Safari Visual Effects Guide

Internet & Web

2008-11-19

Contents

Figures and Listings

Introduction

Safari visual effects allow you to create innovative web content including interactive web applications for the desktop and iPhone OS. These CSS and the DOM visual effects extensions allow you to add 2D and 3D graphics to your web applications, create complex animations, and generate smooth transitions when element properties—such as position or color—change. When you combine these visual effects with DOM mouse and touch events, you can create interactive applications, similar to native applications, that run in Safari.

Some of the properties and classes described in this book are proposed W3C standards or Apple extensions. Properties in CSS that begin with `-webkit` are usually proposed standards—the `-webkit` prefix will be dropped when they are approved. Similarly, JavaScript classes that begin with `WebKit` are also proposed standards or Apple extensions.

Important: Not all CSS properties and JavaScript classes described in this book are supported on all Safari platforms. Refer to the respective reference documents, *Safari CSS Reference* and *Safari DOM Extensions Reference* for support level and availability details.

Who Should Read This Document

You should read this document if you want to use visual effects in your web content and web applications—if you are creating web applications for either Safari on the desktop or iPhone OS. Definitely read this document if you are creating a custom web application for iPhone.

Organization of This Document

The articles in this book are as follows:

- "Transitions" (page 9) explains how to add transitions to your content that animate property value changes.
- "Animations" (page 15) covers the basics on how to add animations to your content, including creating keyframes and setting timing functions.
- "Transforms" (page 23) describes how to add 2D and 3D graphics effects to your content.
- "Interactive Visual Effects" (page 33) provides step-by-step instructions on how to combine DOM events with visual effects to create interactive web applications.

See Also

There are a variety of other resources for Safari web content developers in the ADC Reference Library.

For details on properties and classes discussed in this book, refer to these visual effects reference documents:

- *Safari DOM Extensions Reference* describes the touch event classes that you use to handle multi-touch gestures in JavaScript.
- *Safari CSS Reference* describes the CSS properties supported by various Safari and WebKit applications.

If you are creating content for Safari on iPhone, also read these documents:

- *Safari Web Content Guide for iPhone OS* describes how to create content that is compatible with, optimized for, and customized for iPhone OS.
- *iPhone Human Interface Guidelines for Web Applications* provides user interface guidelines for designing webpages and web applications for Safari on iPhone.

If you want to use the JavaScript media APIs, then you should read these documents:

- *JavaScript Scripting Guide for QuickTime* describes how to use JavaScript to query and control the QuickTime plug-in directly.

If you want to read the WebKit W3C proposals, then go to http://www.webkit.org/specs.

Transitions

Transitions are implicit animations that occur when you change an animatable CSS property. Normally when the value of a CSS property changes, the affected elements are rendered immediately using the new property value. Using CSS transitions, the element automatically transitions from its current state to its new state when you set the property value.

To create a transition, you first create a style that sets up a transition—for example, you specify the element's property that triggers the transition and the duration of the transition. Then you apply the style to an element and set the element's property value. Changing the property value starts the transition from the old to the new value.

Not all element properties are animatable. In general, any CSS property that accepts values that are numbers, lengths, percentages, or colors is animatable. Some CSS properties that accept discrete values, such as the `visibility` property, are animatable too, but they display a jump between values rather than a smooth transition when changed. See *Safari CSS Reference* for which properties are animatable.

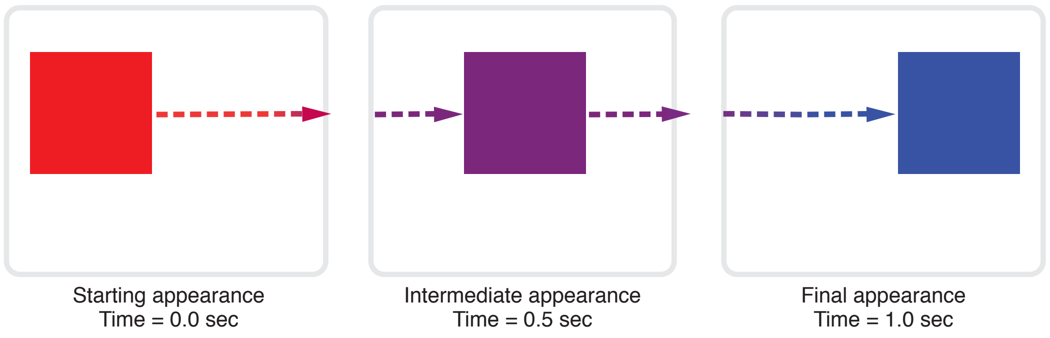
Note: There are actually two types of animations: declared animations and implicit animations. This article describes implicit animations, which are called **transitions**. Read "Animations" (page 15) for details on declared animations.

How Transitions Work

Normally when the value of a CSS property changes, the result is rendered immediately. When applying a transition to an element's property, the change animates smoothly from the old value to the new value over time. The property values are computed over time. Therefore, if you query the value of a property during a transition, you may get an intermediate value that is the current animated value, not the old or new value.

For example, suppose you define a transition for the `left` and `background-color` properties and then set both property values of an element at the same time. The element's old position and color transitions to the new position and color over time as illustrated in Figure 1. If you query the properties in the middle of the transition, you get an intermediate location and color for the element.

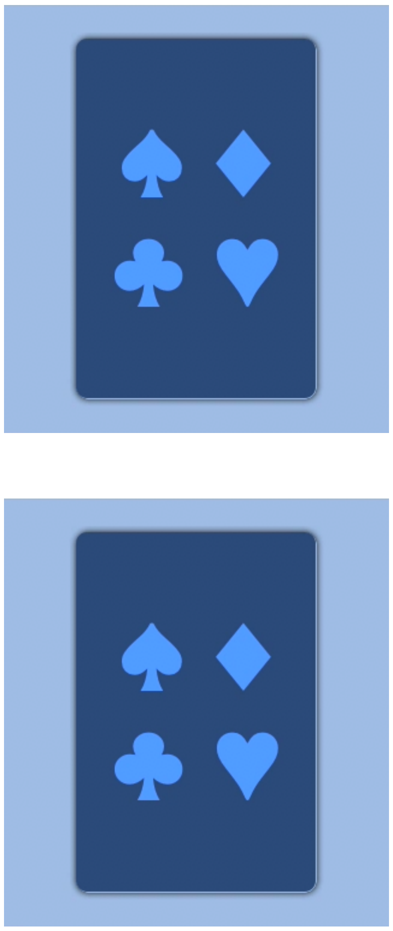
Figure 1 Transition of two properties



You can also control how these intermediate values are computed over time using a timing function. Read "Using Timing Functions" (page 11) to learn more about timing functions.

Transitions are powerful if you combine them with 2D and 3D transforms. For example, Figure 2 shows the results of applying a transition to an element in 3D space. See the *CardFlip* sample code project for the complete source code for this example.

Figure 2 Card Flip example



Setting Transition Properties

You use the CSS transition properties to set parameters of a transition. Typically, you need to set the element's property that the transition applies to and the duration of the transition. You don't need to set the element's property if you want the transition to apply to all properties. In addition, you can set a timing function (how intermediate values are distributed over the duration) and a delay before the transition begins.

Each parameter has a corresponding CSS property beginning with `-webkit-transition`. Use the `-webkit-transition-property` property to set the element's property and the `-webkit-transition-duration` property to set the duration of the transition. In addition, you can use the `-webkit-transition` property as a shorthand to set all the transition parameters using one style rule.

For example, Listing 1 defines a simple 2 second transition when the `opacity` property of an element is set.

Listing 1 Setting transition properties

```
div {
  -webkit-transition-property: opacity;
  -webkit-transition-duration: 2s;
}
```

You can also pass multiple comma-separated parameters to the `-webkit-transition-...` properties to set up multiple transitions at once. The order of the parameters determines which transition the settings apply to. For example, in Listing 2, the first parameter of each line applies to a `-webkit-transform` transition and the second parameter applies to an `opacity` transition.

Listing 2 Creating multiple transitions at once

```
div.zoom-fade {
    -webkit-transition-property: -webkit-transform, opacity;
    -webkit-transition-duration: 4s, 2s;
    -webkit-transition-delay: 2s, 0;
}
```

Listing 3 creates a 500 millisecond transition for the `opacity` property in one style rule using the `-webkit-transition` shorthand.

Listing 3 Setting all transition properties at once

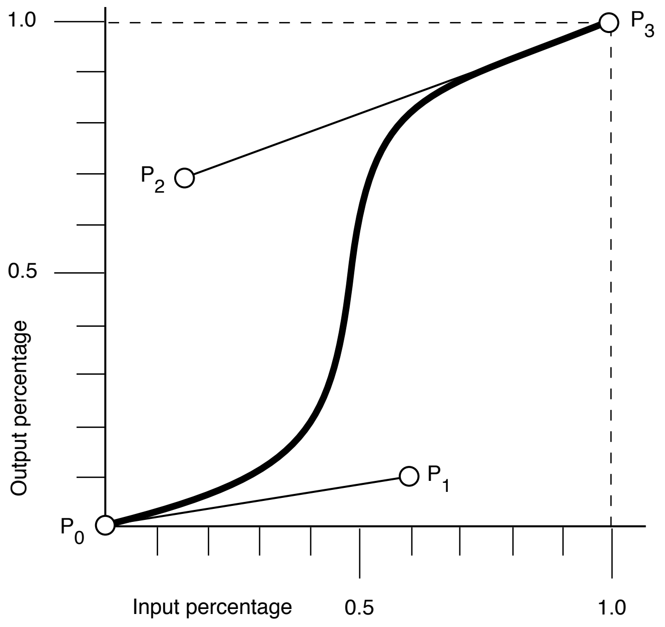
```
div.sliding {
 -webkit-transition: opacity 500ms ease-out 100ms;
}
```

Using Timing Functions

Timing functions allow a transition to change speed over its duration. The timing function is a mathematical function that provides a smooth curve or path for the transition. These effects are commonly called **easing** functions.

You set a timing function using the `-webkit-transition-timing-function` property. For example, you can define a timing function that starts out slowly, speeds up, and slows down at the end. The timing function is specified using a cubic Bezier curve, which is defined by four control points, as illustrated in Figure 3. The first and last control points are always set to (0,0) and (1,1), so you just need to specify the two in-between control points. The points are specified as a percentage of the overall duration.

Figure 3 Cubic Bezier timing function



You can set your control points by passing the `cubic-bezier()` function as the parameter with your custom control points as arguments or by using constants for common timing effects. For example, in Listing 4, transitions are created for the `-webkit-transform` and `opacity` properties. The `-webkit-transform` transition is 4 seconds and uses an `ease-out` timing function. The `opacity` transition is 2 seconds and uses a custom Bezier path.

Listing 4 Setting a transition's timing function

```
div.zoom-fade {
    -webkit-transition-property: -webkit-transform, opacity;
    -webkit-transition-duration: 4s, 2s;
    -webkit-transition-timing-function: ease-out, cubic-bezier(0.5, 0.2, 0.3,
1.0);
}
```

The default value for the timing function is `ease`, where the control points are `(0.25, 0.1)` and `(0.25, 1.0)` respectively.

Starting Transitions

Once you define a transition, you need to apply it to an element to start the transition. For example, normally, changing the value of an element's `left` style property causes the element to jump to the new location. Although this is fine for static pages, it provides for a limited user-interface experience for rich web applications. A JavaScript function could be used to iterate over an array of intermediate values, constantly updating the `left` property with new values, but this is computationally expensive and requires significantly more code. Instead you define a transition as described in "Setting Transition Properties" (page 11) and apply it to an element as follows.

1. Define the transition in CSS.

   This code fragment defines a 2 second opacity transition.

   ```
   div {
       -webkit-transition-property: opacity;
       -webkit-transition-duration: 2s;
   }
   ```

2. Create a corresponding style in CSS that can be applied to an element.

   This code fragment sets the `opacity` property to `0` causing a fade away effect.

   ```
   div.fadeAway {
       opacity:0;
   }
   ```

3. Apply the style to an element in HTML.

   This code fragment uses the `onclick` handler to trigger the fade away effect on the element.

   ```
   <div style="width:100px;
               height:100px;
               background-color:blue;"
               onclick="className = 'fadeAway'">
   ```

Listing 5 shows the complete HTML for this simple web application that displays a 100 x 100 box that fades away when clicked.

Listing 5 Simple fade away effect

```
<html>
<head>
    <meta name="viewport" content="width=device-width, initial-scale=1,
user-scalable=no"/>
    <style type="text/css" media="screen">

div {
    -webkit-transition-property: opacity;
    -webkit-transition-duration: 2s;
}

div.fadeAway {
    opacity:0;
}
```

```
    </style>
</head>
<body>


<div style="width:100px;
            height:100px;
            background-color:blue;"
            onclick="className = 'fadeAway'">
Tap to fade
</div>

</body>
</html>
```

Handling Transition Events

You can set a handler for a DOM event that is sent at the end of a transition. The event is an instance of `WebKitTransitionEvent` and its type is `webKitTransitionEnd` in JavaScript.

For example, the JavaScript code in Listing 6 displays an alert panel whenever a transition ends.

Listing 6 Handling transition end event

```
box.addEventListener( 'webkitTransitionEnd', function( event ) { alert( "Finished
 transition!" ); }, false );
```

Animations

You can use animation properties to animate elements as they move, resize, change color and opacity, and undergo other visual changes. You can also animate 2D and 3D visual effects such as rotating, scaling, and translating elements.

To create an animation, you first set animation properties, then you create keyframes, and then apply the animation to some elements. For example, you set the animation's name, duration, direction, and iteration count. You can also set a timing function that specifies how an animation progresses between keyframes. The timing function defines a Bezier curve—a path that the animation follows between keyframes. You define keyframes using a special keyframes at-rule.

Note: There are actually two types of animations: declared animations and implicit animations. Declared animations, described in this article, are explicitly executed when the animation properties are applied to elements. Implicit animations are called **transitions** and are triggered when you set a new value for an animatable CSS property. Read "Transitions" (page 9) for details on implicit animations.

How Animations Work

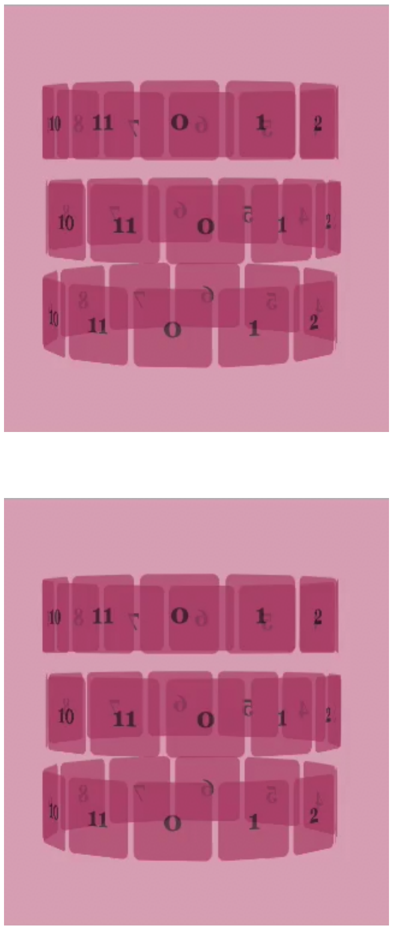
Animations are similar to transitions except you have more fine control by specifying keyframes. A keyframe defines the state of an animation at a particular point in time. A sequence of keyframes define the starting, middle, and ending points of an animation. The frames in between these points are computed based on the animation and style properties you set.

Similar to transitions, when applying an animation to an element's property, the change animates smoothly from the old value to the new value over time. The property values are computed over time. Therefore, if you query the value of a property during an animation, you may get an intermediate value that is the current animated value, not the old or new value. However, unlike transitions, animations do not change property values at the end of the animation.

Similar to transitions, you can control how these intermediate values are computed over time using a timing function, except that the timing function applies to the part of the animation between keyframes. Read "Using Timing Functions" (page 18) to learn more about timing functions.

You can create unique effects combining animations with 2D and 3D transforms. For example, Figure 1 shows an animation of elements in 3D space. See the *PosterCircle* sample code project for the complete source code for this example.

Figure 1 Circle Poster example



Setting Animation Properties

Use the CSS animation properties to set basic parameters about an animation. At a minimum, you need to set an animation name and duration. In addition, you can specify how many times an animation iterates, whether it alternates between the begin and end states, and whether the animation should begin in a running or paused state. You can also set a delay before the animation begins.

Each parameter has a corresponding CSS property beginning with -`webkit-animation`. For example, use the `-webkit-animation-name` property to specify a name. Use the `-webkit-animation-duration` property to specify the duration in seconds. Use the `-webkit-animation-iteration-count` property to specify the number of times the animation repeats. In addition, you can use the `-webkit-animation` property as a shorthand to set all these parameters in one style rule.

For example, an animation called diagonal-slide is declared in Listing 1. The animation name is `diagonal-slide`, its duration is five seconds, and it repeats ten times.

Listing 1 Setting animation properties

```
.diagonalslide {
  -webkit-animation-name: diagonal-slide;
  -webkit-animation-duration: 5s;
  -webkit-animation-iteration-count: 10;
}
```

The animation name is used to associate it with its keyframes. You use the animation name in the keyframes at-rule as described in "Creating Keyframes" (page 17).

Creating Keyframes

Keyframes are specified in CSS using a specialized `@-webkit-keyframes` at-rule. The keyframes rule consists of the `@-webkit-keyframes` keyword, followed by the animation name, and then a set of style rules for each keyframe as shown in Listing 2. The style rules are contained in blocks, surrounded by braces, and are preceded by a percentage value marking the keyframe's point in time. Any style rules can be placed in the blocks including 2D and 3D visual effects.

The value is a percentage of the animation's duration (or the keywords `from` or `to`) . For example, 0% specifies the start of an animation, 50% halfway through an animation, and 100% the end of an animation. The `from` keyword is equivalent to 0% and the `to` keyword is equivalent to 100%. When the animation executes, it transitions smoothly from one state to the next in increasing order, from 0% to 100%.

Listing 2 shows a keyframes at-rule where the animation is called `wobble` and the keyframes move an element back and forth over time. (You set the animation name using the `-webkit-animation-name` property as described in "Setting Animation Properties" (page 16).) In the first keyframe, the value of the element's `left` property is 100 pixels. By 40% of the animation duration, the value of the `left` property animates to 150 pixels. At 60% of the animation duration, the value of the `left` property animates back to 75 pixels. At the end of the animation, the value of the `left` property returns to 100 pixels.

Listing 2 Declaring keyframes

```
@-webkit-keyframes wobble {

  0% {
    left: 100px;
  }

  40% {
    left: 150px;
  }

  60% {
    left: 75px;
  }

  100% {
    left: 100px;
  }

}
```

Using Timing Functions

Timing functions allow an animation to change speed over its duration. The timing function is a mathematical function that provides a smooth curve or path for the transition. These effects are commonly called **easing** functions.

You set a timing function in a keyframe block using the `-webkit-animation-timing-function` property. The timing function determines the speed of an animation from the current to the next keyframe. The timing function is specified using a cubic Bezier curve, which is defined by four control points as illustrated in Figure 3 (page 12). The first control point is always (0,0) and the last control point is (1,1), so you only need to specify the two in-between control points of the curve. The in-between points are specified as a percentage of the overall duration. For example, this line of code sets the second point to (0.5, 0.2) and the third point to (0.3, 1.0) using the `cubic-bezier()` function:

```
-webkit-transition-timing-function: cubic-bezier(0.5, 0.2, 0.3, 1.0);
```

Similar to the `-webkit-transition-timing-function` property, you can set your control points by passing the `cubic-bezier()` function as the parameter with your custom control points as arguments or by using constants for common timing effects. For example, Listing 3 shows a keyframes at-rule that uses two timing function constants: `ease-out` and `ease-in`. Five keyframes are specified for an animation called `bounce`. Between the first and second keyframe (the first quarter of the duration) an `ease-out` timing function is used. Between the second and third keyframe (second quarter of the duration) an `ease-in` timing function is used. And so on. The effect appears as an element that moves up the page 50 pixels, slowing down as it reaches its highest point then speeding up as it falls back to 100 pixels. The second half of the animation behaves in a similar manner, but only moves the element 25 pixels units up the page creating a bouncing ball effect.

Note: The y-axis increases downwards. Therefore, in the first bounce, when the element's y-coordinate changes from 100 to 50 pixels, it moves up by 50 pixels. In the second bounce, when the y-coordinate changes from 100 to 75 pixels, it moves up by 25 pixels, half the height of the first bounce.

Listing 3 Using timing functions in keyframes

```
@-webkit-keyframes bounce {

  from {
    -webkit-transform: translateY(100px);
    -webkit-animation-timing-function: ease-out;
  }

  25% {
    -webkit-transform: translateY(50px);
    -webkit-animation-timing-function: ease-in;
  }

  50% {
    -webkit-transform: translateY(100px);
    -webkit-animation-timing-function: ease-out;
  }

  75% {
    -webkit-transform: translateY(75px);
    -webkit-animation-timing-function: ease-in;
```

```
      }

      to {
        -webkit-transform: translateY(100px);
      }

    }
```

Repeating Animations

You use the `-webkit-animation-iteration-count` property to set the number of times to repeat an animation. Listing 4 defines an animation that slides elements from the upper-left to lower-right corner. The `-webkit-animation-iteration-count` property is set to 10 causing the animation to repeat 10 times when it is applied.

Listing 4 Creating an animation that repeats

```
.diagonalslide {
  -webkit-animation-name: diagonal-slide;
  -webkit-animation-duration: 5s;
  -webkit-animation-iteration-count: 10;
}

@-webkit-keyframes diagonal-slide {

  from {
    left: 0;
    top: 0;
  }

  to {
    left: 100px;
    top: 100px;
  }

}
```

Starting Animations

Once you create an animation and specify its keyframes, you apply it to an element to begin the animation. You do this by applying the animation style to an element. This is typically triggered by some event such as the user clicking an element.

Listing 5 shows how to apply an animation to an element when the user clicks it. (The `diagonalslide` is defined in Listing 4 (page 19).) When the user taps in the blue box that says "Tap to slide" the `onClick` attribute changes the style class of the `div` element to `diagonalslide` and the animation begins.

Listing 5 Starting an animation

```
<div onClick="className='diagonalslide'">
    Tap to slide
```

```
</div>
```

The effects of an animation cease once the animation completes or if the animation is removed. You remove an animation by setting the `-webkit-animation-name` property to a value that does not include the name of that animation.

When an animation finishes running because it has executed the number of times described in its iteration count, the properties that were being animated return to their original values without animation.

To restart an animation, set the `-webkit-animation-name` property to a value that includes the name of that animation. Because style changes are coalesced, you may have to do this after a short delay. Listing 6 shows how to restart the `bounce` animation.

Listing 6 Restarting an animation

```
<style>
top:
.bounce {
  -webkit-animation-name: bounce;
  -webkit-animation-duration: 4s;
  -webkit-animation-iteration-count: 3;
}
</style>

<script type="text/javascript" charset="utf-8">
function restartBounce(element)
{
  element.style.webkitAnimationName = '';
  window.setTimeout(function() {
    element.style.webkitAnimationName = 'bounce';
  }, 0);
}
</script>

<body>
<div class="bounce" onclick="restartBounce(this)">
</div>
</body>
```

Handling Animation Events

Several animation-related events are available through the DOM event system. The start and end of an animation, and the end of each iteration of an animation, all generate DOM events. The events are instances of the `WebKitAnimationEvent` class and the possible types are `webkitAnimationStart`, `webkitAnimationIteration`, and `webkitAnimationEnd` in CSS.

Listing 7 adds an event listener for `webkitAnimationEnd` events at page load time on the `div` element with the `box` identifier. (See Listing 3 (page 18) for the definition of the bounce keyframes in this sample.) When the `webkitAnimationEnd` event occurs, an instance of `WebKitAnimationEvent` is created, the function checks to see if the animation was the `bounce` animation, and, if so, hides the box by setting the element's display style to `hidden`.

Listing 7 Handling animation events

```
<html>
    <head>
        <meta name="viewport" content="width=device-width, initial-scale=1,
user-scalable=no"/>
       <title>Bounce Animation</title>
        <style>
            div#box {
               position: absolute;
               width: 100px;
               height: 100px;
               background-color: blue;
               -webkit-transform: translateY(100px);
            }
            .bounce {
                -webkit-animation-name: bounce;
                -webkit-animation-duration: 4s;
                -webkit-animation-iteration-count: 3;
            }
            @-webkit-keyframes bounce {
                ...
                }
            }
        </style>
        <script type="text/javascript" charset="utf-8">
            function attachEventHandler() {
                box.addEventListener( 'webkitAnimationEnd', function( event )
{ alert( "Finished animation!" ); }, false );
            }
        </script>
    </head>

<body onload="attachEventHandler()">
    <div id="box" class="bounce">
    </div>
</body>
</html>
```

Note that an event is generated for each `animation-name` value and not necessarily for each property being animated since you can animate more than one property of an element at a time. You can animate multiple properties either with a single `animation-name` value with keyframes containing multiple properties, or with multiple `animation-name` values. For the purposes of events, each `animation-name` specifies a single animation.

The time the animation has been running is sent with each event generated. This allows a `webkitAnimationIteration` event handler to determine the current iteration of a looping animation or the current position of an alternating animation. This time does not include any time the animation was in the paused play state.

Read "Interactive Visual Effects" (page 33) for how to create interactive user interfaces using visual effects with DOM touch events.

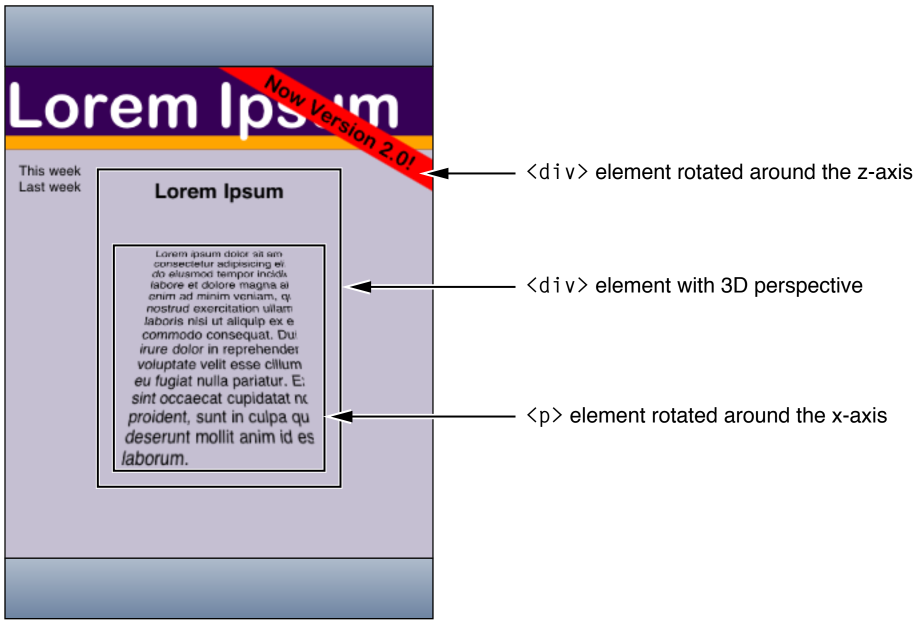
Transforms

CSS transform properties provide powerful formatting possibilities without having to resort to using images or Flash. Using the transform properties, elements can be translated, rotated, and scaled in 2D and 3D space. Perspective can also be applied to elements giving a sense of depth to the way they are rendered.

The CSS visual formatting model defines a coordinate system for each element. This coordinate space can be thought of as being expressed in pixels, starting in the upper-left corner of the parent with positive values proceeding to the right and down. In the basic formatting model, it is possible to set only the position and size of elements. Using CSS transforms, you can position elements in 2D and 3D space.

For example, Figure 1 shows a simple HTML document rendered using CSS transform properties. The "Now Version 2.0!" element is rotated around the z-axis. The "Lorem Ipsum" element specifies a 3D perspective for its children elements. The containing text element is rotated around the x-axis.

Figure 1 HTML page with rotation and perspective transforms

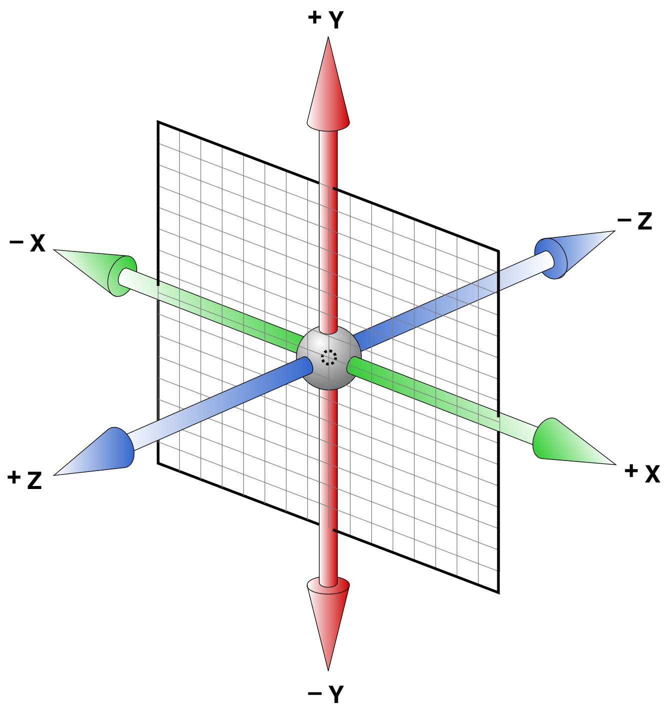

Lorem Ipsum
Now Version 2.0!
This week
Last week
Lorem Ipsum
`<div>` element rotated around the z-axis
`<div>` element with 3D perspective
`<p>` element rotated around the x-axis

Before using transform properties, you should understand how the coordinate systems and rendering work. Then use CSS transform properties to set the transform function per element. Use additional transform properties to set the origin and set other rendering attributes.

Coordinate Systems and Rendering

Transformed coordinate spaces behave differently in 2D and 3D. In 2D, the transformed coordinate space behaves as described in the coordinate system transformations section of the Scalable Vector Graphics (SVG) 1.1 Specification. This is a coordinate system with two axes: The x-axis increases horizontally to the right; the y-axis increases vertically downwards. In 3D, a z-axis is added, with positive z-values conceptually rising perpendicularly out of the window toward the user and negative z-values falling into the window away from the user as illustrated in Figure 2.

Figure 2 3D coordinate space



Setting a transform property for an element establishes a new local coordinate system for that element. You can apply multiple transforms to parent and child elements. Transformations are cumulative. That is, elements establish their local coordinate system within the coordinate system of their parent. In this way, a transform property effectively accumulates all the transform properties of its ancestors. The accumulation of these transforms defines a current transformation matrix (CTM) for the element.

A transform property does not affect the flow of the content surrounding the transformed element. However, the value of the overflow area takes into account transformed elements. This behavior is similar to what happens when elements are translated via relative positioning. Therefore, if the value of the `overflow` property is `scroll` or `auto`, scroll bars appear as needed to see content that is transformed outside the visible area.

If a transform, other than the identity transform, is set for an element, a stacking context and a containing block is created for that element. The object acts as though `position: relative` has been specified, but also acts as a containing block for fixed-positioned descendants. The z-position of a transformed element (its location on the z-axis) does not affect the order within a stacking context. With elements at the same z-index, objects are drawn in order of increasing z-position.

Note that elements are flat, planar surfaces with no z-depth in their default orientation. The transform property can be used to change their size, position, and orientation to position them in 3D space. However, though elements can be positioned to intersect, they are always rendered fully above or below each other. Which elements are rendered later (and therefore above elements rendered earlier) is determined by which is "closer" to the viewer.

Setting Transform Functions

A 2D or 3D transform is applied to an element using the `-webkit-transform` property. The parameter to this property is a list of transform functions, described in "Visual Effects Timing Functions" in *Safari CSS Reference*. The set of transform functions is similar to those allowed by SVG, although there are additional functions to support 3D transforms.

Rotating an Element

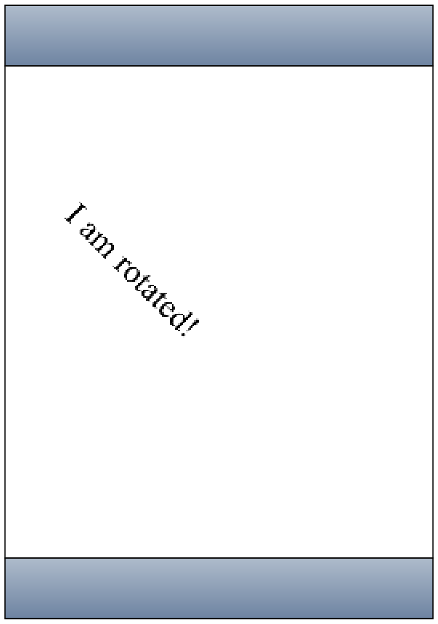
Listing 1 rotates a `div` element by 45 degrees around the z-axis using an inline style.

Listing 1 Rotating an element

```
<div style = "width: 12em;
          margin-top: 5em;
  -webkit-transform: rotate(45deg)">
I am rotated!
</div>
```

The results on iPhone OS are shown in Figure 3.

Figure 3 Rotating text



The `rotate()` function is just one of a number of functions you can apply using the `-webkit-transform` property. For a complete list of transform functions, see "Visual Effects Transform Functions" in *Safari CSS Reference*.

Setting Multiple Transforms

You can apply multiple transforms to the same element and apply transforms to parent and child elements. The `-webkit-transform` property accepts a single transform function or a whitespace-separated list of functions. When a list of functions is provided, the final transformation value for the element is obtained by performing a matrix concatenation of each entry in the list.

Listing 2 sets an element's transform to a list of transform functions and Listing 3 produces the same results by applying different transforms to nested elements.

Listing 2 Setting multiple transforms using a whitespace-separated list

```
<div style="-webkit-transform:translate(-10px,-20px)
           scale(2) rotate(45deg) translate(5px,10px)"/>
```

Listing 3 Nesting transforms

```
<div style="-webkit-transform:translate(-10px,-20px)">
  <div style="-webkit-transform:scale(2)">
    <div style="-webkit-transform:rotate(45deg)">
      <div style="-webkit-transform:translate(5px,10px)">
      </div>
    </div>
  </div>
</div>
```

Changing the Origin

Use the `-webkit-transform-origin` property to change the origin of transforms applied to an element from the center default. The origin is expressed as a percentage of the size of the element. For example, the default value 50% 50% causes transformations to occur around the center of the element. Changing the origin to 100% 0% causes transformation to occur around the top-right corner of on element.

The code in Listing 4 rotates the second box in Figure 4 around the top-right corner.

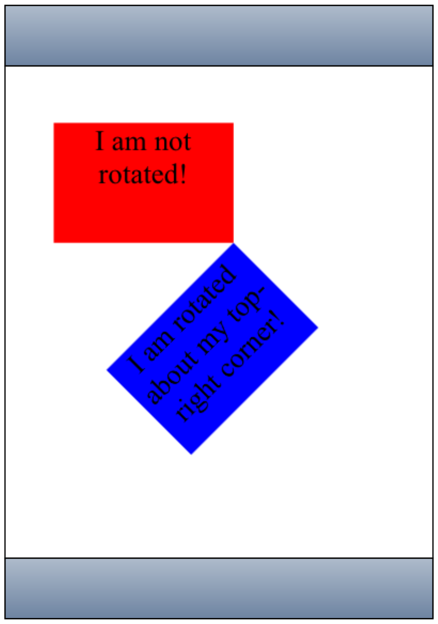
Listing 4 Rotating an element around the top-right corner

```
<div style="height: 200px;
             width: 300px;
         font-size:300%;
        text-align:center;
  background-color:red;">
I am not rotated!
</div>

<div style="height: 200px;
```

```
              width: 300px;
          font-size: 300%;
         text-align: center;
   background-color: blue;
  -webkit-transform: rotate(-45deg);
   -webkit-transform-origin: 100% 0%;">
I am rotated about my top-right corner!
</div>
```

Figure 4 Element rotated around the top-right corner



Setting the Perspective

Use the `-webkit-perspective` property to change the perspective in 3D—give a sense of depth to a scene by causing elements further away from the viewer to appear smaller. The `-webkit-perspective` property applies the same transform as the `perspective()` transform function, except that it applies only to the children of the element, not to the transform applied to the element itself.

The code fragment in Listing 5 sets the parent element's `-webkit-perspective` property to `500` causing the child element to appear in 3D.

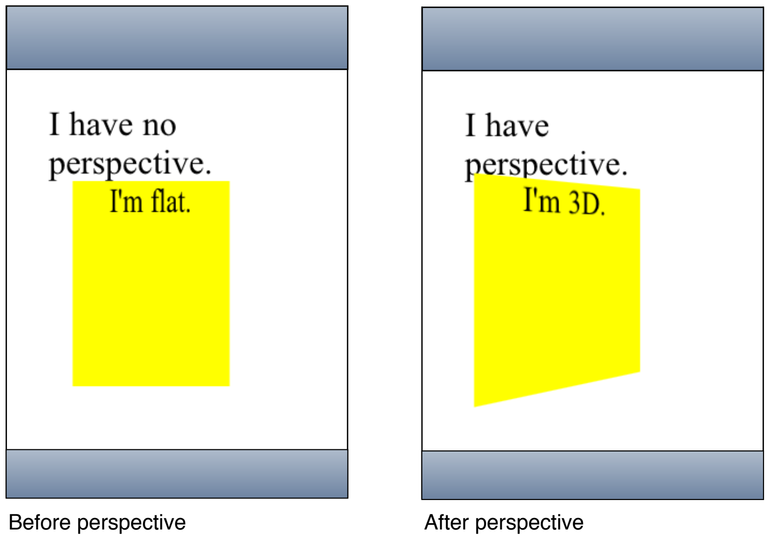
Listing 5 Adding perspective

```
<html>
    <head>
        <meta name="viewport" content="user-scalable=no, width=device-width"/>
        <title>Setting the Perspective</title>
    </head>
    <body>
       <div style="font-size: 200%; margin: 1em 1em; -webkit-perspective: 500;"
 >
        I have perspective.
```

```
            <div style="height: 6em; width: 6em; text-align:center;
background-color: yellow; -webkit-transform: rotateY(40deg);">
            I'm 3D.
            </div>
        </div>
    </body>
</html>
```

Figure 5 shows the before and after results of setting the perspective of a parent element.

Figure 5 Setting the perspective



Setting the Transform Style

Use the `-webkit-transform-style` property to change how nested elements are rendered in 3D space. If `-webkit-transform-style` is `flat`, all children of this element are rendered flattened into the 2D plane of the element. Therefore, rotating the element about the x-axes or y-axes causes children positioned at positive or negative z positions to appear on the element's plane, rather than in front of or behind it. If `-webkit-transform-style` is `preserve-3d`, this flattening is not performed, so children maintain their position in 3D space.

This flattening takes place at each element, so preserving a hierarchy of elements in 3D space requires that each ancestor in the hierarchy have `-webkit-transform-style` set to preserve 3D. But `-webkit-transform-style` affects only an element's children; the leaf nodes in a hierarchy do not require the `preserve-3d` style.

The code in Listing 6 shows how to set the transform style of the parent element.

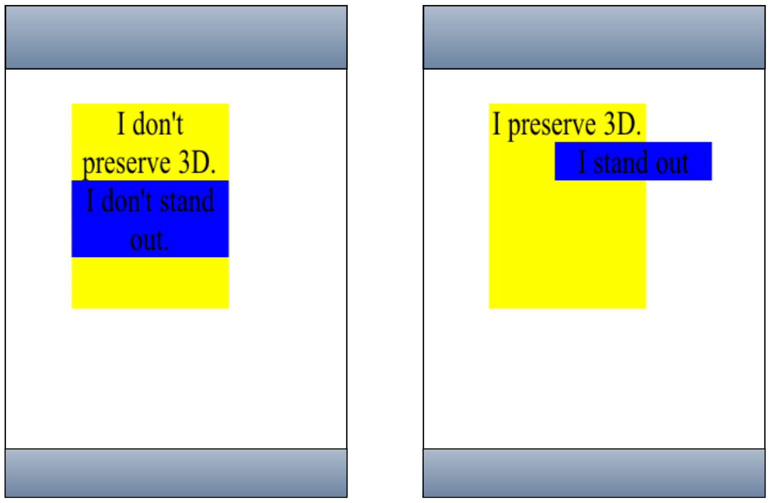
Listing 6 Setting the transform style

```
<div style="font-size: 200%; margin: 1em 1em;
            -webkit-perspective: 500;" >
  <div style="height: 8em;
```

```
            width: 6em;
            text-align:center;
            background-color: yellow;
            -webkit-transform-style: preserve-3d;
            -webkit-transform: rotateY(40deg);">
                I am the parent, and have perspective.
            <div style="-webkit-transform: translateZ(3em);
                        background-color: blue;">
                I stand out from my parent element.
            </div>
  </div>
</div>
```

Figure 6 shows the before and after results of setting this property when the parent element has no perspective.

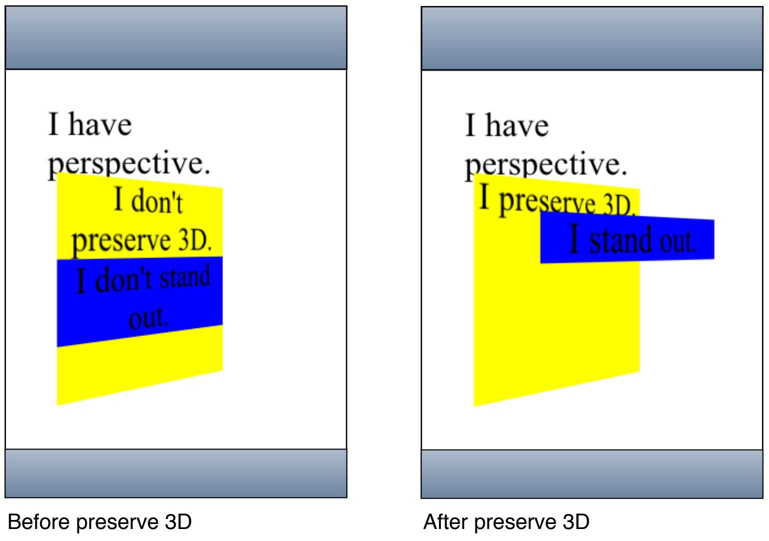
Figure 6 Preserving 3D



Before preserve 3D After preserve 3D

Figure 7 shows the before and after results of setting this property when the parent element has perspective too.

Figure 7 Setting the perspective and preserving 3D



Note that it's possible for the elements in a 3D tree to be located behind an ancestor element and therefore, be invisible or hidden. To prevent this, ensure that ancestor elements in a 3D tree that uses `preserve-3d` set `-webkit-transform-style` to `flat` (the default).

Also, the effect of setting `-webkit-transform-style` to `preserve-3d` may not be possible for all elements. Elements that have `overflow` set to `hidden` are unable to render their child elements in 3D. In this case, the element behaves as if the property is set to `flat`.

Modifying 3D Rendering

Use the `-webkit-backface-visibility` property to set whether or not the "back side" of a transformed element is visible when facing the viewer. If the identity transform is set, the front side of an element faces the viewer. Applying a rotation about the y-axis of 180 degrees (for instance) causes the back side of the element to face the viewer. For example, you might use this setting to create a box out of six elements, but where you want to see the inside faces of the box. You also need this property when creating a backdrop for a 3D stage.

Another example is when you want to place two elements back to back, as in *CardFlip* sample code project. Without this property, the front and back elements of the face card could at times switch places during the flip transition causing a flicker. Listing 7 shows the style settings of the face card that sets the `-webkit-backface-visibility` property to `hidden`. Figure 2 (page 10) shows the results of setting this property—a smooth flip transition with no flicker.

Listing 7 Hiding the back side of a face card

```
/* Styles the card and hides its "back side" when the card is flipped */
.face
{
    position: absolute;
    height: 300px;
    width: 200px;
    /* Give a round layout to the card */
    -webkit-border-radius: 10px;
    /* Drop shadow around the card */
    -webkit-box-shadow: 0px 2px 6px rgba(0, 0, 0, 0.5);
    /* Make sure that users will not be able to select anything on the card */

   /* We create the card by stacking two div elements at the exact same location.
 The back of the card
       is shown when we rotated the card 180 degrees along the y-axis. Setting
 this property to hidden
       ensures that the "back side" is hidden when the card is flipped.
    */
    -webkit-backface-visibility: hidden;
}
```

Creating Transforms in JavaScript

There are also a few JavaScript classes that you can use to access and manipulate transforms. Use the `WebKitCSSMatrix` class to create a 4x4 matrix and the `WebKitCSSTransformValue` class to get and set the transform. There are also methods added to `DOMWindow` to convert points that use instances of the `WebKitPoint` class as parameters. This section presents some examples of manipulating transforms in JavaScript. Refer to *Safari DOM Extensions Reference* for details on all these classes.

Use the window's `getComputedStyle()` method to get an element's style and the `webkitTransform()` method to get an element's transform as follows:

```
var theTransform = window.getComputedStyle(element).webkitTransform();
```

The `webkitTransform()` method returns a string representation of a 4x4 homogeneous matrix.

Similarly, use the `webkitTransform` property to set the transform of an element as follows:

```
var box2 = document.getElementById('box2');
box2.style.webkitTransform = theTransform;
```

You can use the string representation of a matrix as the argument to create an instance of `WebKitCSSMatrix` as follows:

```
var matrix = new WebKitCSSMatrix(theTransform);
```

Once you create a `WebKitCSSMatrix` object, you can apply transform functions to it in JavaScript using the `multiply()`, `inverse()`, `translate()`, `scale()`, `rotate()`, and `rotateAxisAngle()` methods. This code fragment sets the transform of another element to a scaled version of the matrix above:

```
var box3 = document.getElementById('box3');
box3.style.webkitTransform = matrix.scale(0.5, 0.5);
```

See the documentation for WebKitCSSMatrix in *Safari DOM Extensions Reference* for more details on these methods.

Interactive Visual Effects

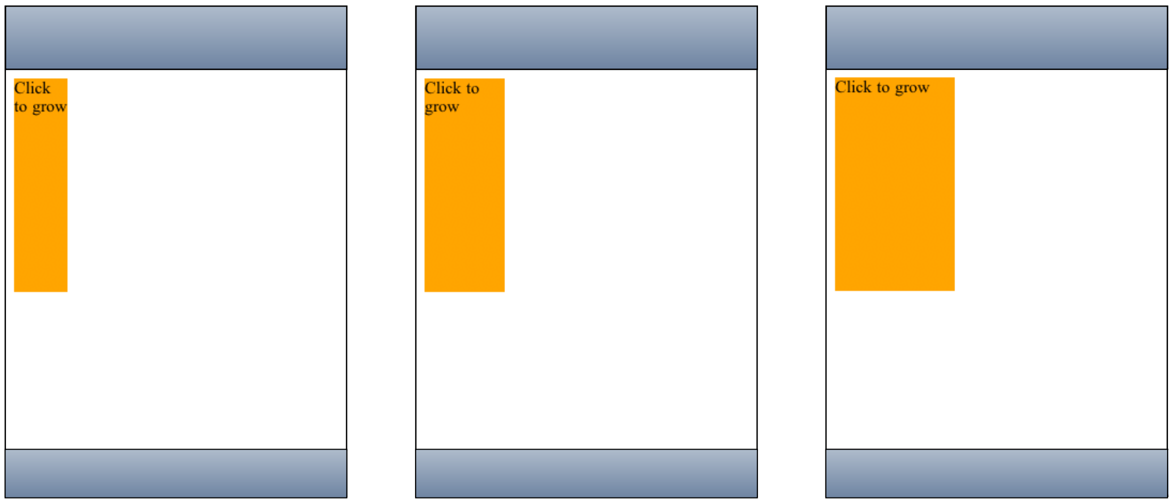
Note: JavaScript visual effects and DOM touch events used in this chapter are available on iPhone OS only.

You can use visual effects in combination with DOM mouse and touch events to create interactive web applications—applications that allow the user to manipulate objects on the screen with the mouse or finger. The first step to building interactive web applications is to register for user events—specifically, register for mouse, touch, and gesture events. Next, you implement the event handlers to apply the desired visual effects. This article contains three step-by-step examples that show you how to combine event handling with visual effects. Each example uses different types of events and visual effects. For example, you can implement web applications that allow the user to manipulate a 2D graphics object using multi-touch gestures.

Using Click or Tap to Trigger a Transition Effect

In this example, the user can click an element using the mouse or tap an element using a finger on an iPhone OS-based device to change the size of an element. The element is a box that increases in width by a multiple of `1.5` when tapped using a transition effect as illustrated in Figure 1.

Figure 1 Click box to enlarge



Follow these steps to implement this example:

1. Create an element for the user to manipulate.

   This CSS fragment creates a box element and uses the `-webkit-transition-...` properties to create a smooth transition when the width of the box changes:

```
div#box
{
    background-color: blue;
    display: block;
    position: absolute;
    width: 50;
    height: 50;
    -webkit-transition-property: width;
    -webkit-transition-duration: 0.5s;
    -webkit-transition-timing-function: default;
}
```

2. **Register for input events.**

   The following HTML registers an `onclick` event handler, `grow()`, for the box element:

```
<body>
    <div id="box" onclick="grow();"></div>
    <h1>Click box to grow</h1>
</body>
```

3. **Implement the event handlers.**

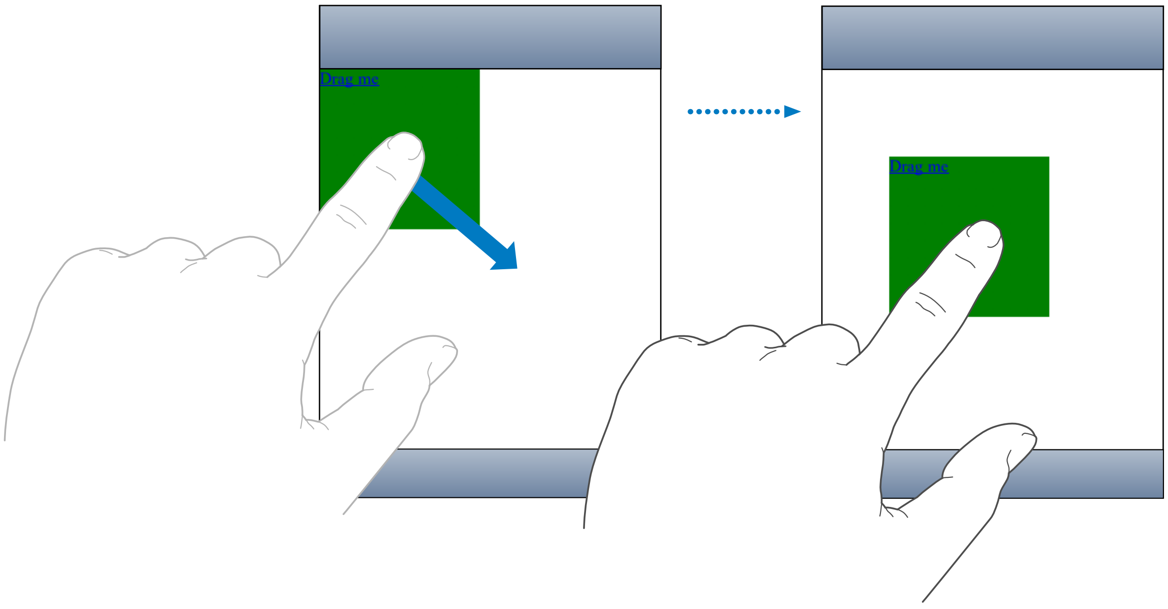
   The following `grow()` JavaScript method sets the width of the box, which triggers the transition effect in step 1:

```
function grow() {
    box = document.getElementById( "box" );
    var old_width = box.offsetWidth;
    box.style.width = old_width * 1.5;
}
```

Using Touch to Drag Elements

In this example, the user can touch a box to drag it around the screen. The high-level steps are the same as before. You create an element to manipulate, register event handlers, and implement event handlers. However, in this example, touch events and transforms are used, and most of the application is written in JavaScript.

Figure 2 Drag box application



Create an Element to Manipulate

Follow these steps to create a simple box that can be dragged across the screen.

1. Implement the element creation method in JavaScript.

   This `Box` creation method sets the initial position of the box and registers a handler for the `touchstart` event.

```
function Box(inElement)
{
    var self = this;

    this.element = inElement;
    this.position = '0,0';
    this.element.addEventListener('touchstart', function(e) { return
self.onTouchStart(e) }, false);
}
```

2. Next create `Box.prototype` to contain additional methods for getting and setting the position of the box.

```
Box.prototype = {
...
}
```

3. Add a `position()` get method as follows:

```
get position()
{
    return this._position;
},
```

4. Add a `position()` set method to set the transform using the `-webkit-transform` CSS property.

```
// position strings are "x,y" with no units
set position(pos)
{
    this._position = pos;

    var components = pos.split(',')
    var x = components[0];
    var y = components[1];

    const kUseTransform = true;
    if (kUseTransform) {
        this.element.style.webkitTransform = 'translate(' + x + 'px, ' + y +
'px)';
    }
    else {
        this.element.style.left = x + 'px';
        this.element.style.top = y + 'px';
    }
},
```

Note: Visual effects properties follow the same conventions as other CSS properties you set in JavaScript. Therefore, setting `this.element.style.webkitTransform` in JavaScript is the same as setting the `-webkit-transform` property in CSS.

5. Add `x()` and `y()` get and set methods to support the `position()` get and set methods above.

```
// position strings are "x,y" with no units
get x()
{
    return parseInt(this._position.split(',')[0]);
},

set x(inX)
{
    var comps = this._position.split(',');
    comps[0] = inX;
    this.position = comps.join(',');
},

get y()
{
    return parseInt(this._position.split(',')[1]);
},

set y(inY)
{
    var comps = this._position.split(',');
    comps[1] = inY;
    this.position = comps.join(',');
},
```

6. Now, add a `loaded()` function that creates the box element:

```
function loaded()
{
        new Box(document.getElementById('main-box'));
    }
```

7. Add this line of code to invoke the `loaded()` method when the application launches:

   ```
   window.addEventListener('load', loaded, true);
   ```

8. Finally, set the appearance of the box using CSS as follows:

   ```
   .box {
       position: absolute;
       height: 150px;
       width: 150px;
       background-color: green;
   }

   .box:active {
       background-color: orange;
   }
   ```

Note: Elements have to be clickable to receive touch events. Read "Making Elements Clickable" in *Safari Web Content Guide for iPhone OS* for how to make a nonclickable element clickable.

Register and Implement Event Handlers

The bulk of the work in dragging the box is implemented in the touch event handlers. The touch events are instances of the `TouchEvent` class. This example registers for all three types of touch events: `touchstart`, `touchmove`, and `touchend` events. It tracks single touches, not gestures, to implement dragging. Read "Using Gestures to Translate, Scale, and Rotate Elements" (page 39) for how to handle gesture events.

1. First implement the `onTouchStart()` event handler to start tracking touch events.

   The `onTouchStart()` method was registered as an event handler in "Create the Element to Manipulate" (page 39). This method first checks to see that there is only one touch, not multiple touches. It then stores the current position and registers for `touchmove` and `touchend` events.

   ```
   onTouchStart: function(e)
   {
       // Start tracking when the first finger comes down in this element
       if (e.targetTouches.length != 1)
           return false;

       this.startX = e.targetTouches[0].clientX;
       this.startY = e.targetTouches[0].clientY;

       var self = this;
       this.element.addEventListener('touchmove', function(e) { return
   self.onTouchMove(e) }, false);
       this.element.addEventListener('touchend', function(e) { return
   self.onTouchEnd(e) }, false);

       return false;
   },
   ```

2. Next implement the `onTouchMove()` event handler to move the element. The `position()` set method, implemented in "Create the Element to Manipulate" (page 39), uses the `-webkit-transform` CSS property to translate the element.

    ```
    onTouchMove: function(e)
    {
        // Prevent the browser from doing its default thing (scroll, zoom)
        e.preventDefault();

       // Don't track motion when multiple touches are down in this element (that's
     a gesture)
        if (e.targetTouches.length != 1)
            return false;

        var leftDelta = e.targetTouches[0].clientX - this.startX;
        var topDelta = e.targetTouches[0].clientY - this.startY;

        var newLeft = (this.x) + leftDelta;
        var newTop = (this.y) + topDelta;

        this.position = newLeft + ',' + newTop;

        this.startX = e.targetTouches[0].clientX;
        this.startY = e.targetTouches[0].clientY;

        return false;
    },
    ```

    Note: Use the `preventDefault()` method to disable the browser default behavior. For example, Safari on iPhone might attempt to scroll or zoom when the user touches and moves a finger on the screen.

3. Finally implement the `onTouchEnd()` event handler to remove the `touchmove` and `touchend` event handlers.

    This method is invoked after the last touch leaves the screen, so you no longer need to observe `touchmove` and `touchend` events until the next `touchstart` event.

    ```
    onTouchEnd: function(e)
    {
        // Prevent the browser from doing its default thing (scroll, zoom)
        e.preventDefault();

        // Stop tracking when the last finger is removed from this element
        if (e.targetTouches.length > 0)
            return false;

        this.element.removeEventListener('touchmove', function(e) { return
    self.onTouchMove(e) }, false);
        this.element.removeEventListener('touchend', function(e) { return
    self.onTouchEnd(e) }, false);

        return false;
    },
    ```

Using Gestures to Translate, Scale, and Rotate Elements

Gesture events are high-level events that encapsulate the lower-level touch events—they are instances of the `GestureEvent` class. Gesture and touch events can occur at the same time. Your application has the choice of handling touch events, gesture events, or both. The advantage of gesture events is that the location and angle of the fingers are already calculated when the events arrive. Thus gesture events support pinch open to zoom in, pinch close to zoom out, and pivoting to rotate elements. You simply apply the event location, scale, and rotation values to your element.

The steps in this example are similar to implementing the draggable box in "Using Touch to Drag Elements" (page 34). Instead of just dragging the box, the user can also pinch open and close to scale it and pivot to rotate it. To do this, you simply extend the example to handle gesture events for scaling and rotating. In summary, you create the element to manipulate, register for touch and gesture events, and implement handlers that translate, scale, and rotate the element.

Create the Element to Manipulate

Follow these steps to create a simple box that can be translated, scaled, and rotated.

1. Implement the element creation method in JavaScript.

   This `Box` creation method sets the initial position, scale, and rotation of the box and registers handlers for the `touchstart` and `gesturestart` events.

   ```
   function Box(inElement)
   {
       var self = this;

       this.element = inElement;

       this.scale = 1.0;
       this.rotation = 0;
       this.position = '0,0';

       this.element.addEventListener('touchstart', function(e) { return
   self.onTouchStart(e) }, false);
       this.element.addEventListener('gesturestart', function(e) { return
   self.onGestureStart(e) }, false);
   }
   ```

2. Follow steps 2 through 8 in "Create an Element to Manipulate" (page 35) to create a box prototype, add the `position()` accessors and associated support methods, and create the box.

Register and Implement Event Handlers

The bulk of the work in this application is in the touch and gesture event handlers. This example builds on the example in "Using Touch to Drag Elements" (page 34) by handling gesture events, too. The touch events are used to implement dragging as before and the gesture events are used to implement translating, scaling, and rotating.

1. First implement the `onTouchStart()`, `onTouchMove()`, and `onTouchEnd()` event handlers by following step 1 through 3 in "Register and Implement Event Handlers" (page 37) to allow single-finger dragging.

2. Next, implement the `onGestureStart()` event handler to begin tracking gesture events for translating, scaling, and rotating.

   This method registers the other gesture event handlers, `onGestureChange()` and `onGestureEnd()`, for the `gesturechange` and `gestureend` events.

```
onGestureStart: function(e)
{
    // Prevent the browser from doing its default thing (scroll, zoom)
    e.preventDefault();

    var self = this;
    this.element.addEventListener('gesturechange', function(e) { return
self.onGestureChange(e) }, true);
    this.element.addEventListener('gestureend', function(e) { return
self.onGestureEnd(e) }, true);

    return false;
},
```

3. Implement the `onGestureChange()` event handler to translate, scale, and rotate the element.

   This method sets the element's scale and rotation to the `GestureEvent` object's precomputed scale and rotation values. You do not need to compute the distance between multiple fingers or the change in angle to scale and rotate the element.

```
onGestureChange: function(e)
{
    // Prevent the browser from doing its default thing (scroll, zoom)
    e.preventDefault();

    // Only interpret gestures when tracking one object.  Otherwise, interpret
 raw touch events
    // to move the tracked objects.
    this.scale = e.scale;
    this.rotation = e.rotation;
    this.position = this.position;

    return false;
},
```

4. Implement the `onGestureEnd()` event handler to remove the handlers for the `gesturechange` and `gestureend` events.

```
onGestureEnd: function(e)
{
    // Prevent the browser from doing its default thing (scroll, zoom)
    e.preventDefault();

   this.element.removeEventListener('gesturechange', this.gestureChangeHandler,
 true);
   this.element.removeEventListener('gestureend', this.gestureEndHandler, true);

    return false;
},
```

Document Revision History

This table describes the changes to *Safari Visual Effects Guide*.

Date	Notes
2008-11-19	New document that describes how to add visual effects to web content that is supported by Safari on the desktop and iPhone OS.