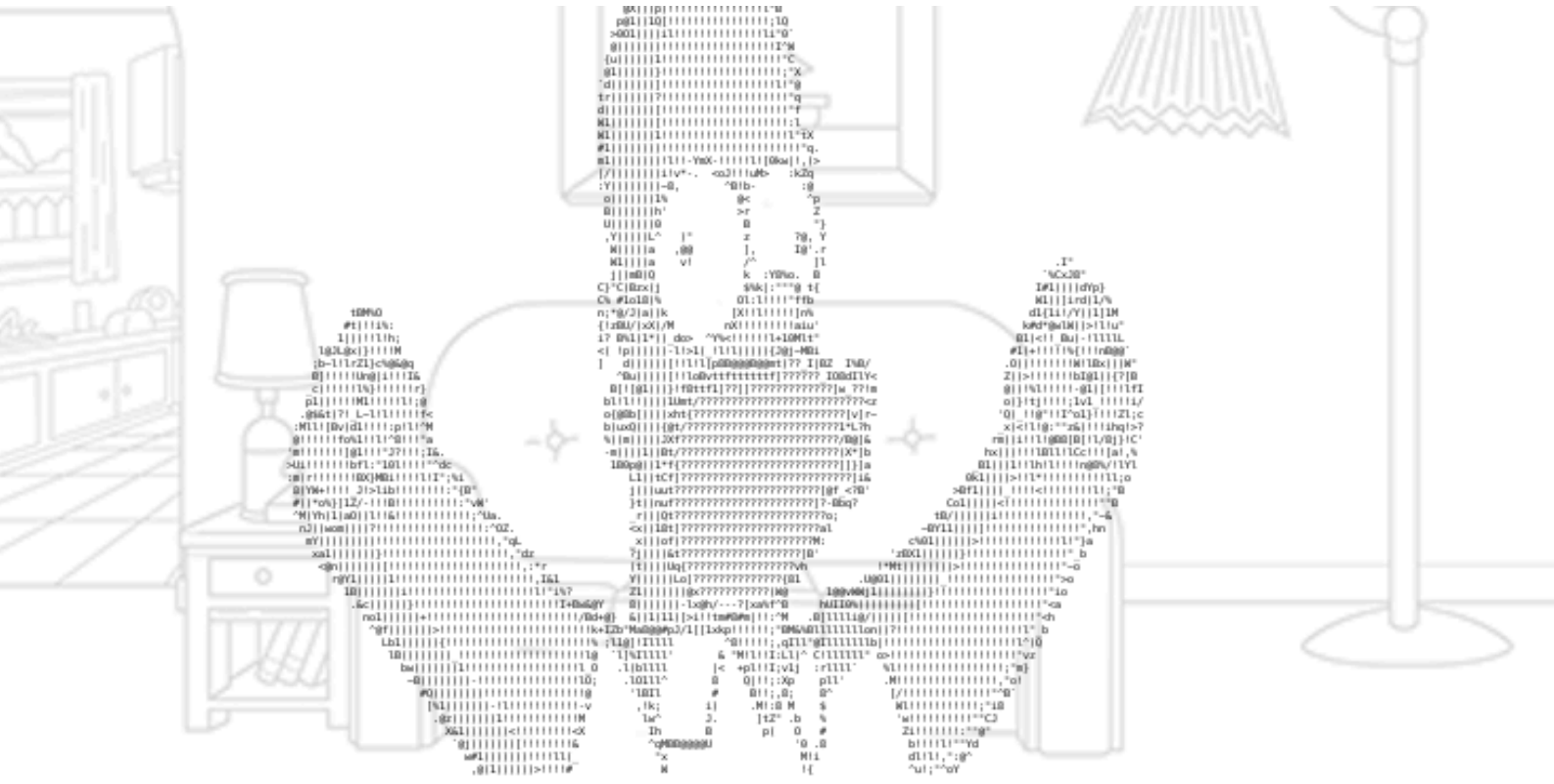


Convert An Image Into An ASCII Art Masterpiece With Pure JavaScript

Jonathan Petitcolas
February 20, 2018

#js #tutorial

While browsing Stack Overflow, I often pay attention to the "Hot Network Questions" sidebar. They bring me to several interesting challenges, not necessarily related to development. Recently, I found an interesting post: [how do ASCII art image conversion algorithms work?]

ASCII art image conversion basically consists in two steps:

1. Convert a picture into gray colors
2. Map each pixel to a given character depending on the grayscale value

For those in a hurry, you can test the converter directly in final demo, or read its source code directly on its GitHub repository.

Uploading an Image into a Canvas

The first step is to allow users to upload a picture. That's the job of the file input element. Moreover, as I'm going to manipulate image pixels, I also need a `canvas`. So here is the main HTML document I'll use:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Ascii Art Converter</title>
</head>
<body>
    <h1>Ascii Art Converter</h1>
    <p>
        <input type="file" name="picture" />
    </p>
    <canvas id="preview"></canvas>
</body>
</html>
```

```
const canvas = document.getElementById("preview");
const fileInput = document.querySelector('input[type="file"');

const context = canvas.getContext("2d");

fileInput.onchange = e => {
  // just handling single file upload
  const file = e.target.files[0];

  const reader = new FileReader();
  reader.onload = event => {
    const image = new Image();
    image.onload = () => {
      canvas.width = image.width;
      canvas.height = image.height;

      context.drawImage(image, 0, 0);
    };

    image.src = event.target.result;
  };

  reader.readAsDataURL(file);
};
```

When the file input changes, I instantiate a new `FileReader` object, which reads the file, and loads it into the `canvas`. Note that I adapt the canvas size to the uploaded image to avoid truncation. The last two arguments of `drawImage` determine the image placement in the canvas. In my case, I we want to start drawing the image from the top left corner (coordinates [0, 0]).

Once I embed this script on the HTML page, I can upload a Homer Simpson image, and it displays in the `canvas` element:

Note: If you want to snap a picture from your webcam, please refer to the Taking Picture From Webcam Using Canvas post in this blog.

Turning an Image into Gray Colors

Now that the image has been uploaded, I need to convert it into a grayscale image. The color of each pixel can be broken down into three distinct components: red, green, and blue values, as in hexadecimal (`#RRGGBB`) colors in CSS.

One simple way to compute corresponding gray scale is to average these three values. However, the human eye is not equally sensitive to these three colors. For instance, our eyes are very sensitive to the green color, while blue is only slightly perceived. Hence, we need to ponderate each color using different weights. After taking a look on the (very) detailed Grayscale Wikipedia Page, I decided to compute the grayscale value using the following formula:

```
GrayScale = 0.21 R + 0.72 G + 0.07 B
```

So I need to iterate on each pixel of the picture, to extract its RGB components, and to replace each pixel by its related grayscale value. The `canvas` API provides a `getImageData` function to manipulate its pixels one by one:

```
  const imageData = context.getImageData(0, 0, width, height);

  const grayScales = [];

  for (let i = 0; i < imageData.data.length; i += 4) {
    const r = imageData.data[i];
    const g = imageData.data[i + 1];
    const b = imageData.data[i + 2];

    const grayScale = toGrayScale(r, g, b);
    imageData.data[i] = imageData.data[i + 1] = imageData.data[
      i + 2
    ] = grayScale;

    grayScales.push(grayScale);
  }

  context.putImageData(imageData, 0, 0);

  return grayScales;
};
```

`getImageData` produces a uni-dimensional array, where each pixel is splitted into four components: red, green, blue, and alpha (for transparency). So I iterate over this array by increments of 4, retrieve the RGB value from the first three items, compute the scale of gray, and then continue until the end.

In this snippet, I modified the original image data. That means the `convertToGrayScales()` function is impure. Indeed, I wasn't able to find a way to update image data using a copy of the `imageData` variable.

After that, it's just a matter of adding a call to `convertToGrayScales` at the end of the `image.onload` listener. And now the uploaded picture displays in gray scale:

Mapping Pixels to Characters

To display the image using the ASCII character set, the next step is to choose one character for each pixel of the image. Some characters are darker than others - for instance, `@` is darker than `.`, because it occupies more space on screen. The following character ramp is generally used for the shade-to-character conversion:

```
$@B%8&WM#*oahkbdpqwmZO0QLCJUYXzcvunxrjft/\|()1{}[]?-_+~<>i!lI;:,"^`'.
```

Mapping a gray scale value to its equivalent character is just an array lookup:

```
const grayRamp =
  "$@B%8&WM#*oahkbdpqwmZO0QLCJUYXzcvunxrjft/|()1{}[]?-_+~<>i!lI;:,\"^`'. ";
const rampLength = grayRamp.length;

// the grayScale value is an integer ranging from 0 (black) to 255 (white)
const getCharacterForGrayScale = grayScale =>
  grayRamp[Math.ceil(((rampLength - 1) * grayScale) / 255)];
```

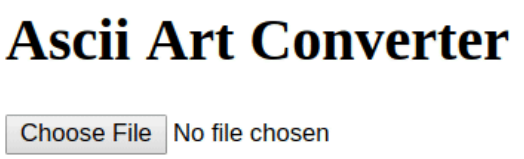
Let's translate the input image into pure characters:

```
const asciiImage = document.querySelector("pre#ascii");
```

```
    asciiImage.textContent = ascii;
  };
```

I use a `<pre>` tag in order to keep the aspect ratio of the picture, as it uses a monospaced font.

Calling the `drawAscii` method at the end of the `image.onload` callback, I get the following result:



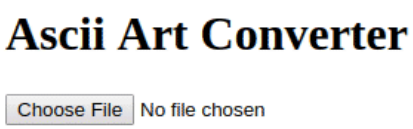
At first glance, it seems it doesn't work. Yet, if you scroll horizontally, you notice some strings wandering through the screen. The picture seems to be on a single line. And indeed: all the pixels are on a single dimensional array. Hence, I need to add a break line every `width` value:

```
const drawAscii = (grayScales, width) => {
  const ascii = grayScales.reduce((asciiImage, grayScale, index) => {
    let nextChars = getCharacterForGrayScale(grayScale);

    if ((index + 1) % width === 0) {
      nextChars += "\n";
    }
```

```
    asciiImage.textContent = ascii;
};
```

The result is now far better, except for a little detail...

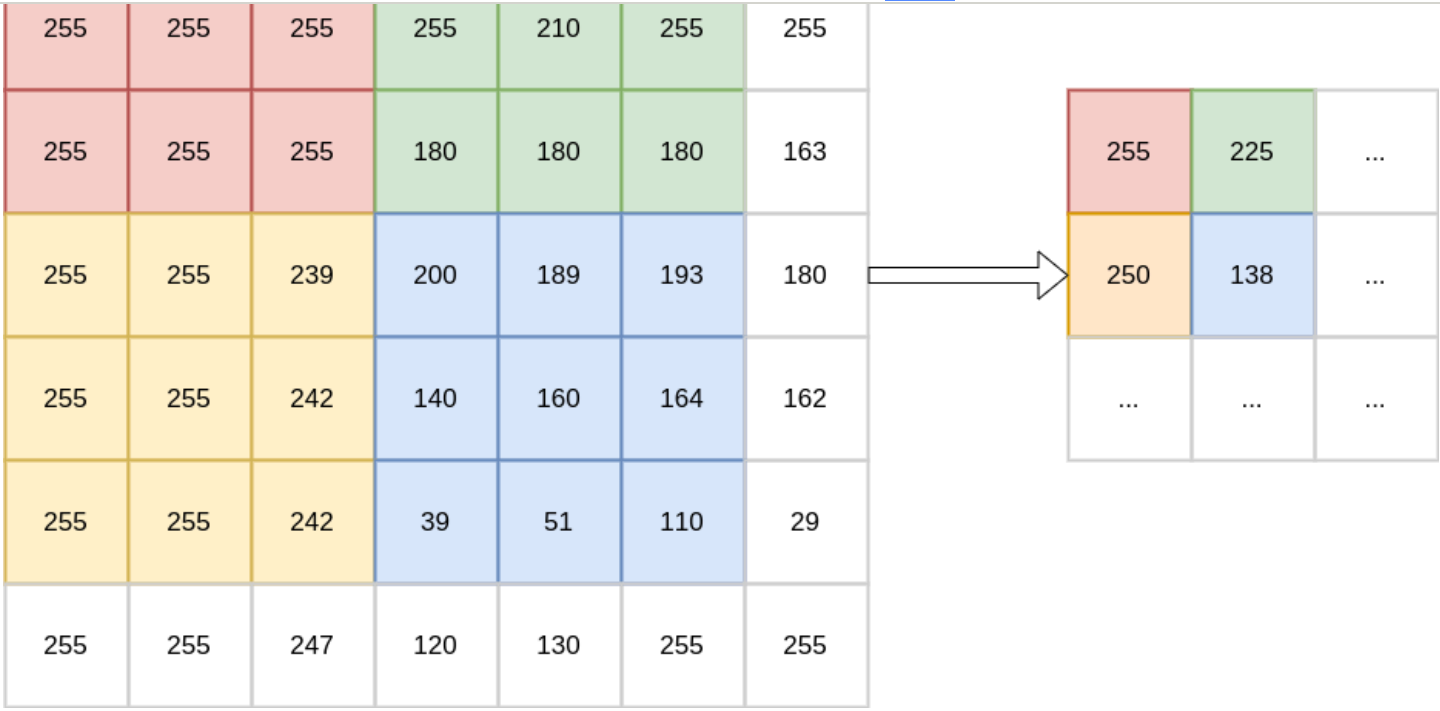


The ASCII representation is huge. Indeed, I mapped every single pixel to a character. Drawing a small picture of 10x10 pixels would then take 10 lines of 10 characters. That's too big.

I could keep this huge text picture and reduce the font size as shown in the previous picture. Yet, that's not optimal, especially to allow to share ASCII images by email.

Lowering ASCII Image Definition

When browsing the Web to check how others achieve resolution downgrade, I found numerous references to the average method, also called subsampling:



This technique consists in taking sub-arrays of pixels, and to compute their average grayscale. Then, instead of drawing 9 white pixels for the red section above, I would draw a single one, still completly white.

I first dove into the code, trying to compute this average on the unidimensional array. Yet, after an hour of tying myself in knots, I remembered the next two arguments of `drawImage` canvas method: the output width and height. Their main goal is to resize the picture before drawing it. Exactly what I need! I wasn't able to find how this is done under the hood, but I guess this is using the same subsampling process.

Let's clamp our image dimension:

```
const MAXIMUM_WIDTH = 80;
const MAXIMUM_HEIGHT = 50;

const clampDimensions = (width, height) => {
  if (height > MAXIMUM_HEIGHT) {
    const reducedWidth = Math.floor((width * MAXIMUM_HEIGHT) / height);
    return [reducedWidth, MAXIMUM_HEIGHT];
  }

  if (width > MAXIMUM_WIDTH) {
    const reducedHeight = Math.floor((height * MAXIMUM_WIDTH) / width);
    return [MAXIMUM_WIDTH, reducedHeight];
  }
```

I focus on height first. Indeed, to better appreciate the artist behind their work, one needs to contemplate their art without scrolling. Also, note that I preserve the image aspect ratio to prevent some weird distortions.

I now need to update the `image.onload` handler to use the clamped values:

```
image.onload = () => {
  const [width, height] = clampDimensions(image.width, image.height);

  canvas.width = width;
  canvas.height = height;

  context.drawImage(image, 0, 0, width, height);
  const grayScales = convertToGrayScales(context, width, height);

  drawAscii(grayScales, width);
};
```

Now, once I upload my favorite Simpson character, here is the result:

```
                                    U88f
                                  mr kzB   C'
                                 8  f   @   t
                                 ^  8@m-!l!{o%
                                w c#1)i!!!!!!!!B
                                B@1L)[!!!!!!!!!IW
                                @)1Y)!!!!!!!!!!,B
                               @o)))[!!!!!!!!!!!"J
                              "1))))!!!!!!!!!!!!!l|
                              @)))))!!!!!!!!!!!!!"|
                              u)))))!!!!!!!!!!!!l,@
                             &gt;1)))))!!!!!!!!!!!!!lf
                             Y1)))))!!!!!!!!!!!!!!I
                             C)))))!!!!!!!!!!!!!!"X
                             ())))))i!!!!l!!!!!!l!ll"X
                             `1)))))?!&]  }&!!)q   p]?
                              t)))))1|      pU       j
                              a)))))0        @       f
                              #))))q         '        ^
                              i))))@  a8      !    &gt;@ l
                               t)1)W  li      !      .
                               8)d1W          "`t@XfC %                          %1
                             ~*@1)@)          @^;ll,|j                           %))
                  '&zo!      ^:fx)X)*       O!!!!!l~^"                          cc/!
```

The resolution has been decreased, and you can't see as many details as before, but that's a mandatory drawback to get shareable ASCII art.

Handling Image Aspect Ratio

The sharpest eyes have probably noticed that the image aspect ratio is not respected. I handled ASCII characters as if they were square, but in reality, characters are circumscribed in a rectangle. Hence, I need to reflect the distortion on the image pixel cut.

As I chose a monospaced font, the width of every character is the same. Hence, the aspect ratio is be the same for all characters. Yet, how can I compute it? I didn't find any

```
  const pre = document.createElement("pre");
  pre.style.display = "inline";
  pre.textContent = " ";

  document.body.appendChild(pre);
  const { width, height } = pre.getBoundingClientRect();
  document.body.removeChild(pre);

  return height / width;
};

const fontRatio = getFontRatio();
```

The trick lies in adding a `<pre>` element (to keep exact styling), and to compute the display dimensions using the `getBoundingClientRect()` function.

Let's update the `clampDimensions` function to take font ratio into account:

```
 const clampDimensions = (width, height) => {
+    const rectifiedWidth = Math.floor(getFontRatio() * width);
+
     if (height > MAXIMUM_HEIGHT) {
-        const reducedWidth = Math.floor(width * MAXIMUM_HEIGHT / height);
+        const reducedWidth = Math.floor(rectifiedWidth * MAXIMUM_HEIGHT / height);
         return [reducedWidth, MAXIMUM_HEIGHT];
     }

     if (width > MAXIMUM_WIDTH) {
-        const reducedHeight = Math.floor(height * MAXIMUM_WIDTH / width);
+        const reducedHeight = Math.floor(height * MAXIMUM_WIDTH / rectifiedWidth);
         return [MAXIMUM_WIDTH, reducedHeight];
     }

-    return [width, height];
+    return [rectifiedWidth, height];
 };
```

I just compute a cross-shaped product between the font ratio and the maximum dimensions.

And now, my ASCII artwork looks like a masterpiece:


JZ WB@ov11<!
@ lB@LB))))_!!!!!!
_ak@j1)zM))1i!!!!!!!
#B()))u&11!!!!!!!!!
;@|&1)))))1)!!!!!!!!!!
}&1)))))))))~!!!!!!!!!!
_B1))))))))))!!!!!!!!!!
"@1))))))))))1!!!!!!!!!!
oQ))))))))))){!!!!!!!!!!
^@()))))))))))1!!!!!!!!!!
<81))))))))))1!!!!!!!!
+W1))))))))))!!!!!!!!!!
,%1))))))))))>!!!!!!!
.%()))))))))))]!!!>m$M
pU))))))))))))!ob.
;W1))))))))))hO
kj)))))))))1B!
!81)))))))a)
&n)))))))1B 1@@h
;&1)))1))18 l]
#x)1aOB11oI
'@B r*)(%1J#1(B'
.B,@I@1Zd)1oU)1Bl
W< 8Bv8t))(@)))x$[
b/ O@M1)))cf)))){l1W$aUju
o1 '@|)))))))){!!!!l
MJ{)))))))))!!!!!
U8[lI+**11))))1!!_Battt
8+il>l!!i)))))))-@Ytf
?h)tx##8U))))))1pM/f)?
^81)/@()1))))))q*tf[?????
+@|)))))11)))v8tt)??????
}B$B@$w1))1&ztf???????
1k))))(@tf|???????
<&1)))|Btt1?
;@1))))Bft)???????
:@1))))wktt???????
I@1))))1#Ot)??????
i81))))))kMt[??
|a1)))))))1%M)????
JJ))))))))))[C@k}?
B()))))))))))
lz@ol<B@; >W{)|nLbM8
l&f^llIc@%0(>;;Illll"
,llU@+lllllll`
^lIWqllllll"
:lfBlllI
.li@_I
]&@@@Q.
8Y1)-!!!~&k
^]c%{)))?!l!!!!hm
@0[i!l<*M1))i!li}cq@-
jo[!!!!!!llhLt@8x-l!!!l]@>
#))!!!!!!!!l@Y}!!!!!!!!!,oC
x1))(_!!!!!@)~!!!!!!!ll!!,W|
+}m@Bv1)>i%/-!!!!!!,!|l!!!^B:
!ll!!lv@vub1!!!!!!;"@~!!!!,_%
!!!!!!!l}@b1!!!!!"^8(!!!!!I"|$i
!!!!!!!!!!BXi!""^QB!l!!!!!!l""`L$-
)!!!!!!!!lzqW$$@ux@!!!!!!!!!!l"""x$X
BM@q-i!!+1&u{!!l!!wJ!!!!!!!!!!!!iI"^>8&,
|@m{1)1{p@|))[l!!!rYl!!!!!!!!!!!!!!!!;^`z$U
))1(XLX11)))))}!!!!!!!!!!!!!!!!!!!!!!!!!!l""l#B!
@m1))))))))))))}!!!!!!!!!!!!!!!!!!!!!!!!!!!:"^(@m
aBj1))))))))))<!!!!!!!!!!!!!!!!!!!!!!!!!!ll!!!:""q@1
d@k|))))))))))+!!!!!!!!!!!!!!!!!!!!!!!!!!!!!l"^?%h'
:B%))))))))))}!!!!!!!!!!!!!!!!!!!!!!!!!!!!!l!i;"`U$x
}$Z1)))))))){!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!;>8@M<&8@~
Q@r))))))))))1>!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!lz@ol<B@; >W{)|nLbM8
`M8()))))))))))~!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!l&f^llIc@%0(>;;Illll"
?@d1))))))))))~l!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!}@ ,llU@+lllllll`
v@U1))))))))))<!!!!!!!!!!!!!!!!!!!!!!!!!!l!Mc. ^lIWqllllll"
k@f))))))))))1>!!!!!!!!!!!!!!!!!!!!!!!!l|B :lfBlllI
;B&)1))))))))))}!!!l!!!!!!!!!!!!!!!!l>@" .li@_I

```
            0@J)))))))))))~l!!(%                        .M
                &Bt))))))))))?kU                         p
                   +@#1))))))f$,
                       UBL1)1WZll`
                       '@:O@%llllll,
```

Far better, isn't it? Reducing the font size also helps to get a better overview of the converted picture.

Conclusion

As usual, here are the related links:

- Final ASCII Art Converter Demo,
- GitHub Repository

Note that I only handled a static image in this case, but the same technique applies to video streams. It exists: take a look at the ASCII camera. Useless, therefore indispensable!

Did you like this article? Share it!

ABOUT THE AUTHOR

Jonathan Petitcolas

Full-stack web developer at marmelab - Node.js, React, Angular, Symfony, Go, Arduino, Docker. Can fit a huge number of puns into a single sentence.

admin and greenframe.io.

LET'S WORK TOGETHER ON YOUR NEXT PROJECT!

CONTACT US

COMMENTS

Login

Add a comment

M↓ MARKDOWN

ADD COMMENT

Upvotes Newest Oldest

Powered by **Commento**

4 rue Girardet, 54 000 Nancy, FRANCE