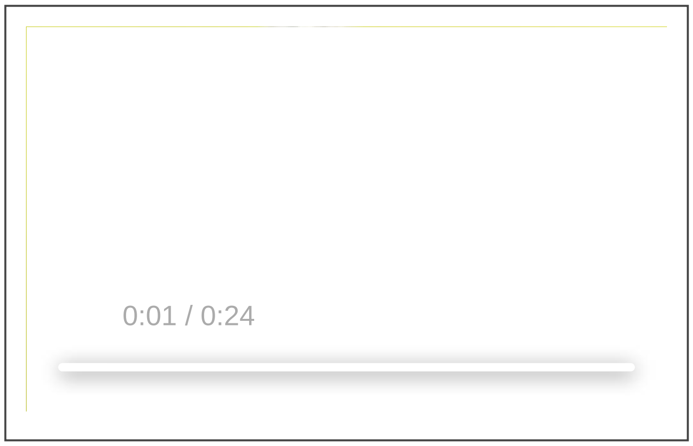
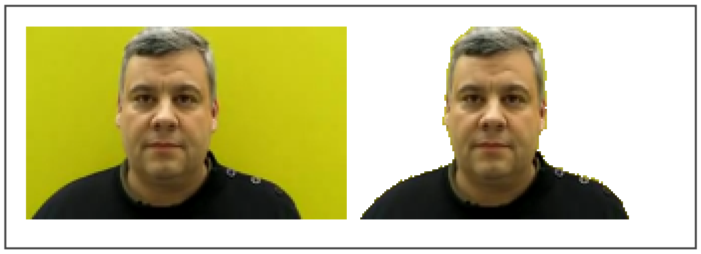
MDN Web Docs
moz://a

Manipulating video using canvas

By combining the capabilities of the `video` element with a `canvas`, you can manipulate video data in real time to incorporate a variety of visual effects to the video being displayed. This tutorial demonstrates how to perform chroma-keying (also known as the "green screen effect") using JavaScript code.





The document content

The HTML document used to render this content is shown below.

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      body {
        background: black;
        color: #CCCCCC;
      }
      #c2 {
        background-image: url(media/foo.png);
        background-repeat: no-repeat;
      }
      div {
        float: left;
        border : 1px solid #444444;
        padding: 10px;
        margin: 10px;
        background: #3B3B3B;
      }
    </style>
  </head>

  <body>
    <div>
      <video id="video" src="media/video.mp4" controls="true" crossorigi
    </div>
    <div>
      <canvas id="c1" width="160" height="96"></canvas>
      <canvas id="c2" width="160" height="96"></canvas>
    </div>
  <script type="text/javascript" src="processor.js"></script>
  </body>
</html>
```

The key bits to take away from this are:

1. This document establishes two canvas elements, with the IDs `c1` and `c2`. Canvas `c1` is used to display the current frame of the original video, while `c2` is used to display the video after performing the chroma-keying effect; `c2` is preloaded with the still image that will be used to replace the green background in the video.
2. The JavaScript code is imported from a script named `processor.js`.

The JavaScript code

The JavaScript code in `processor.js` consists of three methods.

Initializing the chroma-key player

The `doLoad()` method is called when the HTML document initially loads. This method's job is to prepare the variables needed by the chroma-key processing code, and to set up an event listener so we can detect when the user starts playing the video.

```
const processor = {};

processor.doLoad = function doLoad() {
  const video = document.getElementById('video');
  this.video = video;

  this.c1 = document.getElementById('c1');
  this.ctx1 = this.c1.getContext('2d');

  this.c2 = document.getElementById('c2');
  this.ctx2 = this.c2.getContext('2d');

  video.addEventListener('play', () => {
      this.width = video.videoWidth / 2;
      this.height = video.videoHeight / 2;
      this.timerCallback();
    }, false);
};
```

This code grabs references to the elements in the HTML document that are of particular interest, namely the `video` element and the two `canvas` elements. It also fetches references to the graphics contexts for each of the two canvases. These will be used when we're actually doing the chroma-keying effect.

Then `addEventListener()` is called to begin watching the `video` element so that we obtain notification when the user presses the play button on the video. In response to the user beginning playback, this code fetches the width and height of the video, halving each (we will be halving the size of the video when we perform the chroma-keying effect), then calls the `timerCallback()` method to start watching the video and computing the visual effect.

The timer callback

The timer callback is called initially when the video starts playing (when the "play" event occurs), then takes responsibility for establishing itself to be called periodically in order to launch the keying effect for each frame

```
processor.timerCallback = function timerCallback() {
  if (this.video.paused || this.video.ended) {
    return;

  }
  this.computeFrame();
  setTimeout(() => {
      this.timerCallback();
    }, 0);
};
```

The first thing the callback does is check to see if the video is even playing; if it's not, the callback returns immediately without doing anything.

Then it calls the `computeFrame()` method, which performs the chroma-keying effect on the current video frame.

The last thing the callback does is call `setTimeout()` to schedule itself to be called again as soon as possible. In the real world, you would probably schedule this to be done based on knowledge of the video's frame rate.

Manipulating the video frame data

The `computeFrame()` method, shown below, is responsible for actually fetching a frame of data and performing the chroma-keying effect.

```
processor.computeFrame = function computeFrame() {
  this.ctx1.drawImage(this.video, 0, 0, this.width, this.height);
  const frame = this.ctx1.getImageData(0, 0, this.width, this.height);
  const length = frame.data.length;

  for (let i = 0; i < length; i += 4) {
    const red = data[i + 0];
    const green = data[i + 1];
    const blue = data[i + 2];
    if (green > 100 && red > 100 && blue < 43) {
      data[i + 3] = 0;
    }
  }
```

```
    this.ctx2.putImageData(frame, 0, 0);
  };
```

When this routine is called, the video element is displaying the most recent frame of video data, which looks like this:

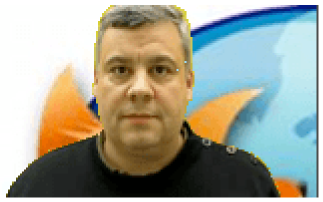
In line 2, that frame of video is copied into the graphics context `ctx1` of the first canvas, specifying as the height and width the values we previously saved to draw the frame at half size. Note that you can pass the video element into the context's `drawImage()` method to draw the current video frame into the context. The result is:

Line 3 fetches a copy of the raw graphics data for the current frame of video by calling the `getImageData()` method on the first context. This provides raw 32-bit pixel image data we can then manipulate. Line 4 computes the number of pixels in the image by dividing the total size of the frame's image data by four.

The `for` loop that begins on line 6 scans through the frame's pixels, pulling out the red, green, and blue values for each pixel, and compares the values against predetermined numbers that are used to detect the green screen that will be replaced with the still background image imported from `foo.png`.

Every pixel in the frame's image data that is found that is within the parameters that are considered to be part of the green screen has its alpha value replaced with a zero, indicating that the pixel is entirely transparent. As a result, the final image has the entire green screen area 100% transparent, so that when it's drawn into the destination context in line 13, the result is an overlay onto the static backdrop.

The resulting image looks like this:



This is done repeatedly as the video plays, so that frame after frame is processed and displayed with the chroma-key effect.

View the full source for this example .

See also

- Web media technologies
- Guide to media types and formats on the web
- Learning area: Video and audio content

Last modified: Apr 7, 2021, by MDN contributors