

B Academy
RO



Manual

Animation

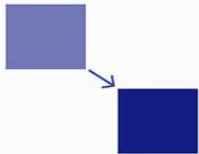
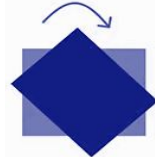


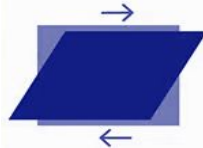
Transform

- Changing the properties of an object **without affecting the document flow.**

You can transform everything except table elements and inline boxes .

Transform

demo

<pre>: translate(x, y)/(-x/-y) : translateX(x) : translateY(y)</pre>	<pre>= translate: x y; = translate: x; = translate: 0 y;</pre>	<pre>% px</pre>	Moving relative to the document flow	
<pre>: rotate(angle)/(-angle) : rotateX(angle) : rotateY(angle)</pre>	<pre>= rotate: angle; = rotate: x angle; = rotate: y angle;</pre>	<pre>deg</pre>	Rotating clockwise or counterclockwise	
<pre>: scale(x, y)/(-x/-y) : scaleX(x) : scaleY(y)</pre>	<pre>= scale: x y; = scale: x; = scale: 0 y;</pre>	<pre>0-10 %</pre>	Scaling up or down. Negative values flip the element	
<pre>: skew(angle, angle)/(-angle) : skewX(angle) : skewY(angle)</pre>		<pre>deg</pre>	Stretching along a specific axis	

translate

Translation allows us to move an item around.

We can use translate to shift an item along in either axis: x moves side to side, y moves up and down. Positive values move down and to the right. Negative values move up and to the left.

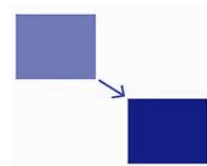
Critically, the item's in-flow position doesn't change. As far as our layout algorithms are concerned, from Flow to Flexbox to Grid, this property has no effect.

This is similar to how top / left / right / bottom work in positioned layout, with relatively-positioned elements.

When we want to move an element along a single axis, we can use translateX and translateY.

When we use a percentage value in translate, that percentage refers to the element's own size, not the available space within the parent container.

Setting transform: translateY(-100%) moves the box up by its exact height, no matter what that height is, to the pixel.



```
transform : translate(x, y);  
: translateX(x)  
: translateY(y)
```

```
translate: x y;  
translate: x;  
translate: 0 y;
```

rotate

rotate will rotate our elements.

We typically use the deg unit for rotation, short for degrees.

Rotating clockwise or counterclockwise



```
transform : rotate(angle)  
: rotateX(angle)  
: rotateY(angle)
```

```
rotate: angle;  
rotate: x angle;  
rotate: y angle;
```

scale

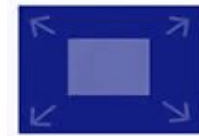
scale allows us to grow or shrink an element.

Scale uses a unitless value that represents a multiple, similar to line-height. `scale(2)` means that the element should be 2x as big as it would normally be.

We can also pass multiple values, to scale the x and y axis independently.

At first glance, this might seem equivalent to setting width and height, but there's one big difference.

The text scales up and down with the element. We aren't just transforming the size and shape of the box, we're transforming the entire element and all of its descendants.



- Scaling up or down.
- Negative values flip the element

```
transform : scale(x, y)  
: scaleX(x)  
: scaleY(y)
```

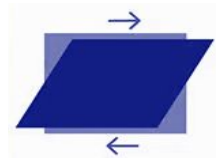
```
scale: x y;  
scale: x;  
scale: 0 y;
```

skew

skew is a seldom-used but pretty-neat transformation.

As with translate, we can skew along either axis.

Skew can be useful for creating diagonal decorative elements.



```
transform : skew(angle, angle)  
: skewX(angle)  
: skewY(angle)
```

Transform rules

Any transformation can be applied to both axes – together or separately	<code>transform: translate(x, y) =</code> <code>transform: translateX(x) translateY(y)</code>
If only one value is specified in the brackets, it applies to both axes	<code>transform: translate(10px) =</code> <code>transform: translateX(10px) translateY(10px)</code>
It's more convenient to use new shorthand properties	<code>translate: x y z;</code>
You can list multiple transformations separated by spaces; order matters , as they will be applied sequentially.	<code>transform: translateX(10px) rotate(10deg)</code> <code>translateY(5px);</code>
The order of individual properties doesn't matter , they will be applied in the established order.	<code>scale: 2; rotate: 15deg; translate: 5px;</code>

Transform: **matrix**

– Combines all the transform values.

- It covers all properties at once.
- Maximum flexibility in settings.
- You might need to recall higher mathematics.
- You will probably never use this.

$$\text{matrix}[1, 2, 3, 4, 5, 6] = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{scale}(a) = \begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{scaleX}(x) = \begin{bmatrix} x & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{scaleY}(y) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{translate}(x, y) = \begin{bmatrix} 0 & 0 & x \\ 0 & 0 & y \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{translateX}(x) = \begin{bmatrix} 0 & 0 & x \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{translateY}(y) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & y \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{skewX}(x) = \begin{bmatrix} 1 & \tan x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{skewY}(y) = \begin{bmatrix} 1 & 0 & 0 \\ \tan y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{rotate}(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Transform-origin

The point relative to which transformations occur.

The transform origin acts as a pivot point!

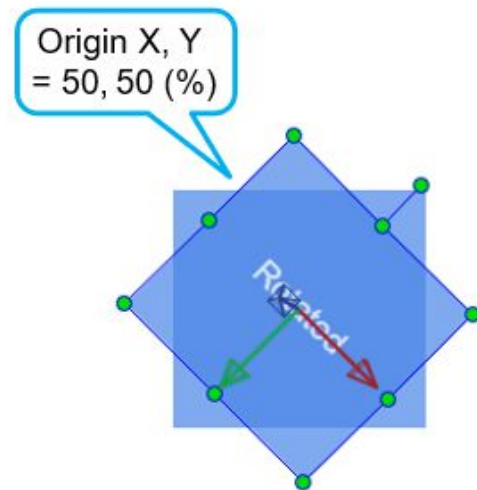
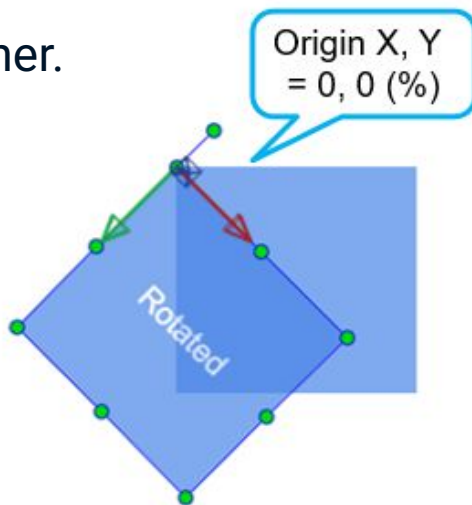
This is useful for certain kinds of effects (for example, an element "growing out of" another one).

By default **(0, 0)** – top-left corner.
center – **(50%, 50%)**

Units:

- px
- %

demo



Reflow and repaint



Any animation on the page involves redrawing some part of the page (in the worst case, the entire page). The larger the area being redrawn, the harder it is to maintain the ideal 60 frames per second, and the animation may appear "choppy."

Whenever possible, instead of animating margins or sizes, animate properties like transform or opacity. Calculating changes to these properties affects smaller areas of the page and is handled by the graphics processor (GPU) rather than the central processor (CPU), which might be busy with other tasks.

Reflow means re-calculating the positions and geometries of elements in the document. The Reflow happens when changes are made to the elements, that affect the layout of the partial or whole page. The Reflow of the element will cause the subsequent reflow of all the child and ancestor elements in the DOM

The Repaint occurs when changes are made to the appearance of the elements that change the visibility, but doesn't affect the layout

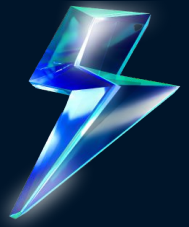
Reflows are very expensive in terms of performance, and is one of the main causes of slow DOM scripts, especially on devices with low processing power, such as phones. In many cases, they are equivalent to laying out the entire page again.

Reflows can be top-down or bottom-up as reflow information is passed to surrounding nodes. Reflows are unavoidable, but you can reduce their impact.

What Causes the Reflows and Repaints:

- Hiding DOM Element with **display: none** will cause both **reflow and repaint**
- Hiding DOM Element with **visibility: hidden** will cause the only **repaint**
- **Moving, animating** a DOM node will trigger **reflow and repaint**
- Activation of CSS pseudo classes such as **:hover**
- **Resizing the window** will trigger **reflow**
- Changing **font-* properties** alters the geometry of the element will need to be a **repaint**
- Content changes, such as a user typing text in an input box
- Adding or removing Stylesheet will cause the reflow/repaint (by js)
- Reflow will happen when Adding, Removing, Updating the DOM nodes (by js)

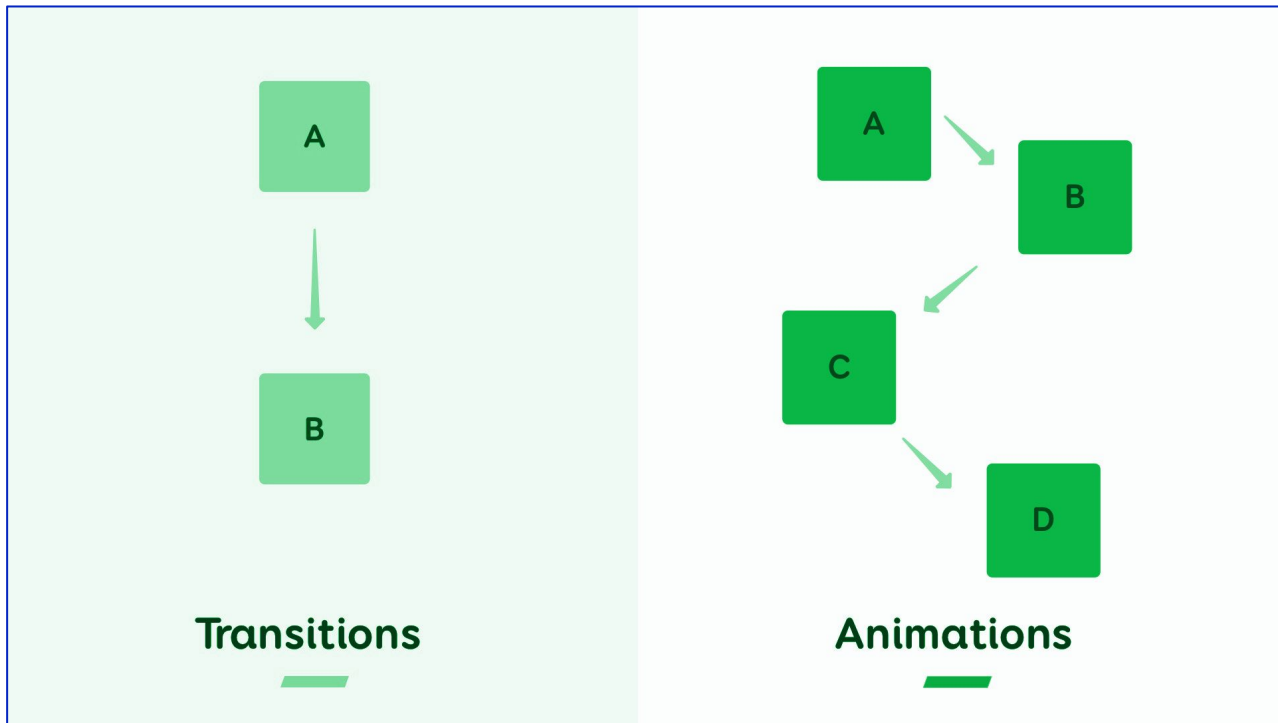
Properties for Changes



CSS Triggers List – What Kind of Changes You Can Make

Cannot be Animated:	Can be Animated, but Better Not to:	Can be Animate	Better to Animate:
display position grid background-image z-index	width height margin padding flex-grow shrink left/right bottom/top Affected document flow	font-size border-width letter-spacing word-spacing	color background-color border-color transform opacity

Ways to Animate an Element



CSS Transition	CSS Animation
A simple way to add animation effects when CSS properties of an element change. For example, you can smoothly change the background color of a button when hovering over it. Transitions require an event to trigger, such as a hover or click.	A more advanced tool for creating complex animations. You can animate almost any property and create complex multi-step animations. Animations work independently of events; for example, they can start automatically when the page loads.

Both methods are great for bringing life to your web pages, but the choice depends on the complexity of the animation you want to create.

VS

CSS Transition	CSS Animation
Triggered by an event (e.g., mouse hover).	Triggered by an event or automatically or controlled via <code>`animation-play-state`</code>
Can have only two frames – start and end.	Can have multiple keyframes .
Changes one or a couple of properties.	Can include many properties.
Has a specific duration (e.g., 1s, 200ms).	Can be infinite or have a set duration .
Suitable for simple effects like color change or movement.	Better for creating complex animations like moving an object along a specific path.
Applicable to any elements (except table elements and inline boxes) and pseudo-elements ::before , ::after	

CSS Transition	CSS Animation
Simple State Changes Example: Smoothly changing the background color of a button on hover	Complex Sequential Changes Example: Rotating an element 360 degrees.
Triggered by State Change (:hover, :focus)	Automatic Start
Limited Capabilities	Control Over Timing and Loops

- **Use `transition`** for simple state changes, where animations are triggered by user interactions like hover or click.
- **Use `animation`** for more complex, multi-step animations that can start automatically or repeat in loops.

Transition

- `transition-property(: all)` – What to animate (color, opacity, etc)
- `transition-duration(: 0s)` – Animation duration (ms, s)
- `transition-timing-function(: ease)` – Animation speed
- `transition-delay(: 0s)` – Delay before animation (ms, s)

```
transition: property duration timing-function delay;
```

Timing functions

– describes how the animation process will be distributed over time (when it speeds up and when it slows down)

Bézier curve

`cubic-bezier(x2, y2, x3, y3)`

Keywords

`ease`, `ease-in`, `ease-out`,
`ease-in-out`, `linear`

Steps

```
steps(  
    the number of steps,  
    jump-start/-end/-none/-both  
)  
step-start = (1, jump-start)  
step-end = (1, jump-end)
```



timing function: cubic-bezier

[demo](#)

[Demo loaders](#)

- Allows you to define a custom animation timing curve.

A Bézier curve is essentially a line from a start point to an end point that is acted upon by one or more control points. A control point curves the line towards it, as if the control point was pulling it in its direction.

cubic-bezier(x1, y1, p1, p2, x2, y2)

- The x-axis represents time: 0 is the starting point, and 1 is the end of the transition-duration.
- The y-axis indicates the completion of the process: 0 is the initial property value, and 1 is the final property value. The control points on the y-axis can have any value: negative or extremely large. In this case, the Bezier curve will jump very high or very low, causing the animation to go beyond its normal limits.

How to create a Bezier curve for a specific task? There are many tools for this.

For example, we can do this on the website cubic-bezier.com.

- This provides the ability to create complex motion paths for objects
- and to set different positions for transitions instead of just two.

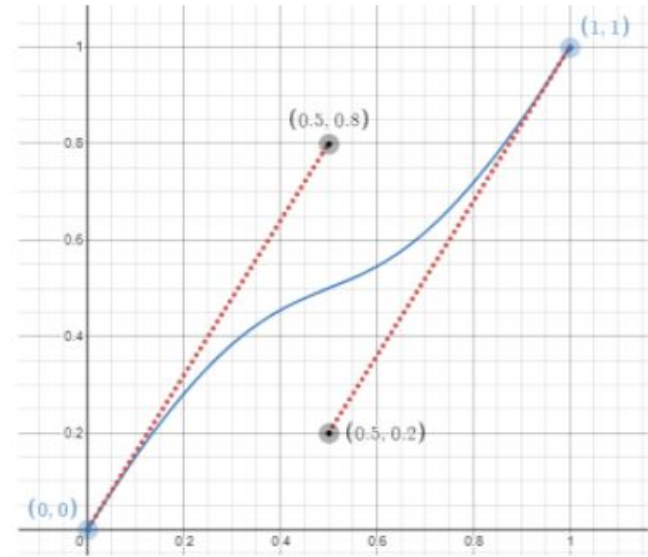
timing function: cubic-bezier

`cubic-bezier(x1, y1, p1, p2, x2, y2)`

`x1, y1` – First coordinate

`x2, y2` – Final coordinate

`p1, p2` – Any other intermediate value pairs



Cubic-bezier(0.5,0.8,0.5,0.2)

timing function: keywords

Demo

There are several built-in Bezier curve options

Uniform speed

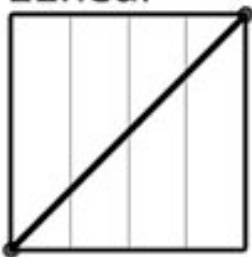
Speeds up in the middle and slows down towards the end (default value).

Starts slow, then speeds up.

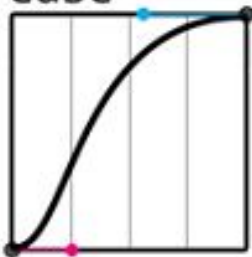
Starts fast, then gradually slows down.

Slow, fast, slow.

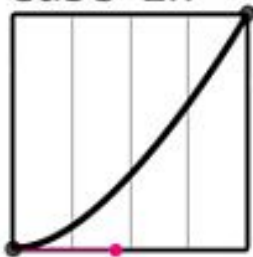
linear



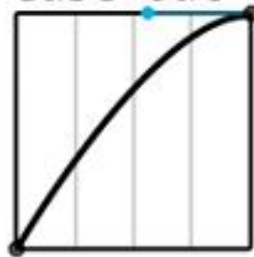
ease



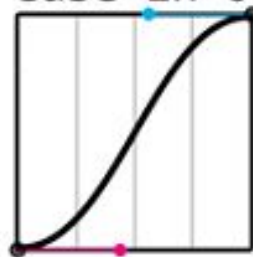
ease-in



ease-out



ease-in-out



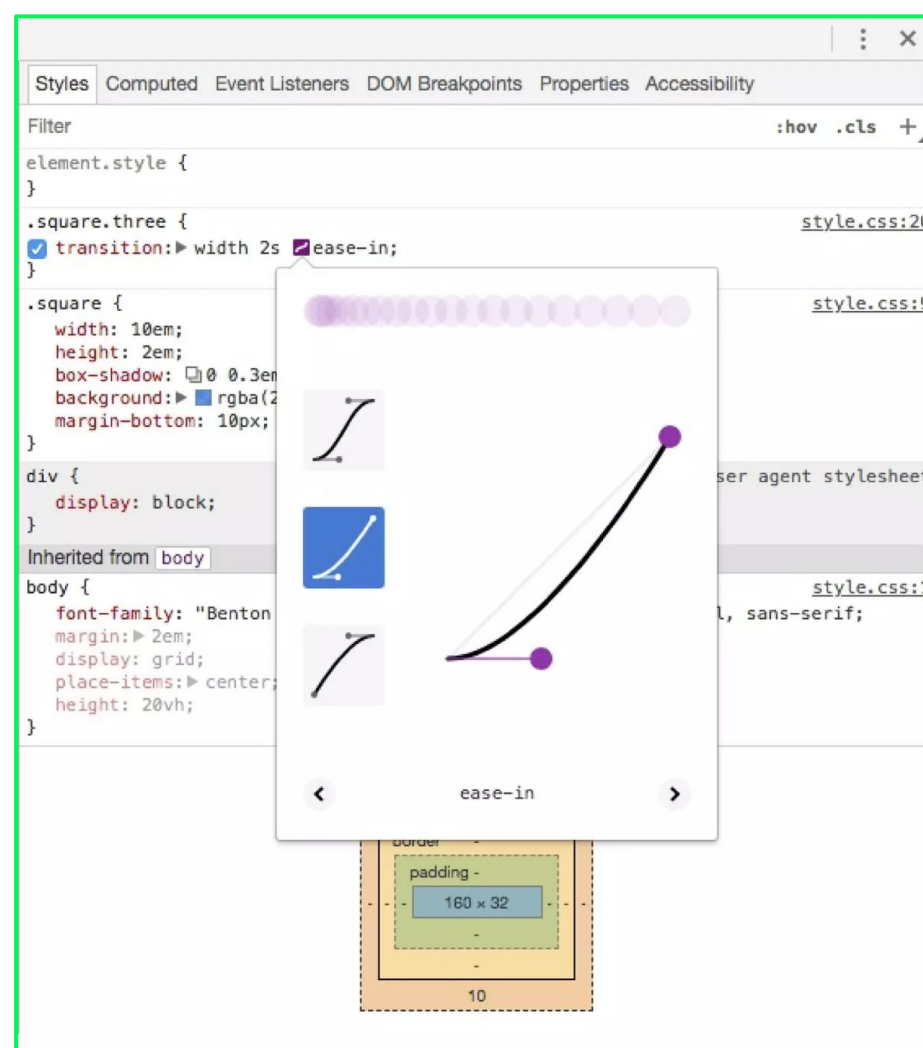
in DevTools

Provides a visual representation of the animation timing

allows you to adjust the curve, including selecting cubic-bezier values.

Browser developer tools also provide support for creating Bezier curves in CSS:

1. Open the developer tools using F12 (Mac: Cmd+Opt+I).
2. Select the "Elements" tab, then look at the "Styles" sub-panel on the right.
3. CSS properties with the word "cubic-bezier" will have an icon next to them.
4. Click the icon to edit the curve.



timing function: **steps**

The animation does not run smoothly but pauses at each step for the same amount of time.



Used together with sequences to create the effect of a moving picture (step or frame animation).

[demo](#)

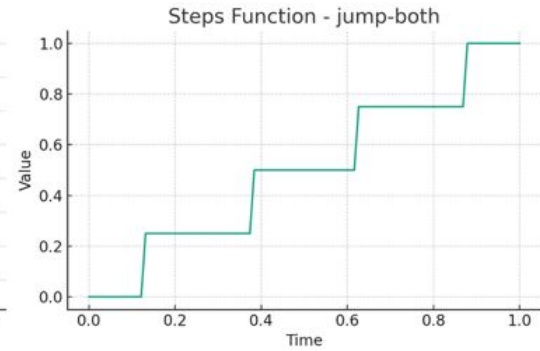
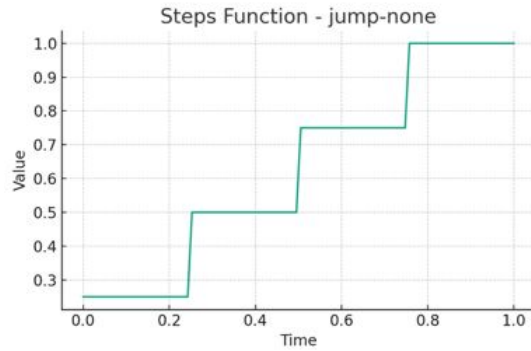
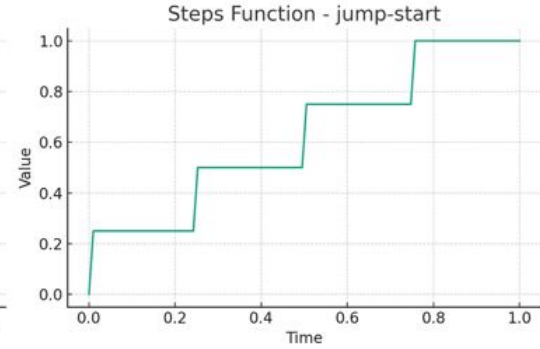
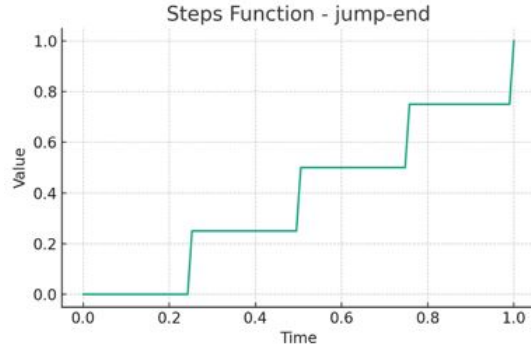
[demo](#)



timing function: values for steps

- **start/jump-start** – The frame changes at the beginning of each second.
- **end/jump-end** – The frame changes at the end of each second.
- **jump-none** – The animation will not change on the first and last steps (-2 frames).
- **jump-both** – The frame sharply changes at the start and end of each step (+2 frames).

```
transition-timing-function: steps(10, jump-start);
```



Demo

`steps(4, *)`

will-change

will-change is a property that allows us to hint to the browser that we're going to animate the selected element, and that it should optimize for this case.

In practice, what this means is that the browser will let the GPU handle this element all the time. No more handing-off between CPU and GPU, no more telltale “snapping into place”.

will-change lets us be intentional about which elements should be hardware-accelerated. Browsers have their own inscrutable logic around this stuff, and I'd rather not leave it up to chance.

- Use it sparingly and only for elements that change frequently.
- It's a good idea to remove this property after the animation is done (can be done using JavaScript).

```
.btn {  
  transition: transform 1s ease;  
  background-color: white;  
  border: 1px solid blue;  
  transform: rotate(15deg);  
}  
  
.btn:hover {  
  will-change: transform;  
}  
  
.btn:active {  
  transform: none;  
}
```

When to use will-change?

Use `will-change` in cases where you know in advance that an element will undergo changes (such as animations or transitions) that involve properties like `transform`, `opacity`, or `top`, which can be expensive for the browser to recalculate.

For example, if you plan to animate an element's `transform` property, specifying it in `will-change` allows the browser to prepare by creating a new rendering layer for that element. This preparation can make the animations run more smoothly, as the browser won't need to recompute layouts or handle reflows during the animation.

key Use Cases:

1. **Smooth animations:** If you're going to animate an element's `transform` or `opacity`, you can tell the browser to optimize those properties in advance to make animations more efficient.
2. **Transitions:** Similarly, for properties that will transition, `will-change` can help reduce jank (lag or frame drops) during the transition.
3. **Performance-critical interactions:** For example, interactive elements like hover effects or dynamically changing content where fast response is crucial.

Warning: Avoid Overuse

While `will-change` can improve performance, overusing it can lead to **memory bloat** because the browser reserves resources (such as creating new layers) when this property is applied. Therefore, it's best to use `will-change` only when necessary and remove it when the change is complete. Avoid applying it to many elements at once or elements that don't need it.

Creating Animation with @keyframes

Sequentially describes the keyframes of the animation.

```
@keyframes animation-name {  
  from {  
    rotate: 0;  
  }  
  to {  
    rotate: 360deg;  
  }  
}
```

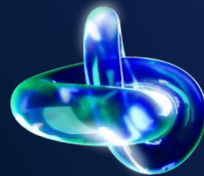
```
@keyframes animation-name {  
  0%, 50% {  
    rotate: 0;  
  }  
  33%, 66% {  
    rotate: 180deg;  
  }  
  100% {  
    rotate: 360deg;  
  }  
}
```

@keyframes



- **Number of frames:** Any number
- **Frames:** from & to or from 0% to 100%
- **Animation name:** Can be any name, but it's better if it describes what the animation does.
- The same animation can be applied to different selectors
- is usually placed at the end of the CSS code.

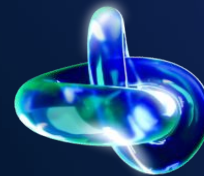
ANIMATION



- **animation-name(: none)** – The name of the animation from @keyframes
- **animation-duration(: 0s)** – Duration of the animation (ms, s)
- **animation-timing-function(: ease)** – Speed of the animation
- **animation-delay(: 0s)** – Delay before the animation starts (ms, s)

```
animation: name duration timing-function delay iteration-count direction  
fill-mode play-state;
```


ANIMATION



- `animation-iteration-count`(: 1) – Number of repetitions (`infinite`, or a specific number from 1)
- `animation-direction`(: `normal`) – Direction of the animation
- `animation-fill-mode`(: `none`) – How styles should be applied before and after the animation
- `animation-play-state`(: `running`) – Controls the running state of the animation (`paused`)

```
animation: name duration timing-function delay iteration-count direction  
fill-mode play-state;
```

animation-**fill-mode**

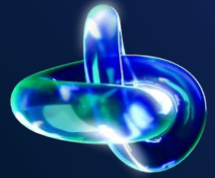
— how styles should be applied before and after the animation
demo

-
- **none** – Animation styles will not be applied to the element before and after it runs.
 - **forwards** – At the end of the animation, the element retains the styles of the last keyframe.
 - **backwards** – The element will have the styles of the first keyframe before the animation starts (useful during **animation-delay**)
 - **both** – The animation will behave as if both **forwards** and **backwards** are applied simultaneously.
-

animation-direction

demo

-
- **normal** – The animation plays from start to finish.
 - **reverse** – The animation plays from finish to start.
 - **alternate** – The animation alternates direction: first from start to finish, then from finish to start.
 - **alternate-reverse** – The animation alternates direction: first from finish to start, then from start to finish.
-



3D transformation & animation

Transform 3D

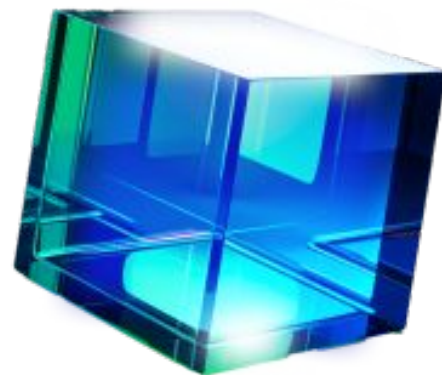
3D animations are a fun and creative way to add depth and movement to your web pages. Imagine your website as a stage, and the elements on it as actors. Usually, these actors can move left, right, up, and down. With 3D animations, they can also move toward you or away from you, and even rotate around their axis, as if they were on a real 3D stage!

transform: translate3d / translateZ = translate: x y z;

transform: rotate3d / rotateZ = rotate: x y z;

transform: matrix3d;

transform: scale3d = scale: x y z;



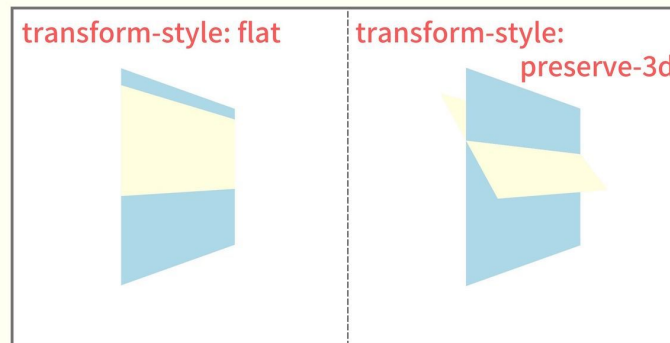
example

Step 1. transform-style

– Defines whether the child elements of a block are placed in 3D space (ensures that 3D effects work).

- **flat** – Child elements of the block lie in the plane of the element itself (default value).
- **preserve-3d** – Child elements of the block should be placed in 3D space.

This is applied to the element being animated.



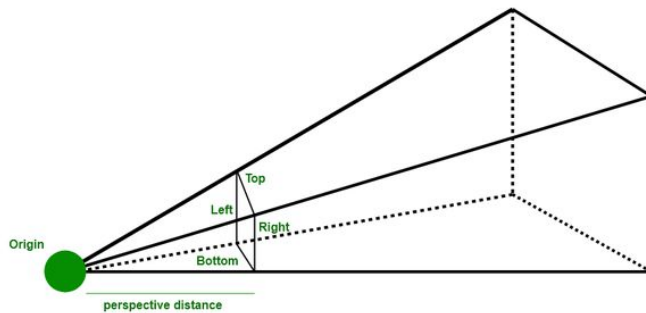
Step 2. perspective

Remember, for the best 3D effect, you often need to set some perspective. You can apply this to the parent element of the one you're transforming.

This is like choosing your seat in a theater. Closer seats make the action more dramatic. In CSS, the perspective property determines how far you are from the $z=0$ plane. A smaller value creates a stronger 3D effect.

- Set on the **parent element** of the transformation.
- Specified in **px**.
- A **smaller** value results in a **more pronounced** 3D effect.

[demo](#)



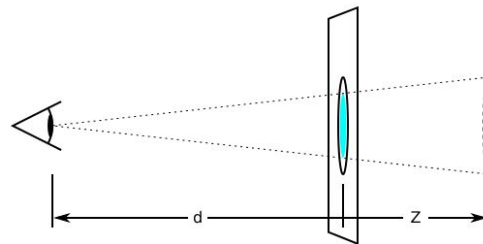
Step 2.1 . perspective-origin

The perspective-origin property in CSS is used alongside perspective to define the "viewpoint" position in 3D space. If you imagine your web page as a window to the world, perspective defines how far you are standing from that window, while perspective-origin defines exactly where you are looking through the window.

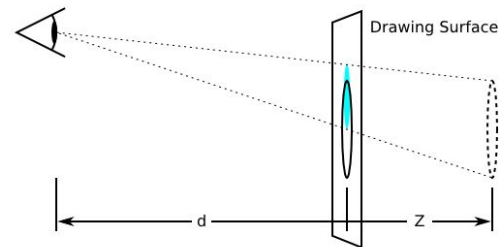
- axes x and y – Horizontal and vertical positions of the viewpoint.
- Specified in **px** and **%**.

Set along with perspective.

[demo](#)



original position of the viewer's eye is at the center



Effect of moving the position of the viewer's eye upwards using perspective-origin

d = distance from the viewer's eye to the screen
Z = position of the element on the z-axis

- **Perspective:** This property determines how strong the 3D effects will appear. A smaller value creates a more dramatic perspective, as if you are getting closer to the objects.
- **Perspective-origin:** This property determines the position from which you, as the viewer, are looking at the 3D objects. It takes two values, specifying the horizontal and vertical position of the viewpoint.

For example, if you set `perspective-origin: 50% 50%`, it means your viewpoint is centered on the element. If objects rotate around their axis, they will rotate around this central point. However, if you change the value to `perspective-origin: 100% 100%`, the viewpoint shifts to the bottom-right corner of the element, and the rotation will happen around that point.

By adjusting `perspective-origin`, you can achieve various interesting visual effects. For instance, if you have a card that flips when hovered over, changing `perspective-origin` will alter the point around which the card rotates, creating the illusion that it flips around different angles.

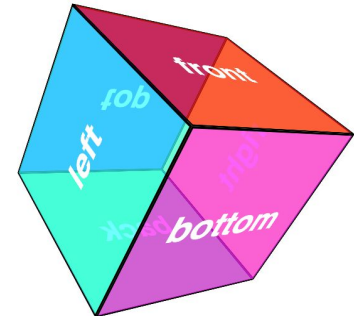
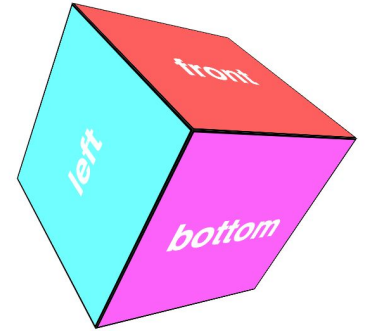
Using `perspective-origin` is especially useful when you want to create more realistic or artistic 3D representations, as it allows you to manipulate the perception of space on the page. Experiment with different values to see how they affect your 3D elements and find the perfect position for your scene!

Step 3. backface-visibility

controls whether the back (reverse) side of an element is visible when it is not facing the user. This is especially useful when creating 3D transformations, such as rotations, where the element can "turn away" from the user and show its back side.

- Useful when an element should be **transparent**.
- **backface-visibility: visible;** (visible): This is the default value. When the element rotates and its back side is facing the user, it remains visible. If the element doesn't have a back background or border, you may see its front side through the back, which can sometimes look strange or unexpected.
- **backface-visibility: hidden;** (hidden): When this property is set, and the element rotates so that its back side is facing the user, that side becomes invisible. This creates the effect as if the element disappears or turns into a flat surface when it's rotated a certain way.

demo



Create a cube

Step 1 . `transform-style`

For element (cube)

Step 2 . `perspective`

Parent element (wrap)

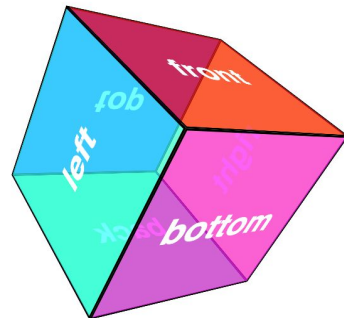
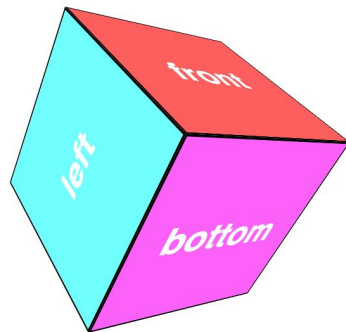
Step 2.1 . `perspective-origin`

Parent element (wrap)

Step 3 . `backface-visibility`

Child element (cube face)

demo



CSS animation

libraries



If the animation is simple,
sometimes it's easier to use
a pre-made option.

animista.net

[Animate.css](https://animate.css)

Demo

Demo

CSS animation

SVG

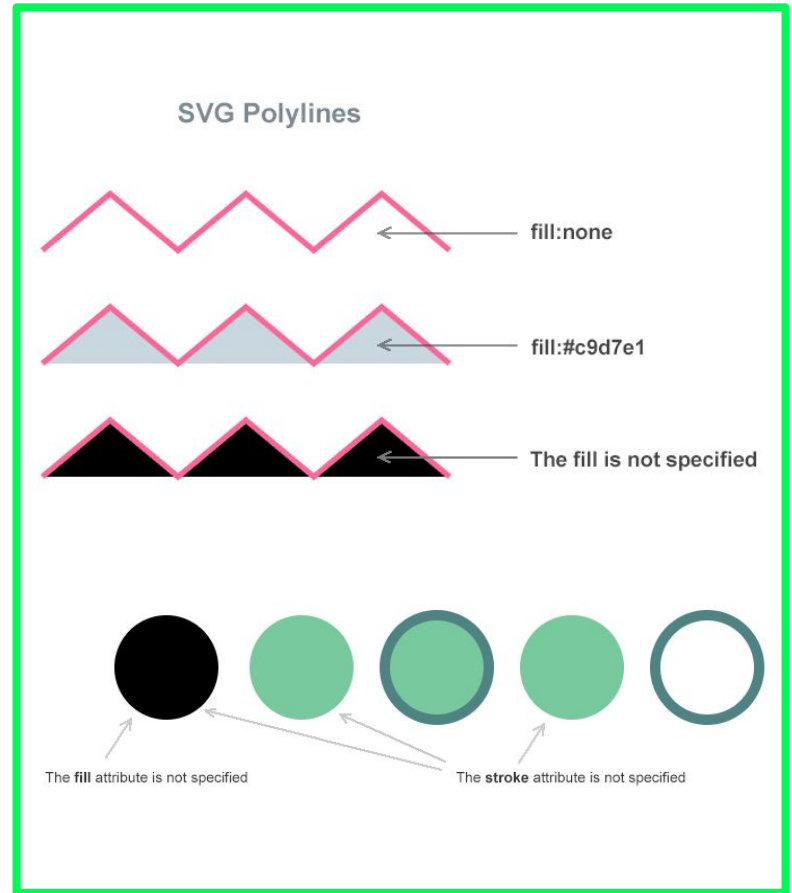
- **SMIL**: Synchronized Multimedia Integration Language
 - Animation is described **inside the SVG** file
 - for drawing along the outline
- **CSS**
 - Changing the **color of an icon** on hover
 - circular preloaders
 - mask, clip-path, linear-gradient, filter
- **libraries**

SVG CSS

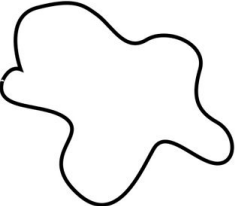
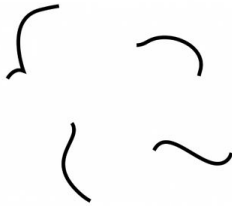
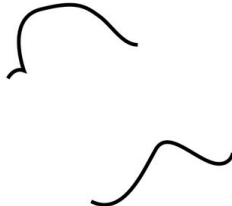
Properties:

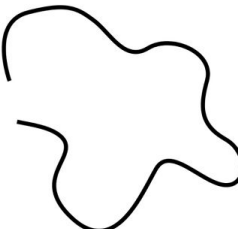


- **fill**
- **fill-opacity**
- **fill-rule**
- **stroke**
- **stroke-width**
- **stroke-opacity**

```
svg: hover { fill: green }
```



Line Animation

원본(1000)	dasharray= 100	dasharray= 200
		

dashoffset= 150	dashoffset= 550	dashoffset= 850
		

- **stroke-dasharray** – Controls the appearance of the dashed stroke.
- **stroke-dashoffset** – Shifts the dashed stroke from its original position.

Works if the shape has a **stroke**.
Ideal for preloaders, progress bars.

step 1

stroke-dasharray

- Value is an array with dash-space pairs
- If one number is specified, the dash equals the space
- The number of pairs can be unlimited and will repeat in a loop.

`stroke-dasharray: 20; without px`

`stroke-dasharray: 20% 10% 30% 10%;`



step 2

stroke-dashoffset

- Moves dashed lines along the contour of the object
- used for line drawing animation.

stroke-dasharray = stroke-dashoffset = path length

stroke-dashoffset: 20; without px
stroke-dashoffset: -20;
stroke-dashoffset: 20%;

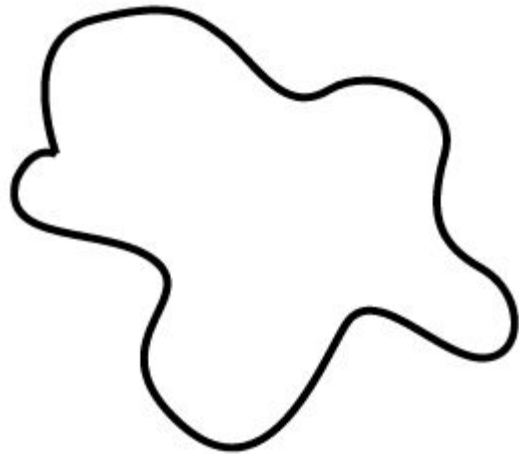


step 3

animation @keyframe

```
.path {  
  stroke-dasharray: 1000;  
  stroke-dashoffset: 1000;  
  animation: dash 5s linear forwards;  
}  
1000 - path length  
  
@keyframes dash {  
  to {  
    stroke-dashoffset: 0;  
  }  
}
```

demo



SVG

libraries

When to Create Something
More Complex Than a
Single Line Animation:

<https://www.svgator.com/>

<https://svgartista.net/>

