
INSTANT
Short | Fast | Focused
Website Touch Integration
Start with an introduction to touch events and end with implementing your own custom gestures
Alexander Dickson
[PACKT]
PUBLISHING

Table of Contents

Instant Website Touch Integration
Credits
About the Author
About the Reviewers
www.PacktPub.com
    Support files, eBooks, discount offers and more
        Why Subscribe?
        Free Access for Packt account holders
Preface
    What this book covers
    What you need for this book
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Errata
        Piracy
        Questions
1. Instant Website Touch Integration
    The why and when of touch events
    Testing while keeping your sanity (Simple)
        How to do it...
        How it works…
        There's more…
            Device simulators
            Testing in Mobile Safari
            Testing on Android's Chrome
    How to get tapping (Simple)
        Getting ready
        How to do it…
        How it works...
        There's more…
    Swiping your way to success (Intermediate)
        Getting ready
        How to do it...
        How it works...
        There's more...
            There is something not quite right with the swipe
            Custom events for swipe
    Welcome gestures (Intermediate)
        Getting ready
        How to do it...
        How it works...
        There's more...

A custom gesture can go a long way (Advanced)
Getting ready
How to do it...
There's more...

Instant Website Touch Integration

Instant Website Touch Integration

First published: November 2013

Production Reference: 1211113

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK

ISBN 978-1-78328-049-0

www.packtpub.com

Credits

Author

Alexander Dickson

Reviewers

Palash Mondal

Damon Oehlman

Acquisition Editors

Martin Bell

Sam Birch

Commissioning Editor

Priyanka Shah

Technical Editor

Amit Ramdas

Copy Editor

Gladson Monteiro

Project Coordinator

Sherin Padayatty

Proofreaders

Maria Gould

Lindsey Thomas

Production Coordinator

Alwin Roy

Cover Work

Alwin Roy

Cover image

Disha Haria

About the Author

Alexander Dickson is a computer tinkerer with a special interest in JavaScript. He has always been interested in learning how things work.

He currently works at Atlassian in Sydney, Australia.

I'd like to thank a few people for making this book possible, my parents, for life and teaching me humor, my partner, Courtnee, for sticking around while I mucked around on the computer, Kev, for showing me the fun of computers, and Kerryn, for being a champ. Finally, thanks to Damon for taking the time to find my mistakes.

About the Reviewers

Palash Mondal is a professional .NET developer, a passionate programmer, and a great fan of John Resig—the creator and lead developer of jQuery. He holds a Bachelor's degree from the Academy of Technology (AOT), West Bengal, India. He is currently working at Mindfire Solutions, Bhubaneswar, India.

His primary focus is on frontend development and web application development.

Damon Oehlman is a software engineer who works with JavaScript and web technologies for both work and fun. He specializes in working with emerging browser technologies and is currently working with Australian technology, research, and commercialization company NICTA (http://www.nicta.com.au). He can be contacted via Twitter at http://twitter.com/DamonOehlman or through his GitHub profile at https://github.com/DamonOehlman.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read, and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface

Welcome to *Instant Website Touch Integration*. This book will take you on a wonderful journey through touch events: what they are and how you can use them to spruce up your web development skills, as well as the web applications you create.

So, time to don your adventure pants; let's dive in! It's time to have fun!

What this book covers

The why and when of touch events kicks off this journey by teaching what touch events are and in what circumstances you would use them. We hear about fancy terms for touching the screen in different ways, such as the tap, swipe, and pinch.

Testing while keeping your sanity (Simple) is the first recipe that shows you how to develop, test, and debug touch events. It explains how to use your traditional desktop environment so you can develop rapidly at the speed you're used to. This will cover browser emulation, device simulators, and remote debugging on real devices.

How to get tapping (Simple) is the most important recipe of the book. After we know what we're dealing with and how to develop, we dive in and discover the low-level events made available to the browser's event system, and what can be handled via JavaScript.

Swiping your way to success (Intermediate) leads on from your knowledge gained in the previous recipe. Now that you know the basics, we tackle swipes and how they can be used to drag views around the screen. We also learn how to add deceleration to these, so your users can swipe in a familiar fashion to which their iPhones and Android devices have accustomed them to.

Welcome gestures (Intermediate) moves things right along and up a notch. Now that you're starting to become a boss with touch events, it would seem that playing with gestures is the next logical step. We take a look at how to rotate views, scale them, and how we can make this work cross browser (only this time, it's not Internet Explorer that hates you).

A custom gesture can go a long way (Advanced) is the final recipe. To finally earn your metaphoric certificate in touch awesomeness, we will take a look at custom gestures. This will allow your web application to respond to any finger pattern you can dream of, contort your fingers to, and describe with code.

What you need for this book

It would be handy to have a touch-enabled device with a web browser to get the most out of this book (such as an iOS or Android device). It's also handy to have Google Chrome on your desktop, as at the time of writing, it has some handy features for doing web development with touch events.

Who this book is for

This book is for existing web developers who want to know about touch events and how to use them with their existing or future websites.

It is expected that you have a decent understanding of JavaScript. But only a dash of HTML/CSS knowledge is required (they're not particularly focused on in this book).

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meanings.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "You can then launch the browser to test the `touch` events".

A block of code is set as follows:

```
document.body
.addEventListener("touchmove", function(event) {
    for (var i = 0; i < event.touches.length;
i++) {
        // Do something with each touch.
    }
});
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Once you have Xcode, go to **Xcode** | **Preferences** | **Downloads** | **Components**".

> ## Note
>
> Warnings or important notes appear in a box like this.

> ## Tip
>
> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <feedback@packtpub.com>, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Instant Website Touch Integration

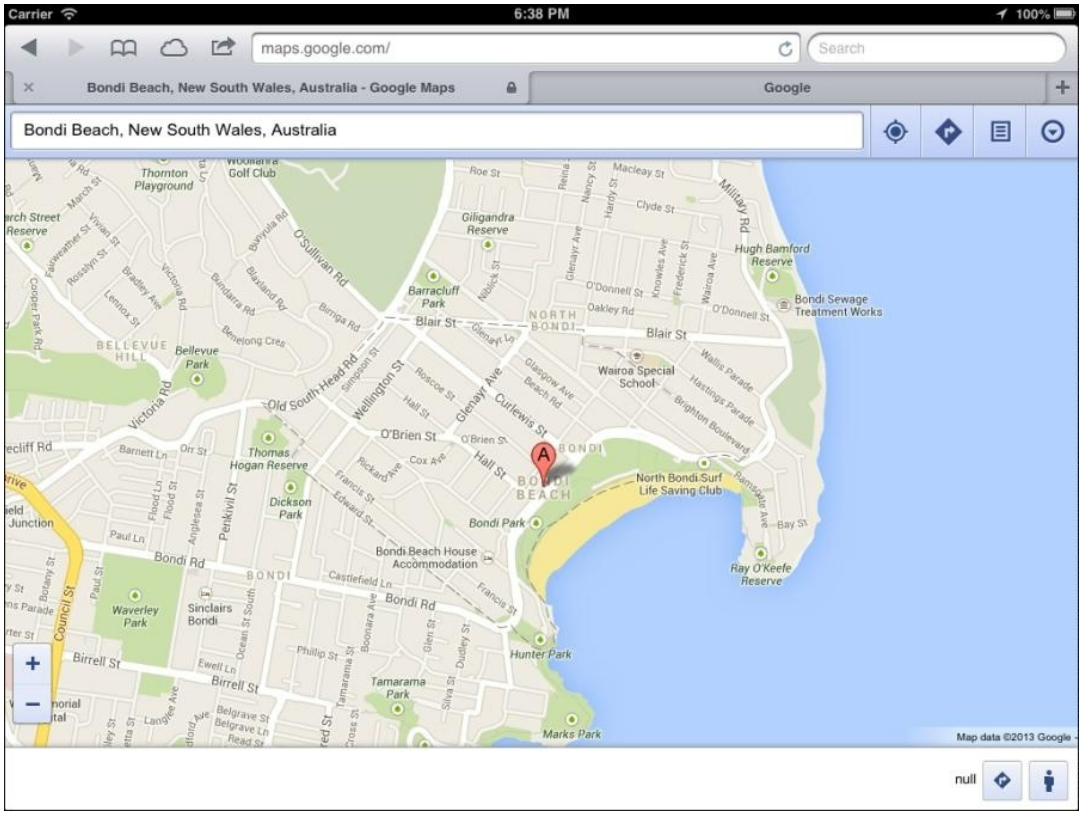
Welcome to *Instant Website Touch Integration*. The aim of this book is to give you the knowledge required to update your web development skills to encompass touch events. This will allow you to make your website's interactions feel more natural on mobile, competing with the feel that is more commonly associated with native applications. You can also have a lot of fun with touch events, and some pain with testing them. But don't get touchy, as I'll guide you through the ups and downs of touch events.

The why and when of touch events

Touch events allow your website to respond to your users' fingers, thus, giving a whole new dimension to foster creative user interactions. Mastering touch events allow you to make your website usable in a way not possible with the standard desktop input peripherals.

These touch events come in a few different flavors. There are taps, swipes, rotations, pinches, and more complicated gestures available for you to make your users' experience more enjoyable. Each type of touch event has an established convention which is important to know, so your user has minimal friction when using your website for the first time:

- The tap is analogous to the mouse click in a traditional desktop environment, whereas, the rotate, pinch, and related gestures have no equivalence. This presents an interesting problem—what can those events be used for?
- Pinch events are typically used to scale views (have a look at the following screenshot of Google Maps accessed from an iPad, which uses pinch events to control the map's zoom level), while rotate events generally rotate elements on screen. Swipes can be used to move content from left to right or top to bottom.
- Multifigure gestures can be different as per your website's use cases, but it's still worthwhile examining if a similar interaction already exists, and thus an established convention.



Web applications can benefit greatly from touch events. For example, imagine a paint-style web application. It could use pinches to control zoom, rotations to rotate the canvas, and swipes to move between color options. There is almost always an intuitive way of using touch events to enhance a web application.

In order to get comfortable with touch events, it's best to get out there and start using them practically. To this end, we will be building a paint tool with a difference, that is, instead of using the traditional mouse pointer as the selection and drawing tool, we will exclusively use touch events. Together, we can learn just how awesome touch events are and the pleasure they can afford your users.

Touch events are the way of the future, and adding them to your developer tool belt will mean that you can stay at the cutting edge. While MC Hammer may not agree with these events, the rest of the world will appreciate touching your applications.

Testing while keeping your sanity (Simple)

Touch events are awesome on touch-enabled devices. Unfortunately, your development environment is generally not touch-enabled, which means testing these events can be trickier than normal testing. Thankfully, there are a bunch of methods to test these events on non-touch enabled devices, so you don't have to constantly switch between environments. In saying that though, it's still important to do the final testing on actual devices, as there can be subtle differences.

How to do it...

1. Learn how to do initial testing with Google Chrome.
2. Debug remotely with Mobile Safari and Safari.
3. Debug further with simulators and emulators.
4. Use general tools to debug your web application.

How it works…

As mentioned earlier, testing touch events generally isn't as straightforward as the normal *update-reload-test* flow in web development. Luckily though, the traditional desktop browser can still get you some of the way when it comes to testing these events.

While you generally are targeting Mobile Safari on iOS and Chrome, or Browser on Android, it's best to do development and initial testing in Chrome on the desktop. Chrome has a handy feature in its debugging tools called **Emulate touch events**. This allows a lot of your mouse-related interactions to trigger their touch equivalents. You can find this setting by opening the Web Inspector, clicking on the **Settings** cog, switching to the **Overrides** tab, and enabling **Emulate touch events** (if the option is grayed out, simply click on **Enable** at the top of the screen).

Using this emulation, we are able to test many touch interactions; however, there are certain patterns (such as scaling and rotating) that aren't really possible just using emulated events and the mouse pointer. To correctly test these, we need to find the middle ground between testing on a desktop browser and testing on actual devices.

There's more…

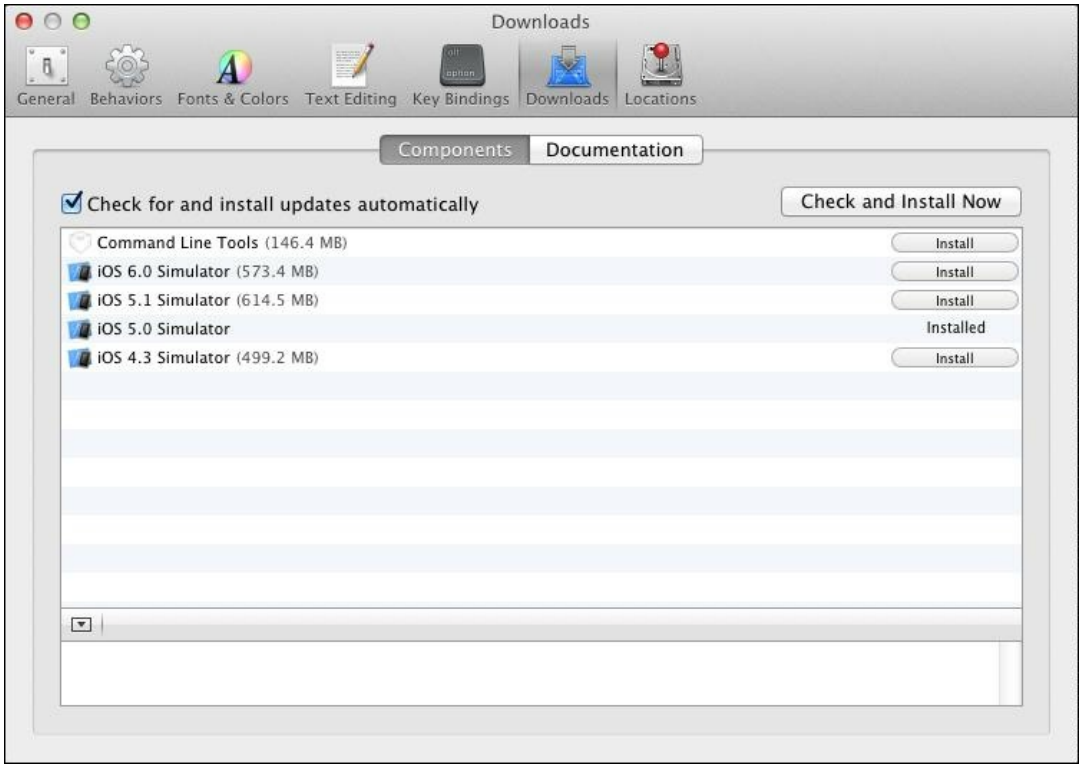
Remember the following about testing with simulators and browsers:

Device simulators

While testing with Chrome will get you some of the way, eventually you will want to use a simulator or emulator for the target device, be it a phone, tablet, or other touch-enabled device.

Testing in Mobile Safari

The iOS Simulator, for testing Mobile Safari on iPhone, iPod, and iPad, is available only on OS X. You must first download Xcode from the App Store. Once you have Xcode, go to **Xcode** | **Preferences** | **Downloads** | **Components**, where you can download the latest iOS Simulator. You can also refer to the following screenshot:

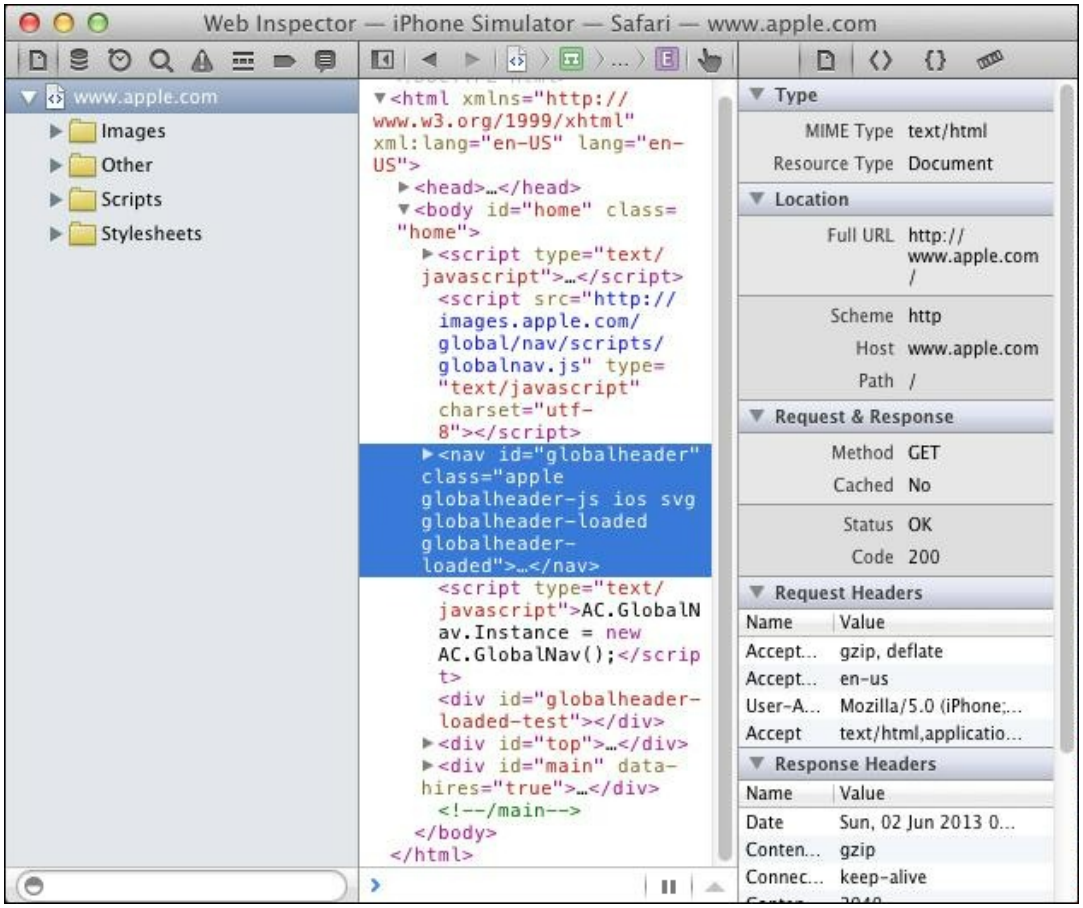


You can then open the iOS Simulator from Xcode by accessing **Xcode** | **Open Developer Tool** | **iOS Simulator**. Alternatively, you can make a direct alias by accessing Xcode's package contents by accessing the Xcode icon's context menu in the **Applications** folder and selecting **Show Package Contents**. You can then access it at `Xcode.app/Contents/Applications`. Here you can create an alias, and drag it anywhere using Finder.

Once you have the simulator (or an actual device); you can then attach Safari's Web Inspector to it, which makes debugging very easy. Firstly, open Safari and then launch Mobile Safari in either the iOS Simulator or on your actual device (ensure it's plugged in via its USB cable). Ensure that

Web Inspector is enabled in Mobile Safari's settings via **Settings** | **Safari** | **Advanced** | **Web Inspector**. You can refer to the following screenshot:

Then, in Safari, go to **Develop** | **<your device name>** | **<tab name>**. You can now use the Web Inspector to inspect your remote Mobile Safari instance as shown in the following screenshot:



Testing in Mobile Safari in the iOS Simulator allows you to use the mouse to trigger all the relevant touch events. In addition, you can hold down the *Alt* key to test pinch gestures.

Testing on Android's Chrome

Testing on the Android on desktop is a little more complicated, due to the nature of the Android emulator. Being an emulator and not a simulator, its performance suffers (in the name of accuracy). It also isn't so straightforward to attach a debugger.

You can download the Android SDK from http://developer.android.com/sdk. Inside the tools directory, you can launch the Android Emulator by setting up a new **Android Virtual Device** (**AVD**). You can then launch the browser to test the `touch` events. Keep in mind to use 10.0.2.2 instead of localhost to access local running servers.

To debug with a real device and Google Chrome, official documentation can be followed at https://developers.google.com/chrome-developer-tools/docs/remote-debugging.

When there isn't a built-in debugger or inspector, you can use generic tools such as **Web Inspector Remote** (**Weinre**). This tool is platform-agnostic, so you can use it for Mobile Safari, Android's Browser, Chrome for Android, Windows Phone, Blackberry, and so on. It can be downloaded from http://people.apache.org/~pmuellr/weinre.

To start using Weinre, you need to include a JavaScript on the page you want to test, and then run the testing server. The URL of this JavaScript file is given to you when you run the `weinre` command from its downloaded directory.

When you have Weinre running on your desktop, you can use its Web Inspector-like interface to do remote debugging and inspecting your touch events, you can send messages to `console.log()` and use a familiar interface for other debugging.

How to get tapping (Simple)

The cornerstone of touch events is the tap, which was first available to JavaScript via events courtesy of Apple's iPhone in 2007. At the time of writing, the W3C had formalized these events and it's finally in Proposed Recommendation status.

We'll take a look at the type of touch events and the data available with each one. With this knowledge, you can begin handling users' touch events.

Getting ready

You will need to open your editor, Google Chrome (so you can use emulated touch events) and preferably a touch-enabled device to do your final testing on. Set up your test environment as explained in the previous recipe.

How to do it…

1. Determine if the user's device supports touch events.
2. Find out which touch events are available to us.
3. Look at the meta information attached to every touch.
4. Start using touch events to develop the basics of a paint application.

How it works...

Before we start with this touchy subject, it's important to know how to detect if the user has touch events supported in their browser. An easy way to detect it is with:

```
var touchEvents = "ontouchstart" in document;
```

However, detecting touch events isn't always so straightforward. You can use a library such as Modernizr to detect if the device supports touch events. You can read more about how Modernizr detects touch events at `http://modernizr.github.io/Modernizr/touch.html`.

Let's take a look at the basic touch events available. There are three you will deal with regularly:

Event name	Triggered when the user…
`touchstart`	…places a finger on the element bound.
`touchmove`	…moves a finger over the element bound.
`touchend`	…lifts a finger over the element bound.

In addition to these three, there are also some more, but they are generally not used as often.

Event name	Triggered when the user's touch…
`touchenter`	….enters the element bound.
`touchleave`	…leaves the element bound.
`touchcancel`	…is disrupted, such as leaving the viewport.

An important caveat of the latter three events is that they do not propagate, that is, ancestor elements won't have those events fired on them (the events won't bubble up).

You bind these events like any other JavaScript events via `addEventListener()`.

```
document.body
.addEventListener("touchstart", function(event)
{ });
```

The `event` object for touch events includes some important properties that we will use to handle the user's interaction. These are:

Property	`TouchList` (array-like object) of fingers that is…
`touches`	…currently on the screen.
`targetTouches`	…currently on the element bound.
`changedTouches`	…associated with the event.

Given that `TouchList` is an array-like object, you can subscript the individual touches like a normal array and it has a `length` property.

```
document.body
.addEventListener("touchstart", function(event) {
    var secondTouch = event.touches[1];
});
```

Note

If you use jQuery to bind these events, be sure to access the `event` object at `event.originalEvent` which is a reference to the native `event` object.

In turn, these `TouchList` objects contain a number of `Touch` objects (one for each relative touch) which all have their own properties.

Property	Contents
`identifier`	A unique identifier for that finger.
`pageX`/`pageY`	The touch's x and y coordinates relative to the top-left corner of the document. Includes scroll offsets.

Property	Contents
`screenX`/`screenY`	The touch's x and y coordinates relative to the top-left corner of the screen.
`clientX`/`clientY`	The touch's x and y coordinates relative to the top-left corner of the viewport, not including scroll offsets.
`target`	The element on which the touch originated.

With this information, it's possible to create any touch behavior you desire. Keep in mind, you may want to cancel a lot of the browser's default behavior with `event.preventDefault()`, so the taps and more complicated events won't also trigger unintended side-effects (such as trying to scroll the document). When you cancel the default behavior of touch events, it also prevents the analogous mouse events from occurring, such as the `click` event.

If you're intending to handle more than one touch, it's best to iterate over the touches and handle them accordingly. This can be done just like iterating over an array with a `for` loop.

```
document.body
.addEventListener("touchmove", function(event) {
    for (var i = 0; i < event.touches.length;
i++) {
        // Do something with each touch.
    }
});
```

If you want to use normal `Array` methods (such as `forEach()`) on a `TouchList`, you can access the array's methods via the prototype and set the new context.

```
Array.prototype.forEach(event.touches,
function(touch) {
    // Do something with each touch.
});
```

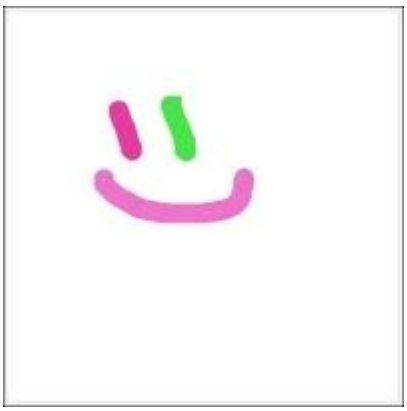
Tip

The click event and the infamous 300 ms delay

The `click` event is still fired on all touch-enabled devices after a touch that doesn't appear to be one of the former events. It's important to know that some touch-enabled browsers won't fire the `click` event straightaway when you tap, but instead wait about 300 ms. This is so the browser can distinguish between a tap and a double-tap (when the user taps twice rapidly). To ensure your application doesn't feel sluggish, you can make the `touchstart` events trigger the `click` events, by using a library such as FastClick (https://github.com/ftlabs/fastclick).

Pretend for a moment that we think wheels ought to be re-invented and also need to design a paint-style web application. Let's call it Finger Painter. Have a look at the following screenshot for an example. We will use this application as the basis of teaching the touch events. Instead of using the mouse, we want to allow the touch events to do the painting. With the knowledge above, we can construct something with the help of the `canvas` element too.

First, we will start with some HTML and CSS before we look at the JavaScript. Keep in mind that for these examples, we will require a browser that supports touch events and the `canvas` element.



We'll start with some simple HTML consisting of a `canvas` element:

```
<canvas width="400" height="400"></canvas>
```

Next, some simple CSS for a visual outline of the said `canvas` element:

```
canvas {
    border: 1px solid #ccc;
}
```

Finally, a dash of JavaScript to translate taps and subsequent drags into lines, so we can allow our users to draw:

```
var fingerPainter = {};
fingerPainter.getRandomColor = function() {
    var v = function() { return
Math.floor(Math.random() * 255); };
    return "rgb(" + [v(), v(), v()] + ")";
};
fingerPainter.init = function () {
    var canvas = document.querySelector("canvas");
    var canvasDimensions =
canvas.getBoundingClientRect();
    var ctx = canvas.getContext("2d");
    ctx.lineWidth = 20;
    ctx.lineCap = "round";

    canvas.addEventListener("touchstart",
function (event) {
        ctx.strokeStyle = ctx.fillStyle =
fingerPainter.getRandomColor();

        // Initial dot for single taps.
        ctx.beginPath();
        ctx.arc(event.targetTouches[0].pageX -
canvasDimensions.left,
event.targetTouches[0].pageY -
canvasDimensions.top, 10, 0, Math.PI * 2);
        ctx.fill();
        ctx.closePath();
        ctx.beginPath();

        // Set start point for drawing line.
        ctx.moveTo(event.targetTouches[0].pageX -
canvasDimensions.left,
event.targetTouches[0].pageY -
canvasDimensions.top);
    });

    canvas.addEventListener("touchmove", function
(event) {
        // Prevent default behavior of scrolling
elements.
        event.preventDefault();

        // Get a reference to the first touch
placed.
        var touch = event.touches[0];

        // Draw a line to the new location.
        ctx.lineTo(touch.pageX -
canvasDimensions.left, touch.pageY -
canvasDimensions.top);
```

```
            ctx.stroke();
        });
    };

    document.addEventListener ("DOMContentLoaded",
    fingerPainter.init);
```

The code is also found at http://alexanderdickson.com/touch-events-how-to/1.html.

In the previous example code, we use the `touchstart` event to choose a random color whenever the user first places his/her finger on the `canvas` element to begin painting. In addition to choosing a color, we also start the context for the line we may draw with `moveTo()`, and draw a circle at the x and y coordinates of the touch as found in the `pageX` and `pageY` properties (after we've deducted the offset of the `canvas` element, so they're relative) of the first `Touch` object. This will allow us to visualize our individual taps if the user doesn't trigger the `touchmove` event.

Then, on `touchmove`, we draw a line using `lineTo()` at the coordinates of the touch, again found in the `pageX` and `pageY` properties of the first `Touch` object. It's as easy as that to let your users' fingers do the work. Because there is nothing that is required to be done when the user lifts his/her finger, we haven't bothered to bind any function to the `touchend` event.

There's more…

Let's take a look now at something more complicated, what if we wanted to add some functionality to our paint application simply, so that we can demonstrate how to handle multiple touches? We could have two touches to draw a line, and three touches to draw a triangle.

In the following code, we wait for at least 2 touches but not more than 3 (determined by the `length` property of the `targetTouches` property on the `event` object) and then draw lines between these touches, again their coordinates determined via the `pageX` and `pageY` properties on each `Touch` object (again we also need to get them relative to the `canvas` element, so we deduct the `canvas` element's `left` and `top` properties).

Use the same HTML and CSS as in the previous example.

```
var fingerPainter = {};

// As implemented in the previous example.
fingerPainter.getRandomColor = function() {};

fingerPainter.init = function () {
    var canvas = document.querySelector("canvas");
```

```
    var canvasDimensions =
canvas.getBoundingClientRect();
    var ctx = canvas.getContext("2d");
    var shapes = [];
    var currentPoints = null;
    var drawShapes = function () {
        ctx.clearRect(0, 0, 400, 400);

        shapes.forEach(function (shape) {
            var isLine = shape.points.length == 2;
            ctx.fillStyle = ctx.strokeStyle =
shape.color;
            ctx.beginPath();

            shape.points.forEach(function (point)
{
                ctx.lineTo(point.x, point.y);
            });
            ctx.closePath();

            ctx[isLine ? "stroke" : "fill"]();
        });
    };

    var setEmptyPoints = function () {
        currentPoints = {
            points: [],
            color:fingerPainter.getRandomColor()
        };
    };

    setEmptyPoints();

    ctx.lineWidth = 5;

    document.body.addEventListener("touchstart",
function (event) {
        event.preventDefault();
    });

    document.body.addEventListener("touchmove",
function (event) {

        if (event.touches.length < 2 ||
event.touches.length > 3) {
            return;
        }

        event.preventDefault();

        drawShapes();

        ctx.strokeStyle = currentPoints.color;
        ctx.beginPath();
```

```
 Array.prototype.forEach.call(event.touches,
function (touch) {
            ctx.lineTo(touch.pageX -
canvasDimensions.left, touch.pageY -
canvasDimensions.top);
        });
        ctx.closePath();
        ctx.stroke();

    });

    document.body.addEventListener("touchend",
function (event) {
        currentPoints.points =
currentPoints.points.concat(Array.prototype.map.call(event.changedTouches
function (touch) {
            return {
                x: touch.pageX -
canvasDimensions.left,
                y: touch.pageY -
canvasDimensions.top
            };
        })).slice(-3);

        setTimeout(function () {
            currentPoints.points.pop();
        }, 100);

        if (event.targetTouches.length == 0) {
            shapes.push(currentPoints);
            setEmptyPoints();
            drawShapes();
        }
    });
};

document.addEventListener("DOMContentLoaded",
fingerPainter.init);
```

The code is also available at http://alexanderdickson.com/touch-events-how-to/2.html.

This allows us to draw lines (if there are 2 fingers) or triangles (if there are 3 fingers). In the event of a triangle, the shape is filled in. You can see that we used `map()` method of `Array` on the `TouchList` object, which doesn't have that method available on it. Given that the `TouchList` object is an array-like object (it has numbered keys and a `length` property), we can simply call `Array.prototype.map()` and set the new context via `call()`.

Swiping your way to success (Intermediate)

The most famous touch interaction is the swipe. It's simple, intuitive and luckily, it isn't a headache to implement. Swiping can perform all sorts of tasks, most commonly, to change between views (swiping through items in a set, such as images) or switching contexts, such as dragging down an auxiliary view for your application (such as dragging down the notifications bar in iOS).

Getting ready

Swiping is relatively straightforward to achieve by using the events we learned about in the previous recipe. Basically, we shift the desired element by the amount of pixels from the original touch, which acts as the origin. In our example, we will add the ability to our paint application to change the color of the brush (otherwise known as your finger)

How to do it...

Our paint application is kind of useless; unless we want to always paint with random colors, we may want to have a palette of colors to choose from. Unfortunately, there are more colors than we can comfortably show on the screen at once (in addition to the larger tap areas for fingers). So, what on earth could we do?

Let's swipe between the colors available in our paint application. The swipe event is not an event, but you can use `touchstart` and `touchmove` to simulate the swipe behavior observed in native applications.

To get started, we will look at how this was implemented for swiping through the color palette:

- For HTML, use the following markup:

```
<canvas width="400" height="400"></canvas>
<div id="colors">
    <ol></ol>
</div>
```

- For CSS, use the following styles:

```
canvas {
    border: 1px solid #ccc;
}

#colors {
```

```
    width: 400px;
    height: 200px;
    overflow: hidden;
    position: relative;
}

#colors ol {
    position: absolute;
    left: 0;
    top: 0;
    padding: 0;
    white-space: nowrap;
}

#colors ol li {
    width: 60px;
    height: 60px;
    display: inline-block;
    border: 1px solid #333;
    margin: 1px;
}
```

- For JavaScript, use the following code:

```
var fingerPainter = {};

// As implemented in the previous example.
fingerPainter.getRandomColor = function() {};

fingerPainter.colorPicker = (function () {
    var colorsWrapper = null;
    var colorsContainer = null;
    var activeColor;
    var originX = 0;
    var colorsX = 0;
    var colorsOriginX = 0;
    var leftBound = 0;

    var getTransformProperty = function () {
        var property = ["transform",
"webkitTransform",
"mozTransform"].filter(function (prefix) {
            return
document.body.style.hasOwnProperty(prefix);
        })[0];
        getTransformProperty = function () {
            return property;
        };
    };

    var generateColors = function () {
        var colorSwatch;
        // Generate 20 random colors.
        for (var i = 1; i < 20; i++) {
```

```
            colorSwatch =
document.createElement("li");
            colorSwatch.style.backgroundColor
= fingerPainter.getRandomColor();

 colorsContainer.appendChild(colorSwatch);
        }

        // Calculate the furthest distance
the colors can be swiped to the left.
        leftBound =
colorsWrapper.getBoundingClientRect().width -
colorsContainer.getBoundingClientRect().width;
    };

    var setupTouchEvents = function () {

 colorsContainer.addEventListener("click",
function (event) {
            activeColor =
event.target.style.backgroundColor;
        });
        // Swipe through the colors.

 colorsWrapper.addEventListener("touchstart",
function (event) {
            originX =
event.targetTouches[0].pageX;
            colorsOriginX = colorsX;
        });

 colorsWrapper.addEventListener("touchmove",
function (event) {
            // Prevent default behavior of
scrolling elements.
            event.preventDefault();
            var deltaX =
event.targetTouches[0].pageX - originX;
            colorsX = colorsOriginX + deltaX;

            if (colorsX > 0) {
                colorsX = 0;
            } else if (colorsX < leftBound) {
                colorsX = leftBound;
            }

 colorsContainer.style[getTransformProperty()]
= "translate3d(" + colorsX + "px, 0, 0)";
        });
    };
    return {
        init: function () {
            colorsWrapper =
document.querySelector("#colors");
```

```
            colorsContainer =
colorsWrapper.querySelector("ol");
            generateColors();
            setupTouchEvents();
        },
        getActiveColor: function () {
            return activeColor;
        }
    };
})();

fingerPainter.init = function () {
    var canvas =
document.querySelector("canvas");
    var canvasDimensions =
canvas.getBoundingClientRect();
    var ctx = canvas.getContext("2d");
    ctx.lineWidth = 20;
    ctx.lineCap = "round";
    canvas.addEventListener("touchstart",
function (event) {
        ctx.strokeStyle = ctx.fillStyle =
fingerPainter.colorPicker.getActiveColor();
        // Initial dot for single taps.
        ctx.beginPath();
        ctx.arc(event.targetTouches[0].pageX
- canvasDimensions.left,
event.targetTouches[0].pageY -
canvasDimensions.top, 10, 0, Math.PI * 2);
        ctx.fill();
        ctx.closePath();
        ctx.beginPath();
        // Set start point for drawing line.

 ctx.moveTo(event.targetTouches[0].pageX -
canvasDimensions.left,
event.targetTouches[0].pageY -
canvasDimensions.top);
    });
    canvas.addEventListener("touchmove",
function (event) {
        // Prevent default behavior of
scrolling elements.
        event.preventDefault();
        // Get a reference to the first touch
placed.
        var touch = event.touches[0];
        // Draw a line to the new location.
        ctx.lineTo(touch.pageX -
canvasDimensions.left, touch.pageY -
canvasDimensions.top);
        ctx.stroke();
    });
    fingerPainter.colorPicker.init();
};
```

```
document.addEventListener("DOMContentLoaded",
fingerPainter.init);
```

The code can also be found at http://alexanderdickson.com/touch-events-how-to/3.html.

How it works...

In the previous code, we have stored three main points of data; the current left of the element (though this could be derived from the DOM), the start left of the element on `touchstart` (as a point of reference), and the x origin of the touch (in order to determine where the touch originated, and in which direction).

When the `touchstart` event occurs, we store the x origin of the touch. This is so when the user triggers the `touchmove` event, we can compare the x position of that touch to the origin, to determine how far and in which direction the user has swiped.

When combining this swipe delta (the difference in value) to the current left of the element, we can calculate how far the element should be moved courtesy of the user's swipe. We are shifting the element with CSS's `transform` property with a `translate3d()` value, as it gives better performance via hardware acceleration, especially on low-powered devices which are common to touch-enabled devices.

> **Note**
>
> Why do we need to use `getTransformProperty()`? Currently, not all browsers are supporting the bare `transform` property, only vendor-prefixed versions. In order to know which vendor prefix to use, we perform a simple lookup to determine it. The value is then memorized so the lookup doesn't need to be recalculated for future invocations.

There's more...

These techniques can be extended to handle up and down swiping (dealing with the y coordinate instead of the x coordinate), or even more complicated interactions such as top-left to bottom-right swiping.

There is something not quite right with the swipe

You may also notice the technique described previously is quite different to how swiping works in native applications. There is no acceleration or snap-

back (when the swipe has gone beyond the limit of one of the boundaries). This can be implemented with a little help of everybody's friend, that is, math!

The following JavaScript code can be appended to the `setupTouchEvents()` function:

```
// Paul Irish's shim for requestAnimationFrame.
window.requestAnimFrame = (function() {
    return window.requestAnimationFrame       ||
      window.webkitRequestAnimationFrame ||
      window.mozRequestAnimationFrame    ||
          function(callback) {
              window.setTimeout(callback, 1000 /
60);
          };
})();

var lastXs = [0, 0];

colorsWrapper.addEventListener("touchend",
function (event) {
    var lastTwoDelta = lastXs[1] - lastXs[0];
    if (Math.abs(lastTwoDelta) < 3) {
        return;
    }

    var amplitude = lastTwoDelta * 4;
    var startTime = Date.now();

    (function animate() {
        var elapsedTime = Date.now() - startTime;
        var timeDelta = Math.min(1, elapsedTime /
300);
        var distanceDelta = amplitude * timeDelta;
        colorsX += distanceDelta;
        amplitude -= distanceDelta;

        if (colorsX > 0) {
            colorsX = 0;
        } else if (colorsX < leftBound) {
            colorsX = leftBound;
        }


 colorsContainer.style[getTransformProperty()] =
"translate3d(" + colorsX + "px, 0, 0)";

        if (timeDelta == 1) {
            return;
        }
        requestAnimFrame(animate);
    })();
```

```
});
```

The next piece of code needs to be appended to the `touchmove` handler inside `setupTouchEvents()`.

```
lastXs.push(event.targetTouches[0].pageX);
lastXs = lastXs.slice(-2);
```

Finally, this piece of code needs to be appended to the `touchstart` event inside `setupTouchEvents()`.

```
lastXs = [0, 0];
```

The code is also available at http://alexanderdickson.com/touch-events-how-to/4.html.

This code that resides in the `touchmove` handler will store the last two x positions (`lastXs.slice(-2)` will return the last two elements of the `lastXs` array) that the `touchmove` handler handled. This is important, as we will use the delta between them to determine how much the user swiped (or if it's large enough, flicked).

When the user removes his/her finger, we grab this delta and choose a distance that we want to animate the colors along. For this example, we multiplied the delta by four in order to determine this distance. This number was chosen because it looks natural enough. Feel free to experiment to see what you prefer.

Now that we know the distance we want to animate the colors along, we simply set up a tick-style function using `requestAnimationFrame()` in order to perform our animation. We use the time delta (the difference in time between the animation step and when the animation started) to determine how far to animate the colors. This has the advantage of moving the colors the same distance and time no matter how slow the computer can call the animation function.

After throwing in some bounds checking (you could also implement the iOS rubber band effect here), we animate the colors and then check to see if 300 ms has elapsed, in which we kill the animation. There is a bit of repetition here with the bounds checking and the setting of the x value that already exists in the `touchmove` handler, so it'd be worthwhile setting up general functions to perform the tasks.

You can perform all sorts of neat things in order to make web applications feel like their native counterparts, which generally get this functionality for free using built-in components. You also have the ability to modify the

experience to suit your application, so experiment and see what makes sense for your application.

Custom events for swipe

In addition to handling swiping on a per pixel basis, it's possible to define a custom event, for example, `swipeleft` that will detect a swipe to the left and trigger the appropriate listeners. This will use custom events, and you can see it will make it much easier to handle these types of touch events.

```
var touchXOrigins = {};

document.addEventListener("touchstart",
function(event) {
    for (var i = 0; i < event.touches.length;
i++) {

 touchXOrigins[event.touches[i].identifier] =
event.pageX;
    }
});


document.addEventListener("touchmove",
function(event) {
    for (var i = 0; i < event.touches.length;
i++) {
        var touch = event.touches[i];
        var originX =
touchXOrigins[touch.identifier];

        if (originX !== undefined && touch.pageX
< originX - 200) {
            var swipeLeftEvent = new
Event("Event");
            swipeLeftEvent.initEvent("swipeleft",
true, true);
            // Propagate event.preventDefault().
            !
event.target.dispatchEvent(swipeLeftEvent) &&
event.preventDefault();
            // Stop it being triggered multiple
times.
            delete
touchXOrigins[touch.identifier];
        }
    }
});
```

This code will now let you listen to the `swipeleft` event on any element, greatly simplifying the handling of that event. For example, if we had an

application that lets the user swipe to the left to go to the previous photo, we could use this far simpler code:

```
document.getElementById("photo-
viewer").addEventListener("swipeleft",
function(event) {
    event.preventDefault();
    photoViewer.previousPhoto();
});
```

In this code, we use the familiar `touchstart` and `touchmove` events to determine if the user has swiped left (one of his/her fingers has moved 200 px to the left).

If that condition is met, we dispatch the event on the element. It's easy to see how this could be adapted for `swiperight`, `swipeup`, `swipedown` or even more exotic events such as `swipeleftrightandshakeitallabout`.

Note

Of course, once we abstract this into the `swipeleft` event, we lose the fine-grained information such as the current touch's exact x position. It's important to still move the element in response to the users' touch, to give your users a visual cue that the element can be swiped. This should be used in conjunction with a custom event such as `swipeleft`.

Welcome gestures (Intermediate)

We have looked at many ways of interacting with a web application by using our fingers. Now, we will look at one final method named gestures. You could call the swiping we looked at the last recipe as a type of basic gesture. We will also take a look at pinching, rotating, and a few others.

Getting ready

Some browsers, for example, Mobile Safari provide new events for gestures, and these are:

Event name	Triggered when a multi-touch event…
`gesturestart`	…starts.
`gesturechange`	…moves.
`gestureend`	…ends (who'd have guessed?)

The events are really just sugar for the previous, widely supported events we have looked at. It's easy enough to return early from handling those events, if the number of active touches is less than two.

```
document.body.addEventListener("touchmove",
function(event) {
    if (event.touches.length < 2) {
        return;
    }
    // This is identical to gesturechange event.
});
```

In order to handle the scaling- and rotation-related events, we need to examine the data provided by the `Touch` objects. On some browsers, the `scale` (determined by pinching) and `rotation` (determined by rotating two fingers) properties are made available on the `event` object as a convenience. On browsers that don't support these properties, we can calculate the same with a bit of math.

Remember our Finger Painter application? Well, your boss called and said they want you to implement scaling and rotating of the canvas. Don't fret, chill out because you're about to find out that it's not difficult at all.

Rotating and/or scaling elements is relatively straightforward to implement, thanks to CSS's `transform()` property, which has the `scale()` and `rotate()` values which means we can map the data from the touch event directly into CSS values. Let's go ahead and implement that now. The CSS and HTML is again the same as before:

How to do it...

1. We'll be writing a handful of JavaScript to augment our existing code from the previous recipes:

```
fingerPainter.rotatorScaler = {
    init: function() {

        var getTransformProperty = function
() {
            var property = ["transform",
"webkitTransform",
"mozTransform"].filter(function (prefix) {
                return
document.body.style.hasOwnProperty(prefix);
            })[0];

            getTransformProperty = function
() {
                return property;
            };
        };

        var rotation = 0;
        var scale = 1;


 fingerPainter.canvas.addEventListener("touchmove",
function(event) {

            var canvasStyles = [];
            event.preventDefault();

            if (event.rotation) {
                canvasStyles.push("rotate(" +
(rotation + event.rotation) + "deg)");
            }

            if (event.scale) {
                canvasStyles.push("scale(" +
(scale * event.scale) + ")");
```

```
            }


            canvasStyles.length &&
(this.style[getTransformProperty()] =
canvasStyles.join(" "));

        });


 fingerPainter.canvas.addEventListener("touchend",
function(event) {
            rotation = event.rotation;
            scale = event.scale;

        });
    }
};
```

2. Simply invoke `fingerPainter.rotatorScaler()` in your `init:` method and assign a reference to the canvas to `fingerPainter.canvas`.

Have a look at the code at http://alexanderdickson.com/touch-events-how-to/5.html.

How it works...

It's relatively simple to see all that is involved is reading the two properties and mapping them via the `transform` CSS property onto the `target` element.

To make the user's rotation affect an element, we simply set:

```
canvas.addEventListener("touchmove",
function(event) {
    this.style[getTransformProperty()] =
"rotate(" + event.rotation + "deg)";
});
```

However, this will always treat 0 degrees as the origin. In order to have future rotations consider the previously rotated value, we need to keep the previous rotation value around and calculate the new one based on that (we could read the current value with CSS, but it's a lot more complicated than simply storing it in a variable).

Using the `scale` property will work in much the same way. However, instead of adding its new value to the old value, we multiply it, as that is how the `scale` function works (returns a number between zero and one).

Combining these two properties makes it super simple to add rotation and scale gestures to your application. You could rotate/scale a `canvas` like in this example, or use `rotation` and `scale` to perform more exotic tasks. You can even invent your own type of gestures.

There's more...

This is all well and good, but you may have noticed that it doesn't support devices which don't make the `scale` and `rotation` properties available. Bummer. However, we're not out of luck as our friend math can deduce these values for the browsers that don't support them.

For determining the scale factor, we can use the Distance Formula, derived from the Pythagorean Theorem. You may remember it from high school, or if you're like me, you had to look it up.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

We can plug the values from the touches into this equation and get the distance between the touches, in order to calculate the initial and subsequent scale distance. After dividing the scale distance over the original, we get a normalized factor between zero and one, just as if we had a browser that supported the `scale` property.

Here is our function for determining the distance between 2 touches:

```
var getDistanceBetweenTouches = function(touch0,
touch1) {
    return Math.sqrt(
        Math.pow(touch0.pageX - touch1.pageX, 2)
+
        Math.pow(touch0.pageY - touch1.pageY, 2)
    );
};
```

On the `touchstart` event, we save the original distance. This is only required if the browser doesn't natively support the `scale` property (which we detect):

```
var startDistanceBetweenTouches = null;

fingerPainter.canvas.addEventListener("touchstart",
function(event) {

    if (event.scale) {
        return;
    }
```

```
    startDistanceBetweenTouches =
getDistanceBetweenTouches (event.touches[0],
event.touches[1]);
});
```

Finally, in the `touchmove` event, we find the scale factor either natively or from our previous calculation:

```
fingerPainter.canvas.addEventListener("touchmove",
function(event) {
    var scale = event.scale ||
      getDistanceBetweenTouches(event.touches[0],
event.touches[1]) / startDistanceBetweenTouches;

});
```

Determining the rotation is done in a similar way, with help from `Math.arctan2()`, which gets the arctangent of the quotient of its arguments as radians:

```
var radians = Math.arctan2(y, x);
```

Although you can pass radians to CSS's `rotate()` value with the `rad` unit, that's not what the `rotation` property normally contains, so let's convert it to degrees, so that we're always dealing with the same unit:

```
var degrees = radians * (180 / Math.PI);
```

This gives us the knowledge to construct a function for determining the rotation value between 2 touches:

```
var getRotationAngleBetweenTouches =
function(touch0, touch1) {
    return Math.arctan2(touch0.pageY -
touch1.pageY, touch0.pageX - touch1.pageX) *
180 / Math.PI;
};
```

Setting this up is similar to using `scale`, that is, we check to see if the property is supported, and if not, we store the original rotation on `touchstart`:

```
var startRotationAngle = null;

fingerPainter.canvas.addEventListener("touchstart",
function(event) {

    if (event.rotation) {
```

```
            return;
        }

        startRotationAngleBetweenTouches =
    getRotationAngleBetweenTouches (event.touches[0],
    event.touches[1]);
    });
```

Again, similar to deriving `scale`, we then calculate the actual rotation in the `touchmove` handler:

```
    fingerPainter.canvas.addEventListener("touchmove",
    function(event) {
        var rotation = event.rotation ||
          getRotationAngleBetweenTouches
    (event.touches[0], event.touches[1]) -
        startRotationAngleBetweenTouches;
    });
```

Note

Instead of dividing over the initial value like we did for the `scale` property, we subtract the initial value. This leaves us with the correct rotation angle.

You can now successfully use `scale` and `rotate` to enhance your web applications' interactions. Scaling and rotation on screen elements is the obvious pattern, but can you think of an interaction pattern using these methods unique to your application?

A custom gesture can go a long way (Advanced)

Nobody's perfect, and thus for our paint application, we need a way of clearing the `canvas`. We can invent our own gesture for this, as anything that can be described in code can become a gesture.

Getting ready

For our use case, we will use two fingers swiping up and away from each other over the canvas. This will look like tracing the outline of an upside down equilateral triangle. Have a look at the following photos of the before and after state of the gesture:

How to do it...

We have built upon the basic Finger Painter application with some modifications, so it won't try and draw while the user attempts to do the custom gesture.

```
var fingerPainter = {};

// As implemented in the previous example.
fingerPainter.getRandomColor = function() {};

fingerPainter.clearer = {
    init: function () {

        var VERTICAL_MOVEMENT_REQUIRED = 150;
        var HORIZONTAL_MOVEMENT_REQUIRED = 100;
        var factor = 1;
        var touchesOrigin = {};
```

```
        var getDistanceBetweenTouches = function
(touch0, touch1) {
            return Math.sqrt(
                Math.pow(touch0.pageX -
touch1.pageX, 2) +
                Math.pow(touch0.pageY -
touch1.pageY, 2)
            );
        };
  // As implemented in the previous example.
        var getTransitionProperty = function ()
{};


 fingerPainter.canvas.addEventListener("touchstart",
function (event) {

 Array.prototype.forEach.call(event.touches,
function (touch) {
                touchesOrigin[touch.identifier] =
{
                    pageX: touch.pageX,
                    pageY: touch.pageY
                };
            });
            factor = 1;
        });


 fingerPainter.canvas.addEventListener("touchmove",
function (event) {
            var touches = event.touches;

            if (Object.keys(touchesOrigin).length
< 2 || touches.length < 2) {
                return;
            }

            event.preventDefault();

            var distanceBetweenTouches =
Math.min(HORIZONTAL_MOVEMENT_REQUIRED,
getDistanceBetweenTouches(touches[0], touches[1])
-
getDistanceBetweenTouches(touchesOrigin[touches[0].identifier],
touchesOrigin[touches[1].identifier]));
            var touch0VerticalDelta =
Math.min(VERTICAL_MOVEMENT_REQUIRED,
touchesOrigin[touches[0].identifier].pageY -
touches[0].pageY);
            var touch1VerticalDelta =
Math.min(VERTICAL_MOVEMENT_REQUIRED,
touchesOrigin[touches[1].identifier].pageY -
touches[1].pageY);
```

```
            // User feedback.
            factor = 1 - Math.min(1,
(distanceBetweenTouches * 5 + touch0VerticalDelta
+ touch1VerticalDelta) /
(VERTICAL_MOVEMENT_REQUIRED +
VERTICAL_MOVEMENT_REQUIRED +
HORIZONTAL_MOVEMENT_REQUIRED * 5));

            fingerPainter.canvas.style.opacity =
factor;

        });


 fingerPainter.canvas.addEventListener("touchend",
function (event) {
            if (factor == 0) {

 fingerPainter.canvas.getContext("2d").clearRect(0,
0, fingerPainter.canvas.width,
fingerPainter.canvas.height);
            }


 fingerPainter.canvas.style[getTransitionProperty()]
= factor == 0 ? "none" : "opacity .3s ease-out"
            fingerPainter.canvas.style.opacity =
1;
        });
    }
};


fingerPainter.init = function () {
    var canvas = fingerPainter.canvas =
document.querySelector("canvas");
    var canvasDimensions =
canvas.getBoundingClientRect();
    var ctx = canvas.getContext("2d");
    ctx.lineWidth = 20;
    ctx.lineCap = "round";

    canvas.addEventListener("touchstart",
function (event) {

        // Only draw if one touch.
        if (event.touches.length > 1) {
            return;
        }

        ctx.strokeStyle = ctx.fillStyle =
fingerPainter.getRandomColor();

        // Initial dot for single taps.
        ctx.beginPath();
```

```
        ctx.arc(event.targetTouches[0].pageX -
canvasDimensions.left,
event.targetTouches[0].pageY -
canvasDimensions.top, 10, 0, Math.PI * 2);
        ctx.fill();
        ctx.closePath();
        ctx.beginPath();

        // Set start point for drawing line.
        ctx.moveTo(event.targetTouches[0].pageX -
canvasDimensions.left,
event.targetTouches[0].pageY -
canvasDimensions.top);
    });

    canvas.addEventListener("touchmove", function
(event) {

        var touch = event.touches[0];

        // Only draw if one touch.
        if (event.touches.length > 1) {
            return;
        }

        // Prevent default behavior of scrolling
elements.
        event.preventDefault();

        // Draw a line to the new location.
        ctx.lineTo(touch.pageX -
canvasDimensions.left, touch.pageY -
canvasDimensions.top);
        ctx.stroke();
    });

    fingerPainter.clearer.init();
};

document.addEventListener("DOMContentLoaded",
fingerPainter.init);
```

That's a lot of code! When you're programming custom gestures, don't be alarmed if it takes a sizeable amount of code to describe the gesture. Obviously, the more complicated gesture, the more complicated code is needed to back it.

The previous code should look mostly familiar; we are checking the vertical distance moved as well as the distance between the two touches to determine if the gesture has been made. If the touches go up and away from each other enough to hit the thresholds, then we call a function to clear the canvas.

In order to give the user feedback that their gesture is doing something, we fade the canvas as the gesture is in progress. This is often a good idea with complicated gestures, as the visual cue can make learning and using the gestures much easier. If the user removes his/her fingers before the gesture has been activated, we fade it back to normal opacity.

An explanation of the `getDistanceBetweenTouches()` function was provided in the previous recipe. When using custom gestures, try to keep them simple as it makes it easier for your users to learn and use; furthermore, it will result in less code for you to write and maintain.

There's more...

In developing your web application using the touch events, it's important to consider using a preexisting library rather than rolling your own. The benefits of using a popular preexisting solution are numerous, such as bug-fixes and community support.

However, knowing how the library works internally, for example, understanding the events on a low-level basis is of tremendous benefit, such as when it comes to debugging existing libraries, or simply doing quick and dirty jobs where the overhead of a library is disadvantageous.