

Pixel manipulation with canvas

[< Previous](#)[Next >](#)

Until now we haven't looked at the actual pixels of our canvas. With the `ImageData` object you can directly read and write a data array to manipulate pixel data. We will also look into how image smoothing (anti-aliasing) can be controlled and how to save images from your canvas.

The ImageData object

The `ImageData` object represents the underlying pixel data of an area of a canvas object. Its `data` property returns a `Uint8ClampedArray` (or `Float16Array` if requested) which can be accessed to look at the raw pixel data; each pixel is represented by four one-byte values (red, green, blue, and alpha, in that order; that is, "RGBA" format). Each color component is represented by an integer between 0 and 255. Each component is assigned a consecutive index within the array, with the top left pixel's red component being at index 0 within the array. Pixels then proceed from left to right, then downward, throughout the array.

The `Uint8ClampedArray` contains `height × width × 4` bytes of data, with index values ranging from 0 to `(height × width × 4) - 1`.

For example, to read the blue component's value from the pixel at column 200, row 50 in the image, you would do the following:

JS

```
const blueComponent = imageData.data[50 * (imageData.width * 4) + 200 * 4 + 2];
```

If given a set of coordinates (X and Y), you may end up doing something like this:

JS

```
const xCoord = 50;
const yCoord = 100;
const canvasWidth = 1024;

const getColorIndicesForCoord = (x, y, width) => {
  const red = y * (width * 4) + x * 4;
  return [red, red + 1, red + 2, red + 3];
};
```

```
const colorIndices = getColorIndicesForCoord(xCoord, yCoord, canvasWidth);

const [redIndex, greenIndex, blueIndex, alphaIndex] = colorIndices;
```

You may also access the size of the pixel array in bytes by reading the `Uint8ClampedArray.length` attribute:

JS

```
const numBytes = imageData.data.length;
```

Creating an ImageData object

To create a new, blank `ImageData` object, you should use the `createImageData()` method. There are two versions of the `createImageData()` method:

JS

```
const myImageData = ctx.createImageData(width, height);
```

This creates a new `ImageData` object with the specified dimensions. All pixels are preset to transparent.

You can also create a new `ImageData` object with the same dimensions as the object specified by `anotherImageData`. The new object's pixels are all preset to transparent black. **This does not copy the image data!**

JS

```
const myImageData = ctx.createImageData(anotherImageData);
```


Getting the pixel data for a context

To obtain an `ImageData` object containing a copy of the pixel data for a canvas context, you can use the `getImageData()` method:

JS

```
const myImageData = ctx.getImageData(left, top, width, height);
```

This method returns an `ImageData` object representing the pixel data for the area of the canvas whose corners are represented by the points `(left, top)`, `(left+width, top)`, `(left, top+height)`, and `(left+width, top+height)`. The coordinates are specified in canvas coordinate space units.

 **Note:** Any pixels outside the canvas are returned as transparent black in the resulting `ImageData` object.

This method is also demonstrated in the article [Manipulating video using canvas](#).

Creating a color picker

In this example, we are using the `getImageData()` method to display the color under the mouse cursor. For this, we need the current position of the mouse, then we look up the pixel data at that position in the pixel array that `getImageData()` provides. Finally, we use the array data to set a background color and a text in the `<div>` to display the color. Clicking on the image will do the same operation but uses the selected color.

HTML

```
<table>
  <thead>
    <tr>
      <th>Source</th>
      <th>Hovered color</th>
      <th>Selected color</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>
        <canvas id="canvas" width="300" height="227"></canvas>
      </td>
      <td class="color-cell" id="hovered-color"></td>
      <td class="color-cell" id="selected-color"></td>
    </tr>
  </tbody>
</table>
```

JS

```
const img = new Image();
img.crossOrigin = "anonymous";
img.src = "/shared-assets/images/examples/rhino.jpg";
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");
img.addEventListener("load", () => {
  ctx.drawImage(img, 0, 0);
  img.style.display = "none";
});
const hoveredColor = document.getElementById("hovered-color");
const selectedColor = document.getElementById("selected-color");
```

```
const pick = (event, destination) => {  
  const bounding = canvas.getBoundingClientRect();  
  const x = event.clientX - bounding.left;  
  const y = event.clientY - bounding.top;  
  const pixel = ctx.getImageData(x, y, 1, 1);  
  const data = pixel.data;  
  
  const rgbColor = `rgb(${data[0]} ${data[1]} ${data[2]} / ${data[3]} / 255)`;  
  destination.style.background = rgbColor;  
  destination.textContent = rgbColor;  
  
  return rgbColor;  
};
```

```
canvas.addEventListener("mousemove", (event) => pick(event, hoveredColor));  
canvas.addEventListener("click", (event) => pick(event, selectedColor));
```

Hover your cursor anywhere over the image to see the result in the "Hovered color" column. Click anywhere in the image to see the result in the "Selected color" column.

Source**Hovered color****Selected color**

Painting pixel data into a context

You can use the `putImageData()` method to paint pixel data into a context:

JS

```
ctx.putImageData(myImageData, dx, dy);
```

The `dx` and `dy` parameters indicate the device coordinates within the context at which to paint the top left corner of the pixel data you wish to draw.

For example, to paint the entire image represented by `myImageData` to the top left corner of the context, you can do the following:

JS

```
ctx.putImageData(myImageData, 0, 0);
```

Grayscale and inverting colors

In this example, we iterate over all pixels to change their values, then we put the modified pixel array back onto the canvas using `putImageData()`. The `invert` function subtracts each color from the max value, `255`. The `grayscale` function uses the average of red, green and blue. You can also use a weighted average, given by the formula $x = 0.299r + 0.587g + 0.114b$, for example. See [Grayscale](#) on Wikipedia for more information.

HTML

```
<canvas id="canvas" width="300" height="227"></canvas>
<form>
  <input type="radio" id="original" name="color" value="original" checked />
  <label for="original">Original</label>

  <input type="radio" id="grayscale" name="color" value="grayscale" />
  <label for="grayscale">Grayscale</label>

  <input type="radio" id="inverted" name="color" value="inverted" />
  <label for="inverted">Inverted</label>

  <input type="radio" id="sepia" name="color" value="sepia" />
  <label for="sepia">Sepia</label>
</form>
```

JS

```
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

const img = new Image();
img.crossOrigin = "anonymous";
img.src = "/shared-assets/images/examples/rhino.jpg";
img.onload = () => {
  ctx.drawImage(img, 0, 0);
}
```

```
};

const original = () => {
  ctx.drawImage(img, 0, 0);
};

const invert = () => {
  ctx.drawImage(img, 0, 0);
  const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
  const data = imageData.data;
  for (let i = 0; i < data.length; i += 4) {
    data[i] = 255 - data[i]; // red
    data[i + 1] = 255 - data[i + 1]; // green
    data[i + 2] = 255 - data[i + 2]; // blue
  }
  ctx.putImageData(imageData, 0, 0);
};

const grayscale = () => {
  ctx.drawImage(img, 0, 0);
  const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
  const data = imageData.data;
  for (let i = 0; i < data.length; i += 4) {
    const avg = (data[i] + data[i + 1] + data[i + 2]) / 3;
    data[i] = avg; // red
    data[i + 1] = avg; // green
    data[i + 2] = avg; // blue
  }
  ctx.putImageData(imageData, 0, 0);
};

const sepia = () => {
  ctx.drawImage(img, 0, 0);
  const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
  const data = imageData.data;
  for (let i = 0; i < data.length; i += 4) {
    let r = data[i], // red
        g = data[i + 1], // green
        b = data[i + 2]; // blue

    data[i] = Math.min(Math.round(0.393 * r + 0.769 * g + 0.189 * b), 255);
    data[i + 1] = Math.min(Math.round(0.349 * r + 0.686 * g + 0.168 * b), 255);
    data[i + 2] = Math.min(Math.round(0.272 * r + 0.534 * g + 0.131 * b), 255);
  }
  ctx.putImageData(imageData, 0, 0);
};

const inputs = document.querySelectorAll("[name=color]");
for (const input of inputs) {
```

```
input.addEventListener("change", (evt) => {  
  switch (evt.target.value) {  
    case "inverted":  
      return invert();  
    case "grayscale":  
      return grayscale();  
    case "sepia":  
      return sepia();  
    default:  
      return original();  
  }  
});  
}
```

Click different options to view the result in action.



☒ Original ☐ Grayscale ☐ Inverted ☐ Sepia

Zooming and anti-aliasing

With the help of the `drawImage()` method, a second canvas, and the `imageSmoothingEnabled` property, we are able to zoom in on our picture and see the details. A third canvas without `imageSmoothingEnabled` is also drawn to allow a side by side comparison.

HTML

```
<table>  
  <thead>  
    <tr>  
      <th>Source</th>  
      <th>Smoothing enabled = true</th>  
      <th>Smoothing enabled = false</th>
```

```

    </tr>
  </thead>
  <tbody>
    <tr>
      <td>
        <canvas id="canvas" width="300" height="227"></canvas>
      </td>
      <td>
        <canvas id="smoothed" width="200" height="200"></canvas>
      </td>
      <td>
        <canvas id="pixelated" width="200" height="200"></canvas>
      </td>
    </tr>
  </tbody>
</table>

```

We get the position of the mouse and crop an image of 5 pixels left and above to 5 pixels right and below. Then we copy that one over to another canvas and resize the image to the size we want it to. In the zoom canvas we resize a 10×10 pixel crop of the original canvas to 200×200:

JS

```

const img = new Image();
img.crossOrigin = "anonymous";
img.src = "/shared-assets/images/examples/rhino.jpg";
img.onload = () => {
  draw(img);
};

const draw = (image) => {
  const canvas = document.getElementById("canvas");
  const ctx = canvas.getContext("2d");
  ctx.drawImage(image, 0, 0);

  const smoothCtx = document.getElementById("smoothed").getContext("2d");
  smoothCtx.imageSmoothingEnabled = true;

  const pixelatedCtx = document.getElementById("pixelated").getContext("2d");
  pixelatedCtx.imageSmoothingEnabled = false;

  const zoom = (ctx, x, y) => {
    ctx.drawImage(
      canvas,
      Math.min(Math.max(0, x - 5), image.width - 10),
      Math.min(Math.max(0, y - 5), image.height - 10),
      10,
      10,

```



```
0,  
0,  
200,  
200,  
);  
};  
  
canvas.addEventListener("mousemove", (event) => {  
  const x = event.layerX;  
  const y = event.layerY;  
  zoom(smoothCtx, x, y);  
  zoom(pixelatedCtx, x, y);  
});  
};
```

Source



Smoothing enabled = true

Smoothing enabled :

Saving images

The `HTMLCanvasElement` provides a `toDataURL()` method, which is useful when saving images. It returns a [data URL](#) containing a representation of the image in the format specified by the `type` parameter (defaults to [PNG](#) [↗](#)). The returned image is in a resolution of 96 dpi.

Note: Be aware that if the canvas contains any pixels that were obtained from another [origin](#) without using CORS, the canvas is **tainted** and its contents can no longer be read and saved. See [Security and tainted canvases](#).

```
canvas.toDataURL('image/png')
```

Default setting. Creates a PNG image.

```
canvas.toDataURL('image/jpeg', quality)
```

Creates a JPG image. Optionally, you can provide a quality in the range from 0 to 1, with one being the best quality and with 0 almost not recognizable but small in file size.

Once you have generated a data URL from your canvas, you are able to use it as the source of any `` or put it into a hyperlink with a [download attribute](#) to save it to disc, for example.

You can also create a `Blob` from the canvas.

```
canvas.toBlob(callback, type, encoderOptions)
```

Creates a `Blob` object representing the image contained in the canvas.

See also

- `ImageData`
- [Manipulating video using canvas](#)
- [Download Canvas API-Generated Images Using toBlob](#) ↗

[← Previous](#)[Next →](#)



Your blueprint for a better internet.