Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

A scaled element will grow outwards from or shrink inwards towards its center; in other words, the element’s center will stay in the same place as its dimensions change. To change this default behavior, you can include the transform-origin property, which we’ll be covering a bit later.

Note that you shouldn’t declare a new transform: because of the cascade, a second transform would override the first. To declare multiple transformations, provide a space-separated list of transform functions. We simply add our scale to the end of that space-separated list

It’s also worth remembering that scaling, like translation, has no impact on the document flow. This means that if you scale inline-block elements, text around it will fail to accommodate it with reflowing.

We’re rotating our span by ten degrees clockwise—adding to the effect of text that’s just been dealt a powerful uppercut. We are declaring the rotation before the translate so that it’s applied first—remember that transforms are applied in the order provided. In this case, the span will be rotated 10 degrees, and then moved 40px along the rotated x axis.

Tip: Negative Delays

Interestingly, a negative time delay that’s less than the duration of the entire transition will cause it to start immediately, but it will start partway through the animation. For example, if you have a delay of -500ms on a 2s transition, the transition will start a quarter of the way through, and will last 1.5 seconds. On the way back, it will jump 75% of the way through the transition, and then transition back to the default state. This might be used to create some interesting effects, so it’s worth being aware of.

Captura de pantalla de computadora

Descripción generada automáticamente

How do we know when a transition has ended? A transitionend event—prefixed and camel-cased as webkitTransitionEnd for older mobile browsers—is fired upon completion of a CSS transition in both directions. The event is fired once per direction for each completed transformed property. In the case where the transition fails to complete, such as if you mouseout before our 250ms transition is over in the example above, it will not fire when it has only partially moved and started changing color, but will fire when it returns to the original default state. If you have more than one property being transitioned, the transitionend event will fire multiple times. In our case, it will fire twice when both the color and transform reach their transitioned state, and twice again when color and transform return to the original state.

Captura de pantalla de un celular

Descripción generada automáticamente

The animation-play-state property defines whether the animation is running or paused. A paused animation displays the current state of the animation statically. When a paused animation is resumed, it restarts from the current position. This provides a simple way to control CSS animations from within your CSS or with JavaScript.

CANVAS API

Captura de pantalla de computadora

Descripción generada automáticamente

You may be asking yourself, why not set the width and height via CSS? It’s because the width and height attributes determine how large the canvas’s coordinate system is. If we don’t specify width and height, the canvas element will default to a width of 300 and a height of 150. If we set the width and height for a canvas only in CSS, the canvas element will be 300 by 150, and the CSS properties will simply determine how large the box is that displays the image.

The object that’s returned by getContext is an instance of CanvasRenderingContext2D. In this chapter, we’ll refer to it as simply “the context object” for brevity.

### Filling Our Brush with Color

On a real-life painting canvas, you must first saturate your brush with paint before you can begin. In the HTML5 canvas, you must do the same, and we do so with the strokeStyle or fillStyle properties. Both strokeStyle and fillStyle are set on a context object, and both take one of three values: a string representing a color, a CanvasGradient object, or a CanvasPattern object.

We can use any CSS color value to set the stroke or fill color, as long as we specify it as a string: a hexadecimal value such as #00FFFF, a color name such as red or blue, or an RGB value such as rgb(0, 0, 255). We can even use the property rgba to set a semitransparent stroke or fill color.

Instead of a color as our fillStyle, we could have used a CanvasGradient or a CanvasPattern object. Let’s create a pattern on the second canvas element (whose ID is demo2) on the canvas.html page.

We can also create a CanvasGradient object to use as our fillStyle. To create a CanvasGradient, we call one of two methods: createLinearGradient() or createRadialGradient(); then we add one or more color stops to the gradient.

createLinearGradient ’s x0 and y0 represent the starting location of the gradient. x1 and y1 represent the ending location.

Next, we specify our color stops. The color stop method is simply addColorStop().

The offset is a value between 0 and 1. An offset of 0 is at the start of the gradient, and an offset of 1 is at the end of the gradient. The color is a string value that, as with the fillStyle, can be a color name, a hexadecimal color value, an rgb() value, or an rgba() value.

**Paths** create a blueprint for your lines, arcs, and shapes, but paths are invisible until you give them a stroke! When we drew rectangles, we first set the strokeStyle and then called fillRect. With more complex shapes, we need to take three steps: layout the path, stroke the path, and fill the path. As with drawing rectangles, we can just stroke the path, or fill the path—or we can do both.

Now we need to create an arc. An **arc** is a segment of a circle, but as there’s no method for creating a circle, we can draw a 360° arc. We create it using the arc method:

The signature for the arc method is: arc(x, y, radius, startAngle, endAngle, anticlockwise).

x and y represent where on the canvas you want the arc’s path to begin. Imagine this as the center of the circle that you’ll be drawing. radius is, of course, the distance from the center to the edge of the circle.

startAngle and endAngle represent the start and end angles along the circle’s circumference that you want to draw. The units for the angles are in radians, and a circle is 2π radians. We want to draw a complete circle, so we’ll use 2π for the endAngle. In JavaScript, we can obtain this value by multiplying Math.PI by 2.

Our next step is to close the path, as we’ve now finished drawing our circle. We do that with the closePath method:

By default, the width of the stroke is one pixel, which is stored in the lineWidth property of the context object. Let’s make our border a bit bigger by setting the lineWidth to 3:

If we create an image programmatically using the Canvas API, but decide we’d like to have a local copy of our drawing, we can use the API’s toDataURL method to save our drawing. toDataURL creates a URL with the image in it, (either a png or jpg, whichever is specified). You can then right-click the image at this URL, and save it as a PNG or JPEG.

Captura de pantalla de un celular

Descripción generada automáticamente

Redrawing an image element from the page onto a canvas is fairly unexciting. It’s really no different from using an img element. Where it does become interesting is how we can manipulate an image after we’ve drawn it into the canvas.

Once we’ve drawn an image on the canvas, we can use the getImageData method from the Canvas API to manipulate the pixels of that image. For example, if we wanted to convert our logo from color to black and white, we can do so using methods in the Canvas API.

getImageData will return an ImageData object, which contains three properties: width, height, and data. The first two are self-explanatory; it’s the last one, data, that interests us.

data contains information about the pixels in the ImageData object in the form of an array. Each pixel on the canvas will have four values in the data array, which correspond to that pixel’s R, G, B, and A values. A stands for Alpha, a measure of the element’s transparency, with 0 meaning the element is totally transparent, 1 meaning it’s totally opaque, and 0.5 meaning it’s 50% transparent.

The getImageData method allows us to examine a small section of a canvas, so let’s use this feature to become more familiar with the data array. getImageData takes four parameters, corresponding to the four corners of a rectangular piece of the canvas we’d like to inspect. If we call getImageData on a very small section of the canvas, say context.getImageData(0, 0, 1, 1), we’d be examining just one pixel (the rectangle from 0,0 to 1,1). The array that’s returned is four items long, as it contains a red, green, blue, and alpha value for this lone pixel:

If you want to test pixel manipulation using canvas in Firefox or Chrome, you’ll need to either test it on a web server running on your computer (http://localhost/), or test it online.

Let’s look at how we’d go about using getImageData to convert a full color image into black and white on a canvas. We’ll create a new function in the canvas.js file called manipulateImage to do so.

Assuming that we’ve already placed an image onto the canvas, as we did above, we can use a for loop to iterate through each pixel in the image and change it to grayscale.

First, we’ll call getImageData(0, 0, 200, 200) to retrieve the entire canvas. Then, we’ll grab the red, green, and blue values of each pixel, which appear in the array in that order:

Notice that our for loop is incrementing i by 4 instead of the usual 1. This is because each pixel takes up four values in the imageData array—one number each for the R, G, B, and A values.

Next, we must determine the grayscale value for the current pixel. It turns out that there’s a mathematical formula for converting RGB to grayscale; you simply need to multiply each of the red, green, and blue values by some specific numbers, seen in this code block:

So now we’ve modified our pixel data by individually converting each pixel to grayscale. The final step? Putting the image data we’ve modified back into the canvas via a method called putImageData. This method does exactly what you’d expect: it takes an image's data and writes it onto the canvas. Here’s the method in action:

SVG