in adoption among open source purists and adopters including Mozilla, according to Chris Blizzard (at Mozilla when the quote was made):

> Mozilla has always been committed to implementing widely adopted royalty-free standards. If the underlying MPEG standards were royalty free we would implement DASH. However, MPEG DASH is currently built on top of MPEG Transport Streams, which are not royalty free. Therefore, we are unlikely to implement at this time.
>
> — What is MPEG DASH? / November 22, 2011

Taking out Firefox market share (almost 12% of the browser market) doesn’t make much sense to deploy a technology that will make more work for us in the long run. On the other hand, we can look at what happened with the support of MP4 and the debacle still ongoing with what combination of container/video/audio codec to support the picture looks a little less bleak, but not by much :)

HLS (HTTP Live Streaming) is a technology developed by Apple as part of the OS X/ iOS media stack that works in a similar fashion to DASH but with different requirements, technologies and features.

The same concerns I raised about DASH apply to HLS. There is also the issue of this being a single vendor specification; there an IETF Informational Internet Draft there hasn’t been any action to ratify the draft as an IETF standard.

For the rest of this post we’ll concentrate on DASH as it’s the one that has the widest level of support and it means that I don’t have to create the packager or the player later on.

DASH Process

As I understand it the process to create content ready to play in a DASH-enabled web browser is as follows:

1. Encode the video to the target bit rates you want to use
2. Create the DASH manifest using the Shaka Packager
3. Upload the content to your server
4. Create the video tag using the Shaka Player or Dash.js

The process assumes that you've already encoded the videos to your target bit rate(s).

Packaging the video

Figure 3: Shaka Packager, what we'll use to create the DASH Manifest

Shaka Packager is a tool developed by Google to create DASH manifest for our content. It will also create separate streams for audio and video.

In the example we'll work with in these sections we'll generate a manifest three diferent versions of the same video. ***Shaka Packager will not encode the video… that's your job and it should be done before we start working in packaging and playing DASH content***.

```
path/to/packager \
input=media/SavingLight.mp4,stream=audio,output=audio.mp4 \
input=media/SavingLight.mp4,stream=video,output=video.mp4 \
input=media/SavingLight-baseline.mp4,stream=audio,output=audio-baseline.mp
input=media/SavingLight-baseline.mp4,stream=video,output=video-baseline.mp
input=media/SavingLight-high.mp4,stream=audio,output=audio-high.mp4 \
input=media/SavingLight-high.mp4,stream=video,output=video-high.mp4 \
--mpd_output example.mpd
```

If you’re comfortable compiling tools manually you can clone the Shaka Packager Github Repository and compile the tools following the instructions in the README file.

The DASH Manifest

One of the files produced by the Packager is the DASH manifest file. It is an XML file that describes the audio and video tracks that make up the video separately from one another.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  Generated with https://github.com/google/shaka-packager
  version 593f513c83-release-->
<MPD xmlns="urn:mpeg:dash:schema:mpd:2011"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xlink="http://www.w3.<MPD xmlns="urn:mpeg:dash:schema:mpd:2011"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xsi:schemaLocation="urn:mpeg:dash:schema:mpd:2011 DASH-MPD.xsd"
     xmlns:cenc="urn:mpeg:cenc:2013"
     profiles="urn:mpeg:dash:profile:isoff-on-demand:2011"
     minBufferTime="PT2S"
     type="static"
     mediaPresentationDuration="PT355.614S">4">
      <Representation id="0" bandwidth="1778582"
                      codecs="avc1.640028"
                      mimeType="video/mp4"
                      sar="1:1"
                      height="1080">
        <BaseURL>video.mp4</BaseURL>
        <SegmentBase indexRange="816-1231" timescale="90000">
          <Initialization range="0-815"/>
        </SegmentBase>
      </Representation>
      <Representation id="1" bandwidth="2261351"
                      codecs="avc1.640028"
                      mimeType="video/mp4"
                      sar="1:1"
                      height="800">
        <BaseURL>video-high.mp4</BaseURL>
        <SegmentBase indexRange="817-1208" timescale="90000">
          <Initialization range="0-816"/>
        </SegmentBase>
      </Representation>
      <Representation id="2" bandwidth="2606321"
                      codecs="avc1.42c028"
                      mimeType="video/mp4"
                      sar="1:1"
                      height="800">
        <BaseURL>video-baseline.mp4</BaseURL>
        <SegmentBase indexRange="815-1218" timescale="90000">
          <Initialization range="0-814"/>
        </SegmentBase>
      </Representation>
    </AdaptationSet>
    <AdaptationSet id="1" contentType="audio" subsegmentAlignment="true">
      <Representation id="3" bandwidth="127078"
                      codecs="mp4a.40.2"
                      mimeType="audio/mp4"
                      audioSamplingRate="44100">
        <AudioChannelConfiguration
        schemeIdUri="urn:mpeg:dash:23003:3:audio_channel_configuration:201
        value="2"/>
        <BaseURL>audio.mp4</BaseURL>
        <SegmentBase indexRange="745-1208" timescale="44100">
          <Initialization range="0-744"/>
        </SegmentBase>
      </Representation>
      <Representation id="4" bandwidth="156471"
                      codecs="mp4a.40.2"
                      mimeType="audio/mp4"
                      audioSamplingRate="44100">
        <AudioChannelConfiguration
        schemeIdUri="urn:mpeg:dash:23003:3:audio_channel_configuration:201
        value="2"/>
        <BaseURL>audio-high.mp4</BaseURL>
        <SegmentBase indexRange="745-1208" timescale="44100">
          <Initialization range="0-744"/>
        </SegmentBase>
      </Representation>
      <Representation id="5" bandwidth="156471"
                      codecs="mp4a.40.2"
                      mimeType="audio/mp4"
                      audioSamplingRate="44100">
        <AudioChannelConfiguration
        schemeIdUri="urn:mpeg:dash:23003:3:audio_channel_configuration:20
        value="2"/>
        <BaseURL>audio-baseline.mp4</BaseURL>
        <SegmentBase indexRange="745-1208" timescale="44100">
          <Initialization range="0-744"/>
        </SegmentBase>
      </Representation>
    </AdaptationSet>
  </Period>
</MPD>
```

Captions

Subtitles and captions are also part of the packaging process. The example belows takes the movie Sintel and splits it into audio and video streams and adds an english caption track.

```
packager \
  input=sintel.mp4,stream=audio,output=sintel_audio.mp4 \
  input=sintel.mp4,stream=video,output=sintel_video.mp4 \
  input=sintel_english_input.vtt,stream=text,output=sintel_english.vtt \
--mpd_output sintel_vod.mpd
```

Playing content: Shaka Player

Figure 4: Shaka Player, the player we'll use for this project

Shaka Player is the playback component of the Shaka ecosystem. Also developed by Google and open sourced on Github.

If you’re used to HTML5 the way you add DASH video is a little more complicated than you’re used to. The process is:

First we create a simple HTML page with a video element. In this page we make sure that we add thescripts we need:

- The shaka-player script
- The script for our application

The `video` element is incomplete on purpose. We will add the rest of the video in the script later on.

```
<!DOCTYPE html>
<html>
  <head><html><!-- Shaka Player compiled library: -->    <script src="pat
    <<script src="path/to/shaka-player.compiled.js"><!-- Your application
    <video id="vide<body></script>
  </head>
  <body>
    <video id="video"
           width="640"
           poster="media/SavingLight.jpg"
           controls autoplay></video>
  </body>
</html>
```

We’ll break the script into three parts:

- Application init

- Player init
- Error handler and event listener

We initialize the application by installing the polyfills built into the Shaka player to make sure that all the supported players behave the same way and that there won't be any unexpected surprises later on.

The next step is to check if the browser is supported using the built in `isBrowserSupported` check. If the browsers supports DASH then we initialize the player by calling `initPlayer()` otherwise we log the error to console.

```
var manifestUri = 'media/example.mpd';

function initApp() {
  // Install built-in polyfills to patch browser incompatibilities.
  shaka.polyfill.installAll();

  // Check to see if the browser supports the basic APIs Shaka needs.
  if (shaka.Player.isBrowserSupported()) {
    // Everything looks good!
    initPlayer();
  } else {
    // This browser does not have the minimum set of APIs we need.
    console.error('Browser not supported!');
  }
}
'Browser not supported!'
```

Initializing the player is the meat of the process and will take serval different steps.

We create variables to capture the video element using `getElementById` and the player by assigning a new instance of `Shaka.Player` and attach it to the video element.

We then attach the player to the window object to make it easier to access the console.

Next we attach the error event handler to the on`ErrorEvent` function defined later in the script. Positioning doesn't matter as far as Javascript is concerned.

The last step in this function is to try and load a manifest using a promise. If the promise succeeds then we log it to console otherwise the `catch` tree of the promise chain is executed and runs the `onError` function (which is different than `onErrorEvent` discussed earlier).

```
function initPlayer() {
  // Create a Player instance.
  var video = document.getElementById('video');
  var player = new shaka.Player(video);

  'video'// Attach player to the window to make it easy to access in the
  window.player = player;

  // Listen for error events.er.addEventListener('error', onErrorEvent);

  // Try 'error'// Try to load a manifest.
  // This is an asynchronous process.
  player.load(manifestUri).then(function() {
    // This runs if the asynchronous load is successful.og('The video has
  }).catch(onError);  // onError is 'The video has now been loaded!'// on
}
```

The last part of the script is to create the functions for errors (`onErrorEvent` and `onError`)

Finally we attach the `initApp()` function to the `DOMContentLoaded` event.

```
function onErrorEvent(event) {
  // Extract the shaka.util.Error object from the event.
  onError(event.detail);
}

function onError(error) {
  // Log the error.
  console.error('Error code', error.code, 'object', error);
}

document.addEventListener('DOMContentLoaded', initApp);
```

```
'Error code'
```

If everything works out OK we should have a video playing on screen.

There is a full example available to show how the player works. We've covered only the player's basic functionality; there's additional capabilities like casting to an Android Play device and playing your content on your TV… I'm more concerned with getting the video working.

Playing content: Dash.js

Figure 5: Dash.js is the reference implementation for MPEG-DASH

Dash.js is the reference DASH implementation, meaning this is the technology that they use to validate and demonstrate the different part of the specification and what they offer developers and implementers to use as the basis of their own player software.

The first way to use Dash.js is to manually initialize the player and attach it to a video element already in the page. It is possible to also create the video element programmatically and then assign it to the player.

The standard setup method uses javascript to initialize and provide video details to dash.js. MediaPlayerFactory provides an alternative declarative setup syntax.

Standard Setup

Using the same files that we used to create the Shaka demo we create the Dash.js video using code like the one below. In this page the script initializes the player and attaches it to the element with the id of video2 (`#video2`)

```
<!DOCTYPE html>
<html lang="en"<html lang="en">ta charset="UTF-8">
```

```
    <title>Title</title>
    <style><title>
      video {
        width: 640px;
        height: 360px;
      }
    >
</head>
<body>
  <div>
    <video id="video2" poster="media/SavingLight.jpg" controls></video>
  </div>

  <script src="http://cdn.dashjs.org/latest/dash.all.min.js"></script>
  <scr</head>  (function(){
      var url = "media/example.mpd";
      var player = dashjs.MediaPlayer().create();
      player.initialize(document.querySelector("#video2"), url, true);
    })();
  </script>

</body>
</html>
</script>
```

MediaPlayerFactory

An alternative way to build a Dash.js player in your web page is to use the `MediaPlayerFactory`. The MediaPlayerFactory will automatically instantiate and initialize the MediaPlayer module on appropriately tagged video elements.

Create a `video` element somewhere in your html and provide the path to your mpd file as src. Also ensure that your video element has the `data-dashjs-player` attribute on it. An example using the `MediaPlayerFactory` looks like this:

```
<!DOCTYPE html>
<html lang="en"<html lang="en">ta charset="UTF-8">
```

```
    <title>Title</title>
    <script src="http://cd<title>s.org/latest/dash.all.min.js"></script>
    <</script>style>
      video {
        width: 640px;
        height: 360px;
      }
    </style>
</head>
<body>
  <div>
    <video data-dashjs-player src="media/example.mpd" controls></video>
  </div>

</body>
</html>
```

Conclusion

Dash works but it requires a lot of work upfront to make the technology work as intended for the use cases use the technology for. The demos cover some of the most basic use cases for video on demand; we have not considered live streams or encrypted video.

As with many things on the web there is no ‘one size fits all’ solution. DASH works and it provides awesome capabilties but with those capabilities come additional cost for storage and delivery. In the example I used for this project used three streams each for audio and video and the weigh between 79 and 115MB for the video stream and betwee 5 and 7MB per audio stream. The more bitrates you add the more you have to consider storage costs.

Video is an awesome tool but one that requires a lot of prep work up front for it to be an effective tool.

Notes about the demo repository

All the code examples in this post are available in the dash-demo Github repository. To store the mp4 content, some of which is over 100mb in size, we’ve set the repo with GIT LFS to handle the large files; this will get around Github’s file size limitation.

A brief summary of the files:

- html5-video.html is a traditional HTML5 video tag using MP4, WebM and OGG video
- index.html uses the Shaka Player to play DASH video
- dashjs.html uses Dash.js’s MediaPlayerFactory method
- dashjs2.html uses Dash.js’s traditional method

Caching DASH Video

When we last discussed encoding DASH adaptive streaming we saw that Shaka Packager creates the master file and a series of segments. We left them in the browser's cache which may force downloads and will not work when offline or with poor connectivity.

Using a service worker we can create a cache for media segments so they can be stored locally for offline use. For this to work we need to do some modifications to our standard service worker to acommodate the special requirements of DASH chunked video.

The first step is to create a variable to hold the name of the cache. We then create a function to determine if we should cache the resource… for this particular case we want to cache files ending in `mp4` and `m4s` (MP4 files and MP4 segments)

```
var CACHE_NAME = 'segment-cache-v1';

function shouldCache(url) {
  return url.endsWith('.mp4') || url.endsWith('.m4s');
}
```

the `loadFromCacheOrFetch` function is the core of this service worker. It will open the specified cache and try to match the request.

If the request matches an item in the cache then it will return that item. The fragment will have an additional header that we add when we cache the item.

If the request doesn't match then we take a clone of the request and fetch it. This is where another peculiarity of working with DASH video comes in. A service worker cannot cache partial responses (HTTP 206) so we have to check if the response is ok, **did not** return a status code 206 and is one of the items we should cache. Only if the three conditions are met we fetch the resource and store it in the cache using the `cacheResponse` function.

```
function loadFromCacheOrFetch(request) {
  return caches.open(CACHE_NAME)
  .then(cache => {
    return cache.match(request)
    .then(response => {
      if (response) {
        console.log('Handling cached request', request.url);
        return response;
      }

      return fetch(request.clone())
      .then(response => {
        if (
          response.ok &&
          response.status != 206 &&
          shouldCache(request.url)) {
          console.log('Caching MP4 segment', request.url);
          cacheResponse(cache, request, response);
        }

        return response;
      });
    });
  })
}
```

The `cacheResponse` function modifiess the response object before putting it in the cache. Because the response is read onl we have to recreate it if we'll make any changes.

We create an array of the data we want to pass to the response object when we actually cache it, that's why the response has uses two parameters, an `arrayBuffer` and the array that we creted with response data and header.

Response objects are single use. This means we need to call clone() so we can both store the ArrayBuffer and give the response to the page.

```
function cacheResponse(cache, request, response) {
  var init = {
    status: response.status,
    statusText: response.statusText,
    headers: {'X-Shaka-From-Cache': true}
  };

  response.headers.forEach((value, key) => {
    init.headers[key] = value;
  });

  return response.clone().arrayBuffer()
  .then(ab => {
    cache.put(request, new Response(ab, init));
  });
}
```

The final piece is the `fetch` event itself which responds using the `loadFromCacheOrFetch` function.

```
self.addEventListener('fetch', (event) => {
  event.respondWith(loadFromCacheOrFetch(event.request));
});
```

In a larger service worker we could use a different

```
self.addEventListener('fetch', (event) => {
  if (event.request.url.endsWith('.mp4') || event.request.url.endsWith('.
    event.respondWith(loadFromCacheOrFetch(event.request));
  }
});
```

Chunking audio and video with the Media Source API

- Media Source API
  - https://developers.google.com/web/updates/2011/11/Stream-video-using-the-MediaSource-API
  - https://developer.mozilla.org/en-US/docs/Web/API/Media_Source_Extensions_API

Media Session API

Currently only works in Chrome for Android. Want to wait until there’s broader support before writing about it

- Media Session API
    - https://googlechrome.github.io/samples/media-session/video.html
    - https://googlechrome.github.io/samples/media-session/audio.html