Chapter VIII transform and transition.

Translation:

Translation functions allow you to move elements left, right, up, or down. These functions are similar to the behavior of position: relative; when declaring top and left , moving elements up and down or left and right along the x and y axes. When you employ a translation function, you’re moving elements without impacting the flow of the document. Unlike position: relative , which allows you to position an element either against its current position or against a parent or other ancestor, a translated element can only be moved relative to its current position.

The translate(x,y) function moves an element x from the left, and y from the top:

transform: translate(45px, -45px);

translate in older browser

-webkit-transform: translate(45px,-45px); /\* iOS8, Android

↵4.4.3, BB10 \*/

-ms-transform: translate(45px,-45px); /\* IE9 only \*/

transform: translate(45px,-45px);

if you want to move horinzontally and vertically we can use translatex or translatey.

Transforms don’t work on inline elements. But that’s easy enough to fix. We’ll just add display: inline-block; to our element.

Scaling:

Transform: scale (x,y) function scales an element by the defined factors horizontally then vertically. If only one value is provided, it will be used for both the x and y values.

We can also use the scaleX(x) scaleY(y) these function Will scale only horizontally or vertically.

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.

Rotate()

function rotates an element around the point of origin by a specified angle value.

Skew ()

The skew(x,y) function specifies a skew along the x and y axes. As you’d expect, the x specifies the skew on the x axis, and the y specifies the skew on the y axis. If the second parameter is omitted, the skew will only occur on the x axis.

Changing the origin of transform

As we hinted at earlier, you can control the origin from which your transforms are applied. This is done using the transform-origin property. It has the same syntax as the background-position property, and defaults to the center of the object (so that scales and rotations will be around the center of the box by default).

Let’s say that you were transforming a circle. Because the default transform-origin is the center of the circle, applying a rotate() transform to a circle would have no visible effect—a circle rotated 90 degrees still looks exactly the same as it did before being rotated. An ellipse rotated 180 degrees around its center would also look the same as it did before being rotated upside down. However, if you gave your circle or ellipse a transform-origin of 10% 10% or top center.

The order of transform functions does matter: if you rotate before translating, your translate direction will be on the rotated axis. The rightmost square in Figure 8.8 was translated then rotated with transform: translateX(200px) rotate(135deg); . The leftmost square was rotated first then translated along the newly rotated axis: transform: rotate(135deg) translateX(200px);.

Here are the steps to create a simple transition using only CSS:

* Declare the original state of the element in the default style declaration.
* Declare the final state of your transitioned element; for example, a :hover state.
* Include the transition functions in your default style declaration using the transition properties, including: transition-property , transition-duration , transition-timing-function , and transition-delay . We’ll look at each of these and how they work shortly.

You can provide any number of CSS properties to the transition-property declaration, separated by commas. Alternatively, you can use the keyword all to indicate that every supported property should be animated as it transitions.

The transition-duration property sets how long the transition will take: the duration of time it takes to go from the default state to the transitioned state. You can specify this either in seconds ( s ) or milliseconds ( ms ). We’d like our animation to be fairly quick, so we’ll specify 0.2 seconds ( 0.2s ), or 200 milliseconds ( 200ms).

The transition-timing-function Property

The transition-timing-function lets you control the pace of the transition in even more granular detail. Do you want your animation to start off slow and become faster, start off fast and end slower, advance at an even keel, or some other variation?

The most common timing functions include the key terms ease , linear , ease-in , ease-out , or ease-in-out . The default ease has a slow start, then it speeds up, and ends slowly. ease-in-out is similar to ease , but accelerates more sharply at the beginning. linear creates a transition that animates at a constant speed. ease-in creates a transition that is slow to start but gains speed, then stops abruptly. The opposite, ease-out , starts at full speed, then slows progressively as it reaches the conclusion of the transition. The best way to familiarize yourself with them is to play around and try them all. Most often, one will just feel right for the effect you’re aiming to create. It’s helpful to set a relatively long transition-duration when testing timing functions—if it’s too fast, you may not be able to tell the difference.

You can also describe your timing function more precisely by defining your own cubic-bezier function. It accepts four numeric parameters; for example, linear is the same as cubic-bezier(0, 0, 1, 1) . If you’ve studied six years of calculus, the method of writing a cubic Bézier function might make sense; otherwise, it’s likely you’ll want to stick to one of the five basic timing functions. You can also look at online tools that let you play with different values, such ashttp://cubic-bezier.com/, which lets you compare the common key terms against each other or against your own cubic Bézier function. Another document,http://estelle.github.io/animation/files/cubicbezierprint.html, allows you to set the timing function and time to watch it, visualizing how Bézier curves work.

In addition to the predefined timing functions and developer-defined cubic Bézier function, you can divide the transition over equidistant steps. With the steps function, you define the number of steps and the direction of either start or end , where either the first step happens at the animation start, or the last step happens at the animation end respectively. For example, steps(5, start) would jump through the equidistant steps of 0%, 20%, 40%, 60%, and 80%, and steps(5, end) would jump throught the equidistant steps of 20%, 40%, 60%, 80%, and 100%. We will use the steps(n, end) timing function when we animate our bicycle with CSS animation later on in this chapter.

Shortand property:

.ad-ad2 h1 span {

transition-property: transform;

transition-duration: 0.2s;

transition-timing-function: ease-out;

transition-delay: 50ms;

}

Combines all

.ad-ad2 h1 span {

transition: transform 0.2s ease-out 50ms;

}

Keyframes

To animate an element in CSS, you first create a named animation, then attach it to an element in that element’s property declaration block. Animations in themselves don’t do anything; in order to animate an element, you’ll need to associate the animation with that element.

To create an animation, use the @keyframes rule for IE10+ and FF16+. Include @-webkit-keyframes for all WebKit implementations followed by a name of your choosing, which will serve as the identifier for the animation. Then, you can specify your keyframes.

For an animation called myAnimation , the @keyframes rule would look like this:

@-webkit-keyframes myAnimation {

/\* put animation keyframes here \*/

}

@keyframes myAnimation {

/\* put animation keyframes here \*/

}

Examples:

@keyframes moveRight {

from {

transform: translateX(-50%);

}

to {

transform: translateX(50%);

}

}

@keyframes appearDisappear {

0%, 100% {

opacity: 0;

}

20%, 80% {

opacity: 1;

}

}

@keyframes bgMove {

100% {

background-position: 120% 0;

}

}

Animation properties:

Animation-name: appearDisappear;

Animation-duration: 300ms;

animation-timing-function

Like the transition-timing-function property, the animation-timing-function determines how the animation will progress over its duration. The options are the same as for transition-timing-function : ease , linear , ease-in , ease-out , ease-in-out , a developer-defined cubic-bezier() function, step-start , step-end , or a developer-defined number of steps with the steps(number, direction).

More animation:

animation-direction: alternate;

animation-iteration-count: infinite;

animation-delay: 50ms;

animation-name: appearDisappear;

animation-duration: 300ms;

animation-timing-function: ease-in;

animation-iteration-count: 1;

animation-direction: alternate;

animation-delay: 5s;

animation-fill-mode: backwards;

animation-play-state: running;

example:

@keyframes bike {

0% {

background-position: 0 0;

}

100% {

background-position: -360px 0;

}

}

@keyframes move {

0% {

transform: translateX(-100px);

}

100% {

transform: translateX(100px);

}

}

h1:after {

content: '';

width: 90px;

height: 92px;

background-image: url(../images/bike\_sprite.png);

display: block;

margin: auto;

animation:

0.4s steps(4, end) infinite 50ms bike,

8s linear infinite 50ms move;

animation-play-state: paused;

}

h1:hover:after {

animation-play-state: running;

}

Capitulo XII Canvas, SVG, and Drag and Drop

Canvas versus SVG

Now that we’ve learned about canvas and SVG, you may be asking yourself which is the right one to use? The answer is: it depends on what you’re doing.

Both canvas and SVG allow you to draw custom shapes, paths, and fonts. But what’s unique about each?

Canvas allows for pixel manipulation, as we saw when we turned our video from color to black and white. One downside of canvas is that it operates in what’s known asimmediate mode. This means that if you ever want to add more to the canvas, you’re unable to simply add to what’s already there. Every time you finish drawing a shape, the canvas no longer has access to that shape, because it won’t persist as an object that you can modify. So if you want to add to what you’ve already drawn on the canvas, you must redraw the new shape from scratch. Still, canvas does allow you to save the images you create to a PNG or JPEG file.

There’s also no access to what’s drawn on the canvas via the DOM. Because of this, canvas is much faster than SVG (here’s onespeed comparison), and canvas is generally the better choice if you’re looking to design a game requiring lots of animations.

By contrast, what you draw to SVG is accessible via the DOM, because its mode isretained mode, meaning that the structure of the image is preserved in the XML document that describes it. SVG also has, at this time, a more complete set of tools to help you work with it, such as the Raphaël library and Inkscape. However, since SVG is a file format—rather than a set of methods that allows you to dynamically draw on a surface—you can’t manipulate SVG images the way you can manipulate pixels on canvas. It would have been impossible, for example, to use SVG to convert our color video to black and white as we did with canvas.

In summary, if you need to paint pixels to the screen and have no concerns about the ability to retrieve and modify your shapes, canvas is probably the better choice. If, on the other hand, you need to be able to access and change specific aspects of your graphics, SVG might be more appropriate.

Practice:

<https://codepen.io/michell17/pen/WNpgKbm?editors=1010>