

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%)";
```

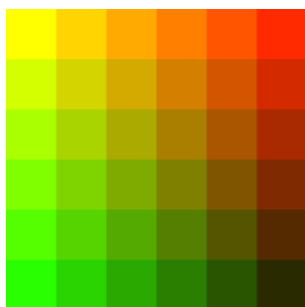
A `fillStyle` example

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:



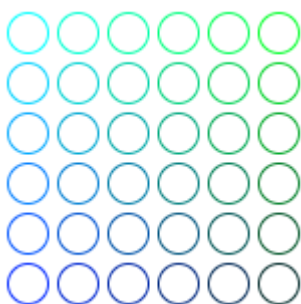
A `strokeStyle` example

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

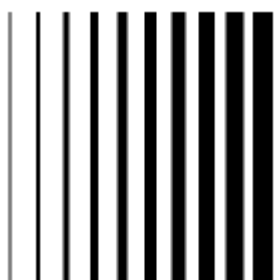
This property sets the current line thickness. Values must be positive numbers. By default this value is set to 1.0 units.

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.

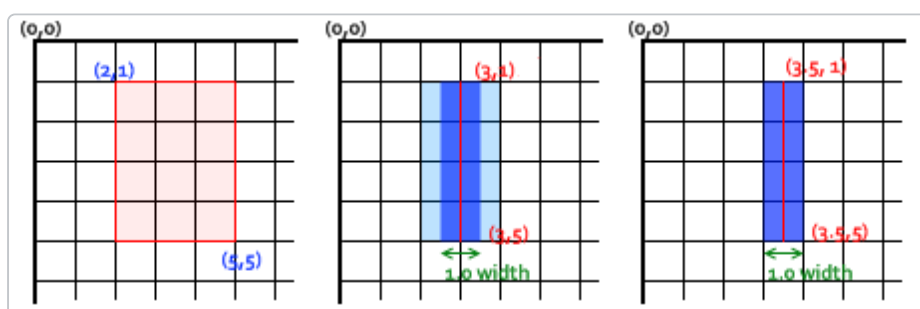
In the example below, 10 straight lines are drawn with increasing line widths. The line on the far left is 1.0 units wide. However, the leftmost and all other odd-integer-width thickness lines do not appear crisp, because of the path's positioning.

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:

- $\text{miterLimit} = \max \text{miterLength} / \text{lineWidth} = 1 / \sin(\min \theta / 2)$
- 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");

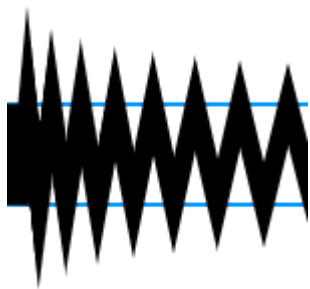
  // Clear canvas
  ctx.clearRect(0, 0, 150, 150);

  // 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

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");

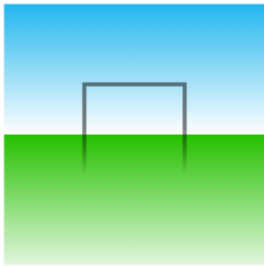
  const linGrad2 = ctx.createLinearGradient(0, 50, 0, 95);
  linGrad2.addColorStop(0.5, "black");
  linGrad2.addColorStop(1, "transparent");

  // 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 its 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);  
}
```

Sample String

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");  
}
```

