Video Rendering with Node.js and FFmpeg

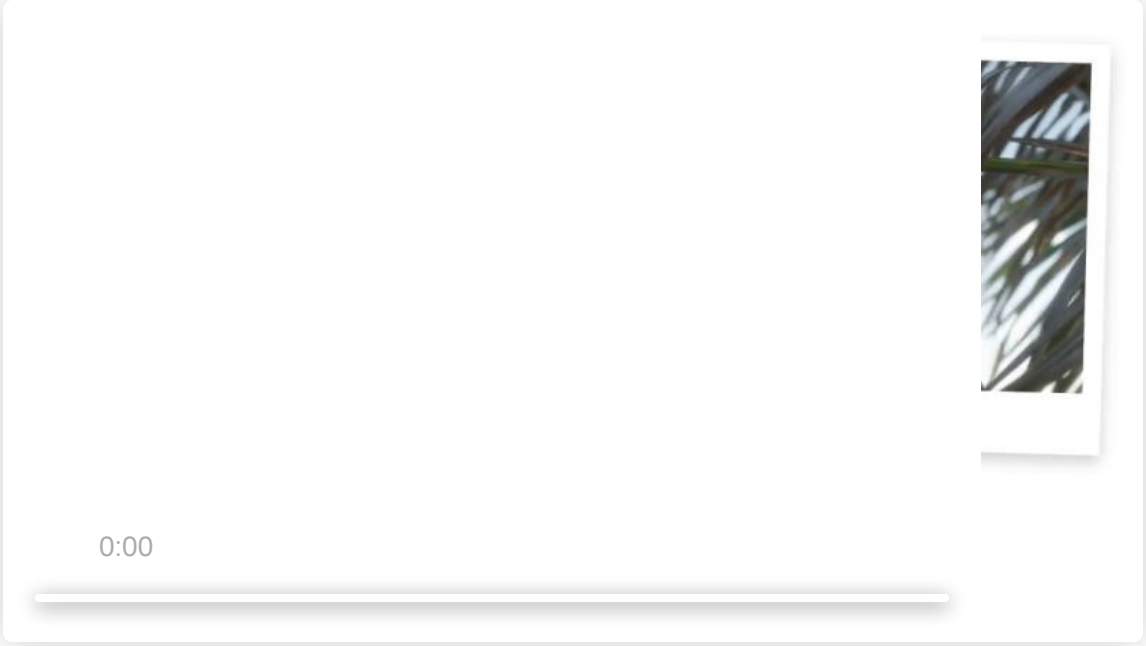
30 September 2022 | 22 min read

Casper Kloppenburg

Introduction

There is no better way to convey your message than by using video, since it is more engaging and fun than any other medium. This is especially true for social media, which has become increasingly video-centric. There is also a growing trend of using video in email campaigns. Wouldn't it be great if we could automate the creation of these videos programmatically, using something like a video template and dynamic data? As a matter of fact, this is possible. The best part is that all of this can be done with open source software and without using C++ or any other low-level language.

In this tutorial, you'll learn how to accomplish this using pure JavaScript and Node.js, along with the help of FFmpeg for the encoding. Let's say we're developing an application for a travel agency, which is looking to generate custom videos tailored to various travel destinations. They want to create these videos in thousands, so we're going to build an application for them to render these custom videos in bulk.



The dynamic video we are going to create using pure JavaScript and FFmpeg.

Step 1 – Setting up the project

Let's start by opening up our terminal and creating our project's directory:

```
$ mkdir video-rendering
```

Navigate to the new directory:

```
$ cd video-rendering
```

Use `npm init` to create a new Node.js project. This will create a package.json file.

```
$ npm init -y
```

Our new project requires a few dependencies. We'll use the `canvas` package to draw our graphics, which is an implementation of the HTML5 Canvas element for Node.js. We also need the `ffmpeg-static` package to get the latest distribution of FFmpeg, and `fluent-ffmpeg` to make it easy to use FFmpeg from our Node.js app.

```
$ npm i canvas ffmpeg-static fluent-ffmpeg
```

The following line needs to be added to our package.json file since we'll be using ES6 features:

```
{
  "name": "video-rendering",
  "type": "module",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo "Error: no test specified" && exit 1"
  },
  "keywords": [],
  "author": "Casper",
  "license": "ISC",
  "dependencies": {
    "canvas": "^2.10.1",
    "ffmpeg-static": "^5.1.0",
    "fluent-ffmpeg": "^2.1.2"
  }
}
```

Let's create a new folder called `src` for our source files, and a folder `assets` for our media files:

```
$ mkdir src
$ mkdir assets
```

We're going to use an audio clip, some videos, an image, and a couple fonts as input for our dynamic video. You can find these files in the **Github repository** with the final code. Make sure to download these files and put them in the `assets` directory that we just created. Our project directory should now look like this:

```
├── assets
│   ├── catch-up-loop-119712.mp3
│   ├── caveat-medium.ttf
│   ├── chivo-regular.ttf
│   ├── logo.svg
│   ├── pexels-2829177.mp4
│   ├── pexels-3576378.mp4
│   └── pexels-4782135.mp4
├── node_modules
│   └── ...
├── src
│   └── index.js
├── package.json
└── package-lock.json
```

Step 2 – Rendering a simple video

In this step, we'll create a simple video with only a single element, just to go over the basics of video rendering. Following these basic concepts, we will introduce easing, keyframes, drawing contexts, and transformations. Let's start from the beginning by drawing a single frame first.

Drawing a single frame

Open `index.js` in your editor of choice. To begin with, we'll create a new `Canvas` instance. Imagine it as a blank sheet of paper 1280 by 720 pixels wide, on which we can draw pictures, shapes, and texts. The context object gives us the interface for drawing on the canvas. Next, we load `assets/logo.svg` and draw it at position x=100 and y=100 with dimensions 500 by 500. The last step is to make an image file from the canvas and save it to disk:

```
import fs from 'fs';
import { Canvas, loadImage } from 'canvas';

// Create a new canvas of 1280 by 720
const canvas = new Canvas(1280, 720);
const context = canvas.getContext('2d');

// Load the image from file
const logo = await loadImage('assets/logo.svg');

// Draw the image to the canvas at x=100 and y=100 with a size of 500x500
context.drawImage(logo, 100, 100, 500, 500);

// Write the image to disk as a PNG
const output = canvas.toBuffer('image/png');
await fs.promises.writeFile('image.png', output);
```

Now let's execute our code. Run the following from your terminal. The file `image.png` should be created in your project directory.

```
$ node src/index.js
```

The image that we just drew. It's not very exciting, but hang on.

It's time to step it up a notch. Rather than making a still image, let's make a video with motion.

Rendering a video with motion

When we think about it, a video is nothing more than a series of images displayed in quick succession. These images are also known as frames. In video, we talk about frames per second as a measure of how many images are displayed every second. The lower the FPS, the more jerky the video will be. A minimum of 25 fps is recommended for video, but at **Creatomate**, we render our videos at 60 fps for smoother animations.

We're going to make a 3-second video with a frame rate of 60, where an image moves from left to right. This requires creating `3 * 60 = 180` slightly different frames, which we then stitch together into a video using FFmpeg. And while we're at it, we also tell it to add a soundtrack to the video.

Let's make an utility function in `utils/stitchFramesToVideo.js` to do this. This is also where `fluent-ffmpeg` comes in. Because FFmpeg is a separate program, it can be awkward to use with Node.js. Fortunately, fluent-ffmpeg takes care of that by providing an interface that we can use from JavaScript. Here's the function:

```
import ffmpeg from 'fluent-ffmpeg';

export async function stitchFramesToVideo(
  framesFilepath,
  soundtrackFilePath,
  outputFilepath,
  duration,
  frameRate,
```

```
) {

  await new Promise((resolve, reject) => {
    ffmpeg()

      // Tell FFmpeg to stitch all images together in the provided directory
      .input(framesFilepath)
      .inputOptions([
        // Set input frame rate
        `-framerate ${frameRate}`,
      ])

      // Add the soundtrack
      .input(soundtrackFilePath)
      .audioFilters([
        // Fade out the volume 2 seconds before the end
        `afade=out:st=${duration - 2}:d=2`,
      ])

      .videoCodec('libx264')
      .outputOptions([
        // YUV color space with 4:2:0 chroma subsampling for maximum compatibility with
        // video players
        '-pix_fmt yuv420p',
      ])

      // Set the output duration. It is required because FFmpeg would otherwise
      // automatically set the duration to the longest input, and the soundtrack might
      // be longer than the desired video length
      .duration(duration)
      // Set output frame rate
      .fps(frameRate)

      // Resolve or reject (throw an error) the Promise once FFmpeg completes
      .saveToFile(outputFilepath)
      .on('end', () => resolve())
      .on('error', (error) => reject(new Error(error)));
  });
}
```

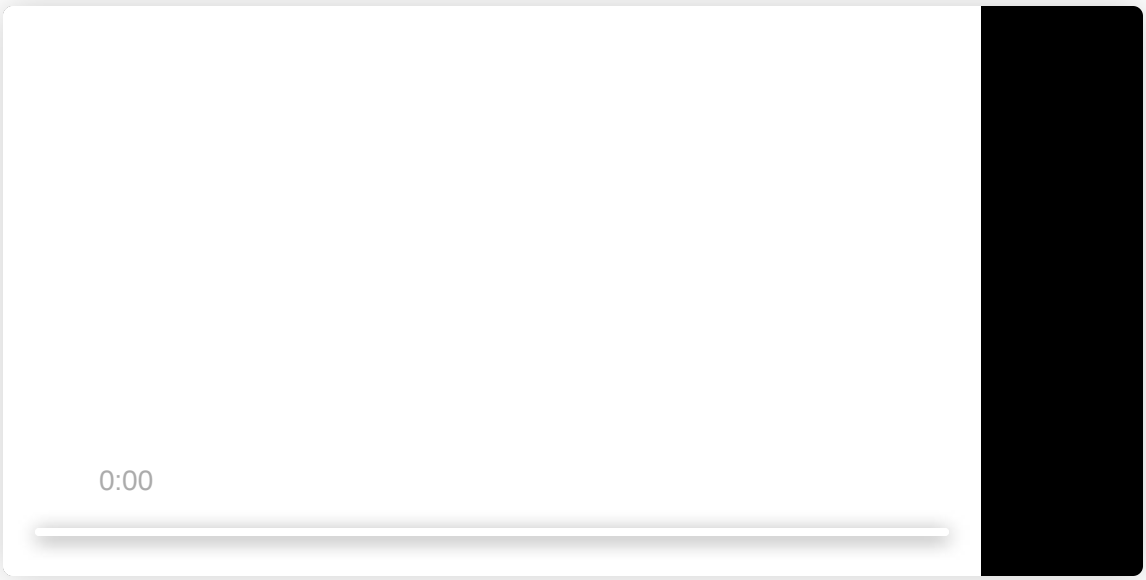
Now let's return to index.js and write the code to generate each frame. In the following code, we're creating 180 images that we store in a temporary directory at `tmp/output`. We then run `stitchFramesToVideo`, which calls FFmpeg to convert the frames to a video.

```
import fs from 'fs';
import ffmpegStatic from 'ffmpeg-static';
import ffmpeg from 'fluent-ffmpeg';
import { Canvas, loadImage, registerFont } from 'canvas';
import { stitchFramesToVideo } from './utils/stitchFramesToVideo.js';
```

```
// Tell fluent-ffmpeg where it can find FFmpeg
ffmpeg.setFfmpegPath(ffmpegStatic);

// Clean up the temporary directories first
for (const path of ['out', 'tmp/output']) {
    if (fs.existsSync(path)) {
        await fs.promises.rm(path, { recursive: true });
    }
    await fs.promises.mkdir(path, { recursive: true });
}

const canvas = new Canvas(1280, 720);
const context = canvas.getContext('2d');

const logo = await loadImage('assets/logo.svg');

// The video length and frame rate, as well as the number of frames required
// to create the video
const duration = 3;
const frameRate = 60;
const frameCount = Math.floor(duration * frameRate);

// Render each frame
for (let i = 0; i < frameCount; i++) {

    const time = i / frameRate;

    console.log(`Rendering frame ${i} at ${Math.round(time * 10) / 10} seconds...`);

    // Clear the canvas with a white background color. This is required as we are
    // reusing the canvas with every frame
    context.fillStyle = '#ffffff';
    context.fillRect(0, 0, canvas.width, canvas.height);

    renderFrame(context, duration, time);

    // Store the image in the directory where it can be found by FFmpeg
    const output = canvas.toBuffer('image/png');
    const paddedNumber = String(i).padStart(4, '0');
    await fs.promises.writeFile(`tmp/output/frame-${paddedNumber}.png`, output);
}

// Stitch all frames together with FFmpeg
await stitchFramesToVideo(
  'tmp/output/frame-%04d.png',
  'assets/catch-up-loop-119712.mp3',
  'out/video.mp4',
  duration,
  frameRate,
);
```

```
function renderFrame(context, duration, time) {

    // Calculate the progress of the animation from 0 to 1
    let t = time / duration;

    // Draw the image from left to right over a distance of 550 pixels
    context.drawImage(logo, 100 + (t * 550), 100, 500, 500);
}
```

Let's run our code again. The file `out/video.mp4` should be created in your project directory. As we can see in the video, the logo appears to be moving, but it feels a little flat. We'll see what we can do about that in the next step.



Still not very exciting, but we're making progress.

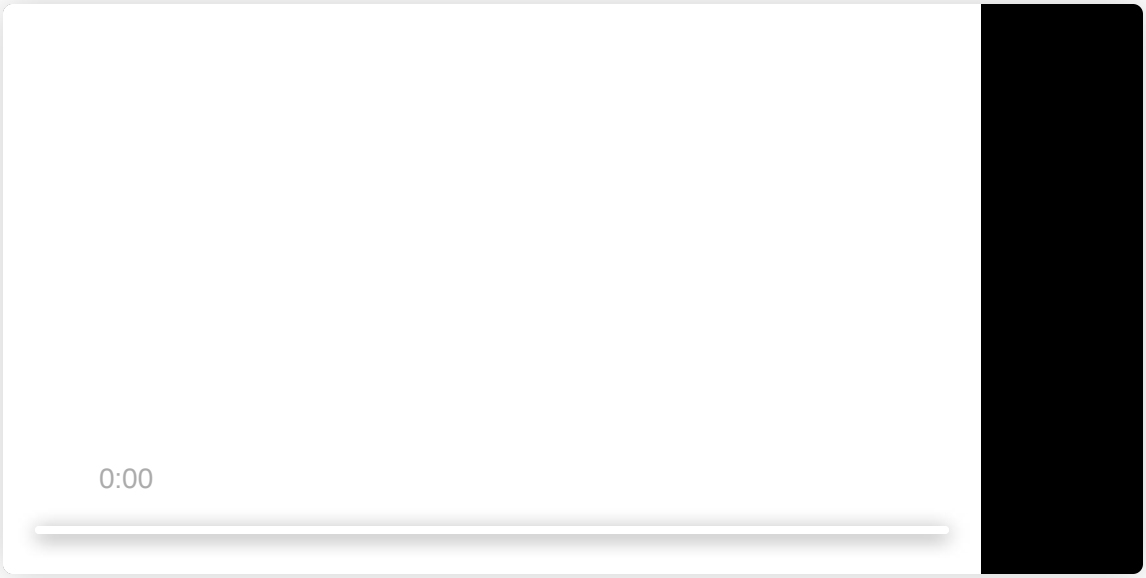
Step 3 – Using keyframes and easing

While we were able to make the object appear to move from left to right, the animation appears bland. Why is that? When it comes to motion in the physical world, we can say that nothing moves linearly. But that is exactly what we did in our animation – we simply interpolated the x position over time in a linear way. Let's get that fixed.

Applying simple easing

To make the motion feel more natural, we have to change the velocity of the motion over time, so that the object starts slowly, builds up speed gradually, then slows down again as it gets close to the destination. Easing functions help us do this. Let's apply an easing to our animation:

```
// ...

function renderFrame(context, duration, time) {

    // Calculate the progress of the animation from 0 to 1
    let t = time / duration;

    // Apply Cubic easing, see https://easings.net/#easeInOutCubic
    t = applyCubicInOutEasing(t);

    // Draw the image from left to right over a distance of 550 pixels
    context.drawImage(logo, 100 + t * 550, 100, 500, 500);
}

function applyCubicInOutEasing(t) {
    return t < 0.5 ? 4 * t * t * t : 1 - Math.pow(-2 * t + 2, 3) / 2;
}
```



The animation after applying easing.

There are a bunch of easing functions we can use to make our animations look more interesting. Look at **easing.net** for a list of the most popular ones with JavaScript examples.

Keyframes with easing

Up until now, we got away with a simple calculation to calculate the position of the object. But what if we want our object to move to multiple locations? Here's where keyframes come in handy. You can think of keyframes as values at specific points in time. It's easier to make complicated animations

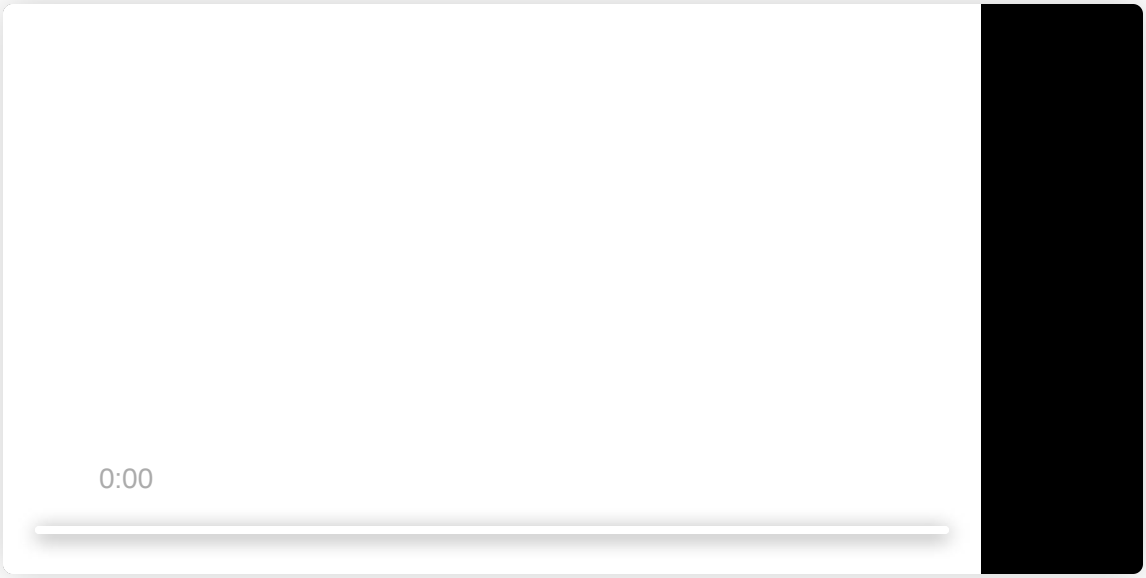
with keyframes, so we're introducing a new utility function that helps us with that. First, let's take a look at how it is used before moving on to its implementation:

```
// ...

function renderFrame(context, duration, time) {

    // Calculate the x position over time
    const x = interpolateKeyframes([
        // At time 0, we want x to be 100
        { time: 0, value: 100},
        // At time 1.5, we want x to be 550 (using Cubic easing)
        { time: 1.5, value: 550, easing: 'cubic-in-out' },
        // At time 3, we want x to be 200 (using Cubic easing)
        { time: 3, value: 200, easing: 'cubic-in-out' },
    ], time);


    // Draw the image
    context.drawImage(logo, x, 100, 500, 500);
}
```

And here is the implementation of `src/utils/interpolateKeyframes.js`. Don't forget to import it into `src/index.js`, then run our code again to see how the video turns out.

```
export function interpolateKeyframes(keyframes, time) {

  if (keyframes.length < 2) {
    throw new Error('At least two keyframes should be provided');
  }

  // Take the value of the first keyframe if the provided time is before it
  const firstKeyframe = keyframes[0];
  if (time < firstKeyframe.time) {
    return firstKeyframe.value;
  }

  // Take the value of the last keyframe if the provided time is after it
  const lastKeyframe = keyframes[keyframes.length - 1];
  if (time >= lastKeyframe.time) {
    return lastKeyframe.value;
  }

  // Find the keyframes before and after the provided time, like this:
  //
  //                   Time
  // ——  [Keyframe] ——┴——— [Keyframe] ——— [...]
  //
```

```
  let index;
  for (index = 0; index < keyframes.length - 1; index++) {
    if (keyframes[index].time <= time && keyframes[index + 1].time >= time) {
      break;
    }
  }

  const keyframe1 = keyframes[index];
  const keyframe2 = keyframes[index + 1];

  // Find out where the provided time falls between the two keyframes from 0 to 1
  let t = (time - keyframe1.time) / (keyframe2.time - keyframe1.time);

  // Apply easing
  if (keyframe2.easing === 'expo-out') {
    t = applyExponentialOutEasing(t);
  } else if (keyframe2.easing === 'cubic-in-out') {
    t = applyCubicInOutEasing(t);
  } else {
    // ... Implement more easing functions
  }

  // Return the interpolated value
  return keyframe1.value + (keyframe2.value - keyframe1.value) * t;
}

// Exponential out easing
function applyExponentialOutEasing(t) {
  return t === 1 ? 1 : 1 - Math.pow(2, -10 * t);
}

// Cubic in-out easing
function applyCubicInOutEasing(t) {
  return t < 0.5 ? 4 * t * t * t : 1 - Math.pow(-2 * t + 2, 3) / 2;
}
```



The video with keyframes applied.

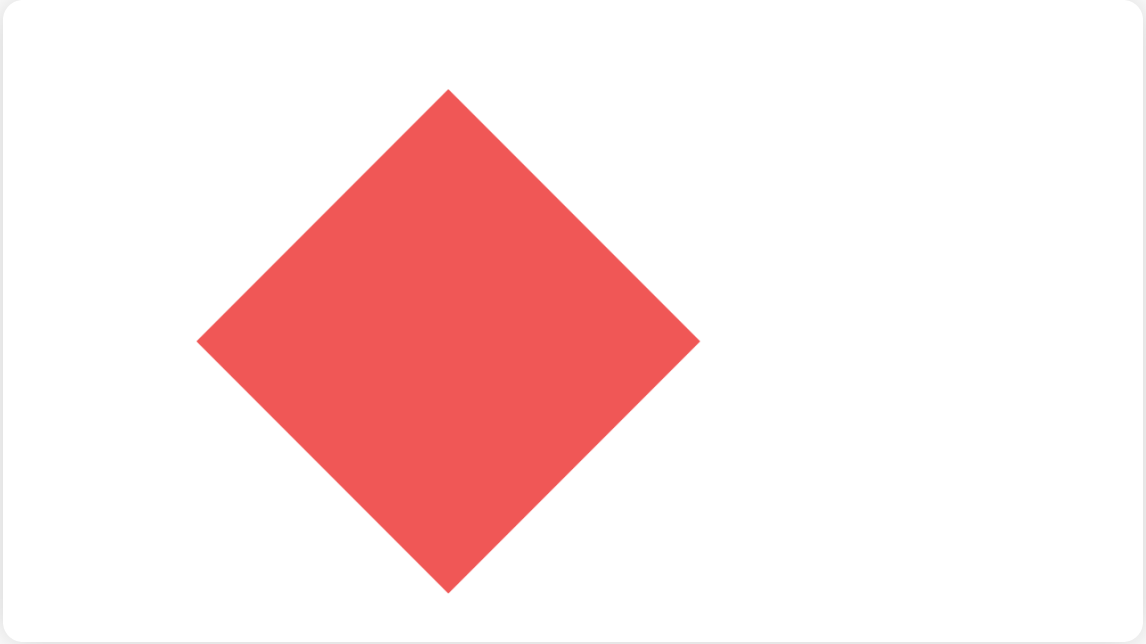
Step 4 – Context and transformation

Now with that in place, we are almost ready to put the video together. But first, let's talk about contexts and transformations, two important concepts that we'll be using a lot in the final video.

The drawing context

Whenever we draw something on the canvas, we are giving the rasterizer information about how we want our graphics to look via the drawing context. This can best be explained with an example. Say we want to draw a red rectangle that is rotated 45 degrees. To draw the shape we want, we can use the context to transform and style it:

```
import fs from 'fs';
import { Canvas } from 'canvas';

// Create a new canvas of 1280 by 720
const canvas = new Canvas(1280, 720);
const context = canvas.getContext('2d');

// Move the drawing context to x=500 and y=100
context.translate(500, 100);

// Rotate by 45 degrees
// The function expects an angle in radians, so we have to convert degrees to radians firs
context.rotate(45 * Math.PI / 180);

// Set the fill style
context.fillStyle = '#f05756';

// Draw a filled rectangle
context.fillRect(0, 0, 400, 400);

// Write the image to disk as a PNG
const output = canvas.toBuffer('image/png');
await fs.promises.writeFile('image.png', output);
```



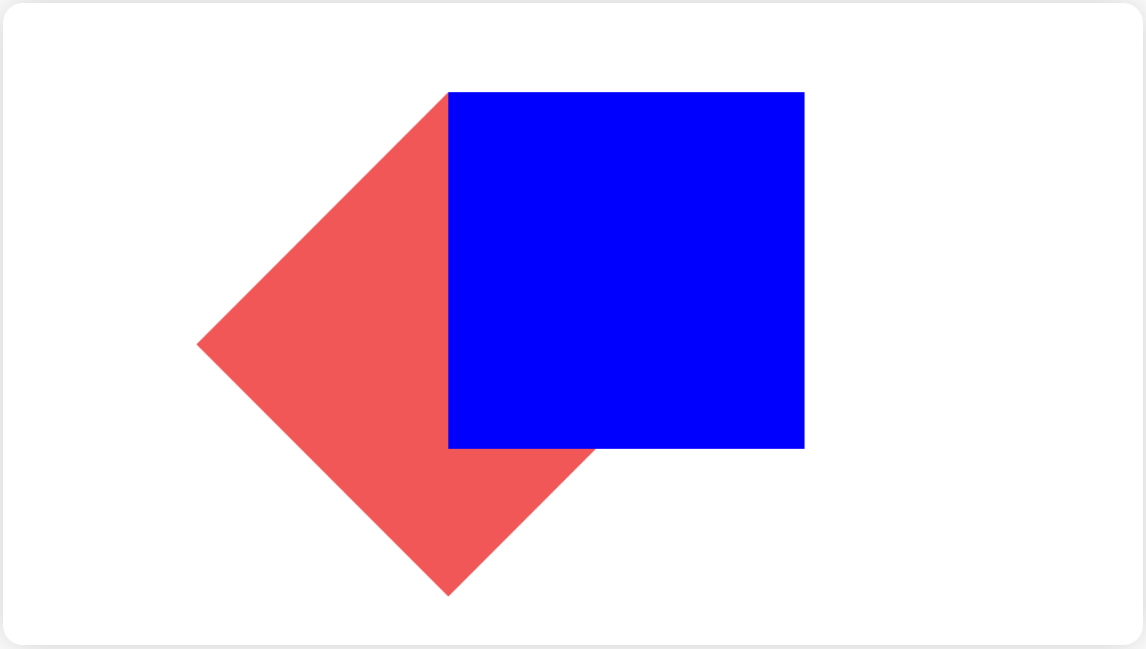
If we run the above code, this is what we'll see.

Save() and restore()

We can switch between context states with `canvas.save()` and `canvas.restore()`. Imagine it as a stack. Every time we call `save()`, we add a copy of the current context to the stack. Using `restore()`, we can get back to the previous state, which removes the topmost context from the stack and makes it the current one. So for every `canvas.save()`, there should be a `canvas.restore()`. Though it might seem complicated, this is actually a very convenient way to draw complex graphics, as you'll see in the final code. Let's see it in action with the following example:

```
import fs from 'fs';
import { Canvas } from 'canvas';

// Create a new canvas of 1280 by 720
const canvas = new Canvas(1280, 720);
const context = canvas.getContext('2d');

// Set the fill style to blue
context.fillStyle = 'blue';

// Move the drawing context to x=500 and y=100
context.translate(500, 100);

// Save the current context
context.save();

// Rotate by 45 degrees
// The function expects an angle in radians, so we have to convert degrees to radians firs
context.rotate(45 * Math.PI / 180);
```

```
// Set the fill style to red
context.fillStyle = '#f05756';

// Draw a filled rectangle. This one should be red
context.fillRect(0, 0, 400, 400);

// Restore the context that was active before we called context.save()
context.restore();

// Draw another filled rectangle
// As we restored the context, this one should be blue and not rotated,
// but still drawn at x=500 and y=100
context.fillRect(0, 0, 400, 400);

// Write the image to disk as a PNG
const output = canvas.toBuffer('image/png');
await fs.promises.writeFile('image.png', output);
```



Two rectangles drawn from different drawing contexts.

This MDN page explains the canvas context in more detail. Although that page talks about the HTML Canvas, almost every property has been ported to Node.js by the `canvas` package.

Step 5 – Extracting frames from our input videos

Saving video frames with FFmpeg

Are you still with me? Good! We're getting closer to our goal. As you can see in the introduction, we used a few videos as inputs for our final video. In order to use these videos in our final video, we

must first extract their frames. Once again, FFmpeg comes to the rescue. Let's make a utility function at `src/utils/extractFramesFromVideo.js` that instructs FFmpeg to do this.

```
import ffmpeg from 'fluent-ffmpeg';

// Example usage: await extractFramesFromVideo('video.mp4', 'frame-%04d.png', 60);
export async function extractFramesFromVideo(inputFilepath, outputFilepath, frameRate) {

  await new Promise((resolve, reject) => {
    ffmpeg()

      // Specify the filepath to the video
      .input(inputFilepath)

      // Instruct FFmpeg to extract frames at this rate regardless of the video's frame ra
      .fps(frameRate)

      // Save frames to this directory
      .saveToFile(outputFilepath)

      .on('end', () => resolve())
      .on('error', (error) => reject(new Error(error)));
  });
}
```

Reading the frames back in

After the frames have been extracted by FFmpeg, we can retrieve them using the `loadImage` method we used before. For our convenience, let's add another utility function at `src/utils/getVideoFrameReader.js` :

```
import fs from 'fs';
import path from 'path';
import { loadImage } from 'canvas';
import { extractFramesFromVideo } from './extractFramesFromVideo.js';

/* Example usage:
  const getNextFrame = await getVideoFrameReader('video.mp4', 'tmp', 60);
  await getNextFrame();    // Returns frame 1
  await getNextFrame();    // Returns frame 2
  await getNextFrame();    // Returns frame 3
*/
export async function getVideoFrameReader(videoFilepath, tmpDir, frameRate) {

  // Extract frames using FFmpeg
  await extractFramesFromVideo(videoFilepath, path.join(tmpDir, 'frame-%04d.png'), frameRa
```

```
  // Get the filepaths to the frames and sort them alphabetically
  // so we can read them back in the right order
  const filepaths = (await fs.promises.readdir(tmpDir))
    .map(file => path.join(tmpDir, file))
    .sort();

  let frameNumber = 0;

  // Return a function that returns the next frame every time it is called
  return async () => {

    // Load a frame image
    const frame = await loadImage(filepaths[frameNumber]);

    // Next time, load the next frame
    if (frameNumber < filepaths.length - 1) {
      frameNumber++;
    }

    return frame;
  };
}
```

Step 6 – Putting it all together

We now have everything we need to make our video rendering script. For the sake of brevity, we'll focus on the most interesting source files, `index.js` and `renderMainComposition.js` as the rest is just rehashing what we've already covered. However, the full project **is available on Github** and you are welcome to check it out and tinker with it.

Let's start with `src/index.js`. We begin by cleaning up any temporary files left over from a previous run. We then extract all the frames from our input videos. Next, we load the logo image and fonts. Once that's done, it's time to start rendering. Whenever we render a frame for our video, we're using the methods returned from `getVideoFrameReader()` to load a frame from our input videos into `image1`, `image2`, and `image3`. Then we let `renderMainComposition()` do the actual rendering, which we'll talk about next. Finally, we stitch the rendered frames of our video into an MP4 using FFmpeg:

```
import fs from 'fs';
import ffmpegStatic from 'ffmpeg-static';
import ffmpeg from 'fluent-ffmpeg';
import { Canvas, loadImage, registerFont } from 'canvas';
import { stitchFramesToVideo } from './utils/stitchFramesToVideo.js';
import { renderMainComposition } from './compositions/renderMainComposition.js';
import { getVideoFrameReader } from './utils/getVideoFrameReader.js';
```

```
// Tell fluent-ffmpeg where it can find FFmpeg
ffmpeg.setFfmpegPath(ffmpegStatic);

// Clean up the temporary directories first
for (const path of ['out', 'tmp/output']) {
  if (fs.existsSync(path)) {
    await fs.promises.rm(path, { recursive: true });
  }
  await fs.promises.mkdir(path, { recursive: true });
}

// The video length and frame rate, as well as the number of frames required
// to create the video
const duration = 9.15;
const frameRate = 60;
const frameCount = Math.floor(duration * frameRate);

console.log('Extracting frames from video 1...');
const getVideo1Frame = await getVideoFrameReader(
  'assets/pexels-4782135.mp4',
  'tmp/video-1',
  frameRate,
);

console.log('Extracting frames from video 2...');
const getVideo2Frame = await getVideoFrameReader(
  'assets/pexels-3576378.mp4',
  'tmp/video-2',
  frameRate,
);

console.log('Extracting frames from video 3...');
const getVideo3Frame = await getVideoFrameReader(
  'assets/pexels-2829177.mp4',
  'tmp/video-3',
  frameRate,
);

const logo = await loadImage('assets/logo.svg');

// Load fonts so we can use them for drawing
registerFont('assets/caveat-medium.ttf', { family: 'Caveat' });
registerFont('assets/chivo-regular.ttf', { family: 'Chivo' });

const canvas = new Canvas(1280, 720);
const context = canvas.getContext('2d');

// Render each frame
for (let i = 0; i < frameCount; i++) {

  const time = i / frameRate;
```

```
  console.log(`Rendering frame ${i} at ${Math.round(time * 10) / 10} seconds...`);

  // Clear the canvas with a white background color. This is required as we are
  // reusing the canvas with every frame
  context.fillStyle = '#ffffff';
  context.fillRect(0, 0, canvas.width, canvas.height);

  // Grab a frame from our input videos
  const image1 = await getVideo1Frame();
  const image2 = await getVideo2Frame();
  const image3 = await getVideo3Frame();

  renderMainComposition(
    context,
    image1,
    image2,
    image3,
    logo,
    canvas.width,
    canvas.height,
    time,
  );

  // Store the image in the directory where it can be found by FFmpeg
  const output = canvas.toBuffer('image/png');
  const paddedNumber = String(i).padStart(4, '0');
  await fs.promises.writeFile(`tmp/output/frame-${paddedNumber}.png`, output);
}

console.log(`Stitching ${frameCount} frames to video...`);

await stitchFramesToVideo(
  'tmp/output/frame-%04d.png',
  'assets/catch-up-loop-119712.mp3',
  'out/video.mp4',
  duration,
  frameRate,
);
```

We'll finish up by looking at renderMainComposition.js . You can see in the final video that we transition between two scenes: one with polaroid pictures, and one with a logo and caption. The function renderMainComposition() takes care of that. We use interpolateKeyframes() to do the interpolation using two keyframes with Cubic easing.

The renderThreePictures() and renderOutro() functions are then called to render those scenes. As you can see, we are measuring width and height using relative fractions. This allows us to render the video at different resolutions. As you might have noticed, we render at 1280 by 720 as specified in

src/index.js , but we can also render at lower resolutions, such as 480×270 to speed up the rendering process, as long as the aspect ratio remains the same.

To get the transition slide effect, we're using `context.translate()` to offset the location where the scenes are drawn during the transition. To fade the scenes in and out, we're adjusting the opacity with `context.globalAlpha` .

As a final step, we restore the context by calling `context.restore()` before returning to the caller, ensuring the same drawing context as when the function was invoked.

```
import { interpolateKeyframes } from '../utils/interpolateKeyframes.js';
import { renderThreePictures } from './renderThreePictures.js';
import { renderOutro } from './renderOutro.js';

export function renderMainComposition(
  context,
  image1,
  image2,
  image3,
  logo,
  width,
  height,
  time,
) {

  // Interpolate the x position to create a slide effect between the polaroid pictures sce
  // and the outro scene
  const slideProgress = interpolateKeyframes([
    { time: 6.59, value: 0 },
    { time: 7.63, value: 1, easing: 'cubic-in-out' },
  ], time);

  // Scene 1 – The three polaroid pictures

  // Move the slide over 25% of the canvas width while adjusting its opacity with globalAl
  context.save();
  context.translate((0.25 * width) * -slideProgress, 0);
  context.globalAlpha = 1 - slideProgress;

  // Render the polaroid picture scene using relative sizes
  renderThreePictures(context, image1, image2, image3, 0.9636 * width, 0.8843 * height, ti

  context.restore();

  // Scene 2 – The outro

  // Move the slide over 25% of the canvas width while adjusting its opacity with globalAl
  context.save();
  context.translate((0.25 * width) * (1 - slideProgress), 0);
```

```
  context.globalAlpha = slideProgress;

  renderOutro(context, logo, width, height, time - 6.59);

  context.restore();
}
```

Final thoughts

I hope this article provided you with some insight into how to render dynamic videos using Node.js and FFmpeg, and the techniques involved. The next step might be to import dynamic data from a spreadsheet, so you can generate them in bulk. Or you can deploy this code to a server to render customized videos for your website visitors.

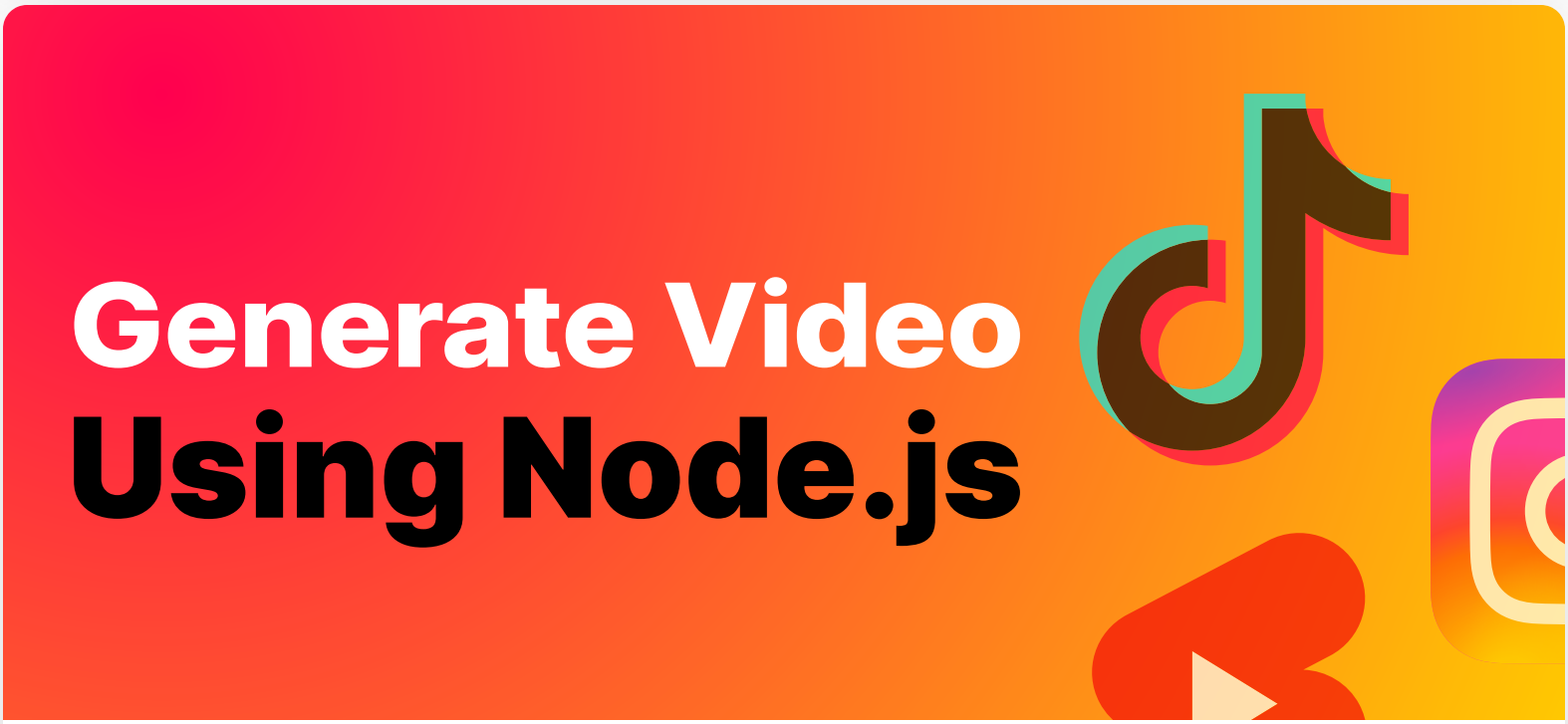
Of course, you need to make your code scalable so that it can handle the load for video rendering, which can quickly get complicated and expensive. This is why we built a cloud service to help you, providing a **template editor** where you can create your videos and then automate them with **no-code integrations** and **API**. If you enjoyed this article, it might be worth checking out.

Happy video rendering!

21 min read

The Best Video Generation APIs for 2023

Looking for the best API to create dynamic video? This list compares features and pricing for Creatomate, Shotstack, Moovly, Bannerbear, Placid, Plainly, Shakr, and more.



15 min read

Using Node.js to Generate Instagram, YouTube, or TikTok Videos

Learn how to use Node.js to programmatically create story videos for posting on social media, such as Instagram Stories, YouTube Shorts, or TikTok videos.

19 min read

Generate Text-to-Speech Video using Node.js and AWS Polly

This tutorial shows how to use code to generate voice over videos for YouTube, Instagram, TikTok, and other platforms.

Start automating today

Start with a full-featured trial with 50 credits – no credit card required.

How to Automate Video Creation with Zapier

Create Many Videos in Bulk using a Spreadsheet

Generate Marketing Videos using a Simple Form

Use ChatGPT to Auto-Create Social Media Content with Zapier

Get Started Creating Video by Code (using the API)

How to Create and Edit Video with PHP

Using ChatGPT's API to Auto-Create Social Media Videos by Code

Using Node.js to Generate Instagram, YouTube, or TikTok Videos

Video Rendering with Node.js and FFmpeg

FFmpeg Tutorials

How-tos

Edit Video by API

Edit Video using Node.js

Slideshow Video using Node.js

Real Estate Video by API

Video Automation

Dynamic Video Creation

Generate Banner Ads by API

Personalized Video by API

Automate Social Media Video

Automate Instagram Video

Automate Twitter Video

Company & Resources

Tutorials & Guides

API Documentation

Knowledge Base

Terms and Conditions

Privacy Policy

About Us