Basic usage of canvas

← Previous Next →

Let's start this tutorial by looking at the `<canvas>` HTML element itself. At the end of this page, you will know how to set up a canvas 2D context and have drawn a first example in your browser.

The `<canvas>` element

HTML

```
<canvas id="canvas" width="150" height="150"></canvas>
```

At first sight a `<canvas>` looks like the `<img>` element, with the only clear difference being that it doesn't have the `src` and `alt` attributes. Indeed, the `<canvas>` element has only two attributes, `width` and `height`. These are both optional and can also be set using DOM properties. When no `width` and `height` attributes are specified, the canvas will initially be **300 pixels** wide and **150 pixels** high. The element can be sized arbitrarily by CSS, but during rendering the image is scaled to fit its layout size: if the CSS sizing doesn't respect the ratio of the initial canvas, it will appear distorted.

> **Note:** If your renderings seem distorted, try specifying your `width` and `height` attributes explicitly in the `<canvas>` attributes, and not using CSS.

The `id` attribute isn't specific to the `<canvas>` element but is one of the global HTML attributes which can be applied to any HTML element (like `class` for instance). It is always a good idea to supply an `id` because this makes it much easier to identify it in a script.

The `<canvas>` element can be styled just like any normal image (`margin`, `border`, `background`...). These rules, however, don't affect the actual drawing on the canvas. We'll see how this is done in a dedicated chapter of this tutorial. When no styling rules are applied to the canvas it will initially be fully transparent.

Accessible content

The `<canvas>` element, like the `<img>`, `<video>`, `<audio>`, and `<picture>` elements, must be made accessible by providing fallback text to be displayed when the media doesn't load or the user is unable to experience it as intended. You should always provide fallback content, captions, and alternative text, as appropriate for the media type.

Providing fallback content is very straightforward: just insert the alternate content inside the `<canvas>` element to be accessed by screen readers, spiders, and other automated bots. Browsers, by default, will ignore the content inside the container, rendering the canvas normally unless `<canvas>` isn't supported.

For example, we could provide a text description of the canvas content or provide a static image of the dynamically rendered content. This can look something like this:

HTML

```
<canvas id="stockGraph" width="150" height="150">
  current stock price: $3.15 + 0.15
</canvas>

<canvas id="clock" width="150" height="150">
  <img src="images/clock.png" width="150" height="150" alt="A clock" />
</canvas>
```

Telling the user to use a different browser that supports canvas does not help users who can't read the canvas at all. Providing useful fallback text or sub DOM adds accessibility to an otherwise non-accessible element.

Required `</canvas>` tag

As a consequence of the way fallback is provided, unlike the `<img>` element, the `<canvas>` element **requires** the closing tag (`</canvas>`). If this tag is not present, the rest of the document would be considered the fallback content and wouldn't be displayed.

If fallback content is not needed, a simple `<canvas id="foo" role="presentation" …></canvas>` is fully compatible with all browsers that support canvas at all. This should only be used if the canvas is purely presentational.

The rendering context

The `<canvas>` element creates a fixed-size drawing surface that exposes one or more **rendering contexts**, which are used to create and manipulate the content shown. In this tutorial, we focus on the 2D rendering context. Other contexts may provide different types of rendering; for example, WebGL uses a 3D context based on OpenGL ES.

The canvas is initially blank. To display something, a script first needs to access the rendering context and draw on it. The `<canvas>` element has a method called `getContext()`, used to obtain the rendering context and its drawing functions. `getContext()` takes one parameter, the type of context. For 2D graphics, such as those covered by this tutorial, you specify `"2d"` to get a `CanvasRenderingContext2D`.

JS

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

The first line in the script retrieves the node in the DOM representing the `<canvas>` element by calling the `document.getElementById()` method. Once you have the element node, you can access the drawing context using its `getContext()` method.

Checking for support

The fallback content is displayed in browsers which do not support `<canvas>`. Scripts can also check for support programmatically by testing for the presence of the `getContext()` method. Our code snippet from above becomes something like this:

JS

```
const canvas = document.getElementById("canvas");

if (canvas.getContext) {
  const ctx = canvas.getContext("2d");
  // drawing code here
} else {
  // canvas-unsupported code here
}
```

A skeleton template

Here is a minimalistic template, which we'll be using as a starting point for later examples.

> **Note:** It is not good practice to embed a script inside HTML. We do it here to keep the example concise.

HTML

```
<!doctype html>
<html lang="en-US">
  <head>
    <meta charset="utf-8" />
    <title>Canvas tutorial</title>
    <style>
      canvas {
        border: 1px solid black;
      }
    </style>
  </head>
  <body>
    <canvas id="canvas" width="150" height="150"></canvas>
    <script>
      function draw() {
        const canvas = document.getElementById("canvas");
        if (canvas.getContext) {
          const ctx = canvas.getContext("2d");
        }
```

```
      }
      window.addEventListener("load", draw);
    </script>
  </body>
</html>
```

The script includes a function called `draw()`, which is executed once the page finishes loading; this is done by putting the script after the main body content. This function, or one like it, could also be called using `setTimeout()`, `setInterval()`, or the `load` event handler, as long as the page has been loaded first.

At this point, this document should be rendered blank.

A simple example

To begin, let's take a look at an example that draws two intersecting rectangles, one of which has alpha transparency. We'll explore how this works in more detail in later examples. Update your `script` element content to this:

JS

```
function draw() {
  const canvas = document.getElementById("canvas");
  if (canvas.getContext) {
    const ctx = canvas.getContext("2d");

    ctx.fillStyle = "rgb(200 0 0)";
    ctx.fillRect(10, 10, 50, 50);

    ctx.fillStyle = "rgb(0 0 200 / 50%)";
    ctx.fillRect(30, 30, 50, 50);
  }
}
draw();
```

This example looks like this: