Practical Approaches For

Designing Accessible Websites

SMASHING
MAGAZINE

Imprint

ISBN (PDF): 978-3-945749-24-1
Cover Design: Veerle Pieters
eBook Strategy and Editing: Vitaly Friedman
Technical Editing: Cosima Mielke
Planning and Quality Control: Vitaly Friedman, Iris Lješnjanin
Tools: Elja Friedman
Syntax Highlighting: Prism by Lea Verou
Idea & Concept: Smashing Magazine GmbH

About This Book

We design with viewports in mind, keep track of loading times, and hunt down even the smallest browser bugs — all to create the best possible user experience. But despite all these efforts to constantly improve our products, there's still one aspect that, unfortunately, comes up short quite often: accessibility.

Have you ever tried to navigate your website using only your keyboard? Your mobile application with a screen reader? And do you consider your color choices with accessibility in mind? With the help of this eBook, you will gain a deeper understanding of common accessibility pitfalls and learn to circumvent them to create a better experience for everyone.

The first step towards making informed decisions about accessible design, though, is fully grasping how the underlying technology works. That's why we'll start off this eBook with a closer look at accessibility APIs. Based on that, our authors consider UX principles for accessibility and share best coding practices that guarantee a better and smoother interaction, no matter how a user interacts with your content. Finally, we cover strategies and tools to simulate how someone with visual impairments experiences your website, as well as key lessons from designing for older people. As you will see, with accessibility in mind, we can serve many more people than we already do. It's about time to finally remove the existing barriers and build a more inclusive web — the effort is reasonable, and all our users will benefit from it.

TABLE OF CONTENTS

Accessibility APIs: A Key To Web Accessibility

BY LÉONIE WATSON &
CHAALS MCCATHIE NEVILE ❧

Web accessibility is about people. Successful web accessibility is about anticipating the different needs of all sorts of people, understanding your fellow web users and the different ways they consume information, empathizing with them and their sense of what is convenient and what frustratingly unnecessary barriers you could help them to avoid.

Armed with this understanding, accessibility becomes a cold, hard technical challenge. A firm grasp of the technology is paramount to making informed decisions about accessible design.

How do assistive technologies present a web application to make it accessible for their users? Where do they get the information they need? One of the keys is a technology known as the accessibility API (or accessibility application programming interface, to use its full formal title).

Reading The Screen

To understand the role of an accessibility API in making web applications accessible, it helps to know a bit about how assistive technologies provide access to applications and how that has evolved over time.

A WORLD OF TEXT

With the text-based DOS operating system, the characters on the screen and the cursor position were held in a screen buffer in the computer's memory. Assistive technologies could obtain this information by reading directly from the screen buffer or by intercepting signals being sent to a monitor. The information could then be manipulated — for example, magnified or converted into an alternative format such as synthetic speech.

GETTING GRAPHIC

The arrival of graphical interfaces such as OS/2, Mac OS and Windows meant that key information about what was on the screen could no longer be simply read from a buffer. Everything was now drawn on screen as a picture, including pictures of text. So, assistive technologies on those platforms had to find a new way to obtain information from the interface.

They dealt with this by intercepting the drawing calls sent to the graphics engine and using that information to create an alternate off-screen version of the interface. As applications made drawing calls through the graphics engine to draw text, carets, text highlights, drop-down windows and so on, information about the appearance of objects on the screen could be captured and stored in a database called an off-screen model. That model could be read by screen readers or used by screen magnifiers to zoom in on the user's current point of focus within the interface. Rich Schwerdtfeger's seminal 1991 article in Byte, "Making the GUI Talk[1]," describes the then-emerging paradigm in detail.

OFF-SCREEN MODELS

Recognizing the objects in this off-screen model was done through heuristic analysis. For example, the operating system might issue instructions to draw a rectangle on screen, with a border and some shapes inside it that represent text. A human might look at that object (in the context of other information on screen) and correctly deduce it is a button. The heuristics required for an assistive technology to make the same deduction are actually very complex, which causes some problems.

To inform a user about an object, an assistive technology would try to determine what the object is by looking for identifying information. For example, in a Windows application, the screen reader might present the Window Class name of an object. The assistive technology would also try to obtain information about the state of an object by the way it is drawn — for example, tracking highlighting might help deduce when an object has been selected. This works when an object's role or state can easily be determined, but in many cases the relevant information is unclear, ambiguous or not available programmatically.

This reverse engineering of information is both fallible and restrictive. An assistive technology could implement support for a new feature only once it had been in-

---

[1] http://www.paciellogroup.com/blog/2015/01/making-the-gui-talk-1991-by-rich-schwerdtfeger/

troduced into the operating system or application. An object might not convey useful information, and in any case it took some time to identify it, develop the heuristics needed to support it and then ship a new version of the screen reader. This created a delay between the introduction of new features and assistive technology's ability to support it.

The off-screen model needs to shadow the graphics engine, but the engines don't make this easy. The off-screen model has to independently calculate things like white-space management and alignment coordination, and errors would almost inevitably mount up. These errors could result in anomalies in the information conveyed to assistive technology users or in garbage buildup and memory leaks that lead to crashes.

Accessibility APIs

From the late 1990s, operating system accessibility APIs were introduced as a more reliable way to pass information to assistive technologies. Instead of applying complex heuristics to determine what an on-screen object might be, assistive technologies could query the accessibility API for specific information about each object. Authors could now provide the necessary information about an application in a form that they knew assistive technology would understand.

An accessibility API represents objects in a user interface, exposing information about each object within the application. Typically, there are several pieces of information for an object, including:

- **its role** (for example, it might be a button, an application window or an image);
- **a name** that identifies it within the interface (if there is a visible label like text on a button, this will typically be its name, but it could be encoded directly in the object);
- **its state** or current condition (for example, a checkbox might currently be selected, partially selected or not selected).

The first platform accessibility API, Microsoft Active Accessibility (MSAA), was made available in a 1997 update to Windows 95. MSAA provided information about the role and state of objects and some of their properties. But it gave no access to things like text formatting, and the relationships between objects in the interface were difficult or impossible to determine.

In 1998, IBM and Sun Microsystems built a cross-platform accessibility API for Java. Java Swing 1.0 gave access to rich text information, relationships, tables, hyperlinks and more. The Java Jive screen reader, built on this platform, was the first time a screen reader's information about the components of a user interface included role, state and associated properties, as well as rich text formatting details.

Notably, Java Jive was written by three developers in roughly five months; developing a screen reader through an off-screen model typically took *several years*.

ACCESSIBILITY APIS GO MAINSTREAM

In 2001 the Assistive Technology Service Provider Interface (AT-SPI) for Linux was released, based on the work done on Java, and in 2002 Apple included the NSAccessibility protocol with Mac OS X (10.2 Jaguar).

Meanwhile on Windows, the situation was getting complicated. Microsoft shipped the User Interface Automation (UIA) API as part of Windows 7, while IBM released IAccessible2 as an open standard for Windows and Linux, again evolved from the work done on Java.

Accessibility APIs existed for mobile platforms before touchscreen smartphones became dominant, but in 2009 Apple added the UI Accessibility API to iOS 3, and Android 1.6 (Donut) shipped with the Accessibility Framework.

By the beginning of 2015, Chrome OS stood out as the most mainstream platform lacking a standard accessibility API. But Google was beta testing its Automation API, intended to fill that gap in the platform.

MODERN ACCESSIBILITY APIS

In modern accessibility APIs, user interfaces are represented as a hierarchical tree. For example, an application window would contain several objects, the first of which might be a menu bar. The menu bar would contain a number of menus, each of which contains a number of menu items, and so on. The accessibility API describes an object’s relationship to other objects to provide context. For example, a radio button would probably be one “sibling” within a group.

Other features such as information about text formatting, applicable headers for content sections or table cells and things such as event notifications have all become commonplace in modern accessibility APIs.

Assistive technologies now make standard method calls to the operating system to get information about the objects on the screen. This is far more reliable, and far more efficient, than intercepting low-level operating system messages and trying to deconstruct them into something meaningful.

From The Web To The Accessibility API

In browsers, the platform accessibility API is used both to make information about the browser itself available to assistive technologies and to expose information about the currently rendered content.

Browsers typically support one or more of the available accessibility APIs for the platform they're running on. For example, on Windows, Firefox, Chrome, Opera and Yandex support MSAA/IAccessible and IAccessible2, while Internet Explorer supports MSAA/IAccessible and UIAExpress. Safari and Chrome support NSAccessibility on OS X and UIAccessibility on iOS.

The browser uses the HTML DOM, along with further information derived from CSS, to generate an accessibility tree hierarchy of the content it is displaying, and it passes that information to the platform accessibility API. Information such as the role, name and state of each object in the content, as well as how it relates to other objects in the content, can then be queried by assistive technologies.

Let's see how this works with some HTML:

```
<p><img src="mc.png" alt="My cat"
longdesc="meeow.html">Rocks!</p>
```

We have an image, rendered as part of a paragraph. A browser exposes several pieces of information about the image to the accessibility API:

1. It has a role of "image" (or "graphic" — details vary between platforms). This is implicitly determined from the fact that it is an HTML `img` element.
2. Its name is "My cat". For images, the name is typically derived from the `alt` attribute.
3. A description is available on request, at the URL `meeow.html` (at the same "base" as the image).
4. The parent is a paragraph element, with a role of "text".
5. The image has a "sibling" in the same container, the text node "Rocks!"

An assistive technology would query the accessibility API for this information, which it would present so the user can interact with it. For example, a screen reader might announce, "Graphic: My cat. Description available."

(Does a cat picture need a full description? Perhaps not, but try explaining that to people who really want to tell you just how amazing and talented their feline friends actually are — or those of their readers who *want* to know all about what *this* cat looks like! Meanwhile, the philistines among us can ignore the extra information.)

ROLES

Most HTML elements have what are called “roles,” which are a way of describing elements. If you are familiar with WAI-ARIA, you will be aware of the `role` attribute, which sets a role explicitly. Most elements already have implicit roles, however, which go along with the element type. For example:

- `<ul>` and `<ol>` have “list” as implicit role,
- `<a>` has “link” or “hyperlink” as implicit role,
- `<body>` has “document” as implicit role.

These role mappings are being standardized and documented in the W3C’s “HTML Accessibility API Mappings[2]” specification.

NAMES

While roles are typically derived from the type of HTML element, the name (sometimes referred to as the “accessible name”) of an object often comes from one of several different sources. In the case of a form field, the name is usually taken from the label associated with the field:

---

[2] http://rawgit.com/w3c/aria/master/html-aam/html-aam.html

```
<input type="radio" id="tequila" name="drinks"
checked>
<label for="tequila">Reposado</label>
```

In this example, a button has the “radio button” role. Its accessible name will be “Reposado,” the text content of the label element. So, when a speech-recognition tool is instructed to “Click Radio button Reposado,” it can target the correct object within the interface.

The `checked` attribute indicates the state of the button, so that a screen reader can announce “Radio button Reposado Checked” or allow a user to navigate directly between the checked options in order to rapidly review a form that contains multiple sets of radio buttons.

Authors have an important role to play, providing the key information that assistive technologies need. If authors don’t do the “right thing,” assistive technologies must look in other places to try to get an accessible name — if there is no label, then a title or some text content might be near the radio button, or its relationship to other elements might help the user through context.

It is important to note that authors should not rely on an assistive technology’s ability to do this, because it is generally unreliable. It is a “repair” strategy that gives assistive technology users some chance of using a poorly authored page or website, such as the following:

```
<p>How good is reposado?<br>
<!--BAD CODE EXAMPLE: DON'T DO THIS-->
<input type="radio" id="fantastic" name="reposado"
checked >
```

```
<label for="reposado">Fantastic</label><br>
<input type="radio" id="notBad" name="tequila"><br>
<input type="radio" id="meh" name="tequila"
title="meh"> Meh
```

Faced with this case, a screen reader might provide information such as “second of three options,” based on information that the browser provides to the accessibility API about the form. Little else can be determined reliably from the code, though.

Nothing in the code associates the question with the set of radio buttons, and nothing informs the browser of what the accessible name for the first two buttons should be. The `for` and `id` attributes of the `<label>` and `<input>` for the first button do not share a common value, and nothing associates the nearby text content with the second button. The browser could use the title of the third button as an accessible name, but it duplicates the nearby text and unnecessarily bloats the code.

A well-authored version of this would use the `fieldset` element to group the radio buttons and use a `legend` element to associate the question with the group. Each of the buttons would also have a properly associated label.

```
<fieldset><legend>How good is reposado?</legend>
<!-- THIS IS A BETTER WAY TO CODE THE EXAMPLE -->
<input type="radio" id="fantastic" name="reposado"
checked>
<label for="fantastic">Fantastic</label><br>
<input type="radio" id="notBad" name="reposado">
```

```
<label for="notBad">Not bad</label><br>
<input type="radio" id="meh" name="reposado">
<label for="meh">Meh</label><br>
</fieldset>
```

Making this information available through the accessibility API is more efficient and less prone to error than relying on assistive technologies to create an off-screen model or guess at the information they need.

Conclusion

Today's technologies — operating systems, browsers and assistive technologies — work together to extract accessibility information from a web interface and appropriately present it to the user. If appropriate content semantics are not available, then assistive technologies will use old and unreliable techniques to make the interface usable.

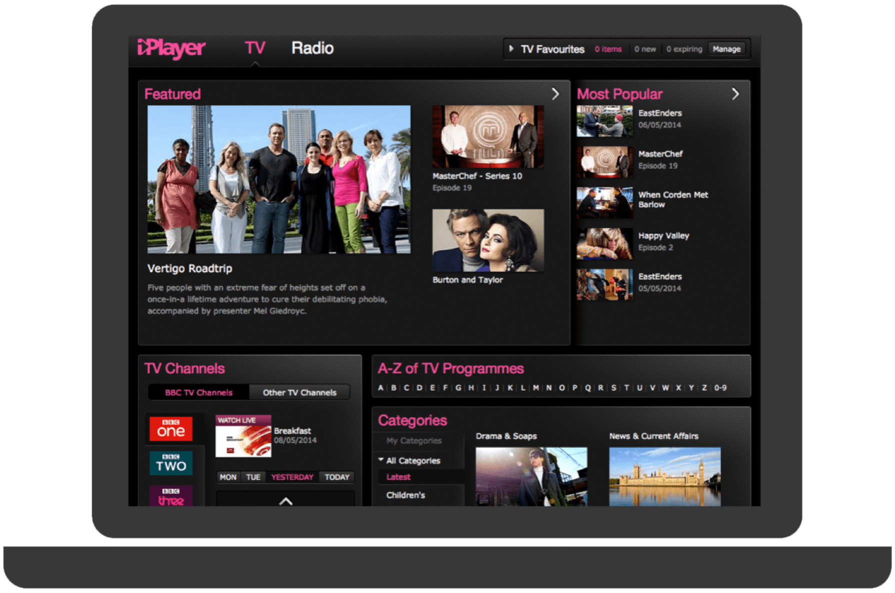
The value of accessibility APIs is in allowing the operating system, browser and assistive technology to efficiently and reliably give users the information they need. It is now easy to make an interface developed with well-written HTML, CSS and JavaScript very accessible and usable for assistive technology users. A big part of accessibility is, therefore, an easily met responsibility of web developers: Know your job, use your tools well, and many pieces will fall into place as if by magic. ❧

With thanks to Rich Schwerdtfeger, Steve Faulkner and Dominic Mazzoni.

Accessibility Originates With UX: A BBC iPlayer Case Study

BY HENNY SWAN ❧

Not long after I started working at the BBC, I fielded a complaint from a screen reader user who was having trouble finding a favorite show via the BBC iPlayer's home page[3]. The website had recently undergone an independent accessibility audit which indicated that, other than the odd minor issue here and there, it was reasonably accessible.



Five people with an extreme fear of heights set off on a once-in-a lifetime adventure to cure their debilitating phobia, accompanied by presenter Mel Giedroyc.

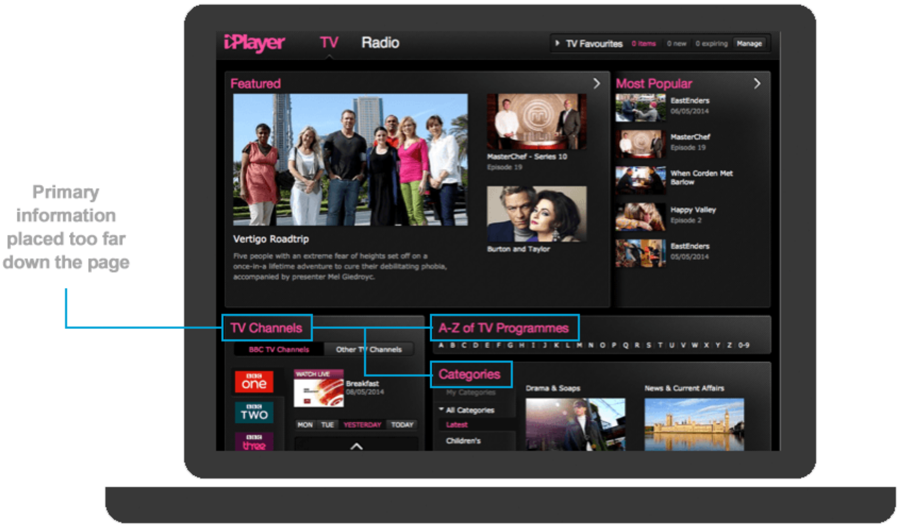
iPlayer's old home page.

---

[3] http://www.bbc.co.uk/iplayer

I called the customer to establish what exactly the problem was, and together we navigated the home page using a screen reader. It was at that point I realized that, while all of the traditional ingredients of an accessible page were in place — headings, WAI ARIA Landmarks[4], text alternatives and so on — it wasn't very usable for a screen reader user.

The first issue was that the subnavigation was made up of only two links: "TV" and "Radio," with links to other key areas such as "Categories," "Channels" and "A to Z" buried further down the content order of the page, making them harder for the user to find.


Primary information placed too far down the page

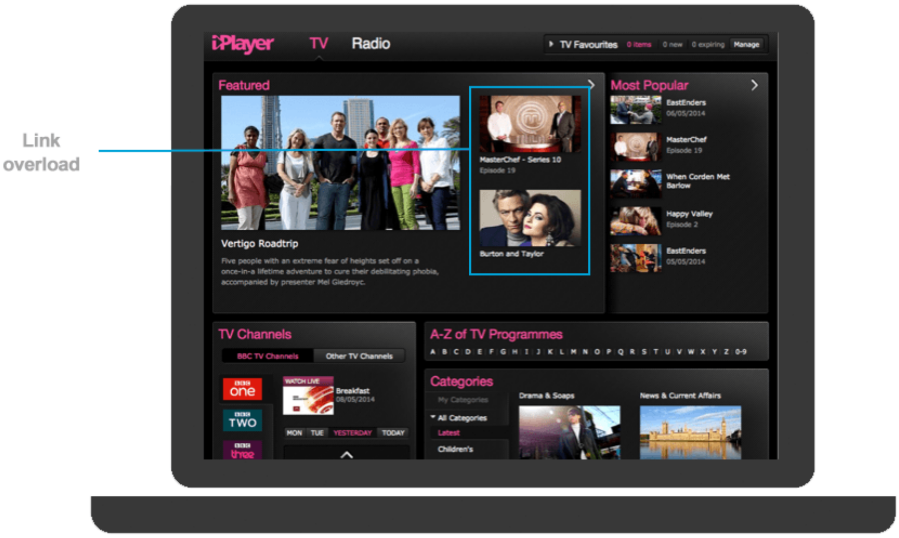
iPlayer's old home page showing "Categories," "Channels" and "A to Z" far down the content order.

The second issue was how verbose the page was to the screen reader user. Instead of hearing a link to a program

---

[4] http://www.w3.org/TR/wai-aria/roles#landmark_roles

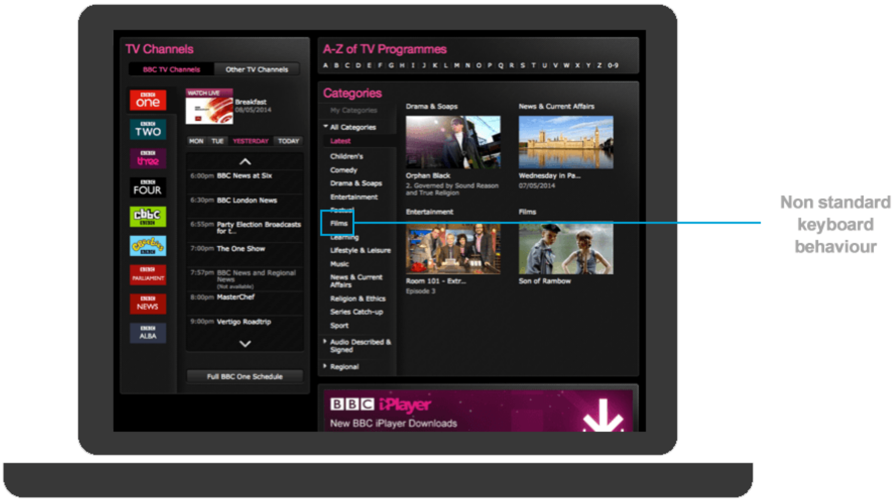
once, the program would be announced twice because the thumbnail image and the heading for the program were presented as two separate links. This made the page longer to listen to and was confusing because links to the same destination were worded differently.



Five people with an extreme fear of heights set off on a once-in-a lifetime adventure to cure their debilitating phobia, accompanied by presenter Mel Giedroyc.

iPlayer's old home page showing duplicate links.

Finally, keyboard access on the page was illogical. In the "Categories" area, for example, a single click on a category would reveal four items in a panel next to it. To access the full list of items in that category, you had to click again on the same link to be taken to a listing page. This was a major hurdle for the user and the place where the customer I was talking to gave up using the application altogether.



Non standard keyboard behaviour

iPlayer's old home page showing the "Categories" links highlighted.

It was clear that, while the website had been built with accessibility in mind, it hadn't been designed with accessibility in mind and this is where the issues originated.

The Challenge

At the BBC, a number of internal standards and guidelines are in place that teams are required to follow when delivering accessible website and mobile applications. Key ones are:

- Accessibility Standards and Guidelines[5],
- Screen Reader Testing Guidelines[6],
- Mobile Accessibility Standards and Guidelines[7].

---

[5] http://www.bbc.co.uk/guidelines/futuremedia/accessibility/
[6] http://www.bbc.co.uk/guidelines/futuremedia/accessibility/screenreader.shtml

There is also a strong culture of accessibility; the BBC is a publicly funded organization[8], and accessibility is considered central to its remit and is a stronger driver than any legal requirement. So, how did this happen?

Part of the issue is that standards and guidelines tend to focus more on code than design, more on output than outcome, more on compliance than experience. As such, technically compliant pages could be built that are not the most usable for disabled users.

It may not seem immediately obvious, but visual design can have a massive impact on users who cannot see the page. I often find that mobile applications and websites that are problematic to make accessible are the ones where the visual design, by dictating structure, does not allow it.

This does not mean that standards and guidelines are redundant — far from it. But what we have found at the BBC is that standards need to sit within, and inform, an accessibility framework that runs through product management, user experience, development and quality assurance. As such, accessibility originates with UX. Most of the thinking and requirements should be considered up front so that poor accessibility isn't designed in.

While redesigning the BBC iPlayer website, renewed focus was given to inclusive design, which, while adhering to the BBC's standards and guidelines, is driven by four principles (more on that below). We then distilled

---

7. http://www.bbc.co.uk/guidelines/futuremedia/accessibility/mobile_access.shtml
8. http://www.bbc.co.uk/corporate2/insidethebbc/whoweare

our standards and guidelines to create a focused list of requirements for the UX to follow. We also started to train designers to annotate their own designs for accessibility.

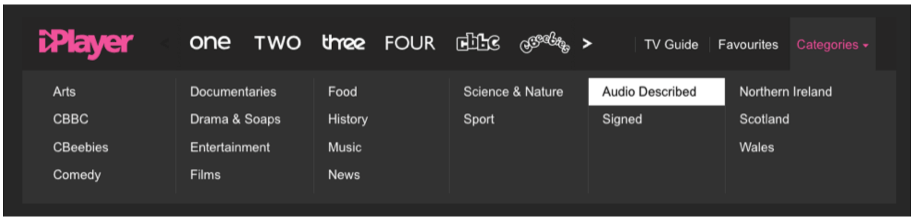
UX Principles

Our four main principles are the following:

- Give users choice.
- Put users in control.
- Design with familiarity in mind.
- Prioritize features that add value.

GIVE USERS CHOICE

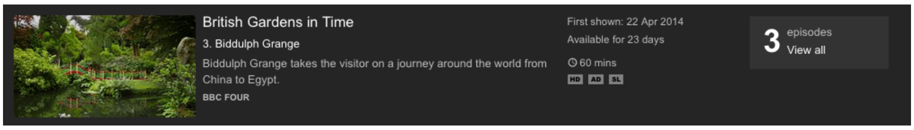
Never assume that just because a user can access content one way that they want to access content in that one way. Because BBC's iPlayer has "audio described" and "sign language" formats, it was never in any doubt that both of these should have their own dedicated listing pages, accessed via the "Categories" dropdown link. (Note that all on-demand content is subtitled, which is why there is no "Subtitled" category. Subtitles can be switched on in the media player.)



The "Categories" dropdown with "Audio Described" and "Signed" sections.

User research and feedback indicated, however, that although people want dedicated categories, they also want to be able to search for and browse content in the same way that any other users would and to select their preferred format from there. I have stayed in touch over the years with the gentleman who complained about the old iPlayer page, and he's said himself, "Don't send us into disability silos!"

This means that from the outset the designs need to signpost "Audio Description" and "Signed" content via search results, A to Z, category and other listing pages. Not making any assumptions or not stereotyping users with disabilities is important — for instance, a person with a severe vision impairment might not always use audio descriptions; news, sports, music programs and live events often aren't supported by audio description because commentators already provide enriched commentary.



British Gardens in Time

3. Biddulph Grange

Biddulph Grange takes the visitor on a journey around the world from China to Egypt.

BBC FOUR

First shown: 22 Apr 2014

Available for 23 days

60 mins

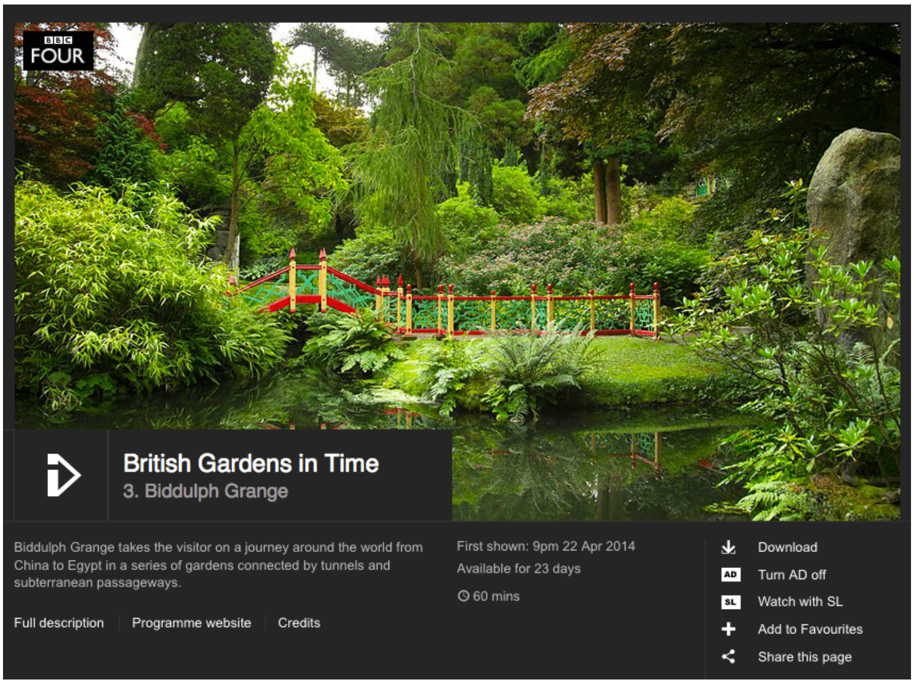
List pages such as search, shown here, indicate what formats programs are available in.

On-demand pages also list alternative formats, allowing users to choose what they want. Looking ahead, the option to choose your format could also be included in the Standard Media Player[9] — the BBC media player used for

---

[9] http://www.bbc.co.uk/blogs/internet/posts/Standard-Media-Player

on-demand and live streaming video across all BBC products, including iPlayer.



Biddulph Grange takes the visitor on a journey around the world from China to Egypt in a series of gardens connected by tunnels and subterranean passageways.

First shown: 9pm 22 Apr 2014

Available for 23 days

60 mins

Screenshot of the playback page showing HD and AD formats.

PUT USERS IN CONTROL

Never taking control away from the user is essential. A key aspect of this in iPlayer, which is responsive, is not suppressing pinch zoom. Time and again in user testing, we have observed users zooming content, even on responsive websites, where text might be intentionally larger.

Due to an iOS bug that was rectified in iOS 6, the ability to pinch zoom was suppressed on many websites due to poor resizing when the orientation is changed from portrait to landscape. Now that this has been fixed, there is no reason to continue suppressing zoom.

Another aspect of control is autoplay. While iPlayer currently has autoplay for live content, this can be a problem because the sound of the video can make it difficult for a screen reader user to hear their reader's output. However, we do know of screen reader users who request autoplay because it means they don't have to navigate to the player, find the play button and activate play. The answer is to look at ways to give users control over playback by opting in or out of autoplay, such as by using a popup and saving preferences with cookies.

DESIGN WITH FAMILIARITY IN MIND

There needs to be a balance between the new and the familiar. Users understand how to interact with pages and apps that use familiar design patterns. This is especially important in native apps for iOS and Android, where standard UI components come with accessibility built in.

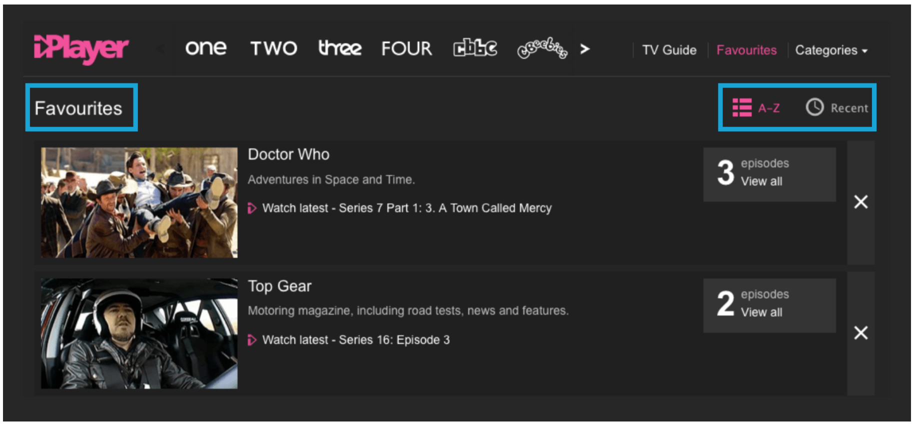
Equally important is the language used across the BBC's native iPlayer apps and responsive website. Where the platform allows, consistent labels for headings, links and buttons — not just visually, but also via alternatives for screen reader users — ensure that the experience is familiar and recognizably "BBC iPlayer," regardless of the platform.

Tied into this, the new designs reinforce a logical heading structure within the code, which in turn supports navigation for screen reader users. Key to this is ensuring that the pattern used for the heading structure is repeated across pages, so that users do not find main headings in different places depending on what page they are on. While structure is typically viewed as a responsibility of developers, it needs to be decided before designs are signed off in order to prevent poor structure getting coded in — more on that later.

PRIORITIZE FEATURES THAT ADD VALUE

Accessibility at the BBC is not just about meeting code, content and design requirements, but also about incorporating helpful features that add value for all users, including disabled users. A large proportion of feedback we get from our disabled users pertains to usability issues that could be experienced by anyone on some level but that seriously adversely affect disabled users. When we incorporate features to help users with specific disabilities, everyone gains access to a richer and easier experience.

One obstacle that comes up time and again is finding a favorite show. I've spoken with many screen reader users who say they save shortcuts to their favorite shows on their desktop but, due to changing URLs, often lose content. A simple way to address this that benefits all users is to ensure that there is a mechanism for saving favorites on the website. Adding in options to sort favorites and list them the way you want further improves this. It may sound unrelated to accessibility, but it was the single most requested feature received from disabled users. Simply accessing the favorites page to watch the latest episode of something, rather than having to search the website, makes all the difference.

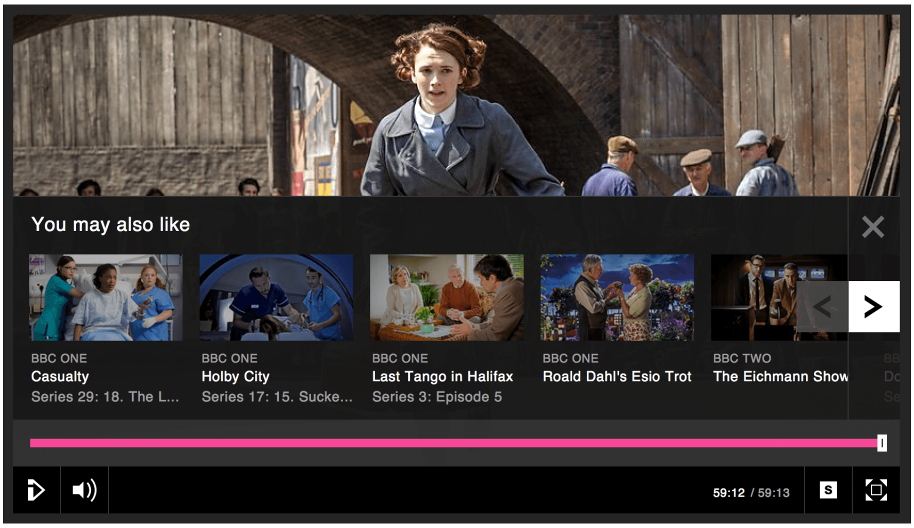


Doctor Who
Adventures in Space and Time.
Watch latest - Series 7 Part 1: 3. A Town Called Mercy
3 episodes View all

Top Gear
Motoring magazine, including road tests, news and features.
Watch latest - Series 16: Episode 3
2 episodes View all

The "Favourites" page, with options to sort by "A to Z" and "Recent".

Finding ways to allow people to get to the content they want more quickly has also influenced what is available within the media player itself. Once an episode has finished playing, exiting the media player and navigating back to the website to find the next episode is a massive overhead for some users. Adding a "More" button to the player itself — showing the next episode or programs



BBC ONE
Casualty
Series 29: 18. The L...

BBC ONE
Holby City
Series 17: 15. Sucke...

BBC ONE
Last Tango in Halifax
Series 3: Episode 5

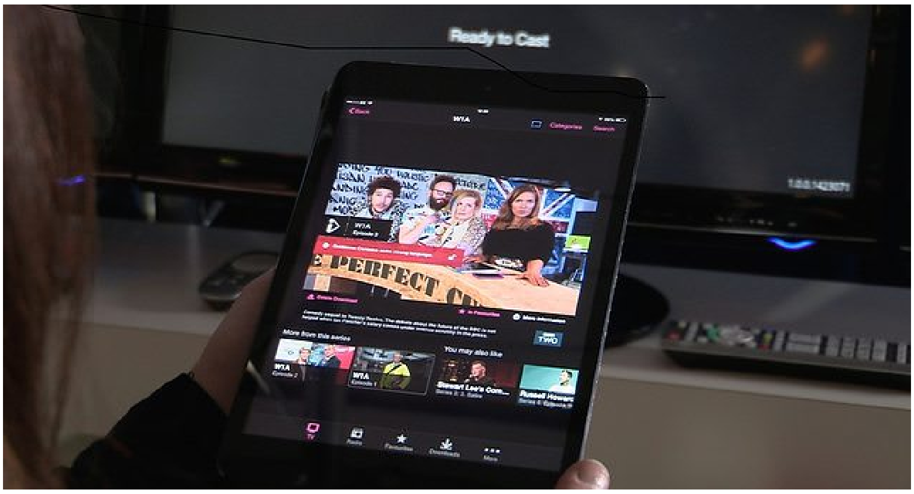
BBC ONE
Roald Dahl's Esio Trot

BBC TWO
The Eichmann Show

The "You may also like" plugin shows related content and next episodes within the Standard Media Player.

similar to the current one — cuts down on the amount of effort it takes users to find new content.

One key feature that has added value to BBC iPlayer's native iOS and Android apps, as well as the website (when viewed in Chrome), is support for Google Chromecast[10]. Being able to control what content you view on TV without having to use a remote or complex TV user interface is invaluable. Using one's device of choice, whether it be iOS or Android, is much easier for a disabled user than using a remote control and a potentially inaccessible TV interface.



BBC iPlayer and Chromecast.

Guidelines

The principles above exist to create a mindset that helps product owners and UX practitioners alike when shaping and designing inclusive products. In addition to the four

---

[10] http://www.bbc.co.uk/blogs/internet/posts/Accessibility-on-BBC-iPlayer-on-Chromecast

principles, a set of guidelines is used to design more accessible interfaces. The following are a subset taken from the “BBC Mobile Accessibility Standards and Guidelines[11]”:

1. **Color contrast**
   Ensure that text and backgrounds exceed the WCAG Double A 4.5:1 contrast minimum.
2. **Color and meaning**
   Information conveyed with color must also be identifiable from context or markup.
3. **Content order**
   Content order must be logical.
4. **Structure**
   When supported by the platform, pages must provide a logical and hierarchical heading structure.
5. **Containers and landmarks**
   When supported by the platform, page containers or landmarks should be used to describe page structure.
6. **Duplicate links**
   Controls, objects and grouped interface elements must be represented as a single component.
7. **Touch target size**
   Targets must be large enough to touch accurately (44 pixels).

---

[11] http://www.bbc.co.uk/guidelines/futuremedia/accessibility/mobile

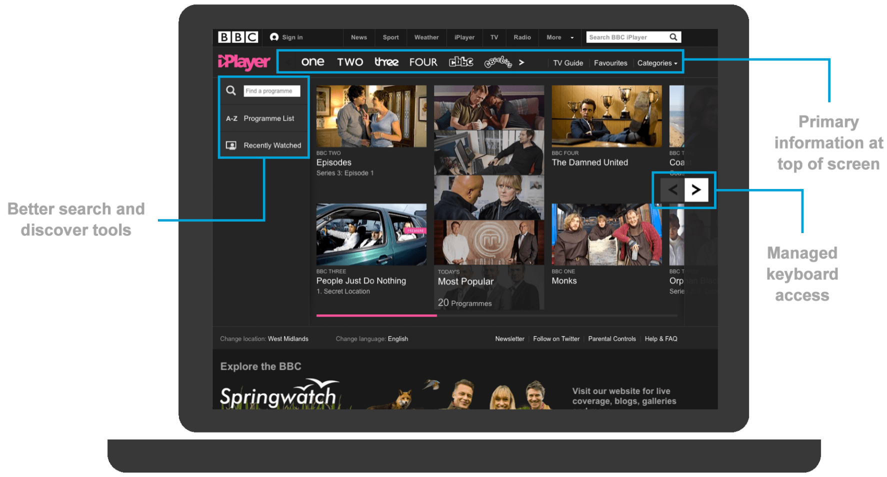
8. **Spacing**
   An inactive space must surround all active elements (unless they are large blocks exceeding 44 pixels).

9. **Zoom**
   Where zoom is supported by the platform, it must not be suppressed.

10. **Actionable elements**
    Links and other actionable elements must be clearly distinguishable.

The New iPlayer

Keeping in mind this backdrop of principles and guidelines, along with the renewed focus on adding value and features that enhance the experience for disabled users, here are a few of the changes introduced in the BBC's new iPlayer:



The BBC's new iPlayer home page has better content order, search tools, structure and keyboard access.

At launch, the iPlayer's navigation housed the BBC's channels, a "TV Guide," "Favourites" and "Categories." These all sit at the start of the page, high up in the content order. While they are visually easy to see, they are also easily discoverable by screen reader users via a hidden heading and labeled navigation landmark:

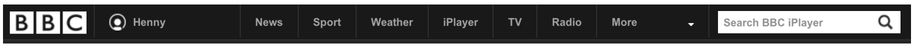
```
<div role="navigation">
<h2>iPlayer navigation</h2>
```

Where previously the "Categories" were unusable for the screen reader user I spoke with, they are now prominent in the page and fully keyboard navigable. Since launch, the addition of more channels has meant that the channel links have been rehoused in their own dropdown menu.

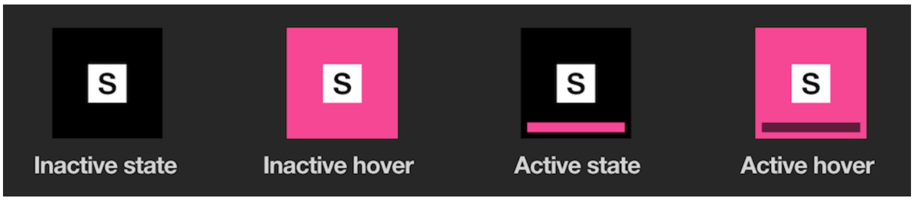
Search tools have also been added, enabling users to carry out predictive search, browse A to Z or view their most recently watched program. This is all keyboard accessible, it makes use of headings, and it has landmarks where appropriate.

The home page carousel is also fully keyboard accessible. Each program in the stream is presented as one link, with the reading order of text starting with the primary information first: channel attribution, program name, episode information, abstract and program duration.

Work has also been carried out to improve visible focus and bring both the iPlayer website and the Standard Media Player in line with the BBC header and footer. The pink underline used for the hover and focus states in the main BBC navigation is now used within the Standard Media Player to indicate when a button is selected — for example, when the subtitles are switched on. This replaces the use of color only to indicate a selected state, which was indistinguishable from the hover and focus states.



The hover and focus pink underline used in the BBC header for iPlayer.



Active and inactive hover and focus states on the subtitle button in the Standard Media Player.

You can read more about what steps were taken to make iPlayer web-accessible[12] and to make the Standard Media Player accessible[13], including creation of an accessible media player in Flash[14], on the BBC's Internet Blog.

Annotated UX

All of the thinking around inclusive design that comes from product owners, UX practitioners and designers needs to be captured and communicated to developers and engineers. At the BBC, we are moving to a model

---

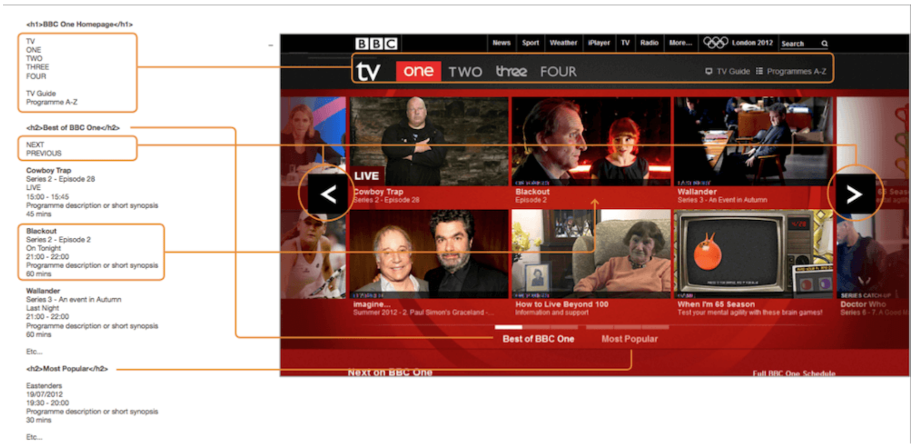
[12] http://www.bbc.co.uk/blogs/internet/posts/Making-the-new-iPlayer-accessible-for-all-users

[13] http://www.bbc.co.uk/blogs/internet/posts/Standard-Media-Player-accessibility

[14] http://www.bbc.co.uk/blogs/internet/posts/Creating-an-accessible-media-player-in-Flash

where designs need to be annotated for accessibility. This includes:

- headings,
- containers,
- content order,
- color contrast,
- alternatives to color and meaning,
- visible focus,
- keyboard and input interactions.



An example of an annotated UX showing headings and labels. (View large version[15])

The design above, showing an early version of the BBC One home page in iPlayer, outlines where the `<h1>` to

---

[15] http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2015/02/114-iPlayerCarousel-opt.png

`<h6>` headings should be. The UX practitioner doesn't need an in-depth knowledge of code, but rather an understanding of the hierarchy of data within a page. As such, an equally acceptable approach would be to indicate the "main heading," "secondary heading," "third-level heading" and so on. Developers can then take this and translate it into semantic markup.

Equally, indicating the logical order of content helps developers to code content in the right sequence (i.e. source order) — something that is essential to a screen reader or sighted keyboard user's comprehension of the page.

Annotating the UX in this way is key to identifying designs that don't allow for a logical page structure, content order or behavior. It is the first step to generating a style guide that documents focus states, colors and so on. Further down the line, these requirements can also be used to generate user acceptance criteria and automated quality assurance tests.

Even if you're working in an agile way, where designs are iterative and not delivered in a complete form, annotation still works. As long as the basic framework of the page is well defined, the visual design can evolve from that.

Summary

It's very easy to get bogged down by accessible output and to forget that, ultimately, accessibility is about people. As such, keep the following in mind, whether you are working in product, UX, development or quality assurance:

- Design with choice in mind.
- Always give users control over the page.
- Prioritize features that add value for disabled users.
- Design with familiarity in mind.
- Integrate accessibility into annotated UX and style guides.
- Make no assumptions. Test ideas and concepts.

Fostering these key principles across the entire team will go a long way to ensuring that products are inclusive and usable for disabled people. Listening to users and actively including their feedback, along with adhering to organizational standards and guidelines, are essential. ❧

Mobile And Accessibility: Why You Should Care And What You Can Do About It

BY TJ VANTOLL ❧

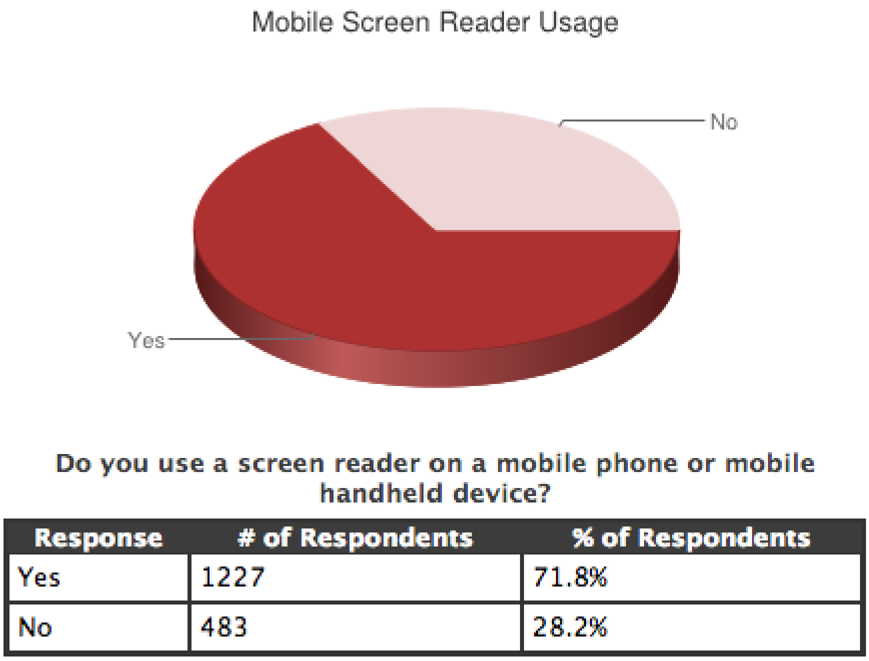
Mobile has revolutionized the way we use the web. This is especially true of disabled users, for whom mobile devices open the door to a whole new spectrum of interactions.

And they are taking advantage of it. A July 2013 survey[16] (PDF) of adults with disabilities done by the Wireless Rehabilitation Engineering Research Center[17] found that 91% of people with disabilities use a “wireless device such as a cell phone or tablet.” Among these users, screen reader usage is common, even on mobile devices.

A study of 1782 screen reader users[18] done by Web Accessibility in Mind[19] (WebAIM) in 2012 showed that 71.8% used screen readers on their mobile devices. And we're not talking about only a handful of people. The 2010 U.S. Census found that nearly one in five people have a disability[20]!

---

16. http://www.wirelessrerc.org/sites/default/files/publications/SUNspot_2013-03_Wireless_Devices_and_Adults_with_Disabilities_2013-07-12%5B1%5D.pdf
17. http://www.wirelessrerc.org/
18. http://webaim.org/projects/screenreadersurvey4/#mobile
19. http://webaim.org/
20. http://www.census.gov/newsroom/releases/archives/miscellaneous/cb12-134.html



Do you use a screen reader on a mobile phone or mobile handheld device?

Response	# of Respondents	% of Respondents
Yes	1227	71.8%
No	483	28.2%

The results of WebAIM's study of screen reader users (Source: WebAIM[21])

Despite this, many basic best practices for accessibility are forgotten on mobile websites. Developers implement complex solutions such as responsive design and responsive images, yet forget about basic techniques such as image replacement. Therefore, disabled users — who have a difficult enough time on the desktop — are frequently presented with interfaces that are at best frustrating, and at worst, impossible to use.

While accessibility can be a complex topic, following a few best practices goes a long way towards building accessible sites and applications. In this article we'll discuss a few practical measures that address the most common issues disabled users encounter. Specifically, we'll look at the importance of the following:

---

[21] http://webaim.org/projects/screenreadersurvey4/#mobile

- making sure everything works with a keyboard,
- marking up forms semantically,
- providing plenty of contrast,
- ensuring that screen readers know what your controls do,
- testing your website on an actual screen reader.

We'll see that following these best practices leads to a better experience for everyone, not just disabled users. Let's get started by looking at a piece of hardware rarely considered in the mobile space: the keyboard.

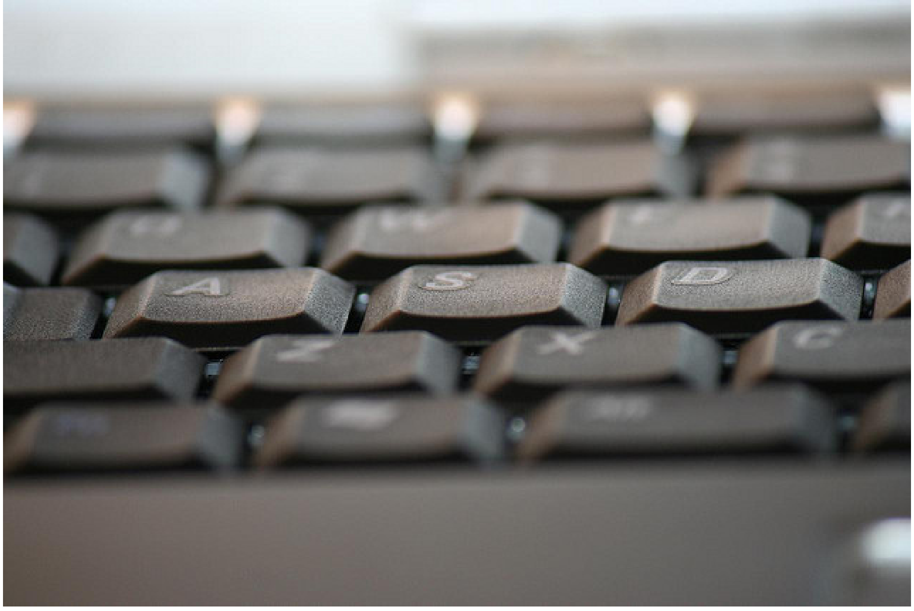
1. *Make Sure Everything Works With The Keyboard*

While we usually associate keyboards with traditional desktops and laptops, the situation is no longer that simple. Microsoft's Surface tablet has a keyboard built into the cover, Bluetooth keyboards are made to work seamlessly on iOS and Android, and some keyboards can even be rolled up[22] for traveling.

Keyboard navigation on mobile websites has become increasingly important also because more websites are built responsively. Because these websites serve the same markup to all devices, the keyboard must function correctly, even if the vast majority of visitors use a touchscreen.

---

[22] https://www.google.com/search?site=&tbm=isch&source=hp&biw=1600&bih=1008&q=roll-up+keyboard&oq=roll-up+keyboard



Keyboard support is important, even on mobile. (Image: Shane Pope[23]*)*

Try putting your mouse away and visiting your website. Can you perform *all* tasks using only your keyboard? If not, then neither can keyboard users. Two main problems break keyboard navigation on the web. Let's discuss each.

NOT USING THE CORRECT ELEMENT FOR THE TASK

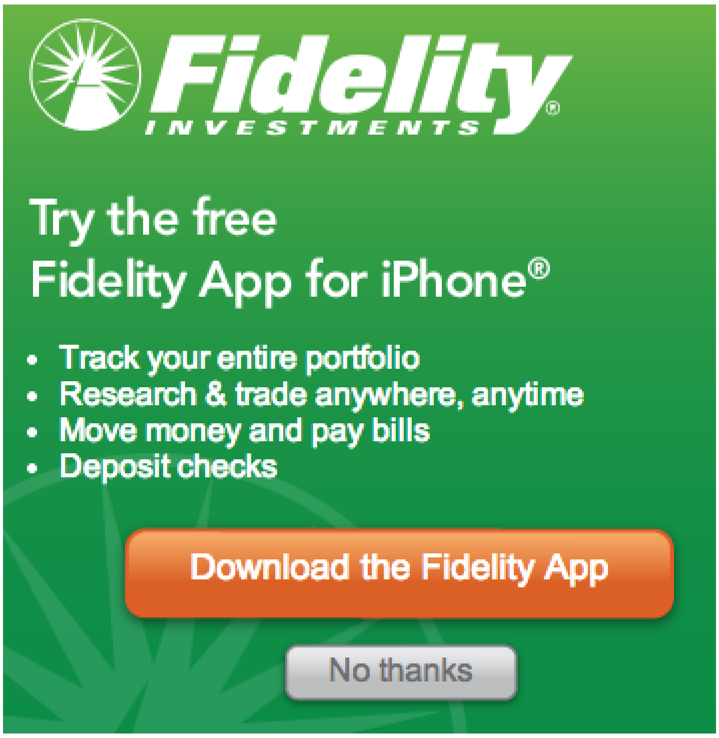
Web browsers are really good at making keyboard navigation work automatically, as long as you use semantically appropriate elements. In particular, the following elements are focusable by default: `<button>`, `<a>` (with an `href` attribute), `<input>`, `<select>` and `<textarea>`. For these elements, keyboard navigation just works without any extra effort. However, developers frequently fail to

---

[23] http://www.flickr.com/photos/24285431@N04/2375499336

use these elements appropriately, specifically `<button>` and `<a>`.

Buttons should be used for controls that perform actions, whereas links should be used for controls that navigate users to other documents. However, developers often incorrectly use generic elements, such as `<span>` and `<div>`, for controls that perform these actions. Consider the iPhone landing page of Fidelity[24], a banking institution.



Try the free
Fidelity App for iPhone®

- Track your entire portfolio
- Research & trade anywhere, anytime
- Move money and pay bills
- Deposit checks

Are those buttons on Fidelity's iPhone landing page or not?

The two controls on this page sure look like buttons, but they're actually generated by the following markup.

---

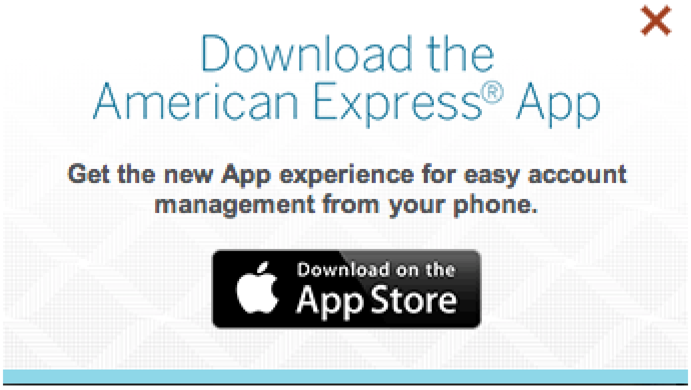
[24] http://www.fidelity.com/interstitial/index.shtml

```
<div id="download_button">
        <div id="download_text"
        onclick="goToStore();">Download the Fidelity
        App</div>
</div>

<div id="no_thanks_button" onclick="goHome();">
        <div id="no_thanks_text"
        onclick="goHome();">No thanks</div>
</div>
```

Because of this, keyboards will not navigate to these controls. They would, however, if the controls were marked up as actual `<button>` elements.

```
<div id="download_button">
        <button id="download_text"
        onclick="goToStore();">Download the Fidelity
        App</button>
</div>

<div id="no_thanks_button" onclick="goHome();">
        <button id="no_thanks_text"
        onclick="goHome();">No thanks</button>
</div>
```

While this most commonly happens with buttons, links are often misused as well. Consider the popup that people see when visiting American Express' home page[25] on an iPad:



Download the
American Express® App

Get the new App experience for easy account management from your phone.

American Express shows this popup to iPad users on its home page.

Despite the "Click here to download the American Express app" being a simple link, American Express uses the following HTML:

```
<div class="asl-link" role="link">
        <div class="asl-app-store" title="...">
                Click here to download the American
                Express® App
        </div>
</div>
```

The linking is done in JavaScript, with a click handler that changes `window.location`. Because a `<div>` is used, keyboards cannot access the control, but they would be able to if an `<a>` element were used.

---

[25] https://www.americanexpress.com/

```
<div class="asl-link">
        <a href="..." class="asl-app-store">
                Click here to download the American
                Express® App
        </a>
</div>
```

(On a related note, the "close" button in American Express' popup is also a `<div>`, so keyboard users can neither perform that action in the popup nor close it.)

American Express could also have used the Apple approved[26] method of informing users that an app is available with an `apple-itunes-app` `<meta>` tag.

```
<meta name="apple-itunes-app"
content="app-id=myAppStoreID,
affiliate-data=myAffiliateData, app-argument=myURL">
```

In general, browsers provide keyboard access automatically for semantic elements. But what if you're using elements more complex than a simple `<button>` or `<a>`?

WRITING YOUR OWN COMPLEX WIDGETS

As web developers, we often use widgets for complex interactions in our interfaces. We do this sometimes to work around deficiencies in a platform (for example, to build a more customizable `<select>` element) and some-

---

[26] https://developer.apple.com/library/ios/documentation/AppleApplications/Reference/SafariWebContent/PromotingAppswithAppBanners/PromotingAppswithAppBanners.html

times to build a powerful UI element (for example, a grid, tab or chart).

Let's look at replacing a `<select>` element. On the surface, this seems like an easy widget to write:

1. Create a `<div>`.
2. Create a `<ul>`, with a `<li>` element for each replacement `<option>`.
3. On a click of the `<div>`, open the menu.
4. On a click of a `<li>` element, transfer the value back into the `<div>`.

Easy.

However, consider all of the keyboard controls that a native `<select>` element has:

- The space bar opens the menu of options.
- The up and down arrows open the menu and cycle between options.
- The page-up and page-down keys cycle through whole pages of options in long lists.
- The "Escape" key closes the menu.
- Typing in values while the element has focus will trigger autocompletion options.

Suddenly, replicating the native `<select>` element has gotten significantly harder. Not to mention that we have to ensure that screen readers can access the options and read them at the appropriate time.

To make these complex interactions possible, a special set of HTML attributes is available to provide the necessary context to screen readers. These are known as Accessible Rich Internet Applications[27], or ARIA attributes.

The main ARIA attribute, `role`, tells browsers and assistive devices the general type of an object — for example, dialog, slider or alert. From there, a number of `aria-*` attributes are used to provide more detailed information about an element's state. Below is a boilerplate for developing an accessible `<select>` replacement.

```
<span tabindex="0" id="button" role="combobox"
        aria-expanded="false"
        aria-autocomplete="list" aria-owns="menu"
        aria-haspopup="true"
        aria-activedescendant="option-1"
        aria-labelledby="option-1"
        aria-disabled="false">
                One
</span>

<ul aria-hidden="true" aria-labelledby="button"
id="menu"
        role="listbox" tabindex="0"
        aria-activedescendant="option-1"
        style="display: none;">
                <li id="option-1" role="option"
```

---

[27] https://developer.mozilla.org/en-US/docs/Accessibility/ARIA

```
                tabindex="-1">One</li>
                <li id="option-2" role="option"
                tabindex="-1">Two</li>
                <li id="option-3" role="option"
                tabindex="-1">Three</li>
</ul>
```

Screen readers use these ARIA attributes to help keyboard users operate these controls. For example, because this example's `<span>` has a `role` of `"combobox"`, VoiceOver on OS X reads "You are currently on a combo box, type text or, to display a list of choices, press Control-Option-Space." VoiceOver knows what choices are available because the `<span>` has an `aria-owns` attribute set to `"menu"`, the `id` of the `<ul>` that contains the options.

But as you can see, there are a whole lot of ARIA attributes to account for; therefore, because of the difficultly of getting this right, most developers are better off using a JavaScript UI library for such controls rather than building them themselves. Many big libraries ensure that these controls are accessible by applying appropriate ARIA roles[28] and keyboard shortcuts automatically. For example, jQuery UI's upcoming `selectmenu`[29] and Kendo UI's `DropDownList`[30] both generate accessible `<select>` replacements.

This concludes our look at supporting keyboard navigation on the web. While this set of guidelines is by no

---

[28] https://developer.mozilla.org/en-US/docs/Accessibility/ARIA/ARIA_Techniques
[29] http://wiki.jqueryui.com/w/page/12138056/Selectmenu
[30] http://demos.kendoui.com/web/dropdownlist/index.html

means comprehensive, it should help you get the most important parts of your website working properly. For a more thorough list of keyboard accessibility, see the W3C's complete guidelines[31].

Next, we'll look at another often abused best practice: building semantic forms.

2. *Mark Up Forms Semantically*

With the proliferation of single-page applications — especially mobile ones — a diminishing number of websites use native HTML form submissions. Instead, data is submitted to the server by JavaScript-initiated `XMLHttpRequest`s. While nothing is inherently wrong with this, using the same semantic markup that you would use if the data were to be sent with a native HTML form submission is still important.

Most screen readers have entire modes dedicated to form interaction that require forms to be marked up correctly. Consider this log-in form:

```
Username: <input type="text" id="username"
name="username">
Password: <input type="password" id="password"
name="password">
<button>Submit</button>
<script>
        document.querySelector( "button" )
                .addEventListener( "click",
```

---

[31] http://www.w3.org/WAI/WCAG20/quickref/#keyboard-operation

```
                function() {
                        // Send AJAX request to log
                        // user in
                });
</script>
```

While this form will log users in, it has a few problems that would make it painful for power users and impossible for impaired users to complete. Let's look at the issues in turn.

ASSOCIATING LABELS WITH INPUTS

Most screen readers require that form elements — `<input>`, `<select>` and `<textarea>` — be associated with `<label>` elements that describe them. Each `<label>` element should have a `for` attribute that corresponds to the appropriate form element's `id` attribute, as shown here:

```
<label for="field">Field:</label>
<input id="field">
```

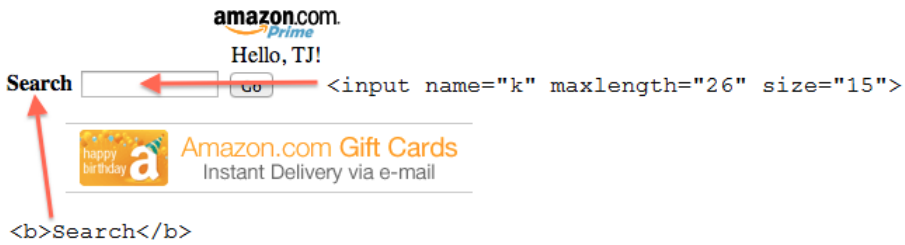
As is, our log-in form does not have this association, which would trip up assistive devices. Below is the user name `<input>` that we're currently using. Note that there is no `<label>`.

```
Username: <input type="text" id="username"
name="username">
```

When this user name `<input>` gets focus, the screen reader NVDA[32] simply reads "Edit text, blank."

This is common in the wild, unfortunately. Amazon, for instance, has a trimmed-down website, amazon.com/access[33], that is “optimized for screen readers and mobile devices.”

The website is appropriately simple, with a single search box and a short list of links. Ironically, though, despite directing screen reader users to this page, Amazon has not given its search `<input>` a `<label>`; thus, many screen reader users will have no idea that the `<input>` can be used to search.



```
<input name="k" maxlength="26" size="15">
<b>Search</b>
```

Amazon’s accessible website does not associate the “Search” label with the text box.

(*Note:* As with browsers, screen readers vary in their support of markup patterns. In the example above, VoiceOver does read the “Search” text, but NVDA does not. We’ll discuss compatibility testing later in this article.)

We can easily solve this problem in our sample form with a few `<label>` elements:

---

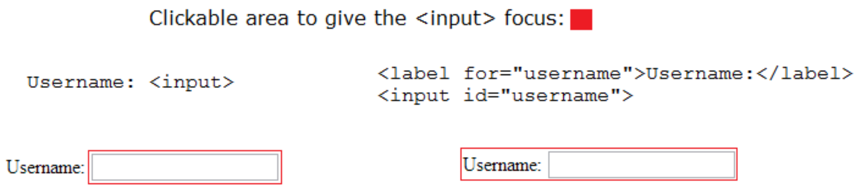
[32] http://www.nvaccess.org/

[33] http://amazon.com/access

```
<label for="username">Username:</label>
<input type="text" id="username" name="username">

<label for="password">Password:</label>
<input type="password" id="password" name="password">

<button>Submit</button>
<script>
        document.querySelector( "button" )
                .addEventListener( "click",
                function() {
                        // Send AJAX request to log
                        // user in
                });
</script>
```

Now all screen readers will associate each form element with a label that describes it. This practice is beneficial to more than users of assistive devices. All browsers are smart enough to transfer clicks on `<label>` elements to their associated `<input>`, `<select>` or `<textarea>` elements.

Ever had trouble clicking a 10 × 10-pixel checkbox to accept a service's terms of use? On websites with semantic markup, you can click a checkbox's far larger label instead. On mobile devices, bigger targets help fat fingers hit them:


Clickable area to give the <input> focus: 🟥
Username: <input>
<label for="username">Username:</label>
<input id="username">
Username:
Username:

Clicking a label gives focus to the corresponding control.

This solves one problem with our log-in form. But there's still one more issue to discuss.

HANDLING THE "ENTER" KEY

Have you ever noticed that some log-in forms on the web can be submitted by pressing the "Enter" key in a textbox and some cannot? What makes the "Enter" key work for a submission?

Submitting with the "Enter" key is known as an implicit submission[34], and it is supported by all browsers — even mobile ones. There are only two requirements to making implicit submission work.

1. All `<input>` elements need to be in a `<form>`.
2. If the form has more than one element — `<input>`, `<select>` or `<textarea>` — then the `<form>` must have a "Submit" button.

---

[34] http://www.whatwg.org/specs/web-apps/current-work/multipage/association-of-controls-and-forms.html#implicit-submission

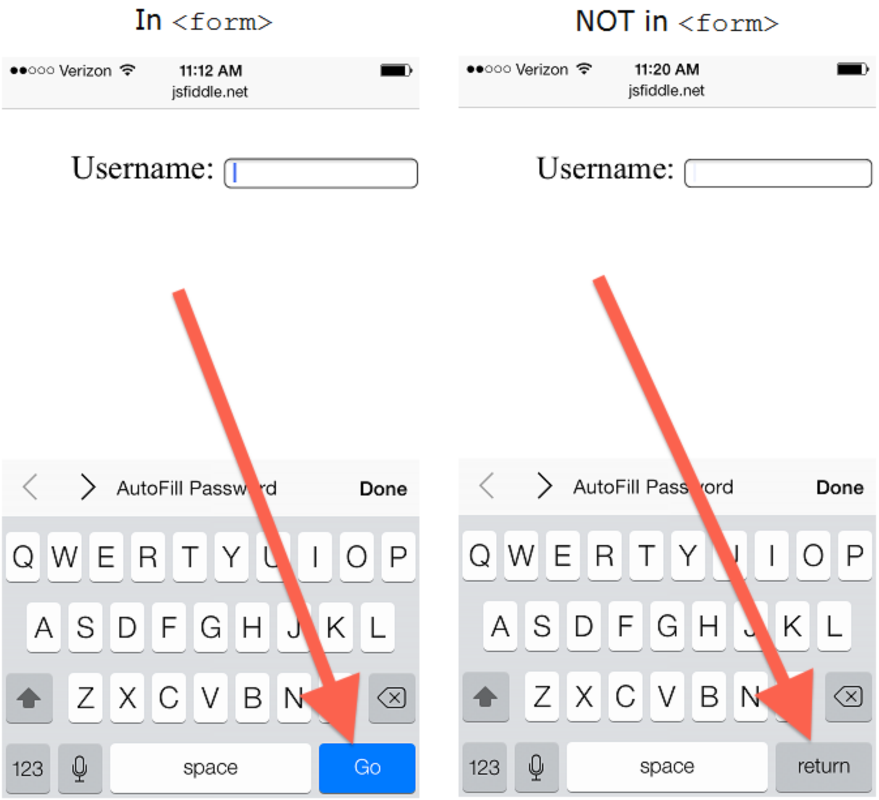
Our sample form already has a “Submit” button (the default `type` of `<button>` is `submit`). Therefore, we only need to add a `<form>`.

```
<form method="POST">
        <label for="username">Username:</label>
        <input type="text" id="username"
        name="username">

        <label for="password">Password:</label>
        <input type="password" id="password"
        name="password">

        <button>Submit</button>
</form>
<script>
        document.querySelector( "button" )
                .addEventListener( "click", function(
                event ) {
                        // Prevent the default form
                        // submission
                        event.preventDefault();

                // Send AJAX request
        });
</script>
```

A couple of things to note:

- When implicit submission occurs, the browser performs a click on the `<form>`’s “Submit” button. Therefore, listening for `click` events on the “Submit” button to perform submit actions is still safe. You could, alternatively, listen for a `submit` event on the `<form>`.

- Even though JavaScript will intercept the form’s submission, `method="POST"` is still explicitly declared. In case JavaScript fails (because of unsupported browsers, network issues, etc.), we don’t want the browser to submit a `GET` request that would place the user-supplied user name and password in the query string.


In `<form>`

NOT in `<form>`

Implicit form submission works only with true form elements.

Enabling implicit submission saves mobile users some clicking. The image above shows two `<input>` elements, one in a `<form>` and one not. Note that the `<form>`-based example can be submitted while the `<input>` has focus — no need for additional taps or hunting for a "Submit" button.

This concludes our look at building accessible forms. As with keyboard access, these guidelines are far from comprehensive, but they address some of the most common problems with forms today. For a more complete list of best practices, check out the W3C's guidelines for collecting input from users[35].

Next, we'll look at a problem that most smartphone owners have experienced at some point: low contrast.

3. *Provide Plenty Of Contrast*

If you've ever used your phone outdoors, especially in harsh sunlight, then you've struggled to read the screen at some point. This is not unlike what some users with vision disabilities experience all of the time.

When designing any website, remember that most users will not be visiting it indoors on a high-end monitor like yours. This is especially critical for mobile websites because of the wide variety of contexts in which people use their phone.

How do you make your website adapt to these contexts? Unlike more subjective aspects, contrast ratio can be calculated, and the W3C puts numeric guidelines for these ratios in its "Web Content Accessibility Guidelines[36]" (WCAG), a series of recommendations for making

---

[35] http://www.w3.org/WAI/WCAG20/quickref/#minimize-error

[36] http://www.w3.org/TR/WCAG/

web content more accessible to individuals with disabilities.

The WCAG defines three levels of conformance: Level A, Level AA and Level AAA. According to the specification[37], for a website to conform, Level A requirements *must* be met, Level AA requirements *should* be met and Level AAA requirements *may* be met.

The Level A contrast ratio is set at 3:1; Level AA is set at 4.5:1; and Level AAA is set at 7:1. As a point of reference, the specification recommends[38] a contrast ratio of 4.5:1 for users with 20/40 vision, which is common among elderly people.

This means that we should make our contrast ratios at least 3:1 and, ideally, 4.5:1 or greater. But, how exactly do we calculate this?

CALCULATING CONTRAST RATIO

Luckily, Lea Verou[39] has created a tool to calculate contrast ratio[40] easily.

It's simple to use. Input a background color and a text color, and the tool will output the contrast ratio:

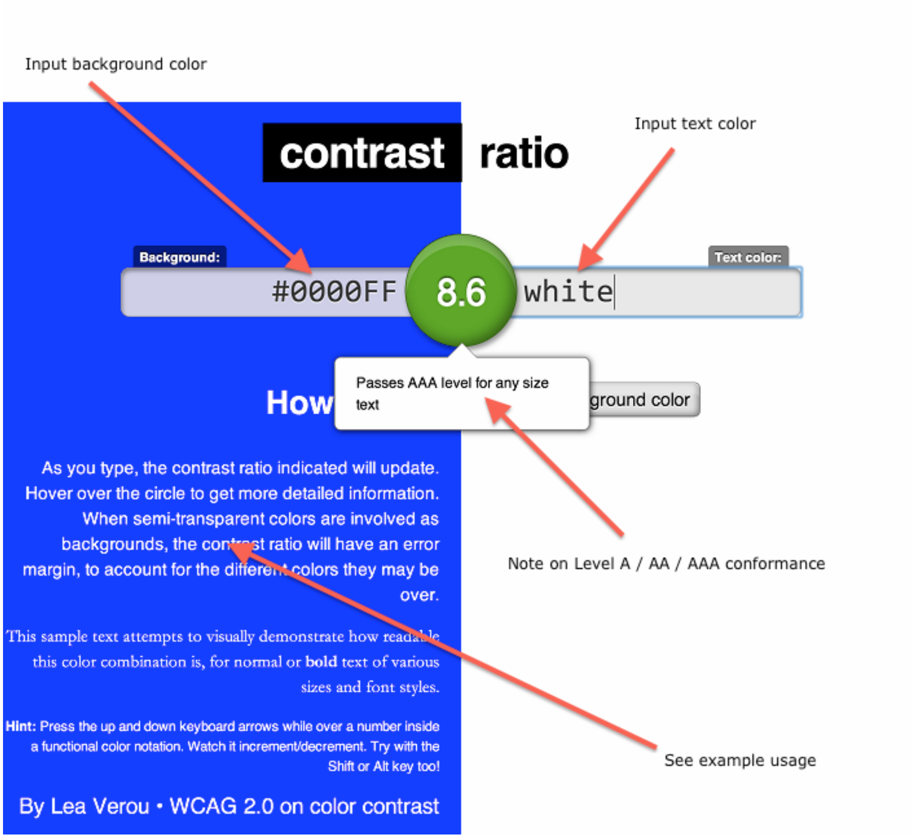
---

[37] http://www.w3.org/TR/WAI-WEBCONTENT/#wc-priority-1

[38] http://www.w3.org/TR/UNDERSTANDING-WCAG20/visual-audio-contrast-contrast.html

[39] http://lea.verou.me/

[40] http://leaverou.github.io/contrast-ratio/



Passes AAA level for any size text

As you type, the contrast ratio indicated will update. Hover over the circle to get more detailed information. When semi-transparent colors are involved as backgrounds, the contrast ratio will have an error margin, to account for the different colors they may be over.

This sample text attempts to visually demonstrate how readable this color combination is, for normal or **bold** text of various sizes and font styles.

Hint: Press the up and down keyboard arrows while over a number inside a functional color notation. Watch it increment/decrement. Try with the Shift or Alt key too!

By Lea Verou • WCAG 2.0 on color contrast

Lea Verou's tool determines the contrast ratio between text and its background.

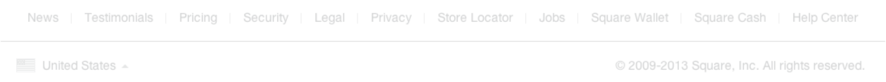
A couple of things to note:

- The tool supports any CSS color that the browser supports. So, keywords (e.g. white), HEX, RGB, RGBa, HSL and HSLa are all accepted.
- The tool generates unique URLs as you input valid values. For developers, this feature is perfect for telling designers that their gray on light-gray design is a bad idea[41].

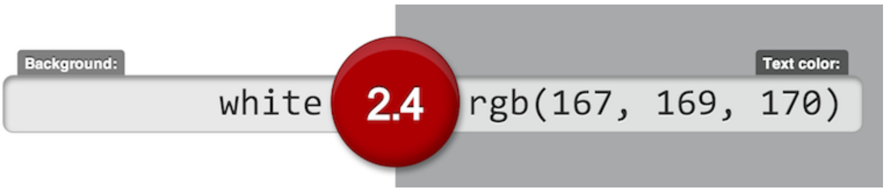
---

[41] http://leaverou.github.io/contrast-ratio/#gray-on-lightgray

While high contrast seems like common sense, plenty of poor contrast can be found in the wild, even on major websites. Square's[42] footer fails Level A:

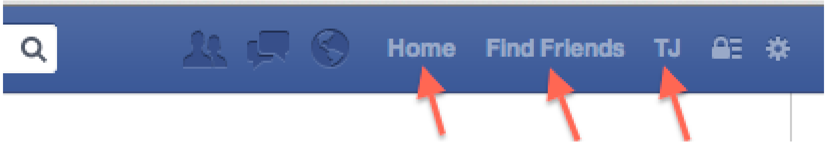


Square's footer has very low contrast.

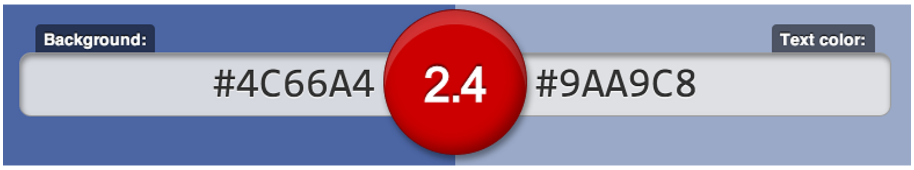


Square's footer fails Level A.

Even Facebook[43] is guilty. The links in its header on the desktop fail to meet Level A as well:



The text in Facebook's header has very low contrast.



Facebook's header fails Level A.

---

[42] https://squareup.com/

[43] http://facebook.com

To summarize, giving all of your text a contrast ratio of at least 3:1 (and, ideally, 4.5:1 or greater) will make it accessible to people with vision impairments, as well as make it more readable for everyone. Lea Verou's tool is perfect for calculating contrast ratios and sharing the results with others.

To find out more about conformance levels, the WCAG has a formatted checklist of criteria[44] for each level.

Next, we'll return to discussing screen readers — specifically, making sure they can understand our websites.

4. *Ensure That Screen Readers Know What Your Controls Do*

Earlier, we looked at a WebAIM study that shows that 71.8% of screen reader users also use a reader on their mobile device. In this same study, users were asked for the most common problems they experience on the web[45]. Third and fourth on this list are the following:

3. links or buttons that do not make sense,
4. images with missing or improper descriptions (`alt` text).

(First and second on the list are Flash and CAPTCHAs. Please don't use either.)

The following snippet shows both of these problems.

---

[44] http://www.w3.org/TR/2006/WD-WCAG20-20060427/appendixB.html
[45] http://webaim.org/projects/screenreadersurvey4/#problems

```
<style>
        button {
                background: url('search.png')
                no-repeat;
        }
</style>
<button></button>

<img src="ABC123.jpg">
```

The `<button>` would not make sense to a user on a screen reader; nothing would be read. For the `<img>`, screen readers would literally read `ABC123.jpg`, which is not very helpful.

The fixes for these problems are well documented. For the `<button>`, we can add text to our control and use one of *many* image-replacement techniques[46] to make the text invisible to sighted users. The snippet below shows one of these techniques: applying a large negative `text-indent` rule.

```
<style>
        button {
                background: url('search.png')
                no-repeat;
                text-indent: -9999px;
        }
</style>
<button>Search</button>
```

---

[46] http://css-tricks.com/css-image-replacement/

The fix for the `<img>` is as simple as adding an `alt` attribute that describes the image.

```
<img src="ABC123.jpg" alt="A view of the trees
outside my window in Lansing, Michigan">
```

Despite these problems being well known and easy to fix, they continue to be abundant across the web. To prove this, let's look at some actual websites. I live in the great US state of Michigan, known for its association with the "Big Three" automakers: Ford, GM and Chrysler. Surely, these large companies have produced accessible mobile websites that don't violate these practices… right?

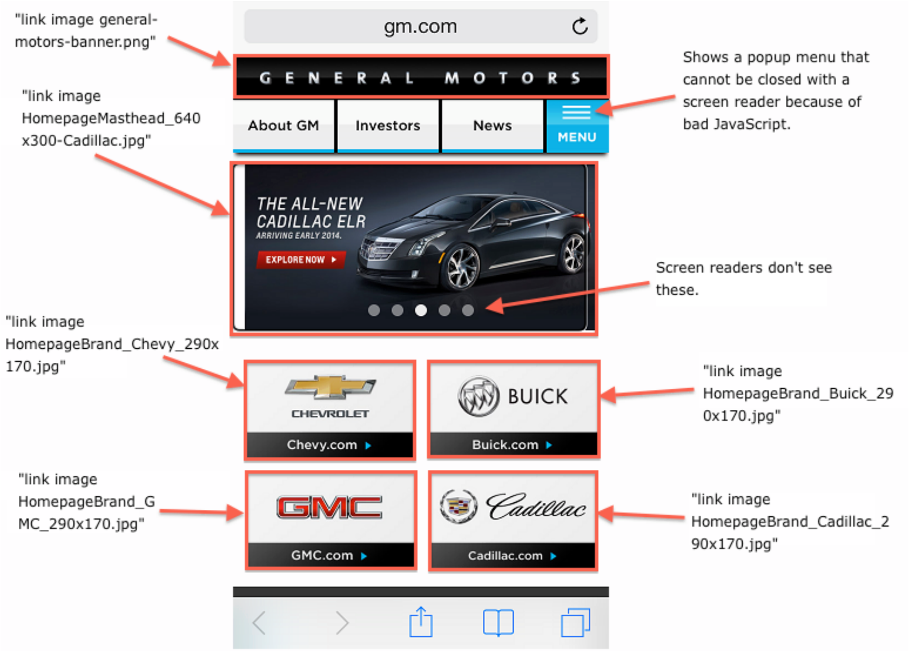
CASE STUDY: THE BIG THREE AUTOMAKERS

Let's start with GM. Its mobile website[47] is shown on the next page. The appended text in quotation marks shows what VoiceOver on OS X actually reads when the corresponding element is selected.

As you can see, GM's website relies heavily on large images that serve as links to additional content. Unfortunately, GM provides no text for these links, so screen readers are limited to the information they can find in the images. GM *does* provide `alt` attributes for these images; but, strangely, they are set to the images' file names.

---

[47] http://m.gm.com



"link image general-motors-banner.png"

"link image HomepageMasthead_640x300-Cadillac.jpg"

Shows a popup menu that cannot be closed with a screen reader because of bad JavaScript.

Screen readers don't see these.

"link image HomepageBrand_Chevy_290x170.jpg"

"link image HomepageBrand_Buick_290x170.jpg"

"link image HomepageBrand_GMC_290x170.jpg"

"link image HomepageBrand_Cadillac_290x170.jpg"

GM's mobile website has accessibility issues. Items in quotation marks are what VoiceOver on OS X actually reads. (View large version[48]*)*

For example, here is the source for the Cadillac image:

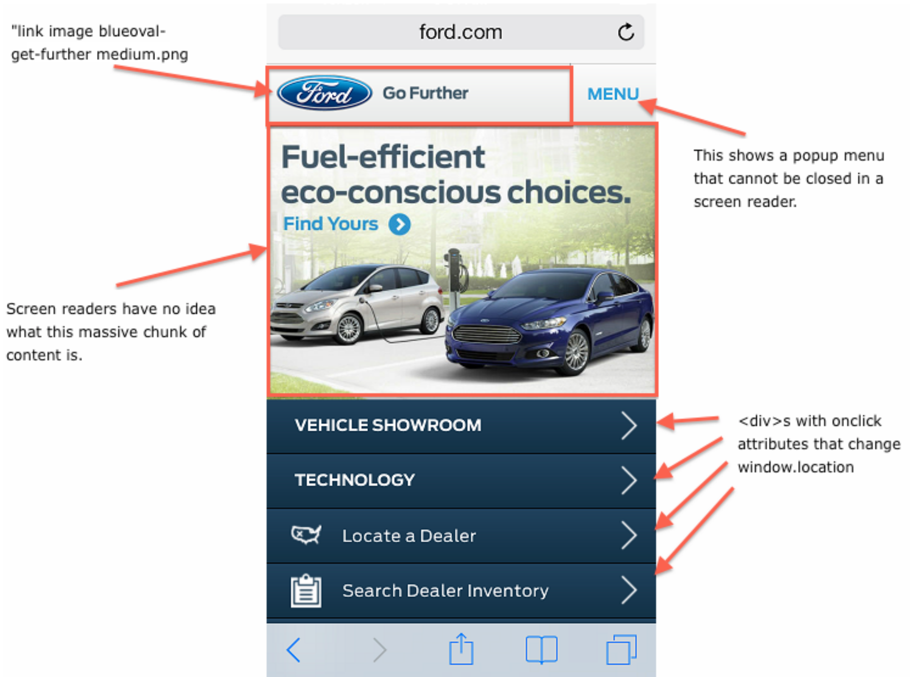
```
<img title="HomepageBrand_Cadillac_290x170.jpg"
height="85"
alt="HomepageBrand_Cadillac_290x170.jpg"
class="side-gutters"
src="/content/dam/gm/Global/master/mobilesite/en/home/
Homepage/HomepageBrand_Cadillac_290x170.jpg">
```

Thus, when this image is selected, VoiceOver literally reads "Homepagebrand underscore Cadillac underscore two nine zero x one seven zero dot jpeg."

---

[48] http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2013/10/gm.png

GM fails our tests, then. Next, let's try Ford, whose mobile website[49] is shown on the next page.



"link image blueoval-get-further medium.png

This shows a popup menu that cannot be closed in a screen reader.

Screen readers have no idea what this massive chunk of content is.

<div>s with onclick attributes that change window.location

Ford's mobile website also has accessibility issues. Items in quotation marks are what VoiceOver on OS X reads. (View large version[50])

Ford has some of the same problems as GM. Its banner is a link that has no text and an image with no `alt` attribute; so, VoiceOver reads "index.html image."

Below this, Ford has a fancy 3-D cube effect with images of its vehicles. Unfortunately, this is implemented with a number of `<div>`s, with `background-image`s and JavaScript that takes the user away on click. Therefore, screen readers have absolutely no idea what's going on in this large portion of the screen. VoiceOver on iOS just

---

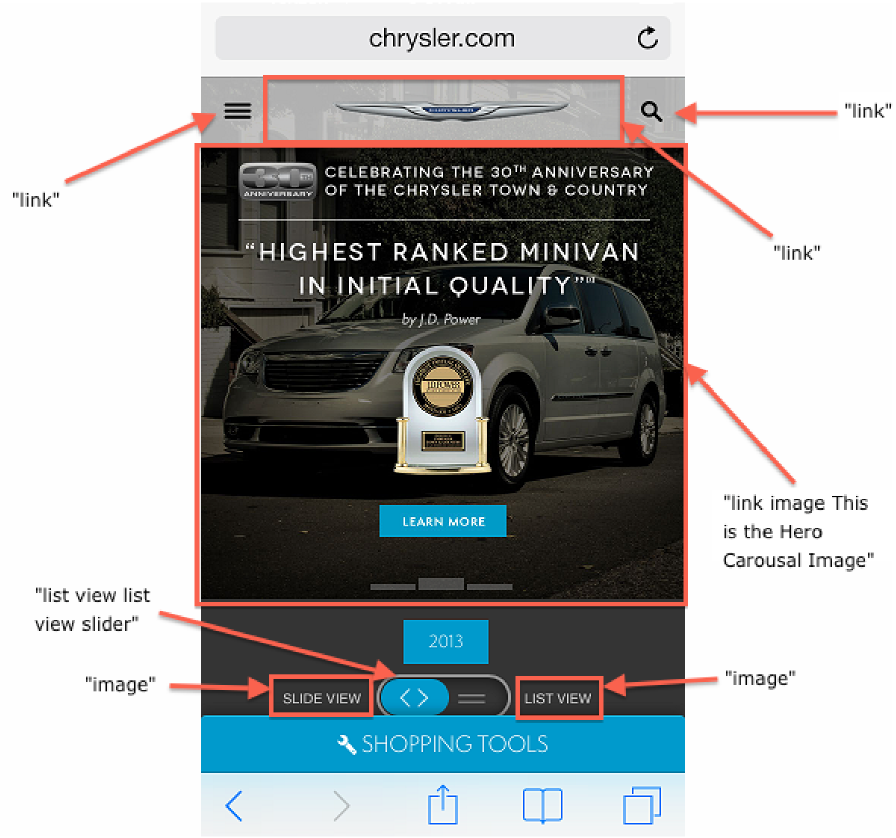
[49] http://m.ford.com/

[50] http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2013/10/ford.png

beeps when you touch this “cube” that takes up half the screen.

Finally, the “links” at the bottom of the screen are not actually links; they are `<div>`s with `onclick` attributes that change `window.location` to navigate the user. Thus, these controls are inaccessible from the keyboard and are confusing for screen readers.

Just when you thought things couldn’t get any worse, let’s look at the last of the Big Three automakers, Chrysler. Its mobile website[51] is shown below.



Chrysler’s mobile website, too, has accessibility issues. Items in quotation marks are what VoiceOver on OS X reads. (View large version[52])

---

[51] http://m.chrysler.com/

You can see that almost nothing on Chrysler's website provides any context for screen readers. We are reminded of the importance of meaningful `alt` attributes. Here, the one `alt` attribute that *is* provided — "This Is the Hero Carousal" — is less than helpful. Worse, all images in the carousel have the exact same `alt` attributes; so, screen reader users would have no idea that different links are being presented. To add insult to injury, the word "carousel" is spelled incorrectly ("carousal"), causing it to be mispronounced.

These carousel images occupy over half the space on an iPhone screen and are almost certainly the most clicked-on items on the screen. An `alt` attribute should tell users something about the image they're seeing, as well as where they will go if they click the link. For example:

```
<img src="/path/to/TnC.jpg" alt="Learn more about the
J.D. Power 2013 award-winning Chrysler Town &
Country">
```

This tells screen reader users what the link is for and what they will see if they select it.

Unfortunately, examples like these are far too common. Even though many accessibility best practices are well documented, they are frequently forgotten — even on big websites and especially on mobile.

Why?

---

52. http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2013/10/chrysler.png

One reason is that the consequences of violating these best practices are not obvious. To sighted touch users, all of these websites operate fine. Secondly, no good way exists to automate your website's accessibility. The W3C's validator[53] will warn of missing `alt` attributes, but it cannot test for more complex scenarios, such as icon buttons and lack of keyboard functionality. It also cannot test whether the `alt` attributes are actually meaningful.

Because of this, we must trust all web developers to learn and remember to use accessibility practices that have no apparent benefit. As we've seen in this case study, frequently they don't.

But all is not doom and gloom. We've seen that applying a few easy-to-implement best practices can drastically improve the accessibility of your sites, and open them to a new, surprisingly large audience. That building accessible sites builds a better experience for everyone.

But if the W3C validator cannot catch accessibility issues, how do you verify that you're applying these best practices correctly? As it turns out, the best way to test the accessibility of your site is also a great way to gain insight into how disabled users interact with it — by using a screen reader.

5. Test Your Website On An Actual Screen Reader

Most web developers have a slew of browsers and devices to test their websites, yet few know how to use a single

---

[53] http://validator.w3.org/

screen reader. Unfortunately, this has led most developers to treat accessibility guidelines as some sort of voodoo. They conjecture about what's best for impaired users without actually testing their theories.

This is a shame, because the best way to discover whether your website is accessible is to try it out as an impaired user would. Personally, I'm not sure why screen readers are shrouded in mystery; they're actually quite easy to use.

If you're on a Mac, type `Command + F5` to activate VoiceOver. Navigate around this page with the `Tab` key and see what's read. You can also press `Control + Option` plus the left and right arrow keys to target content that is not in the tab order. There are more advanced controls available[54], but you can get the idea with these basics.

If you're on Windows, you can download and use NVDA[55] for free. JAWS[56] is the most popular paid reader; it has a demo mode that you can try for free.

Mobile devices are a tad different because there is no keyboard by default. Therefore, using a screen reader forces you to reconsider how you interact with your device. Let's explore this by looking at the screen readers built into iOS and Android — VoiceOver and TalkBack, respectively.

---

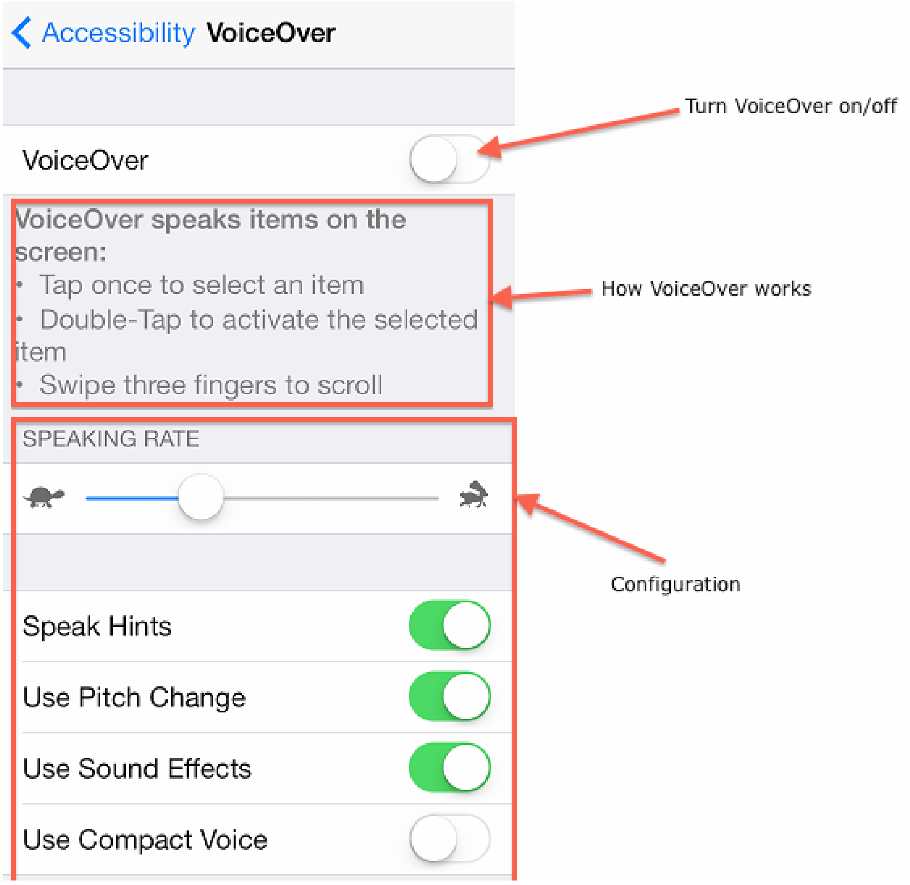
[54] http://www.apple.com/voiceover/info/guide/_1131.html

[55] http://www.nvaccess.org/

[56] http://www.freedomscientific.com/products/fs/jaws-product-page.asp

USING VOICEOVER ON IOS

VoiceOver is iOS' primary accessibility aid for all applications, not just the Web. Because most users don't use it, VoiceOver is disabled by default. To enable it, go to `Settings` → `General` → `Accessibility` → `VoiceOver`. You'll see the screen shown below.



< Accessibility **VoiceOver**

VoiceOver ○

VoiceOver speaks items on the screen:

- Tap once to select an item
- Double-Tap to activate the selected item
- Swipe three fingers to scroll

SPEAKING RATE

Speak Hints ●

Use Pitch Change ●

Use Sound Effects ●

Use Compact Voice ○

Turn VoiceOver on/off

How VoiceOver works

Configuration

The settings screen for VoiceOver on iOS

Be warned — once you turn it on, VoiceOver will fundamentally change the way you interact with the phone. Therefore, you *must* learn the basics; otherwise, you won't be able to turn VoiceOver back off.

Once VoiceOver is on, a single tap on the screen will select an item but will not activate it. For example, if you tap on the Safari icon, VoiceOver will read “Safari” but will not launch the application. After a selection, a double tap is needed to actually start the application. This makes sense if you consider the perspective of someone who cannot see the icon. They would need confirmation that they’ve selected the correct item before activating it.

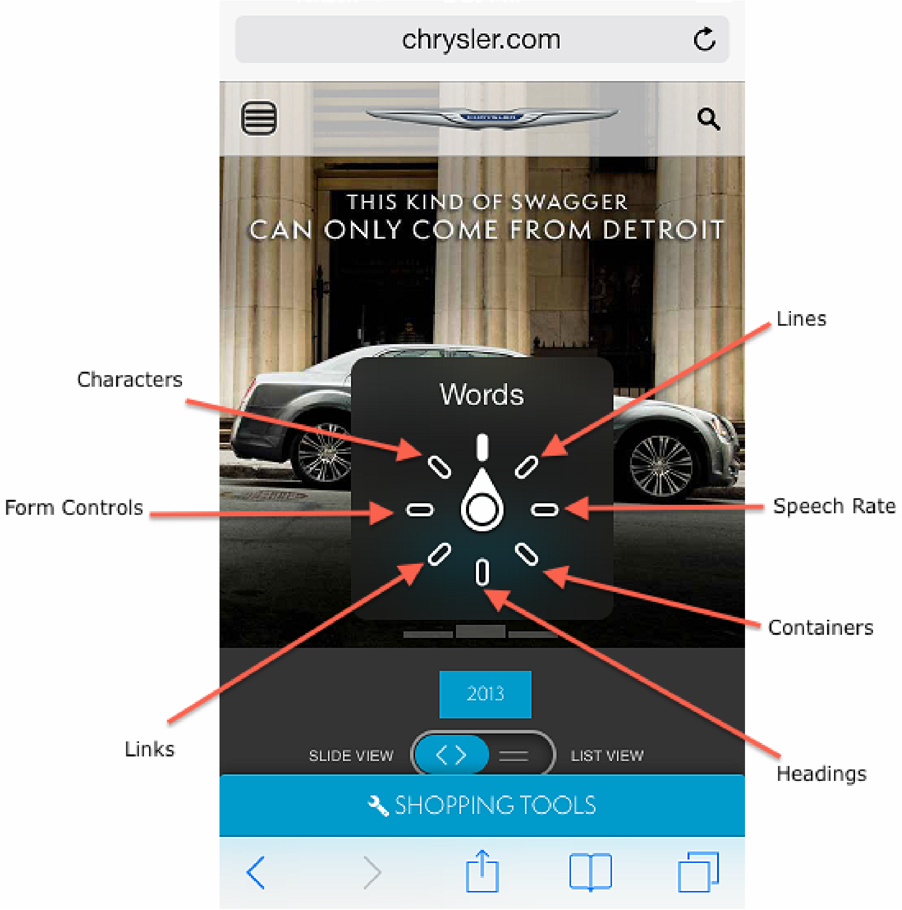
The other important difference with VoiceOver is that you have to use three fingers to scroll. Why? Because while in VoiceOver mode, iOS listens for one-finger “flick” gestures in all four directions:

- “Up”
  Move to previous item based on rotor setting
- “Down”
  Move to next item based on rotor setting
- “Right”
  Move to previous item
- “Left”
  Move to next item

What’s this “rotor”? We’ll get to that in a moment. First, try flicking left and right on the screen to move between available items. These flicks let vision-impaired users discover what’s on the screen without having to tap around.

Things get more interesting with VoiceOver’s rotor, a means of configuring how to navigate between items on the screen.

To activate the rotor, rotate two fingers on the screen as if you were turning a dial. The rotor and its default options are shown below.



The rotor control in VoiceOver for iOS

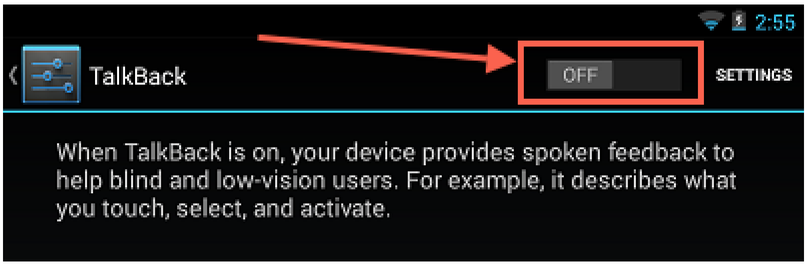
By setting the rotor, you can change how up and down flicks navigate the page. For instance, if the rotor is set to “Links,” then up and down flicks will cycle between links on the page and nothing else. In this sense, the rotor acts as a filter, enabling users to sift through the type of content that they’re interested in.

The rotor shows the importance of semantic markup. If your links are not actually `<a>` tags, then they won’t show up in the rotor setting for "Links." If your form's controls are not actual form elements, then they won't show up in the rotor setting for "Form Controls."

As you can see, using a screen reader forces you to fundamentally change how you use and approach the web on a mobile device. Play around with any websites you maintain to see how well you can navigate and accomplish tasks. As an added challenge, once you're familiar with VoiceOver, try closing your eyes and see what you can get done.

USING TALKBACK ON ANDROID

Like VoiceOver, TalkBack is an accessibility service for vision-impaired users that is native to Android devices. Because most people do not need the service, it is also disabled by default. To enable it, go to `Settings → Accessibility → TalkBack` and tap the switch shown below.



When TalkBack is on, your device provides spoken feedback to help blind and low-vision users. For example, it describes what you touch, select, and activate.

The settings page for TalkBack on Android

Again, be warned. Once TalkBack is activated, you'll at least need to know the basic controls to turn it off.

TalkBack's controls are extremely similar to VoiceOver's. One tap selects an item, and a double tap activates it. Scrolling with TalkBack requires two fingers (recall that VoiceOver requires three), and TalkBack listens for the same flick actions to move between items on the page.

While TalkBack has nothing comparable to VoiceOver's rotor, it does support a number of additional gestures[57] to customize the navigation.

Wrapping Up

In this article, we've discussed a number of best practices to improve the accessibility of your websites. How do you apply this information to your existing or new websites? Here are a few action items:

- Keyboard
  - Make sure that all tasks can be performed using only the keyboard.
  - Use semantic elements — `<button>` for buttons and `<a>` for links.
  - If you're having trouble making complex widgets keyboard-friendly, consider using a framework.
- Forms
  - Use the actual `<form>` element.
  - Associate all form elements (`<input>`, `<select>` and `<textarea>`) with a `<label>`.

---

[57] https://support.google.com/nexus/answer/2926960?hl=en

  - Make sure the “Enter” key can be used to submit the form.

- Contrast
  - Ensure that all text has a contrast ratio of at least 3.0, using Lea Verou’s Contrast Ratio[58]. Ideally, all text should meet this criterion, but focus on the main content first.

- Images
  - Include `alt` attributes that describe the images.

- Links and buttons
  - Always give these controls readable content. If the content should not be shown to sighted users, then use an image-replacement technique to hide it.

While this list is not comprehensive, it does provide a number of practices that are easy to implement and that address the most common problems affecting disabled users. Furthermore, we’ve seen that adhering to these guidelines benefits everyone, not just the disabled.

Finally, the best way to test your website is on an actual screen reader. By taking mere minutes to learn some commands, you will gain insight into how a good portion of the population interacts on the web.

---

[58] http://leaverou.github.io/contrast-ratio/

RESOURCES

- "Web Content Accessibility Guidelines 2.0[59]," W3C
  If you're looking for a more comprehensive checklist than what's provided in the conclusion above, this is it.

- "Shared Web Experiences: Barriers Common to Mobile Device Users and People With Disabilities[60]," W3C Web Accessibility Initiative
  This page describes the issues that people with disabilities encounter on the web and how they are addressed by the W3C's guidelines and specifications.

- "Talk To Me[61]," Jörn Zaefferer
  These slides are from Zaefferer's 2013 talk on making websites accessible.

- Contrast Rebellion[62]
  An amusing look at why contrast is important on the web.

- "VoiceOver Getting Started[63]," Apple
  The official guide to getting started with VoiceOver on OS X.

- The Accessibility Project[64]
  A nice collection of accessibility tips.

---

[59] http://www.w3.org/TR/WCAG/
[60] http://www.w3.org/WAI/mobile/experiences
[61] http://jzaefferer.github.io/talk-to-me/
[62] http://contrastrebellion.com/
[63] http://help.apple.com/voiceover/info/guide/10.8/English.lproj/
[64] http://a11yproject.com/

- "You Can't Create a Button[65]," Nicholas Zakas
  Zakas explains when to use `<a>` and `<button>` elements.

❧

---

[65] http://www.nczonline.net/blog/2013/01/29/you-cant-create-a-button/

Making Modal Windows Better For Everyone

BY SCOTT O'HARA ❧

To you, modal windows[66] might be a blessing of additional screen real estate, providing a way to deliver contextual information, notifications and other actions relevant to the current screen. On the other hand, modals might feel like a hack that you've been forced to commit in order to cram extra content on the screen. These are the extreme ends of the spectrum, and users are caught in the middle. Depending on how a user browses the Internet, modal windows can be downright confusing.

Modals quickly shift visual focus from one part of a website or application to another area of (hopefully related) content. The action is usually not jarring if initiated by the user, but it can be annoying and disorienting if it occurs automatically, as happens with the modal window's evil cousins, the "nag screen" and the "interstitial."

However, modals are merely a mild annoyance in the end, right? The user just has to click the "close" button, quickly skim some content or fill out a form to dismiss it.

Well, imagine that you had to navigate the web with a keyboard. Suppose that a modal window appeared on the screen, and you had very little context to know what it is and why it's obscuring the content you're trying to browse. Now you're wondering, "How do I interact with

---

[66] http://www.smashingmagazine.com/2009/05/27/modal-windows-in-modern-web-design/

this?" or "How do I get rid of it?" because your keyboard's focus hasn't automatically moved to the modal window.

This scenario is more common than it should be. And it's fairly easy to solve, as long as we make our content accessible to all through sound usability practices.

For an example, I've set up a demo of an inaccessible modal window[67] that appears on page load and that isn't entirely semantic. First, interact with it using your mouse to see that it actually works. Then, try interacting with it using only your keyboard.

Better Semantics Lead To Better Usability And Accessibility

Usability and accessibility are lacking in many modal windows. Whether they're used to provide additional actions or inputs for interaction with the page, to include more information about a particular section of content, or to provide notifications that can be easily dismissed, modals need to be easy for everyone to use.

To achieve this goal, first we must focus on the semantics of the modal's markup. This might seem like a no-brainer, but the step is not always followed.

Suppose that a popular gaming website has a full-page modal overlay and has implemented a "close" button with the code below:

---

[67] http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2014/inaccessible.html

```
<div id="modal_overlay">
  <div id="modal_close" onClick="modalClose()">
    X
  </div>
  …
</div>
```

This `div` element has no semantic meaning behind it. Sighted visitors will know that this is a "close" button because it looks like one. It has a hover state, so there is some visual indication that it can be interacted with.

But this element has no inherit semantic meaning to people who use a keyboard or screen reader.

There's no default way to enable users to tab to a `div` without adding a `tabindex` attribute to it. However, we would also need to add a `:focus` state to visually indicate that it is the active element. That still doesn't give screen readers enough information for users to discern the element's meaning. An "X" is the only label here. While we can assume that people who use screen readers would know that the letter "X" means "close," if it was a multiplication sign (using the HTML entity `&times;`) or a cross mark (`&#x274c;`), then some screen readers wouldn't read it at all. We need a better fallback.

We can circumvent all of these issues simply by writing the correct, semantic markup for a button and by adding an ARIA label for screen readers:

```
<div id="modal_overlay">
  <button type="button" class="btn-close"
id="modal_close" aria-label="close">
```

```
    X
  </button>
</div>
```

By changing the `div` to a button, we've significantly improved the semantics of our "close" button. We've addressed the common expectation that the button can be tabbed to with a keyboard and appear focused, and we've provided context by adding the ARIA label for screen readers.

That's just one example of how to make the markup of our modals more semantic, but we can do a lot more to create a useful and accessible experience.

Making Modals More Usable And Accessible

Semantic markup goes a long way to building a fully usable and accessible modal window, but still more CSS and JavaScript can take the experience to the next level.

INCLUDING FOCUS STATES

Provide a focus state! This obviously isn't exclusive to modal windows; many elements lack a proper focus state in some form or another beyond the browser's basic default one (which may or may not have been cleared by your CSS reset). At the very least, pair the focus state with the hover state you've already designed:

```
.btn:hover, .btn:focus {
  background: #f00;
}
```

However, because focusing and hovering are different types of interaction, giving the focus state its own style makes sense.

```
.btn:hover {
  background: #f00;
}

:focus {
  box-shadow: 0 0 3px rgba(0,0,0,.75);
}
```

Really, any item that can be focused should have a focus state. Keep that in mind if you're extending the browser's default dotted outline.

SAVING LAST ACTIVE ELEMENT

When a modal window loads, the element that the user last interacted with should be saved. That way, when the modal window closes and the user returns to where they were, the focus on that element will have been maintained. Think of it like a bookmark. Without it, when the user closes the modal, they would be sent back to the beginning of the document, left to find their place. Add the following to your modal's opening and closing functions to save and reenable the user's focus.

```
var lastFocus;

function modalShow () {
  lastFocus = document.activeElement;
}

function modalClose () {
  lastFocus.focus(); // place focus on the saved
  // element
}
```

SHIFTING FOCUS

When the modal loads, focus should shift from the last active element either to the modal window itself or to the first interactive element in the modal, such as an input element. This will make the modal more usable because sighted visitors won't have to reach for their mouse to click on the first element, and keyboard users won't have to tab through a bunch of DOM elements to get there.

```
var modal =
document.getElementById('your-modal-id-here');

function modalShow () {
   modal.setAttribute('tabindex', '0');
   modal.focus();
}
```

GOING FULL SCREEN

If your modal takes over the full screen, then obscure the contents of the main document for both sighted users and screen reader users. Without this happening, a keyboard user could easily tab their way outside of the modal without realizing it, which could lead to them interacting with the main document before completing whatever the modal window is asking them to do.

Use the following JavaScript to confine the user's focus to the modal window until it is dismissed:

```
function focusRestrict ( event ) {
  document.addEventListener('focus', function( event
) {
    if ( modalOpen && !modal.contains( event.target )
) {
      event.stopPropagation();
      modal.focus();
    }
  }, true);
}
```

While we want to prevent users from tabbing through the rest of the document while a modal is open, we don't want to prevent them from accessing the browser's chrome (after all, sighted users wouldn't expect to be stuck in the browser's tab while a modal window is open). The JavaScript above prevents tabbing to the document's content outside of the modal window, instead bringing the user to the top of the modal.

If we also put the modal at the top of the DOM tree, as the first child of `body`, then hitting `Shift + Tab` would take the user out of the modal and into the browser's chrome. If you're not able to change the modal's location in the DOM tree, then use the following JavaScript instead:

```
var m = document.getElementById('modal_window'),
    p = document.getElementById('page');

// Remember that <div id="page"> surrounds the whole
// document, so aria-hidden="true" can be applied to
// it when the modal opens.
function swap () {
  p.parentNode.insertBefore(m, p);
}

swap();
```

If you can't move the modal in the DOM tree or reposition it with JavaScript, you still have other options for confining focus to the modal. You could keep track of the first and last focusable elements in the modal window. When the user reaches the last one and hits `Tab`, you could shift focus back to the top of the modal. (And you would do the opposite for `Shift + Tab`.)

A second option would be to create a list of all focusable nodes in the modal window and, upon the modal firing, allow for tabbing only through those nodes.

A third option would be to find all focusable nodes outside of the modal and set `tabindex="-1"` on them.

The problem with these first and second options is that they render the browser's chrome inaccessible. If you must take this route, then adding a well-marked "close" button to the modal and supporting the `Escape` key are critical; without them, you will effectively trap keyboard users on the website.

The third option allows for tabbing within the modal and the browser's chrome, but it comes with the performance cost of listing all focusable elements on the page and negating their ability to be focused. The cost might not be much on a small page, but on a page with many links and form elements, it can become quite a chore. Not to mention, when the modal closes, you would need to return all elements to their previous state.

Clearly, we have a lot to consider to enable users to effectively tab within a modal.

DISMISSING

Finally, modals should be easy to dismiss. Standard `alert()` modal dialogs can be closed by hitting the `Escape` key, so following suit with our modal would be expected — and a convenience. If your modal has multiple focusable elements, allowing the user to just hit `Escape` is much better than forcing them to tab through content to get to the "close" button.

```
function modalClose ( e ) {
  if ( !e.keyCode || e.keyCode === 27 ) {
    // code to close modal goes here
  }
}
```

```
document.addEventListener('keydown', modalClose);
```

Moreover, closing a full-screen modal when the overlay is clicked is conventional. The exception is if you don't want to close the modal until the user has performed an action.

Use the following JavaScript to close the modal when the user clicks on the overlay:

```
mOverlay.addEventListener('click', function( e )
  if (e.target == modal.parentNode)
    modalClose( e );
  }
}, false);
```

Additional Accessibility Steps

Beyond the usability steps covered above, ARIA roles, states and properties[68] will add yet more hooks for assistive technologies. For some of these, nothing more is required than adding the corresponding attribute to your markup; for others, additional JavaScript is required to control an element's state.

ARIA-HIDDEN

Use the `aria-hidden` attribute. By toggling the value `true` and `false`, the element and any of its children will be either hidden or visible to screen readers. However, as with all ARIA attributes, it carries no default style and,

---

[68] http://www.w3.org/TR/wai-aria/

thus, will not be hidden from sighted users. To hide it, add the following CSS:

```
.modal-window[aria-hidden="true"] {
  display: none;
}
```

Notice that the selector is pretty specific here. The reason is that we might not want all elements with `aria-hidden="true"` to be hidden (as with our earlier example of the “X” for the “close” button).

ROLE="DIALOG"

Add `role="dialog"` to the element that contains the modal’s content. This tells assistive technologies that the content requires the user’s response or confirmation. Again, couple this with the JavaScript that shifts focus from the last active element in the document to the modal or to the first focusable element in the modal.

However, if the modal is more of an error or alert message that requires the user to input something before proceeding, then use `role="alertdialog"` instead. Again, set the focus on it automatically with JavaScript, and confine focus to the modal until action is taken.

ARIA-LABEL

Use the `aria-label` or `aria-labelledby` attribute along with `role="dialog"`. If your modal window has a heading, you can use the `aria-labelledby` attribute to point to it by referencing the heading’s ID. If your modal doesn’t have a heading for some reason, then you can at least use the `aria-label` to provide a concise label about the element that screen readers can parse.

What About HTML5's Dialog Element?

Chrome 37 beta and Firefox Nightly 34.0a1 support the `dialog` element, which provides extra semantic and accessibility information for modal windows. Once this native `dialog` element is established, we won't need to apply `role="dialog"` to non-dialog elements. For now, even if you're using a polyfill for the `dialog` element, also use `role="dialog"` so that screen readers know how to handle the element.

The exciting thing about this element is not only that it serves the semantic function of a dialog, but that it come with its own methods, which will replace the JavaScript and CSS that we currently need to write.

For instance, to display or dismiss a dialog, we'd write this base of JavaScript:

```
var modal = document.getElementById('myModal'),
  openModal = document.getElementById('btnOpen'),
  closeModal = document.getElementById('btnClose');

// to show our modal
openModal.addEventListener( 'click', function( e ) {
  modal.show();
  // or
  modal.showModal();
});
```

```
// to close our modal
closeModal.addEventListener( 'click', function( e ) {
  modal.close();
});
```

The `show()` method launches the dialog, while still allowing users to interact with other elements on the page. The `showModal()` method launches the dialog and prevents users from interacting with anything but the modal while it's open.

The `dialog` element also has the `open` property, set to `true` or `false`, which replaces `aria-hidden`. And it has its own `::backdrop` pseudo-element, which enables us to style the modal when it is opened with the `showModal()` method.

There's more to learn about the `dialog` element than what's mentioned here. It might not be ready for prime time, but once it is, this semantic element will go a long way to helping us develop usable, accessible experiences.

Where To Go From Here?

Whether you use a jQuery plugin or a homegrown solution, step back and evaluate your modal's overall usability and accessibility. As minor as modals are to the web overall, they are common enough that if we all tried to make them friendlier and more accessible, we'd make the web a better place.

I've prepared a demo of a modal window[69] that implements all of the accessibility features covered in this article. ❧

---

[69] http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2014/accessible.html

Notes On Client-Rendered Accessibility

BY MARCY SUTTON ❧

As creators of the web, we bring innovative, well-designed interfaces to life. We find satisfaction in improving our craft with each design or line of code. But this push to elevate our skills can be self-serving: Does a new CSS framework or JavaScript abstraction pattern serve our users or us as developers?

If a framework encourages best practices in development while also improving our workflow, it might serve both our users' needs and ours as developers. If it encourages best practices in accessibility alongside other areas, like performance, then it has potential to improve the state of the web.

Despite our pursuit to do a better job every day, sometimes we forget about accessibility, the practice of designing and developing in a way that's inclusive of people with disabilities. We have the power to improve lives through technology — we should use our passion for the craft to build a more accessible web.

These days, we build a lot of client-rendered web applications, also known as single-page apps, JavaScript MVCs and MV-whatever. AngularJS, React, Ember, Backbone.js, Spine: You may have used or seen one of these JavaScript frameworks in a recent project. Common user experience-related characteristics include asynchronous postbacks, animated page transitions, and dynamic UI filtering. With frameworks like these, creating a poor user experience for people with disabilities is, sadly, pretty easy. Fortunately, we can employ best practices to make things better.

In this article, we will explore techniques for building accessible client-rendered web applications, making our jobs as web creators even more worthwhile.

Semantics

Front-end JavaScript frameworks make it easy for us to create and consume custom HTML tags like `<pizza-button>`, which you'll see in an example later on. React, AngularJS and Ember enable us to attach behavior to made-up tags with no default semantics, using JavaScript and CSS. We can even use Web Components[70] now, a set of new standards holding both the promise of extensibility and a challenge to us as developers. With this much flexibility, it's critical for users of assistive technologies such as screen readers that we use semantics to communicate what's happening without relying on a visual experience.

Consider a common form control[71]: A checkbox opting you out of marketing email is pretty significant to the user experience. If it isn't announced as "Subscribe checked check box" in a screen reader, you might have no idea you'd need to uncheck it to opt out of the subscription. In client-side web apps, it's possible to construct a form model from user input and post JSON to a server re-

---

[70] http://www.smashingmagazine.com/2014/03/04/introduction-to-custom-elements/

[71] http://webaim.org/techniques/forms/controls

gardless of how we mark it up — possibly even without a `<form>` tag. With this freedom, knowing how to create accessible forms is important.

To keep our friends with screen readers from opting in to unwanted email, we should:

- use native inputs to easily announce their role (purpose) and state (checked or unchecked);
- provide an accessible name using a `<label>`, with `id` and `for` attribute pairing — `aria-label` on the input or `aria-labelledby` pointing to another element's `id`.

```
<form>
  <label for="subscribe">
    Subscribe
  </label>
  <input type="checkbox" id="subscribe" checked>
</form>
```

NATIVE CHECKBOX WITH LABEL

If native inputs can't be used (with good reason), create custom checkboxes with `role=checkbox`, `aria-checked`, `aria-disabled` and `aria-required`, and wire up keyboard events. See the W3C's "Using WAI-ARIA in HTML[72]."

---

[72] http://www.w3.org/TR/aria-in-html/

CUSTOM CHECKBOX WITH ARIA

```
<form>
  <some-checkbox role="checkbox" tabindex="0"
  aria-labelledby="subscribe"
  aria-checked="true">
  </some-checkbox>
  <some-label id="subscribe">Subscribe</some-label>
</form>
```

Form inputs are just one example of the use of semantic HTML[73] and ARIA attributes to communicate the purpose of something — other important considerations include headings and page structure, buttons, anchors, lists and more. ARIA[74], or Accessible Rich Internet Applications, exists to fill in gaps where accessibility support for HTML falls short (in theory, it can also be used for XML or SVG). As you can see from the checkbox example, ARIA requirements quickly pile up when you start writing custom elements. Native inputs, buttons and other semantic elements provide keyboard and accessibility support for free. The moment you create a custom element and bolt ARIA attributes onto it, you become responsible for managing the role and state of that element.

Although ARIA is great and capable of many things, understanding and using it is a lot of work. It also doesn't have the broadest support. Take Dragon NaturallySpeaking[75] — this assistive technology, which people use all the

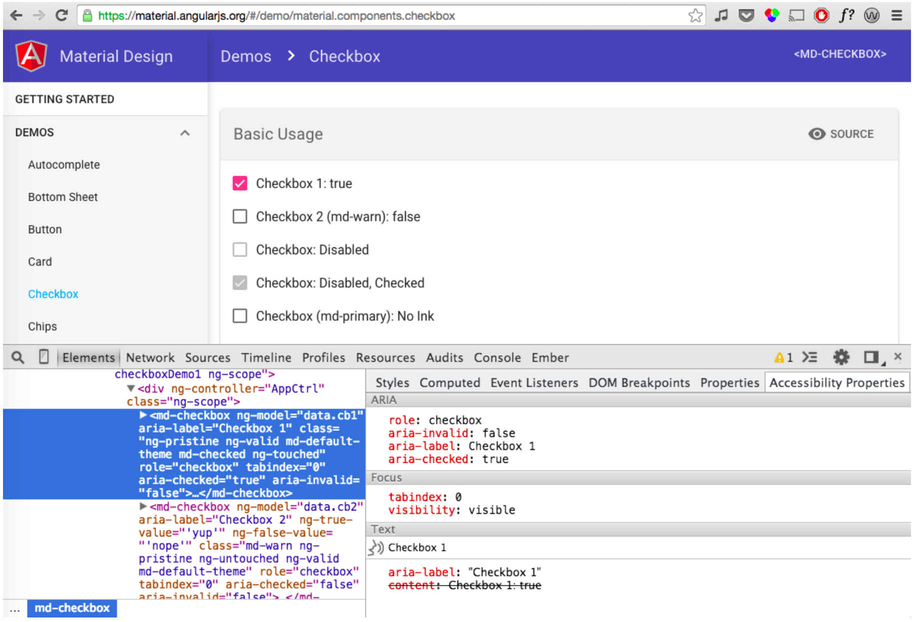
---

[73] http://webaim.org/techniques/semanticstructure/

[74] http://www.w3.org/TR/wai-aria/

time to make their life easier, is just starting to gain ARIA support. Were I a browser implementer, I'd focus on native element support first, too — so it makes sense that ARIA might be added later. For this reason, use native elements, and you won't often need to use ARIA roles or states (`aria-checked`, `aria-disabled`, `aria-required`, etc.). If you must create custom controls, read up on ARIA to learn the expected keyboard behavior[76] and how to use attributes correctly.

Tip: Use Chrome's Accessibility Developer Tools[77] to audit your code for errors, and you'll get the bonus "Accessibility Properties" inspector.



https://material.angularjs.org/#/demo/material.components.checkbox
Material Design
Demos > Checkbox
<MD-CHECKBOX>
GETTING STARTED
DEMOS
Autocomplete
Bottom Sheet
Button
Card
Checkbox
Chips
Basic Usage
SOURCE
Elements Network Sources Timeline Profiles Resources Audits Console Ember
Styles Computed Event Listeners DOM Breakpoints Properties Accessibility Properties
ARIA
Focus
Text
md-checkbox

☑ Checkbox 1: true
☐ Checkbox 2 (md-warn): false
☐ Checkbox: Disabled
☑ Checkbox: Disabled, Checked
☐ Checkbox (md-primary): No Ink

```
          checkboxDemo1 ng-scope">
            ▼<div ng-controller="AppCtrl"
            class="ng-scope">
              ▶<md-checkbox ng-model="data.cb1"
              aria-label="Checkbox 1" class=
              "ng-pristine ng-valid md-default-
              theme md-checked ng-touched"
              role="checkbox" tabindex="0"
              aria-checked="true" aria-invalid=
              "false">…</md-checkbox>
              ▶<md-checkbox ng-model="data.cb2"
              aria-label="Checkbox 2" ng-true-
              value="'yup'" ng-false-value=
              "'nope'" class="md-warn ng-
              pristine ng-untouched ng-valid
              md-default-theme" role="checkbox"
              tabindex="0" aria-checked="false"
              aria-invalid="false">…</md-
```

```
role: checkbox
aria-invalid: false
aria-label: Checkbox 1
aria-checked: true
```

```
tabindex: 0
visibility: visible
```

Checkbox 1

```
aria-label: "Checkbox 1"
```

~~content: Checkbox 1: true~~

AngularJS material in Chrome with accessibility inspector open.

---

75. http://assistivetechnology.about.com/od/SpeechRecognition/p/Dragon-Naturallyspeaking-As-Assistive-Technology.htm
76. http://www.w3.org/WAI/PF/aria-practices/#keyboard
77. https://chrome.google.com/webstore/detail/accessibility-developer-t/fpkknkljclfencbdbgkenhalefipecmb

WEB COMPONENTS AND ACCESSIBILITY

An important topic in a discussion on accessibility and semantics is Web Components, a set of new standards landing in browsers that enable us to natively create reusable HTML widgets. Because Web Components are still so new, the syntax is majorly in flux. In December 2014, Mozilla said it wouldn't support HTML imports[78], a seemingly obvious way to distribute new components; so, for now that technology is natively available in Chrome and Opera[79] only. Additionally, up for debate is the syntax for extending native elements (see the discussion about `is=""` syntax[80]), along with how rigid the shadow DOM boundary should be. Despite these changes, here are some tips for writing semantic Web Components:

- Small components are more reusable and easier to manage for any necessary semantics.
- Use native elements within Web Components to gain behavior for free.
- Element IDs within the shadow DOM do not have the same scope as the host document.
- The same non-Web Component accessibility guidelines apply.

For more information on Web Components and accessibility, have a look at these articles:

---

[78] https://hacks.mozilla.org/2014/12/mozilla-and-web-components/
[79] http://caniuse.com/#feat=imports
[80] https://lists.w3.org/Archives/Public/public-webapps/2015JanMar/0361.html

- “Polymer and Web Component Accessibility: Best Practices[81],” Dylan Barrell
- “Web Components Punch List[82],” Steve Faulkner
- “Accessible Web Components[83],” Addy Osmani and Alice Boxhall, Polymer

Interactivity

Native elements such as buttons and inputs come prepackaged with events and properties that work easily with keyboards and assistive technologies. Leveraging these features means less work for us. However, given how easy JavaScript frameworks and CSS make it to create custom elements, such as `<pizza-button>`, we might have to do more work to deliver pizza from the keyboard if we choose to mark it up as a new element. For keyboard support, custom HTML tags need:

- `tabindex`, preferably `0` so that you don’t have to manage the entire page’s tab order (WebAIM discusses this[84]);
- a keyboard event such as `keypress` or `keydown` to trigger callback functions.

---

[81] http://unobfuscated.blogspot.com/2015/03/polymer-and-web-component-accessibility.html

[82] http://www.paciellogroup.com/blog/2014/09/web-components-punch-list/

[83] https://www.polymer-project.org/0.5/articles/accessible-web-components.html

[84] http://webaim.org/techniques/keyboard/tabindex

Focus Management

Closely related to interactivity but serving a slightly different purpose is focus management. The term "client-rendered" refers partly to a single-page browsing experience where routing is handled with JavaScript and there is no server-side page refresh. Portions of views could update the URL and replace part or all of the DOM, including where the user's keyboard is currently focused. When this happens, focus is easily lost, creating a pretty unusable experience for people who rely on a keyboard or screen reader.

Imagine sorting a list with your keyboard's arrow keys. If the sorting action rebuilds the DOM, then the element that you're using will be rerendered, losing focus in the process. Unless focus is deliberately sent back to the element that was in use, you'd lose your place and have to tab all the way down to the list from the top of the page again. You might just leave the website at that point. Was it an app you needed to use for work or to find an apartment? That could be a problem.

In client-rendered frameworks, we are responsible for ensuring that focus is not lost when rerendering the DOM. The easy way to test this is to use your keyboard. If you're focused on an item and it gets rerendered, do you bang your keyboard against the desk and start over at the top of the page or gracefully continue on your way? Here is one focus-management technique from Distiller[85] us-

---

[85] http://drinkdistiller.com

ing Spine, where focus is sent back into relevant content after rendering:

```
class App.FocusManager
constructor:
$(‘body’).on ‘focusin’, (e) =>
@oldFocus = e.target

App.bind 'rendered', (e) =>
return unless @oldFocus

if @oldFocus.getAttribute('data-focus-id')
@_focusById()
else
@_focusByNodeEquality()

_focusById: ->
focusId = @oldFocus.getAttribute('data-focus-id')
newFocus = document.querySelector("##{focusId}")
App.focus(newFocus) if newFocus

_focusByNodeEquality: ->
allNodes = $('body *:visible').get()
for node in allNodes
if App.equalNodes(node, @oldFocus)
App.focus(node)
```

In this helper class, JavaScript (implemented in CoffeeScript) binds a `focusin` listener to `document.body` that checks anytime an element is focused, using event delegation[86], and it stores a reference to that focused element. The helper class also subscribes to a Spine `rendered` event, tapping into client-side rendering so that it can gracefully handle focus. If an element was focused before the rendering happened, it can focus an element in one of two ways. If the old node is identical to a new one somewhere in the DOM, then focus is automatically sent to it. If the node isn't identical but has a `data-focus-id` attribute on it, then it looks up that `id`'s value and sends focus to it instead. This second method is useful for when elements aren't identical anymore because their text has changed (for example, "item 1 of 5" becoming labeled off screen as "item 2 of 5").

Each JavaScript MV-whatever framework will require a slightly different approach to focus management. Unfortunately, most of them won't handle focus for you, because it's hard for a framework to know what should be focused upon rerendering. By testing rendering transitions with your keyboard and making sure focus is not dropped, you'll be empowered to add support to your application. If this sounds daunting, inquire in your framework's support community about how focus management is typically handled (see React's GitHub repo[87] for an example). There are people who can help!

Notifying The User

There is a debate about whether client-side frameworks are actually good for users[88], and plenty of people have an

---

[86] http://learn.jquery.com/events/event-delegation/
[87] https://github.com/facebook/react/issues/1791#issuecomment-82987932
[88] http://tantek.com/2015/069/t1/js-dr-javascript-required-dead

opinion[89] on them. Clearly, most client-rendered app frameworks could improve the user experience by providing easy asynchronous UI filtering, form validation and live content updates. To make these dynamic updates more inclusive, developers should also update users of assistive technologies when something is happening away from their keyboard focus.

Imagine a scenario: You're typing in an autocomplete widget and a list pops up, filtering options as you type. Pressing the down arrow key cycles through the available options, one by one. One technique to announce these selections would be to append messages to an ARIA live region[90], a mechanism that screen readers can use to subscribe to changes in the DOM. As long as the live region exists when the element is rendered, any text appended to it with JavaScript will be announced (meaning you can't add bind `aria-live` and add the first message at the same time). This is essentially how Angular Material[91]'s autocomplete handles dynamic screen-reader updates:

```
<md-autocomplete md-selected-item="ctrl.selectedItem"
aria-disabled="false">
<md-autocomplete-wrap role="listbox">
  <input type="text" aria-label="{{ariaLabel}}"
  aria-owns="ul_001">
</md-autocomplete-wrap>
<ul role="presentation" id="ul_001">
```

---

[89] https://adactio.com/journal/8245

[90] https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA_Live_Regions

[91] https://material.angularjs.org/

```
  <li ng-repeat="(index, item) in
  $mdAutocompleteCtrl.matches" role="option"
  tabIndex="0">
</ul>
<aria-status class="visually-hidden" role="alert">
  <p ng-repeat="message in messages">{{message}}</p>
</aria-status>
</md-autocomplete>
```

In the simplified code above (the full directive[92] and related controller[93] source are on GitHub), when a user types in the `md-autocomplete` text input, list items for results are added to a neighboring unordered list. Another neighboring element, `aria-status`, gets its `aria-live` functionality from the `alert` role. When results appear, a message is appended to `aria-status` announcing the number of items, “There is one match” or “There are four matches,” depending on the number of options. When a user arrows through the list, that item’s text is also appended to `aria-status`, announcing the currently highlighted item without the user having to move focus from the input. By curating the list of messages sent to an ARIA live region, we can implement an inclusive design that goes far beyond the visual. Similar regions can be used to validate forms.

For more information on accessible client-side validation, read Marco Zehe’s “Easy ARIA Tip #3: `aria-invalid`

---

[92] https://github.com/angular/material/blob/master/src/components/autocomplete/js/autocompleteDirective.js#L43

[93] https://github.com/angular/material/blob/master/src/components/autocomplete/js/autocompleteController.js

and Role `alert`[94]" or Deque's post on accessible forms[95].

Conclusion

So far, we've talked about accessibility with screen readers and keyboards. Also consider readability: This includes color contrast, readable fonts and obvious interactions. In client-rendered applications, all of the typical web accessibility principles[96] apply, in addition to the specific ones outlined above. The resources listed below will help you incorporate accessibility in your current or next project.

It is up to us as developers and designers to ensure that everyone can use our web applications. By knowing what makes an accessible user experience, we can serve a lot more people, and possibly even make their lives better. We need to remember that client-rendered frameworks aren't always the right tool for the job. There are plenty of legitimate use cases for them, hence their popularity. There are definitely drawbacks to rendering everything on the client[97]. However, even as solutions for seamless server- and client-side rendering improve over time, these same accessibility principles of focus management, semantics and alerting the user will remain true, and they will enable more people to use your apps. Isn't it cool that

---

94. https://www.marcozehe.de/2008/07/16/easy-aria-tip-3-aria-invalid-and-role-alert/
95. http://www.deque.com/blog/accessible-client-side-form-validation-html5-wai-aria/
96. http://webaim.org/intro/
97. http://alistapart.com/article/let-links-be-links

we can use our craft to help people through technology?

Resources

- "Web Accessibility for Designers[98]," WebAIM
- Accessibility Developer Tools[99]," Chrome plugin
- "Using WAI-ARIA in HTML[100]," W3C
- "How I Audit a Website for Accessibility[101]," Marcy Sutton, Substantial
- "Using ngAria[102]," Marcy Sutton
- "Protractor Accessibility Plugin[103]," Marcy Sutton
  Protractor is AngularJS' end-to-end testing framework. ❧

Thanks to Heydon Pickering for reviewing this article.

---

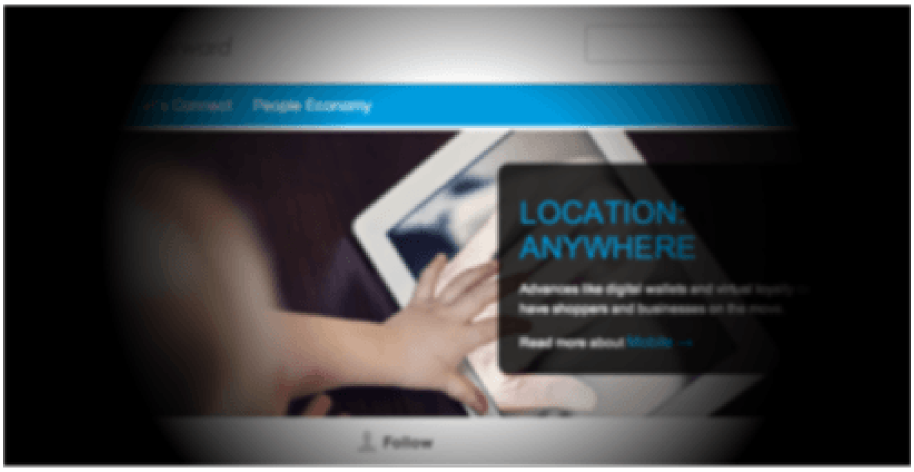
98. http://webaim.org/resources/designers/
99. https://chrome.google.com/webstore/detail/accessibility-developer-t/fpkknkljclfencbdbgkenhalefipecmb
100. http://www.w3.org/TR/aria-in-html/
101. http://substantial.com/blog/2014/07/22/how-i-audit-a-website-for-accessibility/
102. http://angularjs.blogspot.com/2014/11/using-ngaria.html
103. http://marcysutton.com/angular-protractor-accessibility-plugin/

Design Accessibly, See Differently: Color Contrast Tips And Tools

BY CATHY O'CONNOR ❧

When you browse your favorite website or check the latest version of your product on your device of choice, take a moment to look at it differently. Step back from the screen. Close your eyes slightly so that your vision is a bit clouded by your eyelashes.

- Can you still see and use the website?
- Are you able to read the labels, fields, buttons, navigation and small footer text?
- Can you imagine how someone who sees differently would read and use it?



Web page viewed with NoCoffee low-vision simulation.

In this article, I'll share one aspect of design accessibility: making sure that the look and feel (the visual design of the content) are sufficiently inclusive of differently sighted users.

I am a design consultant on PayPal's accessibility team. I assess how our product designs measure up to the Web Content Accessibility Guidelines (WCAG) 2.0, and I review our company's design patterns and best practices.

I created our "Designers' Accessibility Checklist," and I will cover one of the most impactful guidelines on the checklist in this article: making sure that there is sufficient color contrast for all content. I'll share the strategies, tips and tools that I use to help our teams deliver designs that most people can see and use without having to customize the experiences.

Our goal is to make sure that all visual designs meet the minimum color-contrast ratio for normal and large text on a background, as described in the WCAG 2.0, Level AA, "Contrast (Minimum): Understanding Success Criterion 1.4.3[104]."

Who benefits from designs that have sufficient contrast? Quoting from the WCAG's page:

> *The 4.5:1 ratio is used in this provision to account for the loss in contrast that results from moderately low visual acuity, congenital or acquired color deficiencies, or the loss of contrast sensitivity that typically accompanies aging.*

---

[104] http://www.w3.org/TR/UNDERSTANDING-WCAG20/visual-audio-contrast-contrast.html

As an accessibility consultant, I'm often asked how many people with disabilities use our products. Website analytics do not reveal this information. Let's estimate how many people could benefit from designs with sufficient color contrast by reviewing the statistics:

- 15% of the world's population have some form of disability[105], which includes conditions that affect seeing, hearing, motor abilities and cognitive abilities.
- About 4% of the population have low vision, whereas 0.6% are blind.
- 7 to 12% of men have some form of color-vision deficiency (color blindness), and less than 1% of women do.
- Low-vision conditions increase with age, and half of people over the age of 50 have some degree of low-vision condition.
- Worldwide, the fastest-growing population is 60 years of age and older[106].
- Over the age of 40, most everyone will find that they need reading glasses or bifocals to clearly see small objects or text, a condition caused by the natural aging process, called presbyopia[107].

---

[105] http://www.who.int/mediacentre/factsheets/fs352/en/

[106] http://www.un.org/esa/population/publications/worldageing19502050/

[107] http://www.mayoclinic.org/diseases-conditions/presbyopia/basics/causes/con-20032261

Let's estimate that 10% of the world population would benefit from designs that are easier to see. Multiply that by the number of customers or potential customers who use your website or application. For example, out of 2 million online customers, 200,000 would benefit.

Some age-related low-vision conditions[108] include the following:

- **Macular degeneration**
  Up to 50% of people are affected by age-related vision loss.
- **Diabetic retinopathy**
  In people with diabetes, leaking blood vessels in the eyes can cloud vision and cause blind spots.
- **Cataracts**
  Cataracts clouds the lens of the eye and decreases visual acuity.
- **Retinitis pigmentosa**
  This inherited condition gradually causes a loss of vision.

All of these conditions reduce sensitivity to contrast, and in some cases reduce the ability to distinguish colors.

Color-vision deficiencies, also called color-blindness, are mostly inherited and can be caused by side effects of medication and age-related low-vision conditions.

---

[108] https://www.nei.nih.gov/healthyeyes/aging_eye.asp

Here are the types of color-vision deficiencies[109]:

- **Deuteranopia**
  This is the most common and entails a reduced sensitivity to green light.
- **Protanopia**
  This is a reduced sensitivity to red light.
- **Tritanopia**
  This is a reduced sensitivity to blue light, but not very common.
- **Achromatopsia**
  People with this condition cannot see color at all, but it is not very common.

Reds and greens or colors that contain red or green can be difficult to distinguish for people with deuteranopia or protanopia.

Experience Seeing Differently

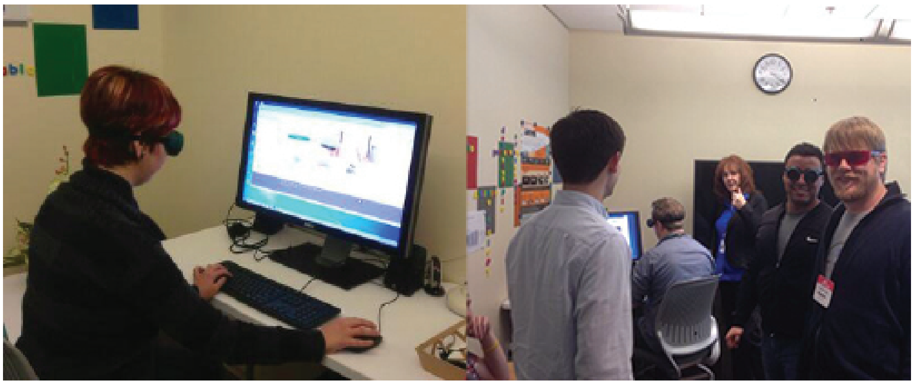
Creating a checklist and asking your designers to use it is easy, but in practice how do you make sure everyone follows the guidelines? We've found it important for designers not only to intellectually understand the why, but to experience for themselves what it is like to see differently. I've used a couple of strategies: immersing designers in interactive experiences through our Accessibility

---

[109] http://webaim.org/articles/visual/colorblind

Showcase, and showing what designs look like using software simulations.

In mid-2013, we opened our PayPal Accessibility Showcase[110] (video). Employees get a chance to experience firsthand what it is like for people with disabilities to use our products by interacting with web pages using goggles and/or assistive technology. We require that everyone who develops products participates in a tour. The user scenarios for designing with sufficient color contrast include wearing goggles that simulate conditions of low or partial vision and color deficiencies. Visitors try out these experiences on a PC, Mac or tablet. For mobile experiences, visitors wear the goggles and use their own mobile devices.



Showcase visitors wear goggles that simulate low-vision and color-blindness conditions.

Fun fact: One wall in the showcase was painted with magnetic paint. The wall contains posters, messages and concepts that we want people to remember. At the end of

---

[110] https://www.youtube.com/watch?feature=player_embedded&v=7MyHZofcNnk

the tour, I demonstrate vision simulators on our tablet. I view the message wall with the simulators to emphasize the importance of sufficient color contrast.

Some of the goggles used in the Accessibility Showcase.

Software Simulators

MOBILE APPS

Free mobile apps are available for iOS and Android devices:

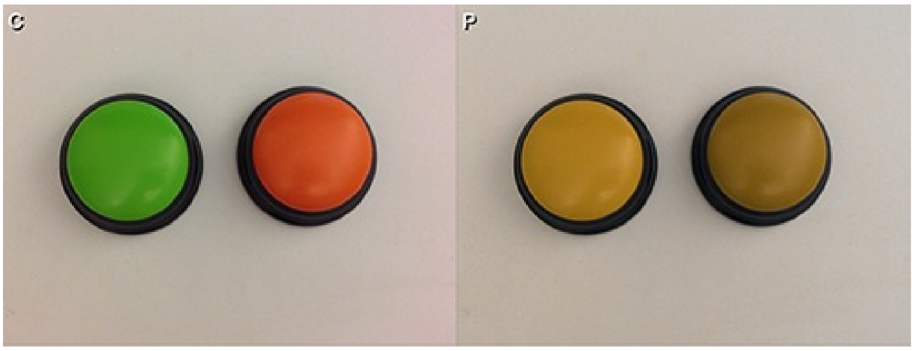
- **Chromatic Vision Simulator**
  Kazunori Asada's app simulates three forms of color deficiencies: protanope (protanopia), deuteranope (deuteranopia) and tritanope (tritanopia). You can view and then

save simulations using the camera feature, which takes a screenshot in the app. (Available for iOS[111] and Android[112].)

- **VisionSim**
  The Braille Institute's app simulates a variety of low-vision conditions and provides a list of causes and symptoms for each condition. You can view and then save simulations using the camera feature, which takes a screenshot in the app. (Available for iOS[113] and Android[114].)

CHROMATIC VISION SIMULATOR

The following photos show orange and green buttons viewed through the Chromatic Vision Simulator:



Seen through Chromatic Vision Simulator, the green and orange buttons show normal (C) and protanope (P).

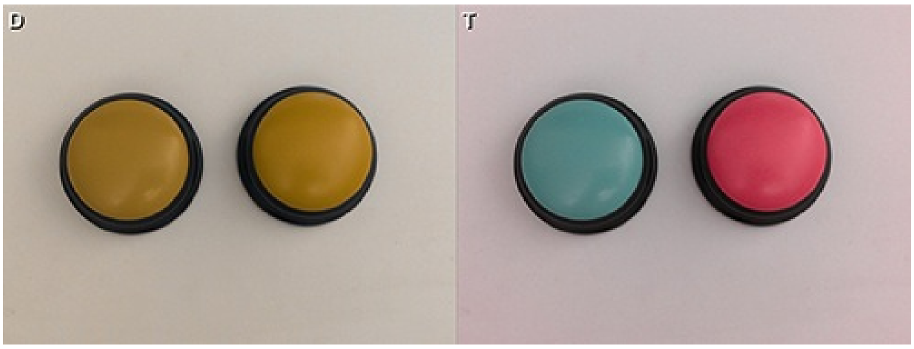
---

[111] https://itunes.apple.com/us/app/chromatic-vision-simulator/id389310222?mt=8

[112] https://play.google.com/store/apps/details?id=asada0.android.cvsimulator&hl=en

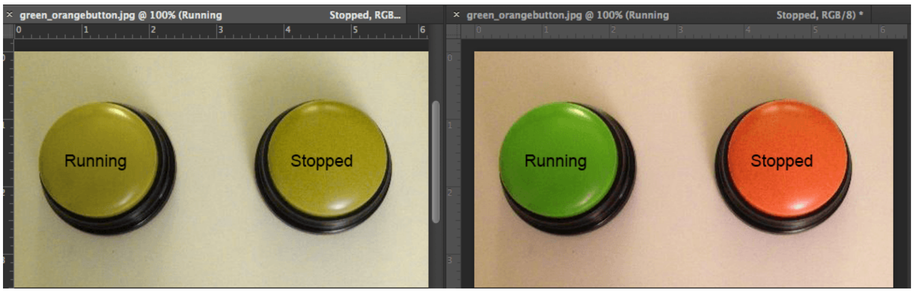
[113] https://itunes.apple.com/us/app/visionsim-by-braille-institute/id525114829?mt=8

[114] https://play.google.com/store/apps/details?id=com.BrailleIns.VisionSim&hl=en



Seen through Chromatic Vision Simulator, the green and orange buttons show deuteranope (D) and tritanope (T).

This example highlights the importance of another design accessibility guideline: Do not use color alone to convey meaning. If these buttons were online icons representing a system's status (such as up or down), some people would have difficulty understanding it because there is no visible text and the shapes are the same. In this scenario, include visible text (i.e. text labels), as shown in the following example:


green_orangebutton.jpg @ 100% (Running
Stopped, RGB...
Running
Stopped
green_orangebutton.jpg @ 100% (Running
Stopped, RGB/8) *
Running
Stopped

The green and orange buttons are viewed in Photoshop with deuteranopia soft proof and normal (text labels added).

MOBILE DEVICE SIMULATIONS

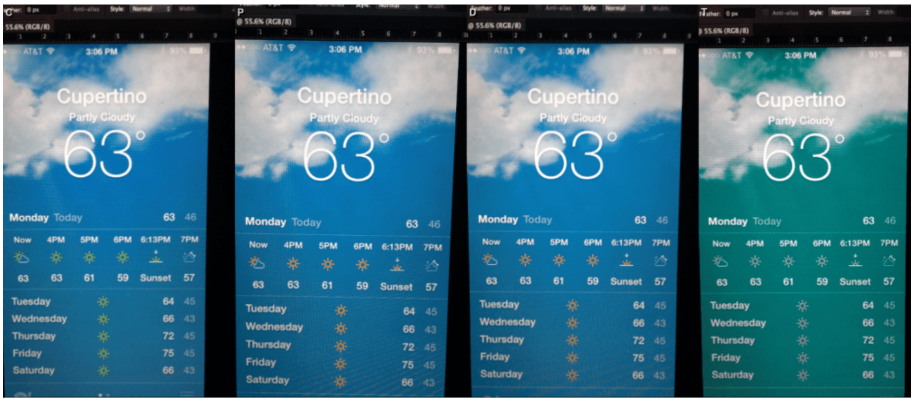
Checking for sufficient color contrast becomes even more important on mobile devices. Viewing mobile applications through VisionSim or Chromatic Vision Simulator is easy if you have two mobile phones. View the mobile app that you want to test on the second phone running the simulator.

If you only have one mobile device, you can do the following:

1. Take screenshots of the mobile app on the device using the built-in camera.
2. Save the screenshots to a laptop or desktop.
3. Open and view the screenshots on the laptop, and use the simulators on the mobile device to view and save the simulations.

HOW'S THE WEATHER IN CUPERTINO?

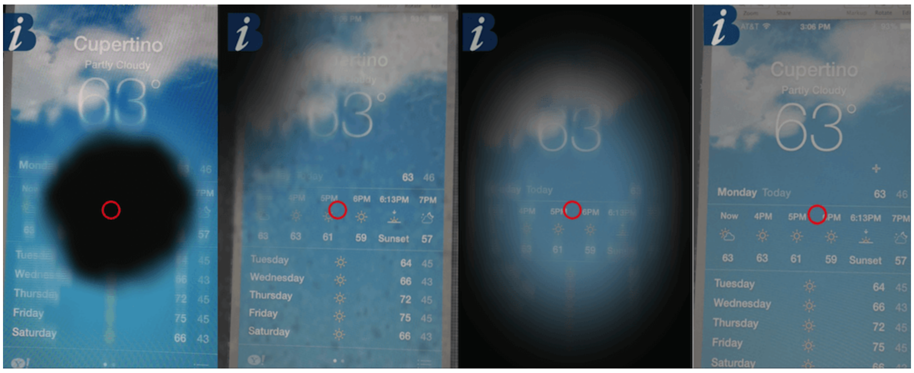
The following example highlights the challenges of using a photograph as a background while making essential information easy to see. Notice that the large text and bold text are easier to see than the small text and small icons.



The Weather mobile app, viewed with Chromatic Vision Simulator, shows normal, deuteranope, protanope and tritanope simulations.

LOW-VISION SIMULATIONS

Using the VisionSim app, you can simulate macular degeneration, diabetic retinopathy, retinitis pigmentosa and cataracts.



The Weather mobile app is being viewed with the supported condition simulations.

Adobe Photoshop

PayPal's teams use Adobe Photoshop to design the look and feel of our user experiences. To date, a color-contrast ratio checker or tester is not built into Photoshop. But designers can use a couple of helpful features in Photoshop to check their designs for sufficient color contrast:

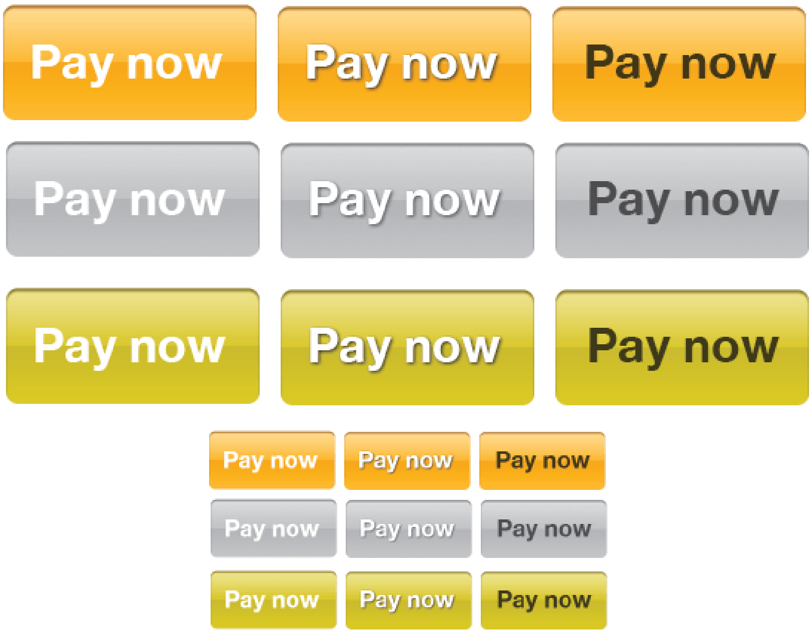
- Convert designs to grayscale by going to "Select View" → "Image" → "Adjustments" → "Grayscale."
- Simulate color blindness conditions by going to "Select View" → "Proof Setup" → "Color Blindness" and choosing protanopia type or deuteranopia type. Adobe provides soft-proofs for color blindness[115].

EXAMPLES

If you're designing with gradient backgrounds, verify that the color-contrast ratio passes for the text color and background color on both the lightest and darkest part of the gradient covered by the content or text.

In the following example of buttons, the first button has white text on a background with an orange gradient, which does not meet the minimum color-contrast ratio. A couple of suggested improvements are shown:

- add a drop-shadow color that passes (center button),
- change the text to a color that passes (third button).



Button with gradients: normal view; view in grayscale; and as a proof, deuteranopia.

---

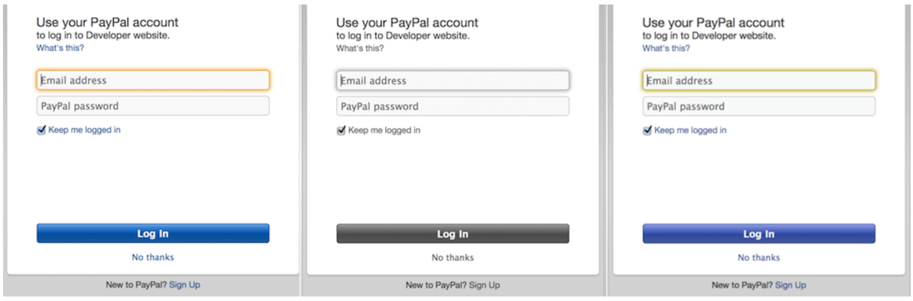
115. http://help.adobe.com/en_US/creativesuite/cs/using/WS3F71DA01-0962-4b2e-B7FD-C956F8659BB3.html#WS473A333A-7F61-4aba-8F67-5553208E349C

Checking in Photoshop with the grayscale and deuteranopia proof, the modified versions with the drop shadow and dark text are easier to read than the white text.

If you design in sizes larger than actual production sizes, make sure to check how the design will appear in the actual web page or mobile device.

In the following example of a form, the body text and link text pass the minimum color-contrast ratio for both the white and the gray background. I advise teams to always check the color contrast of text and links against all background colors that are part of the experience.

Even though the "Sign Up" link passes, if we view the experience in grayscale or with proof deuteranopia, distinguishing that "Sign Up" is a link might be difficult. To improve the affordance of "Sign Up" as a link, underline the link or link the entire phrase, "New to PayPal? Sign Up."



Use your PayPal account to log in to Developer website.

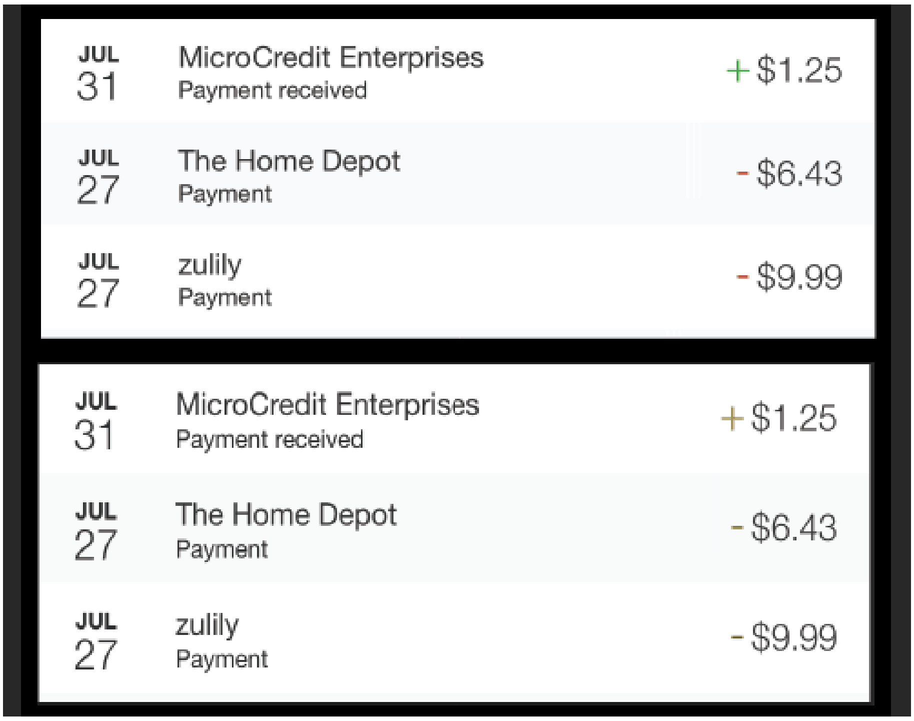
Use your PayPal account to log in to Developer website.

Use your PayPal account to log in to Developer website.

Form example: normal view; in Photoshop, a view in grayscale; and as a proof, deuteranopia.

Because red and green can be more difficult to distinguish for people with conditions such as deuteranopia and protanopia, should we avoid using them? Not necessarily. In the following example, a red minus sign ("-") indicates purchasing or making a payment. Money received or refunded is indicated by a green plus sign ("+"). Viewing the design with proof, deuteranopia, the colors are not easy to distinguish, but the shapes are legible and unique. Next to the date, the description describes the type of payment. Both shape and content provide context for the information.

Also shown in this example, the rows for purchases and refunds alternate between white and light-gray backgrounds. If the same color text is used for both backgrounds, verify that all of the text colors pass for both white and gray backgrounds.



JUL 31	MicroCredit Enterprises Payment received	+ $1.25
JUL 27	The Home Depot Payment	- $6.43
JUL 27	zulily Payment	- $9.99

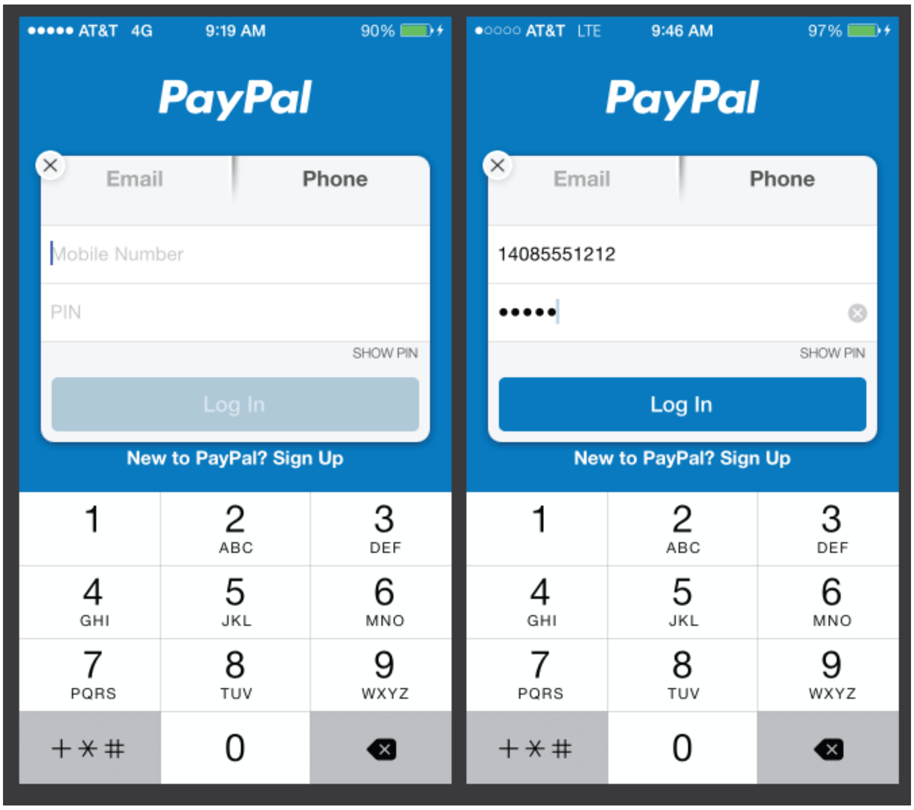
JUL 31	MicroCredit Enterprises Payment received	+ $1.25
JUL 27	The Home Depot Payment	- $6.43
JUL 27	zulily Payment	- $9.99

Normal view and as a proof, deuteranopia: Check the text against the alternating background colors.

In some applications, form fields and/or buttons may be disabled until information has been entered by the user.

Our design guidance does not require disabled elements to pass, in accordance with the WCAG 2.0's "Contrast (Minimum): Understanding Success Criterion 1.4.3[116]:

> *Incidental: Text or images of text that are part of an inactive user interface component,… have no contrast requirement.*



14085551212

•••••

Mobile app form showing disabled fields and button (left) and then enabled (right).

In the above example of a mobile app's form, the button is disabled until a phone number and PIN have been en-

---

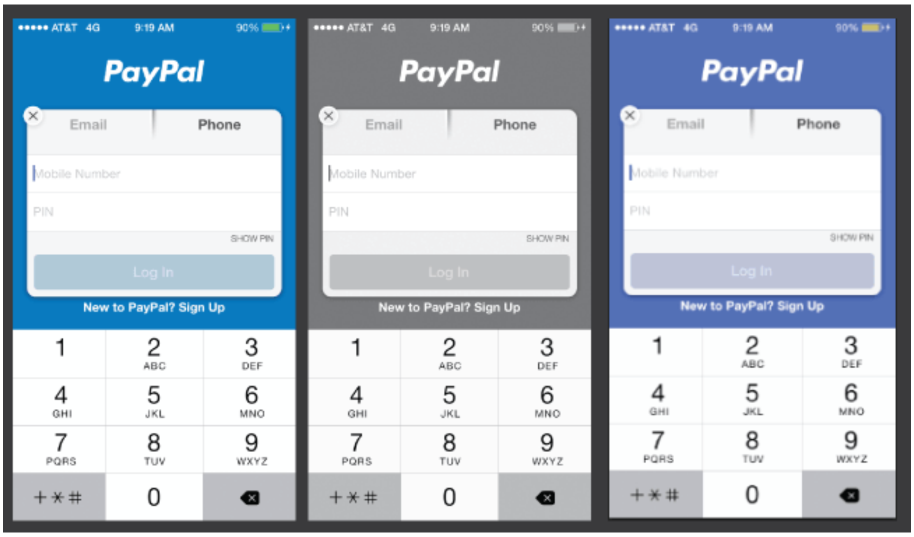
[116] http://www.w3.org/TR/2014/NOTE-UNDERSTANDING-WCAG20-20140311/visual-audio-contrast-contrast.html

tered. The text labels for the fields are a very light gray over a white background, which does not pass the minimum color-contrast ratio.

If the customer interprets that form elements with low contrast are disabled, would they assume that the entire form is disabled?

The same mobile app form is shown in a size closer to what I see on my phone in the following example. At a minimum, the text color needs to be changed or darkened to pass the minimum color-contrast ratio for normal body text and to improve readability.

To help distinguish between labels in fields and user-entered information, try to explore alternative visual treatments of form fields. Consider reversing foreground and background colors or using different font styles for labels and for user-entered information.

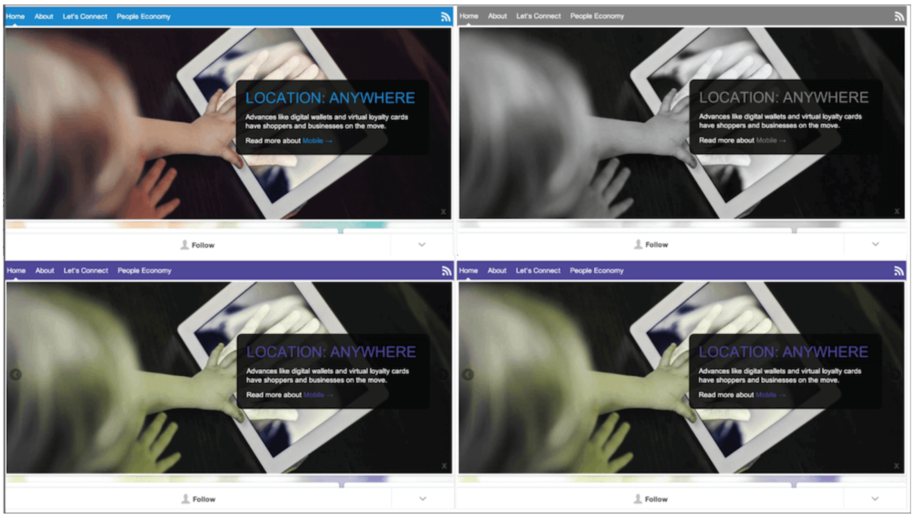


Mobile app form example: normal, grayscale and proof deuteranopia.

NoCoffee Vision Simulator for Chrome

NoCoffee Vision Simulator[117] can be used to simulate color-vision deficiencies and low-vision conditions on any pages that are viewable in the Chrome browser. Using the “Color Deficiency” setting “achromatopsia,” you can view web pages in grayscale.

The following example shows the same photograph (featuring a call to action) viewed with some of the simulations available in NoCoffee.



LOCATION: ANYWHERE

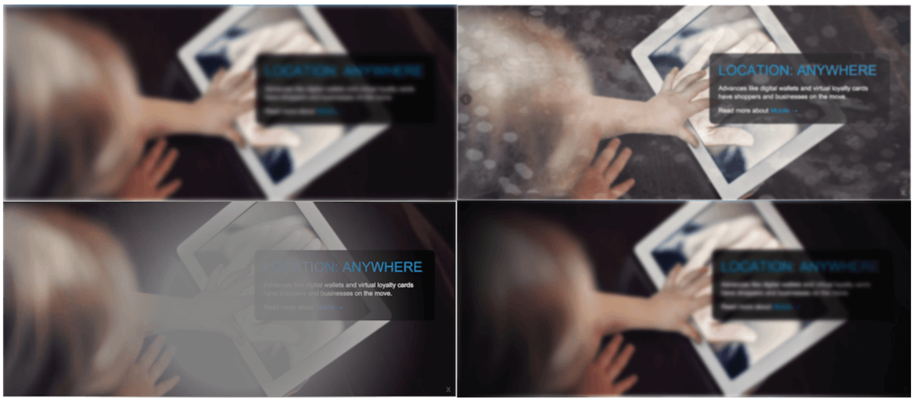
Advances like digital wallets and virtual loyalty cards have shoppers and businesses on the move.

Read more about Mobile →

Simulating achromatopsia (no color), deuteranopia, protanopia using NoCoffee.

---

[117] https://chrome.google.com/webstore/search/NoCoffee%20Vision%20Simulator?hl=en&gl=US



Simulating low visual acuity, diabetic retinopathy, macular degeneration and low visual acuity plus retinitus pigmentosa, using NoCoffee.

The message and call to action are separated from the background image by a practically opaque black container. This improves readability of the message and call to action. Testing the color contrast of the blue color in the headline against solid black passes for large text. Note that the link “Mobile” is not as easy to see because the blue does not pass the color-contrast standard for small body text. Possible improvements could be to change the link color to white and underline it, and/or make the entire phrase “Read more about Mobile” a link.

Using Simulators

Simulators are useful tools to visualize how a design might be viewed by people who are aging, have low-vision conditions or have color-vision deficiencies.

For design reviews, I use the simulators to mock up a design in grayscale, and I might use color-blindness filters to show designers possible problems with color contrast. Some of the questions I ask are:

- Is anything difficult to read?
- Is the call to action easy to find and read?
- Are links distinguishable from other content?

After learning how to use simulators to build empathy and to see their designs differently, I ask designers to use tools to check color contrast to verify that all of their designs meet the minimum color-contrast ratio of the WCAG 2.0 AA. The checklist includes a couple of tools they can use to test their designs.

Color-Contrast Ratio Checkers

The tools we cite in the designers' checklist are these:

- WebAIM Color Contrast Checker[118], a browser-based tool, tests color codes specified in hexadecimal values.
- The Paciello Group's Colour Contrast Checker[119], an application available for Macs or PCs, tests color codes specified in RGB values.

There are many tools to check color contrast, including ones that check live products. I've kept the list short to make it easy for designers to know what to use and to allow for consistent test results.

---

[118] http://webaim.org/resources/contrastchecker

[119] http://paciellogroup.com/resources/contrastAnalyser

Our goal is to meet the WCAG 2.0 AA color-contrast ratio, which is 4.5 to 1 for normal text and 3 to 1 for large text.

What are the minimum sizes for normal text and large text? The guidance provides recommendations on size ratios in the WCAG's Contrast (Minimum): Understanding Success Criterion 1.4.3[120] but not a rule for a minimum size for body text. As noted in the WCAG's guidance, thin decorative fonts might need to be larger and/or bold.

TESTING COLOR-CONTRAST RATIO

You should test:

- early in the design process;
- when creating a visual design specification for any product or service (this documents all of the color codes and the look and feel of the user experience);
- all new designs that are not part of an existing visual design guideline.

TEST HEXADECIMAL COLOR CODES FOR WEB DESIGNS

Let's use the WebAIM Color Contrast Checker[121] to test sample body-text colors on a white background (#FFFFFF):

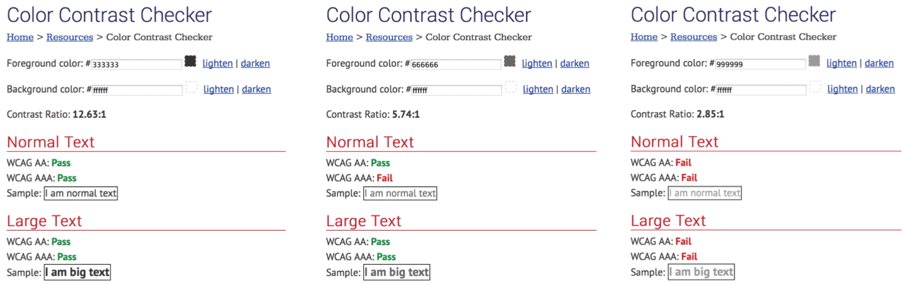
---

[120] http://www.w3.org/TR/2014/NOTE-UNDERSTANDING-WCAG20-20140311/visual-audio-contrast-contrast.html

[121] http://webaim.org/resources/contrastchecker

- dark-gray text (#333333).
- medium-gray text (#666666).
- light-gray text (#999999).

We want to make sure that body and normal text passes the WCAG 2.0 AA. Note that light gray (#999999) does not pass on a white background (#FFFFFF).



Foreground color: #333333
Background color: #ffffff
Contrast Ratio: **12.63:1**

Normal Text
WCAG AA: **Pass**
WCAG AAA: **Pass**
Sample: I am normal text

Large Text
WCAG AA: **Pass**
WCAG AAA: **Pass**
Sample: **I am big text**

Foreground color: #666666
Background color: #ffffff
Contrast Ratio: **5.74:1**

Normal Text
WCAG AA: **Pass**
WCAG AAA: **Fail**
Sample: I am normal text

Large Text
WCAG AA: **Pass**
WCAG AAA: **Pass**
Sample: **I am big text**

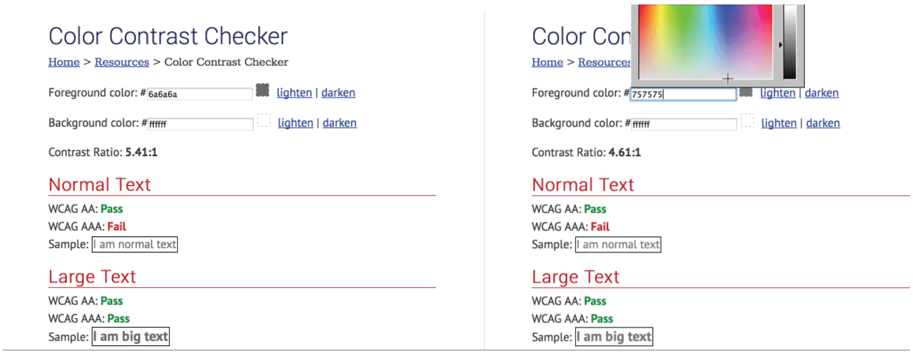
Foreground color: #999999
Background color: #ffffff
Contrast Ratio: **2.85:1**

Normal Text
WCAG AA: **Fail**
WCAG AAA: **Fail**
Sample: I am normal text

Large Text
WCAG AA: **Fail**
WCAG AAA: **Fail**
Sample: **I am big text**

Test dark-gray, medium-gray and light-gray using the WebAim Color Contrast Checker.

In the tool, you can modify the light gray (#999999) to find a color that does pass the AA. Select the “Darken” option to slightly change the color until it passes. By clicking the color field, you will have more options, and you can change color and luminosity, as shown in the second part of this example.



Color Contrast Checker
Home > Resources > Color Contrast Checker
Foreground color: # 6a6a6a

Background color: # ffffff

Contrast Ratio: 5.41:1
Normal Text
WCAG AA: Pass
WCAG AAA: Fail
Sample: I am normal text
Large Text
WCAG AA: Pass
WCAG AAA: Pass
Sample: I am big text

Color Con
Home > Resources
Foreground color: # 757575

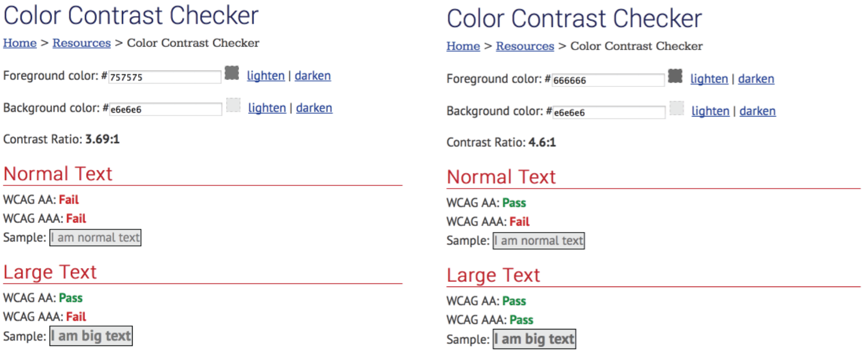
Background color: # ffffff

Contrast Ratio: 4.61:1
Normal Text
WCAG AA: Pass
WCAG AAA: Fail
Sample: I am normal text
Large Text
WCAG AA: Pass
WCAG AAA: Pass
Sample: I am big text

In the WebAim Color Contrast Checker, modify the light gray using the "Darken" option, or use the color palette to find a color that passes.

Tabular information may be designed with alternating white and gray backgrounds to improve readability. Let's test medium-gray text (#666666) and light-gray text (#757575) on a gray background (#E6E6E6).

Note that with the same background, the medium text passes, but the lighter gray passes only for large text. In this case, use medium gray for body text instead of white or gray backgrounds. Use the lighter gray only for large text, such as headings on white and gray backgrounds.



Color Contrast Checker
Home > Resources > Color Contrast Checker
Foreground color: # 757575

Background color: # e6e6e6

Contrast Ratio: 3.69:1
Normal Text
WCAG AA: Fail
WCAG AAA: Fail
Sample: I am normal text
Large Text
WCAG AA: Pass
WCAG AAA: Fail
Sample: I am big text

Color Contrast Checker
Home > Resources > Color Contrast Checker
Foreground color: # 666666

Background color: # e6e6e6

Contrast Ratio: 4.6:1
Normal Text
WCAG AA: Pass
WCAG AAA: Fail
Sample: I am normal text
Large Text
WCAG AA: Pass
WCAG AAA: Pass
Sample: I am big text

Test light-gray and medium-gray text on a gray background.

TEST RGB COLOR CODES

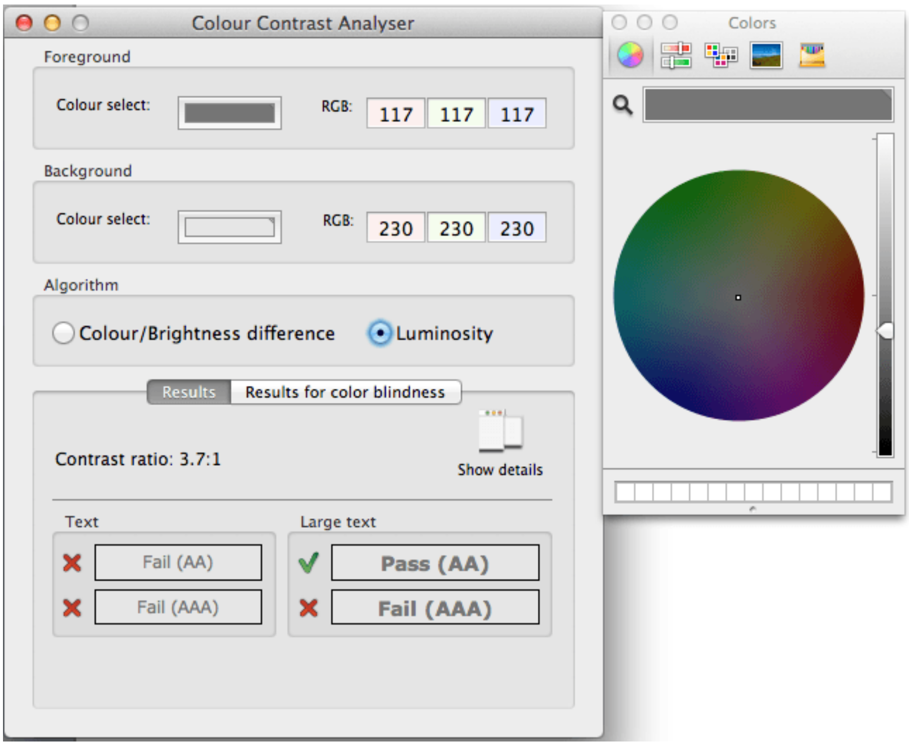
For mobile applications, designers might use RGB color codes to specify visual designs for engineering. You can use the TPG Colour Contrast Checker[122]. you will need to install either the PC or Mac version and run it side by side with Photoshop.

Let's use the Colour Contrast Checker to test medium-gray text (`102 102 102` in RGB and `#666666` in hexadecimal) and light-gray text (`#757575` in hexadecimal) on a gray background (`230 230 230` in RGB and `#E6E6E6` in hexadecimal).

1. Open the Colour Contrast Checker application.
2. Select "Options" → "Displayed Color Values" → "RGB."
3. Under "Algorithm," select "Luminosity."
4. Enter the foreground and background colors in RGB: `102 102 102` for foreground and `230 230 230` for background. Mouse click or tab past the fields to view the results. Note that this combination passes for both text and large text (AA).
5. Select "Show details" to view the hexadecimal color values and information about both AA and AAA requirements.

---

[122] http://paciellogroup.com/resources/contrastAnalyser



Foreground

Colour select: RGB: 117 117 117

Background

Colour select: RGB: 230 230 230

Algorithm

○ Colour/Brightness difference ● Luminosity

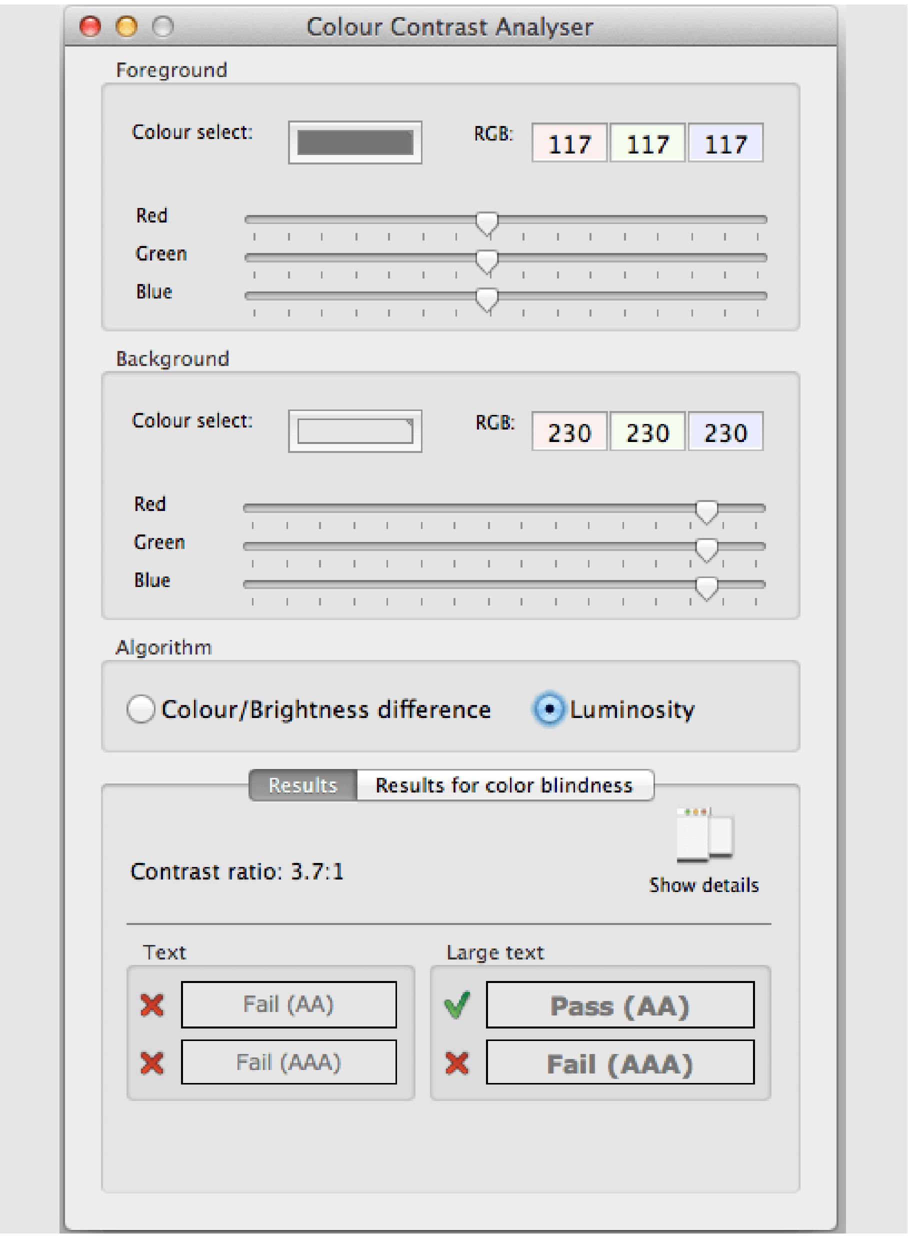
Results | Results for color blindness

Contrast ratio: 3.7:1

Text	Large text
✗ Fail (AA)	✓ Pass (AA)
✗ Fail (AAA)	✗ Fail (AAA)

Colour Contrast Analyser, and color wheel to modify colors.

In our example, light-gray text (`117 117 117` in RGB) on a gray background (`230 230 230` in RGB) does not meet the minimum AA contrast ratio for body text. To modify the colors, view the color wheels by clicking in the "Color" select box to modify the foreground or background. Or you can select "Options" → "Show Color Sliders," as shown in the example.



Foreground RGB: 117 117 117

Background RGB: 230 230 230

○ Colour/Brightness difference ● Luminosity

Contrast ratio: 3.7:1

Text	Large text
✗ Fail (AA)	✓ Pass (AA)
✗ Fail (AAA)	✗ Fail (AAA)

Colour Contrast Analyser, with RGB codes. Show color sliders to modify any color that does not meet minimum AA guidelines.

In most cases, minor adjustments to colors will meet the minimum contrast ratio, and comparisons before and after will show how better contrast enables most people to see and read more easily.

Best Practices

Test for color-contrast ratio, and document the styles and color codes used for all design elements. Create a visual design specification that includes the following:

- typography for all textual elements, including headings, text links, body text and formatted text;
- icons and glyphs and text equivalents;
- form elements, buttons, validation and system error messaging;
- background color and container styles (making sure text on these backgrounds all pass);
- the visual treatments for disabled links, form elements and buttons (which do not need to pass a minimum color-contrast ratio).

Documenting visual guidelines for developers brings several benefits:

- Developers don’t have to guess what the designers want.
- Designs can be verified against the visual design specification during quality testing cycles, by engineers and designers.
- A reference point that meets design accessibility guidelines for color contrast can be shared and leveraged by other teams.

Summary

If you are a designer, try out the simulators and tools on your next design project. Take time to see differently. One of the stellar designers who reviewed my checklist told me a story about using Photoshop's color-blindness proofs. On his own, he used the proofs to refine the colors used in a design for his company's product. When the redesigned product was released, his CEO thanked him because it was the first time he was able to see the design. The CEO shared that he was color-blind. In many cases, you may be unaware that your colleague, leader or customers have moderate low-vision or color-vision deficiencies. If meeting the minimum color-contrast ratio for a particular design element is difficult, take the challenge of thinking beyond color. Can you innovate so that most people can pick up and use your application without having to customize it?

If you are responsible for encouraging teams to build more accessible web or mobile experiences, be prepared to use multiple strategies:

- Use immersive experiences to engage design teams and gain empathy for people who see differently.
- Show designers how their designs might look using simulators.
- Test designs that have low contrast, and show how slight modifications to colors can make a difference.
- Encourage designers to test, and document visual specifications early and often.

- Incorporate accessible design practices into reusable patterns and templates both in the code and the design.

  Priorities and deadlines make it challenging for teams to deliver on all requests from multiple stakeholders. Be patient and persistent, and continue to engage with teams to find strategies to deliver user experiences that are easier to see and use by more people out of the box.

References

- "Contrast (Minimum): Understanding Success Criterion 1.4.3[123]" and the note[124], Web Content Accessibility Guidelines 2.0, Level AA
- "Get a Sneak Peek Into PayPal Accessibility Showcase[125]," Victor Tsaran and Cathy O'Connor, PayPal Engineering
- "Adobe Photoshop[126]" Accessibility, Adobe
- "Soft-Proof for Color Blindness (Photoshop and Illustrator)[127]," Adobe
- Web Accessibility in Mind[128] (WebAIM)

---

[123] http://www.w3.org/TR/UNDERSTANDING-WCAG20/visual-audio-contrast-contrast.html

[124] http://www.w3.org/TR/2014/NOTE-UNDERSTANDING-WCAG20-20140311/visual-audio-contrast-contrast.html

[125] https://www.paypal-engineering.com/2014/03/13/get-a-sneak-peek-into-paypal-accessibility-showcase/

[126] http://www.adobe.com/accessibility/products/photoshop.html

[127] http://help.adobe.com/en_US/creativesuite/cs/using/WS3F71DA01-0962-4b2e-B7FD-C956F8659BB3.html#WS473A333A-7F61-4aba-8F67-5553208E349C

[128] http://webaim.org

  - Web-based Color Contrast Checker[129] (web-based)
  - Web Accessibility Evaluation Tool[130] (WAVE)
  - “Visual Disabilities: Color-Blindness[131]”
- TPG Colour Contrast Analyser[132] (for Mac and PC), The Paciello Group
- Color Vision Simulator for iOS[133] and Android[134], Kazunori Asada
- VisionSim for iOS[135] and Android[136], The Braille Institute
- “NoCoffee Vision Simulator[137], Aaron Leventhal (also see Levanthal’s blog post about it[138])
- “Age-Related Eye Diseases[139],” National Eye Institute
- “Disability and Health[140],” World Health Organization
- “Presbyopia[141],” Mayo Clinic

---

[129] http://webaim.org/resources/contrastchecker/
[130] http://wave.webaim.org
[131] http://webaim.org/articles/visual/colorblind
[132] http://www.paciellogroup.com/resources/contrastAnalyser/
[133] https://itunes.apple.com/us/app/chromatic-vision-simulator/id389310222?mt=8
[134] https://play.google.com/store/apps/details?id=asada0.android.cvsimulator&hl=en
[135] https://itunes.apple.com/us/app/visionsim-by-braille-institute/id525114829?mt=8
[136] https://play.google.com/store/apps/details?id=com.BrailleIns.VisionSim&hl=en
[137] https://chrome.google.com/webstore/search/NoCoffee%20Vision%20Simulator?hl=en&gl=US
[138] http://accessgarage.wordpress.com/2013/02/09/458/
[139] https://www.nei.nih.gov/healthyeyes/aging_eye.asp
[140] http://www.who.int/mediacentre/factsheets/fs352/en/

- "World Population Ageing: 1950–2050[142]," UN Department of Economic and Social Affairs

LOW-VISION GOGGLES AND RESOURCES

- Zimmerman Low Vision Simulation Kit[143]
- Low Vision Simulators[144], Fork In the Road ❧

---

[141] http://www.mayoclinic.org/diseases-conditions/presbyopia/basics/causes/con-20032261

[142] http://www.un.org/esa/population/publications/worldageing19502050/

[143] http://www.lowvisionsimulationkit.com

[144] http://www.lowvisionsimulators.com/find-the-right-low-vision-simulator

Designing For The Elderly: Ways Older People Use Digital Technology Differently

BY OLLIE CAMPBELL ❧

If you work in the tech industry, it's easy to forget that older people exist. Most tech workers are really young[145], so it's easy to see why most technology is designed for young people. But consider this: By 2030, around 19% of people in the US will be over 65[146]. Doesn't sound like a lot? Well it happens to be about the same number of people in the US who own an iPhone today. Which of these two groups do you think Silicon Valley spends more time thinking about?

This seems unfortunate when you consider all of the things technology has to offer older people. A great example is Speaking Exchange[147], an initiative that connects retirees in the US with kids who are learning English in Brazil. Check out the video below, but beware — it's a tear-jerker.

---

[145] http://bits.blogs.nytimes.com/2013/07/05/technology-workers-are-young-really-young/

[146] http://www.aoa.gov/Aging_Statistics/

[147] http://www.cna.com.br/speakingexchange/

CNA – Speaking Exchange. (Watch the video on YouTube[148]*)*

While the ageing process is different for everyone, we all go through some fundamental changes. Not all of them are what you'd expect. For example, despite declining health, older people tend to be significantly happier[149] and better at appreciating what they have[150].

But ageing makes some things harder as well, and one of those things is using technology. If you're designing technology for older people, below are seven key things you need to know.

(How old is old? It depends. While I've deliberately avoided trying to define such an amorphous group using chronological boundaries, it's safe to assume that each of the following issues becomes increasingly significant after 65 years of age.)

---

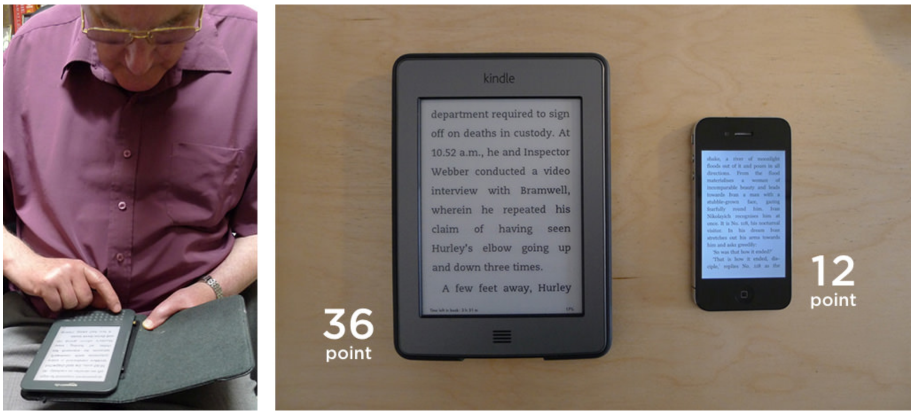
[148] https://youtu.be/-S-5EfwpFOk

[149] http://www.economist.com/node/17722567

[150] http://newoldage.blogs.nytimes.com/2014/02/11/what-makes-older-people-happy/

Vision And Hearing

From the age of about 40, the lens of the eye begins to harden, causing a condition called "presbyopia." This is a normal part of ageing that makes it increasingly difficult to read text that is small and close.



department required to sign off on deaths in custody. At 10.52 a.m., he and Inspector Webber conducted a video interview with Bramwell, wherein he repeated his claim of having seen Hurley's elbow going up and down three times.

A few feet away, Hurley

Here's a 75-year-old with his Kindle. Take a look at the font size he picks when he's in control. Now compare it to the average font size on an iPhone. (Image: Navy Design[151]*.)*

Color vision also declines with age, and we become worse at distinguishing between similar colors. In particular, shades of blue appear to be faded or desaturated.

Hearing also declines in predictable ways, and a large proportion of people over 65 have some form of hearing loss[152]. While audio is seldom fundamental to interaction with a product, there are obvious implications for certain types of content.

---

[151] http://www.navydesign.com.au

[152] http://www.nidcd.nih.gov/health/hearing/Pages/Age-Related-Hearing-Loss.aspx

Key lessons:

- Avoid font sizes smaller than 16 pixels (depending of course on device, viewing distance, line height etc.).
- Let people adjust text size themselves.
- Pay particular attention to contrast ratios[153] with text.
- Avoid blue for important interface elements.
- Always test your product using screen readers[154].
- Provide subtitles when video or audio content is fundamental to the user experience.

Motor Control

Our motor skills decline with age, which makes it harder to use computers in various ways. For example, during some user testing at a retirement village, we saw an 80-year-old who always uses the mouse with two hands. Like many older people, she had a lot of trouble hitting interface targets and moving from one thing to the next.

In the general population, a mouse is more accurate[155] than a finger. But in our user testing, we've seen older people perform better using touch interfaces. This is consistent with research that shows that finger tapping declines later[156] than some other motor skills.

---

[153] http://webaim.org/resources/contrastchecker/
[154] http://www.afb.org/prodBrowseCatResults.asp?CatID=49
[155] http://www.yorku.ca/mack/hfes2009.html

Key lessons:

- Reduce the distance between interface elements that are likely to be used in sequence (such as form fields), but make sure they're at least 2 millimeters apart.
- Buttons on touch interfaces should be at least 9.6 millimeters diagonally[157] (for example, 44 × 44 pixels on an iPad) for ages up to 70, and larger for older people.
- Interface elements to be clicked with a mouse (such as forms and buttons) should be at least 11 millimeters diagonally.
- Pay attention to sizing in human interface guidelines (Luke Wroblewski has a good roundup of guidelines[158] for different platforms).

Device Use

> *If you want to predict the future, just look at what middle-class American teens are doing. Right now, they're using their mobile phones for everything.*
> *– Dustin Curtis*[159]

It's safe to assume Dustin has never watched a 75-year-old use a mobile phone. Eventually, changes in vision and

---

156. http://www.medicaldaily.com/finger-tapping-test-shows-no-motor-skill-decline-until-after-middle-age-244927
157. http://dl.acm.org/citation.cfm?id=1152260
158. http://www.lukew.com/ff/entry.asp?1085
159. http://dcurt.is/the-death-of-the-tablet

motor control make small screens impractical for everyone. Smartphones are a young person's tool[160], and not even the coolest teenager can escape their biological destiny.

In our research, older people consistently described phones as "annoying" and "fiddly." Those who own them seldom use them, often not touching them for days at a time. They often ignore SMS' entirely.

Examples of technology used by the elderly (Image: Navy Design[161])

But older people aren't afraid to try new technology when they see a clear benefit. For example, older people are the

---

[160] http://www2.deloitte.com/content/dam/Deloitte/global/Documents/Technology-Media-Telecommunications/gx-tmt-2014prediction-smartphone.pdf

[161] http://www.navydesign.com.au

largest users of tablets[162]. This makes sense when you consider the defining difference between a tablet and a phone: screen size. The recent slump in tablet sales[163] also makes sense if you accept that older people have longer upgrade cycles than younger people.

Key lessons:

- Avoid small-screen devices (i.e. phones).
- Don't rely on SMS to convey important information.

Relationships

Older people have different relationships than young people, at least partly because they've had more time to cultivate them. For example, we conducted some research into how older people interact with health care professionals. In many cases, they've seen the same doctors for decades, leading to a very high degree of trust.

> *I regard it like going to see old pals.... I feel I could tell my GP almost anything.*
> *– George, 73, on visiting his medical team*

But due to health and mobility issues, the world available to the elderly is often smaller — both physically and socially. Digital technology has an obvious role to play here,

---

[162] http://dcurt.is/the-death-of-the-tablet
[163] http://recode.net/2014/08/26/in-defense-of-tablets/

by connecting people virtually when being in the same room is hard.

Key lessons:

- Enable connection with a smaller, more important group of people (not a big, undifferentiated social network).
- Don't overemphasize security and privacy controls when trusted people are involved.
- Be sensitive to issues of isolation.

Life Stage

During a user testing session, I sat with a 66-year-old as she signed up for an Apple ID. She was asked to complete a series of security questions. She read the first question out loud. “What was the model of your first car?” She laughed. “I have no idea! What car did I have in 1968? What a stupid question!”

It's natural for a 30-year-old programmer to assume that this question has meaning for everyone, but it contains an implicit assumption about which life stage the user is at. Don't make the same mistake in your design.

Key lessons:

- Beware of content or functionality that implicitly assumes someone is young or at a certain stage in life.

Experience With Technology

I once sat with a man in his 80s as he used a library interface. “I know there are things down there that I want to read” he said, gesturing to the bottom of the screen, “but I can’t figure out how to get to them.” After I taught him how to use a scrollbar, his experience changed completely. In another session, two of the older participants told me that they’d never used a search field before.

Generally when you’re designing interfaces, you’re working within a certain kind of scaffolding. And it’s easy to assume that everyone knows how that scaffolding works. But people who didn’t grow up with computers might have never used the interface elements we take for granted. Is a scrollbar a good design for moving content up and down? Is its function self-evident? These aren’t questions most designers often ask. But the success of your design might depend on a thousand parts of the interface that you can’t control and probably aren’t even aware of.

Key lessons:

- Don’t make assumptions about prior knowledge.
- Interrogate all parts of your design for usability, even the parts you didn’t create.

Cognition

The science of cognition is a huge topic, and ageing changes how we think in unpredictable ways. Some people are razor-sharp in their 80s, while others decline as early as in their 60s.

Despite this variability, three areas are particularly relevant to designing for the elderly: memory, attention and decision-making. (For a more comprehensive view of cognitive change with age, chapter 1 of *Brain Aging: Models, Methods, and Mechanisms*[164] is a great place to start.)

MEMORY

There are different kinds of memory, and they're affected differently by the ageing process. For example, procedural memory (that is, remembering how to do things) is generally unaffected. People of all ages are able to learn new skills and reproduce them over time.

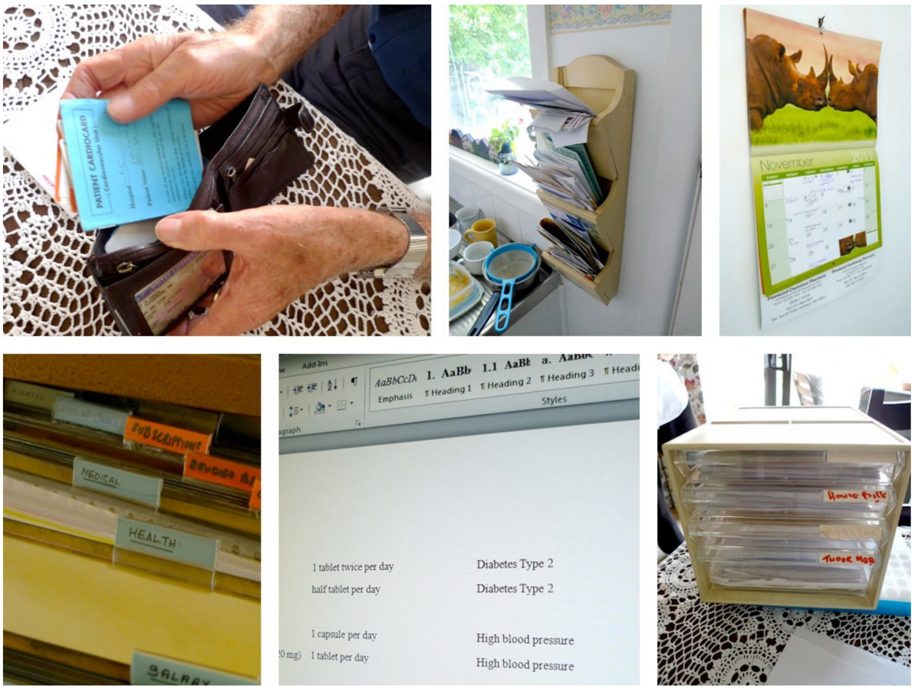
But other types of memory suffer as we age. Short-term memory and episodic memory are particularly vulnerable. And, although the causes are unclear, older people often have difficulty manipulating the contents of their working memory[165]. This means that they may have trouble understanding how to combine complex new concepts in a product or interface.

Prospective memory (remembering to do something in the future) also suffers[166]. This is particularly relevant for habitual tasks, like remembering to take medication at the right time every day.

---

164. http://www.ncbi.nlm.nih.gov/books/NBK3885/
165. http://www.psych.utoronto.ca/users/hasher/abstracts/hasher_zacks_88.htm
166. http://www.oxfordscholarship.com/view/10.1093/acprof:oso/9780195156744.001.0001/acprof-9780195156744-chapter-10

How do people manage this decline? In our research, we've found that paper is king. Older people almost exclusively use calendars and diaries to supplement their memory. But well-designed technology has great potential to provide cues for these important actions.



For older people, paper is king. (Image: Navy Design[167])

Key lessons:

- Introduce product features gradually over time to prevent cognitive overload.
- Avoid splitting tasks across multiple screens if they require memory of previous actions.

---

[167] http://www.navydesign.com.au

- During longer tasks, give clear feedback on progress and reminders of goals.
- Provide reminders and alerts as cues for habitual actions.

ATTENTION

It's easy to view ageing as a decline, but it's not all bad news. In our research, we've observed one big advantage: Elderly people consistently excel in attention span, persistence and thoroughness. Jakob Nielsen has observed similar things, finding that 95% of seniors are "methodical"[168] in their behaviors. This is significant in a world where the average person's attention span has actually dropped below the level of a goldfish[169].

It can be a great feeling to watch an older user really take the time to explore your design during a testing session. And it means that older people often find things that younger people skip right over. I often find myself admiring this way of interacting with the world. But the obvious downside of a slower pace is increased time to complete tasks.

Older people are also less adept at dividing their attention[170] between multiple tasks. In a world obsessed with multitasking, this can seem like a handicap. But because multi-tasking is probably a bad idea[171] in the first place,

---

[168] http://www.nngroup.com/articles/usability-for-senior-citizens/

[169] http://www.statisticbrain.com/attention-span-statistics/

[170] http://www.era.lib.ed.ac.uk/handle/1842/8572

[171] http://news.stanford.edu/news/2009/august24/multitask-research-study-082409.html

designing products that help people to focus on one thing at a time can have benefits for all age groups.

Key lessons:

- Don't be afraid of long-form text and deep content.
- Allow for greater time intervals in interactions (for example, server timeouts, inactivity warnings).
- Avoid dividing users' attention between multiple tasks or parts of the screen.

DECISION-MAKING

Young people tend to weigh a lot of options before settling on one. Older people make decisions a bit differently. They tend to emphasize prior knowledge[172] (perhaps because they've had more time to accumulate it). And they give more weight to the opinions of experts (for example, their doctor for medical decisions).

The exact reason for this is unclear, but it may be due to other cognitive limitations that make comparing new options more difficult.

Key lessons:

- Prioritize shortcuts to previous choices ahead of new alternatives.

---

[172] http://psycnet.apa.org/index.cfm?fa=search.displayRecord&uid=2000-07430-014

- Information framed as expert opinion may be more persuasive (but don’t abuse this bias).

Conclusion

A lot of people in the tech industry talk about “changing the world” and “making people’s lives better.” But bad design is excluding whole sections of the population from the benefits of technology. If you’re a designer, you can help change that. By following some simple principles, you can create more inclusive products that work better for everyone, especially the people who need them the most. ❧

Payment for this article was donated to Alzheimer’s Australia[173].

---

[173] https://fightdementia.org.au

About The Authors

Cathy O'Connor

Cathy O'Connor has worked as a senior web application designer for fifteen years in large and small companies. She relishes the challenge of incorporating complex and sometimes conflicting requirements such as compliance, security, globalization, and accessibility to distill and deliver useful, appealing, end-to-end experiences for customers. For the past six years she has been working with product teams to make sure PayPal products can be used by as many people as possible as an accessibility subject matter expert and now program manager. She enjoys coming up with new techniques and strategies to keep accessibility top of mind as product teams rapidly deliver new products in a constantly changing environment. She has spoken on some of these strategies at the California State University Northridge International Technology and persons with disabilities Conferences (CSUN): Web Accessibility Training for Product teams, with Jared Smith, WebAIM, and Color Contrast Tips and Tools for Designers[174] at CSUN 2014. Outside of work, Cathy is a part-time Zumba Fitness instructor, and enjoys designing in the physical world: crafting handmade toys, decor, and accessories for her family. Twitter: @cagocon[175].

---

[174] http://www.csun.edu/cod/conference/2014/sessions/index.php/public/presentations/view/280

[175] http://www.twitter.com/cagocon

Chaals McCathie Nevile

Chaals McCathie Nevile has over 30 years of professional experience with hypertext systems, for the last two decades focused on the web. He currently works in the CTO group of Russian internet giant Yandex, primarily focused on web standards and accessibility. Chaals was previously Head of Standards and a member of the board of directors at Opera, after working at the W3C specialising in the Semantic Web and accessibility. Chaals is still closely involved with the W3C, as co-chair of the Web Apps working group, and HTML Accessibility task force, and a long-standing member of the Advisory Board. He is interested in many different things, but current areas of focus there include SVG accessibility, the interaction model of the web, and Web Components. He also works on schema.org as part of his role at Yandex. Twitter: @chaals[176].

Henny Swan

In her capacity as User Experience and Design Lead at the Paciello Group (TPG) Henny Swan looks at ways of integrating the principles of inclusive design early on in projects. She joined TPG in 2014 having previously worked at BBC, Opera Software and Royal National Institute of Blind People. Henny has a particular focus on mobile and multimedia bringing her experience to bear on products delivered on multiple platforms. Previous work has in-

---

[176] https://twitter.com/chaals

cluded BBC iPlayer (web, iOS and Android), BBC Olympics, BBC Sport and the BBC cross platform Standard Media Player. Twitter: @iheni[177].

Léonie Watson

Léonie Watson is a Senior Accessibility Engineer with The Paciello Group (TPG) and owner of LJ Watson Consulting. Amongst other things she is a director of the British Computer Association of the Blind, a member of the W3C HTML and SVG working groups, and HTML Accessibility Task Force. In her spare time Léonie blogs on tink.uk[178], talks on the AccessTalk podcast[179], and loves cooking, dancing and drinking tequila (although not necessarily in that order). Twitter: @leoniewatson[180].

Marcy Sutton

Marcy Sutton is an international public speaker, Angular core team member and accessibility engineer at Adobe. She is a primary contributor to ngAria, Angular's accessibility module, as well as the author of an accessibility plug-in for Protractor, the end-to-end testing framework. Recently Marcy launched Accessibility Wins[181], a Tumblr highlighting successes in web accessibility. She loves rid-

---

[177] http://www.twitter.com/iheni

[178] http://accesstalk.co.uk/

[179] http://accesstalk.co.uk/

[180] https://twitter.com/leoniewatson

[181] http://a11ywins.tumblr.com/

ing bicycles and throwing the frisbee for her dog, Wally. Twitter: @marcysutton[182].

Ollie Campbell

Ollie is one of four co-founders at Navy Design[183], a design consultancy which specializes in digital products. He's interested in how good design can make people healthier, happier and safer. He writes about design on Medium[184] and occasionally contributes to publications such as Smashing Magazine[185] and UXMatters[186] and Creative Review[187]. His most recent speaking engagement was at the Medical Software Industry Association in Sydney. Ollie has a degree in computer science and is currently completing a postgraduate diploma in psychology. Twitter: @oliebol[188].

Scott O'Hara

Scott O'Hara is a UX designer and developer based out of Boston, Massachusetts. He loves pushing the limits of CSS, designing usable experiences for everyone, writing about what he knows and what he's learning. Website: scottohara.me[189]. Twitter: @scottohara[190].

---

[182] http://www.twitter.com/marcysutton
[183] http://www.navydesign.com.au
[184] https://medium.com/@oliebol
[185] http://www.smashingmagazine.com
[186] http://www.uxmatters.com
[187] http://www.creativereview.co.uk/
[188] http://www.twitter.com/oliebol
[189] http://www.scottohara.me

TJ VanToll

TJ VanToll is a senior developer advocate for Telerik, a jQuery team member[191], and the author of *jQuery UI in Action*[192]. He has over a decade of web development experience — specializing in performance and the mobile web. TJ speaks about his research and experiences at conferences around the world, and has written for publications such as Smashing Magazine, HTML5 Rocks, and MSDN Magazine. TJ is @tjvantoll[193] on Twitter and tjvantoll[194] on GitHub.

---

[190] http://www.twitter.com/scottohara
[191] https://jquery.org/team/
[192] http://tjvantoll.com/jquery-ui-in-action.html
[193] http://www.twitter.com/tjvantoll
[194] https://github.com/tjvantoll/

About Smashing Magazine

Smashing Magazine[195] is an online magazine dedicated to Web designers and developers worldwide. Its rigorous quality control and thorough editorial work has gathered a devoted community exceeding half a million subscribers, followers and fans. Each and every published article is carefully prepared, edited, reviewed and curated according to the high quality standards set in Smashing Magazine's own publishing policy[196].

Smashing Magazine publishes articles on a daily basis with topics ranging from business, visual design, typography, front-end as well as back-end development, all the way to usability and user experience design. The magazine is — and always has been — a professional and independent online publication neither controlled nor influenced by any third parties, delivering content in the best interest of its readers. These guidelines are continually revised and updated to assure that the quality of the published content is never compromised. Since its emergence back in 2006 Smashing Magazine has proven to be a trustworthy online source.

---

195. http://www.smashingmagazine.com
196. http://www.smashingmagazine.com/publishing-policy/