**What is HTML5 Canvas?**

<canvas> is an HTML element which can be used to draw graphics via scripting (usually JavaScript). Canvas element is place on the page only as a container for graphics, thus using JavaScript to draw graphics.

It can be used to draw graphs, combine photos or create simple and complex animation. Canvas was first introduced in WebKit by Apple for the OS X Dashboard and since its became well supported for all major browsers. ( Chrome, internet Explorer 9 and up, FireFox 2.0, Safari 3.1, Opera 9.0 ).

**Start to draw on Canvas**

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

We can declare id/class to canvas element and also width and height properties, but this is not required. We could use stylesheet css file and style our canvas giving it background color, border, height, width ..etc.

It is important to note that order of drawing to canvas does matter. Drawing a ball that bounces on the screen and rectangular that might be a background block, ball will have to be drawn after the block. We will see this later in our code.

**The rendering context**

The [<canvas>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas) element creates a fixed-size drawing surface that exposes one or more **rendering contexts**. We will focus on 2D rendering context.

Canvas is initially blank. To display something , a script first needs to access the rendering context and draw on it.

Using a build in method called [getContext()](https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement/getContext) is making use of obtaining the rendering context and its drawing functions. This method takes one parameter – the type of context. In our case it is “2d”.

var canvas = document.getElementById('myCanvas');

var ctx = canvas.getContext('2d');

**Simple example**

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8"/>

<script>

function draw() {

var canvas = document.getElementById(myCanvas);

var ctx = canvas.getContext('2d');

ctx.fillStyle = 'rgb(200, 0, 0)';

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

ctx.fillRect(10, 10, 50, 50);

ctx.fillStyle = 'rgba(0, 0, 200, 0.5)';

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

ctx.fillRect(30, 30, 50, 50);

}

</script>

</head>

<body onload="draw();">

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

</body>

</html>

**Drawing Rectangles**

There are three functions that draw rectangles on the canvas:

[**fillRect(x, y, width, height)**](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/fillRect)

Draws a filled rectangle.

[**strokeRect(x, y, width, height)**](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/strokeRect)

Draws a rectangular outline.

[**clearRect(x, y, width, height)**](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/clearRect)

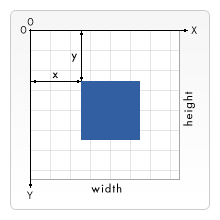
Clears the specified rectangular area, making it fully transparent.

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.

**The Grid**

This is a coordinate space on which we draw. So let’s look into how we calculate elements position and their width and height.

Normally 1 unit in the grid corresponds to 1 pixel on the canvas. The origin of this grid is positioned in the top leftcorner 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).



**Drawing a Arc/Circle**

[**arc(x, y, radius, startAngle, endAngle, anticlockwise)**](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/arc)

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).

 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 anticlockwise parameter is a Boolean value which, when true, draws the arc anticlockwise; 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.

So our example for the above will look like this.

function draw() {

var canvas = document.getElementById(myCanvas);

var ctx = canvas.getContext('2d');

ctx.fillStyle = 'rgb(200, 0, 0)';

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

ctx.fillRect(10, 10, 50, 50);

ctx.fillStyle = 'rgba(0, 0, 200, 0.5)';

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

ctx.fillRect(30, 30, 50, 50);

ctx.fillStyle = 'rgba(0, 0, 200, 0.5)';

// arc( centerX , centerY, radius, 0, Math.PI\*2, true)

ctx.arc( 20, 20, 50, 0, Math.PI\*2, true);

}

**Colors**

When applying color to the a shape that we drew, we can use two properties fillStyle and strokeStyle.

[**fillStyle = color**](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/fillStyle)

Sets the style used when filling shapes.

[**strokeStyle = color**](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/strokeStyle)

Sets the style for shapes' outlines.

color is a string representing a CSS [<color>](https://developer.mozilla.org/en-US/docs/Web/CSS/color_value), 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*.

**Drawing Text**

The canvas rendering context provides two methods to render text:

[**fillText(text, x, y [, maxWidth])**](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/fillText)

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

[**strokeText(text, x, y [, maxWidth])**](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/strokeText)

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

function draw() {

var ctx = document.getElementById('canvas').getContext('2d');

ctx.font = '48px serif';

ctx.fillText('Hello world', 10, 50);

}

**Drawing an Image**

create new [HTMLImageElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLImageElement) objects in our script. To do this, you can use the convenient Image() constructor:

var img = new Image(); // Create new img element

img.src = 'myImage.png'; // Set source path

When this script gets executed, the image starts loading.

If you try to call drawImage() before the image has finished loading, it won't do anything (or, in older browsers, may even throw an exception). So you need to be sure to use the load event so you don't try this before the image has loaded:

var img = new Image(); // Create new img element

img.addEventListener('load', function() {

// execute drawImage statements here

}, false);

img.src = 'myImage.png'; // Set source path

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)**](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/drawImage)

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

var img = new Image();

img.onload = function() {

ctx.drawImage(img, 0, 0);

};

img.src = 'https://example-image.jpg';

The drawImage()method places the backdrop at the coordinate (0, 0), which is the top-left corner of the canvas.

They are other methods that extend the use of drawImage() method like Scaling image and Slicing image.

We should avoid this as it might return weird image size and positioning. Instead use styles and width/height properties.

**Animation**

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

[**clearRect()**](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/clearRect)**method sets the pixels in a rectangular area to transparent black (rgba(0,0,0,0)). The rectangle's corner is at (x, y), and its size is specified by width and height.**

clearRect(x, y, width, height);

const canvas = document.getElementById('canvas');

const ctx = canvas.getContext('2d');

ctx.clearRect(0, 0, canvas.width, canvas.height);

.

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.

1. **Using Intervals**

[**setInterval(function, delay)**](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setInterval)

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

[**setTimeout(function, delay)**](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setTimeout)

Executes the function specified by function in delay milliseconds.

[**requestAnimationFrame(callback)**](https://developer.mozilla.org/en-US/docs/Web/API/Window/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.

*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 **requestAnimationFrame()** method 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. The method takes a callback as an argument to be invoked before the repaint.

**Note:** Your callback routine must itself call requestAnimationFrame() if you want to animate another frame at the next repaint.

You should call this method whenever you're ready to update your animation onscreen. This will request that your animation function be called before the browser performs the next repaint.

function animateMe(){

window.requestAnimationFrame(animateMe);

}

You can pass this value to [window.cancelAnimationFrame()](https://developer.mozilla.org/en-US/docs/Web/API/Window/cancelAnimationFrame) to cancel the refresh callback request.

Example of animated ball

<https://codepen.io/pen/?&editable=true>

**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: