**Chapter 4**

**HTML 5 canvas and SVG 1,2**

HTML5 is an updated version of HTML which introduces new elements to further enrich web pages, and allow designers to create animations and graphical elements in new ways. SVG and canvas are two such new elements that allow designers to create rich graphics inside the browser with code.

HTML5 canvas is a JavaScript API, which can be used to perform complex drawing operations using programming. This is achieved by using the canvas element in an HTML document—a blank area inside which you can draw.

# HTML5 Canvas[[1]](#footnote-0)

One of the most important instruments in a painter’s toolkit is their canvas. It gives them the freedom to express all kinds of feelings, impressions, ideas, and so forth, in almost unlimited variations and combinations. And that freedom can be restricted only by two things — their skill level and their imagination. This chapter describes how to use the <canvas> and <SVG> elements to draw 2D graphics. Using the <canvas> element is not very difficult, but you do need a basic understanding of HTML and JavaScript. The <canvas> element is not supported in some older browsers, but is supported in recent versions of all major browsers. The default size of the canvas is 300 pixels × 150 pixels (width × height). But custom sizes can be defined using the HTML height and width property. In order to draw graphics on the canvas we use a JavaScript context object, which creates graphics on the fly.

# What is HTML5 Canvas?

The canvas element is an element defined in HTML code using width and height attributes. The real power of the canvas element, however, is accomplished by taking advantage of the HTML5 Canvas API. This API is used by writing JavaScript that can access the canvas area through a full set of drawing functions, thus allowing for dynamically generated graphics.

The <canvas> element differs from an <img> tag in that, like for <video>, <audio>, or <picture> elements, it is easy to define some fallback content, to be displayed in older browsers not supporting it, like versions of Internet Explorer earlier than version 9 or textual browsers. You should always provide fallback content to be displayed by those browsers.

**<canvas>** is an HTML element which can be used to draw graphics via scripting (usually JavaScript). This can, for instance, be used to draw graphs, combine photos, or create simple (and not so simple) animations.

### 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" ...></canvas> is fully compatible with all browsers that support canvas at all.

# Canvas Rendering Contexts

Every HTML5 canvas element must have a ***context***. The context defines what HTML5 Canvas API you’ll be using. The 2d context is used for drawing 2D graphics and manipulating bitmap images. The 3d context is used for 3D graphics creation and manipulation. The latter is actually called WebGL and it’s based on OpenGL ES.

To get started with canvas, all you need is a code editor and a browser with HTML5 canvas support.

Our HTML to start will look like this:

<canvas id="exampleCanvas" width="500" height="300">

<! -- OPTION 1: leave a message here if browser doesn't support canvas -->

<p>Your browser doesn’t currently support HTML5 Canvas.

Please check caniuse.com/#feat=canvas for information on

browser support for canvas. </p>

<! -- OPTION 2: put fallback content (text, image, Flash, etc.)

if the browser doesn't support canvas -->

</canvas>

And here is our JavaScript, which we can include at the bottom of our HTML page:

var canvas = document.getElementById('exampleCanvas'),

context = canvas.getContext('2d');

// code examples will continue here...

By default, the browser creates canvas elements with a width of 300 pixels and a height of 150 pixels, even if these aren’t specified in the HTML. You can change the size by specifying the width and height, as you have done in the HTML.

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 chapter, you specify "2d" to get a CanvasRenderingContext2D.

Notice also that we’ve given the canvas an id attribute that we’ll use in our JavaScript. And finally, if you want, you can use CSS to add a border to make the canvas visible by means of a thin frame. This is not required; it’s used as a visual aid for us to see the boundary of the canvas element.

Notice that between the opening and closing <canvas> tags, there is a content that will be displayed if the browser doesn’t support canvas. This can be just about any type of content that an older browser supports.

JavaScript starts with two lines. In the first line we create a variable that caches our canvas element via its ID. The second line creates a variable (context) that references the 2D context for the canvas using the getContext() function. We’ll use the context variable to access all canvas drawing functions and properties.

With our canvas ready to go we can start experimenting with the Canvas API. But before that, let’s clarify a few more aspects of the canvas feature.

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. 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.

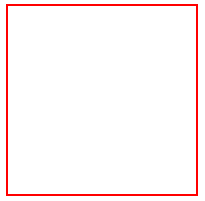
The <canvas> element can be styled just like any normal image (margin, border, background…). These rules, however, don't affect the actual drawing on the canvas.

Here is an example that consider a template for all the following examples



The script includes a function called draw(), which is executed once the page finishes loading; this is done by listening for the load event on the document.

the output looks like this



## Advantages of HTML5 Canvas?

Here are some reasons you might want to consider learning to use HTML5’s canvas feature:

1. **Interactivity.** Canvas is fully interactive. It can respond to a user’s actions by listening for keyboard, mouse, or touch events. So a developer is not restricted to only static graphics and images.
2. **Animation.** Every object drawn on the canvas can be animated. This allows for all levels of animations to be created, from simple bouncing balls to complex animations.
3. **Flexibility.** Developers can create just about anything using canvas. It can display lines, shapes, text, images, etc. — with or without animation. Plus, adding video and/or audio to a canvas application is also possible.
4. **Browser/Platform Support.** HTML5 Canvas is supported by all major browsers and can be accessed on a wide range of devices including desktops, tablets, and smartphones.
5. **Popularity.** Canvas popularity is rapidly and steadily growing so there is no shortage of resources available.
6. **It’s a web standard.** Unlike Flash and Silverlight, Canvas is open technology that’s part of HTML5. And although not all of its features are implemented in all browsers, developers can be certain canvas will be around indefinitely.
7. **Develop once, run everywhere.** HTML5 Canvas offers great portability. Once created, Unlike Flash and Silverlight, a canvas application can run almost anywhere — from the largest computers to the smallest mobile devices.
8. **Free and accessible development tools.** The basic tools for creating HTML5 canvas applications are just a browser and a good code editor. Plus, there are many great and free libraries and tools to help developers with advanced canvas development.

# What Applications Can You Create with HTML5 Canvas?

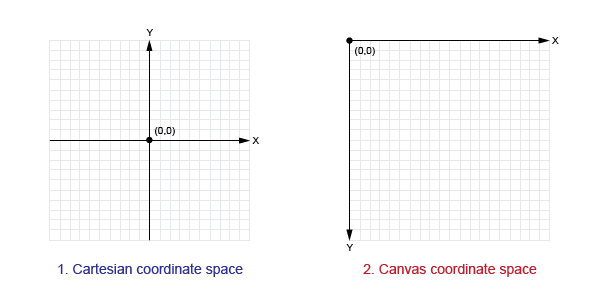
1. **Gaming.** The HTML5 Canvas’ feature set is an ideal candidate for producing all sorts of 2D and 3D games.
2. **Advertising.** HTML5 Canvas is a great replacement for Flash-based banners and ads. It has all the needed features for attracting buyers’ attention.
3. **Data Representation.** HTML5 can collect data from global data sources and use it to generate visually appealing and interactive graphs and charts with the canvas element.
4. **Education and Training.** HTML5 canvas can be used to produce effective and attractive learning experiences, combining text, images, videos, and audio.
5. **Art and Decoration.** With a little bit of imagination and canvases wide variety of colors, gradients, patterns, transparency, shadows, and clipping features, all kinds of artistic and decorative graphics can be drawn.

Now that we know why we should learn canvas, let’s look at the basics of HTML5 Canvas and how to start using it.

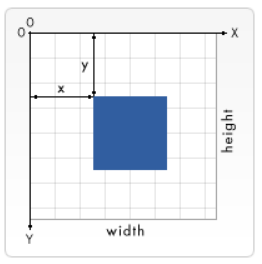
# Understanding the Canvas Coordinate System

In a 2D space, positions are referenced using X and Y coordinates. The X axis extends horizontally, and the Y axis extends vertically. The center has a position x = 0 and y = 0, that can also be expressed as (0, 0). This method of positioning objects, used in mathematics, is known as the Cartesian coordinate system.

The Canvas coordinate system, however, places the origin at the ***upper-left corner*** of the canvas, with X coordinates increasing to the right and Y coordinates increasing toward the bottom of the canvas (Fig.1). *So unlike a standard Cartesian coordinate space, the Canvas space doesn’t have visible negative points*. Using negative coordinates won’t cause your application to fail, but objects positioned using negative coordinate points won’t appear on the page.



*Figure 1: The Cartesian and canvas coordinates*



## 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)



# Basic canvas drawing

## 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:

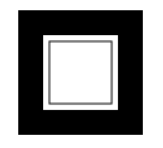
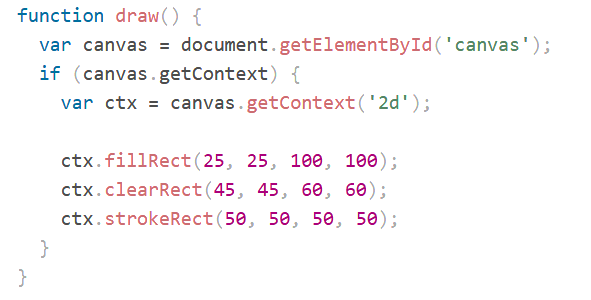
***fillRect(x, y, width, height):*** Draws a filled rectangle.

***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 example, but now it is making use of these three functions.



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.

## 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:**

***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.

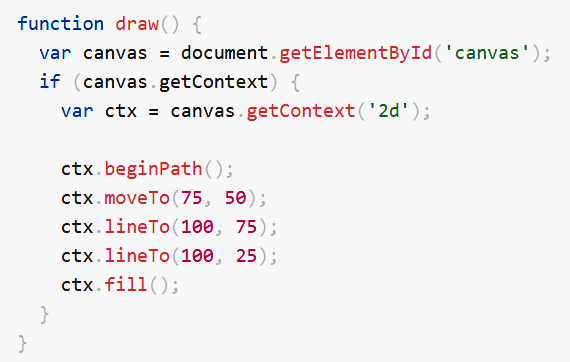
The second step is calling the methods that actually specify the paths to be drawn. We'll see these shortly.

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:



## 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.

## Arcs

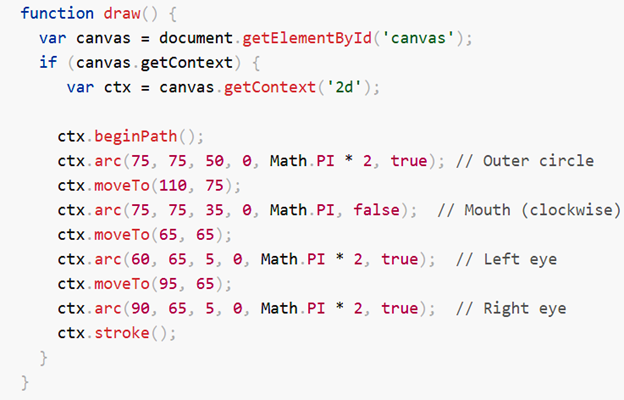
To draw arcs or circles, we use the arc() or arcTo() methods.

***arc(x, y, radius, startAngle, endAngle, anticlockwise) :***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 *anticlockwise* (defaulting to clockwise).The anticlockwise parameter is a Boolean value which, when true, draws the arc anticlockwise; otherwise, the arc is drawn 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.

**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.

To try this for yourself, you can use the code snippet below.



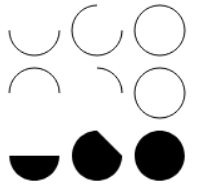
If you'd like to see the connecting lines, you can remove the lines that call moveTo().

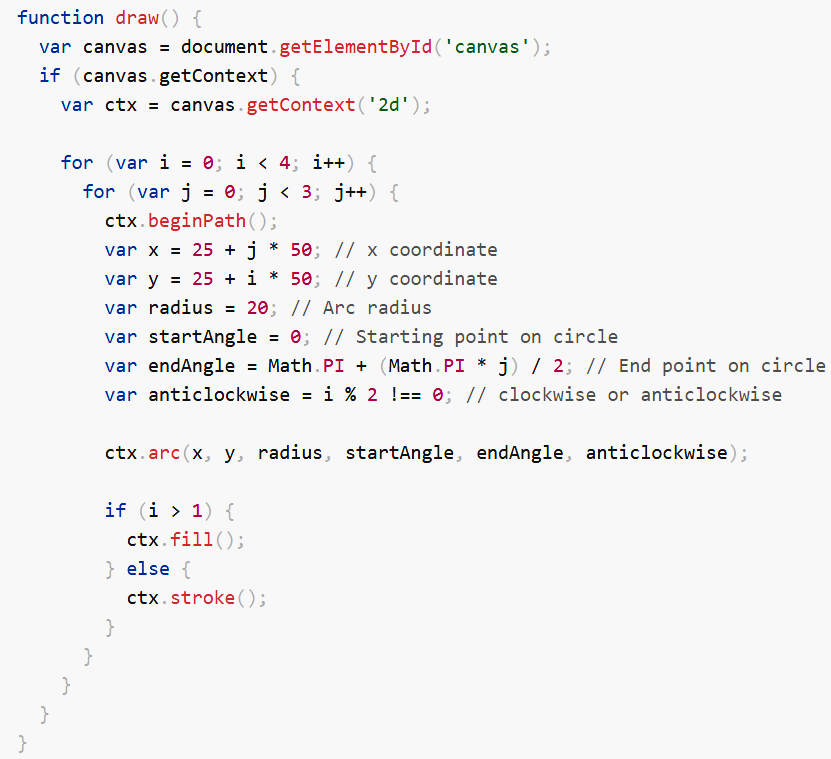
The following example is a little more complex than the ones we've seen above. It draws 12 different arcs all with different angles and fills.

The two [for loops](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for) are for looping through the rows and columns of arcs. For each arc, we start a new path by calling beginPath().

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.





## 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.



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.

## [Line styles](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Applying_styles_and_colors#line_styles)

There are several properties which allow us to style lines.

[lineWidth = value](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/lineWidth): Sets the width of lines drawn in the future.

[lineCap = type](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/lineCap): Sets the appearance of the ends of lines.

[lineJoin = type](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/lineJoin): Sets the appearance of the "corners" where lines meet.

[miterLimit = value](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/miterLimit): Establishes a limit on the miter when two lines join at a sharp angle, to let you control how thick the junction becomes.

[getLineDash()](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/getLineDash): Returns the current line dash pattern array containing an even number of non-negative numbers.

[setLineDash(segments)](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/setLineDash): Sets the current line dash pattern.

[lineDashOffset = value](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/lineDashOffset): Specifies where to start a dash array on a line.

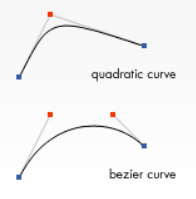
## Bezier and quadratic curves

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).

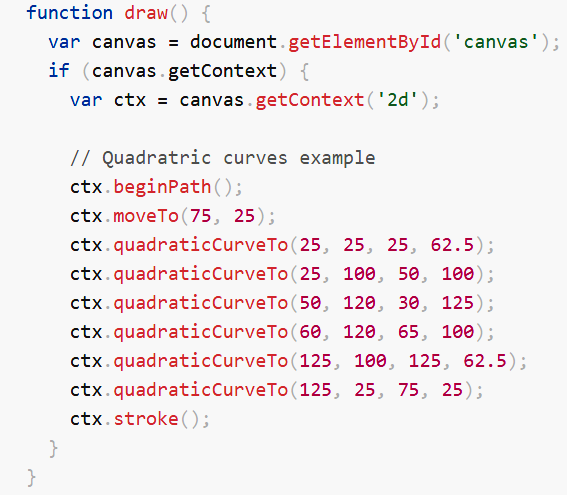
The difference between these can best be described using the image on below (Fig.3). A quadratic Bézier curve has a start and an end point (blue dots) and just one **control point** (indicated by the red dot) while a cubic Bézier curve uses two control points.



*Figure 3: Bezier and quadratic curves*

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.



This example draws a heart using cubic Bezier curves.

## 

## 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.

# Applying styles and colors

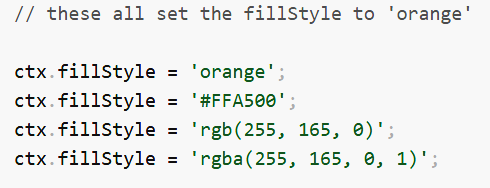
Up until now we have only seen methods of the drawing context. If we want to apply colors to a shape, there are two important properties we can use: fillStyle and strokeStyle.

***fillStyle = color***: Sets the style used when filling shapes.

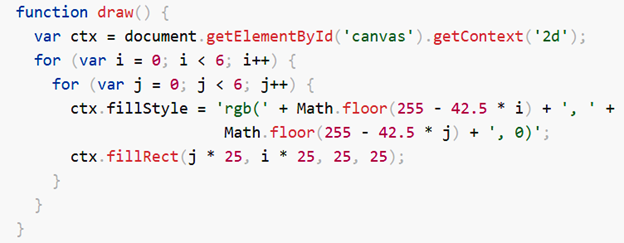
***strokeStyle = color***: Sets the style for shapes' outlines.

color is a string representing a CSS <color>, a gradient object, or a pattern object. By default, the stroke and fill color are set to black (CSS color value #000000).

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>](https://developer.mozilla.org/en-US/docs/Web/CSS/color_value) values. Each of the following examples describe the same color.



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.



# [Transparency](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Applying_styles_and_colors#transparency)

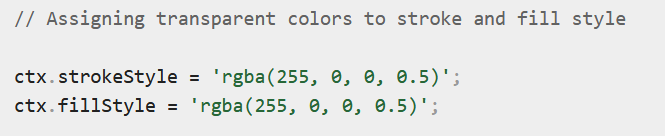
In addition to drawing opaque shapes to the canvas, we can also draw semi-transparent (or translucent) shapes. This is done by either setting the globalAlpha property or by assigning a semi-transparent color to the stroke and/or fill style.

[globalAlpha = transparencyValue](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/globalAlpha)

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.

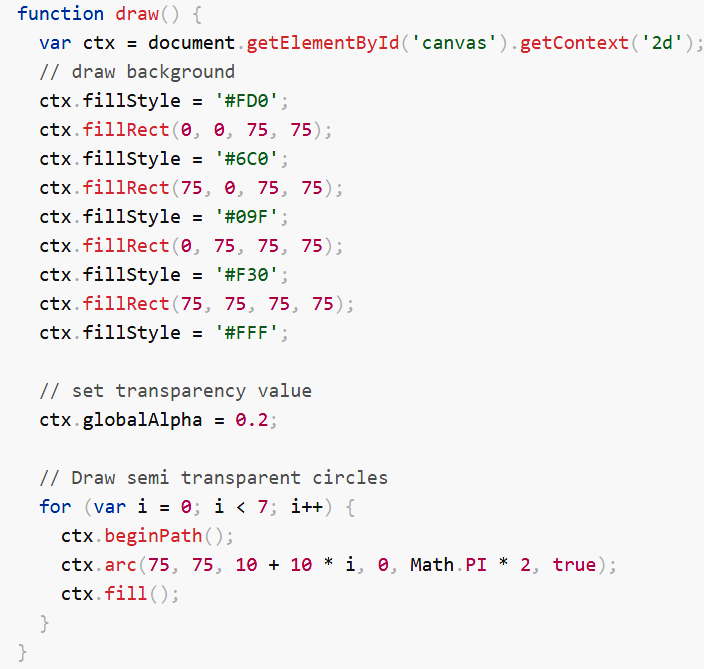
The globalAlpha property can be useful if you want to draw a lot of shapes on the canvas with similar transparency, but otherwise it's generally more useful to set the transparency on individual shapes when setting their colors.

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



The rgba() function is similar to the rgb() function but it has one extra parameter. The last parameter sets the transparency value of this particular color. The valid range is again between 0.0 (fully transparent) and 1.0 (fully opaque).

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.

## [Gradients](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Applying_styles_and_colors#gradients)

Just like any normal drawing program, we can fill and stroke shapes using linear, radial and conic gradients. We create a [CanvasGradient](https://developer.mozilla.org/en-US/docs/Web/API/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)](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/createLinearGradient): 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)](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/createRadialGradient): 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)](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/createConicGradient): Creates a conic gradient object with a starting angle of angle in radians, at the position (x, y).

For example:

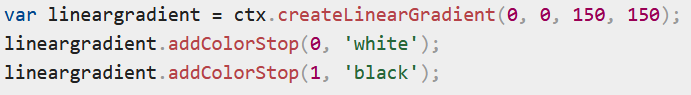
var lineargradient = ctx.createLinearGradient(0, 0, 150, 150);

var 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)](https://developer.mozilla.org/en-US/docs/Web/API/CanvasGradient/addColorStop): 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>](https://developer.mozilla.org/en-US/docs/Web/CSS/color_value), 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.



## [Patterns](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Applying_styles_and_colors#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)](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/createPattern): Creates and returns a new canvas pattern object. image is a [CanvasImageSource](https://developer.mozilla.org/en-US/docs/Web/API/CanvasImageSource) (that is, an [HTMLImageElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLImageElement), another canvas, a [<video>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video) element, or the like. 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](https://developer.mozilla.org/en-US/docs/Web/API/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:



## [Shadows](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Applying_styles_and_colors#shadows)

Using shadows involves just four properties:

[shadowOffsetX = float](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/shadowOffsetX): 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](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/shadowOffsetY): 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](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/shadowBlur): 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.

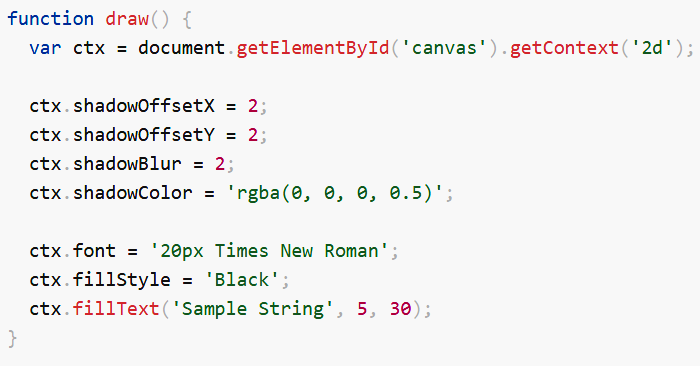
[shadowColor = color](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/shadowColor): A standard CSS color value indicating the color of the shadow effect; by default, it is fully-transparent black.

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.

Example:



# 

# 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.





# 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 CanvasImageSource specified by the image parameter at the coordinates (x, y).

# Transformations

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.

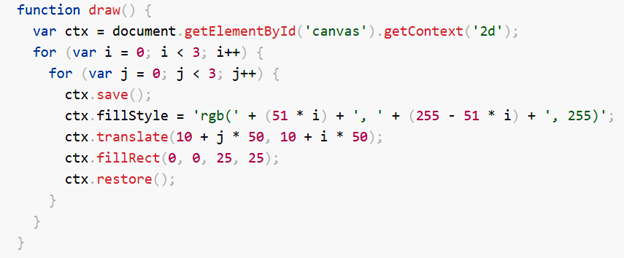
***restore()***: Restores the most recently saved canvas state.

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.

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.

# SVG (scalable vector graphics)

SVG is an XML language, similar to XHTML, which can be used to draw vector graphics, such as the ones shown below. It can be used to create an image either by specifying all the lines and shapes necessary, by modifying already existing raster images, or by a combination of both. The image and its components can also be transformed, composited together, or filtered to change their appearance completely.

The Scalable Vector Graphics (SVG) is an XML-based image format that is used to define two-dimensional vector based graphics for the web. Unlike raster images (e.g. .jpg, .gif, .png, etc.), a vector image can be scaled up or down to any extent without losing the image quality.

An SVG image is drawn out using a series of statements that follow the XML schema — that means SVG images can be created and edited with any text editor, such as Notepad. There are several other advantages of using SVG over other image formats like JPEG, GIF, PNG, etc.

SVG images can be searched, indexed, scripted, and compressed.

SVG images can be created and modified using JavaScript in real time.

SVG images can be printed with high quality at any resolution.

SVG content can be animated using the built-in animation elements.

SVG images can contain hyperlinks to other documents.

SVG came about in 1999 after several competing formats had been submitted to the W3C and failed to be fully ratified. SVG is supported by all major browsers. A downside is loading SVG can be slow. SVG does offer benefits, some of which include having a DOM interface available for it, and not requiring third-party extensions. Whether or not to use it often depends on your specific use case.

HTML provides elements for defining headers, paragraphs, tables, and so on. In much the same way SVG provides elements for circles, rectangles, and simple and complex curves. A simple SVG document consists of nothing more than the <svg> root element and several basic shapes that build a graphic together. In addition, there is the <g> element, which is used to group several basic shapes together.

## Embedding SVG into HTML Pages

You can embed SVG graphics directly into your document using the HTML5 <svg> element.

Let's take a look at the following example to understand how it basically works:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Embedding SVG in HTML</title>

</head>

<body>

<svg width="300" height="200">

<text x="10" y="20" style="font-size:14px;">

Your browser supports SVG.

</text>

Sorry, your browser does not support SVG.

</svg>

</body>

</html>



The rendering process involves the following:

1. We start with the <svg> root element:
   * a doctype declaration as known from (X)HTML should be left off because DTD based SVG validation leads to more problems than it solves
   * before SVG 2, to identify the version of the SVG for other types of validation the version and baseProfile attributes should always be used instead. Both version and baseProfile attributes are deprecated in SVG 2.
   * as an XML dialect, SVG must always bind the namespaces correctly (in the xmlns attribute). See the Namespaces Crash Course page for more info.
2. The background is set to red by drawing a rectangle <rect> that covers the complete image area.
3. A green circle <circle> with a radius of 80px is drawn atop the center of the red rectangle (center of circle offset 150px to the right, and 100px downward from the top left corner).
4. The text "SVG" is drawn. The interior of each letter is filled in with white. The text is positioned by setting an anchor where we want the midpoint to be: in this case, the midpoint should correspond to the center of the green circle. Fine adjustments can be made to the font size and vertical position to ensure the final result is aesthetically pleasing.

**Note:** All the major modern web browsers like Chrome, Firefox, Safari, and Opera, as well as Internet Explorer 9 and above support inline SVG rendering.

## Drawing Path and Shapes with SVG

The following section will explain how to draw basic vector-based paths and shapes on the web pages using the newly introduced HTML5 <svg> element.

## Drawing a Line

The most basic path you can draw with SVG is a straight line. The following example will show you how to create a straight line using the SVG <line> element:

<svg width="300" height="200">

<line x1="50" y1="50" x2="250" y2="150" style="stroke:red; stroke-width:3;" />

</svg>

The attributes x1, x2, y1 and y2 of the <line> element draw a line from (x1,y1) to (x2,y2).

## Drawing a Rectangle

You can create simple rectangle and square shapes using the SVG <rect> element. The following example will show you how to create and style a rectangular shape with SVG:

<svg width="300" height="200">

<rect x="50" y="50" width="200" height="100" style="fill:orange; stroke:black; stroke-width:3;" />

</svg>

The attributes x and y of the <rect> element define the coordinates of the top-left corner of the rectangle. The attributes width and height specifies the width and height of the shape.

## Drawing a Circle

You can also create the circle shapes using the SVG <circle> element. The following example will show you how to create and style a circular shape with SVG:

<svg width="300" height="200">

<circle cx="150" cy="100" r="70" style="fill:lime; stroke:black; stroke-width:3;" />

</svg>

The attributes cx and cy of the <circle> element defines the coordinates of the center of the circle and the attribute r specifies the radius of the circle. However, if the attributes cx and cy are omitted or not specified, the center of the circle is set to (0,0).

## Drawing Text with SVG

You can also draw text on the web pages with SVG. The text in SVG is rendered as a graphic so you can apply all the graphic transformation to it but it still acts like text — that means it can be selected and copied as text by the user. Let's try an example to see how this works:

<svg width="300" height="200">

<text x="20" y="30" style="fill:purple; font-size:22px;">

Welcome to Our Website!

</text>

<text x="20" y="30" dx="0" dy="20" style="fill:navy; font-size:14px;">

Here you will find lots of useful information.

</text>

</svg>

The attributes x and y of the <text> element defines the location of the top-left corner in absolute terms whereas the attributes dx and dy specifies the relative location.

You can even use the <tspan> element to reformat or reposition the span of text contained within a <text> element. Text contained in separate tspans, but inside the same text element can all be selected at once — when you click and drag to select the text. However, the text in separate text elements cannot be selected at the same time. Let's check out an example:

<svg width="300" height="200">

<text x="30" y="15" style="fill:purple; font-size:22px; transform:rotate(30deg);">

<tspan style="fill:purple; font-size:22px;">

Welcome to Our Website!

</tspan>

<tspan dx="-230" dy="20" style="fill:navy; font-size:14px;">

Here you will find lots of useful information.

</tspan>

</text>

</svg>

# Differences between SVG and Canvas

The HTML5 introduced the two new graphical elements <canvas> and <svg> for creating rich graphics on the web, but they are fundamentally different.

The following table summarizes some of the basic differences between these two elements, which will help you to understand how to use these elements effectively and appropriately.

|  |  |
| --- | --- |
| **SVG** | **Canvas** |
| Vector based (composed of shapes) | Raster based (composed of pixel) |
| Multiple graphical elements, which become the part of the page's DOM tree | Single element similar to [<img>](https://www.tutorialrepublic.com/html-reference/html-img-tag.php) in behavior. Canvas diagram can be saved to PNG or JPG format |
| Modified through script and CSS | Modified through script only |
| Good text rendering capabilities | Poor text rendering capabilities |
| Give better performance with smaller number of objects or larger surface, or both | Give better performance with larger number of objects or smaller surface, or both |
| Better scalability. Can be printed with high quality at any resolution. Pixelation does not occur | Poor scalability. Not suitable for printing on higher resolution. Pixelation may occur |

1. <https://www.sitepoint.com/html5-canvas-tutorial-introduction/>

   2 <https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D> [↑](#footnote-ref-0)