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.

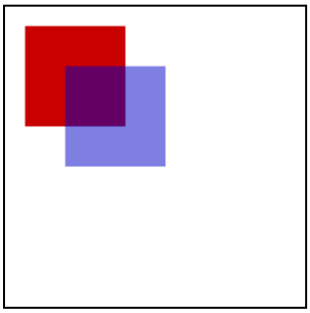
A simple example

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:



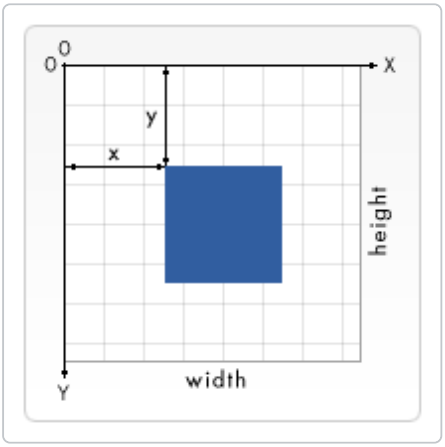
Drawing shapes with canvas

← Previous Next →

Now that we have set up our canvas environment, we can get into the details of how to draw on the canvas. By the end of this article, you will have learned how to draw rectangles, triangles, lines, arcs and curves, providing familiarity with some of the basic shapes. Working with paths is essential when drawing objects onto the canvas and we will see how that can be done.

The grid

Before we can start drawing, we need to talk about the canvas grid or **coordinate space**. Our HTML skeleton from the previous page had a canvas element 150 pixels wide and 150 pixels high.



Normally 1 unit in the grid corresponds to 1 pixel on the canvas. The origin of this grid is positioned in the *top left* corner at coordinate (0,0). All elements are placed relative to this origin. So the position of the top left corner of the blue square becomes x pixels from the left and y pixels from the top, at coordinate (x,y). Later in this tutorial we'll see how we can translate the origin to a different position, rotate the grid and even scale it, but for now we'll stick to the default.

Drawing rectangles

Unlike SVG, `<canvas>` only supports two primitive shapes: rectangles and paths (lists of points connected by lines). All other shapes must be created by combining one or more paths. Luckily, we have an assortment of path drawing functions which make it possible to compose very complex shapes.

First let's look at the rectangle. There are three functions that draw rectangles on the canvas:

`fillRect(x, y, width, height)`

Draws a filled rectangle.

`strokeRect(x, y, width, height)`

Draws a rectangular outline.

`clearRect(x, y, width, height)`

Clears the specified rectangular area, making it fully transparent.

Each of these three functions takes the same parameters. `x` and `y` specify the position on the canvas (relative to the origin) of the top-left corner of the rectangle. `width` and `height` provide the rectangle's size.

Below is the `draw()` function from the previous page, but now it is making use of these three functions.

Rectangular shape example

JS

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

    ctx.fillRect(25, 25, 100, 100);
    ctx.clearRect(45, 45, 60, 60);
    ctx.strokeRect(50, 50, 50, 50);
  }
}
```

This example's output is shown below.

The `fillRect()` function draws a large black square 100 pixels on each side. The `clearRect()` function then erases a 60x60 pixel square from the center, and then `strokeRect()` is called to create a rectangular outline 50x50 pixels within the cleared square.

In upcoming pages we'll see two alternative methods for `clearRect()`, and we'll also see how to change the color and stroke style of the rendered shapes.

Unlike the path functions we'll see in the next section, all three rectangle functions draw immediately to the canvas.

Drawing paths

Now let's look at paths. A path is a list of points, connected by segments of lines that can be of different shapes, curved or not, of different width and of different color. A path, or even a subpath, can be closed. To make shapes using paths, we take some extra steps:

1. First, you create the path.
2. Then you use drawing commands to draw into the path.
3. Once the path has been created, you can stroke or fill the path to render it.

Here are the functions used to perform these steps:

`beginPath()`

Creates a new path. Once created, future drawing commands are directed into the path and used to build the path up.

Path methods

Methods to set different paths for objects.

`closePath()`

Adds a straight line to the path, going to the start of the current sub-path.

`stroke()`

Draws the shape by stroking its outline.

`fill()`

Draws a solid shape by filling the path's content area.

The first step to create a path is to call the `beginPath()`. Internally, paths are stored as a list of sub-paths (lines, arcs, etc.) which together form a shape. Every time this method is called, the list is reset and we can start drawing new shapes.

> **Note:** When the current path is empty, such as immediately after calling `beginPath()`, or on a newly created canvas, the first path construction command is always treated as a `moveTo()`, regardless of what it actually is. For that reason, you will almost always want to specifically set your starting position after resetting a path.

The second step is calling the methods that actually specify the paths to be drawn. We'll see these shortly.

The third, and an optional step, is to call `closePath()`. This method tries to close the shape by drawing a straight line from the current point to the start. If the shape has already been closed or there's only one point in the list, this function does nothing.

> **Note:** When you call `fill()`, any open shapes are closed automatically, so you don't have to call `closePath()`. This is **not** the case when you call `stroke()`.

Drawing a triangle

For example, the code for drawing a triangle would look something like this:

JS

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

    ctx.beginPath();
    ctx.moveTo(75, 50);
    ctx.lineTo(100, 75);
    ctx.lineTo(100, 25);
    ctx.fill();
  }
}
```

The result looks like this:

Moving the pen

One very useful function, which doesn't actually draw anything but becomes part of the path list described above, is the `moveTo()` function. You can probably best think of this as lifting a pen or pencil from one spot on a piece of paper and placing it on the next.

`moveTo(x, y)`

Moves the pen to the coordinates specified by `x` and `y`.

When the canvas is initialized or `beginPath()` is called, you typically will want to use the `moveTo()` function to place the starting point somewhere else. We could also use `moveTo()` to draw unconnected paths. Take a look at the smiley face below.

To try this for yourself, you can use the code snippet below. Just paste it into the `draw()` function we saw earlier.

JS

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

```
    ctx.beginPath();
    ctx.arc(75, 75, 50, 0, Math.PI * 2, true); // Outer circle
    ctx.moveTo(110, 75);
    ctx.arc(75, 75, 35, 0, Math.PI, false); // Mouth (clockwise)
    ctx.moveTo(65, 65);
    ctx.arc(60, 65, 5, 0, Math.PI * 2, true); // Left eye
    ctx.moveTo(95, 65);
    ctx.arc(90, 65, 5, 0, Math.PI * 2, true); // Right eye
    ctx.stroke();
  }
}
```

The result looks like this:

If you'd like to see the connecting lines, you can remove the lines that call `moveTo()`.

> **Note:** To learn more about the `arc()` function, see the Arcs section below.

Lines

For drawing straight lines, use the `lineTo()` method.

`lineTo(x, y)`

Draws a line from the current drawing position to the position specified by `x` and `y`.

This method takes two arguments, `x` and `y`, which are the coordinates of the line's end point. The starting point is dependent on previously drawn paths, where the end point of the previous path is the starting point for the following, etc. The starting point can also be changed by using the `moveTo()` method.

The example below draws two triangles, one filled and one outlined.

JS

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

```
    // Filled triangle
    ctx.beginPath();
    ctx.moveTo(25, 25);
    ctx.lineTo(105, 25);
    ctx.lineTo(25, 105);
    ctx.fill();

    // Stroked triangle
    ctx.beginPath();
    ctx.moveTo(125, 125);
    ctx.lineTo(125, 45);
    ctx.lineTo(45, 125);
    ctx.closePath();
    ctx.stroke();
  }
}
```

This starts by calling `beginPath()` to start a new shape path. We then use the `moveTo()` method to move the starting point to the desired position. Below this, two lines are drawn which make up two sides of the triangle.

You'll notice the difference between the filled and stroked triangle. This is, as mentioned above, because shapes are automatically closed when a path is filled, but not when they are stroked. If we left out the `closePath()` for the stroked triangle, only two lines would have been drawn, not a complete triangle.

Arcs

To draw arcs or circles, we use the `arc()` or `arcTo()` methods.

`arc(x, y, radius, startAngle, endAngle, counterclockwise)`

Draws an arc which is centered at *(x, y)* position with radius *r* starting at *startAngle* and ending at *endAngle* going in the given direction indicated by *counterclockwise* (defaulting to clockwise).

`arcTo(x1, y1, x2, y2, radius)`

Draws an arc with the given control points and radius, connected to the previous point by a straight line.

Let's have a more detailed look at the `arc` method, which takes six parameters: `x` and `y` are the coordinates of the center of the circle on which the arc should be drawn. `radius` is self-explanatory. The `startAngle` and `endAngle` parameters define the start and end points of the arc in radians, along the curve of the circle. These are measured from

the x axis. The `counterclockwise` parameter is a Boolean value which, when `true`, draws the arc counterclockwise; otherwise, the arc is drawn clockwise.

> **Note:** Angles in the `arc` function are measured in radians, not degrees. To convert degrees to radians you can use the following JavaScript expression: `radians = (Math.PI/180)*degrees`.

The following example is a little more complex than the ones we've seen above. It draws 12 different arcs all with different angles and fills.

The two `for` loops are for looping through the rows and columns of arcs. For each arc, we start a new path by calling `beginPath()`. In the code, each of the parameters for the arc is in a variable for clarity, but you wouldn't necessarily do that in real life.

The `x` and `y` coordinates should be clear enough. `radius` and `startAngle` are fixed. The `endAngle` starts at `180` degrees (half a circle) in the first column and is increased by steps of `90` degrees, culminating in a complete circle in the last column.

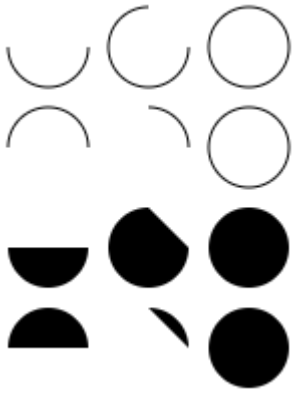
The statement for the `clockwise` parameter results in the first and third row being drawn as clockwise arcs and the second and fourth row as counterclockwise arcs. Finally, the `if` statement makes the top half stroked arcs and the bottom half filled arcs.

> **Note:** This example requires a slightly larger canvas than the others on this page: `150` x `200` pixels.

JS

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

    for (let i = 0; i < 4; i++) {
      for (let j = 0; j < 3; j++) {
        ctx.beginPath();
        const x = 25 + j * 50; // x coordinate
        const y = 25 + i * 50; // y coordinate
        const radius = 20; // Arc radius
        const startAngle = 0; // Starting point on circle
        const endAngle = Math.PI + (Math.PI * j) / 2; // End point on circle
        const counterclockwise = i % 2 !== 0; // clockwise or counterclockwise

        ctx.arc(x, y, radius, startAngle, endAngle, counterclockwise);

        if (i > 1) {
          ctx.fill();
        } else {
          ctx.stroke();
        }
      }
```

```
    }
  }
}
```



Bezier and quadratic curves

The next type of paths available are Bézier curves, available in both cubic and quadratic varieties. These are generally used to draw complex organic shapes.

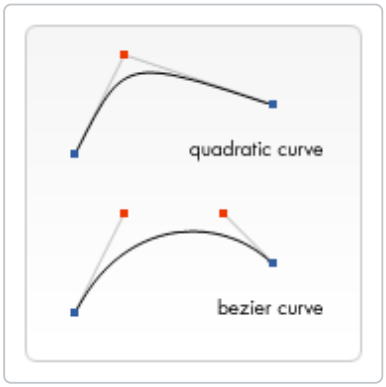
`quadraticCurveTo(cp1x, cp1y, x, y)`

Draws a quadratic Bézier curve from the current pen position to the end point specified by `x` and `y`, using the control point specified by `cp1x` and `cp1y`.

`bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)`

Draws a cubic Bézier curve from the current pen position to the end point specified by `x` and `y`, using the control points specified by (`cp1x`, `cp1y`) and (`cp2x`, `cp2y`).

The difference between these is that a quadratic Bézier curve has a start and an end point (blue dots) and just one **control point** (indicated by the red dot) while a cubic Bézier curve uses two control points.



The `x` and `y` parameters in both of these methods are the coordinates of the end point. `cp1x` and `cp1y` are the coordinates of the first control point, and `cp2x` and `cp2y` are the coordinates of the second control point.

Using quadratic and cubic Bézier curves can be quite challenging, because unlike vector drawing software like Adobe Illustrator, we don't have direct visual feedback as to what we're doing. This makes it pretty hard to draw complex

shapes. In the following example, we'll be drawing some simple organic shapes, but if you have the time and, most of all, the patience, much more complex shapes can be created.

There's nothing very difficult in these examples. In both cases we see a succession of curves being drawn which finally result in a complete shape.

QUADRATIC BEZIER CURVES

This example uses multiple quadratic Bézier curves to render a speech balloon.

JS

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

    // Quadratic curves example
    ctx.beginPath();
    ctx.moveTo(75, 25);
    ctx.quadraticCurveTo(25, 25, 25, 62.5);
    ctx.quadraticCurveTo(25, 100, 50, 100);
    ctx.quadraticCurveTo(50, 120, 30, 125);
    ctx.quadraticCurveTo(60, 120, 65, 100);
    ctx.quadraticCurveTo(125, 100, 125, 62.5);
    ctx.quadraticCurveTo(125, 25, 75, 25);
    ctx.stroke();
  }
}
```

CUBIC BEZIER CURVES

This example draws a heart using cubic Bézier curves.

JS

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

    // Cubic curves example
    ctx.beginPath();
    ctx.moveTo(75, 40);
    ctx.bezierCurveTo(75, 37, 70, 25, 50, 25);
    ctx.bezierCurveTo(20, 25, 20, 62.5, 20, 62.5);
    ctx.bezierCurveTo(20, 80, 40, 102, 75, 120);
    ctx.bezierCurveTo(110, 102, 130, 80, 130, 62.5);
    ctx.bezierCurveTo(130, 62.5, 130, 25, 100, 25);
    ctx.bezierCurveTo(85, 25, 75, 37, 75, 40);
    ctx.fill();
  }
}
```

Rectangles

In addition to the three methods we saw in Drawing rectangles, which draw rectangular shapes directly to the canvas, there's also the `rect()` method, which adds a rectangular path to a currently open path.

`rect(x, y, width, height)`

Draws a rectangle whose top-left corner is specified by (`x`, `y`) with the specified `width` and `height`.

Before this method is executed, the `moveTo()` method is automatically called with the parameters (x,y). In other words, the current pen position is automatically reset to the default coordinates.

Making combinations

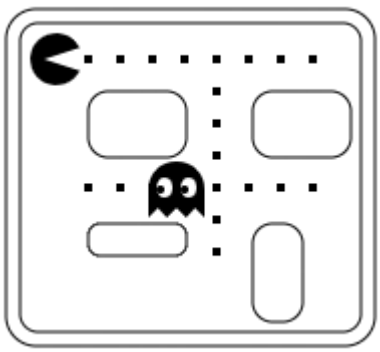
So far, each example on this page has used only one type of path function per shape. However, there's no limitation to the number or types of paths you can use to create a shape. So in this final example, let's combine all of the path functions to make a set of very famous game characters.

JS

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

    roundedRect(ctx, 12, 12, 184, 168, 15);
    roundedRect(ctx, 19, 19, 170, 154, 9);
    roundedRect(ctx, 53, 53, 49, 33, 10);
    roundedRect(ctx, 53, 119, 49, 16, 6);
    roundedRect(ctx, 135, 53, 49, 33, 10);
    roundedRect(ctx, 135, 119, 25, 49, 10);

    ctx.beginPath();
    ctx.arc(37, 37, 13, Math.PI / 7, -Math.PI / 7, false);
    ctx.lineTo(31, 37);
    ctx.fill();

    for (let i = 0; i < 8; i++) {
      ctx.fillRect(51 + i * 16, 35, 4, 4);
    }

    for (let i = 0; i < 6; i++) {
      ctx.fillRect(115, 51 + i * 16, 4, 4);
    }

    for (let i = 0; i < 8; i++) {
      ctx.fillRect(51 + i * 16, 99, 4, 4);
    }

    ctx.beginPath();
    ctx.moveTo(83, 116);
    ctx.lineTo(83, 102);
    ctx.bezierCurveTo(83, 94, 89, 88, 97, 88);
    ctx.bezierCurveTo(105, 88, 111, 94, 111, 102);
    ctx.lineTo(111, 116);
    ctx.lineTo(106.333, 111.333);
    ctx.lineTo(101.666, 116);
    ctx.lineTo(97, 111.333);
    ctx.lineTo(92.333, 116);
    ctx.lineTo(87.666, 111.333);
    ctx.lineTo(83, 116);
    ctx.fill();

    ctx.fillStyle = "white";
    ctx.beginPath();
    ctx.moveTo(91, 96);
    ctx.bezierCurveTo(88, 96, 87, 99, 87, 101);
    ctx.bezierCurveTo(87, 103, 88, 106, 91, 106);
    ctx.bezierCurveTo(94, 106, 95, 103, 95, 101);
```

```
    ctx.bezierCurveTo(95, 99, 94, 96, 91, 96);
    ctx.moveTo(103, 96);
    ctx.bezierCurveTo(100, 96, 99, 99, 99, 101);
    ctx.bezierCurveTo(99, 103, 100, 106, 103, 106);
    ctx.bezierCurveTo(106, 106, 107, 103, 107, 101);
    ctx.bezierCurveTo(107, 99, 106, 96, 103, 96);
    ctx.fill();

    ctx.fillStyle = "black";
    ctx.beginPath();
    ctx.arc(101, 102, 2, 0, Math.PI * 2, true);
    ctx.fill();

    ctx.beginPath();
    ctx.arc(89, 102, 2, 0, Math.PI * 2, true);
    ctx.fill();
  }
}

// A utility function to draw a rectangle with rounded corners.

function roundedRect(ctx, x, y, width, height, radius) {
  ctx.beginPath();
  ctx.moveTo(x, y + radius);
  ctx.arcTo(x, y + height, x + radius, y + height, radius);
  ctx.arcTo(x + width, y + height, x + width, y + height - radius, radius);
  ctx.arcTo(x + width, y, x + width - radius, y, radius);
  ctx.arcTo(x, y, x, y + radius, radius);
  ctx.stroke();
}
```

The resulting image looks like this:



We won't go over this in detail, since it's actually surprisingly simple. The most important things to note are the use of the `fillStyle` property on the drawing context, and the use of a utility function (in this case `roundedRect()`). Using utility functions for bits of drawing you do often can be very helpful and reduce the amount of code you need, as well as its complexity.

We'll take another look at `fillStyle`, in more detail, later in this tutorial. Here, all we're doing is using it to change the fill color for paths from the default color of black to white, and then back again.

Shapes with holes

To draw a shape with a hole in it, we need to draw the hole in different clock directions as we draw the outer shape. We either draw the outer shape clockwise and the inner shape anticlockwise or the outer shape anticlockwise and the inner shape clockwise.

JS

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

    ctx.beginPath();

    // Outer shape clockwise ↻
    ctx.moveTo(0, 0);
    ctx.lineTo(150, 0);
    ctx.lineTo(75, 129.9);

    // Inner shape anticlockwise ↺
    ctx.moveTo(75, 20);
    ctx.lineTo(50, 60);
    ctx.lineTo(100, 60);

    ctx.fill();
  }
}
```

In the example above, the outer triangle goes clockwise (move to the top-left corner, then draw a line to the top-right corner, and finish at the bottom) and the inner triangle goes anticlockwise (move to the top, then line to the bottom-left corner, and finish at the bottom-right).

Path2D objects

As we have seen in the last example, there can be a series of paths and drawing commands to draw objects onto your canvas. To simplify the code and to improve performance, the `Path2D` object, available in recent versions of browsers, lets you cache or record these drawing commands. You are able to play back your paths quickly. Let's see how we can construct a `Path2D` object:

`Path2D()`

The **`Path2D()`** constructor returns a newly instantiated `Path2D` object, optionally with another path as an argument (creates a copy), or optionally with a string consisting of SVG path data.

JS

```
new Path2D(); // empty path object
new Path2D(path); // copy from another Path2D object
new Path2D(d); // path from SVG path data
```

All path methods like `moveTo`, `rect`, `arc` or `quadraticCurveTo`, etc., which we got to know above, are available on `Path2D` objects.

The `Path2D` API also adds a way to combine paths using the `addPath` method. This can be useful when you want to build objects from several components, for example.

`Path2D.addPath(path [, transform])`

Adds a path to the current path with an optional transformation matrix.

Path2D example

In this example, we are creating a rectangle and a circle. Both are stored as a `Path2D` object, so that they are available for later usage. With the new `Path2D` API, several methods got updated to optionally accept a `Path2D` object to use instead of the current path. Here, `stroke` and `fill` are used with a path argument to draw both objects onto the canvas, for example.

JS

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

    const rectangle = new Path2D();
    rectangle.rect(10, 10, 50, 50);

    const circle = new Path2D();
    circle.arc(100, 35, 25, 0, 2 * Math.PI);

    ctx.stroke(rectangle);
    ctx.fill(circle);
  }
}
```

Using SVG paths

Another powerful feature of the new canvas `Path2D` API is using SVG path data to initialize paths on your canvas. This might allow you to pass around path data and re-use them in both, SVG and canvas.

The path will move to point (`M10 10`) and then move horizontally 80 points to the right (`h 80`), then 80 points down (`v 80`), then 80 points to the left (`h -80`), and then back to the start (`z`). You can see this example on the `Path2D` constructor page.

JS

```
const p = new Path2D("M10 10 h 80 v 80 h -80 Z");
```

← Previous Next →

Applying styles and colors

← Previous Next →

In the chapter about drawing shapes, we used only the default line and fill styles. Here we will explore the canvas options we have at our disposal to make our drawings a little more attractive. You will learn how to add different colors, line styles, gradients, patterns and shadows to your drawings.

> **Note:** Canvas content is not accessible to screen readers. If the canvas is purely decorative, include `role="presentation"` on the `<canvas>` opening tag. Otherwise, include descriptive text as the value of the `aria-label` attribute directly on the canvas element itself or include fallback content placed within the opening and closing canvas tag. Canvas content is not part of the DOM, but nested fallback content is.

Colors

Up until now we have only seen methods of the drawing context. If we want to apply colors to a shape, there are two important properties we can use: `fillStyle` and `strokeStyle`.

`fillStyle = color`

Sets the style used when filling shapes.

`strokeStyle = color`

Sets the style for shapes' outlines.

`color` is a string representing a CSS `<color>`, a gradient object, or a pattern object. We'll look at gradient and pattern objects later. By default, the stroke and fill color are set to black (CSS color value `#000000`).

> **Note:** When you set the `strokeStyle` and/or `fillStyle` property, the new value becomes the default for all shapes being drawn from then on. For every shape you want in a different color, you will need to reassign the `fillStyle` or `strokeStyle` property.

The valid strings you can enter should, according to the specification, be CSS `<color>` values. Each of the following examples describe the same color.

JS

```
// these all set the fillStyle to 'orange'

ctx.fillStyle = "orange";
ctx.fillStyle = "#FFA500";
```

```
ctx.fillStyle = "rgb(255 165 0)";
ctx.fillStyle = "rgb(255 165 0 / 100%)";
```

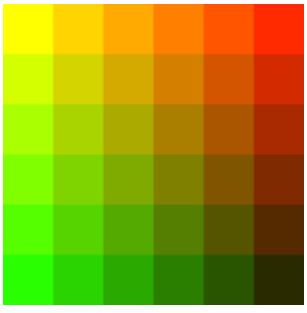
A `fillStyle` example

In this example, we once again use two `for` loops to draw a grid of rectangles, each in a different color. The resulting image should look something like the screenshot. There is nothing too spectacular happening here. We use the two variables `i` and `j` to generate a unique RGB color for each square, and only modify the red and green values. The blue channel has a fixed value. By modifying the channels, you can generate all kinds of palettes. By increasing the steps, you can achieve something that looks like the color palettes Photoshop uses.

JS

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  for (let i = 0; i < 6; i++) {
    for (let j = 0; j < 6; j++) {
      ctx.fillStyle = `rgb(${Math.floor(255 - 42.5 * i)} ${Math.floor(
        255 - 42.5 * j,
      )} 0)`;
      ctx.fillRect(j * 25, i * 25, 25, 25);
    }
  }
}
```

The result looks like this:



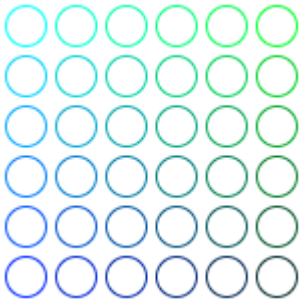
A `strokeStyle` example

This example is similar to the one above, but uses the `strokeStyle` property to change the colors of the shapes' outlines. We use the `arc()` method to draw circles instead of squares.

JS

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  for (let i = 0; i < 6; i++) {
    for (let j = 0; j < 6; j++) {
      ctx.strokeStyle = `rgb(0 ${Math.floor(255 - 42.5 * i)} ${Math.floor(
        255 - 42.5 * j,
      )})`;
      ctx.beginPath();
      ctx.arc(12.5 + j * 25, 12.5 + i * 25, 10, 0, 2 * Math.PI, true);
      ctx.stroke();
    }
  }
}
```

The result looks like this:



Transparency

In addition to drawing opaque shapes to the canvas, we can also draw semi-transparent (or translucent) shapes. This is done by either setting the `globalAlpha` property or by assigning a semi-transparent color to the stroke and/or fill style.

`globalAlpha = transparencyValue`

Applies the specified transparency value to all future shapes drawn on the canvas. The value must be between `0.0` (fully transparent) to `1.0` (fully opaque). This value is `1.0` (fully opaque) by default.

The `globalAlpha` property can be useful if you want to draw a lot of shapes on the canvas with similar transparency, but otherwise it's generally more useful to set the transparency on individual shapes when setting their colors.

Because the `strokeStyle` and `fillStyle` properties accept CSS rgb color values, we can use the following notation to assign a transparent color to them.

JS

```
// Assigning transparent colors to stroke and fill style

ctx.strokeStyle = "rgb(255 0 0 / 50%)";
ctx.fillStyle = "rgb(255 0 0 / 50%)";
```

The `rgb()` function has an optional extra parameter. The last parameter sets the transparency value of this particular color. The valid range is specified as a percentage between `0%` (fully transparent) and `100%` (fully opaque) or as a number between `0.0` (equivalent to `0%`) and `1.0` (equivalent to `100%`).

A `globalAlpha` example

In this example, we'll draw a background of four different colored squares. On top of these, we'll draw a set of semi-transparent circles. The `globalAlpha` property is set at `0.2` which will be used for all shapes from that point on. Every step in the `for` loop draws a set of circles with an increasing radius. The final result is a radial gradient. By overlaying ever more circles on top of each other, we effectively reduce the transparency of the circles that have already been drawn. By increasing the step count and in effect drawing more circles, the background would completely disappear from the center of the image.

JS

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  // draw background
  ctx.fillStyle = "#ffdd00";
  ctx.fillRect(0, 0, 75, 75);
  ctx.fillStyle = "#66cc00";
  ctx.fillRect(75, 0, 75, 75);
  ctx.fillStyle = "#0099ff";
  ctx.fillRect(0, 75, 75, 75);
  ctx.fillStyle = "#ff3300";
  ctx.fillRect(75, 75, 75, 75);
  ctx.fillStyle = "white";

  // set transparency value
  ctx.globalAlpha = 0.2;

  // Draw semi transparent circles
  for (let i = 0; i < 7; i++) {
    ctx.beginPath();
    ctx.arc(75, 75, 10 + 10 * i, 0, Math.PI * 2, true);
    ctx.fill();
  }
}
```

An example using `rgb()` with alpha transparency

In this second example, we do something similar to the one above, but instead of drawing circles on top of each other, I've drawn small rectangles with increasing opacity. Using `rgb()` gives you a little more control and flexibility because we can set the fill and stroke style individually.

JS

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

  // Draw background
  ctx.fillStyle = "rgb(255 221 0)";
  ctx.fillRect(0, 0, 150, 37.5);
  ctx.fillStyle = "rgb(102 204 0)";
  ctx.fillRect(0, 37.5, 150, 37.5);
  ctx.fillStyle = "rgb(0 153 255)";
  ctx.fillRect(0, 75, 150, 37.5);
  ctx.fillStyle = "rgb(255 51 0)";
  ctx.fillRect(0, 112.5, 150, 37.5);

  // Draw semi transparent rectangles
  for (let i = 0; i < 10; i++) {
    ctx.fillStyle = `rgb(255 255 255 / ${(i + 1) / 10})`;
    for (let j = 0; j < 4; j++) {
      ctx.fillRect(5 + i * 14, 5 + j * 37.5, 14, 27.5);
    }
  }
}
```

Line styles

There are several properties which allow us to style lines.

`lineWidth = value`

Sets the width of lines drawn in the future.

`lineCap = type`

Sets the appearance of the ends of lines.

`lineJoin = type`

Sets the appearance of the "corners" where lines meet.

`miterLimit = value`

Establishes a limit on the miter when two lines join at a sharp angle, to let you control how thick the junction becomes.

`getLineDash()`

Returns the current line dash pattern array containing an even number of non-negative numbers.

`setLineDash(segments)`

Sets the current line dash pattern.

`lineDashOffset = value`

Specifies where to start a dash array on a line.

You'll get a better understanding of what these do by looking at the examples below.

A `lineWidth` example

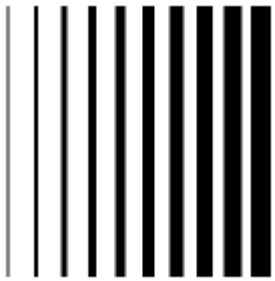
This property sets the current line thickness. Values must be positive numbers. By default this value is set to 1.0 units.

The line width is the thickness of the stroke centered on the given path. In other words, the area that's drawn extends to half the line width on either side of the path. Because canvas coordinates do not directly reference pixels, special care must be taken to obtain crisp horizontal and vertical lines.

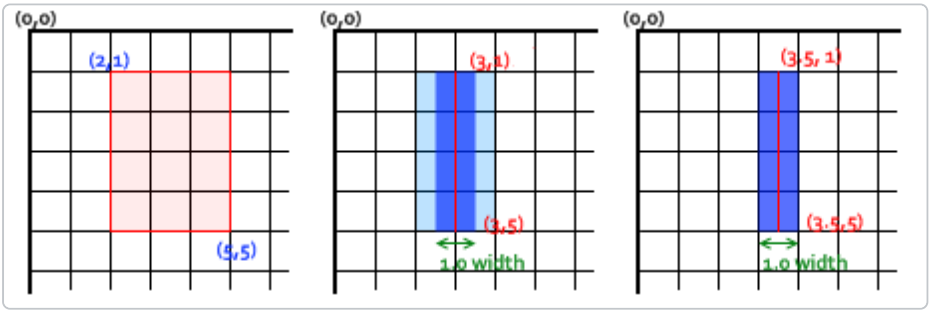
In the example below, 10 straight lines are drawn with increasing line widths. The line on the far left is 1.0 units wide. However, the leftmost and all other odd-integer-width thickness lines do not appear crisp, because of the path's positioning.

JS

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  for (let i = 0; i < 10; i++) {
    ctx.lineWidth = 1 + i;
    ctx.beginPath();
    ctx.moveTo(5 + i * 14, 5);
    ctx.lineTo(5 + i * 14, 140);
    ctx.stroke();
  }
}
```



Obtaining crisp lines requires understanding how paths are stroked. In the images below, the grid represents the canvas coordinate grid. The squares between grid lines are actual on-screen pixels. In the first grid image below, a rectangle from (2,1) to (5,5) is filled. The entire area between them (light red) falls on pixel boundaries, so the resulting filled rectangle will have crisp edges.



If you consider a path from (3,1) to (3,5) with a line thickness of `1.0`, you end up with the situation in the second image. The actual area to be filled (dark blue) only extends halfway into the pixels on either side of the path. An approximation of this has to be rendered, which means that those pixels being only partially shaded, and results in the entire area (the light blue and dark blue) being filled in with a color only half as dark as the actual stroke color. This is what happens with the `1.0` width line in the previous example code.

To fix this, you have to be very precise in your path creation. Knowing that a `1.0` width line will extend half a unit to either side of the path, creating the path from (3.5,1) to (3.5,5) results in the situation in the third image—the `1.0` line width ends up completely and precisely filling a single pixel vertical line.

> **Note:** Be aware that in our vertical line example, the Y position still referenced an integer grid line position—if it hadn't, we would see pixels with half coverage at the endpoints (but note also that this behavior depends on the current `lineCap` style whose default value is `butt`; you may want to compute consistent strokes with half-pixel coordinates for odd-width lines, by setting the `lineCap` style to `square`, so that the outer border of the stroke around the endpoint will be automatically extended to cover the whole pixel exactly).
>
> Note also that only start and final endpoints of a path are affected: if a path is closed with `closePath()`, there's no start and final endpoint; instead, all endpoints in the path are connected to their attached previous and next segment using the current setting of the `lineJoin` style, whose default value is `miter`, with the effect of automatically extending the outer borders of the connected segments to their intersection point, so that the rendered stroke will exactly cover full pixels centered at each endpoint if those connected segments are horizontal and/or vertical. See the next two sections for demonstrations of these additional line styles.

For even-width lines, each half ends up being an integer amount of pixels, so you want a path that is between pixels (that is, (3,1) to (3,5)), instead of down the middle of pixels.

While slightly painful when initially working with scalable 2D graphics, paying attention to the pixel grid and the position of paths ensures that your drawings will look correct regardless of scaling or any other transformations involved. A 1.0-width vertical line drawn at the correct position will become a crisp 2-pixel line when scaled up by 2, and will appear at the correct position.

A `lineCap` example

The `lineCap` property determines how the end points of every line are drawn. There are three possible values for this property and those are: `butt`, `round` and `square`. By default this property is set to `butt`:

`butt`

: The ends of lines are squared off at the endpoints.

`round`

: The ends of lines are rounded.

`square`

: The ends of lines are squared off by adding a box with an equal width and half the height of the line's thickness.

In this example, we'll draw three lines, each with a different value for the `lineCap` property. I also added two guides to see the exact differences between the three. Each of these lines starts and ends exactly on these guides.

The line on the left uses the default `butt` option. You'll notice that it's drawn completely flush with the guides. The second is set to use the `round` option. This adds a semicircle to the end that has a radius half the width of the line. The line on the right uses the `square` option. This adds a box with an equal width and half the height of the line thickness.

JS

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

  // Draw guides
  ctx.strokeStyle = "#0099ff";
  ctx.beginPath();
  ctx.moveTo(10, 10);
  ctx.lineTo(140, 10);
  ctx.moveTo(10, 140);
  ctx.lineTo(140, 140);
  ctx.stroke();

  // Draw lines
  ctx.strokeStyle = "black";
  ["butt", "round", "square"].forEach((lineCap, i) => {
    ctx.lineWidth = 15;
    ctx.lineCap = lineCap;
    ctx.beginPath();
    ctx.moveTo(25 + i * 50, 10);
    ctx.lineTo(25 + i * 50, 140);
    ctx.stroke();
  });
}
```

A `lineJoin` example

The `lineJoin` property determines how two connecting segments (of lines, arcs or curves) with non-zero lengths in a shape are joined together (degenerate segments with zero lengths, whose specified endpoints and control points are exactly at the same position, are skipped).

There are three possible values for this property: `round`, `bevel` and `miter`. By default this property is set to `miter`. Note that the `lineJoin` setting has no effect if the two connected segments have the same direction, because no joining area will be added in this case:

`round`

Rounds off the corners of a shape by filling an additional sector of disc centered at the common endpoint of connected segments. The radius for these rounded corners is equal to half the line width.

`bevel`

Fills an additional triangular area between the common endpoint of connected segments, and the separate outside rectangular corners of each segment.

`miter`

Connected segments are joined by extending their outside edges to connect at a single point, with the effect of filling an additional lozenge-shaped area. This setting is effected by the `miterLimit` property which is explained below.

The example below draws three different paths, demonstrating each of these three `lineJoin` property settings; the output is shown above.

JS

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  ctx.lineWidth = 10;
  ["round", "bevel", "miter"].forEach((lineJoin, i) => {
    ctx.lineJoin = lineJoin;
    ctx.beginPath();
    ctx.moveTo(-5, 5 + i * 40);
    ctx.lineTo(35, 45 + i * 40);
    ctx.lineTo(75, 5 + i * 40);
    ctx.lineTo(115, 45 + i * 40);
    ctx.lineTo(155, 5 + i * 40);
    ctx.stroke();
  });
}
```

A demo of the `miterLimit` property

As you've seen in the previous example, when joining two lines with the `miter` option, the outside edges of the two joining lines are extended up to the point where they meet. For lines which are at large angles with each other, this point is not far from the inside connection point. However, as the angles between each line decrease, the distance (miter length) between these points increases exponentially.

The `miterLimit` property determines how far the outside connection point can be placed from the inside connection point. If two lines exceed this value, a bevel join gets drawn instead. Note that the maximum miter length is the product of the line width measured in the current coordinate system, by the value of this `miterLimit` property (whose default value is 10.0 in the HTML `<canvas>`), so the `miterLimit` can be set independently from the current display scale or any affine transforms of paths: it only influences the effectively rendered shape of line edges.

More exactly, the miter limit is the maximum allowed ratio of the extension length (in the HTML canvas, it is measured between the outside corner of the joined edges of the line and the common endpoint of connecting segments specified in the path) to half the line width. It can equivalently be defined as the maximum allowed ratio of the distance between the inside and outside points of junction of edges, to the total line width. It is then equal to the cosecant of half the minimum inner angle of connecting segments below which no miter join will be rendered, but only a bevel join:

- `miterLimit` = **max** `miterLength` / `lineWidth` = 1 / **sin** (**min** θ / 2)
- The default miter limit of 10.0 will strip all miters for sharp angles below about 11 degrees.
- A miter limit equal to $\sqrt{2} \approx 1.4142136$ (rounded up) will strip miters for all acute angles, keeping miter joins only for obtuse or right angles.
- A miter limit equal to 1.0 is valid but will disable all miters.
- Values below 1.0 are invalid for the miter limit.

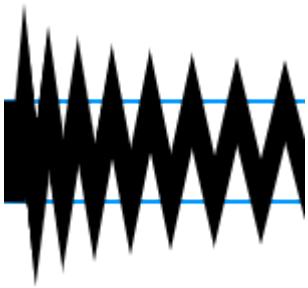
Here's a little demo in which you can set `miterLimit` dynamically and see how this effects the shapes on the canvas. The blue lines show where the start and endpoints for each of the lines in the zig-zag pattern are.

If you specify a `miterLimit` value below 4.2 in this demo, none of the visible corners will join with a miter extension, but only with a small bevel near the blue lines; with a `miterLimit` above 10, most corners in this demo should join with a miter far away from the blue lines, and whose height is decreasing between corners from left to right because they connect with growing angles; with intermediate values, the corners on the left side will only join with a bevel near the blue lines, and the corners on the right side with a miter extension (also with a decreasing height).

JS

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

  // Clear canvas
  ctx.clearRect(0, 0, 150, 150);

  // Draw guides
  ctx.strokeStyle = "#0099ff";
  ctx.lineWidth = 2;
  ctx.strokeRect(-5, 50, 160, 50);

  // Set line styles
  ctx.strokeStyle = "black";
  ctx.lineWidth = 10;

  // check input
  if (document.getElementById("miterLimit").checkValidity()) {
    ctx.miterLimit = parseFloat(document.getElementById("miterLimit").value);
  }
```

```
  // Draw lines
  ctx.beginPath();
  ctx.moveTo(0, 100);
  for (let i = 0; i < 24; i++) {
    const dy = i % 2 === 0 ? 25 : -25;
    ctx.lineTo(i ** 1.5 * 2, 75 + dy);
  }
  ctx.stroke();
  return false;
}
```



Change the `miterLimit` by entering a new value below and clicking the redraw button.

Miter limit 10 Redraw

Using line dashes

The `setLineDash` method and the `lineDashOffset` property specify the dash pattern for lines. The `setLineDash` method accepts a list of numbers that specifies distances to alternately draw a line and a gap and the `lineDashOffset` property sets an offset where to start the pattern.

In this example we are creating a marching ants effect. It is an animation technique often found in selection tools of computer graphics programs. It helps the user to distinguish the selection border from the image background by animating the border. In a later part of this tutorial, you can learn how to do this and other basic animations.

JS

```
const ctx = document.getElementById("canvas").getContext("2d");
let offset = 0;

function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.setLineDash([4, 2]);
  ctx.lineDashOffset = -offset;
  ctx.strokeRect(10, 10, 100, 100);
}

function march() {
  offset++;
  if (offset > 5) {
```

```
    offset = 0;
  }
  draw();
  setTimeout(march, 20);
}

march();
```

Gradients

Just like any normal drawing program, we can fill and stroke shapes using linear, radial and conic gradients. We create a `CanvasGradient` object by using one of the following methods. We can then assign this object to the `fillStyle` or `strokeStyle` properties.

`createLinearGradient(x1, y1, x2, y2)`

Creates a linear gradient object with a starting point of (`x1`, `y1`) and an end point of (`x2`, `y2`).

`createRadialGradient(x1, y1, r1, x2, y2, r2)`

Creates a radial gradient. The parameters represent two circles, one with its center at (`x1`, `y1`) and a radius of `r1`, and the other with its center at (`x2`, `y2`) with a radius of `r2`.

`createConicGradient(angle, x, y)`

Creates a conic gradient object with a starting angle of `angle` in radians, at the position (`x`, `y`).

For example:

JS

```
const lineargradient = ctx.createLinearGradient(0, 0, 150, 150);
const radialgradient = ctx.createRadialGradient(75, 75, 0, 75, 75, 100);
```

Once we've created a `CanvasGradient` object we can assign colors to it by using the `addColorStop()` method.

`gradient.addColorStop(position, color)`

Creates a new color stop on the `gradient` object. The `position` is a number between `0.0` and `1.0` and defines the relative position of the color in the gradient, and the `color` argument must be a string representing a CSS `<color>`,

indicating the color the gradient should reach at that offset into the transition.

You can add as many color stops to a gradient as you need. Below is a very simple linear gradient from white to black.

JS

```
const lineargradient = ctx.createLinearGradient(0, 0, 150, 150);
lineargradient.addColorStop(0, "white");
lineargradient.addColorStop(1, "black");
```

A `createLinearGradient` example

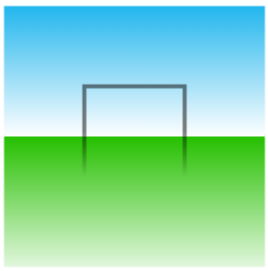
In this example, we'll create two different gradients. As you can see here, both the `strokeStyle` and `fillStyle` properties can accept a `canvasGradient` object as valid input.

JS

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

  // Create gradients
  const linGrad = ctx.createLinearGradient(0, 0, 0, 150);
  linGrad.addColorStop(0, "#00ABEB");
  linGrad.addColorStop(0.5, "white");
  linGrad.addColorStop(0.5, "#26C000");
  linGrad.addColorStop(1, "white");

  const linGrad2 = ctx.createLinearGradient(0, 50, 0, 95);
  linGrad2.addColorStop(0.5, "black");
  linGrad2.addColorStop(1, "transparent");

  // assign gradients to fill and stroke styles
  ctx.fillStyle = linGrad;
  ctx.strokeStyle = linGrad2;

  // draw shapes
  ctx.fillRect(10, 10, 130, 130);
  ctx.strokeRect(50, 50, 50, 50);
}
```

The first is a background gradient. As you can see, we assigned two colors at the same position. You do this to make very sharp color transitions—in this case from white to green. Normally, it doesn't matter in what order you define the color stops, but in this special case, it does significantly. If you keep the assignments in the order you want them to appear, this won't be a problem.

In the second gradient, we didn't assign the starting color (at position `0.0`) since it wasn't strictly necessary, because it will automatically assume the color of the next color stop. Therefore, assigning the black color at position `0.5` automatically makes the gradient, from the start to this stop, black.



A `createRadialGradient` example

In this example, we'll define four different radial gradients. Because we have control over the start and closing points of the gradient, we can achieve more complex effects than we would normally have in the "classic" radial gradients we see in, for instance, Photoshop (that is, a gradient with a single center point where the gradient expands outward in a circular shape).

JS

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

  // Create gradients
  const radGrad = ctx.createRadialGradient(45, 45, 10, 52, 50, 30);
  radGrad.addColorStop(0, "#A7D30C");
  radGrad.addColorStop(0.9, "#019F62");
  radGrad.addColorStop(1, "transparent");

  const radGrad2 = ctx.createRadialGradient(105, 105, 20, 112, 120, 50);
  radGrad2.addColorStop(0, "#FF5F98");
  radGrad2.addColorStop(0.75, "#FF0188");
  radGrad2.addColorStop(1, "transparent");

  const radGrad3 = ctx.createRadialGradient(95, 15, 15, 102, 20, 40);
  radGrad3.addColorStop(0, "#00C9FF");
  radGrad3.addColorStop(0.8, "#00B5E2");
  radGrad3.addColorStop(1, "transparent");

  const radGrad4 = ctx.createRadialGradient(0, 150, 50, 0, 140, 90);
  radGrad4.addColorStop(0, "#F4F201");
  radGrad4.addColorStop(0.8, "#E4C700");
  radGrad4.addColorStop(1, "transparent");

  // draw shapes
  ctx.fillStyle = radGrad4;
  ctx.fillRect(0, 0, 150, 150);
  ctx.fillStyle = radGrad3;
  ctx.fillRect(0, 0, 150, 150);
  ctx.fillStyle = radGrad2;
  ctx.fillRect(0, 0, 150, 150);
  ctx.fillStyle = radGrad;
  ctx.fillRect(0, 0, 150, 150);
}
```

In this case, we've offset the starting point slightly from the end point to achieve a spherical 3D effect. It's best to try to avoid letting the inside and outside circles overlap because this results in strange effects which are hard to predict.

The last color stop in each of the four gradients uses a fully transparent color. If you want to have a nice transition from this to the previous color stop, both colors should be equal. This isn't very obvious from the code because it uses two different CSS color methods as a demonstration, but in the first gradient `#019F62 = rgb(1 159 98 / 100%)`.

A `createConicGradient` example

In this example, we'll define two different conic gradients. A conic gradient differs from a radial gradient as, instead of creating circles, it circles around a point.

JS

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

  // Create gradients
  const conicGrad1 = ctx.createConicGradient(2, 62, 75);
  conicGrad1.addColorStop(0, "#A7D30C");
  conicGrad1.addColorStop(1, "white");

  const conicGrad2 = ctx.createConicGradient(0, 187, 75);
  // we multiply our values by Math.PI/180 to convert degrees to radians
  conicGrad2.addColorStop(0, "black");
  conicGrad2.addColorStop(0.25, "black");
  conicGrad2.addColorStop(0.25, "white");
  conicGrad2.addColorStop(0.5, "white");
  conicGrad2.addColorStop(0.5, "black");
  conicGrad2.addColorStop(0.75, "black");
  conicGrad2.addColorStop(0.75, "white");
  conicGrad2.addColorStop(1, "white");

  // draw shapes
  ctx.fillStyle = conicGrad1;
  ctx.fillRect(12, 25, 100, 100);
  ctx.fillStyle = conicGrad2;
  ctx.fillRect(137, 25, 100, 100);
}
```

The first gradient is positioned in the center of the first rectangle and moves a green color stop at the start, to a white one at the end. The angle starts at 2 radians, which is noticeable because of the beginning/end line pointing south east.

The second gradient is also positioned at the center of it's second rectangle. This one has multiple color stops, alternating from black to white at each quarter of the rotation. This gives us the checkered effect.

Patterns

In one of the examples on the previous page, we used a series of loops to create a pattern of images. There is, however, a much simpler method: the `createPattern()` method.

`createPattern(image, type)`

Creates and returns a new canvas pattern object. `image` is the source of the image (that is, an `HTMLImageElement`, a `SVGImageElement`, another `HTMLCanvasElement` or a `OffscreenCanvas`, an `HTMLVideoElement` or a `VideoFrame`, or an `ImageBitmap`). `type` is a string indicating how to use the image.

The type specifies how to use the image in order to create the pattern, and must be one of the following string values:

`repeat`

Tiles the image in both vertical and horizontal directions.

`repeat-x`

Tiles the image horizontally but not vertically.

`repeat-y`

Tiles the image vertically but not horizontally.

`no-repeat`

Doesn't tile the image. It's used only once.

We use this method to create a `CanvasPattern` object which is very similar to the gradient methods we've seen above. Once we've created a pattern, we can assign it to the `fillStyle` or `strokeStyle` properties. For example:

JS

```
const img = new Image();
img.src = "some-image.png";
const pattern = ctx.createPattern(img, "repeat");
```

> **Note:** Like with the `drawImage()` method, you must make sure the image you use is loaded before calling this method or the pattern may be drawn incorrectly.

A `createPattern` example

In this last example, we'll create a pattern to assign to the `fillStyle` property. The only thing worth noting is the use of the image's `onload` handler. This is to make sure the image is loaded before it is assigned to the pattern.

JS

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

  // create new image object to use as pattern
  const img = new Image();
  img.src = "canvas_create_pattern.png";
  img.onload = () => {
    // create pattern
    const pattern = ctx.createPattern(img, "repeat");
    ctx.fillStyle = pattern;
    ctx.fillRect(0, 0, 150, 150);
  };
}
```

Shadows

Using shadows involves just four properties:

`shadowOffsetX = float`

Indicates the horizontal distance the shadow should extend from the object. This value isn't affected by the transformation matrix. The default is `0`.

`shadowOffsetY = float`

Indicates the vertical distance the shadow should extend from the object. This value isn't affected by the transformation matrix. The default is `0`.

`shadowBlur = float`

Indicates the size of the blurring effect; this value doesn't correspond to a number of pixels and is not affected by the current transformation matrix. The default value is `0`.

`shadowColor = color`

A standard CSS color value indicating the color of the shadow effect; by default, it is fully-transparent black.

The properties `shadowOffsetX` and `shadowOffsetY` indicate how far the shadow should extend from the object in the X and Y directions; these values aren't affected by the current transformation matrix. Use negative values to cause the shadow to extend up or to the left, and positive values to cause the shadow to extend down or to the right. These are both `0` by default.

The `shadowBlur` property indicates the size of the blurring effect; this value doesn't correspond to a number of pixels and is not affected by the current transformation matrix. The default value is `0`.

The `shadowColor` property is a standard CSS color value indicating the color of the shadow effect; by default, it is fully-transparent black.

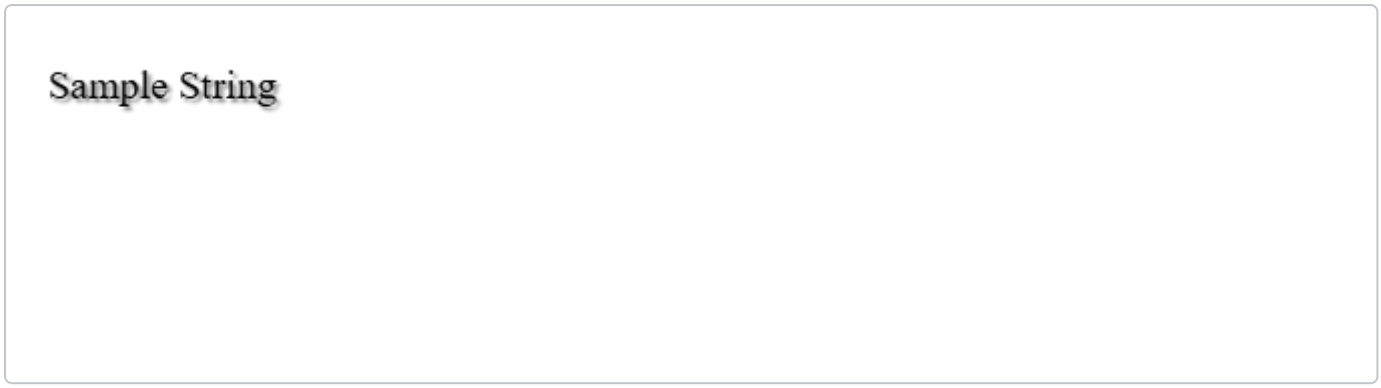
> **Note:** Shadows are only drawn for `source-over` compositing operations.

A shadowed text example

This example draws a text string with a shadowing effect.

JS

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

  ctx.shadowOffsetX = 2;
  ctx.shadowOffsetY = 2;
  ctx.shadowBlur = 2;
  ctx.shadowColor = "rgb(0 0 0 / 50%)";

  ctx.font = "20px Times New Roman";
  ctx.fillStyle = "Black";
  ctx.fillText("Sample String", 5, 30);
}
```



We will look at the `font` property and `fillText` method in the next chapter about drawing text.

Canvas fill rules

When using `fill` (or `clip` and `isPointInPath`) you can optionally provide a fill rule algorithm by which to determine if a point is inside or outside a path and thus if it gets filled or not. This is useful when a path intersects itself or is nested.

Two values are possible:

`nonzero`

The non-zero winding rule, which is the default rule.

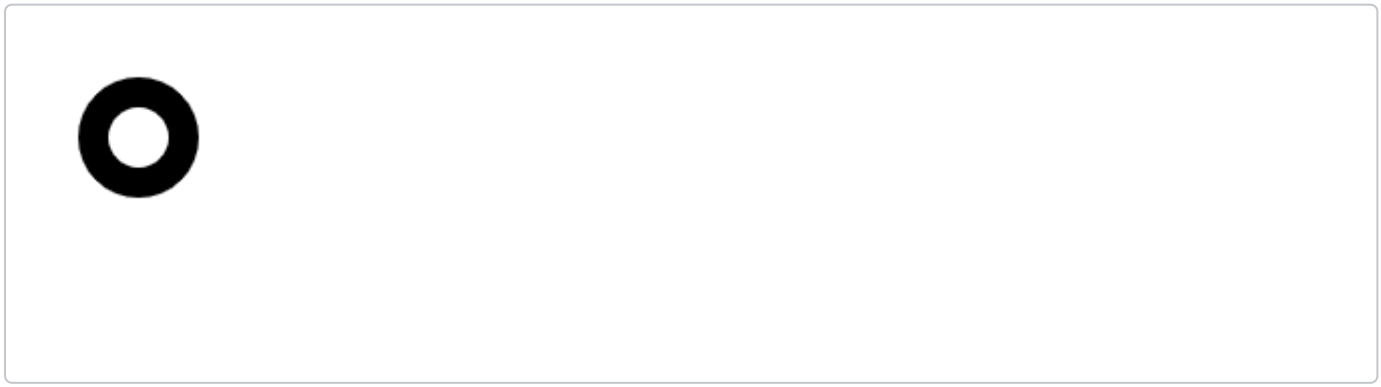
`evenodd`

The even-odd winding rule.

In this example we are using the `evenodd` rule.

JS

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  ctx.beginPath();
  ctx.arc(50, 50, 30, 0, Math.PI * 2, true);
  ctx.arc(50, 50, 15, 0, Math.PI * 2, true);
  ctx.fill("evenodd");
}
```



Drawing text

← Previous Next →

After having seen how to apply styles and colors in the previous chapter, we will now have a look at how to draw text onto the canvas.

Drawing text

The canvas rendering context provides two methods to render text:

`fillText(text, x, y [, maxWidth])`

Fills a given text at the given (x,y) position. Optionally with a maximum width to draw.

`strokeText(text, x, y [, maxWidth])`

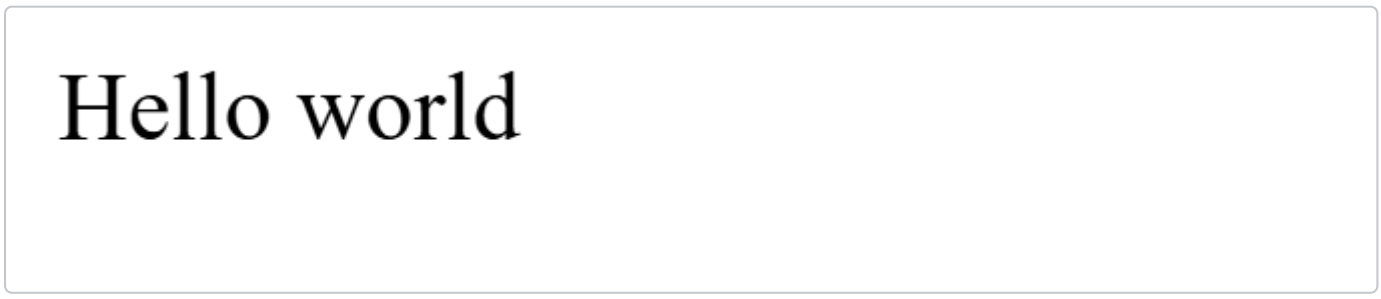
Strokes a given text at the given (x,y) position. Optionally with a maximum width to draw.

A `fillText` example

The text is filled using the current `fillStyle`.

JS

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  ctx.font = "48px serif";
  ctx.fillText("Hello world", 10, 50);
}
```



A `strokeText` example

The text is filled using the current `strokeStyle`.

JS

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  ctx.font = "48px serif";
  ctx.strokeText("Hello world", 10, 50);
}
```

Styling text

In the examples above we are already making use of the `font` property to make the text a bit larger than the default size. There are some more properties which let you adjust the way the text gets displayed on the canvas:

`font = value`

The current text style being used when drawing text. This string uses the same syntax as the CSS `font` property. The default font is 10px sans-serif.

`textAlign = value`

Text alignment setting. Possible values: `start`, `end`, `left`, `right` or `center`. The default value is `start`.

`textBaseline = value`

Baseline alignment setting. Possible values: `top`, `hanging`, `middle`, `alphabetic`, `ideographic`, `bottom`. The default value is `alphabetic`.

`direction = value`

Directionality. Possible values: `ltr`, `rtl`, `inherit`. The default value is `inherit`.

These properties might be familiar to you, if you have worked with CSS before.

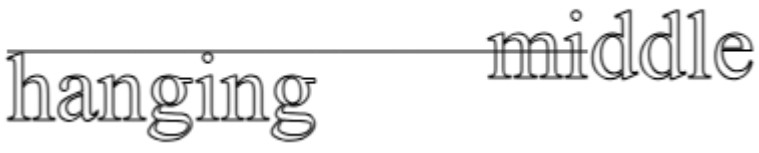
The following diagram from the HTML spec demonstrates the various baselines supported by the `textBaseline` property.

A `textBaseline` example

This example demonstrates the various `textBaseline` property values. See the `CanvasRenderingContext2D.textBaseline` page for more information and detailed examples.

JS

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  ctx.font = "48px serif";

  ctx.textBaseline = "hanging";
  ctx.strokeText("hanging", 10, 50);

  ctx.textBaseline = "middle";
  ctx.strokeText("middle", 250, 50);

  ctx.beginPath();
  ctx.moveTo(10, 50);
  ctx.lineTo(300, 50);
  ctx.stroke();
}
```



Advanced text measurements

In the case you need to obtain more details about the text, the following method allows you to measure it.

`measureText()`

Returns a `TextMetrics` object containing the width, in pixels, that the specified text will be when drawn in the current text style.

The following code snippet shows how you can measure a text and get its width.

JS

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  const text = ctx.measureText("foo"); // TextMetrics object
  text.width; // 16;
}
```

Accessibility concerns

The `<canvas>` element is just a bitmap and does not provide information about any drawn objects. Text written on canvas can cause legibility issues with users relying on screen magnification. The pixels within a canvas element do not scale and can become blurry with magnification. This is because they are not a vector but letter-shaped collection of pixels. When zooming in on it, the pixels become bigger.

Canvas content is not exposed to accessibility tools like semantic HTML is. In general, you should avoid using canvas in an accessible website or app. An alternative is to use HTML elements or SVG instead of canvas.

← Previous Next →

Using images

← Previous Next →

Until now we have created our own shapes and applied styles to them. One of the more exciting features of `<canvas>` is the ability to use images. These can be used to do dynamic photo compositing or as backdrops of graphs, for sprites in games, and so forth. External images can be used in any format supported by the browser, such as PNG, GIF, or JPEG. You can even use the image produced by other canvas elements on the same page as the source!

Importing images into a canvas is basically a two step process:

1. Get a reference to an `HTMLImageElement` object or to another canvas element as a source. It is also possible to use images by providing a URL.
2. Draw the image on the canvas using the `drawImage()` function.

Let's take a look at how to do this.

Getting images to draw

The canvas API is able to use any of the following data types as an image source:

`HTMLImageElement`

These are images created using the `Image()` constructor, as well as any `<img>` element.

`SVGImageElement`

These are images embedded using the `<image>` element.

`HTMLVideoElement`

Using an HTML `<video>` element as your image source grabs the current frame from the video and uses it as an image.

`HTMLCanvasElement`

You can use another `<canvas>` element as your image source.

`ImageBitmap`

A bitmap image, eventually cropped. Such type are used to extract part of an image, a *sprite*, from a larger image

`OffscreenCanvas`

A special kind of `<canvas>` that is not displayed and is prepared without being displayed. Using such an image source allows to switch to it without the composition of the content to be visible to the user.

`VideoFrame`

An image representing one single frame of a video.

There are several ways to get images for use on a canvas.

Using images from the same page

We can obtain a reference to images on the same page as the canvas by using one of:

- The `document.images` collection
- The `document.getElementsByTagName()` method
- If you know the ID of the specific image you wish to use, you can use `document.getElementById()` to retrieve that specific image

Using images from other domains

Using the `crossorigin` attribute of an `<img>` element (reflected by the `HTMLImageElement.crossOrigin` property), you can request permission to load an image from another domain for use in your call to `drawImage()`. If the hosting domain permits cross-domain access to the image, the image can be used in your canvas without tainting it; otherwise using the image will taint the canvas.

Using other canvas elements

Just as with normal images, we access other canvas elements using either the `document.getElementsByTagName()` or `document.getElementById()` method. Be sure you've drawn something to the source canvas before using it in your target canvas.

One of the more practical uses of this would be to use a second canvas element as a thumbnail view of the other larger canvas.

Creating images from scratch

Another option is to create new `HTMLImageElement` objects in our script. To do this, we have the convenience of an `Image()` constructor:

JS

```
const img = new Image(); // Create new img element
img.src = "myImage.png"; // Set source path
```

When this script gets executed, the image starts loading, but if you try to call `drawImage()` before the image has finished loading, it won't do anything. Older browsers may even throw an exception, so you need to be sure to use the load event so you don't draw the image to the canvas before it's ready:

JS

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

```
img.addEventListener("load", () => {
  ctx.drawImage(img, 0, 0);
});

img.src = "myImage.png";
```

If you're using one external image, this can be a good approach, but once you want to use many images or lazy-load resources, you probably need to wait for all the files to be available before drawing to the canvas. The examples below that deal with multiple images use an async function and Promise.all to wait for all images to load before calling `drawImage()`:

JS

```
async function draw() {
  // Wait for all images to be loaded:
  await Promise.all(
    Array.from(document.images).map(
      (image) =>
        new Promise((resolve) => image.addEventListener("load", resolve)),
    ),
  );

  const ctx = document.getElementById("canvas").getContext("2d");
  // call drawImage() as usual
}
draw();
```

Embedding an image via data: URL

Another possible way to include images is via the data: URL. Data URLs allow you to completely define an image as a Base64 encoded string of characters directly in your code.

JS

```
const img = new Image(); // Create new img element
img.src =

"
rIXiGBYAOw==";
```

One advantage of data URLs is that the resulting image is available immediately without another round trip to the server. Another potential advantage is that it is also possible to encapsulate in one file all of your CSS, JavaScript, HTML, and images, making it more portable to other locations.

Some disadvantages of this method are that your image is not cached, and for larger images the encoded URL can become quite long.

Using frames from a video

You can also use frames from a video being presented by a `<video>` element (even if the video is not visible). For example, if you have a `<video>` element with the ID "myVideo", you can do this:

JS

```
function getMyVideo() {
  const canvas = document.getElementById("canvas");
  if (canvas.getContext) {
    const ctx = canvas.getContext("2d");
    return document.getElementById("myVideo");
  }
}
```

This returns the `HTMLVideoElement` object for the video, which, as covered earlier, can be used as an image source for the canvas.

Drawing images

Once we have a reference to our source image object we can use the `drawImage()` method to render it to the canvas. As we will see later the `drawImage()` method is overloaded and has several variants. In its most basic form it looks like this:

`drawImage(image, x, y)`

Draws the image specified by the `image` parameter at the coordinates (`x`, `y`).

> **Note:** SVG images must specify a width and height in the root <svg> element.

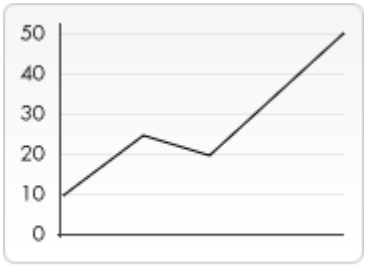
Example: A small line graph

In the following example, we will use an external image as the backdrop for a small line graph. Using backdrops can make your script considerably smaller because we can avoid the need for code to generate the background. In this example, we're only using one image, so I use the image object's `load` event handler to execute the drawing statements. The `drawImage()` method places the backdrop at the coordinate (`0`, `0`), which is the top-left corner of the canvas.

JS

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  const img = new Image();
  img.onload = () => {
    ctx.drawImage(img, 0, 0);
    ctx.beginPath();
    ctx.moveTo(30, 96);
    ctx.lineTo(70, 66);
    ctx.lineTo(103, 76);
    ctx.lineTo(170, 15);
    ctx.stroke();
  };
  img.src = "backdrop.png";
}

draw();
```

The resulting graph looks like this:



Scaling

The second variant of the `drawImage()` method adds two new parameters and lets us place scaled images on the canvas.

`drawImage(image, x, y, width, height)`

This adds the `width` and `height` parameters, which indicate the size to which to scale the image when drawing it onto the canvas.

Example: Tiling an image

In this example, we'll use an image as a wallpaper and repeat it several times on the canvas. This is done by looping and placing the scaled images at different positions. In the code below, the first `for` loop iterates over the rows. The second `for` loop iterates over the columns. The image is scaled to one third of its original size, which is 50x38 pixels.

> **Note:** Images can become blurry when scaling up or grainy if they're scaled down too much. Scaling is probably best not done if you've got some text in it which needs to remain legible.

JS

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  const img = new Image();
  img.onload = () => {
    for (let i = 0; i < 4; i++) {
      for (let j = 0; j < 3; j++) {
        ctx.drawImage(img, j * 50, i * 38, 50, 38);
      }
    }
  };
  img.src = "https://mdn.github.io/shared-assets/images/examples/rhino.jpg";
}

draw();
```

The resulting canvas looks like this:

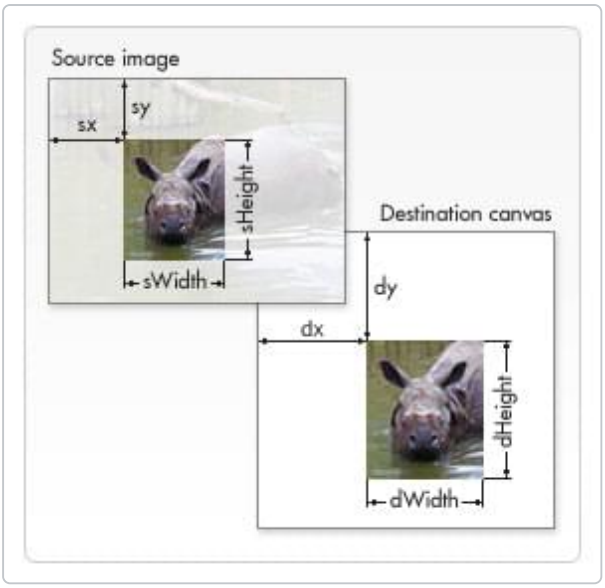
Slicing

The third and last variant of the `drawImage()` method has eight parameters in addition to the image source. It lets us cut out a section of the source image, then scale and draw it on our canvas.

`drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight)`

> Given an `image`, this function takes the area of the source image specified by the rectangle whose top-left corner is (`sx`, `sy`) and whose width and height are `sWidth` and `sHeight` and draws it into the canvas, placing it on the canvas at (`dx`, `dy`) and scaling it to the size specified by `dWidth` and `dHeight`, maintaining its aspect ratio.

To really understand what this does, it may help to look at this image:



The first four parameters define the location and size of the slice on the source image. The last four parameters define the rectangle into which to draw the image on the destination canvas.

Slicing can be a useful tool when you want to make compositions. You could have all elements in a single image file and use this method to composite a complete drawing. For instance, if you want to make a chart you could have a PNG image containing all the necessary text in a single file and depending on your data could change the scale of your chart fairly easily. Another advantage is that you don't need to load every image individually, which can improve load performance.

Example: Framing an image

In this example, we'll use the same rhino as in the previous example, but we'll slice out its head and composite it into a picture frame. The picture frame image is a 24-bit PNG which includes a drop shadow. Because 24-bit PNG images include a full 8-bit alpha channel, unlike GIF and 8-bit PNG images, it can be placed onto any background without worrying about a matte color.

HTML

```
<canvas id="canvas" width="150" height="150"></canvas>
<div class="hidden">
  <img
    id="source"
    src="https://mdn.github.io/shared-assets/images/examples/rhino.jpg"
    width="300"
    height="227" />
  <img id="frame" src="canvas_picture_frame.png" width="132" height="150" />
</div>
```

JS

```
async function draw() {
  // Wait for all images to be loaded.
  await Promise.all(
    Array.from(document.images).map(
      (image) =>
        new Promise((resolve) => image.addEventListener("load", resolve)),
    ),
  );

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

  // Draw slice
  ctx.drawImage(
    document.getElementById("source"),
    33,
    71,
    104,
    124,
    21,
    20,
    87,
    104,
  );

  // Draw frame
  ctx.drawImage(document.getElementById("frame"), 0, 0);
}

draw();
```

We took a different approach to loading the images this time. Instead of loading them by creating new `HTMLImageElement` objects, we included them as `<img>` tags in our HTML source and retrieved the images from those when drawing to the canvas. The images are hidden from page by setting the CSS property `display` to `none` for those images.

Each `<img>` is assigned an ID attribute, so we have one for a `source` and one for the `frame`, which makes them easy to select using `document.getElementById()`. We're using Promise.all to wait for all images to load before calling `drawImage()`. `drawImage()` slices the rhino out of the first image and scales it onto the canvas. Lastly, we draw the picture frame using a second `drawImage()` call.

Art gallery example

In the final example of this chapter, we'll build a little art gallery. The gallery consists of a table containing several images. When the page is loaded, a `<canvas>` element is inserted for each image and a frame is drawn around it.

In this case, every image has a fixed width and height, as does the frame that's drawn around them. You could enhance the script so that it uses the image's width and height to make the frame fit perfectly around it.

In the code below, we're using Promise.all to wait for all images to load before drawing any images to the canvas. We loop through the `document.images` container and add new canvas elements for each one. One other thing to note is the use of the `Node.insertBefore` method. `insertBefore()` is a method of the parent node (a table cell) of the element (the image) before which we want to insert our new node (the canvas element).

HTML

```
<table>
  <tr>
    <td><img src="gallery_1.jpg" /></td>
    <td><img src="gallery_2.jpg" /></td>
    <td><img src="gallery_3.jpg" /></td>
    <td><img src="gallery_4.jpg" /></td>
  </tr>
  <tr>
    <td><img src="gallery_5.jpg" /></td>
    <td><img src="gallery_6.jpg" /></td>
    <td><img src="gallery_7.jpg" /></td>
    <td><img src="gallery_8.jpg" /></td>
  </tr>
</table>
<img id="frame" src="canvas_picture_frame.png" width="132" height="150" />
```

And here's some CSS to make things look nice:

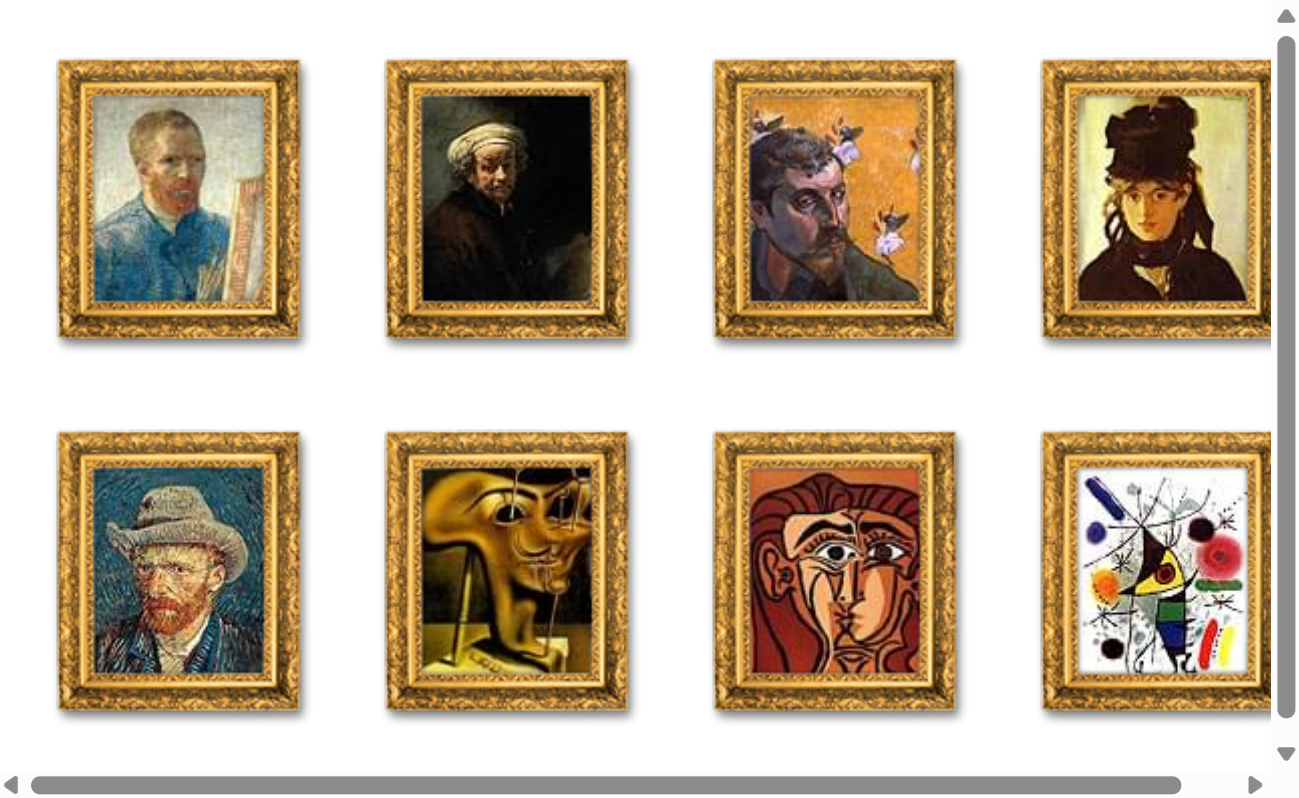
CSS

```css
body {
  background: 0 -100px repeat-x url("bg_gallery.png") #4f191a;
  margin: 10px;
}

img {
  display: none;
}

table {
  margin: 0 auto;
}

td {
  padding: 15px;
}
```

Tying it all together is the JavaScript to draw our framed images:

JS

```
async function draw() {
  // Wait for all images to be loaded.
  await Promise.all(
    Array.from(document.images).map(
      (image) =>
        new Promise((resolve) => image.addEventListener("load", resolve)),
    ),
  );

  // Loop through all images.
  for (const image of document.images) {
    // Don't add a canvas for the frame image
    if (image.getAttribute("id") !== "frame") {
      // Create canvas element
      const canvas = document.createElement("canvas");
      canvas.setAttribute("width", 132);
      canvas.setAttribute("height", 150);

      // Insert before the image
      image.parentNode.insertBefore(canvas, image);

      ctx = canvas.getContext("2d");

      // Draw image to canvas
      ctx.drawImage(image, 15, 20);

      // Add frame
      ctx.drawImage(document.getElementById("frame"), 0, 0);
    }
  }
}

draw();
```



Controlling image scaling behavior

As mentioned previously, scaling images can result in fuzzy or blocky artifacts due to the scaling process. You can use the drawing context's `imageSmoothingEnabled` property to control the use of image smoothing algorithms when scaling images within your context. By default, this is `true`, meaning images will be smoothed when scaled.

← Previous Next →

Transformations

← Previous Next →

Earlier in this tutorial we've learned about the canvas grid and the **coordinate space**. Until now, we only used the default grid and changed the size of the overall canvas for our needs. With transformations there are more powerful ways to translate the origin to a different position, rotate the grid and even scale it.

Saving and restoring state

Before we look at the transformation methods, let's look at two other methods which are indispensable once you start generating ever more complex drawings.

`save()`

Saves the entire state of the canvas.

`restore()`

Restores the most recently saved canvas state.

Canvas states are stored on a stack. Every time the `save()` method is called, the current drawing state is pushed onto the stack. A drawing state consists of

- The transformations that have been applied (i.e., `translate`, `rotate` and `scale` – see below).
- The current values of the following attributes:
  - `strokeStyle`
  - `fillStyle`
  - `globalAlpha`
  - `lineWidth`
  - `lineCap`
  - `lineJoin`
  - `miterLimit`
  - `lineDashOffset`
  - `shadowOffsetX`
  - `shadowOffsetY`
  - `shadowBlur`
  - `shadowColor`
  - `globalCompositeOperation`
  - `font`
  - `textAlign`
  - `textBaseline`
  - `direction`
  - `imageSmoothingEnabled`.

- The current clipping path, which we'll see in the next section.

You can call the `save()` method as many times as you like. Each time the `restore()` method is called, the last saved state is popped off the stack and all saved settings are restored.

A `save` and `restore` canvas state example

JS

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

  ctx.fillRect(0, 0, 150, 150); // Draw a Black rectangle with default settings
  ctx.save(); // Save the original default state

  ctx.fillStyle = "#0099ff"; // Make changes to saved settings
  ctx.fillRect(15, 15, 120, 120); // Draw a Blue rectangle with new settings
  ctx.save(); // Save the current state

  ctx.fillStyle = "white"; // Make changes to saved settings
  ctx.globalAlpha = 0.5;
  ctx.fillRect(30, 30, 90, 90); // Draw a 50%-White rectangle with newest settings

  ctx.restore(); // Restore to previous state
  ctx.fillRect(45, 45, 60, 60); // Draw a rectangle with restored Blue setting

  ctx.restore(); // Restore to original state
  ctx.fillRect(60, 60, 30, 30); // Draw a rectangle with restored Black setting
}
```

The first step is to draw a large rectangle with the default settings. Next we save this state and make changes to the fill color. We then draw the second and smaller blue rectangle and save the state. Again we change some drawing settings and draw the third semi-transparent white rectangle.

So far this is pretty similar to what we've done in previous sections. However once we call the first `restore()` statement, the top drawing state is removed from the stack, and settings are restored. If we hadn't saved the state using `save()`, we would need to change the fill color and transparency manually in order to return to the previous state. This would be easy for two properties, but if we have more than that, our code would become very long, very fast.

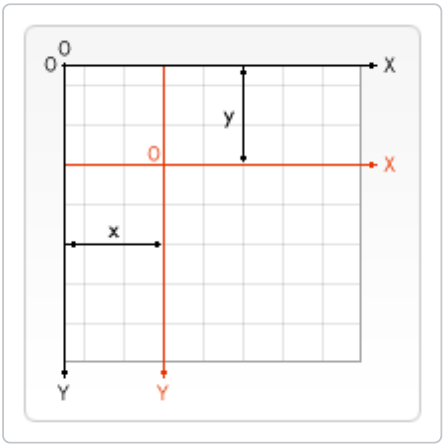
When the second `restore()` statement is called, the original state (the one we set up before the first call to `save`) is restored and the last rectangle is once again drawn in black.

Translating

The first of the transformation methods we'll look at is `translate()`. This method is used to move the canvas and its origin to a different point in the grid.

`translate(x, y)`

> Moves the canvas and its origin on the grid. `x` indicates the horizontal distance to move, and `y` indicates how far to move the grid vertically.



It's a good idea to save the canvas state before doing any transformations. In most cases, it is just easier to call the `restore` method than having to do a reverse translation to return to the original state. Also if you're translating inside a loop and don't save and restore the canvas state, you might end up missing part of your drawing, because it was drawn outside the canvas edge.

A `translate` example

This example demonstrates some of the benefits of translating the canvas origin. Without the `translate()` method, all of the rectangles would be drawn at the same position (0,0). The `translate()` method also gives us the freedom to place the rectangle anywhere on the canvas without having to manually adjust coordinates in the `fillRect()` function. This makes it a little easier to understand and use.

In the `draw()` function, we call the `fillRect()` function nine times using two `for` loops. In each loop, the canvas is translated, the rectangle is drawn, and the canvas is returned back to its original state. Note how the call to `fillRect()` uses the same coordinates each time, relying on `translate()` to adjust the drawing position.

JS

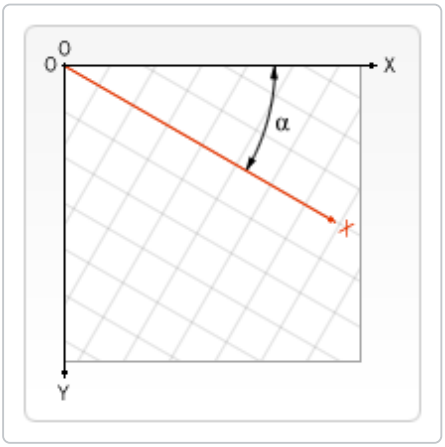
```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  for (let i = 0; i < 3; i++) {
    for (let j = 0; j < 3; j++) {
      ctx.save();
      ctx.fillStyle = `rgb(${51 * i} ${255 - 51 * i} 255)`;
      ctx.translate(10 + j * 50, 10 + i * 50);
      ctx.fillRect(0, 0, 25, 25);
      ctx.restore();
    }
  }
}
```

Rotating

The second transformation method is `rotate()`. We use it to rotate the canvas around the current origin.

`rotate(angle)`

Rotates the canvas clockwise around the current origin by the `angle` number of radians.



The rotation center point is always the canvas origin. To change the center point, we will need to move the canvas by using the `translate()` method.

A `rotate` example

In this example, we'll use the `rotate()` method to first rotate a rectangle from the canvas origin and then from the center of the rectangle itself with the help of `translate()`.

> **Note:** Angles are in radians, not degrees. To convert, we are using: `radians = (Math.PI/180)*degrees`.

JS

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

  // left rectangles, rotate from canvas origin
  ctx.save();
  // blue rect
  ctx.fillStyle = "#0095DD";
  ctx.fillRect(30, 30, 100, 100);
  ctx.rotate((Math.PI / 180) * 25);
  // grey rect
  ctx.fillStyle = "#4D4E53";
  ctx.fillRect(30, 30, 100, 100);
  ctx.restore();

  // right rectangles, rotate from rectangle center
  // draw blue rect
  ctx.fillStyle = "#0095DD";
  ctx.fillRect(150, 30, 100, 100);

  ctx.translate(200, 80); // translate to rectangle center
  // x = x + 0.5 * width
  // y = y + 0.5 * height
  ctx.rotate((Math.PI / 180) * 25); // rotate
  ctx.translate(-200, -80); // translate back

  // draw grey rect
  ctx.fillStyle = "#4D4E53";
  ctx.fillRect(150, 30, 100, 100);
}
```

To rotate the rectangle around its own center, we translate the canvas to the center of the rectangle, then rotate the canvas, then translate the canvas back to 0,0, and then draw the rectangle.

Scaling

The next transformation method is scaling. We use it to increase or decrease the units in our canvas grid. This can be used to draw scaled down or enlarged shapes and bitmaps.

`scale(x, y)`

> Scales the canvas units by x horizontally and by y vertically. Both parameters are real numbers. Values that are smaller than 1.0 reduce the unit size and values above 1.0 increase the unit size. Values of 1.0 leave the units the same size.

Using negative numbers you can do axis mirroring (for example using `translate(0,canvas.height); scale(1,-1);` you will have the well-known Cartesian coordinate system, with the origin in the bottom left corner).

By default, one unit on the canvas is exactly one pixel. If we apply, for instance, a scaling factor of 0.5, the resulting unit would become 0.5 pixels and so shapes would be drawn at half size. In a similar way setting the scaling factor to 2.0 would increase the unit size and one unit now becomes two pixels. This results in shapes being drawn twice as large.

A `scale` example

In this last example, we'll draw shapes with different scaling factors.

JS

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

  // draw a simple rectangle, but scale it.
  ctx.save();
  ctx.scale(10, 3);
  ctx.fillRect(1, 10, 10, 10);
  ctx.restore();

  // mirror horizontally
  ctx.scale(-1, 1);
  ctx.font = "48px serif";
  ctx.fillText("MDN", -135, 120);
}
```

Transforms

Finally, the following transformation methods allow modifications directly to the transformation matrix.

`transform(a, b, c, d, e, f)`

Multiplies the current transformation matrix with the matrix described by its arguments. The transformation matrix is described by:

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

If any of the arguments are `Infinity` the transformation matrix must be marked as infinite instead of the method throwing an exception.

The parameters of this function are:

`a` (`m11`)

Horizontal scaling.

`b` (`m12`)

Horizontal skewing.

`c` (`m21`)

Vertical skewing.

`d` (`m22`)

Vertical scaling.

`e` (`dx`)

Horizontal moving.

`f` (`dy`)

Vertical moving.

`setTransform(a, b, c, d, e, f)`

Resets the current transform to the identity matrix, and then invokes the `transform()` method with the same arguments. This basically undoes the current transformation, then sets the specified transform, all in one step.

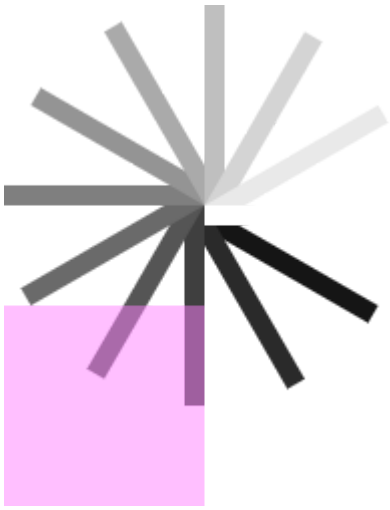
`resetTransform()`

Resets the current transform to the identity matrix. This is the same as calling: `ctx.setTransform(1, 0, 0, 1, 0, 0);`

Example for `transform` and `setTransform`

JS

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

  const sin = Math.sin(Math.PI / 6);
  const cos = Math.cos(Math.PI / 6);
  ctx.translate(100, 100);
  let c = 0;
  for (let i = 0; i <= 12; i++) {
    c = Math.floor((255 / 12) * i);
    ctx.fillStyle = `rgb(${c} ${c} ${c})`;
    ctx.fillRect(0, 0, 100, 10);
    ctx.transform(cos, sin, -sin, cos, 0, 0);
  }

  ctx.setTransform(-1, 0, 0, 1, 100, 100);
  ctx.fillStyle = "rgb(255 128 255 / 50%)";
  ctx.fillRect(0, 50, 100, 100);
}
```



← Previous Next →

Compositing and clipping

← Previous Next →

In all of our previous examples, shapes were always drawn one on top of the other. This is more than adequate for most situations, but it limits the order in which composite shapes are built. We can, however, change this behavior by setting the `globalCompositeOperation` property. In addition, the `clip` property allows us to hide unwanted parts of shapes.

`globalCompositeOperation`

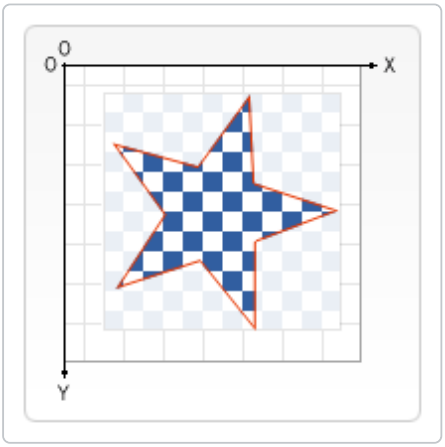
We can not only draw new shapes behind existing shapes but we can also use it to mask off certain areas, clear sections from the canvas (not limited to rectangles like the `clearRect()` method does) and more.

`globalCompositeOperation = type`

> This sets the type of compositing operation to apply when drawing new shapes, where type is a string identifying which of the twelve compositing operations to use.

Clipping paths

A clipping path is like a normal canvas shape but it acts as a mask to hide unwanted parts of shapes. This is visualized in the image below. The red star shape is our clipping path. Everything that falls outside of this path won't get drawn on the canvas.



If we compare clipping paths to the `globalCompositeOperation` property we've seen above, we see two compositing modes that achieve more or less the same effect in `source-in` and `source-atop`. The most important differences between the two are that clipping paths are never actually drawn to the canvas and the clipping path is never affected by adding new shapes. This makes clipping paths ideal for drawing multiple shapes in a restricted area.

In the chapter about drawing shapes I only mentioned the `stroke()` and `fill()` methods, but there's a third method we can use with paths, called `clip()`.

`clip()`

Turns the path currently being built into the current clipping path.

You use `clip()` instead of `closePath()` to close a path and turn it into a clipping path instead of stroking or filling the path.

By default the `<canvas>` element has a clipping path that's the exact same size as the canvas itself. In other words, no clipping occurs.

A `clip` example

In this example, we'll use a circular clipping path to restrict the drawing of a set of random stars to a particular region.

JS

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  ctx.fillRect(0, 0, 150, 150);
  ctx.translate(75, 75);

  // Create a circular clipping path
  ctx.beginPath();
  ctx.arc(0, 0, 60, 0, Math.PI * 2, true);
  ctx.clip();

  // Draw background
  const linGrad = ctx.createLinearGradient(0, -75, 0, 75);
  linGrad.addColorStop(0, "#232256");
  linGrad.addColorStop(1, "#143778");

  ctx.fillStyle = linGrad;
  ctx.fillRect(-75, -75, 150, 150);

  generateStars(ctx);
}

function generateStars(ctx) {
  for (let j = 1; j < 50; j++) {
    ctx.save();
    ctx.fillStyle = "white";
    ctx.translate(
      75 - Math.floor(Math.random() * 150),
      75 - Math.floor(Math.random() * 150),
    );
    drawStar(ctx, Math.floor(Math.random() * 4) + 2);
    ctx.restore();
  }
}

function drawStar(ctx, r) {
  ctx.save();
  ctx.beginPath();
  ctx.moveTo(r, 0);
  for (let i = 0; i < 9; i++) {
    ctx.rotate(Math.PI / 5);
    if (i % 2 === 0) {
      ctx.lineTo((r / 0.525731) * 0.200811, 0);
    } else {
      ctx.lineTo(r, 0);
    }
  }
```

```
  ctx.closePath();
  ctx.fill();
  ctx.restore();
}
```

In the first few lines of code, we draw a black rectangle the size of the canvas as a backdrop, then translate the origin to the center. Next, we create the circular clipping path by drawing an arc and calling `clip()`. Clipping paths are also part of the canvas save state. If we wanted to keep the original clipping path we could have saved the canvas state before creating the new one.

Everything that's drawn after creating the clipping path will only appear inside that path. You can see this clearly in the linear gradient that's drawn next. After this a set of 50 randomly positioned and scaled stars is drawn, using the custom `drawStar()` function. Again the stars only appear inside the defined clipping path.

Inverse clipping path

There is no such thing as an inverse clipping mask. However, we can define a mask that fills the entire canvas with a rectangle and has a hole in it for the parts that you want to skip. When drawing a shape with a hole, we need to draw the hole in the opposite direction as the outer shape. In the example below we punch a hole into the sky.

A rectangle does not have a drawing direction, but it behaves as if we drew it clockwise. By default, the arc command also goes clockwise, but we can change its direction with the last argument.

JS

```
function draw() {
  const canvas = document.getElementById("canvas");
  if (canvas.getContext) {
    const ctx = canvas.getContext("2d");
    ctx.translate(75, 75);

    // Clipping path
    ctx.beginPath();
    ctx.rect(-75, -75, 150, 150); // Outer rectangle
    ctx.arc(0, 0, 60, 0, Math.PI * 2, true); // Hole anticlockwise
    ctx.clip();

    // Draw background
```

```
    const linGrad = ctx.createLinearGradient(0, -75, 0, 75);
    linGrad.addColorStop(0, "#232256");
    linGrad.addColorStop(1, "#143778");

    ctx.fillStyle = linGrad;
    ctx.fillRect(-75, -75, 150, 150);

    generateStars(ctx);
  }
}
```

← Previous Next →

Basic animations

← Previous Next →

Since we're using JavaScript to control `<canvas>` elements, it's also very easy to make (interactive) animations. In this chapter we will take a look at how to do some basic animations.

Probably the biggest limitation is, that once a shape gets drawn, it stays that way. If we need to move it we have to redraw it and everything that was drawn before it. It takes a lot of time to redraw complex frames and the performance depends highly on the speed of the computer it's running on.

Basic animation steps

These are the steps you need to take to draw a frame:

1. **Clear the canvas** Unless the shapes you'll be drawing fill the complete canvas (for instance a backdrop image), you need to clear any shapes that have been drawn previously. The easiest way to do this is using the `clearRect()` method.
2. **Save the canvas state** If you're changing any setting (such as styles, transformations, etc.) which affect the canvas state and you want to make sure the original state is used each time a frame is drawn, you need to save that original state.
3. **Draw animated shapes** The step where you do the actual frame rendering.
4. **Restore the canvas state** If you've saved the state, restore it before drawing a new frame.

Controlling an animation

Shapes are drawn to the canvas by using the canvas methods directly or by calling custom functions. In normal circumstances, we only see these results appear on the canvas when the script finishes executing. For instance, it isn't possible to do an animation from within a `for` loop.

That means we need a way to execute our drawing functions over a period of time. There are two ways to control an animation like this.

Scheduled updates

First there's the `setInterval()`, `setTimeout()`, and `requestAnimationFrame()` functions, which can be used to call a specific function over a set period of time.

`setInterval()`

Starts repeatedly executing the function specified by `function` every `delay` milliseconds.

`setTimeout()`

Executes the function specified by `function` in `delay` milliseconds.

`requestAnimationFrame()`

Tells the browser that you wish to perform an animation and requests that the browser call a specified function to update an animation before the next repaint.

If you don't want any user interaction you can use the `setInterval()` function, which repeatedly executes the supplied code. If we wanted to make a game, we could use keyboard or mouse events to control the animation and use `setTimeout()`. By setting listeners using `addEventListener()`, we catch any user interaction and execute our animation functions.

> **Note:** In the examples below, we'll use the `Window.requestAnimationFrame()` method to control the animation. The `requestAnimationFrame` method provides a smoother and more efficient way for animating by calling the animation frame when the system is ready to paint the frame. The number of callbacks is usually 60 times per second and may be reduced to a lower rate when running in background tabs. For more information about the animation loop, especially for games, see the article Anatomy of a video game in our Game development zone.

An animated solar system

This example animates a small model of our solar system.

HTML

HTML

```
<canvas id="canvas" width="300" height="300"></canvas>
```

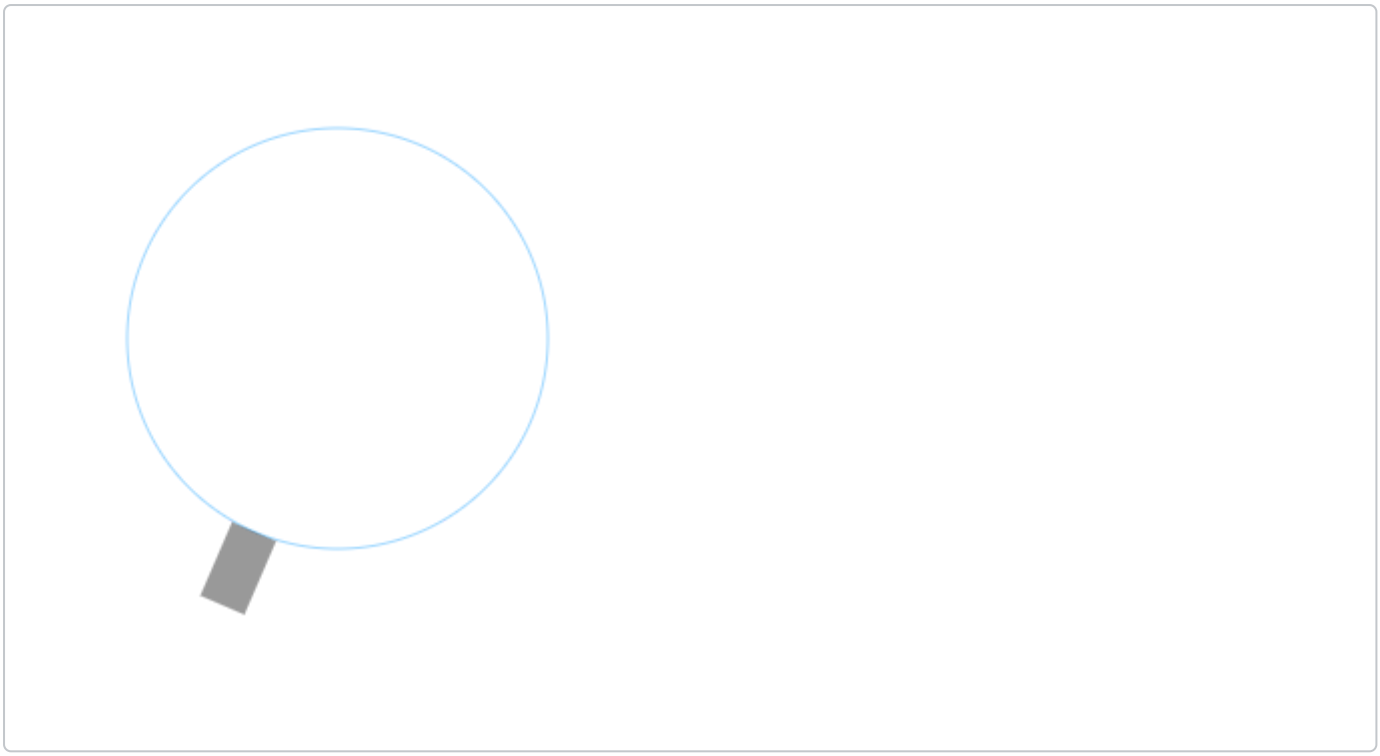
JavaScript

JS

```
const sun = new Image();
const moon = new Image();
const earth = new Image();
const ctx = document.getElementById("canvas").getContext("2d");

function init() {
  sun.src = "canvas_sun.png";
  moon.src = "canvas_moon.png";
  earth.src = "canvas_earth.png";
  window.requestAnimationFrame(draw);
}
```

```
function draw() {
  ctx.globalCompositeOperation = "destination-over";
  ctx.clearRect(0, 0, 300, 300); // clear canvas

  ctx.fillStyle = "rgb(0 0 0 / 40%)";
  ctx.strokeStyle = "rgb(0 153 255 / 40%)";
  ctx.save();
  ctx.translate(150, 150);

  // Earth
  const time = new Date();
  ctx.rotate(
    ((2 * Math.PI) / 60) * time.getSeconds() +
      ((2 * Math.PI) / 60000) * time.getMilliseconds(),
  );
  ctx.translate(105, 0);
  ctx.fillRect(0, -12, 40, 24); // Shadow
  ctx.drawImage(earth, -12, -12);

  // Moon
  ctx.save();
  ctx.rotate(
    ((2 * Math.PI) / 6) * time.getSeconds() +
      ((2 * Math.PI) / 6000) * time.getMilliseconds(),
  );
  ctx.translate(0, 28.5);
  ctx.drawImage(moon, -3.5, -3.5);
  ctx.restore();

  ctx.restore();

  ctx.beginPath();
  ctx.arc(150, 150, 105, 0, Math.PI * 2, false); // Earth orbit
  ctx.stroke();

  ctx.drawImage(sun, 0, 0, 300, 300);

  window.requestAnimationFrame(draw);
}

init();
```

Result



An animated clock

This example draws an animated clock, showing your current time.

HTML

HTML

```
<canvas id="canvas" width="150" height="150">The current time</canvas>
```

JavaScript

JS

```
function clock() {
  const now = new Date();
  const canvas = document.getElementById("canvas");
  const ctx = canvas.getContext("2d");
  ctx.save();
  ctx.clearRect(0, 0, 150, 150);
  ctx.translate(75, 75);
  ctx.scale(0.4, 0.4);
  ctx.rotate(-Math.PI / 2);
  ctx.strokeStyle = "black";
  ctx.fillStyle = "white";
```

```
  ctx.lineWidth = 8;
  ctx.lineCap = "round";

  // Hour marks
  ctx.save();
  for (let i = 0; i < 12; i++) {
    ctx.beginPath();
    ctx.rotate(Math.PI / 6);
    ctx.moveTo(100, 0);
    ctx.lineTo(120, 0);
    ctx.stroke();
  }
  ctx.restore();

  // Minute marks
  ctx.save();
  ctx.lineWidth = 5;
  for (let i = 0; i < 60; i++) {
    if (i % 5 !== 0) {
      ctx.beginPath();
      ctx.moveTo(117, 0);
      ctx.lineTo(120, 0);
      ctx.stroke();
    }
    ctx.rotate(Math.PI / 30);
  }
  ctx.restore();

  const sec = now.getSeconds();
  // To display a clock with a sweeping second hand, use:
  // const sec = now.getSeconds() + now.getMilliseconds() / 1000;
  const min = now.getMinutes();
  const hr = now.getHours() % 12;

  ctx.fillStyle = "black";

  // Write image description
  canvas.innerText = `The time is: ${hr}:${min}`;

  // Write Hours
  ctx.save();
  ctx.rotate(
    (Math.PI / 6) * hr + (Math.PI / 360) * min + (Math.PI / 21600) * sec,
  );
  ctx.lineWidth = 14;
  ctx.beginPath();
  ctx.moveTo(-20, 0);
  ctx.lineTo(80, 0);
  ctx.stroke();
```

```
  ctx.restore();

  // Write Minutes
  ctx.save();
  ctx.rotate((Math.PI / 30) * min + (Math.PI / 1800) * sec);
  ctx.lineWidth = 10;
  ctx.beginPath();
  ctx.moveTo(-28, 0);
  ctx.lineTo(112, 0);
  ctx.stroke();
  ctx.restore();

  // Write seconds
  ctx.save();
  ctx.rotate((sec * Math.PI) / 30);
  ctx.strokeStyle = "#D40000";
  ctx.fillStyle = "#D40000";
  ctx.lineWidth = 6;
  ctx.beginPath();
  ctx.moveTo(-30, 0);
  ctx.lineTo(83, 0);
  ctx.stroke();
  ctx.beginPath();
  ctx.arc(0, 0, 10, 0, Math.PI * 2, true);
  ctx.fill();
  ctx.beginPath();
  ctx.arc(95, 0, 10, 0, Math.PI * 2, true);
  ctx.stroke();
  ctx.fillStyle = "transparent";
  ctx.arc(0, 0, 3, 0, Math.PI * 2, true);
  ctx.fill();
  ctx.restore();

  ctx.beginPath();
  ctx.lineWidth = 14;
  ctx.strokeStyle = "#325FA2";
  ctx.arc(0, 0, 142, 0, Math.PI * 2, true);
  ctx.stroke();

  ctx.restore();

  window.requestAnimationFrame(clock);
}

window.requestAnimationFrame(clock);
```

Result

> **Note:** Although the clock updates only once every second, the animated image is updated at 60 frames per second (or at the display refresh rate of your web browser). To display the clock with a sweeping second hand, replace the definition of `const sec` above with the version that has been commented out.

A looping panorama

In this example, a panorama is scrolled left-to-right. We're using an image of Yosemite National Park we took from Wikipedia, but you could use any image that's larger than the canvas.

HTML

The HTML includes the `<canvas>` in which the image is scrolled. Note that the width and height specified here must match the values of the `canvasXSize` and `canvasYSize` variables in the JavaScript code.

HTML

```
<canvas id="canvas" width="800" height="200"
  >Yosemite National Park, meadow at the base of El Capitan</canvas
>
```

JavaScript

JS

```
const img = new Image();

// User Variables - customize these to change the image being scrolled, its
// direction, and the speed.
img.src = "capitan_meadows_yosemite_national_park.jpg";
const canvasXSize = 800;
const canvasYSize = 200;
```

```
const speed = 30; // lower is faster
const scale = 1.05;
const y = -4.5; // vertical offset

// Main program
const dx = 0.75;
let imgW;
let imgH;
let x = 0;
let clearX;
let clearY;
let ctx;

img.onload = () => {
  imgW = img.width * scale;
  imgH = img.height * scale;

  if (imgW > canvasXSize) {
    // Image larger than canvas
    x = canvasXSize - imgW;
  }

  // Check if image dimension is larger than canvas
  clearX = Math.max(imgW, canvasXSize);
  clearY = Math.max(imgH, canvasYSize);

  // Get canvas context
  ctx = document.getElementById("canvas").getContext("2d");

  // Set refresh rate
  return setInterval(draw, speed);
};

function draw() {
  ctx.clearRect(0, 0, clearX, clearY); // clear the canvas

  // If image is <= canvas size
  if (imgW <= canvasXSize) {
    // Reset, start from beginning
    if (x > canvasXSize) {
      x = -imgW + x;
    }

    // Draw additional image1
    if (x > 0) {
      ctx.drawImage(img, -imgW + x, y, imgW, imgH);
    }

    // Draw additional image2
```

```
    if (x - imgW > 0) {
      ctx.drawImage(img, -imgW * 2 + x, y, imgW, imgH);
    }
  } else {
    // Image is > canvas size
    // Reset, start from beginning
    if (x > canvasXSize) {
      x = canvasXSize - imgW;
    }

    // Draw additional image
    if (x > canvasXSize - imgW) {
      ctx.drawImage(img, x - imgW + 1, y, imgW, imgH);
    }
  }

  // Draw image
  ctx.drawImage(img, x, y, imgW, imgH);

  // Amount to move
  x += dx;
}
```

Result

Mouse following animation

HTML

HTML

```
<canvas id="cw"
  >Animation creating multi-colored disappearing stream of light that follow the
  cursor as it moves over the image
</canvas>
```

CSS

CSS

```
#cw {
  position: fixed;
  z-index: -1;
}

body {
  margin: 0;
  padding: 0;
  background-color: rgb(0 0 0 / 5%);
}
```

JavaScript

JS

```
const canvas = document.getElementById("cw");
const context = canvas.getContext("2d");
context.globalAlpha = 0.5;

const cursor = {
  x: innerWidth / 2,
  y: innerHeight / 2,
};

let particlesArray = [];

generateParticles(101);
setSize();
anim();

addEventListener("mousemove", (e) => {
  cursor.x = e.clientX;
  cursor.y = e.clientY;
});
```

```
addEventListener(
  "touchmove",
  (e) => {
    e.preventDefault();
    cursor.x = e.touches[0].clientX;
    cursor.y = e.touches[0].clientY;
  },
  { passive: false },
);

addEventListener("resize", () => setSize());

function generateParticles(amount) {
  for (let i = 0; i < amount; i++) {
    particlesArray[i] = new Particle(
      innerWidth / 2,
      innerHeight / 2,
      4,
      generateColor(),
      0.02,
    );
  }
}

function generateColor() {
  let hexSet = "0123456789ABCDEF";
  let finalHexString = "#";
  for (let i = 0; i < 6; i++) {
    finalHexString += hexSet[Math.ceil(Math.random() * 15)];
  }
  return finalHexString;
}

function setSize() {
  canvas.height = innerHeight;
  canvas.width = innerWidth;
}

function Particle(x, y, particleTrailWidth, strokeColor, rotateSpeed) {
  this.x = x;
  this.y = y;
  this.particleTrailWidth = particleTrailWidth;
  this.strokeColor = strokeColor;
  this.theta = Math.random() * Math.PI * 2;
  this.rotateSpeed = rotateSpeed;
  this.t = Math.random() * 150;

  this.rotate = () => {
    const ls = {
```

```
      x: this.x,
      y: this.y,
    };
    this.theta += this.rotateSpeed;
    this.x = cursor.x + Math.cos(this.theta) * this.t;
    this.y = cursor.y + Math.sin(this.theta) * this.t;
    context.beginPath();
    context.lineWidth = this.particleTrailWidth;
    context.strokeStyle = this.strokeColor;
    context.moveTo(ls.x, ls.y);
    context.lineTo(this.x, this.y);
    context.stroke();
  };
}

function anim() {
  requestAnimationFrame(anim);

  context.fillStyle = "rgb(0 0 0 / 5%)";
  context.fillRect(0, 0, canvas.width, canvas.height);

  particlesArray.forEach((particle) => particle.rotate());
}
```

Result

Advanced animations

← Previous Next →

In the last chapter we made some basic animations and got to know ways to get things moving. In this part we will have a closer look at the motion itself and are going to add some physics to make our animations more advanced.

Drawing a ball

We are going to use a ball for our animation studies, so let's first draw that ball onto the canvas. The following code will set us up.

HTML

```
<canvas id="canvas" width="600" height="300"></canvas>
```

As usual, we need a drawing context first. To draw the ball, we will create a `ball` object which contains properties and a `draw()` method to paint it on the canvas.

JS

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

const ball = {
  x: 100,
  y: 100,
  radius: 25,
  color: "blue",
  draw() {
    ctx.beginPath();
    ctx.arc(this.x, this.y, this.radius, 0, Math.PI * 2, true);
    ctx.closePath();
    ctx.fillStyle = this.color;
    ctx.fill();
  },
};

ball.draw();
```

Nothing special here, the ball is actually a simple circle and gets drawn with the help of the `arc()` method.

Adding velocity

Now that we have a ball, we are ready to add a basic animation like we have learned in the last chapter of this tutorial. Again, `window.requestAnimationFrame()` helps us to control the animation. The ball gets moving by adding a velocity vector to the position. For each frame, we also clear the canvas to remove old circles from prior frames.

JS

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

const ball = {
  x: 100,
  y: 100,
  vx: 5,
  vy: 2,
  radius: 25,
  color: "blue",
  draw() {
    ctx.beginPath();
    ctx.arc(this.x, this.y, this.radius, 0, Math.PI * 2, true);
    ctx.closePath();
    ctx.fillStyle = this.color;
    ctx.fill();
  },
};

function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ball.draw();
  ball.x += ball.vx;
  ball.y += ball.vy;
  raf = window.requestAnimationFrame(draw);
}

canvas.addEventListener("mouseover", (e) => {
  raf = window.requestAnimationFrame(draw);
});

canvas.addEventListener("mouseout", (e) => {
  window.cancelAnimationFrame(raf);
});

ball.draw();
```

Boundaries

Without any boundary collision testing our ball runs out of the canvas quickly. We need to check if the `x` and `y` position of the ball is out of the canvas dimensions and invert the direction of the velocity vectors. To do so, we add the following checks to the `draw` method:

JS

```
if (
  ball.y + ball.vy > canvas.height - ball.radius ||
  ball.y + ball.vy < ball.radius
) {
  ball.vy = -ball.vy;
}
if (
  ball.x + ball.vx > canvas.width - ball.radius ||
  ball.x + ball.vx < ball.radius
) {
  ball.vx = -ball.vx;
}
```

First demo

Let's see how it looks in action so far.

HTML

HTML

```
<canvas id="canvas" width="600" height="300"></canvas>
```

JAVASCRIPT

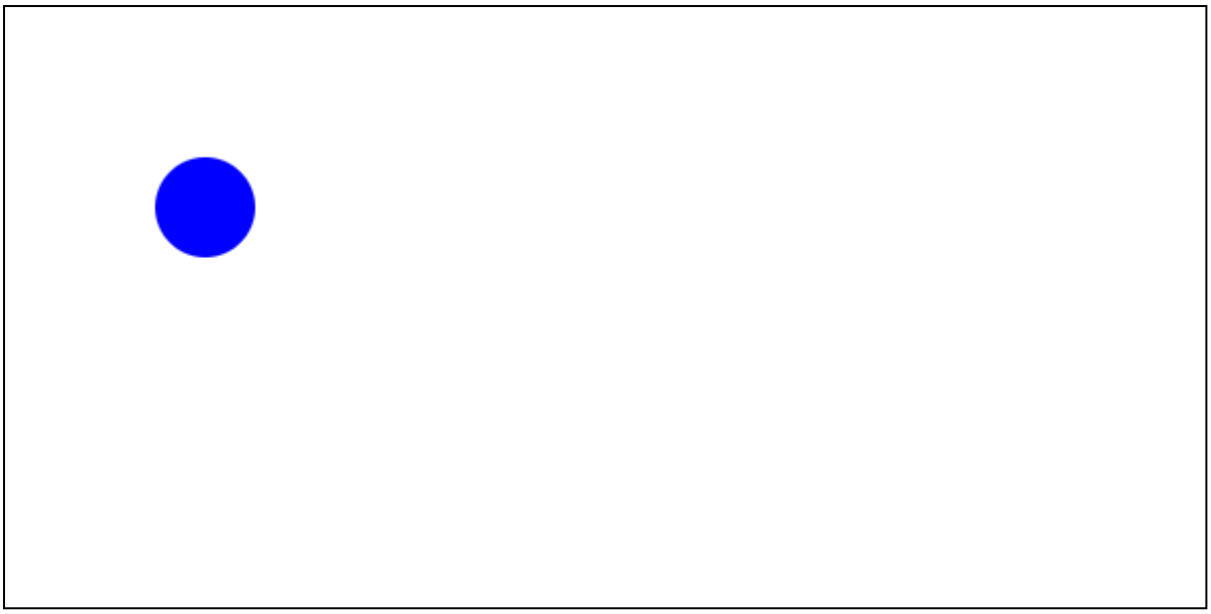
JS

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

const ball = {
  x: 100,
  y: 100,
  vx: 5,
```

```
  vy: 2,
  radius: 25,
  color: "blue",
  draw() {
    ctx.beginPath();
    ctx.arc(this.x, this.y, this.radius, 0, Math.PI * 2, true);
    ctx.closePath();
    ctx.fillStyle = this.color;
    ctx.fill();
  },
};

function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ball.draw();
  ball.x += ball.vx;
  ball.y += ball.vy;

  if (
    ball.y + ball.vy > canvas.height - ball.radius ||
    ball.y + ball.vy < ball.radius
  ) {
    ball.vy = -ball.vy;
  }
  if (
    ball.x + ball.vx > canvas.width - ball.radius ||
    ball.x + ball.vx < ball.radius
  ) {
    ball.vx = -ball.vx;
  }

  raf = window.requestAnimationFrame(draw);
}

canvas.addEventListener("mouseover", (e) => {
  raf = window.requestAnimationFrame(draw);
});

canvas.addEventListener("mouseout", (e) => {
  window.cancelAnimationFrame(raf);
});

ball.draw();
```

RESULT

Move your mouse into the canvas to start the animation.



Acceleration

To make the motion more real, you can play with the velocity like this, for example:

JS

```
ball.vy *= 0.99;
ball.vy += 0.25;
```

This slows down the vertical velocity each frame, so that the ball will just bounce on the floor in the end.

Second demo

HTML

HTML

```
<canvas id="canvas" width="600" height="300"></canvas>
```

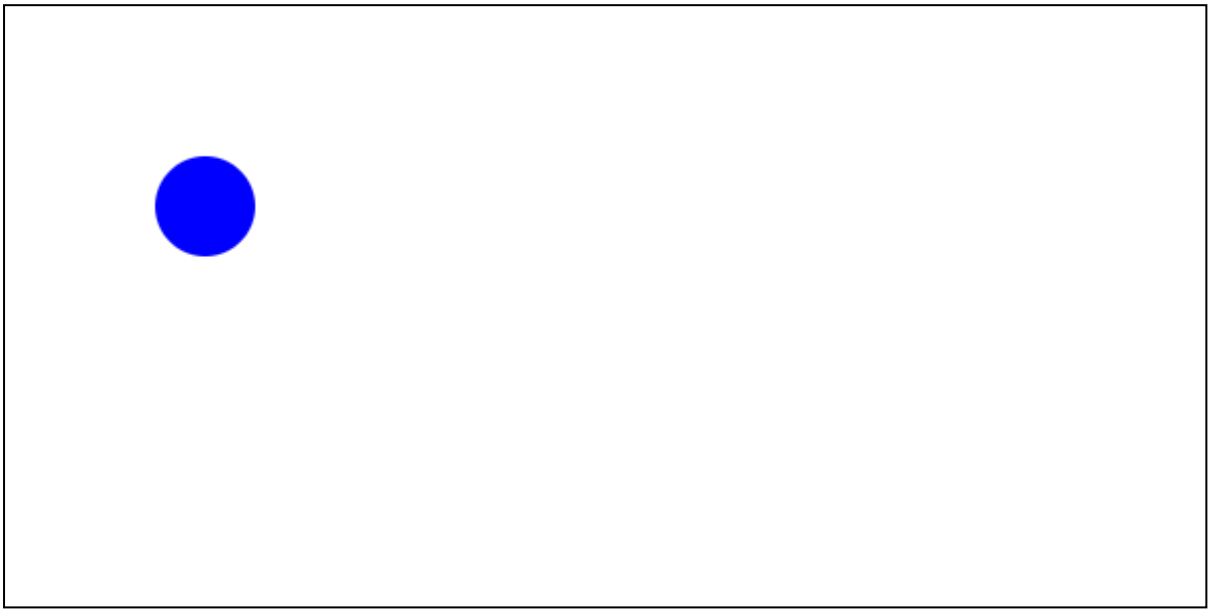
JAVASCRIPT

JS

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

const ball = {
  x: 100,
  y: 100,
  vx: 5,
  vy: 2,
  radius: 25,
  color: "blue",
  draw() {
    ctx.beginPath();
    ctx.arc(this.x, this.y, this.radius, 0, Math.PI * 2, true);
    ctx.closePath();
    ctx.fillStyle = this.color;
    ctx.fill();
  },
};

function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ball.draw();
  ball.x += ball.vx;
  ball.y += ball.vy;
  ball.vy *= 0.99;
  ball.vy += 0.25;

  if (
    ball.y + ball.vy > canvas.height - ball.radius ||
    ball.y + ball.vy < ball.radius
  ) {
    ball.vy = -ball.vy;
  }
  if (
    ball.x + ball.vx > canvas.width - ball.radius ||
    ball.x + ball.vx < ball.radius
  ) {
    ball.vx = -ball.vx;
  }

  raf = window.requestAnimationFrame(draw);
}

canvas.addEventListener("mouseover", (e) => {
  raf = window.requestAnimationFrame(draw);
});

canvas.addEventListener("mouseout", (e) => {
```

```
  window.cancelAnimationFrame(raf);
});

ball.draw();
```

RESULT



Trailing effect

Until now we have made use of the `clearRect` method when clearing prior frames. If you replace this method with a semi-transparent `fillRect`, you can easily create a trailing effect.

JS

```
ctx.fillStyle = "rgb(255 255 255 / 30%)";
ctx.fillRect(0, 0, canvas.width, canvas.height);
```

Third demo

HTML

HTML

```
<canvas id="canvas" width="600" height="300"></canvas>
```

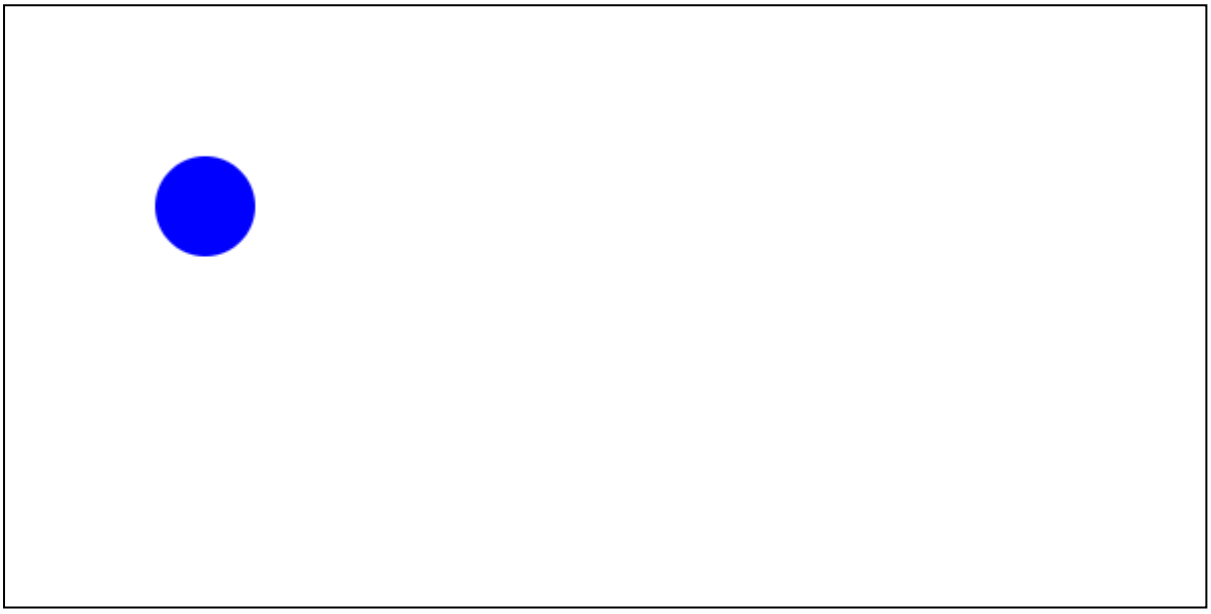
JAVASCRIPT

JS

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

const ball = {
  x: 100,
  y: 100,
  vx: 5,
  vy: 2,
  radius: 25,
  color: "blue",
  draw() {
    ctx.beginPath();
    ctx.arc(this.x, this.y, this.radius, 0, Math.PI * 2, true);
    ctx.closePath();
    ctx.fillStyle = this.color;
    ctx.fill();
  },
};

function draw() {
  ctx.fillStyle = "rgb(255 255 255 / 30%)";
  ctx.fillRect(0, 0, canvas.width, canvas.height);
  ball.draw();
  ball.x += ball.vx;
  ball.y += ball.vy;
  ball.vy *= 0.99;
  ball.vy += 0.25;

  if (
    ball.y + ball.vy > canvas.height - ball.radius ||
    ball.y + ball.vy < ball.radius
  ) {
    ball.vy = -ball.vy;
  }
  if (
    ball.x + ball.vx > canvas.width - ball.radius ||
    ball.x + ball.vx < ball.radius
  ) {
    ball.vx = -ball.vx;
  }

  raf = window.requestAnimationFrame(draw);
}
```

```
canvas.addEventListener("mouseover", (e) => {
  raf = window.requestAnimationFrame(draw);
});

canvas.addEventListener("mouseout", (e) => {
  window.cancelAnimationFrame(raf);
});

ball.draw();
```

RESULT



Adding mouse control

To get some control over the ball, we can make it follow our mouse using the `mousemove` event, for example. The `click` event releases the ball and lets it bounce again.

Fourth demo

HTML

HTML

```
<canvas id="canvas" width="600" height="300"></canvas>
```

JAVASCRIPT

JS

```
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");
let raf;
let running = false;

const ball = {
  x: 100,
  y: 100,
  vx: 5,
  vy: 1,
  radius: 25,
  color: "blue",
  draw() {
    ctx.beginPath();
    ctx.arc(this.x, this.y, this.radius, 0, Math.PI * 2, true);
    ctx.closePath();
    ctx.fillStyle = this.color;
    ctx.fill();
  },
};

function clear() {
  ctx.fillStyle = "rgb(255 255 255 / 30%)";
  ctx.fillRect(0, 0, canvas.width, canvas.height);
}

function draw() {
  clear();
  ball.draw();
  ball.x += ball.vx;
  ball.y += ball.vy;

  if (
    ball.y + ball.vy > canvas.height - ball.radius ||
    ball.y + ball.vy < ball.radius
  ) {
    ball.vy = -ball.vy;
  }
  if (
    ball.x + ball.vx > canvas.width - ball.radius ||
    ball.x + ball.vx < ball.radius
  ) {
    ball.vx = -ball.vx;
  }
```

```
  raf = window.requestAnimationFrame(draw);
}

canvas.addEventListener("mousemove", (e) => {
  if (!running) {
    clear();
    ball.x = e.clientX;
    ball.y = e.clientY;
    ball.draw();
  }
});

canvas.addEventListener("click", (e) => {
  if (!running) {
    raf = window.requestAnimationFrame(draw);
    running = true;
  }
});

canvas.addEventListener("mouseout", (e) => {
  window.cancelAnimationFrame(raf);
  running = false;
});

ball.draw();
```

RESULT

Move the ball using your mouse and release it with a click.

Breakout

This short chapter only explains some techniques to create more advanced animations. There are many more! How about adding a paddle, some bricks, and turn this demo into a Breakout game? Check out our Game development area for more gaming related articles.

See also

- `window.requestAnimationFrame()`

← Previous Next →

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);
```

Grayscaling 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 `<img>` 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 →

Optimizing canvas

← Previous Next →

The `<canvas>` element is one of the most widely used tools for rendering 2D graphics on the web. However, when websites and apps push the Canvas API to its limits, performance begins to suffer. This article provides suggestions for optimizing your use of the canvas element to ensure that your graphics perform well.

Performance tips

The following is a collection of tips to improve canvas performance.

Pre-render similar primitives or repeating objects on an offscreen canvas

If you find yourself repeating some of the same drawing operations on each animation frame, consider offloading them to an offscreen canvas. You can then render the offscreen image to your primary canvas as often as needed, without unnecessarily repeating the steps needed to generate it in the first place.

JS

```
myCanvas.offscreenCanvas = document.createElement("canvas");
myCanvas.offscreenCanvas.width = myCanvas.width;
myCanvas.offscreenCanvas.height = myCanvas.height;

myCanvas.getContext("2d").drawImage(myCanvas.offScreenCanvas, 0, 0);
```

Avoid floating-point coordinates and use integers instead

Sub-pixel rendering occurs when you render objects on a canvas without whole values.

JS

```
ctx.drawImage(myImage, 0.3, 0.5);
```

This forces the browser to do extra calculations to create the anti-aliasing effect. To avoid this, make sure to round all co-ordinates used in calls to `drawImage()` using `Math.floor()`, for example.

Don't scale images in `drawImage`

Cache various sizes of your images on an offscreen canvas when loading as opposed to constantly scaling them in `drawImage()`.

Use multiple layered canvases for complex scenes

In your application, you may find that some objects need to move or change frequently, while others remain relatively static. A possible optimization in this situation is to layer your items using multiple `<canvas>` elements.

For example, let's say you have a game with a UI on top, the gameplay action in the middle, and a static background on the bottom. In this case, you could split your game into three `<canvas>` layers. The UI would change only upon user input, the gameplay layer would change with every new frame, and the background would remain generally unchanged.

HTML

```
<div id="stage">
  <canvas id="ui-layer" width="480" height="320"></canvas>
  <canvas id="game-layer" width="480" height="320"></canvas>
  <canvas id="background-layer" width="480" height="320"></canvas>
</div>
```

CSS

```
#stage {
  width: 480px;
  height: 320px;
  position: relative;
  border: 2px solid black;
}

canvas {
  position: absolute;
}
#ui-layer {
  z-index: 3;
}
#game-layer {
  z-index: 2;
}
#background-layer {
  z-index: 1;
}
```

Use plain CSS for large background images

If you have a static background image, you can draw it onto a plain `<div>` element using the CSS `background` property and position it under the canvas. This will negate the need to render the background to the canvas on every tick.

Scaling canvas using CSS transforms

CSS transforms are faster since they use the GPU. The best case is to not scale the canvas, or have a smaller canvas and scale up rather than a bigger canvas and scale down.

JS

```
const scaleX = window.innerWidth / canvas.width;
const scaleY = window.innerHeight / canvas.height;

const scaleToFit = Math.min(scaleX, scaleY);
const scaleToCover = Math.max(scaleX, scaleY);

stage.style.transformOrigin = "0 0"; // Scale from top left
stage.style.transform = `scale(${scaleToFit})`;
```

Turn off transparency

If your application uses canvas and doesn't need a transparent backdrop, set the `alpha` option to `false` when creating a drawing context with `HTMLCanvasElement.getContext()`. This information can be used internally by the browser to optimize rendering.

JS

```
const ctx = canvas.getContext("2d", { alpha: false });
```

Scaling for high resolution displays

You may find that canvas items appear blurry on higher-resolution displays. While many solutions may exist, a simple first step is to scale the canvas size up and down simultaneously, using its attributes, styling, and its context's scale.

JS

```
// Get the DPR and size of the canvas
const dpr = window.devicePixelRatio;
const rect = canvas.getBoundingClientRect();

// Set the "actual" size of the canvas
```

```
canvas.width = rect.width * dpr;
canvas.height = rect.height * dpr;

// Scale the context to ensure correct drawing operations
ctx.scale(dpr, dpr);

// Set the "drawn" size of the canvas
canvas.style.width = `${rect.width}px`;
canvas.style.height = `${rect.height}px`;
```

More tips

- Batch canvas calls together. For example, draw a polyline instead of multiple separate lines.
- Avoid unnecessary canvas state changes.
- Render screen differences only, not the whole new state.
- Avoid the `shadowBlur` property whenever possible.
- Avoid text rendering whenever possible.
- Try different ways to clear the canvas (`clearRect()` vs. `fillRect()` vs. resizing the canvas).
- With animations, use `Window.requestAnimationFrame()` instead of `setInterval()`.
- Be careful with heavy physics libraries.

← Previous Next →

Finale

← Previous

Congratulations! You finished the Canvas tutorial! This knowledge will help you to make great 2D graphics on the web.

More examples and tutorials

There are a variety of demos and further explanations about canvas on these sites:

Canvas Codepens

Front End Developer Playground & Code Editor in the Browser.

Game development

Gaming is one of the most popular computer activities. New technologies are constantly arriving to make it possible to develop better and more powerful games that can be run in any standards-compliant web browser.

Other Web APIs

These APIs might be useful when working further with canvas and graphics:

WebGL

Advanced API for rendering complex graphics, including 3D.

SVG

Scalable Vector Graphics let you describe images as sets of vectors (lines) and shapes in order to allow them to scale smoothly regardless of the size at which they're drawn.

Web Audio

The Web Audio API provides a powerful and versatile system for controlling audio on the Web, allowing developers to choose audio sources, add effects to audio, create audio visualizations, apply spatial effects (such as panning) and much more.

Questions

Stack Overflow

Questions tagged with "canvas".

Comments about this tutorial – the MDN documentation community