# **Advanced topics**

The difference between a site that looks good and one that looks great is attention to detail. After you lay out and style a component of your page, train yourself to slow down and look at it with a critical eye. Does it look better if you increase, or decrease, paddings? Adjust the colors a bit—do they look better a little

darker or little lighter, or a little less vivid? If you’re working from a detailed mockup by the designer, does your implementation match everything as closely as possible? Your designer spent a lot of time on those details. Make sure you’re doing the design justice.

## **Gradients**

[https://developer.mozilla.org/en-US/docs/ Web/CSS/radial-gradient.]

there’s a lot more to explore when it comes to the background property; it’s, in fact, a shorthand for eight properties:

* **background-image**—Specifies an image from a file or a generated color gradient.
* **background-position**—Sets the initial position of the background image.
* **background-size**—Specifies how large to render the background image within the element.
* background-repeat—Determines whether to tile the image if necessary to fill the entire element.
* **background-origin**—Determines whether background positioning is relative to the element’s border-box, padding-box (initial value), or content-box.
* **background-clip**—Specifies whether the background should fill the element’s border-box (initial value), padding-box, or content-box.
* **background-attachment—**Specifies whether the background image will scroll up and down along with the element (the initial value), or if the image will be fixed in place in the viewport. Note that using the value fixed can have negative performance implications on the page.
* **background-color**—Specifies a solid background color. This will render behind any background image.

I tend to prefer using individual properties when I’m doing anything that requires more than a few

of them.

The linear-gradient function has three parameters defining its behavior: angle, starting color, and ending color. The angle here is to right, meaning the gradient starts on the left edge of the element (where it’s white) and blends evenly to the right edge (where it’s blue). You can also use other color syntaxes, such as hex (#0000ff), RGB (rgb(0, 0, 255)), or the **transparent** keyword. You can specify the angle of the gradient in several ways. In this example, you used ***to right***, but you can also use **to top or to bottom.** You can even specify a corner such as **to bottom right**;

For more precise control of the angle, you can use more specific units, such as degrees. The value 0deg ,90deg;

### **Using multiple color stops**

you can define a gradient with multiple colors, which are each called a *color stop;*

|  |
| --- |
| background-image: linear-gradient(90deg, red, white, blue); // **Specifies multiple color stops** |

You can also explicitly set the position of these color stops in the gradient function.

|  |
| --- |
| linear-gradient(90deg, red 0%, white 50%, blue 100%) |

### **STRIPES**

If you place two color stops at the same position, the gradient’s color will instantly switch from one to the

other, rather than a smooth transition;

|  |
| --- |
| **background-image: linear-gradient(90deg, red 40%, white 40%, white 60%, blue 60%);** |

The first color stop is red at 40%, so the gradient is solid red from the left edge all theway to 40%. The next color stop is white, also at 40%, so the gradient makes a hard switch to white. This is followed by another white color stop at 60%, so the gradient is pure white from 40% to 60%. Then the final color stop, also at 60%, is blue. This makes a hard switch to blue, then remains blue all the way to the right edge.

### **REPEATING GRADIENTS**

the technique can be used for some interesting effects. In particular, it can be used with a slightly different gradient function, repeating-linear-gradient(). This works like the regular linear-gradient

function, except the pattern repeats; With repeating gradients, it’s better to use a specific length rather than a percentage,because specified values determine the size of the pattern to repeat.

|  |
| --- |
| **background-image: repeating-linear-gradient(-45deg,#57b, #57b 10px, #148 10px, #148 20px);** |

### **Using radial gradients**

radial gradients start at a single point and proceed outward in all directions; By default, the gradient is centered in the element, transitioning evenly to its corners. It’s elliptical in shape, matching the proportions of the element; It supports all the properties of the linear gradient;

|  |
| --- |
| radial-gradient(circle, white, midnightblue)  …  radial-gradient(circle, midnightblue 0%, white 75%, red 100%) |

## **Shadows**

Two properties that create shadows are box-shadow, which creates a shadow of an element’s box

shape, and text-shadow, which creates a shadow of rendered text.

The declaration ***box-shadow: 1em 1em black*** produces a shadow; The 1em values are the offsets: how far the shadow will be shifted from the element’s position (horizontal, then vertical). If these have a value of 0, then the shadow will be directly behind the element. The value black specifies the color of

the shadow;

box-shadow: horizontal offset (*x*), vertical offset (*y*), and color should always be specified for the shadow. Two other values can optionally be added: a blur radius and a spread radius.

|  |
| --- |
| box-shadow: 2px 2px 2px 1px black; |

The blur radius controls how much the edges of the shadow are to be blurred. This will give it a softer, slightly transparent edge. The spread radius controls the size of the shadow. A positive spread radius makes the shadow larger in all directions; a negative value makes the shadow smaller.

### **Defining depth with gradients and shadows**

Let’s use gradients and shadows to build a button as shown in figure 11.10. A top-to bottom gradient gives it a curved, 3-D appearance. The shadow enhances this effect. In this example, you’ll also use the :**active** pseudo-class to create an alternate type of shading when the button is depressed.

|  |
| --- |
| .button {  padding: 1em;  border: 0;  font-size: 0.8rem;  color: white;  border-radius: 0.5em;  background-image: linear-gradient(to bottom, #57b, #148);  box-shadow: 0.1em 0.1em 0.5em #124;  }  .button:active {  box-shadow: inset 0 0 0.5em #124,  inset 0 0.5em 1em rgba(0,0,0,0.4);  } |

I’ve done two new things here. Instead of a normal box shadow, I’ve added the inset keyword. This makes the shadow appear inside the border of the element, rather than outside. I’ve also added more than one shadow definition, separating them with a comma. Multiple shadows can be added in this way.

The first inset shadow (inset 0 0 0.5em #124) has offsets of zero and a slight blur. This adds a ring of shade inside the edges of the element. The second (inset 0 0.5em 1em rgba(0,0,0,0.4)) has a bit of vertical offset, making the shadow more prevalent along the top of the button. The RGBA color definition defines a semi transparent black.

## **Creating elements with a flat design**

Flat design embraces the modern world’s digital nature. It emphasizes vivid, uniform colors and a simpler

appearance. This means fewer gradients, shadows, and rounded corners. Ironically, this trend emerged only after these long-awaited effects arrived in CSS.

|  |
| --- |
| .button {  padding: 1em;  border: 0;  color: white;  background-color: #57b;  font-size: 1rem;  padding: 0.8em;  box-shadow: 0 0.2em 0.2em rgba(0, 0, 0, 0.15);  }  .button:hover {  background-color: #456ab6;  }  .button:active {  background-color: #148;  } |

### **Creating buttons with a more modern look**

This is still a minimal design, but it has a thick border along the bottom, giving it an appearance much like the front side of a 3-D cube-like object. This dark line is in fact not a border at all, but rather a box-shadow with no blur, which allows for the curved edges mirroring the border radius. In the active state, the button is shifted down by a few pixels. This produces the illusion of it moving into the page as it’s pressed.

|  |
| --- |
| .button {  padding: 0.8em;  border: 0;  font-size: 1rem;  color: white;  border-radius: 0.5em;  background-color: #57b;  box-shadow: 0 0.4em #148;  text-shadow: 1px 1px #148;  }  .button:active {  background-color: #456ab5;  transform: translateY(0.1em);  box-shadow: 0 0.3em #148;  } |

## **Blend modes**

The majority of the time, an element will have only one background image, whether an actual image or a gradient. But there are instances where you might want two or more backgrounds. CSS allows for this.

The background image property accepts any number of values, each separated by a comma:

background-image: url(bear.jpg), linear-gradient(to bottom, #57b, #148); When you apply multiple background images, those listed first render in front of those listed afterward;

**NOTE If a background image has some transparency, other backgrounds behind it will show through the transparent areas, even without the use of blend modes;**

|  |
| --- |
| .blend {  min-height: 400px;  background-image: url(images/bear.jpg), url(images/bear.jpg);  background-size: cover;  background-repeat: no-repeat;  background-position: -30vw, 30vw;  background-blend-mode: multiply;  } |

Most of these background properties can accept multiple values, separated by a comma. The background position has two such values. The first will apply to the first image, the second to the second image. The background-size and backgroundrepeat properties also accept multiple values, but by specifying only one, that value is applied to both background images. The min-height property is included to ensure that the element doesn’t collapse to a height of zero (because it’s empty).

The background-size property accepts two special keyword values, cover and contain. Using cover resizes the background image so it fills the entire element; this can result in edges of the image being clipped. Using contain ensures the entire image is visible, though this may result in some of the element not being covered by the background (a “letterboxing” effect). This property also accepts length values to explicitly set the height and width of the element.

### **Tinting an image**

You can use a blend mode to take a full-color image and tint it with a single hue. To illustrate, you’ll take the bear image and color it blue, A background-blend-mode merges not only multiple background images, but also the background-color. These stacked layers are then combined by the blend mode, so you can set a background color to the desired hue;

|  |
| --- |
| .blend {  min-height: 400px;  background-image: url("images/bear.jpg");  background-color: #148;  background-size: cover;  background-repeat: no-repeat;  background-position: center;  **background-blend-mode: luminosity; //Uses luminosity blend mode**  } |

The luminosity blend mode takes the luminosity from the front layer (the bear image) and blends it with the hue and saturation from the back layer (the blue background color).

If, for instance, you were to place a blue layer in front of the bear image rather than behind (using a gradient instead of a background color), the result would be different;

### **Understanding types of blend mode**

CSS supports 15 blend modes. Each uses a different mathematical formula to control how images are blended. For every pixel, the color of one layer is combined with the corresponding color in the other, resulting in a new pixel color for the composite image.

They each fall into one of five types: **darkening, lightening, contrasting, compositing, or comparing.**

|  |
| --- |
| Darken  **multiply The lighter the front color, the more the base color will show through.**  **darken Selects darker of the two colors.**  **color-burn Darkens the base color, increasing contrast.** |
| Lighten  **screen The darker the front color, the more the base color will show through.**  **lighten Selects the lighter of the colors.**  **color-dodge Lightens base color, decreasing contrast.** |
| Contrast  **overlay Increases contrast by applying multiply to dark colors and screen to light colors, at half strength.**  **hard-light Greatly increases contrast. Like overlay, but applies multiply or screen at full strength.**  **soft-light Similar to hard-light, but uses burn/dodge instead of multiply/screen.** |
| Composite  **hue Applies hue from the top color onto the bottom color.**  **saturation Applies saturation from the top color onto the bottom color.**  **luminosity Applies luminosity from the top color onto the bottom color.**  **color Applies hue and saturation from the top color onto the bottom color.** |
| **Comparative**  difference Subtracts the darker color from the lighter one.  exclusion Similar to difference, with less contrast |

### **Adding texture to an image**

Another application of blend modes is to add texture to an image. You may have a clear, modern image, but sometimes you want it to appear differently for stylistic reasons. You can use a grayscale image to artificially add film grain or some other texture to the image.

**This sort of effect can be achieved with one of the contrast blend modes: overlay, hard-light, or soft-light;**

## **Using mix blend modes**

|  |
| --- |
| .blend {  background-image: url("images/bear.jpg");  background-size: cover;  background-position: center;  padding: 5em 0 10em;  }  .blend > h1 {  margin: 0;  font-family: Helvetica, Arial, sans-serif;  font-size: 6rem;  text-align: center;  mix-blend-mode: hard-light;  background-color: #c33;  color: #808080;  border: 0.1em solid #ccc;  border-width: 0.1em 0;  } |

## **Summary**

* Use gradients and shadows to add the appearance of depth to the page.
* Even basic flat designs can benefit from some subtle shadows or gradients.
* Use gradients with explicit color stops to add stripes to an element.
* A subtle background gradient rather than a flat color provides a little more complexity to the design.
* Use blend modes to colorize or texture an image.

# **Transitions**

The web is a living medium, where we can do so much more. Elements can fade out. Menus can slide in. Colors can shift from one value to another, and the easiest way to do any of these is with *transitions*.

## **From here to there**

Transitions are done with the **transition-\* family of properties**. If these are applied to an element while one of its property values changes, then that property will transition instead of changing immediately to the new value;

|  |
| --- |
| button {  background-color: hsl(180, 50%, 50%);  border: 0;  color: white;  font-size: 1rem;  padding: .3em 1em;  transition-property: all;  transition-duration: 0.5s;  }  button:hover {  background-color: hsl(0, 50%, 50%);  border-radius: 1em;  } |

The **transition-property** property specifies which properties to transition. In this case, the special keyword all means to transition any properties that change. The transition-duration property indicates how long the transition will take before reaching the final value; A transition takes place any time a property on this element is changed. This can occur on a state change like :**hover;**

You can also use the shorthand property, transition. The shorthand accepts up to **four** values for the four transition properties: **transition-property, transition-duration, transition-timing-function, and transition-delay;**

|  |
| --- |
|  |

* The first value specifies which properties to transition. you to need to transition to only apply to one property, specify that property here;
* The third value is the timing function. This controls how the intermediate values of the property are computed, effectively controlling how the rate of change accelerates or decelerates throughout the transition effect. This is either a **linear or ease-in;**
* the delay, allows you to specify a waiting period before the transition begins to take effect after the property value changes.

If you need to apply two different transitions to two different properties, you can do that by adding multiple transition rules, each separated by a comma;

|  |
| --- |
| **transition: border-radius 0.3s linear, background-color 0.6s ease;** |

## **Timing functions**

The transition makes a property value “move” from one value to another; the timing function says *how* it moves. Does it move at a steady speed? Does it start slowly and accelerate? You can use several keyword values, such as **linear, ease-in, and ease-out**, to define this movement. With a linear transition, the value changes at a constant rate. With ease-in, the rate of change starts out slow, but accelerates until the end of the transition. Ease-out decelerates, starting with a rapid change and ending slowly;

### **Understanding Bézier curves**

Timing functions are based on mathematically defined Bézier curves. The browser uses these curves to calculate a property’s value as a function of change over time.

These curves **begin at the bottom left and proceed to the top right**. Time will progress to the right, and the curve represents how the value changes during that progression before arriving at the final value. The linear timing function is a steady progression throughout the duration of the transition—a straight line. The other values curve, representing acceleration and deceleration.

You’re not limited to these five keyword values, however. You can define your own cubic Bézier curve for more gentle or more drastic transitions. You can even add a bit of a “bounce” effect.

**[ inspect the green box element. You should see a small symbol beside the timing function in the Styles pane (Chrome). Click that symbol and a small popup opens, allowing you to modify the timing function’s curve. ]**

### **Steps**

One last type of timing function uses the steps() function; It moves it in a number of discrete, instant “steps.”; The function takes two parameters: the number of steps and a keyword (either start or end), indicating whether each change should take place at the start or end of each step.

## **Non-animatable properties**

If JavaScript changes something on the page, you might want to consider whether adding a transition is appropriate. In some cases, this is as simple as adding a transition property to that element. In other cases it can require a little more arrangements;

**I used a transition duration of 0.2 seconds for these hover effects. As a rule of thumb, most of your transitions should be somewhere between 200 and 500 ms. Any longer, and users will perceive your page as slow and feel as if they are waiting unnecessarily for the page to respond.**

## **Properties that cannot be animated**

Not all properties can be animated. The display property is one of these. You can only toggle it between display: none and display: block; you can’t transition between values, so any transition properties applied to display are ignored. If you look up a property in a reference guide such as MDN (<https://developer.mozilla.org/en-US/>);

**NOTE** Most properties that accept a length, number, color, or the function calc() can be animated. Most properties that take a keyword or other discrete values, like url(), **can’t.**

### **Fading in and out**

Next, let’s use an opacity transition to give our menu a fade in and fade out effect as it opens and closes. The **opacity** property can be any value between 0 (fully invisible) and 1 (fully opaque);

Now the menu fades in and out as you open and close it. Unfortunately, the menu isn’t gone when it’s closed—it’s fully transparent, but it’s still present on the page. And, if you try to click the Read more link, it won’t work. Instead, you’ll click the transparent menu item in front of it, taking you to the Features page.

You need to transition the opacity, but also fully remove the menu drawer when it’s not visible. You can do this with the help of another property, visibility.

The **visibility** property lets you remove an element from the page, similar to the display property. You can give it the values visible or hidden. Unlike display, visibility is animatable.

|  |
| --- |
| .dropdown\_\_drawer {  position: absolute;  background-color: white;  width: 10em;  opacity: 0;  visibility: hidden;  transition: opacity 0.2s linear,  visibility 0s linear 0.2s;  }  .dropdown.is-open .dropdown\_\_drawer {  opacity: 1;  visibility: visible;  transition-delay: 0s;  } |

**TIP You can use the JavaScript transitionend event to perform an action after a transition completes.**

## Summary

* You can use transitions to smooth sudden changes in the page.
* To catch the user’s attention, use an accelerating motion.
* To show the user that their action has taken effect, use a decelerating motion.
* You can use JavaScript to coordinate transitions with class name changes when
* CSS alone cannot do what you need.

# **Transforms**

## **Rotate, translate, scale, and skew**

The transform function rotate() specifies how the element is to be transformed. You’ll find several other transform functions, but they generally all fall into one of four categories (illustrated in figure 15.1).

* *Rotate*—Spins the element a certain number of degrees around an axis
* *Translate*—Moves the element left, right, up, or down (similar to relative positioning)
* *Scale*—Shrinks or expands the element
* *Skew*—Distorts the shape of the element, sliding its top edge in one direction and its bottom edge in the opposite direction;

skew(20deg)—Skews the card 20 degrees. Try a negative angle to skew in the other direction. scale(0.5)—Shrinks the card to half of its initial size. The scale() function takes a unitless number. Values less than 1 shrink the element; values greater than 1 expand it. translate(20px, 40px)—Shifts the element 20 pixels right and 40 pixels down. Again, you can use negative values to transform in the opposite direction.

when rotating an element, a corner of it may shift off the edge of the screen. Similarly, it could potentially cover up portions of another element beside it;

**Transforming one element doesn’t cause other elements to move, so they might overlap.**

**Transforming one element doesn’t cause other elements to move, so they might overlap.**

### **Changing the transform origin**

A transform is made around a *point of origin*. This point serves as the **axis of rotation**, or the spot where scaling or skewing begins. This means the origin point of the element remains locked in place, while the rest of the element transforms around it; By default, the point of origin is the center of the element, but you can change this with the transform-origin property;

|  |
| --- |
| **transform-origin: right center;**  **transform-origin: 100% 50%**; |

NOTE You can also use a length to specify the origin in pixels, ems, or another unit. Though, in my experience, the **keywords top, right, bottom, left, and center are all you’ll need in most instances**;

### **Applying multiple transforms**

You can specify multiple values for the transform property, each separated by a space. Each transform value is applied in succession from right to left.

transform: rotate(15deg) translate(15px, 0), the element is translated 15 px to the right,

then rotated 15 degrees clockwise.

I generally find it’s easier to do translate() manipulations last chronologically (first in source order for transform);

## **Transforms in motion**

Transforms by themselves aren’t all that practical. A box with a skew() applied may look interesting, but it’s not exactly easy to read. But when used in conjunction with motion, transforms become much more useful;

You’ll keep the menu like this on smaller viewports. But on larger screens, you can layer on more effects. For the desktop layout, you’ll dock it to the left-hand side of the screen using fixed positioning.

The **position: fixed** declaration puts the menu into place and keeps it there, even as the page scrolls. The display: block rule overrides the display: flex from the mobile styles, causing the menu items to stack atop one another;

### **Scaling up the icon**

|  |
| --- |
| .nav-links\_\_icon {  transition: transform 0.2s ease-out;  }  .nav-links a:hover > .nav-links\_\_icon,  .nav-links a:focus > .nav-links\_\_icon {  transform: scale(1.3);  } |

### **Creating “fly in” labels**

The menu labels don’t necessarily need to be visible at all times. You can hide them by default, leaving the icons in place to indicate to the user that the menu is there;

|  |
| --- |
| .nav-links\_\_label {  display: inline-block;  margin-left: 1em;  padding-right: 1em;  opacity: 0;  transform: translate(-1em);  transition: transform 0.4s cubic-bezier(0.2, 0.9, 0.3, 1.3), opacity 0.4s linear;  }  .nav-links:hover .nav-links\_\_label,  .nav-links a:focus > .nav-links\_\_label {  opacity: 1;  transform: translate(0);  } |

### **Staggering the transitions**

The menu looks pretty good at this point. Let’s make one last tweak to make it feel polished. You’ll use the transition-delay property to set a slightly different delay for each menu item. This’ll stagger the animations so they fly in a rolling “wave” rather than all at once;

## **Animation performance**

If you’re doing any sort of transition or animation, you should always favor a transform over positioning or explicit sizing if you can. To understand why this is, we need to look closer at how the page is rendered in the browser;

After the browser computes which styles apply to which elements on the page, it needs to translate those styles into pixels on the screen. This is the process of *rendering*, which can be broken down into three stages: layout, paint, and composite.

* **LAYOUT:** In the first stage, ***layout*, the browser calculates how much space each element is going to take on the screen;**
* **PAINT:** This is the process of filling in pixels: t**ext is drawn; images and borders and shadows are all colored**.
* COMPOSITE: **browser takes all of the layers that have been painted and draws them into the final image that’ll be displayed onscreen**.

## **Three-dimensional (3D) transforms**

Rotation and translation can be performed in all three dimensions: X, Y, and Z; You can also translate on the **Z dimension using translateZ(), which moves an element conceptually closer to or further from the user.**

### **Controlling perspective**

Before you add 3D transforms; you need to specify one more thing—*perspective*. You can think of perspective as the distance between the “camera” and the scene. Moving the camera around changes the way the scene appears in the final image. If the camera is close (that is, the perspective is small), then the 3D effects are much stronger. If the camera is far away (that is, the perspective is large), then the 3D effects are much subtler;

You can specify this perspective distance in two ways: using a **perspective()**transform or using the perspective property. Each behaves **a little differently.**

In this example, each box looks the same. Each has its own perspective, applied using the perspective() function. This method applies a perspective to a single element; in this example, you’ve applied it directly to each box. It’s as if four separate pictures were taken of each element, each from the same position.

|  |
| --- |
| **transform: perspective(200px) rotateX(30deg);** |

Sometimes you’ll want multiple elements to share a common perspective, as if they all exist within the same 3D space.

|  |
| --- |
| **.row {**  **perspective: 200px; // Adds the perspective to the container**  **}**  **.box {**  **transform: rotateX(30deg);**  **}** |

By applying one common perspective to the parent (or other ancestor) container, all the elements within the parent that have 3D transforms applied will share that perspective;

### **Implementing advanced 3D transforms**

### PERSPECTIVE-ORIGIN PROPERTY

By default, the perspective is rendered as if the viewer (or camera) is positioned directly in front of the center of the element. The perspective-origin property shifts the camera position left or right and up or down.

|  |
| --- |
| perspective: 200px;  perspective-origin: left bottom |

### **BACKFACE-VISIBILITY PROPERTY**

If you use rotateX() or rotateY() to spin an element more than 90 degrees, something interesting happens: the “face” of the element is no longer directed toward you. This is the backface of the element. By default, the backface is visible, but you can change this by applying backface-visibility: hidden to the element.

## Summary

* Use transforms to scale, rotate, translate, and skew elements in two and three dimensions.
* Transforms are essential for performant transitions and animations.
* Understand how the rendering pipeline works and keep it in mind when building animations.
* To use a custom timing function curve to add a bounce effect to transitions.

# **Animations**

A *keyframe* refers to a specific point in an animation. You define some number of keyframes, and the browser fills in, or interpolates, all the frames in between;

## **Keyframes**

Animations in CSS contain two parts: the @keyframes at-rule, which defines an animation, and the animation property, which applies that animation to an element.

A keyframe animation needs a name; this example defines an animation named overand-back. It then defines three keyframes using percentages. These percentages indicate when in the animation each keyframe occurs: one at the beginning of the animation (0%), one in the middle (50%), and one at the end (100%). The declarations inside each of these blocks define how that keyframe appears.